PROGRAMME DE SCAN RÉCURSIF DE RÉPERTOIRE SOUS LINUX

→ Dans le cadre de l'UV LP25 au semestre de printemps 2021

utbm
université de technologie
Belfort-Montbéliard

SOMMAIRE

I) Introduction

1/ Contexte

Nous avons, au cours de ce semestre de printemps, découvert l'étendue et les nombreuses possibilités de gestion et de stockage de l'information sous un système Unix. En effet, nous avons appris à maîtriser un système Linux, à le comprendre et à l'exploiter. Ce grâce à la programmation en C. Ainsi nous avons appris et mis en pratique nos connaissances afin de programmer ce système d'analyse de répertoire.

Afin d'avancer efficacement sur le projet, nous avons décidé de diviser les diverses parties du programme et de les attribuer à chacun.Par la suite nous avons organisé des réunions journalières afin de statuer sur l'avancement de chaque partie, de s'entraider les uns et les autres sur les diverses difficultés rencontrées et de progresser rapidement sur le projet.

2/ Objectifs

Ce projet a pour but d'écrire un programme scannant récursivement un répertoire, en extrayant son arborescence complète. Cette arborescence doit être stockée dans la mémoire de l'ordinateur lors de l'exécution du programme, puis ce contenu, une fois trié, devra être sauvegardé dans un fichier. Les informations récoltées et stockées divergent s'il s'agit d'un fichier ou d'un répertoire mais dans l'ensemble le programme doit retenir les informations suivantes :

- le type de fichier/répertoire,
- la date de modification,
- la somme MD5, si demandée (uniquement pour les fichiers réguliers),
- la taille du fichier,
- le nom du fichier/répertoire.

Les objectifs comprennent également d'utiliser l'allocation dynamique de la mémoire tout au long du projet ainsi que de gérer la manipulation des fichiers et répertoires. Par manque de temps et de compétences, nous avons décidé de ne pas inclure la partie bonus concernant la sauvegarde de l'arborescence dans une base de donnée SQLite.

II) Retour d'expérience & Analyse des fonctions

1/ Fichier main.c

A) Fonctionnalités implémentées

Ce fichier contient une unique fonction *int main(int argc, char *argv[])* qui a divers objectifs: il s'agit à la fois de traiter les arguments donnés au programme et de vérifier leur validité et d'adapter le comportement du programme selon ceux-ci, puis d'appeler les autres fonctions permettant l'exécution complète du programme.

Pour traiter les arguments passés au programme nous avons recours à la fonction getopt, qui permet de gérer:

- Un argument optionnel: -s, permettant d'activer le calcul des sommes md5 des fichiers.
- Deux arguments obligatoires:
  - -i, permettant de préciser le répertoire à scanner.
  - -o, permettant de préciser le fichier dans lequel effectuer cette sauvegarde

Si ces arguments ne sont pas précisés, alors:

- Il y a création d'un répertoire nommé *.filescanner* (s'il n'existe pas déjà) , suivi de la création d'un fichier dans *.filescanner* portant la date précise de l'exécution du programme.
- Le répertoire à scanner est défini sur le répertoire depuis lequel a été lancé le programme: "."

Ensuite on appelle les fonctions *process_dir* et **save_to_file** (expliquées plus bas) avec les trois paramètres cités ci dessus.

B) Résultats obtenus

Les arguments donnés au programme sont bien traités et vérifiés en conséquence. Nous avons également rajouté l'affichage d'une aide sur l'utilisation des arguments lorsqu'un argument inconnu est transmis au programme. Nous avons eu un problème au niveau des droits d'accès au répertoire *.filescanner*, à cause d'un oubli de chiffre dans la fonction *mkdir()*. Cependant nous avons trouvé cette partie assez facile à implémenter.

2/ Fichier scan.c

A) Fonctionnalités implémentées

La fonction *process_dir* permet de scanner un répertoire et ses éléments.

- Tout d'abord on déclare une structure de description du répertoire (*s_directory*) que nous initialisons.
- Puis nous exécutons une boucle qui permet de lire un élément après l'autre à l'intérieur du répertoire.
- Sur chaque élément nous appelons la fonction de *scan* suivant sont type (fichier ou répertoire).
- En retour des fonctions de *scan* il y a la structure de description de l'élément, ce qui permet de connaître la structure des éléments enfants répertoire scanné.

Avec cette information nous pouvons ainsi compléter la structure de description du répertoire pour qu'il puisse pointer sur ces éléments.
La fonction *process_file* permet de scanner un fichier.

