Projet: Réalisation d'un logiciel de gestion de versions

Nous considérons dans ce projet la réalisation d'un outil de suivi et de versionnage de code (type git). Ce projet est découpée en plusieurs parties, qui seront ajoutées progressivement au sujet pendant le semestre. Il faudra donc télécharger le sujet du projet à chaque début de séance de TME (pour avoir accès à la suite du projet). Chaque partie est divisée en exercices, qui vont vous permettre de concevoir progressivement le programme final. Il est impératif de travailler régulièrement pendant le semestre, afin de ne pas prendre de retard et de pouvoir profiter des séances de TME associées à chacune des parties du projet.

Cadre du projet

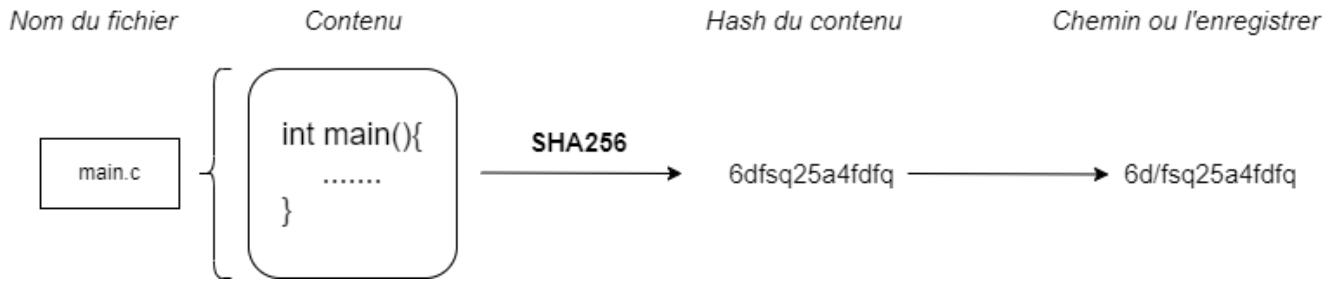
Un logiciel de gestion de versions est un outil permettant le stockage, le suivi et la gestion de plusieurs versions d'un projet (ou d'un ensemble de fichiers). En particulier, ces outils offrent un accès aisé à l'historique de toutes les modifications effectuées sur les fichiers, permettant notamment de récupérer une version antérieure en cas de problème. Par ailleurs, ces outils sont très utiles dans le cadre de travail collabaratif, permettant de fusionner de manière intelligente différentes versions d'un même projet. Par exemple, ces outils sont couramment utilisés en développement logiciel, pour faciliter le travail en équipe et conserver le code source relatifs à différentes versions d'un même logiciel.

L'objectif de ce projet est d'étudier le fonctionnement d'un logiciel de gestion de versions, en détaillant différentes structures de données impliquées dans sa mise en oeuvre. En particulier, nous allons nous intéresser aux fonctionnalités suivantes :

- Comment permettre à un utilisateur de créer des enregistrements instantanés de son projet ?
- Comment lui permettre de naviguer librement à travers les différents instantanés ?
- Comment construire et maintenir une arborescence des différentes versions de son projet ?
- Comment vérifier l'identité des utilisateurs ?
- Comment sauvegarder des changements qui ne sont pas dans un instantané ?

Vers la création d'enregistrements instantanés

Sous git, tout les objets, qu'ils soient relatifs aux fichiers versionnés ou à leurs méta-données, sont enregistrés sous forme de fichiers. Ces fichiers ont pour particularité de pouvoir dériver le chemin où ils sont stockés à partir de leur contenu, par exemple comme décrit dans la figure suivante :



Ici la fonction de hachage SHA256 est appliquée sur le contenu du fichier, puis le chemin où doit être stocké le fichier est obtenu en insérant un "/" entre le deuxième et le troisième caractères du hash. Faire dépendre le chemin du contenu permet notamment de sauvegarder toutes les différentes versions du fichier. En effet, modifier le contenu du fichier va modifier le chemin vers lequel le sauvegarder et ainsi créer différentes sauvegardes correspondant à différents états du fichier. Quand on réalise une telle sauvegarde, on dit communément qu'on "enregistre un instantané" ou encore qu'on crée un "enregistrement instantané". L'objectif de cette première partie du projet est d'écrire un programme permettant d'enregistrer un instantané, comme décrit dans l'exemple.

