Projet TTFS

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

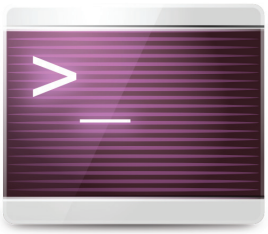
Arborescence du dossier

- ***_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 bibliothèqye dynamique libtfs.so*
- ***tar :*** *Exporte le projet en .tar*
- ***install :*** *Execute le script d'installation*
- ***remove :*** *Execute 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 **héxadécimal** écrit en **little-indian**. (Fig. A)*

*Un uint32 = **4 octets***
*Un bloc = 256 nombres (1024/4) **[0 ; 255]***
*Blocs = **[0 ; size-1]***

(1000) b0 b1 ... (**b999**)

E803 0000 | 0000 0000 | 0000 0000 | 0000 0000 | ... **(255)**	

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 premiere partition, **b0[3]** celle de la deuxième etc. (Fig. B)*

b0 b1 ... (**b999**)

E803 0000 | ***0200 0000*** | ***C800 0000*** | ***9001 0000*** | 0000 0000 | ... **(255)**	

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 32 MonDisk

Cette commande formate une partition donnée du disque en y ajoutant le bloc de description, les blocs de la table des fichiers et l'entrée de la racine. (Fig. D)

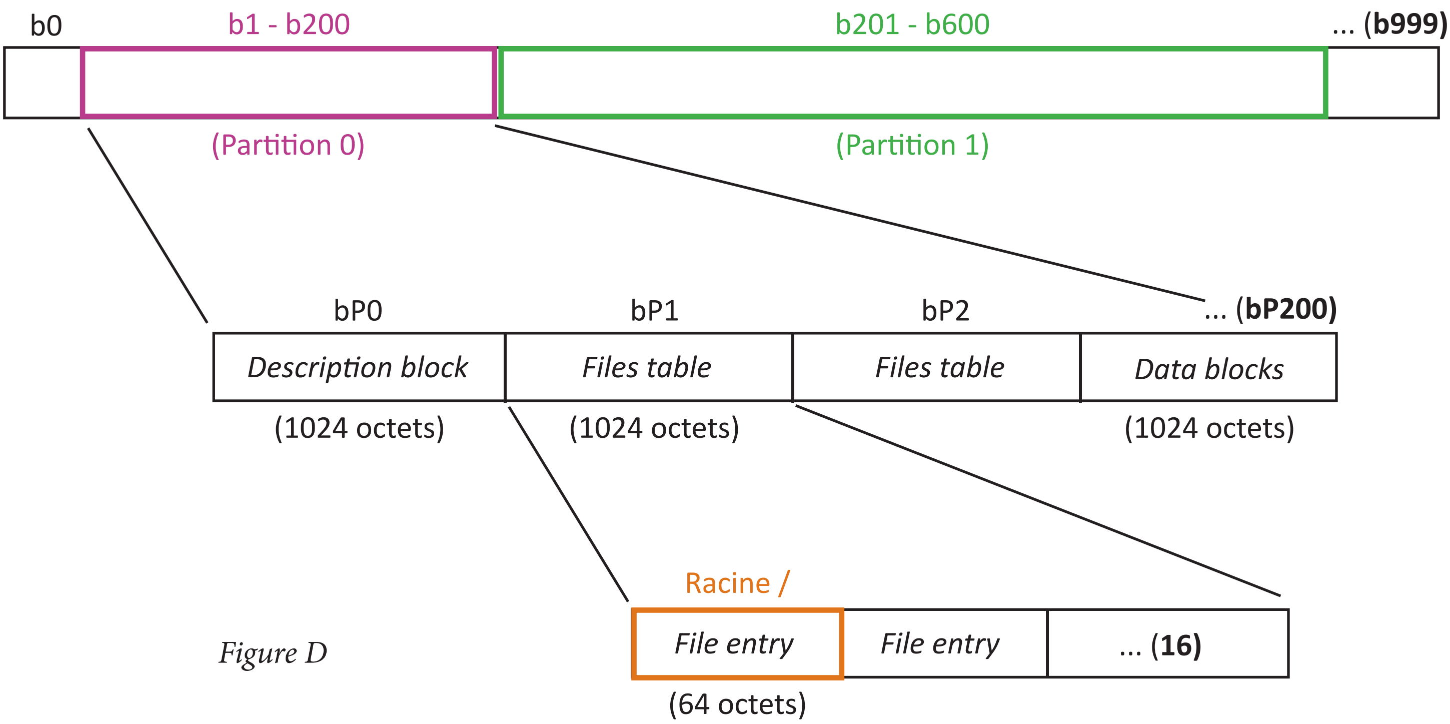


Figure D

Description détaillée des fonctions de ll.h

Les types

Type error :

```
typedef int error;
```

Un simple int correspondant au type de l'erreur.

