Compte rendu projet LU2IN006 : Réalisation d'un logiciel de gestion de versions

CAMPAN Ina, KACI CHAOUCHE Sarah

Semestre 4

Table de matières

1 Introduction et organisation du projet **3**
1.1 Contexte 3
1.2 Organisation 3

2 Vers la création d'enregistrements instantanés **4**
2.1 Prise en main du langage Bash : appels systèmes et fonctions de hachage 4
2.2 Implémentation d'une liste de chaînes de caractères 4
2.3 Gestion de fichiers sous git 5

3 Enregistrement de plusieurs instantanés **7**
3.1 Fonctions de manipulation de base 7
3.2 Enregistrement instantané et restoration d'un WorkTree . . . 8
3.3 Remarques et observations 8

4 Gestion de commits **9**
4.1 Fonctions de base pour les commits 9
4.2 Gestion temporelle des commits de manière linéaire 10
4.3 Manipulation de références 10
4.4 Simulation de la commande GitAdd 10
4.5 Simulation de la commande GitCommit 10

5 Gestion d'une timeline arborescente **12**
5.1 Fonctions de base de manipulation des branches 12
5.2 Git checkout-branch 12
5.3 Git checkout-commit 12

6 Gestion des fusions de branches **14**

7 Tests par semaine, myGit et observations **15**

8 Conclusion **16**

1 Introduction et organisation du projet

1.1 Contexte

Il est ici question d'exploiter diffèrentes structures de données utilisables en langage C pour découvrir le versionnage et suivi de code (comme pour l'outil Git). Ce processus se fait en plusieurs étapes simplifiées à cet effet.

1.2 Organisation

Jumelé avec nos fichiers C contenant nos algorithmes, se trouve un fichier Makefile où se trouvent les diffèrents éxécutables de ce projet. Par ailleurs, nous séparons les fichiers C par thématique et par tests :

1. ListLC.h et ListLC.c , tests en mainS1.c
2. Work.h et Work.c, mainS2.C
3. Commit.h et Commit.c, mainS3
4. References.h et References.c
5. Merge.h et Merge.c, mainS5.c
6. myGit.c qui représente l'interface de ce projet.

De plus, ces mêmes fichiers sont à leur tour organisés en dossiers selon leur nature : tous les fichiers .h seront ainsi dans un dossier "Headers", les fichiers .c correspondants sont eux dans un dossier "Bibliotheque". Enfin, les tests restent dans le répertoire courant.

2 Vers la création d'enregistrements instantanés

L'objectif de ce premier chapitre est de se familiariser avec le concept d'instantané à travers la manipulation de fichiers et de ce qu'on appelle les fonctions de hachage, qui permettent de stocker une empreinte de nos fichiers.

Nous utiliserons notamment des listes chaînées et leurs fonctions associées afin de mener à bien de telles opérations.

2.1 Prise en main du langage Bash : appels systèmes et fonctions de hachage

Nous avons pris soin d'écrire les signatures des fonctions suivantes dans le fichier ListLC.h et leur code dans ListLC.c

Dans le cadre de cette première partie, il était question de prendre connaissance avec le concept de hash, caractérisant un fichier et permettant de l'identifier.

```
int hashFile(char* source, char* dest);
```

qui calcule le hash du contenu du premier fichier et l'écrit dans le deuxième fichier.

```
char* sha256file(char* file);
```

qui renvoie une chaîne de caractères contenant le hash du fichier donnée en paramètre.

2.2 Implémentation d'une liste de chaînes de caractères

Nous avons utilisé la structure suivante dans le cadre de cet exercice :

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

(trouvable dans le fichier ListLC.h).

Dans le cadre de cette partie, il fallait manipuler la structure définie plus haut. Nous avons donc pu : créer une cellule, génerer une liste, convertir une cellule puis une liste en une chaîne de caractères (puis l'opération inverse). Nous avons également établit des fonctions permettant de trouver une cellule dans une liste. Enfin, nous avons écrit des fonctions permettant de recopier le contenu d'une liste dans un fichier destinataire, puis l'inverse.

