LP25

Quentin Trombini, Jules Ramos, William Mann, Erwann Le Guilly

Juin 2021

Table des matières

1 Introduction

L'objectif de ce projet est de créer un programme qui va parcourir récursivement une arborescence de fichier. Durant ce parcours, il va stocker les informations sur tous les fichiers dans une liste chaînée. Une fois que le programme a aura parcouru tout le fichier il va transcrire la liste chaînée dans un fichier nommé en fonction de la date et de l'heure.

Chapitre 1

Le projet

Notre projet est composé de plusieurs parties. Nous allons d'abord présenter le projet dans sa globalité, suivi d'une présentation des différentes parties détaillant les fonctions pertinentes

1 Vue globale du projet

Notre projet est constitué d'un Makefile et de 10 fichiers : 5 fichiers C (main.c, save.c, tree.c, scan.c, md5sum.c) et cinq fichiers headers (type.h, save.h, tree.h, scan.h, md5sum.h). Quatre des fichiers C vont donc de paire avec un fichier header qui leur est attribué contenant les prototypes de ses fonctions. Le fichier type.h contient toutes les structures et énumérations définies afin d'être utilisées dans l'ensemble du programme, expliquées ci-dessous, ainsi que l'ensemble des inclusions de librairies C qui sont donc regroupées et données une unique fois en un unique morceau de code. Le main contient une gestion des options et la structure principale du programme faisant appel aux différentes fonctions des quatre autres fichiers C de manière assez classique.

2 Les structures et les types utilisés

Nous avons dû utiliser certains types prédéfinis et certaines structures imposées afin de compléter le projet.

Deux structures et une énumération étaient imposées pour contenir l'arborescence, une structure était adaptée aux répertoires et une aux fichiers :

```
typedef enum {DIRECTORY, REGULAR_FILE, OTHER_TYPE} e_type;

typedef struct _file {
    e_type file_type;
    char name[NAME_MAX+1];
    time_t mod_time;
    uint64_t file_size;
    u_char md5sum[MD5_DIGEST_LENGTH];
    struct _file *next_file;
} s_file;
```

```
typedef struct _directory {
    char name[NAME_MAX+1];
    time_t mod_time;
    struct _directory *subdirs;
    s_file *files;
    struct _directory *next_dir;
} s_directory;
```

Il est facile de comprendre le rôle de l'énumération : expliciter les trois types d'éléments traités par le programme plutôt que de donner un lexique entre entiers et types dont tout le monde doit se souvenir sans faire d'erreur.

Les deux structures définies sont relativement similaires, les fichiers nécessitant plus de variables pour stocker des informations telles que leur taille alors que les répertoires doivent contenir leur arborescence. La présence de pointeurs vers d'autres structures de fichiers ou répertoires nous a immédiatement rappelé la LO21 et mis sur la piste d'utiliser ces structures comme des listes chaînées plus ou moins complexes (simplement chaînée pour les fichiers, jusqu'à triplement chaînée pour les répertoires).

3 Le fichier scan

Le fichier scan va prendre en entrée un char* contenant le chemin du répertoire dans lequel il doit s'effectuer et le node de la liste chaînée correspondante. Lors du premier appel a la fonction la node renverra vers NULL. Ensuite pour chaque élément du répertoire (à l'exception de ./ et ../) le programme va vérifier le type de l'élément(directory, file, link, etc.).

Une fois la vérification terminée, si l'élément est un fichier, il va crée une node rempli avec toutes les informations du fichier, puis envoyé la node nouvellement créer dans la liste chaînée. Dans le cas ou l'élément est un dossier une fonction avec le même objectif va être appelé : mais après avoir envoyé la node nouvellement créée, le programme va relancer la fonction scan() avec le chemin vers ce nouveau chemin vers le sous-dossier.

4 Le fichier save

Une seule fonction était demandée pour ce fichier, cependant nous avons décidé d'en ajouter 2. La fonction imposée, save_to_file, a pour but de parcourir la structure contenant l'arborescence complète du répertoire donné en paramètre et d'en afficher les éléments sous une forme imposée :

```
    0 date chemin/
```

pour un répertoire,

```
    1 date taille MD5 (si le parametre est actif) chemin
```

pour un fichier standard et

```
    2 date chemin
```

pour un autre type de fichier, tous les champs étant séparés par des tabulations.

