



# Projet TTFs

*Diallo Elhadj Amadou, Lefranc Joaquim,  
Skoda Jérôme, Vic Benjamin*



## Arborescence du dossier

---

- **\_INSTALL.sh** : Script d'installation des outils
- **\_REMOVE.sh** : Script de désinstallation des outils
- **bin/** : Contient les exécutables
- **header/** : Contient tout les .h du projet
- **src/** : Contient tout les .c du projet
- **Makefile/** : Différentes règles pour la gestion du projet
- **README** : Fichier lisez moi
- **Rapport/** : Rapport de projet format InDesign CS6 (.pdf disponible)



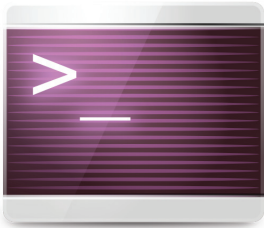
## Règles du Makefile

---

- **all** : Compile toutes les commandes et bibliothèques
- **clean** : Supprime les .o dans bin/
- **tfs\_create** : Compile la commande tfs\_create
- **tfs\_partition** : Compile la commande tfs\_partition
- **tfs\_analyze** : Compile la commande tfs\_analyze
- **tfs\_format** : Compile la commande tfs\_format
- **libll** : Compile la bibliothèque dynamique libll.so
- **libtfs** : Compile la bibliothèque dynamique libtfs.so
- **tar** : Exporte le projet en .tar
- **install** : Exécute le script d'installation
- **remove** : Exécute le script de désinstallation



**Attention les scripts d'installation et de désinstallation doivent être exécutés en mode sudo. (La règle du Makefile en tient compte)**



## Description détaillée des commandes



### Commande `tfs_create`

```
$ tfs_create -s 1000 MonDisk
```

Cette commande permet de créer un disque (fichier de x blocs de 1024 octets).

Concrètement, elle ajoute **.tfs** au nom donné au disque, elle teste si le fichier n'existe pas encore et le crée en insérant dans le premier **uint32\_t** le nombre de blocs en **hexadécimal** écrit en **little-indian**. (Fig. A)

Un **uint32** = **4 octets**

Un bloc = 256 nombres (1024/4) [**0 ; 255**]

Blocs = [**0 ; size-1**]

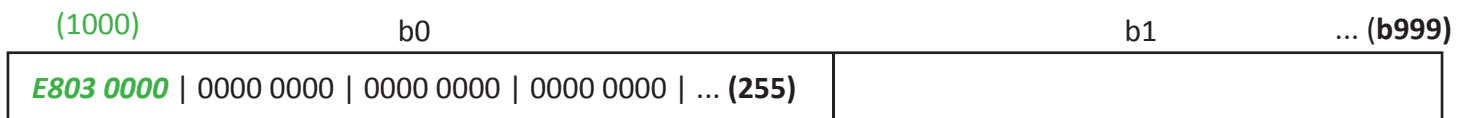


Figure A



### Commande `tfs_partition`

```
$ tfs_partition -p 200 -p 400 MonDisk
```

Cette commande partitionne le disque en ajoutant dans **b0[1]** le nombre de partitions, puis dans **b0[2]** la taille de la première partition, **b0[3]** celle de la deuxième etc. (Fig. B)

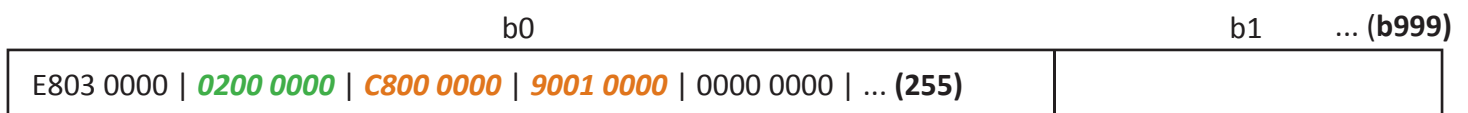


Figure B

(2) (200) (400)



## Commande tfs\_analyze

---

```
$ tfs_analyze MonDisk
```

*Cette commande fait une lecture du bloc 0 du disque et affiche les informations. (Fig. C)*

```
=====
# Disk : zDisk.tfs
-> Size : 1000 blocks | 1024ko
-> Partitions : 2
  - 0 = 200 blocks | 204ko
  - 1 = 400 blocks | 409ko
=====
```

*Figure C*



## Commande tfs\_format

---

```
$ tfs_format -p 0 -mf 200 MonDisk
```



# Description détaillée des fonctions de ll.h



## Gestion des disques

---

La gestion des disque ce fait par le biais d'une liste d'une taille maximum définie par **MAX\_OPEN\_DISK**. C'est un tableau de type **DISK**, qui est un type définissant un disque dynamique (utilisable par les fonctions). Elle symbolise en quelques sorte la grappe SATA dans un contexte physique. (Qui est de taille finie)

Tableau de **DISK** :

```
DISK _disks[MAX_OPEN_DISK] = {};
```

Type **disk\_id** :

```
typedef int disk_id;
```

Un simple **int** permettant d'accéder au disque voulue dans la liste des disques.

