Kernel TP3 - Rapport système de fichiers - TexFS

Maxime Lovino Loic Willy

28 décembre 2017

1 Introduction

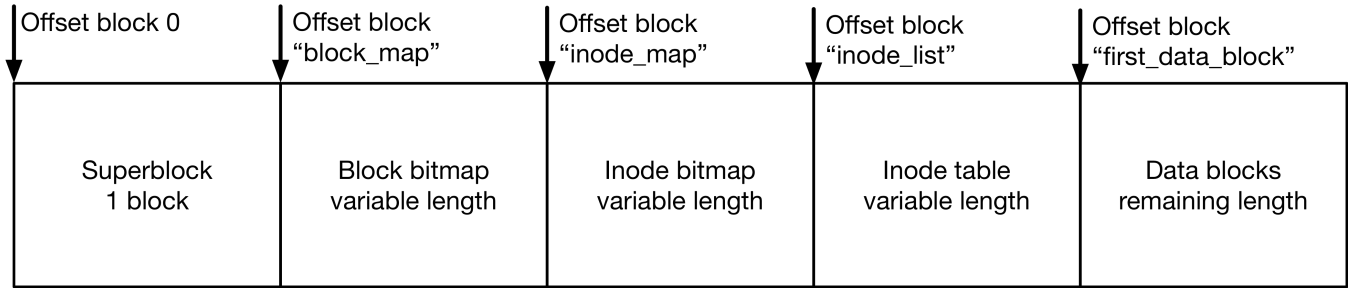
Pour réaliser le système de fichiers utilisé par notre Kernel, nous nous sommes inspirés du système de fichiers Ext2 sur lequel nous avons travaillé pendant un semestre en deuxième année. Nous avons adapté Ext2 par rapport aux besoins qui étaient spécifiés pour ce projet. C'est-à-dire que nous avons principalement simplifié Ext2 en supprimant les doubles et triples indirections, qui ne sont pas nécessaires compte tenu de la taille des fichier que nous devons stocker. Nous n'avons également pas tenu compte de la notion de groupes.

De ce fait, lors de la création d'une image TexFS, il est nécessaire de spécifier la taille de bloc, le nombre de bloc à allouer et le nombre de fichiers maximum pour l'image. La taille de bloc et le nombre de blocs devaient être spécifiés de toute façon, mais compte tenu de la structure utilisée, il est nécessaire d'entrer également le nombre de fichiers maximum que contiendra l'image. (Si nous avions des groupes, nous aurions pu allouer un nouveau groupe en cas de besoins de nouveaux fichiers)

2 Structure du système de fichiers

TexFS a une structure sur le disque similaire à Ext2. Nous n'avons par contre pas besoin de réserver le premier block pour le MBR car nous ne démarrons pas sur notre image. Du coup nous commençons au block 0 par le superblock, suivi du block bitmap, puis de l'inode bitmap et de la table des inodes. Après toutes ces blocs de métadonnées, nous avons les blocs de données.

A noter que l'offset `block_map` est toujours égal à 1.



2.1 Blocs de métadonnées

Cette section va rentrer plus en détail sur les blocs de métadonnées, étant donné que les blocs de données sont juste composées de données brutes ou d'index de blocs indirects (expliqué plus loin).

2.1.1 Superblock

```
#define MAX_LABEL_LENGTH 30
typedef struct tex_fs_superblock_st {
    uint16_t magic;
    uint8_t version;
    char label[MAX_LABEL_LENGTH];
```

```
    uint16_t block_size;
    uint32_t block_map;
    uint32_t block_count;
    uint32_t inode_bitmap;
    uint32_t inode_list;
    uint32_t inode_count;
    uint32_t first_data_block;
} __attribute__((packed)) tex_fs_superblock_t;
```

Le superblock comprend le `magic` de TexFS, c'est-à-dire `0xD0D0` ainsi que la version du filesystem (version 1) et le label choisi à la création de l'image. Ensuite le champ `block_size` nous permet d'avoir la taille des blocs, à noter qu'il s'agit d'un entier signé sur 16 bits, du coup nous avons une taille de bloc maximale de $2^{16} - 1 = 65535$ bytes, ce qui est suffisant dans notre cas.