Nous avons donc décidé d'utiliser cette fonction pour effectuer la récurrence qui va permettre d'afficher l'ensemble des répertoires dans un ordre logique (répertoire, ses sous-répertoires, puis le répertoire suivant). Pour cela, cette fonction ne va afficher les informations que pour le répertoire qui lui est donné, puis appeler les deux fonctions rajoutées pour donner les informations sur le contenu de ce répertoire, puis s'appeler elle-même pour le premier sous-répertoire s'il y en a un, puis s'appeler une seconde fois pour le répertoire suivant. Les sous-répertoires ayant la même structure qu'un répertoire standard, la récurrence va alors se ré-appliquer dans le sous-répertoire, listant alors tous les sous-sous-répertoires, et ainsi de suite. L'affichage des informations se fait dans un fichier dont le chemin d'accès est donné en second paramètre. Un 3e paramètre est le booléen indiquant si l'option d'affichage de la md5 a été demandée ou non.

La première fonction rajoutée est write_files et va permettre, en lui donnant le pointeur du premier élément de la liste chaînée de fichiers, d'afficher les informations demandées sur chaque fichier de la liste. Cette fonction est récursive et ressemble fortement aux fonctions d'affichage de liste chaînée réalisées en LO21, c'est pourquoi un seul appel dans save_to_file est nécessaire. Les paramètres sont donc le premier élément de la liste, le fichier d'affichage et le booléen qui indique si la md5 doit être donnée.

Suivant le même principe, write_directories est utilisée pour, à partir du premier élément de la liste chaînée de répertoires formant les sous-répertoires du répertoire source de l'itération actuelle de save_to_file, d'afficher uniquement la liste (avec bien sûr les informations demandées) des sous-répertoires directs de ce répertoire "source". De nouveau, une simple récurrence est utilisée. Les paramètres sont le premier élément de la liste et le fichier d'affichage.

save_to_file va donc appeler, avant sa double récurrence, ces deux fonctions, listant ainsi tous les fichiers et sous-répertoires que le répertoire source contient.

Nous avons pris à la fin du projet la décision de réaliser les free() des différentes structures dans save.c après en avoir extrait les informations. Cela est simplement revenu à ajouter un free du s_directory après l'appel doublement récursif dans save_to_file et du s_file après l'appel récursif dans write_files.

5 Le fichier tree

```
int append_subdir(s_directory *child, s_directory *parent);
```

Cette fonction ajoute les sous-répertoires d'un répertoire à ce dernier pour l'affichage qui en suivra. La valeur de retour est 0 si la fonction a rencontré une erreur et 1 sinon.

```
int append_file(s_file *child, s_directory *parent);
```

Cette fonction ajoute les fichiers d'un répertoire à ce dernier pour l'affichage qui en suivra. La valeur de retour est 0 si la fonction a rencontré une erreur et 1 sinon.

```
void clear_files(s_directory *parent);
```

Cette fonction supprime la mémoire des fichiers contenus dans de parent.

```
void erase_file_data(s_file *parent);
```

Cette fonction supprime les fichiers contenus dans la mémoire de parent de façon récursive.

```
void clear_subdirs(s_directory *parent);
void erase_dir_data(s_directory *parent);
```

Ces fonctions suppriment respectivement les sous-répertoires puis les répertoires de la mémoire contenue dans parent.

6 Le fichier md5sum

Pareillement au fichier save, une seule fonction était demandée pour ce fichier, mais afin de répondre aux demandes du projet, c'est-à-dire d'afficher le résultat de la fonction de hachage MD5 pour chaque fichier, et ce, en hexadécimal, une fonction de conversion vers le système hexadécimal a été ajoutée.

```
int compute\_md5(char *path, unsigned char buffer[])
```

Cette fonction calcule la somme MD5 du fichier indiqué par path*, et stocke cette valeur dans le tableau de caractères non signés buffer[]. Elle renvoie 0 (false) en cas d'erreur et 1 (true) en cas de succès.

