**Rapport du projet de système**

Mise en place d’un système de gestion de fichiers basé sur linux

**Réalisé par :**

Iheb BESBES : IATIC3

Iyed CHEBERLI : IATIC3

sami FAKHFAKH : IATIC3

Seifeddine BEN JEMAA :IATIC3

Med Ilyes EL AJROUD : IATIC3

Med firas ESSOURI : IATIC3

JOOBER Cyrine: IATIC3

**Encadrement :**

Mme Dhekra ABOUDA

Mr Thierry GARCIA

Sommaire

[Table des figures 3](#_Toc516689773)

[Introduction 4](#_Toc516689774)

[1- Conception 5](#_Toc516689775)

[1.1- BESOIN UTILISATEUR: 5](#_Toc516689776)

[1.2- CAHIER DE CHARGE 5](#_Toc516689777)

[1.3- REPARTITION DES TACHES 6](#_Toc516689778)

[2. Réalisation 6](#_Toc516689779)

[2.1- STRATEGIE D’IMPLEMENTATION : 6](#_Toc516689780)

[2.2- STRUCTURE DES BLOCS 8](#_Toc516689781)

[2.2-1. STRUCTURE d’un super-block 8](#_Toc516689782)

[2.2-2. STRUCTURE d’un fichier 9](#_Toc516689783)

[2.2-3. STRUCTURE d’un répertoire 10](#_Toc516689784)

[2.3- ORGANISATION DU DISQUE VIRTUEL : 11](#_Toc516689785)

[2.3-1. Organisation détaillé des blocs 12](#_Toc516689786)

[2.3-2. Utilité d’un MAP de table 12](#_Toc516689788)

[2.3-3. Variables globales 13](#_Toc516689789)

[2.4- FONCTIONS DE MANIPULATION 13](#_Toc516689790)

[2.4-1. Fonctions prédéfinis 13](#_Toc516689791)

[2.4-2. Fonctions génériques 14](#_Toc516689792)

[2.4-3. Fonction de formatage de disque 15](#_Toc516689793)

[2.5- LES PRIMITIVES 16](#_Toc516689794)

[2.6- LES COMMANDES DU SHELL 19](#_Toc516689795)

[2.6-1. Functions interne du shell 20](#_Toc516689796)

[2.6-2. Commandes utilisateur 20](#_Toc516689797)

[3- Conclusion 21](#_Toc516689798)

# Table des figures

[Figure 1 : Représentation simplifié du disque dur virtuel 7](file:///C:\Users\ASUS\Downloads\Proposition-de-projet-1-1.docx#_Toc516688254)

[Figure 2 : Exemple d'utilisation d'un super bloc pour la création d'un nouveau fichier 8](#_Toc516688255)

[Figure 3 :Accès à un fichier à partir de son nom 10](#_Toc516688256)

[Figure 4 : Accès d’un dossier et ces sous-dossiers à partir de son nom 11](#_Toc516688257)

[Figure 5 : l’organisation détaillé des blocs 12](file:///C:\Users\ASUS\Downloads\Proposition-de-projet-1-1.docx#_Toc516688258)

[Figure 6 : Relation entre MAP\_INODE et INODE\_TABLE 15](#_Toc516688259)

[Figure 7 : Processus de lecture d'un fichier via my\_Read 17](#_Toc516688260)

[Figure 8 : Création d'un dossier ou fichier via MyCreat 18](#_Toc516688261)

Introduction

“UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.”

Dennis Ritchie

UNIX, un système d’exploitation décrit “simple” par ses concepteurs, mais qui a laissé son empreinte sur pratiquement tous les aspects de nos systèmes d’exploitations modernes, nous étudiant parmi ces aspects les systèmes de gestions de fichiers sous linux, dans ce cadre un projet d'implémentation SGF nous été proposé, l’objectif principal est de modéliser les notions d’inodes et de blocs

Nous allons décrire dans ce rapport notre approche envers de ce sujet on décrivant;

Les besoins de l’utilisateur.

Le cahier des charges

Notre stratégie d'implémentation et prévision de structures de données

Et enfin notre méthodologie de travail d'équipe envers ce projet

# Conception

## BESOIN UTILISATEUR:

Nous prévoyons que l’utilisateur sera capable de d’effectuer des opérations de créations, modifications sur des fichiers stocké dans un environnement de notre propre conception à travers notre système.

Un utilisateur de notre système doit être capable de gérer les fichiers présent dans cet environnement en modifiant leurs emplacements, noms, droits d'accès ainsi qu’un contrôle de cet environnement commençant par la gestion des espaces libres, la modification de l'arborescence de répertoires, jusqu'à la réalisations des opérations de recherches et modifications complexes.

Notre utilisateur doit pouvoir interagir avec ce système avec l’invite commande élaboré suivant les opérations décrites ci-dessus.