- Tout d'abord on déclare une structure de description du fichier (*s_file*) que nous initialisons.
- Puis nous cherchons si c'est un fichier régulier.
- Si c'est le cas nous complétons la structure de description suivant le cahier des charges.
- De même si c'est un fichier d'un autre type.
- Enfin cette fonction retourne la structure de description qui sera complété dans la fonction *process_dir* pour qu'il puisse pointer sur le prochain fichier.

La fonction *recuperation_nom* permet d'extraire le nom principal du fichier ou du répertoire du chemin complet.

- Une première boucle parcours la chaîne de caractère du chemin à partir de la fin.
- Elle enregistre la position du premier "/".
- Puis une deuxième boucle copie caractère par caractère dans une nouvelle chaîne à partir de la position du premier "/".
- Enfin elle retourne le nom principal.

B) Résultats obtenus

Grâce à ces trois fonctions, en appelant la fonction *process_dir* avec un chemin vers un répertoire, récursivement chaque élément va être scanné et stocké dans une liste chaînée. Une des erreurs les plus instructives lors de l'implémentation de cette partie était d'avoir oublié d'initialiser certaines structures. Nous nous sommes pas tout de suite rendu compte de l'erreur car en essayant le programme sur un petit dossier, ces structures étaient initialiser sur NULL comme voulu, par pure coïncidence.

Puis lors de notre test sur de plus gros répertoires, nous avons eu une erreur de segmentation. Comme cela faisait un moment que nous avions développé cette fonction, nous avons cherché notre erreur sur les derniers ajouts.

Nous avons finalement trouvé l'erreur après un temps assez long pour une erreur de ce genre. Cela nous permettra de faire attention et de penser à cette erreur lors des prochains projets.

3/ Fichier md5sum.c

A) Fonctionnalités implémentées

Ces fichiers comportent une seule fonction, *int compute_md5(char *path, unsigned char buffer[])* Elle prend en entrée un chemin vers un fichier ainsi qu'un buffer. A la fin de la fonction, le buffer contient la somme md5 du fichier. Pour ce faire, notre fonction se base sur l'utilisation de la fonction *FILE *popen(const char *command, const char *type)* qui permet d'inscrire dans un fichier la sortie d'une commande shell.

- Dans un premier temps nous concaténons une chaîne de caractères afin d'obtenir la commande à exécuter (sous la forme "md5sum "nom_du_fichier"").
- Puis nous ouvrons un fichier dans lequel nous écrivons le résultat de la commande
- Enfin nous lisons les 32 premiers caractères de ce fichier (contenant la somme md5), qui sont par la suite copiés dans le buffer.

B) Résultats obtenus

L'avantage de cette façon de procéder est qu'elle ne nécessite pas l'utilisation d'une librairie annexe, et de plus elle est relativement courte et facilement compréhensible. Néanmoins cela rend la fonctionnalité plus fragile car dépendante du système d'exploitation sur lequel est lancé le programme. Nous en avons fait les frais car au début nous avions omis de mettre des guillemets autour du nom du fichier. Cela aurait pu provoquer des erreurs (si par exemple le programme tombait sur un fichier nommé: "fichier ; rm -rvf ./" ou bien un fichier contenant des espaces).

Dans la documentation officielle, la somme md5 est codée sur 128bits, et la commande md5sum renvoie 32 caractères hexadécimal. Cependant la structure imposée contenait uniquement 16 caractères pour la somme md5, ce qui nous a laissé perplexe.

4/ Fichier save.c

Le fichier save comporte trois fonctions :

- la fonction *save_to_file*, qui prend en paramètre un répertoire racine, un chemin d'accès au fichier de sauvegarde, un booléen pour effectuer ou non, la somme MD5. Cette fonction prend aussi deux paramètres utilisés pour sa récurrence, à savoir: le nombre d'indentation ainsi que le chemin du répertoire actuellement sauvegardé.
- la fonction *string_builder_of_dir* se charge de construire une chaîne de caractères contenant les informations du répertoire passé en paramètre.
- la fonction *string_builder_of_file* qui n'est que la version pour fichier de la fonction précédente.

A) Fonctionnalités implémentées

La sauvegarde de l'arborescence du répertoire scanné se fait de façon récursive et incrémentale. En effet, à chaque sous-répertoire détecté, la sauvegarde ajoute une tabulation au début de la ligne. Afin de respecter les consignes, les répertoires et fichiers sont différenciés par leur numéro *e_type*:

- 0 pour un répertoire,
- 1 pour un fichier normal
- 2 pour tout autre fichier (notamment les liens symboliques)

De plus, pour chaque fichier/répertoire est ajouté leur date de modification en utilisant la fonction strftime() et on retrouve le chemin d'accès du fichier/répertoire en fin de ligne. Lorsqu'il s'agit d'un fichier, il peut être demandé, via le booléen doMD5, d'inscrire la somme MD5 dans la chaîne de caractères en plus de la taille du fichier.