La génération du hachage MD5 se fait à partir de plusieurs fonctions :

```
MD5_Init(&ctx);
```

Cette fonction initialise une structure MD5_CTX, ici ctx.

```
MD5_Update(&ctx, tmp_buffer, length);
```

Cette fonction met à jour la valeur du hachage MD5 actuellement contenue dans ctx, et permet de générer le hachage MD5 d'un fichier morceau par morceau. La valeur correspondant à la longueur, ici length, doit être exprimée en octets, ce qui est fait avec le type size_t, et à l'aide de la fonction fread().

```
MD5_Final(buffer, &ctx);
```

Cette fonction place le contenu du hachage MD5, ctx, dans buffer[], si assez d'espace y est présent (16 octets d'espace). Le contenu de ctx est ensuite effacé.

Les fonctions présentées précédemment renvoient 0 en cas d'erreur et 1 en cas de succès.

```
void md5\_to\_hexa(unsigned char md5\_array[], char* md5\_old)
```

Cette fonction convertit la valeur de la somme MD5 du fichier contenue dans md5_old*, et stocke cette valeur dans le tableau de caractères non signés md5_array[] caractère par caractère.

7 Le fichier main

Le fichier main du programme n'est pas très complexe, puisqu'il ne fait que reprendre les fonctions définies par les fichiers présentés précédemment afin de composer le programme. Il est ainsi composé principalement d'un getopt, quelques vérification pour les options entrée puis la fonction de lecture et de sauvegarde des données.

L'inclusion ou non des options dans le programme est gérée grâce à 3 variables de type booléen qui sont true dans le cas où l'option est demandée et donc reconnue par getopt. Pour la spécification du répertoire à analyser et du fichier de sauvegarde, cela entraîne simplement via un if une différence du chemin donné aux variables de type string qui sont ensuite utilisées dans les différents appels de fonctions. Pour l'affichage de la md5, le booléen est donné à la fonction save_to_file de save.c qui va ensuite écrire ou non la md5 dans le fichier de sauvegarde en fonction de cette variable. Cette solution de passer par 3 booléen nous a paru la plus simple et efficace pour gérer les 3 options demandées.

Chapitre 2

RETEX

Tout au long de la réalisation du projet, nous avons rencontré des difficultés :

1 Difficultés rencontrées

1.1 Code

En écrivant le code pour chacun de nos fichiers, nous avons rencontré quelques problèmes auxquels nous avons éventuellement remédié :

- Structures : Le problème majeur a été de ne pas inclure plusieurs fois les mêmes structures afin d'éviter les conflits lors de la mise en commun des fichiers.
- Scan : scan n'avait pas de problème majeur sauf sa longueur et répétitivité. Cependant, l'utilisation de fonctions qui ont depuis été supprimées de scan empêchaient à l'arborescence créée de subsister au delà de l'appel de la fonction scan().
- Save : Le principal problème de save a été de concevoir la récursivité de save_to_file, les autres fonctions étant relativement classiques pour des listes chainées. save_to_file a notamment demandé une schématisation de la récursivité, conduisant à des schémas similaires à des fractales, et une bonne compréhension de la construction de l'arborescence grâce aux structures demandées. La logique a ici été l'élément le plus bloquant, une fois comprise il a suffi d'une petite recherche pour comprendre le fonctionnement de localtime et le code a été écrit très rapidement.
- Tree : tree était assez simple et n'avait pas de problème majeur, sauf un free() mal placé qui a rendu le débogage infernal
- MD5 : L'utilisation des fonctions en rapport avec le hachage MD5 a posé problème lors de la compilation de cette partie avant de la lier au reste, car elles nécessitaient l'import de librairies jusque-là inutilisées.