Type block :

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

*C'est simplement un tableau de char de taille **BLCK_SIZE**.*

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**.*

Type DISK_INFO :

```
typedef struct{
    uint32_t size;
    uint32_t nb_partitions;
    uint32_t p_size[ (BLCK_SIZE) / 4) - 2 ];
} DISK_INFO;
```

*Une structure de **DISK_INFO** permet de contenir les informations du premier bloc du disque, sa taille, son nombre de partition et les tailles de celles-ci.*

Type PARTITION_INFO :

```
typedef struct{
    uint32_t TTTFS_MAGIC_NUMBER;
    uint32_t TTTFS_VOLUME_BLOCK_SIZE;
    uint32_t TTTFS_VOLUME_BLOCK_COUNT;
    uint32_t TTTFS_VOLUME_FREE_BLOCK_COUNT;
    uint32_t TTTFS_VOLUME_FIRST_FREE_BLOCK;
    uint32_t TTTFS_VOLUME_MAX_FILE_COUNT;
    uint32_t TTTFS_VOLUME_FREE_FILE_COUNT;
    uint32_t TTTFS_VOLUME_FIRST_FREE_FILE;
} PARTITION_INFO;
```

*Idem que pour **DISK_INFO** mais pour une partition, cela correspond au **Description_block** de la partition.*

Type FILE_ENTRY :

```
typedef struct{
    uint32_t tfs_size;
    uint32_t tfs_type;
    uint32_t tfs_subtype;
    uint32_t tfs_direct[10];
    uint32_t tfs_indirect1;
    uint32_t tfs_indirect2;
    uint32_t tfs_next_free;
} FILE_ENTRY;
```

Ce type correspond à l'architecture d'une entrée de fichier dans la table.

Type DIR_ENTRY :

```
typedef struct{
    uint32_t number;
    char name[28];
} DIR_ENTRY;
```

Ce type correspond à l'architecture d'une entrée de répertoire contenu dans un bloc de données .

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] = {};
```

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 du bloc d'information d'un disque :

```
error readDiskInfos(disk_id id, DISK_INFO *infDisk)
```

*Permet de récupérer dans la structure **DISK_INFO** passée en argument les informations du premier bloc du disque.*

Lecture du bloc d'information d'une partition :

```
error readPartitionInfos(disk_id id, PARTITION_INFO *infPartition,
                         int partition)
```

*Permet de récupérer dans la structure **PARTITION_INFO** passée en argument les informations du premier bloc de la partition voulue. (Si elle existe, sinon une erreur est levée)*

Ecriture du bloc d'information d'une partition :

```
error writePartitionInfos(disk_id id, PARTITION_INFO infPartition,
                          int partition)
```

*Permet d'écrire la structure **PARTITION_INFO** passée en argument sur le premier bloc de la partition voulue. (Si elle existe, sinon une erreur est levée)*

Récupération du numéro du premier bloc d'une partition :

```
error getFirstPartitionBlck(disk_id id, int partition, uint32_t *number)
```

Modifie le nombre passé en argument en lui donnant la valeur du premier bloc de la partition voulue.

Récupération de la taille d'une partition :

```
error getPartitionSize(disk_id id, int partition, uint32_t *number)
```

*Ecrit dans **number** , si la partition existe, sa taille en nombre de blocs.*

Récupération de la taille de la table :

```
error getFilesTableSize(disk_id id, int partition, uint32_t *number)
```

*Ecrit dans **number** , si la partition existe, la taille de sa table.*

Affichage sur stdout les informations de la partition :

```
void printInfoPartition(PARTITION_INFO infPartition)
```

Simple affichage pour tests et debug.

Lecture et écriture de blocs

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.*

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.*

Manipulation des blocs

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

Lecture d'un nombre en position p :

```
uint32_t 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.

Lecture d'une String en position p :

```
error readBlockToStr(block b, int position, char* str, int size)
```

De la même façon que pour les entiers, cette fonction récupère une string de taille voulue à partir de l'emplacement désiré.

Ecriture d'une String en position p :

```
error writeStrToBlock(block b, int position, char* str, int size)
```

Idem que la fonction précédente mais pour l'écriture.

Ecriture d'une entrée de répertoire :

```
error writeDirEntryToBlock(block b, int position, DIR_ENTRY dir_ent)
```

*Ecrit les informations de **DIR_ENTRY** dans le bloc à la position voulue.*

Lecture d'une entrée de répertoire :

```
error readBlockToDirEntry(block b, int position, DIR_ENTRY *dir_ent)
```

*Lit les informations de **DIR_ENTRY** dans le bloc à la position voulue.*

Affichage d'un bloc :

```
error printBlock(block b)
```

Affiche un bloc sur stdin, fonction utile pour les tests et le développement.

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.