B) Résultats obtenus

Les résultats obtenus sont probants et permettent de comprendre facilement l'arborescence complète d'un répertoire :
Voici le résultat obtenu lorsqu'on effectue la commande suivante : *./projet.exe -s -i dirtest*

```
0   2021-06-15-11:05:23 dirtest/
   1   2021-06-13-21:41:15 166985  0c5d4c056d1d6a1 dirtest/source
   1   2021-06-13-21:41:45 5117    daff437b8ee9da9 dirtest/t1.txt
   1   2021-06-14-16:29:42 5983    31d71342e145b8b dirtest/t2
   1   2021-06-14-16:29:42 5983    31d71342e145b8b dirtest/t2_2
   1   2021-06-14-16:29:42 5983    31d71342e145b8b dirtest/t2_3
   1   2021-06-14-16:29:42 5983    31d71342e145b8b dirtest/t2_3 (copie 2)
   2   2021-06-14-17:18:37 dirtest/l-t2
   2   2021-06-14-17:19:34 dirtest/l-dest
   1   2021-06-15-11:05:23 5983    31d71342e145b8b dirtest/;;eefôej
   0   2021-06-11-14:33:55 dirtest/d2/
      1   2021-06-13-21:40:12 5   3ed7ebb03bd052a dirtest/d2/t4
   0   2021-06-14-16:29:42 dirtest/d_1/
      1   2021-06-13-21:39:12 10  e9d86d6e1ca8768 dirtest/d_1/t3.txt
      0   2021-06-11-14:33:55 dirtest/d_1/d_3/
         1   2021-06-13-21:37:37 14  a391a4971d7511e dirtest/d_1/d_3/t5.txt
         1   2021-06-13-21:38:53 10  1bc45205f529013 dirtest/d_1/d_3/t6
   0   2021-06-14-16:29:42 dirtest/dest/
      1   2021-06-13-21:40:35 11  df2364e4e0a7086 dirtest/dest/t1.txt
      1   2021-06-13-21:40:46 7   5c952ae695a4af3 dirtest/dest/t2
```

On remarque ainsi que la présence de lien symbolique n'empêche pas le bon fonctionnement du programme.

5/ Fichiers tree

A) Fonctionnalités implémentées

La fonction *clear_subdirs* permet de supprimer toute la liste chaînée stockée en mémoire.

- Vérifie si le répertoire pointe vers un sous-répertoire ou un fichier ou un répertoire du même niveau.
- Pour chaque cas il appelle la fonction de destruction appropriée : *clear_subdirs* pour les répertoires et *clear_files* pour les fichiers.
- Enfin la structure de description est libérée.

La fonction *clear_files* permet de supprimer les fichiers stockée en mémoire. En voici la procédure:

- Vérifie si le fichier pointe vers un fichier du même niveau.
- La fonction *clear_files* est exécuté sur le fichier du même niveau.
- Enfin la structure de description est libérée.

La fonction *append_subdir* permet d'ajouter un sous-répertoire à un répertoire parent. L'algorithme cherche si le répertoire parent possède déjà ou non un sous-répertoire, si tel est le cas, alors, la fonction attache le sous-répertoire au dernier sous-répertoire présent.

La fonction *append_file* possède le même fonctionnement que son homologue *append_subdir* mais attache un fichier à un répertoire, ou bien lorsqu'un fichier est déjà attaché, a celui-ci.

B) Résultats obtenus

Les 2 fonctions *clear_files* et *clear_subdirs* sont essentielles car elle permettent de terminer de vider l'espace alloué par le programme, il est donc impératif de les exécuter à la fin du programme afin de garantir sa fiabilité.

En revanche, les deux fonctions d'ajout ne sont appelées qu'une fois dans le programme et semblent superflues, pourtant elles permettent de détacher la partie ajout de la partie scan du programme et permettent une meilleur accessibilité du code source.

III) Conclusion

Ce projet nous a permis d'apprendre des méthodes concrètes de programmation telles que les liste chaînées ou la gestion de fichier. Nous avons ainsi pu découvrir l'intérêt de celles-ci pour construire un ensemble d'informations de manière récursive, rendant cette méthode puissante par son évolution facile et "automatique".

Nous avons également pu affiner nos connaissances en gestion de mémoire, notamment dynamique, en nous confrontant aux divers problèmes rencontrés et surmontés lors de ce projet. De plus malgré la présence de certains bugs, nous avons réussi à les corriger et ainsi rendre notre programme fonctionnel.

Pour finir, il est important de remarquer que malgré la crise sanitaire actuelle, la cohésion et le travail de notre équipe nous ont permis d'avancer rapidement et de manière concrète sur le projet.