Type **DISK** :

```
typedef struct{  
    FILE *disk_descriptor;  
    uint32_t nb_blocks;  
    int flag;  
} DISK;
```

Une structure de **DISK** contient un pointeur sur le fichier correspondant au disque, son nombre de blocs et enfin un **flag** pouvant prendre comme valeur **\_MOUNTED** ou **\_UNMOUNTED**.

Démarrage d'un disque :

```
error start_disk(char *name, disk_id id)
```

Cette fonction fait quelques vérification (**id** existante , fichier existant, etc) puis insère un nouveau **DISK** dans le tableau **\_disks**. Le **\*disk\_descriptor** correspond au pointeur sur fichier **name**. Elle s'occupe également de remplir les autres champs comme il se doit, dont passer le disque en **\_MOUNTED**.

Arrêt d'un disque :

```
error stop_disk(disk_id id)
```

L'arrêt d'un disque consiste simplement à passer le flag du **DISK** correspondant à **\_UNMOUNTED**, de fermer le **\*disk\_descriptor** puis de le mettre à 0 par sécurité. Cela signifie que l'emplacement est de nouveau libre pour un autre disque.



## Lecture et écriture de blocs

Pour les fonctions suivantes nous avons également besoin d'un **type block** de façon à pouvoir gérer plus facilement la modification de ceux ci. C'est simplement un tableau de char de taille **BLCK\_SIZE**.

```
typedef unsigned char block[BLCK_SIZE];
```

Pour facilité le travail et pour une lisibilité plus aisée du code, nous avons besoin de quelques fonctions auxiliaires.

Passage d'un bloc en little-indian :

```
void blockToLtleIndian(block b)
```

Cette fonction passe tout les octets du bloc en question en **little-indian**, de ce fait nous pouvons ensuite facilement écrire tout le bloc sans se poser de questions. Nous pouvons également l'appliquée à un bloc déjà en little-indian ce qui à pour effet de le repasser en **big-indian**. Le but est de pouvoir écrire des nombres dans le bloc de façon classique, puis de le convertir juste avant l'écriture. De la même façon pour la lecture et l'utilisation du bloc lu.

Lecture d'un bloc physique :

```
error read_physical_block(disk_id id, block b, uint32_t num)
```

Après les vérifications d'usage (le disque est bien monté, l'id existe, etc) la fonction se déplace dans le **\*disk\_descriptor** grâce à **fseek()** avec la formule **(num \* BLCK\_SIZE)** de façon à atteindre le bloc en question. Puis à l'aide d'un **fread()** remplit le **block b** passé en paramètre après avoir converti celui-ci avec **blockToLtleIndian()**.

Ecriture d'un bloc physique :

```
error write_physical_block(disk_id id, block b, uint32_t num)
```

De la même façon que pour **read\_physical\_block()** avec un **fwrite()** cette fois-ci.



## Manipulation des blocs

---

La manipulation des «objets» de type *block* peut se faire grâce à différentes fonctions.

**Lecture d'un nombre en position *p* :**

```
int readBlockToInt(block b, int position)
```

Cette fonction permet de récupérer le nombre en position *p* du bloc en question. Ce nombre est converti de l'hexadécimal en *uint32\_t*.

**Ecriture d'un nombre en position *p* :**

```
error writeIntToBlock(block b, int position, uint32_t number)
```

De la même façon que pour la fonction précédente, avec cette fois l'écriture du nombre en question à la position *p*. Préalablement converti en hexadécimal.

**Effacement d'un bloc :**

```
error eraseBlock(block b, int debut, int fin)
```

Nous devons parfois pouvoir effacer simplement tout ou partie d'un bloc. Dans ce cas cette fonction est utile. Elle prend le premier nombre à effacer et le dernier exclus.



## Effacement d'un disque

---

Pour les commandes de partitionnement et de formatage, il est nécessaire d'effacer préalablement le contenu du disque. Pour ce faire nous avons une fonction à notre disposition.

**Effacement d'un disque :**

```
error eraseDisk(disk_id id, int block_debut, int block_fin)
```

Cette fonction permet d'effacer tout les blocs du disque (remise à 0 des nombres) à partir de **block\_debut** jusqu'à **block\_fin** exclus.



## Gestion de la table des fichiers

---