Gestion de la table des fichiers

*La table des fichiers commence en position 1 de la partition, juste après le bloc de description. Sa taille est calculée par rapport au **MAX_FILE_COUNT** en sachant qu'un bloc de 1024 octets ne peux contenir que 16 entrées de 64 octets.*

Initialisation de la table des fichiers :

```
error initFilesTable(disk_id id, int partition)
```

Initialise une table de bonne dimension en fonction du bloc d'informations de la partition. Elle chaîne également les entrées entre-elles.

Ecriture d'une entrée de fichier dans la table :

```
error writeFileEntryToTable(disk_id id, int partition, FILE_ENTRY file_ent,
                            int file_pos)
```

Ecrit l'entrée à la position voulue, cette position ne peux excéder la taille de la table. Elle s'occupe de choisir le bloc nécessaire dans le cas d'une table sur plusieurs blocs.

Lecture d'une entrée de fichier dans la table :

```
error readFileEntryFromTable(disk_id id, int partition,
                             FILE_ENTRY *file_ent, int file_pos)
```

De la même façon que pour l'écriture.

Ajout d'une entrée de fichier dans la table :

```
error addFileEntryToTable(disk_id id, int partition, int file_size,
                          int file_type, int file_subtype)
```

Ajoute une entrée dans ta table en spécifiant les informations de cette entrée. Cette fonction tient à jour le chaînage des entrées et la description de la partition.

Suppression d'une entrée de fichier dans la table :

```
error removeFileEntryInTable(disk_id id, int partition, int file_pos)
```

Supprime une entrée dans ta table. Cette fonction tient à jour le chaînage des entrées et la description de la partition.

Affichage sur stdout les informations de l'entrée :

```
void printFileEntry(FILE_ENTRY file_ent)
```

Simple affichage pour tests et debug.

Gestion du chainage des blocs libres

Initialisation de la chaine des blocs libres :

```
error initFreeBlockChain(disk_id id, int partition)
```

Construit la chaine avec tout les blocs libre de la partition.

Suppression du premier bloc libre :

```
error removeFirstBlockInChain(disk_id id, int partition)
```

Supprime le premier bloc de la chaine et met à jour la description de la partition.

Ajout d'un bloc libre :

```
error addBlockInChain(disk_id id, int partition, int number)
```

Ajoute le bloc voulu en début de chaine et met à jour la description de la partition.

Gestion des entrées de fichier

Initialisation d'une entrée vierge :

```
void initFileEntry(FILE_ENTRY *file_ent)
```

Initialise tout les champs de l'entrée.

Ajout d'un nouveau bloc de données :

```
error addNewBlock(disk_id id, int partition, int file_pos)
```

Ajoute un nouveau bloc dans les direct blocs ou indirections. Met à jour le chainage et les informations de la partition.

Suppression du dernier bloc de données :

```
error removeLastBlock(disk_id id, int partition, int file_pos)
```

Supprime le dernier bloc dans les direct blocs ou indirections. Met à jour le chainage et les informations de la partition.

Liste des erreurs

Erreur 0 : _NOERROR

```
Aucune erreur.
```

Erreur 1 : _DISK_NOT_FOUND

```
Le disque n’existe pas.
```

Erreur 2 : _DISK_UNMOUNTED

```
Le disque n’est pas monté.
```

Erreur 3 : _NUM_BLCK_TOO_BIG

```
Le numéro de block est plus grand que le nombre total de blocks.
```

Erreur 4 : _DISK_ID_TOO_BIG

```
L’id demandée doit être inférieur à MAX_OPEN_DISK.
```

Erreur 5 : _DISK_ID_EXIST

```
L’id demandée existe déjà.
```

Erreur 6 : _READ_ERROR

```
La lecture du block à échoué.
```

Erreur 7 : _WRITE_ERROR

```
L’écriture du block à échoué.
```

Erreur 8 : _POS_IN_BLCK_TOO_BIG

```
La position est supérieur à BLCK_SIZE / 4 (256).
```

Erreur 9 : _PARTITION_NOT_FOUND

```
La partition n’existe pas sur le disque.
```

Erreur 10 : _MAX_FILES_TOO_BIG

```
Pas assez d'espace sur la partition.
```

Erreur 11 : _POS_IN_TABLE_TOO_BIG

```
Pas assez de place dans la table des fichiers.
```

Erreur 12 : _POS_IN_PARTITION_TOO_BIG

```
Pas assez de place dans la partition.
```

Erreur 13 : _TABLE_IS_FULL

```
La table des fichiers est pleine.
```

Erreur 14 : _CANNOT_REMOVE_FILE_ENTRY

```
La racine ne peut-être supprimée.
```

Erreur 15 : _PARTITION_IS_FULL

```
Plus aucun bloc n'est libre.
```

Erreur 16 : _DIRECT_TAB_IS_FULL

```
Il n'y à plus de place dans le tableau de blocs direct.
```

Erreur 17 : _DIRECT_TAB_IS_EMPTY

```
Le tableau de blocs direct est vide.
```