## CAHIER DE CHARGE

**Objectif Principal :** Création d’un émulateur de système de gestion de fichier inspiré du SGF présent dans LINUX.

Nous décrivons dans ce qui suit nos estimations (d’après le cahier des charges fournis par nos encadrants) de sous objectif à réaliser afin de réaliser l’objectif principal:

1. La création d’un fichier qui jouera le rôle d’un disque dur virtuel
2. La création d’un mini Shell qui sera l’interface avec laquelle l’utilisateur exploite le système de gestion de fichier envisagé, ceci dit ;
   1. Notre mini Shell doit être capable de détecter les commandes fournis par l’utilisateur
   2. La séparation des options et des paramètres
   3. La capacité de ce mini Shell doit distinguer les commandes correctes de commandes erronées
3. Mise en place d’une stratégie d’un système de gestion de fichier qui comprend :
   * + 1. La structuration d’un bloc de donnée et un inode
       2. La gestion de ces blocs et ses inodes dans un disque dur (création, modification, suppression)
4. Exploiter le langage C afin d'implémenter cette stratégie ;
   * + 1. Chercher les fonctions et méthodologies qui permettent cette implémentation
       2. Acquérir la capacité de diviser et exploiter le disque dur virtuel suivant nos besoins
       3. Utiliser ces acquis afin de définir des primitives qui interagissent directement avec le disque dur. Elaboration partielle de notre implémentation de l’SGF
       4. Tester statiquement ces primitives et finalisation de la création de notre SGF
5. Effectuer le lien entre les commandes utilisateur et les primitives créés

=> Encapsulation des fonctions inutiles pour l’utilisateur (Rendre le système “User friendly”), rendre ce système une **boîte noire.**

Stratégie d’implémentation

## REPARTITION DES TACHES

|  |  |
| --- | --- |
| Chargé de la tâche | Tâche |
| Iheb BESBES | -Recherche individuelle sur les notion d'implémentation d’un système de gestion de fichiers  -Mise en place de la stratégie du système de gestion de fichier (en trinôme)  -implémentation du sgf (primitives et fonctions) (en binôme) |
| Mohamed Ilyess EL AJROUD | -Recherche individuelle sur les notion d'implémentation d’un système de gestion de fichiers  -Mise en place de la stratégie du système de gestion de fichier (en trinôme)  -test des primitives et fonction implémentés(en binôme) |
| Mohamed Firas SOURI | -Recherche individuelle sur les notion d'implémentation d’un système de gestion de fichiers  -Mise en place de la stratégie du système de gestion de fichier (en trinôme)  -liaison entre les primitives et commandes utilisateurs(en binôme) |
| Iyed CHEBERLI | -Recherche individuelle sur les notion d'implémentation d’un système de gestion de fichiers  -La creation d’un mini shell (en binome)  -test des primitives et fonction implémentés(en binôme) |
| Sami FAKHFAKH | -Recherche individuelle sur les notion d'implémentation d’un système de gestion de fichiers  -La creation d’un mini shell (en binome)  -liaison entre les primitives et commandes utilisateurs(en binôme) |
| Seifeddin BENJEMAA | -Recherche individuelle sur les notion d'implémentation d’un système de gestion de fichiers  -Création du disque dur virtuel  -implémentation du sgf (primitives et fonctions) (en binôme) |

# Réalisation

## STRATEGIE D’IMPLEMENTATION :

Notre stratégie d'implémentation et prévision de structures de données:

Nous avons choisi une stratégie d'implémentation de système de gestion de fichiers très proche de celle de LINUX mais plus simplifié qui s'oriente principalement autour du cahier des charges élaboré, notre disque dur sera divisé comme suit :

-Les blocs d’inodes : Ces blocs stockent les informations vitaux d’un fichier ou répertoire .On peut en citer le type du fichier, les droits d’accès, ....

Ces inodes contiendront aussi une table de pointeurs qui pointent sur les blocs de données. On a choisis de créer un seul pointeur indirect pour simplifier l'implémentation de ces inodes

-Un super bloc : Ce bloc contiendra les informations sur la totalité du disque tel que le nombre total de blocs, le nombre de blocs vides le nombre d’inodes vides.

Remarque : On dit toujours UN super bloc parce qu'il n’existe qu’un seul qui caractérise le disque dur mais physiquement il est formé d’un ensemble de blocs.

-Des blocs de données : Ces blocs constituent chaque fichier de notre SGF.Un bloc est présenté sous forme d’un espace alloué sur disque dur qui permettra de stocker les données d’une partie ou d’une totalité d’un fichier .Les blocs de données sont alloués au fur et à mesure de l’extension du fichier.

Nous prévoyons aussi les constantes qui suivent :

* La taille d’un bloc qui sera 1024 octets
* Le nombre de blocs inodes qui sera 2500
* Le nombre restant des blocs qui sera 99900
* Ces nombres sont calculés d'après la taille de notre disque dur qui est 104 857 600 octets (environ 100 Mo)