Remarques et observations :

Nous avons remarqué qu'il fallait etre méticuleux avec certaines des fonctions. En effet, aux questions 2.4 et 2.7, Nous avons développé les fonctions suivantes :

```
char* ltos(List* L);

List* stol(char* s);
```

lorque l'on convertit le contenu d'une liste en chaine de caracteres, le processus se fait avec le format suivant : chaine1|chaine2|chaine3|... Cependant, l'opération inverse doit se faire avec minutie et faire attention à tous les caractères, y compris les espaces et sauts à la ligne.

Ces fonctions sont directement liées à celles créées en 2.8 :

```
void ltof(List* L, char* path);
List* ftol(char* path);
```

La première permet d'écrire le contenu d'une liste dans un fichier. La deuxième lit le contenu d'un fichier et le transforme en liste. Ces fonctions supposent la maîtrise des fichiers et de leur manipulation. De plus, il faut avoir bien codé les fonctions en 2.4 et 2.8.

2.3 Gestion de fichiers sous git

Ici, en plus d'exploiter les foctions créées plus tot, il s'agit de s'introduire à la notion d'instantané à partir des fonctions suivantes :

```
int file_exists(char * file);
```

une fonction qui rend 1 si le fichier "file" existe dans le répertoire courant et 0 sinon.

```
void cp(char *to, char *from);
```

permet de copier le contenu du fichier "from", à l'aide d'une lecture ligne par ligne , dans le fichier "to".

```
char * hashToPath(char * hash);
```

rend le chemin d'un fichier à partir de son hash, en y insérant le caractère '/'.

```
void blobFile(char * file);
```

permet d'enregistrer un instantané de "file", qui sera stocké dans un répertoire créé grâce à la commande "mkdir", avce en plus un test sur l'existence du répertoire.

3 Enregistrement de plusieurs instantanés

Il est maintenant important de comprendre les workfiles : des structures représentant des fichiers ou répertoires, de par un nom, un hash, et un mode qui renseigne les autorisations que l'on a sur le fichier/dossier en question.

Un WorkTree est une structure qui organise des WorkFile dans un tableau, dont la gestion est décrite par les fonctions qui suivent.

3.1 Fonctions de manipulation de base

Toutes les structures ainsi que signatures des fonctions définies dans ce qui suit se trouvent dans le fichier Work.h Les algorithmes des fonctions sont dans le fichier Work.c

Un WorkFile, en tant que structure, se définit de la façon suivante :

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

Pour la fonction donnée suivante :

```
void setMode ( int mode , char * path );
```

nous avons géré le cas des octales par rapport au mode, en remplaçant "%d" par "%o".

Par ailleurs, pour la fonction :

```
wf *stwf(char *ch);
```

Il faut prendre en considération si le hash du workfile est initialisé à NULL ou pas. En effet, un hash NULL équivaut à deux tabulations de suite.

Nous avons également ajouté la fonction :

```
int isDir(const char * fileName);
```

qui précise si fileName est un fichier ou un repertoire.

Comme pour la partie précedente, nous avons pris soin de définir des fonctions de manipulation de base des WorkFile, puis des WorkTree : initialisation, conversion en chaine de caractères, et insertion de workfile dans un workTree, qui est d'ailleurs défini ainsi :

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

Remarques et observations

Nous avons pris soin de définir une fonction qui désalloue un workfile, pour mieux gérer les questions de mémoire, plus tard dans nos tests. Cette fonction est la suivante :

```
void freeWorkFile(WorkFile * wf);
```

3.2 Enregistrement instantané et restoration d'un WorkTree

On se sert accessoirement des fonctions suivantes :

```
char* blobWorkTree(WorkTree* wt);
```

retourne le hash de l'instantané créé à partir du WorkTree wt.

