TP4 – FAT 32

Dû le 30 avril à 23h00

Préambule

Ce TP requiert que vous aillez assisté à la démonstration sur FAT32. Si ce n'est pas le cas, vous aurez de la difficulté à compléter le TP. Celui-ci ne requiert pas beaucoup de lignes de code pour compléter. Cependant, le système de fichier FAT32 est intimidant au départ et une réflection est nécéssaire pour correctement accéder aux fichier. Afin d'alléger les difficultées reliées à suivre une démonstration en ligne, un document est accessible à l'addresse `https://github.com/IFT2245/FAT32`.

Introduction

Le système de fichier FAT aurait été dévelopé par Bill Gates lui-même sur une napkin dans un hôtel. C'est un système de fichier simple, robuste et relativement compact. La première implémentation visait l'utilisation sur des disquettes, mais son adoption générale en fait un système de fichier encore compatible avec beaucoup de matériel moderne. Bien que les besoins de performances associées à des utilisations commerciales des systèmes de fichier rendent FAT32 obsolète (il ne permet pas un accès très rapide aux fichiers), il reste important de nos jours et son étude est encore très pertinente.

Instructions générales

Afin de tester votre implémentation, un fichier **floppy.img** est fourni. L'arborescence de ce fichier est donnée en fin de l'énoncé. Ce fichier peut être accédé par les mécanismes standards de lecture de fichier en C. Cependant, la structure de ce fichier correspond **exactement** à ce que vous retrouveriez sur un disque dur (ou un floppy).

Si vous n'avez pas été en mesure d'assister à la séance de démonstration, où que vous avez besoin de ressources supplémentaires, voici quelques liens qui pourraient vous être utiles :

1. La documentation officielle de FAT32
2. `https://github.com/IFT2245/FAT32`

Afin de simplifier ce travail, vous n'avez pas à vous soucier du partitionnement du disque représenté par l'image. En effet, il y aura toujours une partition qui occupe l'entièreté du disque. De plus, l'image donnée ainsi que celles utilisées pour les tests correspondront toujours à de vraies images FAT32 correctes. Cependant, vous ne pouvez pas assumer que la géométrie de l'image sera toujours la même. Par exemple, un *cluster* pourrait contenir plus de secteurs pour un disque qu'un autre. De plus, les secteurs ne seront **pas** toujours 512 bytes.

Vous pouvez aussi assumer que le système de fichier n'est pas corrompu et donc que la première table FAT est valide.

Avant de commencer à regarder les fonctions à implémenter, prenez connaissance du squelette de code donné ainsi que des déclarations qui sont faites.

L'ordre des fonctions à coder peut sembler arbitraire. Cependant, il est conçu pour vous forcer à avoir des questions en premier et ainsi éviter que vous devez refaire des sections au complet parce que vous avez mal compris quelque chose. Cependant, vous pouvez faire les questions dans l'ordre que vous voulez. La question 5 pourrait être faite en premier afin de tester les autres fonctions. Cependant, comme dans la vraie vie, il faut parfois faire une quantitée de code non-triviale afin de pouvoir tester des choses plus petites. Cela fait partie de la difficulté de ce TP.

Objectifs d'apprentissage

Ce TP vise à vous faire découvrire les joies du dévelopement de support pour un système de fichier. En effet, vous aurez à implémenter une très petite partie de ce qui est nécéssaire pour accéder à des fichier sur un système de fichier FAT32. Comme vous verrez, il s'agit d'une tâche non triviale, même pour un système de fichier aussi simple que FAT32.

Voici une liste non-exhaustive des éléments pédagogiques que vous retrouverez dans ce TP
- Lecture et compréhension de documents techniques
- Principes généraux des systèmes de fichiers
- Implémentation de code résistant aux erreurs

Gestion des erreurs

Plusieurs erreurs peuvent se produire lors de la lecture d'un disque. Des outils sont mis à votre disposition dans l'entête du fichier pour traiter les erreurs. Vous devez respecter le fait qu'une erreur possède un code d'erreur négatif mais vous pouvez ajouter des codes d'erreurs et redéfinir les codes à votre guise. Parfois, le code d'erreur est utilisé pour retourner une donnée sur l'exécution de la fonction. Portez attention à la description des fonctions dans le code ainsi qu'à ce document pour faire la bonne chose. Portez attention aux arguments des fonctions. Des arguments nuls sont une condition suffisante pour provoquer une erreur lors de l'appel d'une fonctions.

