Rapport du projet : Mini S.G.F

Lucas **LOIGNON** – Christopher **CAUET**

Table des matières

Introduction

Le but ce de projet était de comprendre le fonctionnement d'un système de gestion de fichier en simulant un mini-système.

Certaines fonctionnalités étaient implémentées par défaut, notamment celles très « bas-niveau », d'autres telles que l'affichage du contenu du disque, la lecture d'un fichier ou l'écriture bloc par bloc étaient à réaliser.

Ce projet s'est révélé, par moment, complexe et difficile à appréhender mais nous a permis de vraiment nous former et d'assimiler les concepts essentiels de la gestion des fichiers sur une machine. C'est un composant absolument essentiel d'un système d'exploitation, et pas assez souvent mis en avant ; avant ces TPs et ce projet nous n'avions pas forcement conscience d'à quel point un SGF était important et difficile à mettre en œuvre.

Bien évidemment notre « mini-SGF » est très basique, mais il nous a ouvert les yeux une fois de plus sur la puissance et la complexité de nos systèmes informatique modernes, nous imaginons maintenant à quel point une simple ouverture-écriture d'un document PDF implique des traitements immenses du côté de la machine.

Arborescence du projet

Notre archive de rendu possède un dossier *CODE* qui, comme son nom l'indique, rassemble les fichiers du « mini-SGF ».

Nous avons principalement modifié les fichiers :

- *main.c* : Initialisation du SGF et lancement d'une fonction de test (voir la section Tests).

- *sgf-io.c* : Comme son nom l'indique (IO), c'est ici que se rassemblent l'ensemble des fonctionnalités d'entrées et de sorties. C'est dans ce fichier que la plupart du travail a été réalisé.

- *Makefile* : Pour la compilation ; Ajout des options « -w », « -g » et « -std=c99 » à des fins de débogage ou d'autoriser certaines formes de code (commentaires en //, déclaration du int dans la condition d'un for, etc…)

Lecture dans un fichier

LIST_DIRECTORY

La fonction *list_directory* consiste, comme vu en TD, à parcourir toutes les entrées de la **FAT** afin de chercher les fichiers présents sur notre disque. Ces fichiers sont décrits par un **inode**, par conséquent, au cours de notre parcours, des lors que nous tombons sur un bloc **inode**, nous le lisons afin d'extraire les informations du fichier comme sa longueur, son premier et son dernier bloc de contenu.

SGF_READ_BLOC

La fonction *sgf_read_bloc* consiste à chercher le bloc logique *n* d'un fichier *f*. Pour cela, on part du premier bloc (ciblé par *f->first*), et on passe n fois au bloc suivant (adresse retournée par get_fat(bloc_actuel)). Si aucune erreur n'a été rencontrée jusqu'à la fin de ce *while*, alors nous avons l'adresse du bloc logique n et nous pouvons alors charger son contenu dans le buffer du fichier f.

Accès direct en lecture

SGF_SEEK

La fonction *sgf_seek* consiste à décaler le pointeur dans un fichier (*f->ptr*) du nombre de caractères précisé en paramètre.

La première chose à faire est donc de vérifier que le décalage demandé est possible, c'est-à-dire que à partir de la position actuelle, le décalage n'excède pas la taille du fichier.

Cette fonction va donc décaler le pointeur du fichier *f* de *n* caractères, le nombre passé en paramètre, et s'assurer de charger dans le tampon le bloc courant si le décalage ne se situe pas aux « bornes » du fichier (%*BLOCK_SIZE*).

Ecriture dans un fichier caractères après caractères

SGF_PUTC

La fonction *sgf_putc* consiste, comme vu en TD, à écrire le caractère *c* passé en paramètre dans le tampon du fichier *f* ouvert (en *WRITE_MODE* ou *APPEND_MODE*).

D'abord, nous vérifions que le fichier *f* soit bien ouvert en écriture.

Par la suite la fonction va écrire le caractère à la fin du buffer du fichier *f* (*f->buffer*) et incrémenter son pointeur de 1 (*f->ptr*). Si le pointeur atteint la taille maximum d'un bloc (*BLOCK_SIZE*) on fais alors appel à *sgf_append_block* afin de faire écrire le bloc alors complet sur le disque et pouvoir continuer sur un nouveau bloc (traité par la fonction *sgf_append_block*, *sgf_putc* se contente de manipuler le buffer du fichier via le pointeur modulo la taille des blocs).

SGF_APPEND_BLOCK

La fonction *sgf_append_block* va, comme vu dans le dernier TD, ajouter un bloc à la fin d'un fichier ouvert (pointée par *f->last*). Cette fonction va aussi mettre à jour les nouvelles informations du fichier (*f->first, f->last, f->length*).

Deux cas sont à différencier : le mode d'ouverture du fichier implique des traitements différents :

- *WRITE_MODE* Dans ce cas, on ne cherche pas à préserver le contenu précédent, on cherche directement un bloc libre (via la fonction *alloc_block*) et on écrit le contenu du buffer dedans

- *APPEND_MODE* Dans ce cas se place au dernier bloc (*f->last*) pour ne pas perdre le contenu des autres blocs, on écrit le contenu du buffer (les fonctions d'écriture/ouverture sont tenues de ne pas « *override* » le contenu précédent du bloc et on suppose donc que le contenu du buffer contient bien **ancien_contenu** + **nouveau contenu**). On change alors le mode d'ouverture à *WRITE_MODE*, car une fois le bloc complété et écrit, nous sommes dans le cas ou du nouveau contenu implique un nouveau bloc donc le comportement du *WRITE_MODE*)

SGF_CLOSE

Comme vu dans le TD 10, la fonction *sgf_close* va fermer correctement le fichier *f* passé en paramètre, en prenant soin de sauvegarder sur le disque le tampon si le fichier était ouvert en écriture (*WRITE* ou *APPEND*) et que le tampon n'était pas vide (via la fonction *sgf_append_block*)

SGF_REMOVE

La fonction *sgf_remove* consiste à “détruire” un fichier f dont l’adresse de **l’inode** est passé en paramètre.

