LP25 - Rapport de projet

I. Introduction

Ce projet a été réalisé dans le cadre de l'UV LP25 à l'Université de Technologie de Belfort Montbéliard, par Victor GOUBET, Arnaud KIENY et Tobias FRESARD en utilisant le langage C.

Il a pour but de réaliser un programme qui, pour un dossier renseigné en paramètre, va scanner récursivement l'entièreté de son arborescence et écrire les informations de celle-ci dans un fichier texte.

Notre rapport s'articulera en 3 axes principaux qui sont la présentation du projet en détail, notre organisation et notre RETEX.

II. Présentation du projet

Comme préconisé dans le sujet, nous avons opté pour de la compilation séparée ce qui simplifie grandement le travail en groupe car nous pouvions mieux nous répartir les tâches et limiter les conflits.

Nous avons décidé de laisser SQLite de côté, car un seul d'entre nous a suivi l'UV IFD (où on apprend SQL), et nous avons préféré nous concentrer sur la qualité du reste du projet dans le temps qui nous était accordé.

Module core

Ces fichiers contiennent les définitions et fonctions utiles à tous les autres fichiers. Nous les avons donc isolés dans des fichiers à part pour ne pas avoir à les implémenter à chaque fois. Ils contiennent notamment la définition des structure *s_file* et *s_directory* qui ont été donnés avec le sujet ; ainsi que la fonction *merge_path()* qui permet de fusionner deux *path* entre eux, sans oublier le "/" de séparation.

Module tree

Ce sont les premiers fichiers auxquels nous nous sommes attaqués car ils permettent de structurer l'arborescence grâce à laquelle tout le programme va fonctionner. De ce fait, il a fallu les retravailler au fur et à mesure de l'avancée du projet.

Au vu des structures qui nous ont été proposées (voir *core.h*), nous avons procédé avec des listes chaînées, où les fichiers ont une variable qui pointe sur le prochain fichier avec *next_file* et les dossier ont une variable pointant sur le prochain dossier et une sur le premier sous-dossier.

C'est en se basant sur ces choix de conception, que nous avons pu programmer les fonctions suivantes :

Tout d'abord, *append_file()* et *append_subdir()* vont servir à ajouter des éléments en queue de la liste chaînée dont la tête est, respectivement, pointé par *files* et par *subdirs* contenu dans *parent.*

Puis, la fonction *clear_file()* permet de libérer la mémoire allouer à tous les fichiers contenu dans la liste chaînée appropriée. Et *clear_subdir()* va quant à elle libérer la mémoire

allouer à tous les dossiers ainsi que les fichiers que *parent* contient (en appelant *clear_file()*). Et cela à travers l'arborescence de *parent* grâce à la récursivité.

Module scan

Après avoir mis en place les fichiers tree qui gèrent la manipulation de l'arborescence, nous avons pu nous pencher sur les fichiers de scan. Ceux-ci vont parcourir progressivement le dossier cible et son arborescence pour enregistrer ses informations dans les structures discutées plus haut. On y retrouve les fonctions suivantes :

- *process_file()* qui va enregistrer les différentes informations nécessaires en fonction du type du fichier dans la structure *s_file*: type de fichier, chemin, taille, somme md5 (optionnelle) et date de modification.
- *process_dir()* est plus conséquente car elle va parcourir récursivement l'arborescence du dossier et s'appeler à chaque sous-dossier rencontré. Elle appelle aussi *process_file()* pour chaque fichier contenu dans le répertoire. Elle enregistre les informations suivantes dans *s_directory* : type de fichier, chemin, date, les fichiers et sous dossiers qu'elle contient.

Module save

Dans la partie *save.c*, la fonction de base recommandées dans le sujet ont été agrémentées d'un certain nombre d'autres fonctions.
Pour accommoder avec les différents types de fichiers existants, nous avons créé différentes fonctions, à savoir *save_directory()* pour les répertoires, *save_reg_file()* pour les fichiers ordinaires, et *save_other_file()* pour les autres fichiers. Ainsi, chaque fonction permet un affichage adéquat dans le fichier contenant l'arborescence.

La fonction *default_file_name()* a été faite à part dans l'objectif de garder la fonction principale claire. Elle permet de générer le nom du fichier de sauvegarde de l'arborescence en fonction de la date et de l'heure.

Enfin, la fonction *process_save_dir()* permet de naviguer au sein de la structure obtenue précédemment, cherchant récursivement toutes les informations.

Ainsi, save_to_file crée l'endroit où l'arborescence sera sauvegardée (en utilisant *default_file_name()*), puis utilise *process_save_dir()* pour parcourir tout ce qui a été enregistré en mémoire, et les trie dans le fichier de sauvegarde en fonction du type, via *save_directory()*, *save_reg_file()* ou *save_other_file()*.