Exercice 1 – Prise en main du langage Bash

Les données sur le disque sont stockées dans des fichiers (file en anglais), qui sont des structures de données qui apparaissent aux programmes comme des suites finies d'octets. La structure de données qui organise les fichiers sur le disque s'appelle le système de fichiers (file system). Le système de fichiers est l'une des fonctionnalités principales d'un système d'exploitation. Si généralement on y a recours en utilisant une interface graphique, nous apprendrons ici à le contrôler par du code. Pour cela, nous allons commencer par quelques exercices de prise en main du langage Bash permettant d'utiliser une interface en ligne de commande.

Q 1.1 Pour vous familiariser avec les commandes usuelles, commencez par suivre la séquence de d'instructions suivantes, et utiliser à chaque fois la commande `man` pour afficher la documentation de la commande concernée :

- Ouvrez un terminal (sous linux, cela positionne par défaut sur le chemin principal "∼/").
- Utilisez la commande `pwd` pour afficher le répertoire courant.
- Utilisez la commande `cd` pour vous positionner sur votre répertoire de travail pour cette UE.
- Utilisez la commande `mkdir` pour créer un répertoire nommé "projet_scv".
- Positionnez vous dans le répertoire nouvellement créé, puis utilisez la commande `touch` pour créer les fichiers "main.c", "Makefile" et "test.txt".

— Utilisez `ls` pour lister les fichiers de votre répertoire.
— Après avoir modifié le fichier "main.c" via votre éditeur de texte préféré, utilisez la commande `cat` pour en afficher le contenu.
— Utilisez la commande `rm` pour supprimer le fichier "test.txt".

Linux met en place des flux globaux permettant à des commandes appelées, de dialoguer avec le programme appelant, en écrivant ou en lisant depuis ces flux. Ces flux globaux sont :
— `stdin` (entrée standard) : flux d'entrée du programme. Par défaut, il s'agit des données saisies au clavier. Ce flux permet notamment de définir des programmes interactifs, récupérant des données saisies sur le terminal par l'utilisateur, par le biais de la fonction `scanf` (par exemple).
— `stdout` (sortie standard) : ce flux correspond à la sortie du programme. Par défaut, il s'agit du terminal ayant lancé le programme. Il permet notamment d'afficher des données sur le terminal.
— `stderr` (sortie standard d'erreur) : ce flux sert à récupérer les messages d'erreurs, qui par défaut sont affichés sur le terminal qui a lancé le programme.

Sous linux, il est possible, après avoir appelée une commande, de rediriger une des sorties vers un fichier. Par exemple, avec la commande `ls > list.txt`, vous obtenez un fichier "list.txt" contenant la liste des fichiers et répertoires présents dans le répertoire courant. Il est également possible de rediriger la sortie d'une commande vers une autre, en utilisant ce que l'on appelle une "pipeline". Par exemple, la commande `cat noms.txt | sort` permet de lire la liste de noms présente dans le fichier noms.txt, et de la transmettre à la fonction sort qui va la trier (par ordre alphabétique).

Q 1.2 Sous linux, la commande `sha256sum` permet de hacher le contenu d'un fichier en utilisant la fonction de hachage SHA256. En utilisant une redirection et une pipeline, écrivez une commande qui transmets le contenu du fichier "main.c" à la commande `sha256sum` puis écrit le hash correspondant dans un fichier (temporaire) appelé "file.tmp".
Ces commandes peuvent aussi être utilisées à travers un code C, par le biais de la fonction `system` qui, comme son nom l'indique, permet de faire des appels système. Pour bien comprendre le fonctionnement, commencez par exécuter le programme C suivant :

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    system("ls");
}
```

Q 1.3 Écrivez une fonction `int hashFile(char* source, char* dest)` qui, étant donné le chemin de deux fichiers, calcule le hash du contenu du premier fichier et l'écrit dans le deuxième fichier.
Bien que la fonction `system` permette d'exécuter des commande Bash, ce n'est pas le seul moyen de manipuler le système de fichiers. Un autre moyen, qui est préférable quand il est disponible, est d'utiliser des librairies C prévues à cet effet. Par exemple, on préférera ce code :

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    FILE* f = fopen("test.txt", "w");
    fprintf(f, "Test");
}
```

à son équivalent suivant :