Cette fonction nous aide dans le développement des fonctions suivantes.

```
char* saveWorkTree(WorkTree* wt, char* path);
```

cette fonction va enregistrer le contenu du WorkTree wt dans son intégralité de façon récursive (les fichiers, ainsi que le contenu des répertoires).

```
void restoreWorkTree(WorkTree* wt, char* path);
```

restaure fichiers comme dossiers. Cependant, dans notre implémentation, si on restaure un dossier, il y a risque de ne pas pouvoir y accèder à cause d'un changement de mode aprés l'appel à la fonction. Cela rend la fonction restrictive.

3.3 Remarques et observations

Un détail non sans importance ici est le mode. En effet, il a pu nous paraître difficile de gérer le fait que les fonctions getChmod et Setmode sont définies sur des bases diffèrentes. Cependant, nous avons pu contrer ce problème dans la fonction blobWorkTree en imposant le mode d'accès grâce à la commande "setMode(0777, hashPath);".

4 Gestion de commits

Dans la suite, on va manipuler les instantanés de WorkTree au travers de commits : des tables de hachage, qui renseignent la chronologie d'implantation des instantanés, et donc permettent d'organiser les différentes versions d'un même projet.

4.1 Fonctions de base pour les commits

Les fonctions et structures définies dans ce qui suit sont dans le fichier Commit.h Les algorithmes associés sont dans Commit.c

Une constante que nous avons définies est la taille maximale d'une table de hachage, qui est de 1000.

Un commit est représenté par la structure suivante :

```
typedef struct key_value_pair {
    char * key ;
    char * value ;
} kvp ;

typedef struct hash_table {
    kvp ** T ;
    int n ;
    int size ;
} HashTable ;
typedef HashTable Commit ;
```

Comme pour les parties précèdentes, nous avons écrit du code de manipulation de base de la structure de commit : initialisation d'un commit et de ses éléments(key-value), récupération d'éléments, conversion en chaine de caractères puis écriture du contenu dans un fichier.

Pour mener à bien ces opérations, nous avons choisi la fonction suivante :

```
unsigned long HashSdbm(char *str);
```

Pour la fonction :

```
kvp *kvts(char * str);
```

Il faut aussi faire attention à l'initialisation du champ value : lui donner une chaîne de caractères si le sscanf rend 2, et NULL sinon.

4.2 Gestion temporelle des commits de manière linéaire

Les signatures des fonctions renseignées ci-joint se trouvent dans le fichier References.h.
Les commits nous aident à récupérer le hash d'un WorkTree associé, ainsi que des informations sur les prédesseceurs, l'auteur du commit et le message de mise à jour de la part de l'auteur.

4.3 Manipulation de références

Dans cet exercice, on créé le repertoire caché .refs et on y insère les fichiers vides "master" et "HEAD" à l'aide de la fonction suivante :

```
void initRefs();
```

À partir de là, on apprend à manipuler les réfèrences en modifiant le hash associé, ou encore supprimer une référence.

La fonction suivante avait cependant besoin de plus de rigueur :

```
char* getRef(char* ref name)
```

En effet, lorsque l'on recopie un hash dans un fichier initiallement vide, un saut à la ligne est généré automatiquement, ce que l'on a pu géré. Aprés ça, on pouvait y lire un hash de format correct, et récupérer son path.

4.4 Simulation de la commande GitAdd

On crée ici une fonction qui imite la commande Git add :

```
void myGitAdd(char* file_or_folder);
```

L'utilisateur, à travers la commande, place progressivement dans la zone de préparation (.add), les fichiers associés à ses prochains commit. La fonction crée alors d'abord le .add s'il n'existe pas déjà et ajoute au WorkTree associé le fichier ou repertoire passé en paramètre.