Fonctions à implémenter

1- Cluster vers LBA

La première fonction que vous aurez à implémenter est la suivante :

```
uint32 cluster_to_lba(BPB *block,
                      uint32 cluster,
                      uint32 first_data_sector);
```

Cette fonction prend un numéro de cluster et retourne le LBA (logical block address) qui correspond au secteur à lire. La conversion doit s'effectuer selon la spécification pour que la lecture de fichier soit correcte. De plus, le "BIOS Parameter Block" (BPB) doit être utilisé pour tenir compte de la géométrie du disque. L'algorithme de conversion est donné verbatim dans notre document.

2- Aller chercher le prochain maillon de la chaîne FAT

```
error_code get_cluster_chain_value(BPB *block,
                                   uint32 cluster,
                                   uint32 *value,
                                   FILE *archive);
```

Cette fonction prend en paramètre (entre autre) un numéro de cluster. Cette fonction doit placer dans le pointeur value le cluster qui suit le cluster courant. Cette information se trouve dans la table FAT du disque. Portez attention à bien calculer l'adresse en byte du cluster courant. Si vous faites une erreur, le mauvais cluster sera lu et la lecture de fichier sera impossible. Il n'est pas nécéssaire de vérifier l'information avec la deuxième table FAT, seule la première suffira ici.

3- Vérifier le nom d'une entrée

```
bool file_has_name(FAT_entry *entry, char *name)
```

Le but de cette fonction est de comparer un nom de fichier ou de dossier FAT avec un nom de fichier entré par un utilisateur. Attention ! Il ne s'agit **pas** d'une simple comparaison de chaînes de caractères. Veuillez vous fier à la spécification officielle (qui est **très** claire au sujet des noms) pour comparer les noms. Vous n'avez pas nécéssairement besoin de gérer tous les cas spéciaux, mais cela pourrait être utile dans d'autres fonctions.

4- Analyser un chemin

```
error_code break_up_path(char *path, uint8 level, char **output);
```

Cette fonction va vous servir à prendre un chemin dans l'archive et le séparer en morceau. La spécification exacte de la fonction est donnée en docstring au dessus de celle-ci. Cette fonction ne requiert pas de connaissances particulières du système de fichier FAT32. Il faut bien sûr allouer une chaîne de caractère pour y placer le morceau du chemin voulu. De plus, un indicateur de chemin relatif (ici ".", "..") constitue un morceau de chemin valide, puisque ces marqueurs sont construits comme des fichiers dans FAT32 (on laisse donc ces morceaux dans la chaîne). Le erreur code, en cas ou il n'y a pas d'erreur, retourne la longueur de ce tableau.

5- Aller lire le *boot block*

```
error_code read_boot_block(FILE *archive, BPB **block);
```

Cette fonction va lire le boot block dans l'archive et va allouer la mémoire nécéssaire pour le placer. Cette fonction est essentielle pour le bon fonctionnement du reste du TP, puisque le boot block contient toute l'information nécéssaire pour la lecture de l'archive. En particulier, le BPB donne des informations quant à la géométrie du disque.

6- Trouver un descripteur de fichier

```
error_code find_file_descriptor(FILE *archive,
                                BPB *block,
                                char *path,
                                FAT_entry **entry);
```

Cette fonction reçoit en entrée un path qui correspond au chemin du fichier à aller chercher. Cette fonction doit donc rechercher le fichier au travers l'arborescence de dossiers. La fonction doit être capable de déterminer que le chemin n'est pas valide. Par exemple, un chemin qui utilise un fichier comme un dossier ne devrait pas fonctionner.

7- Aller lire le contenu d'un fichier

```
error_code read_file(FILE *archive,
                     BPB *block,
                     FAT_entry *entry,
                     void *buff,
                     size_t max_len);
```

Cette fonction prend en paramètre une entrée FAT ainsi qu'un buffer et sa longueur maximale. Si le descripteur décrit bien un fichier, le contenu du fichier est lu au complet (à moins de dépasser la longueur maximale, dans ce cas, il est partiellement lu) et placé dans le buffer. Le code d'erreur doit retourner le nombre de bytes lu s'il n'y a pas d'erreur. Si le fichier n'est pas dans l'arborescence, vous devez retourner un code d'erreur.

Le path pourrait contenir des éléments de chemin relatif (par exemple, des ".."). Afin de minimiser le travaille à faire, je vous conseille **fortemment** de lire la spécification FAT32 au sujet de la structure des dossiers, qui explique comment les relations entre ceux-ci sont modélisées dans le système de fichier FAT32.