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    system("echo Test $>$ test.txt");
}
```

En particulier, la librairie `<unistd.h>` permet une gestion efficace des fichiers temporaires, ce qui nous sera très utile pour stocker le hash d'un fichier en attente de lecture par notre programme. Voici un exemple de code illustrant comment créer un fichier temporaire avec la fonction `mkstemp` de cette librairie :

```c
static char template[] = "/tmp/myfileXXXXXX";
char fname[1000];
strcpy(fname, template);
int fd = mkstemp(fname);
```

Dans cet exemple, l'appel à `mkstemp(fname)` crée un fichier temporaire dont le nom sera stocké dans `fname`. Ce nom sera généré de manière unique à partir du motif `template`. Plus précisément, il faut que les six derniers caractères de `template` soient "XXXXXX" pour que la fonction `mkstemp` les remplacent de sorte à créer un nom de fichier unique. La fonction `mkstemp` ouvre ensuite le fichier temporaire nouvellement créé, et renvoie un descripteur de fichier ouvert (en lecture et écriture).

Q 1.4 En utilisant la commande `sha256sum`, un pipe, une redirection, et un fichier temporaire (qu'il faudra supprimé après usage), écrivez une fonction `char* sha256file(char* file)` qui renvoie une chaîne de caractères contenant le hash du fichier donné en paramètre.

Exercice 2 – Implémentation d'une liste de chaînes de caractères

Dans cet exercice, il s'agira d'implémenter les fonctions permettant de gérer une structure de données de type liste chaînée dont la définition est la suivante :

```c
typedef struct cell {
   char* data;
   struct cell* next;
} Cell;

typedef Cell* List;
```

Q 2.1 Écrivez une fonction `List* initList()` qui initialise une liste vide. Veillez par la suite à ne plus initialiser de liste autrement que par cette fonction.

Q 2.2 Écrivez une fonction `Cell* buildCell(char* ch)` permettant d'allouer et de retourner une cellule de la liste.

Q 2.3 Écrivez une fonction `void insertFirst(List *L, Cell* C)` permettant d'ajouter un élément en tête d'une liste.

Q 2.4 Écrivez une fonction `char* ctos(Cell* c)` qui retourne la chaîne de caractères qu'elle représente, puis utilisez cette fonction pour écrire la fonction `char* ltos(List* L)` qui transforme une liste en une chaîne de caractères avec le format suivant : chaîne1|chaîne2|chaîne3|...

Q 2.5 Écrivez une fonction `Cell* listGet(List* L, int i)` qui renvoie le $i^{\text{ème}}$ élément d'une liste.

Q 2.6 Écrivez une fonction `Cell* searchList(List* L, char* str)` qui recherche un élément dans une liste à partir de son contenu et renvoie une référence vers lui ou NULL s'il n'est pas dans la liste.

Q 2.7 Écrivez une fonction `List* stol(char* s)` qui permet de transformer une chaîne de caractères représentant une liste en une liste chaînée.

Q 2.8 Écrivez la fonction `void ltof(List* L, char* path)` permettant d'écrire une liste dans un fichier, et la fonction `List* ftol(char* path)` permettant de lire une liste enregistrée dans un fichier.

Exercice 3 – Gestion de fichiers sous git

L'objectif final de cet exercice est de produire une fonction qui enregistre un instantané d'un fichier dont le nom est donné en paramètre.

Q 3.1 Les fonctions `opendir` et `readdir` de la librairie `<dirent.h>` permettent d'explorer un répertoire. Par exemple, on peut afficher le noms des fichiers et des répertoires qui le composent en procédant de la manière suivante :

```
1 DIR *  dp = opendir (root_dir);
2 struct dirent *ep;
3 if (dp != NULL)
4 {
5     while ((ep = readdir (dp)) != NULL)
6     {
7         printf("\%s␣\n", ep->d_name);
8     }
9 }
```

Écrivez une fonction `List* listdir(char* root_dir)` qui prend en paramètre une adresse et renvoie une liste contenant le noms des fichiers et répertoires qui s'y trouvent.

Q 3.2 Utilisez la question précédente pour écrire une fonction `int file_exists(char *file)` qui retourne 1 si le fichier existe dans le répertoire courant et 0 sinon.

Q 3.3 Écrivez une fonction `void cp(char *to, char *from)` qui copie le contenu d'un fichier vers un autre, en faisant une lecture ligne par ligne du fichier source.
Indication : pensez à vérifier que le fichier source existe avant.

Q 3.4 Écrivez une fonction `char* hashToPath(char* hash)` qui retourne le chemin d'un fichier à partir de son hash (on rappelle que le chemin s'obtient en insérant un "/" entre le deuxième et le troisième caractères du hash).

Q 3.5 En utilisant la commande Bash `mkdir` (qui permet de créer un répertoire), écrivez une fonction `void blobFile(char* file)` qui enregistre un instantané du fichier donné en entrée.

Enregistrement de plusieurs instantanés

Dans la partie précédente, nous avons vu comment enregistrer un instantané d'un fichier. Dans un projet, on est souvent amené à manipuler un ensemble de fichiers, structurés en arborescence (avec des répertoires). Dans ce cas, on peut vouloir enregistrer un instantané de plusieurs fichiers et/ou répertoires du projet. Pour ce faire, nous allons travailler avec la structure suivante :