Ensuite, `block_map`, `inode_bitmap`, `inode_list` et `first_data_block` représentent les numéros de blocks à laquelle commencent les sections correspondantes. `block_count` et `inode_count` représentent respectivement le nombre de blocks de l'image et le nombre d'inodes.

2.1.2 Block bitmap et inode bitmap

Les blocks et inode bitmap sont simplement des bitmap servant à spécifier quels blocks et inodes sont respectivement occupés. Pour ce faire, nous allouons un `uint_8` par block et par inode dans chaque cas. Il s'agit de la façon la plus simple de faire car les méthodes de lectures nous permettent de lire au minimum 1 byte. Nous aurions pu également utiliser simplement 1 bit et grouper 8 booléens dans un byte du coup, mais cela aurait ajouté de la complexité au moment de la lecture avec des décalages bits à bits.

2.1.3 Inode Table

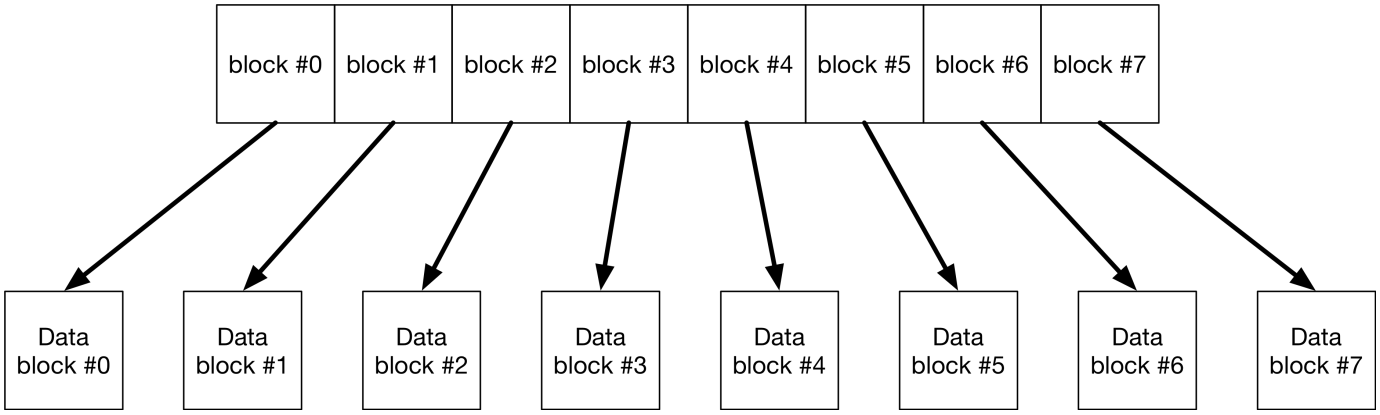
```
#define DIRECT_BLOCKS 8
#define INDIRECT_BLOCKS 4
#define MAX_FILENAME_LENGTH 64
typedef struct tex_fs_inode_st {
    char name[MAX_FILENAME_LENGTH];
    uint32_t size;
    uint32_t direct_blocks[DIRECT_BLOCKS];
    uint32_t indirect_blocks[INDIRECT_BLOCKS];
} __attribute__((packed)) tex_fs_inode_t;
```

L'inode contient toutes les métadonnées d'un fichier ainsi que les indices des blocs directs et indirects qui contiennent les données de ce fichier. En terme de métadonnées, on stocke la taille et le nom (maximum 64 char).