1.2 Compilation

Après que chacun ai fini sa partie, nous avons également rencontré des erreurs lors de la compilation du projet dans son entièreté :

- Erreurs lors de l'utilisation du Makefile pour trouver les fichiers headers.
- Problème d'inclusions multiples de variables lors de la compilation :

```
stdint-uintn.h:39: multiple definition of specific_save_file';
    struct_stat.h:39: first defined here
```

2 Solutions

Les solutions respectives sont :

2.1 Code

— Structures : Ajout d'instructions préprocesseur manquantes dans le fichier header :

```
#ifndef TYPE
#define TYPE
    /*stuff that was already there*/
#endif
```

— Scan : Le fichier a été retravaillé avec tout le monde pour le raccourcir et le rendre plus rapide. Le problème de suppression de l'arborescence a été corrigé en intégrant les anciennes fonctions à scan() directement.
— Save : La double récursivité expliquée précédemment ainsi que l'utilisation de deux fonctions ajoutées par rapport au sujet et récursives ont permis de résoudre de manière logique le problème du parcours de la structure.
— Tree : Le free() a été enlevé.
— MD5 : Ajout de -lssl et de -lcrypto lors de la compilation du fichier md5sum.c

2.2 Compilation

— Ajout d'instructions préprocesseur dans les headers puis suppression des fichiers objets (qui ne pouvaient être actualisés sinon) avant la nouvelle tentative de compilation.
— On a eu quelques soucis au départ avec le MakeFile, ce qui nous a contraint à mettre les ".h" et ".c" dans le même fichier

3 Expériences individuelles

3.1 Erwann

Malgré quelques complications, la réalisation du projet n'a pas posé de problèmes trop contraignants et s'est donc faite dans le temps imparti. La division des tâches au début s'est faite rapidement et a garanti un travail efficace, nous permettant ensuite de gérer la compilation du projet ensemble avec assez de temps épargné afin de correctement tester le programme. N'ayant jamais utilisé GitLab, mais ayant déjà travaillé à l'aide de GitHub dans le cadre de projets à plusieurs, La transition n'a pas posé de problème. De plus, nous avons pu nous voir, à la fois en ligne et sur place, afin d'avancer ensemble sur la résolution de certains problèmes rencontrés vers la fin du projet.

3.2 Jules

En me basant sur le déroulement du projet, la seule chose que je changerais serait de commencer à le travailler plus tôt (mais cela aurait nécessité un effort de prévision pour savoir quand la charge de travail s'intensifierait en avance). Autrement, je trouve que nous avons bien réussi à nous organiser en groupe, le débug a été complexe, mais cela n'est pas inhabituel compte tenu du langage utilisé, C étant un langage de bas niveau. Les tâches ont été rapidement et efficacement réparties, et tout le monde s'est mis au travail simultanément, chacun à son rythme, en fonction de la complexité du fichier et de la proportion de débug nécessaire. Le résultat final me satisfait, en prenant en compte tous les facteurs externes, et je pense que notre rendu montre bien la variété des notions acquises au cours de ce semestre.

3.3 Quentin

Ce projet est le premier vrai projet qu'il a été nécessaire de faire en groupe de plus de deux personnes, c'est donc une toute nouvelle organisation qu'il a fallu avoir, car aucun de nous n'était habitué à ce type de distribution des taches. Heureusement tout le monde dans le groupe était motivé pour réussir à mener ce projet à terme et la communication était bonne. L'utilisation de Gitlab avec autant de personnes a aussi été une nouveauté, les commits des autres était bien plus fréquents que ce à quoi j'ai été habitué.

3.4 William

Nonobstant le départ tardif de ce projet, celui-ci c'est déroulé sans trop de soucis. Ayant fait principalement des projets en duo, ce projet à quatre fut un vrai renouveau. On s'était dit que en terme d'organisation ça allait être plus compliqué, mais tout s'est fait naturellement et le projet a avancé très rapidement. Évidemment la partie la plus éprouvant fut le débogage et donc de retravailler la partie des autres membres pour que tout fonctionne ensemble. Je conclurais donc en disant que ce projet fut très intéressant et j'ai découvert de nouvelles manières de travailler en équipes et sous un nouvel OS.