4.5 Simulation de la commande GitCommit

La commande Git commit est ici imitée à l'aide de la fonction suivante :

```
void myGitcommit(char* branch name, char* message);
```

Cette fonction effectue beaucoup de vérification : est ce que le répertoire .refs existe ? Est-ce-que la branche passée en paramètre existe ? Est-ce-que HEAD pointe bien sur le dernier commit de celle-ci ?
On emploie par exemple les fonctions :

```
int file_exists(char *file);
Cell *searchList(List *L, char *str);
```

On supprime alors le .add aprés avoir récupéré le WorkTree associé et on fait appel à saveWorkTree définie précedemment pour récuperer le hash de ce même WorkTree.
On crée alors un commit c et on indique que le hash de la branche passée en paramètre constitue le prédecesseur de c (si le fichier n'est pas vide). On associe également à ce nouveau commit le message passé en paramètre tout en vérifiant sa nature : rien n'est ajouté si le message vaut NULL; sinon, c'est le message lui-même.
L'enregistrement de l'instantané est effectué grâce à la fonction :

```
char *blobCommit(Commit *c);
```

Ensuite, il faut mettre à jour les informations de la branche passée en paramètre et HEAD, en remplaçant leur contenu avec le hash de c.

5 Gestion d'une timeline arborescente

Dans cette partie, il est question d'apprendre à manipuler diffèrentes branches d'un même projet tout en conservant une branche principale. Dans le cadre de notre projet, on utilise un répertoire ".refs" qui contient des réfèrences, comme "master" qui renferme en elle le hash du dernier commit de la branche principale. La manipulation de plusieurs branches supposent cependant des contraintes, que l'on verra plus en détail plus bas.

5.1 Fonctions de base de manipulation des branches

Premièrement, il faut créér un fichier .currentbranch contenant le nom de la branche courante, initialisée à "master", à l'aide de la fonction suivante :

```
void initBranch();
```

À partir de là, on peut, à l'aide de fonctions à opérations relativement simples : récupérer le nom de la branche courante, vérifier qu'elle existe, afficher son hash, et rendre une liste contenant tous les commits associés à une même branche voire toutes les branches.

En plus, on apprend à créér une nouvelle branche pointant vers le même commit que HEAD :

```
void createBranch(char* branch);
```

5.2 Git checkout-branch

```
void myGitCheckoutBranch(char* branch);
```

Qui modifie le fichier .currentbranch défini plus haut et y copier le nom de la branche passé en paramètre (donc le contenu de HEAD aussi). Ensuite, on restaure la version des fichiers que l'on avait associé à cette même branche (ce que l'on fait grâce à la fonction restoreCommit).

5.3 Git checkout-commit

```
myGitCheckoutCommit(char* pattern);
```

Cette commande est un peu plus délicate car elle permet d'intéragir avec l'utlisateur via le terminal, pouvant l'amener à faire des choix.

en effet, l'utilisateur, selon la chaîne de caractères qu'il écrit sur le terminal (appelons-la "patern"), lui sera retournée la liste filtrée (obtenue grâce à filterList) des commits commançant par pattern. Aprés ça, il doit choisir le commit vers lequel il souhaite se rediriger.

La fonction gère bien sûr les diffèrents cas de figure : si pattern n'est associé qu'à un seul hash, le contenu de HEAD est directement modifié; s'il est associé à plusieurs hash, la liste des possibilités est affichiée et l'utilisateur doit affiner sa recherche. Enfin, si pattern ne correspond à aucun des commits, un message d'erreur est affiché.

Une fois que le commit cherché est trouvé, on fait un appel à la fonction restoreCommit pour restaurer l'état du fichier associé.

6 Gestion des fusions de branches

Aprés s'être habituées à un environnement avec plusieurs branches, nous essayons ici de fusionner deux branches, opération à utilité non négligeable. En effet, il existe une commande Git appelée "merge", qui a pour but de créer un nouveau commit dont le WorkTree associé est une fusion des WorkTree associés au dernier commit de chacune des deux branches. Cette opération ne se fait pas sans contraintes, comme on pourra le voir plus tard avec la gestion des éventuels conflits.

Dans cette partie, nous avons pris soin de créér deux fonctions pour faciliter toutes nos étapes :

```
WorkTree *recuperer_WorkTree(char *branch);
void divideList(char*branch, List *conflicts,
List **conflicts_current, List **conflicts_remote);
```

La première, à partir du nom d'une branche, va récupérer le path du dernier commit réalisé qui va ensuite récupérer le path du champ "tree" et le WorkTree associé. La fonction s'éxecute correctement sous l'hypothèse que la branche contient au moins un commit.