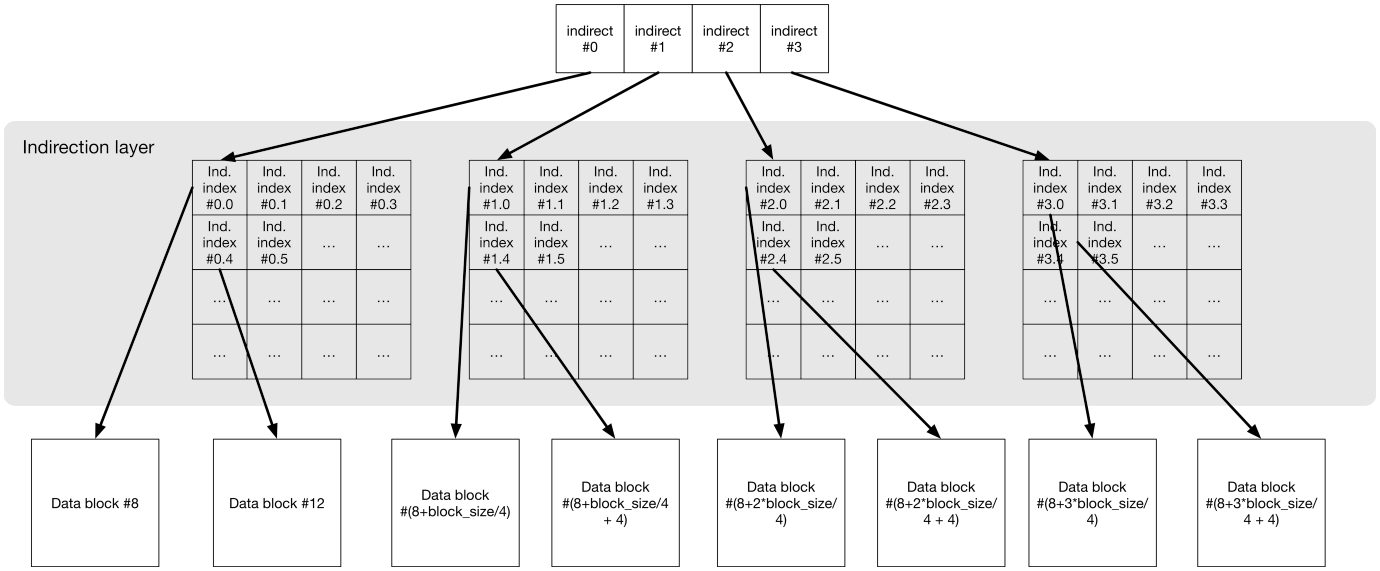
On a 8 blocs directs et 4 blocs indirects. En se basant sur le fait que chaque indice de bloc est stocké sur 4 bytes, on peut calculer la taille maximum d'un fichier en fonction de la taille d'un bloc n, on obtient $8 * n + (4 * \frac{n}{4} * n)$. Pour une taille de bloc minimale de 512 bytes, on peut stocker un fichier de taille $8 * 512 + (4 * \frac{512}{4} * 512) = 266240$ bytes, ce qui

est suffisant par rapport à la taille demandée de 256 kbytes.

Les données des blocs directs sont stockés de cette façon :



Les données des blocs indirects sont elles stockées de cette façon :



3 Exemples

Dans ces exemples, nous allons nous baser sur une image avec une taille de bloc de 512 bytes.

3.1 Block bitmap

Imaginons qu'à la création de l'image, l'utilisateur a demandé de créer une image contenant 1000 blocs.

Il faudra donc créer une block bitmap pour ces 1000 blocs demandés. Etant donné que chaque booléen de la bitmap est stocké sur 1 byte, nous aurons besoin de 1000 bytes pour stocker le block bitmap. Du coup il faut calculer le nombre de blocs nécessaire pour stocker le block bitmap. Dans ce cas, nous aurons besoin de 2 blocs car :

$$\lceil \frac{1000}{512} \rceil = 2$$

Du coup nous pouvons déjà savoir que l'offset `inode_map` sera 3 car l'offset `block_map` était de 1 et on y ajoute le nombre de blocs nécessaire, c'est-à-dire 2 dans ce cas.

3.2 Inode bitmap

Ensuite, imaginons que l'utilisateur a demandé d'allouer 100 inodes. Dans ce cas, un bloc suffit pour stocker l'inode bitmap.

Du coup l'offset suivant, c'est-à-dire `inode_table` sera 4.