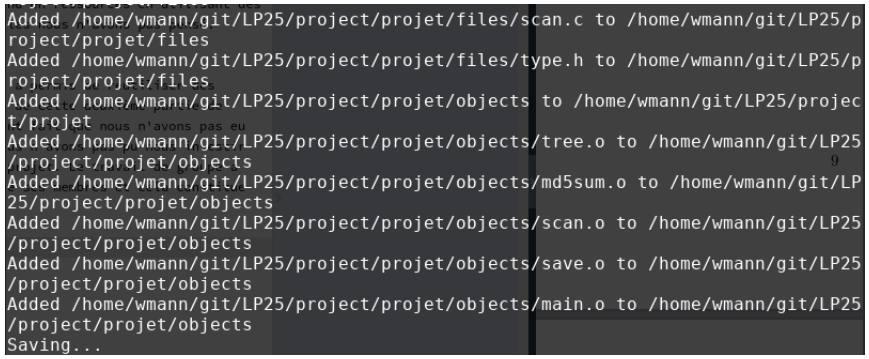
Chapitre 3

Conclusion

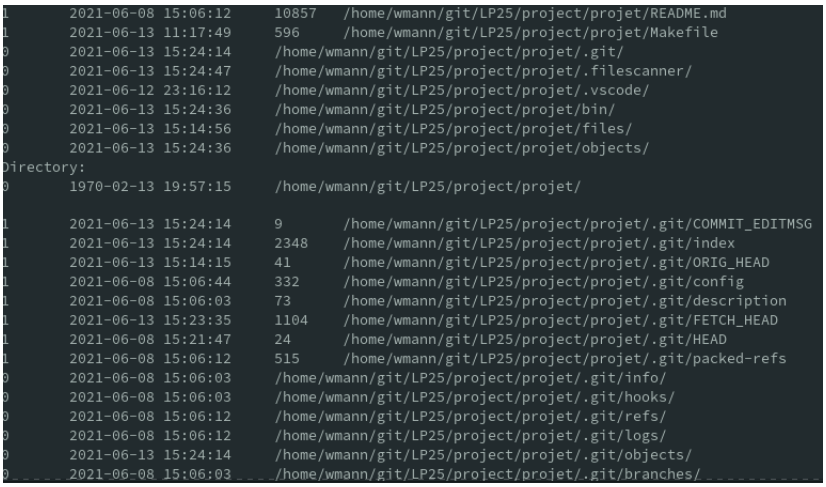
1 Résultats

Étant donné que nous avons dû réaliser ce projet en une semaine en raison du nombre de projets qui se sont accumulés et nous ont surpris, nous n'avons pas pu faire de compte rendu de synchronisation hebdomadaire et cette conclusion sur le résultat final tient donc lieu d'unique compte rendu de projet que nous avons.

Lors de l'utilisation du programme, le résultat affiché sur le terminal est :



Dans le terminal



Dans le fichier save.

On peut voir sur les captures ci-dessus que le programme se déroule jusqu'à la fin sans erreur, écrivant dans le terminal les répertoires et fichiers rajoutés au fur et à mesure. Nous avons choisi de montrer ces additions afin que l'utilisateur puisse voir le programme se dérouler et se finir avec la ligne "Saving...". Dans le cas d'arborescences complexes, cela permettra de "rassurer" l'utilisateur que le programme fonctionne comme prévu.

La seconde capture montre également que le programme sauvegarde toutes les informations demandées comme prévu (ici avec l'option de md5 non active). Les objectifs du programme sont donc réalisés, l'arborescence est comme demandée stockée dans la mémoire avant d'être intégralement écrite dans le fichier de sauvegarde demandé.

2 Améliorations possibles

L'une des améliorations possibles est notamment celle proposée dans le sujet, lier notre programme avec une base de données SQLite, cependant nous n'avons pas eu le temps d'implémenter cette fonction. Une autre amélioration serait d'optimiser notre code, car il est sûrement possible de le rendre plus rapide et moins gourmand en ressources en utilisant des fonctions/librairies différentes auxquelles nous n'avons pas pensé.

3 Conclusion

Ce projet était intéressant, car il nous a permis de réutiliser des connaissances que nous avons acquis lors de cette deuxième partie de semestre, cependant les circonstances ont fait que nous n'avons pas eu énormément de temps disponible et que nous n'avons pas pu nous investir autant que nous l'aurions voulu dans ce projet. Le travail de groupe a été bien organisé selon l'avis unanime de ses membres et cela constitue un point positif important.

Tout n'est pas exactement comme demandé, le fichier .filescanner est situé là où est le fichier main et non dans ~, mais à part cela tout est fonctionnel.