La seconde permet d'intéragir avec l'utilisateur car pour chaque fichier dans la liste conflicts, celui-ci peut choisir de l'ajouter à conflict_current (liste des fichiers en conflit à garder depuis la branche courante), ou conflicts_remote (celle associée à la branche passée en paramètre). Si l'utilisatieur ne rentre pas de proposition valide, alors le conflit est ajouté à conflicts_current par défaut.

Ces fonctions seront primordiales dans l'opération de "merge", introduite plus haut, à l'aide des fonctions suivantes :

```
List *merge(char *remote_branch, char *message);
void createDeletionCommit(char *branch,
List *conflicts, char *message);
```

Pour la deuxième, nous avons fait attention de vérifier que dans la zone de préparation, il y a au moins un fichier qui a été ajouté. Sous cette condition, on crée le commit de déletion. Dans la fonction myGitCommit, lorsqu'on gère des conflits, il y a l'option de le faire manuellement, alors l'appel de createDeletionCommit n'était d'abord pas évident. La création du commit de déletion pour la branche courante se fait en tenant compte des conflits qu'on garde depuis remote, et inversement.

7 Tests par semaine, myGit et observations

Pour tester l'ensemble des fonctions, nous avons à disposition diffèrents fichiers main :

MainS1.c qui permet de tester les fonctions en relation avec les listes, MainS2.c qui fait de même avec les workfiles et worktrees, mainS3.c avec les commits, et enfin mainS5.c avec les fonctions de type merge.

Nous avons pris soin d'utiliser les fonctions free associées au type de données adaptées pour gérer au mieux les fuites de mémoire.

Les tests et alorithmes décrivant les fonctionnalités de ce logiciel sont dans le fichier myGit.c

1. ./myGit init pour initialiser .refs
2. ./myGit add A.c pour ajouter le fichier A.c dans le .add
3. ./myGit commit master -m "message" pour ajouter le la version du fichier A.c dans la branche désignée avec un message choisi par l'utilisateur.
4. ./myGit branch Feature qui va créer la branche avec un nom donné, ici Feature.
5. ./myGit list-refs qui donne la liste des branches.
6. ./myGit get-current-branch qui retourne la branche courante.
7. ./myGit checkout-branch Feature qui permet de basculer vers Feature.
8. ./myGit branch-print qui affiche le hash des fichiers associés à la branche passée en paramètre .
9. ./myGit checkout-commit pattern qui permet de restaurer le commit qui commence par pattern et donc par conséquent le fichier associé.
10. ./myGit merge : cas 1 : Aucun fichier en conflit.

    cas 2 : au moins 1 conflit : soit prendre tout depuis master soit tout depuis remote soit employer divideList.

Lors de l'écriture de myGit.c, plutôt que d'utiliser un if , nous avons chosi l'instruction switch pour tous les cas possibles de nombre d'arguments sur la ligne de commande.

8 Conclusion

En définitive, le logiciel créé par le biais de nos algorithmes reprend les codes du réel logiciel Git, et nous y avons trouvé un grand intérêt.

Il est vrai le code connaît des limitations. Pourtant, il fut trés instructif et nous a appris les bases du vrai logiciel, que l'on pouura utiliser à l'avenir, dans nos futurs projets.

Ce projet peut être amélioré : en effet, nous pensons que certaines fonctions pouvaient être modifiées pour se rapprocher du vrai git, comme c'est le cas pour myGitCommit, à laquelle on peut ajouter la fonctionalité de savoir dans quelle branche on se situe; ce qui allègera le nombre d'arguments requis lors de l'appel de cette commande sur le terminal. De même, nous pouvons améliorer certaines fonctions en ajoutant des messages personnalisés pour les cas limites, lors d'un merge par exemple, pour guider au mieux l'utilisateur.