Module md5sum

Dans ces fichiers nous gérons le calcul de la somme md5 grâce à la méthode *compute_md5()*. Cette fonction sera utilisée dans scan.c afin d'y enregistrer la somme dans la structure *s_file*. Nous avons eu de nombreuses difficultés à faire fonctionner ce calcul dans tous les cas de figure, car le sujet stipulait que nous devions lire les fichiers par blocs (dans un souci de vitesse d'exécution). Le problème crucial vient donc du fait que la taille de la plupart des fichiers n'est pas un multiple de cette taille de blocs (512 octets en

l'occurrence). Ce reste de taille variable a beaucoup compliqué la tâche. Néanmoins, nous aurions pu lire les fichiers octet par octet, ce qui aurait été plus lent à l'exécution, mais bien plus rapide à coder.

Fonction main

Ce fichier contient le programme principal pour faire tourner le programme en utilisant les fonctions précédemment décrites. C'est aussi dans ce fichier que nous gérons les différents arguments optionnels au programme.

Makefile

Nous avons utilisé un Makefile générique et très flexible nous permettant de rapidement modifier les flags et d'ajouter des librairies. Ce fichier est quasiment essentiel lorsqu'on procède par compilation séparée, car sans lui, il aurait fallu compiler chaque fichier à la main et dans le bon ordre.

III. Méthodes de travail

Organisation

Nous nous sommes organisés avec des réunions régulières comme improvisées. Ces réunions ont fait l'objet de mise au point et d'état des lieux. C'est au cours de celles-ci que nous nous fixions de nouveaux objectifs et répartissions le travail en fonction des prédispositions de chacun. Nous n'avons pas hésité à partager nos problèmes de code pour rallier toutes nos connaissances et donc perdre le moins de temps possible sur le debuggage.

Logiciels utilisés

Pour communiquer entre le groupe, Discord s'est montré comme le plus propice étant donné que nous étions déjà bien familiarisé avec ce logiciel.

Concernant notre environnement de travail, nous utilisons tous Visual Studio Code car il met à disposition de nombreuses extensions facilitant le travail, tout en restant un environnement très léger (on doit toujours passer par un Makefile pour compiler, par exemple, contrairement à certains IDE qui génèrent tout automatiquement).

Pour mettre notre travail en commun, nous avons opté pour GitHub car nous l'avions tous plus utilisé auparavant que GitLab.

IV. RETEX

Compte rendu de Tobias

Ce projet a été moins conséquent que ce que j'ai pu rencontrer précédemment cependant il n'en est pas moins complexe, il aborde des thématiques où je n'ai que très peu d'expérience si l'on omet les ressources mis à disposition en LP25 qui se sont avérées très utiles. J'ai pu au cours de ce projet, et ce semestre en général, apprendre énormément de nouvelles notions concernant la programmation bas niveau mais aussi le fonctionnement de nos ordinateurs.

Avec le recul, je pense que je prendrai plus le temps de comprendre le sujet et ce qui nous est demandé avant de commencer à programmer mes premières lignes. Cette erreur nous a fait perdre un temps non négligeable lorsqu'il a fallu fusionner le code de chacun.

Compte rendu d'Arnaud

Le projet en tant que tout semblait un peu intimidant pour un débutant, mais le diviser en partie l'a rendu bien plus abordable. Avec ce sujet, j'ai pu appliquer ce que j'ai appris lors de ce semestre, et même approfondir un peu certaines notions.

J'en ai surtout appris à mieux travailler en groupe et éviter les erreurs de compilation à cause de codes incomplets. C'est quelque chose dont je n'ai pas encore l'habitude mais qui sera de plus en plus important.

Compte rendu de Victor

Au cours de ce projet, j'ai redécouvert l'importance d'un code clair et commenté, lorsque j'ai dû comprendre le code de mes camarades, ce qui était certainement réciproque.

Si c'était à refaire, je ne réaliserai pas ce projet en C, mais dans un langage bien plus moderne (tout en restant rapide *et* de bas niveau), par exemple Rust. En effet, mettre en place l'environnement de travail en C demande beaucoup de temps au début du projet (pour automatiser la compilation et le lancement du debugger). De plus, trop de fonctionnalités manquent cruellement, comme les chaînes de caractères.

Cela dit, c'est un projet où tous les membres du groupe ont bien contribué à l'avancement: notre coordination était bonne. Grâce à mes camarades, j'ai pu acquérir davantage d'expérience vis-à-vis du travail en groupe.

V. Conclusion

Ce projet a été un exercice complet, mettant en pratique une grande partie de nos acquis durant ce semestre, en plus de nous permettre de développer nos capacités à travailler en groupe.

Ayant atteint les objectifs principaux, nous sommes majoritairement fiers de notre programme bien que nous aurions aimé ajouter d'autres fonctionnalités comme l'implémentation de SQLite.

Nous avons fait preuve d'un bon esprit de groupe, en communiquant régulièrement sur l'avancée de notre projet et en répartissant le travail convenablement. Cela nous a permis de travailler dans de bonnes conditions.