Concrètement, *sgf_remove* va parcourir tous les blocs du fichier (via la fonction *get_fat* qui nous donne le bloc suivant), les mettre à *FAT_FREE* (via la fonction *set_fat*). Une fois fait elle remettra à l’état initial **l’inode** du fichier (**first** = *FAT_EOF*, **last** = *FAT_EOF* et **length** = 0) et enfin elle libera également **l’inode** en lui correspondant la valeur *FAT_FREE* dans la table **FAT**.

Ouvrir un fichier en mode ajout

SGF_OPEN_APPEND

Le contenu de base de la fonction *sgf_open_append* est le même que celui de la fonction *sgf_open_write*. Il diverge lors de la définition des valeurs de notre fichier. En effet, puisqu’on veut ajouter et non écraser le contenu de notre fichier ouvert, on doit commencer par mettre les valeurs du fichier au dernier « état connu ». Ainsi, la longueur du fichier, est celle que possède l’inode du fichier, le premier et le dernier bloc sont les mêmes que ceux décrits par l’inode du fichier, et le pointeur (*ptr*) est placé au dernier caractère du fichier, donc la taille totale du fichier.

Deux cas sont ensuite à distinguer :

Si la taille du fichier était une taille multiple de la taille des blocs (=> le dernier bloc est complet), alors on doit commencer par allouer un nouveau bloc, cette situation revient au mode *WRITE_MODE*. On change donc le mode du fichier pour *WRITE_MODE*.

Dans le cas contraire, on lit le contenu du dernier bloc (incomplet donc) et on le charge dans le buffer, pour que lors de l’écriture en append, le contenu soit préservé.

Ecrire une zone de données

SGF_WRITE

Le but de la fonction *sgf_write* est d’écrire du contenu dans un fichier mais contrairement à la fonction *sgf_puts*, qui ajoute caractère par caractère, on veut ajouter bloc par bloc.

On commence par créer un pointeur sur la position de notre chaine de caractère a écrire, qui commence à 0. Tant que ce pointeur n’est pas égal à la taille du fichier (c’est à dire que la chaine n’a pas été entièrement copiée) on l’ajoute autant que possible au buffer, et une fois le buffer complet, on ajoute ledit buffer au fichier (via *sgf_append_block*). On utilise la fonction *memcpy* pour copier directement des zones mémoires, et on utilise des décalages de pointeur pour compléter notre buffer avec un bout de notre chaine d’entrée précis.

Fonctions de test

- ## test_seek

  - On commence par lister le contenu du disque afin de confirmer la présence du fichier
  - On lit le fichier une première fois pour voir le texte qu'il contient.
  - On lit une nouvelle fois le fichier mais cette fois en ne lisant que les caractères dont la position est multiple de 8 (on lit, on décale de 8, on lit, on décale de 8, etc…)

- ## test_remove

  - On liste le contenu du disque puis on affiche le contenu du fichier (*essai.txt*)
  - On le supprime via la fonction *sgf_remove*
  - On ouvre ensuite le fichier en *WRITE_MODE* et on y écrit du contenu
  - On liste à nouveau le contenu du disque et on s'assure que le premier fichier à bien été supprimé, puis recrée et contient le nouveau contenu. (Même **inode**)

- ## test_append

  - On crée un fichier (via le *WRITE_MODE*) et on y écrit quelques caractères
  - On liste le contenu du disque pour s'assurer de sa création et de sa taille
  - On rentre dans une boucle qui va ouvrir le fichier en *APPEND_MODE*, y écrire un caractère et le fermer, et cela 500 fois.
  - On liste une nouvelle fois le contenu du disque et on lit le fichier pour vérifier que les caractères ont bien été écrits (et exactement où il devaient être écrits)

- ## test_write

  - On crée un fichier (*WRITE_MODE*) et on écrit dedans un premier contenu
  - On vérifie la création du fichier et on lit son contenu
  - On ouvre maintenant le fichier en *APPEND_MODE* puis on écrit dedans avec *sgf_write*
  - On liste une nouvelle fois le contenu du disque et celui du fichier pour s'assurer que l'écriture « bloc par bloc » s'est bien déroulée. L'affichage pendant l'écriture permet aussi de confirmer ladite écriture.

Notes

Dans la présentation des fonctions de test :

- « lire » le fichier correspond à l'ouvrir en *READ_MODE* et à faire un *sgf_gets* / *printf* (ou *putc*) pour lire et afficher caractère par caractère le contenu du fichier.

- « écrire » dans le fichier correspond à l'ouvrir en *WRITE_MODE* ou *APPEND_MODE* et à faire un *sgf_puts* pour y écrire du contenu. Quand c'est la fonction *sgf_write* qui est utilisée, cela est précisé.

- « lister » le contenu du disque est un appel à la fonction *list_directory*, ou dans notre cas *ls* qui est définie dans *main.c* et fais appel à *list_directory* mais avec de l'affichage supplémentaire.

- Dans la fonction main sont définies trois variables chaines de caractères (char *) *text1*, *text2* et *text3* de respectivement 140, 280 et 1000 caractères, à utiliser en paramètre de nos fonctions de test pour tester plus facilement sur différentes tailles d'écriture.