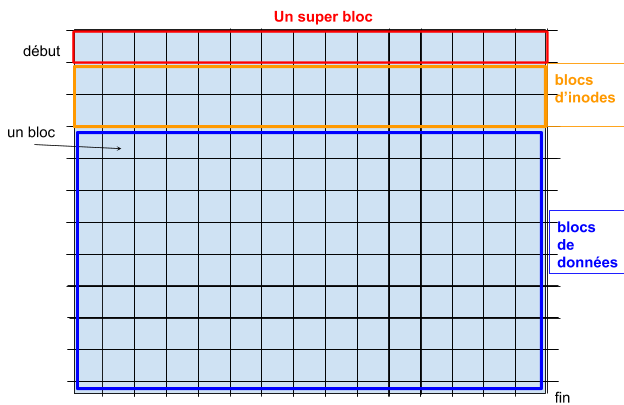


Figure 1 : Représentation simplifié du disque dur virtuel

## STRUCTURE DES BLOCS

## STRUCTURE d’un super-block

struct super\_block

{

   int magic\_number;

   int block\_num;

   int inode\_num;

   int free\_blk;

   int free\_inode;

};

Un super bloc contient les informations généraux du disque tel que  l’espace disponible allouable ....ces informations sont stockées dans les variables internes du super bloc :

**magic\_number :** un entier souvent trouvé dans le début ou près du début d’un fichier, il indique le format du fichier, il est appelé aussi “file signature” , nous utiliserons ce numéro pour caractériser notre disque dur et vérifier si un disque a été formaté selon notre stratégie ou pas.

* **block\_num:** un entier qui représente le nombre de blocs totales du disque dur
* **inode\_num:** représenté par un entier, il indique le nombre total d’inodes dans le disque
* **free\_blk:** un entier qui représente le nombre de blocs libres dans notre disque dur
* **free\_inodes:** le nombre d’inodes libre dans le système , type; entier.

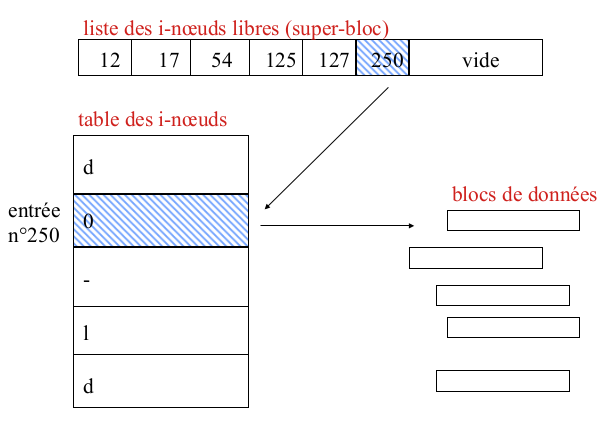


Figure 2 : Exemple d'utilisation d'un super bloc pour la création d'un nouveau fichier

## STRUCTURE d’un fichier

**Dans notre système, un fichier est constitué d’un inode qui contient toute informations nécessaire à la manipulation de ce fichier. Cet inode pointera sur un ou plusieurs blocs de données dans le disque dur.**

**Structure d’un bloc d’inode:**

struct inode {

   int type;

   int num;

   int size;

   int uid;

   int gid;

   char mode[11];

   char name[MAX\_FILE\_NAME\_LENGTH];

   int blocks[10];

   int ind\_blocks[30];

   char unused [15] ;};

Cette structure représente les champs à implémenter dans un bloc d’inode, un inode est caractérisé par les champs qui suivent:

* **Type** : Un entier qui peut prendre 1 pour dire que c’est un inode de fichier ou l’objet pointé par cet inode est un fichier et 2 pour dire que c’est un inode de répertoire
* **Num** : est le numéro de l’inode dans un disque dur, c’est un numéro unique qui caractérise l’inode en utilisant ce numéro nous pouvons trouver l’inode.
* Size : Un entier qui décrit la taille du fichier
* **Uid et gid** : l’id de l’utilisateur et du groupe (on a introduit ces valeur pour faciliter l'implémentation de la notion d’utilisateur dans notre projet si possible)
* **Mode [11]:** Une chaîne de caractère qui contiendra les droit d'accès au fichier de l’inode.
* **Name:** Une chaîne de caractère qui contiendra le nom du fichier, normalement le nom du fichier n’est pas stocké dans l’inode, on a choisi de le mettre dans l’inode afin de simplifier l'implémentation.
* **Blocks [10]:** Un tableau d’entier qui contiendra les pointeur vers les blocs de données (ou sera stocké le contenu de notre fichier)
* **Ind\_blocks[30]:** Un tableau de pointeurs indirectes simple ( rajout de double et triple direction une fois le système est fonctionnel si possible ).
* **Unused[15]:** un tableau non utilisé , nous voulons que la taille d’un inode soit égale à 256 octets

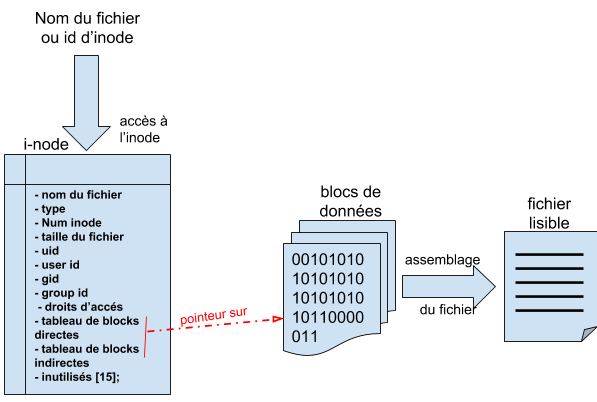


Figure 3 :Accès à un fichier à partir de son nom

**Structure d’un bloc de données:** Ce bloc n’a pas de structure spécifique vu qu’il sera en binaire. C’est le niveau le plus dans notre système de fichiers . **La taille d’un bloc sera  de 1024 octets.**

## STRUCTURE d’un répertoire

**Un répertoire est un ensemble de fichiers ou sous répertoires groupés en un seul endroit**

**Cette définition est représenté par la structure suivante :**

struct dir

{

   struct dirEntry dentry[32];

};

Un répertoire est constitué de 32 DIREntry qui peuvent être soit des fichiers soit des sous-répertoires .