```
typedef struct {
    char* name;
    char* hash;
    int mode;
} WorkFile;

typedef struct {
    WorkFile* tab;
    int size;
    int n;
} WorkTree;
```

Un `Workfile` représente un fichier ou répertoire dont on souhaite enregistrer un instantané. Il possède trois champs :

- `name` correspond au nom du fichier ou du répertoire.
- `hash` correspond au hash associé à son contenu, et est initialisé à NULL.
- `mode` donne les autorisations associés au fichier (modification, lecture et exécution) et est initialisé à zéro.

Un `WorkTree` est simplement un tableau de `Workfile`. Les autorisations associés à un fichier décrivent qui peut le modifier, le lire et l'exécuter, et sont représentées par un nombre de 3 digits en octal (par exemple 777 donne à tout le monde le droit de lire, écrire et exécuter le fichier). Ces autorisations peuvent être récupérées en utilisant la fonction suivante qui nous sera utile dans le deuxième exercice de cette partie (c-à-d dans l'exercice 5) :

```
int getChmod(const char *path){
    struct stat ret;

    if (stat(path, &ret) == -1) {
        return -1;
    }

    return (ret.st_mode & S_IRUSR)|(ret.st_mode & S_IWUSR)|(ret.st_mode & S_IXUSR)|/*owner*/
        (ret.st_mode & S_IRGRP)|(ret.st_mode & S_IWGRP)|(ret.st_mode & S_IXGRP)|/*group*/
        (ret.st_mode & S_IROTH)|(ret.st_mode & S_IWOTH)|(ret.st_mode & S_IXOTH);/*other*/
}
```

Pour modifier les autorisations, on pourra utiliser la fonction suivante :