Contenu de l'image

Si vous utilisez linux, vous pouvez *monter* l'image pour modifier son contenu. Cela vous permet d'ajouter des fichiers, et de changer l'architecture de l'image. La commande s'effectue ainsi :

```
mount floppy.img mntpnt
umount mntpnt
```

La commande *mount* permet de mapper l'archive à un dossier, ce qui permet d'opérer dessus comme si c'était un disque. La commande *umount* permet de défaire l'opération et de continuer à utiliser l'archive.

Attention ! La commande *mount* ne fonctionne pas sur le WSL Attention !

Si vous modifier l'archive, portez attention à entrer des noms de fichiers en majuscules, de taille inférieure ou égale à 11, incluant l'extention. Sinon, votre système créera des fichiers en utilisant

l'extention *long file name* à FAT32, ce qui rendra les noms de fichiers différents de ce que vous vous attendez. L'archive donnée tiens compte de cette particularité.

L'image **floppy.img** contient le système de fichier suivant :

```
----AFOLDER
|     |---ANOTHER
|        |-- CANDIDE.TXT
|---SPANISH
|    |---LOS.TXT
|    |---TITAN.TXT
----ZOLA.TXT
----HELLO.TXT
```

Cette représentation à été obtenue en appelant la commande *tree* à la racine du dossier de l'archive. Les fichiers sont des livres du Projet Gutenberg. Leur longueur dépend des fichiers, vous pouvez les monter pour aller les lire. Le fichier HELLO.TXT contient le texte "Bonne chance pour le TP4". Celui-ci est plus petit et est contenu dans un cluster ce qui devrait faciliter la tâche au départ.

Remise

Ce travail est à faire **en équipe de 2**. Le code est à remettre par remise électronique **ici** avant la date indiquée.

Chaque jour de retard est -15%, mais après le deuxième jour la remise ne sera pas acceptée.

Indiquez clairement votre nom et votre matricule dans un commentaire en bloc au début de chaque fichier, dans ce format :

```
Prenom1 NomDeFamille1 Matricule1
```

Vous pouvez valider l'entête de votre fichier en utilisant le projet situé ici : `https://github.com/IFT2245/NameValidator`. Une entête invalide se méritera la note de 0.

Le programme doit être exécutable sur les ordinateurs du DIRO, mais comme ils vous est difficile d'y accéder, nous ferons preuve de bonne foi.

Pour la remise, vous devez remettre un **unique** fichier (main.c) par la page Studium du cours dans un fichier zip.

Évaluation du TP

Nous évaluerons votre TP principalement grâce à la fonction read_file. Si le contenu d'un fichier est lu correctement, le test passera.

Barème

— Votre note sera divisé équitablement entre chaque question, sauf les questions 6 et 7 qui ont pour valeur l'équivalent de 2 questions.
— Tout usage de matériel (code ou texte) emprunté à quelqu'un d'autre (trouvé sur le web, etc.) doit être dûment mentionné, sans quoi cela sera considéré comme du plagiat. Si pour une question votre solution est directement copiée, même si il y a attribution de la source, cette question se verra attribuée la note de zéro. Vous pourrez cependant l'utiliser dans les sections suivantes sans pénalité.
— Vous serez pénalisés pour chaque warning lors de la compilation à raison de 1% par warning, sauf pour les warning reliés à l'assignation à NULL, à la comparaison avec NULL, et aux override des fonctions de librairie. Les warning qui sont les mêmes que ceux retrouvées
— Si une fuite mémoire mémoire est identifiée, vous perdrez 15%. Vous ne perdrez pas plus de points pour les fuites si vous en avez une ou trente.
— Les accès mémoire illégaux identifiés par Valgrind entraîneront jusqu'à 5% de perte, à raison de 1% par accès. La répétition d'un même accès sera comptée comme 1% de plus quand même.
— Votre devoir sera corrigé automatiquement en très grande partie. Si vous déviez de ce qui est demandé en output, les points que vous perdrez seront perdus pour de bon. Si vous n'êtes pas certains d'un caractère demandé, demandez sur le forum studium et nous répondrons de façon à ce que chaque étudiant puisse voir la réponse.
— La méthode de développement recommandée est d'utiliser CLion et son intégration avec Valgrind. Si vous voulez utiliser d'autres techniques, vous pouvez le faire. Cependant, nous ne vous aiderons pas si vous rencontrez des problèmes avec ces techniques.