**Structure d’un dirEntry :**

struct dirEntry

{

   int inode;

   int type;

   int length;

   char name[20];

};

Cette structure permet de représenter le contenu d’un dossier dans notre SGF

dirEntry peut etre associer à un inode d’un fichier ou un inode d’un sous-dossier .

Cette structure va nous faciliter ultérieurement le déplacement entre dossiers par la création des liaisons entre les différents fichiers et la génération  de liens symboliques .

Cette structure contiendra;

* **Name :** nom du fichier ou du dossier
* **Inode :**le numéro de l’inode (entier),
* **Type**: le type du fichier ( entier ) (1 pour fichier ou 2 pour dossier)
* **Length**: qui représente la longueur du  nom du fichier (entier) .
* **Name:** nom du fichier (chaîne de caractère).

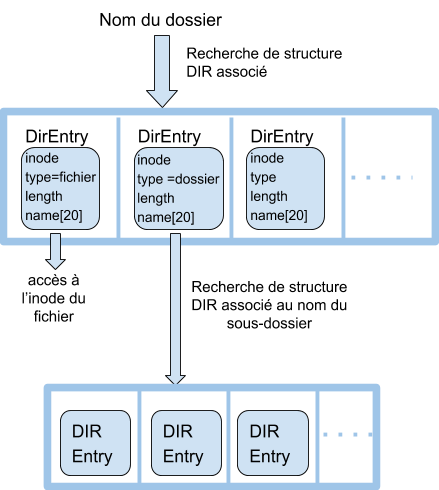


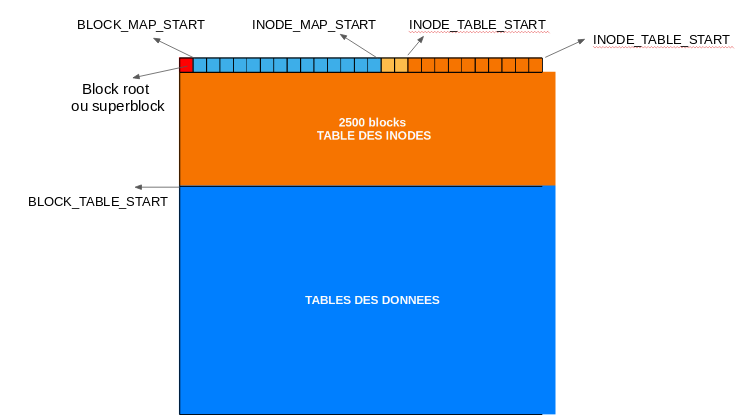
Figure 4 : Accès d’un dossier et ces sous-dossiers à partir de son nom

## ORGANISATION DU DISQUE VIRTUEL :

Dans ce paragraphe nous allons détailler de manière plus technique l’organisation du fichiers ainsi que les variables globales utilisées tout au long de notre système .

### Organisation détaillé des blocs

L’illustration ci-dessous résume l’organisation des blocs dans le disque ainsi que l’emplacement de chaque variable définis



## 

Figure : l’organisation détaillé des blocs

-**Super\_Block ou block ROOT :** c’est le bloc d’index 0 . C’est ce bloc qui va être réinitialisé avec la fonction format

-**BLOCK\_MAP** :Composé de 16 blocs,  La carte des blocs de données nous permet de trouver plus facilement le premier élément vide dans la table de données .Elle nous permet aussi d’estimer l’espace restant dans le fichier. L’utilisation de cette carte sera principalement dans la fonction get\_block() quand détailler dans le paragraphe suivant .

-**INODE\_MAP**: Composé de 2 blocs, La carte des inodes a la même utilité que le BLOCK\_MAP mais dédier  pour les inodes

**-INODE\_TABLE:** c’est la table des inodes ou la tables des pointeurs qui vont pointer sur les blocs de données .La table commence du 16 ème bloc jusqu’au 2516 me blocs .

-**BLOCK\_TABLE:** C’est la table des données .Tous les fichiers ,dossiers seront stockés dans cette espace du disque .

### Utilité d’un MAP de table

Nous avons choisi  d’implémenter deux MAP afin de faciliter la manipulation et la gestion des tables de données et d’inodes .Ces Maps seront d’une utilité majeure lors de la création et la suppression d’un bloc dans une table .Prenons l’exemple de suppression de fichier : Sans MAP, Le système est amené a supprimé l’inode , détruire les relations avec les blocs de données , réinitialiser l’arborescence …Avec un MAP il suffit de mettre l’octet concerné a zéro pour qu’à la prochaine insertion dans la table d’inodes , l’inode supposé supprimé sera automatiquement écrasé sans avoir besoin de manipulation nécessaire .