3.3 Inode table

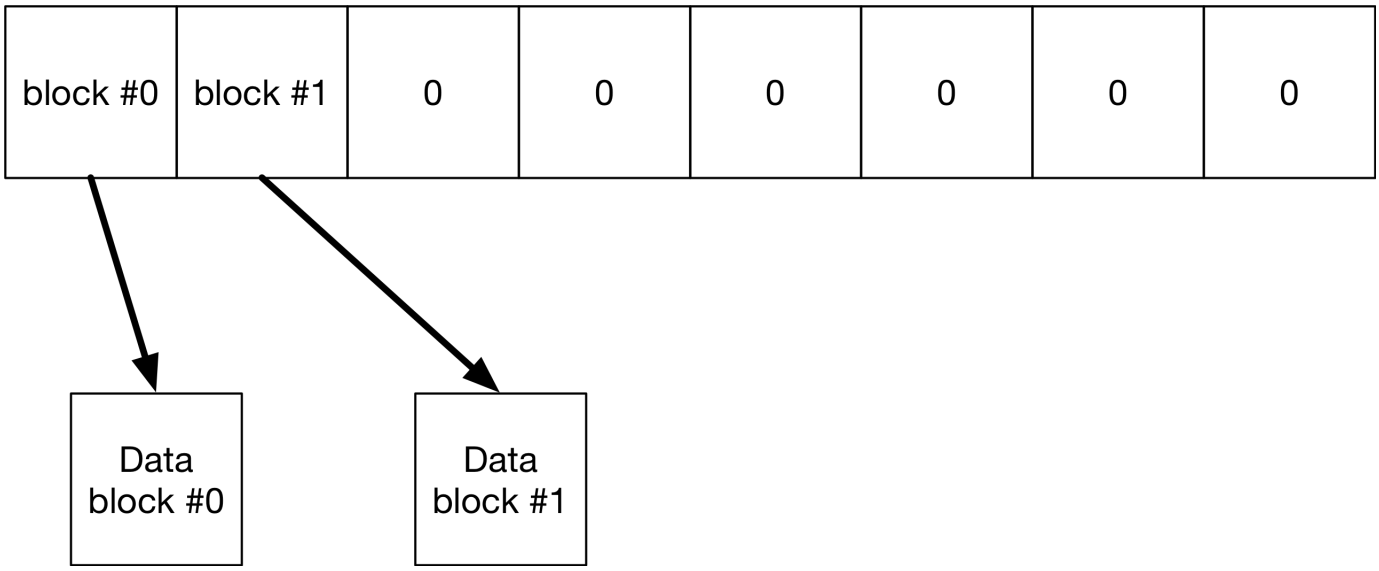
Pour l'inode table par contre, étant donné que la taille d'un inode est de 116 bytes (voir structure plus haut), il faudrait $100 * 116 = 1160$ bytes, dans ce cas il faudra :

$$\lceil \frac{1160}{512} \rceil = 3$$

Du coup 3 blocs seront nécessaires pour stocker la table des inodes, ce qui fait que l'offset pour le premier bloc de données sera $4 + 3 = 7$.

3.4 Stockage d'un petit fichier sans indirection

Imaginons maintenant que sur cette image terminée, nous voulons stocker un fichier de 800 bytes. Dans ce cas, le nombre de bloc nécessaire est de 2, donc on peut se contenter des blocs directs. Il suffit donc de trouver deux blocs libres et une inode libre. Ensuite au niveau du stockage du fichier, cela donne :



3.5 Stockage d'un grand fichier avec indirection

4 Implémentation

En terme d'implémentation, nous chargeons au niveau des tools et au niveau du kernel tous les blocs de métadonnées de l'image en RAM. Côté kernel, la structure et les tableaux contenant les métadonnées sont alloués statiquement au niveau de la fonction `kernel_entry` du kernel, le pointeur de `tex_fs_metadata` est ensuite passé à la fonction `fs_init()` qui va stocker ce pointeur dans une variable globale au fichier `fs.c`.

5 Avantages du système choisi

— Structure inspirée de Ext2, donc déjà connue
— Séparation des métadonnées et données
— Facile de stocker toutes les métadonnées en RAM
— La bitmap d'inodes et de blocks permet de trouver facilement des inodes et blocks libres
— Itération sur les fichiers simple (il suffit de prendre la prochaine inode utilisée)
— Les fichiers peuvent facilement grandir
— Accès aléatoire dans un fichier rapide (accès direct si block direct, lecture d'un bloc indirect puis accès direct si indirection)
— Pas de fragmentation externe

6 Inconvénients du système choisi

— Retrouver tous les blocks d'un fichier avec indirections est compliqué
— Cela influe sur la lecture et la suppression d'un fichier
— Pour itérer sur les fichiers, il faut parcourir toutes les inodes

— Overhead de stockage pour les blocks métadonnées
— Fragmentation interne dans certains blocks (superblock par exemple)
— L'accès séquentiel n'est pas plus rapide que l'accès aléatoire, car les blocks de données ne sont pas forcément contigus