```
void setMode(int mode, char* path){
    char buff[100];
    sprintf(buff, "chmod %d %s", mode, path);
    system(buff);
}
```

Le but de cette partie est de pouvoir créer un enregistrement instantané d'un WorkTree et de son contenu, puis de permettre la restauration de cet ensemble de fichiers comme décrit dans ces enregistrements instantanés.

Exercice 4 – Fonctions de manipulation de base

Dans cet exercice, il s'agit d'écrire les fonctions de manipulation de base.

MANIPULATION DE WORKFILE

Q 4.1 Écrivez une fonction `WorkFile* createWorkFile(char* name)` qui permet de créer un `WorkFile` et de l'initialiser.

Q 4.2 Écrivez une fonction `char* wfts(WorkFile* wf)` qui permet de convertir un `WorkFile` en chaîne de caractères contenant les différents champs séparés par des tabulations (caractère '\t').

Q 4.3 Écrivez une fonction `WorkFile* stwf(char* ch)` qui permet de convertir une chaîne de caractères représentant un `WorkFile` en un `WorkFile`.

MANIPULATION DE WORKTREE

Q 4.4 Écrivez une fonction `WorkTree* initWorkTree()` permettant d'allouer un `WorkTree` de taille fixée (donnée par une constante du programme) et de l'initialiser.

Q 4.5 Écrivez une fonction `int inWorkTree(WorkTree* wt, char* name)` qui vérifie la présence d'un fichier ou répertoire dans un `WorkTree`. Cette fonction doit retourner la position du fichier dans le tableau s'il est présent, et -1 sinon.

Q 4.6 Écrivez une fonction `int appendWorkTree(WorkTree* wt, char* name, char* hash, int mode)` qui ajoute un fichier ou répertoire au `WorkTree` (s'il n'existe pas déjà).

Q 4.7 Écrivez une fonction `char* wtts(WorkTree* wt)` qui convertit un `WorkTree` en une chaîne de caractères composée des représentations des `WorkFile` séparées par un saut de ligne (caractère '\n').

Q 4.8 Écrivez une fonction qui convertit une chaîne de caractères représentant un `WorkTree` en un `WorkTree`.

Q 4.9 Écrivez une fonction `int wttf(WorkTree* wt, char* file)` qui écrit dans le fichier `file` la chaîne de caractères représentant un `WorkTree`.

Q 4.10 Écrivez une fonction `WorkTree* ftwt(char* file)` qui construit un `WorkTree` à partir d'un fichier qui contient sa représentation en chaîne de caractères.

Exercice 5 – Enregistrement instantané et restauration d'un WorkTree

Réaliser un enregistrement instantané d'un `WorkTree` revient à créer un enregistrement instantané du fichier qui le représente. Néanmoins, pour pouvoir ensuite le distinguer d'un fichier classique, on ajoutera l'extension ".t" au nom de son enregistrement instantané. Par exemple, si le hash du fichier représentant le `WorkTree` est "dsfsd245azd", alors l'enregistrement instantané est dans "ds/fsd245azd.t".

Q 5.1 Écrivez une fonction `char* blobWorkTree(WorkTree* wt)` qui crée un fichier temporaire représentant le `WorkTree` pour pouvoir ensuite créer l'enregistrement instantané du `WorkTree` (avec l'extension ".t"). Cette fonction devra retourner le hash du fichier temporaire.

On s'intéresse maintenant à une fonction `char* saveWorkTree(WorkTree* wt, char* path)` qui, étant donné un `WorkFile` dont le chemin est donné en paramètre, crée un enregistrement instantané de tout son contenu (de manière récursive), puis de lui même. Plus précisément, la fonction parcourt le tableau de `WorkFile` de `wt`, et pour chaque `WorkFile WF`, elle réalise le traitement suivant :

- Si `WF` correspond à un fichier, alors la fonction `blobFile` est utilisée pour créer un enregistrement instantané de ce fichier, puis on récupère le hash du fichier et son mode (avec la fonction `getChmod`) pour le sauvegarder dans `WF`.
- Si `WF` correspond à répertoire, la fonction doit réaliser un appel récursif sur ce répertoire. Pour cela, il faudra créer un nouveau `WorkTree newWT` représentant tout le contenu de ce répertoire (fonction `listdir`), réaliser un appel récursif sur `newWT`, puis récupérer son hash et son mode pour le sauvegarder dans `WF`.

Après avoir traité tout le tableau de `wt`, la fonction `saveWorkTree` se termine en appelant `blobWorkTree` sur `wt` pour créer son enregistrement instantané et retourner son hash.

Q 5.2 Écrivez la fonction `char* saveWorkTree(WorkTree* wt, char* path)`.

L'appel à `saveWorkTree` permet de conserver une sauvegarde de l'état de plusieurs fichiers à un instant donné. Si entre temps, les fichiers on été modifiés et que l'on souhaite revenir en arrière, il convient d'avoir une fonction récursive qui restaure un `WorkTree`, c'est-à-dire qui recrée l'arborescence des fichiers comme décrit par ses enregistrements instantanés. Pour restaurer un `WorkTree`, on va définir une fonction `void restoreWorkTree(WorkTree* wt, char* path)`, qui parcourt le tableau de `wt`, en réalisant pour chacun de ses `WorkFile WF` le traitement suivant :

- Trouver l'enregistrement instantané correspondant au hash de `WF`.
- Si l'enregistrement ne possède pas l'extenstion ".t", il s'agit d'un fichier. Dans ce cas, on doit créer une copie de l'enregistrement à l'endroit indiqué par la variable `path`, et lui donner le nom et les autoristations correspondans aux champs `name` et `mode`.
- Si l'enregistrement possède l'extension ".t", alors il s'agit d'un répertoire. Dans ce cas, il faut créer le `WorkTree` associé, modifier la variable `path` en y ajoutant ce répertoire à la fin, puis faire un appel récursif sur ce nouveau `WorkTree`.

Q 5.3 Écrivez la fonction `void restoreWorkTree(WorkTree* wt, char* path)`.
Conseil : pensez à sauvegarder votre code dans un autre répertoire avant de tester cette fonction, au risque de supprimer tout votre projet !