### Variables globales

Afin de faciliter la compréhension du code source, nous avons choisi de définir des variables globales qu’on va utiliser dans notre programme .

**MAX\_FILE\_NUM:**C’est le nombre de maximum d’inodes .Théoriquement , on aura des numéros d’inodes allant de 0 à MAX\_FILE\_NUM;

**MAX\_OPEN\_FILES:** Le nombre de fichiers maximums quand peut ouvrir simultanément .Cette variable limite la surcharge du système par les descripteurs de fichiers ouverts.

**MAX\_FILE\_NAME\_LENGTH:** Chemin le plus possible dans le système de fichiers

**BLOCKSIZE:** C’est la taille d’un seul bloc qui est 1024 octets(bytes en anglais) soit 8192 bits

**MAX\_LEVEL** : Le nombre maximum de sous dossiers possibles dans notre système de fichiers

**FILE\_TYPE et DIR\_TYPE:** Afin de faciliter la compréhension du code , nous avons défini le type fichier en tant que 1 et les dossiers en tant que 2

**char cwd\_name[100] et int cwd\_inode :**La première variable est le chemin du fichier ou du dossier entrée par l’utilisateur et la deuxième variable désigne l’inode du répertoire courant .ces deux variables globaux seront d’une grande utilité lors de la recherche et l'exécution des commandes dans le Shell.

## FONCTIONS DE MANIPULATION

Les fonctions de manipulation sont les piliers de notre système de fichiers. Ils permettent principalement la manipulation (lecture, écriture, modification ,suppression) de tous les blocs qui constituent le disque dur .

### Fonctions prédéfinis

Ces fonctions sont prédéfinis en c et on choisit de les citer puisqu’on les a utilisé de façon récurrente dans nos fonctions basiques .

**lseek() :** cette fonction permet de positionner un curseur offset sur un emplacement précis à l’intérieur du fichier .Cette dernière nous a été très utile lors de la navigation, la lecture, l'écriture et la modification de tranches de bits dans le fichier .

**fsync(descripteur) :**Cette fonction permet de transférer tous les morceaux chargés en mémoire vers le fichier dont son descripteur a été passée en paramètre . Ceci va assurer que les données ont été bien écrit physiquement sur disque .

### Fonctions génériques

Ces fonctions sont dite générique car ils sont valable que pour un seul bloc ou fichiers . Ces fonctions vont être utilisés de manière répétitive lors des traitements important comme la copie d’un fichier ou sa suppression .

* **int read\_block(int number, void \* block )**

Void \* permet de passer plusieurs types de structures différentes à cette fonction en entrée .ceci lui permettra de prendre n’importe quel type de bloc écrit dans le disque dur virtuel et d’en extraire les données stockés qui peuvent être des inodes , un super bloc ou des blocs de données .Cette extraction se traduit par le positionnement d’un curseur ou offset dans le fichier

* **int write\_block(int number, void \* block)**

Cette fonction permet d'insérer un bloc donné dans notre disque dur elle prend en paramètres un entier qui désigne le numéro de ce blocs dans la carte des blocs du disque et un void \* qui désigne le bloc à écrire l’entier renvoyé permet de déterminer si la fonction a été exécuté avec succès ou pas

* **int write\_inode(int number , void \* inode)**

Similaire aux fonctions précédents cette fonctions permet d’ajouter un inode suivant un numéro donné dans la carte d’inode du disque dur l’entier renvoyé permet aussi de voir si la fonction a été exécuté avec succès ou pas

* **int read\_inode(int number , void \* inode)**

Cette fonction permet de récupérer un inode dans void inode suivant le numéro d’inode donné, le type de retour permet de savoir si la fonction a été exécuté correctement ou pas

* **Int get\_inode():**

Cette fonction nous permettra de calculer le numéro du premier bloc vide dans la table des inodes sans avoir besoin de parcourir toute la table. La fonction va parcourir tout la INODE\_MAP pour chercher le premier octet de valeur 0 .Ce numéro d’octet va correspondre à un inode non utilisé dans la table d’inodes .Elle va par la suite mettre la valeur 0xFF dans l’octet de la Map et retourner le numéro de l’inode dans la table .

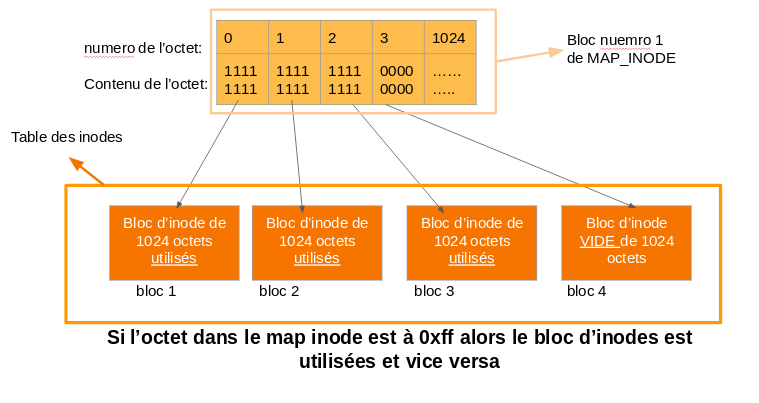


Figure 6 : Relation entre MAP\_INODE et INODE\_TABLE

* **Int get\_block():**

Même principe que get\_inode() mais appliqué à la table des données .La sortie sera le numéro du bloc vide dans la table de données

### Fonction de formatage de disque

La fonction **format\_disk ()** est un peu spéciale vu qu’elle sera responsable du formatage du disque virtuel suivant nos besoins mentionnés au pare avant dans ce rapport .Cette fonction va construire les piliers de notre SGF en passant par différentes parties :

-Allocation des espaces ROOT composés d’un Super\_Block , un inode ROOT et un bloc de données ROOT. Ces allocations seront principalement au début de chaque compartiment .Par exemple l’inode ROOT sera le premier bloc de la table d’inodes du disque .

- Configuration du Super\_Block en lui attribuant le numéro magique qui caractérise le disque virtuel , le nombres total de blocs de données et d’inodes et l’espace libre disponible .

- Configuration de l’arborescence du système et la création du dossier ROOT ( appelé aussi la racine ‘/’ ) qui sera stocké dans le bloc de donnée ROOT. Cette configuration comporte aussi la mise à jour des INODE\_MAP et BLOCK\_MAP ,L’attribution de droits d’accès ,définition des utilisateurs ainsi que l'établissement de connexion entre l’inode ROOT et le block ROOT.

-initialisation des descripteurs du système

Cette fonction imbrique une fonction de test qui permet de tester si le disque a été déjà formaté ou non .Cela évitera tout écrasement des blocs ROOT déjà crée et s’échapper à une destruction totale du SGF.

## LES PRIMITIVES

Comme indiqué dans le cahier des charges , Notre SGF doit être complètement indépendant de tout autre système , ce qui veut dire qu’on doit définir nous même les fonctions de manipulation de fichiers comme Read, write open et close .Ces fonctions sont primordiales pour manipuler ultérieurement notre disque virtuel et faire fonctionner les commandes du mini-Shell.

* **struct inode\* find (const char\* path)** :

Cette fonction sert à trouver un inode à travers le chemin du fichier ou répertoire donné en paramètre. La fonction teste tout d'abord la profondeur du chemin (path) qui ne doit pas dépasser MAX\_LEVEL .

Cette variable caractérise le nombre de niveau de profondeur pour les répertoires il ne faut pas dépasser. Vient ensuite la décomposition du chemin en sous parties séparé par / et qui vont être traité un à un .La fonction comporte 3 principaux traitements :

-La détection du début de chemin et l’identification de son type (absolu ou directe )

-La détection du dernier élément du chemin :On différencie entre la recherche d’un fichier ou d’un répertoire par le dernier caractère du chemin « / » .Si on recourt à un dossier , on doit obligatoirement passer par ces sous-dossiers s’ils sont mis dans le chemin .

-La recherche des inodes liés parties du chemins. Théoriquement , La recherche d’un inode revient à trouver dans la table des inodes le nom qui correspond à un segment du chemin indiqué dans le path.

cette recherche inclut aussi la recherche approfondi dans les Direntry de chaque dossier afin d'accéder au niveau inférieur de profondeur ( des dossiers de niveau 1 vers leurs sous dossiers qui sont considérés de niveau 2)

-l'accès aux blocs directes et indirectes des données via les inodes trouvés à l'étape précédente .Après un test de validité sur ces blocs , on peut affirmer ou nom si le fichier ou dossier existe ou non .

La fonction nous donne en sortie l’inode du dernier fichier ou dossier du chemin .

* **int my\_open(const char\* path) et void my\_close(const char\* path) :**

C’est deux fonctions sont responsables de l’ouverture et la fermeture du descripteur du fichier ou dossier spécifié dans le chemin path .

Dans notre système de gestion de fichier , un descripteur n’est autre qu’un tableau de MAX\_OPEN\_FILES entiers .les primitives des fichiers vont attribués une valeur spécifique selon l’état du fichier en cours .

Si un descripteur contient un numéro d’inode d’un fichier c’est que le fichier est ouvert et en cours d’utilisation .Par ailleurs, si on une valeur nulle ou négative alors le descripteur est soit fermé ou vide .Dans ce cas on peut l’utiliser pour ouvrir un fichier .

* **int my\_read (int fdd, void \* buf ,int nbtes):**

Cette fonction prend en entrée le descripteur de fichier à lire(équivalent à son numéro d’inode) , le buffer qui va contenir les données à lire et le nombres de blocs nbtes à lire .Si nbtes est inférieur à 10 alors on lira de la tables d’adresse de l’inode que les blocs directes .

Dans le cas contraire nous devons parcourir les blocs indirectes qui contiennent chacun 256 blocs directes .

On commence L’assemblage par la lecture du bloc avec read\_block() puis la copie de son contenu dans le buffer de lecture à travers la fonction memcpy().

La figure ci-dessous détaille le processus de lecture d’un fichier quelconque appartenant à notre SGF.

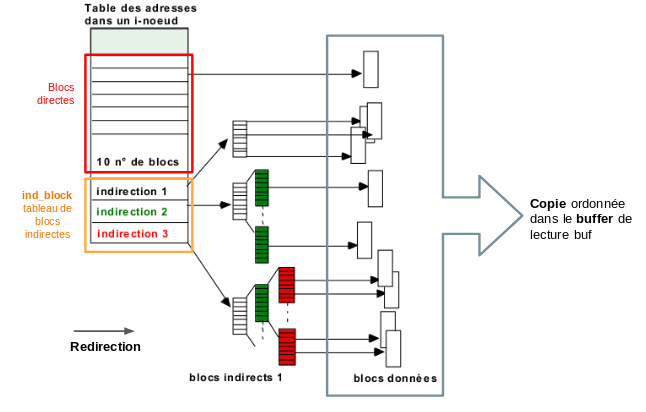
****

Figure 7 : Processus de lecture d'un fichier via my\_Read

* **int my\_write (int fdd, char \* buf ,int nbtes)**

L'écriture d’un bloc est l'opération inverse de la lecture : On commence par préparer le bloc à écrire en allouant un bloc dans la table de données et faire une copie dans un buffer d'écriture .

On cherche ensuite des emplacement vides dans la tables d’adresse de l’inode en privilégiant les blocs directes sur une redirection dans un bloc indirecte .

L'écriture physique dans le fichier se fait à travers la fonction write\_block() avec le buffer d'écriture en paramètre

* **int mycreat(const char\* path)**

mycreat permet de créer un fichier ou un dossier à partir d’un path donné . Cette fonction commence par découper le chemin en différents sous chaînes ( le séparateur = /) pour être traiter un à un . Ce découpage est suivie d’une batterie de tests qui vont nous permettre de vérifier si le chemin est valide ou non et respecte la forme générale d’un chemin .cette forme se résume en quelques règles .On peut en citer :

1. Chaque chemin doit commencer par le répertoire ROOT ou la racine /
2. Chaque niveau de profondeur doit être séparé par un /
3. Un fichier ne peut pas contenir un dossier et le contraire est vrai
4. Le dernier nom du fichier ou du dossier ne doit pas être déjà existant dans l’arborescence du système

Suite à ces tests , on passe à la phase de création physique du fichier .la fonction commence par allouer un inode ,Le configurer avec les informations adéquate puis faire la liaison entre l’inode et l’adresse de début d’insertion dans le bloc de données .La primitive a eu recours à get\_inode et get \_block afin d’identifier les premiers éléments vides dans la tables de données et de blocs .

Si la fin du path se termine par / ;C’est un dossier qu’on doit créer .Dans ce cas , on doit parcourir les DIRentry du dossier d’avant vide et injecter le nouveau inode dedans.

L’image ci-dessous explique les étapes essentiels de la création d’un fichier ou dossier .

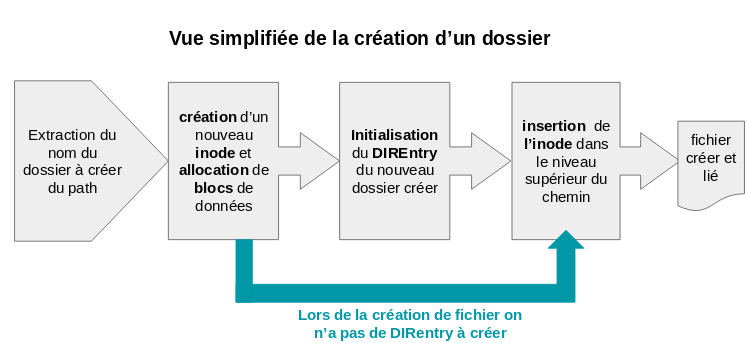


Figure 8 : Création d'un dossier ou fichier via MyCreat

* **void my\_mkdir(const char\* path):**

En bref, elle permet de créer un répertoire en créant tout d’abord séparé le nom du fichier ou du répertoire à créer et le stocker.

Elle a un fonctionnement similaire à celui de my\_create :

On Récupère l’inode du répertoire courant (dir\_name) et vérification de son existence.

Si dir\_name existe on doit s’assurer qu’il est bel et bien un répertoire et non un fichier et qu’il ne contient pas un même dossier que celui à créer en faisant un parcours sur les DIRentry (0 à 32).

Après ces tests notre travail consiste à trouver un DIRentry vide dans lequel on stock un nouveau inode obtenu par la fonction get\_inode() qui doit retourner une valeur autre que -1

Sinon on informe l’utilisateur que l’espace est insuffisant pour créer un répertoire.

On initialise le type avec la valeur DIR\_TYPE dans la DIRentry ainsi que tous les champs (num,name,type,size,mode…).

En fin on Sauvegarde de l’inode avec write\_inode()

* **void my\_rmdir (const char\* path)**

My\_rmdir permet la suppression du dossier ou du fichier pointé dans le path .La suppression se fait en 3 parties :

* La mise a zero des octets correspondant au fichier dans le MAP d’inode et de données
* Suppression des blocs de données dans la table des données
* Suppression de la liaison dans le dentry du niveau supérieur
* Si c’est un dossier , on supprime tout le tableau dentry et détruire tout les fichiers dedans
* **void seek\_block (int\* i, int\* j,int nbck) et void seek\_Inode (int\* i, int\* j,int nbck)**

Ces deux fonctions permettent de calculer les coordonnées ou le correspondant du numero du bloc dans le MAP

## LES COMMANDES DU SHELL

Ces Commandes et fonctions sont la base de notre Shell .Les fonctions internes assurent le bon fonctionnement du Shell et les commandes Shell sont eux qui seront exposé à l’utilisateur

## Functions interne du shell

* **char\* supprimer\_espace (char\* cmd)**

Cette fonction permet de formater la chaîne de caractère entrée selon des règles bien précise :

* La chaîne ne doit ni commencer ni finir par un espace
* Le caractère espace doit être tapé qu’une seul fois à chaque utilisation
* Lors d’une redirection avec le caractère >, on doit avoir un seul espace avant la redirection et aucun après la redirection.
* **int verif\_cmd (char par [] [MAX], int indice) :**

Vérifie si la commande à l’indice est une commande valide ou non

* **int sep\_opt\_arg (char par[][MAX],char opt[][MAX],char argu[][MAX],int nbparam)**

Cette fonction permet de séparer les arguments et les options ainsi que compter le nombres de paramètre passer à la commande et les stocker ensuite dans deux tableaux différents opt et argu

* **int verif\_redirection (char par[][MAX],int nbparam,char red[])**

Cette fonction permet de vérifier l’emplacement du caractère de redirection et la correction du nombre de paramètre en décrémentant le paramètre passé en redirection

void moteur\_cmd (char par[],char argu[][MAX],char opt[][MAX],char red[],int i,int verifcm)

Cette fonction permet de valider ou non la syntaxe d’une commande après le remplissage de ces entrées.

* **void analyse\_cmd (char\* cmd)**

Comme son nom l’indique, analyse cmd est la fonction principale qui va regrouper tous les sous fonctions vue précédemment afin de générer les tableaux d’arguments argu[ ] [ ] , d’options opt[ ] [ ] et de commandes par[ ][ ] .Cette fonction va aussi traiter la redirection où elle va verifier et stocker le fichier lequel on va faire la redirection. Enfin c’est elle qui affichera l'entête de notre Shell [ISTY]:~$

## Commandes utilisateur

|  |  |
| --- | --- |
| void **my\_touch**(char par [],char argu[][MAX],char opt[][MAX],int i,int nbopt); | Creation d’un fichier vide |
| void **my\_cd** (char par [],char argu[][MAX],char opt[][MAX],int i,int nbopt); | Change de repertoire |
| void **my\_cp** (char par [],char argu[][MAX],char opt[][MAX],int i,int nbopt); | Copie d’un fichier ou d’un dossier .Permet aussi de renommer un fichier |
| void **my\_rmdir** (char par [],char argu[][MAX],char opt[][MAX],int i,int nbopt); | Efface un répertoire |
| void **my\_rm** (char par [],char argu[][MAX],char opt[][MAX],int i,int nbopt); | Plus générique que my\_rmdir, elle peut effacer toute sorte de fichier |
| void **my\_mv** (char par [],char argu[][MAX],char opt[][MAX],int i,int nbopt); | Deplace un fichier ou dossier |
| void **my\_cat** (char par [],char argu[][MAX],char opt[][MAX],int i,int nbopt,char red[]); | Concatène deux fichiers un au-dessous de l’autre |
| void **my\_ln** (char par [],char argu[][MAX],char opt[][MAX],int i,int nbopt); |  |
| void **my\_echo** (char par [],char argu[][MAX],char opt[][MAX],int i,int nbopt,char red[]); | Affiche dans la console un message tapé par l’utilisateur |
| void **my\_ls** (char par [],char argu[][MAX],char opt[][MAX],int i,int nbopt); | Affiche en détail les fichiers et dossier contenu dans le chemin passé en paramètre |
| void **my\_mkdir** (char par [],char argu[][MAX],char opt[][MAX],int i,int nbopt); | Creer un dossier |
| void **my\_df** (char par [],char argu[][MAX],char opt[][MAX],int i,int nbopt); | Fournit la quantité d'espace occupé des systèmes de fichiers. |

# Conclusion

Tout au long de ce projet nous avons approfondi nos connaissances en langage C et en programmation système.

En effet, la mise en œuvre d'un système de gestion de fichier était une tache assez délicate. Nous avons essayé

De répartir les différentes tâches entre les membres du groupe et de collaborer ensemble dans le but d'avoir un

Système de gestion de fichier équivalent à celui d’UNIX.

Nous avons pu créer SGF ayant une structure plus au moins équivalente à celle de UNIX, on a pu trouver une façon de stocker

Et organiser les fichiers. Cela nous a permis de nos données et les partager entre plusieurs utilisateurs.

Cette organisation a été réalisée au sein d'un disque virtuel qu'on a créé en respectant cette hiérarchie.

Enfin, nous pouvons dire que notre répartition des taches a été laborieuse et ont pu réaliser notre projet et avoir une démo fonctionnel

Ayant les principales fonctionnalités demandées.