Rapport de projet: Git-version-control

> Réalisation d'un *outil de suivi et de versionnage de code* (`git`)

> **Current progress**: 100% finished

Préliminaire

Highlight

Information principale

- **Nom de projet**: Git-version-control
- **Lien Github du projet**: https://github.com/Meriadoc-gitgit/GIT-version-control.git
- **Instructeurs**: Paul Hermouet & Louis Serrano

Cadre de projet

> **Objectif**: Étudier le fonctionnement d'un logiciel de gestion de versions, *en détaillant* différentes structures de données impliquées dans sa mise en œuvre.

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)* qui

- offrent un **accès aisé à l'historique de toutes les modifications effectuées** sur les fichiers + permettent notamment de *récupérer une version antérieure en cas de problème*.
- très utiles dans le cadre de travail collaboratif, permettant de *fusionner de manière intelligente différentes versions d'un même projet*.

Crew member

1. **Vu Hoang Thuy Duong**
    - Numéro d'étudiant: 21110221
    - Groupe: MONO 6
    - Titre: MVP d'algorithme
    - Pseudo: authentic vietnamese
    - LinkedIn: https://www.linkedin.com/in/meriadocdgv/
    - Github: https://github.com/Meriadoc-gitgit
2. **Halimatou Diallo**
    - Numéro d'étudiant: 21114613
    - Groupe: MONO 6
    - Titre: reine de valgrind *(vaincre toute sorte de data leak)*

Pré-requis

1. Connaissances de base des commandes en Bash pour la manipulation du système: `mkdir`, `rm`...
2. Connaissances aisées en langage C *(projet programmé 100% en C)*

3. Connaissances de base en Structures des données
    - Algorithme importante implémentée:
        - Algorithme de hachage
        - Gestion de collisions
        - Manipulation d'une structure arborescente
        - ...
    - Cours suivi: LU2IN006 - Structure des données | Sorbonne Université | 2022-2023
4. Exigence secondaire: extra autonomie, bonne gestion de stress + extra sens de team-player *(tout au long du projet)*

Manuel d'usage

Choix de fonction de hachage

> Algorithme choisie: **DJB2** - la magie du nombre 33

Implémentation

```c
int hash(char* str) {
        unsigned long hash = 5381;
        int c;
        while ((c = *str++))
                hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
        return hash;
}
```

Pourquoi un tel choix ?

L'explication est simple. En comparant avec les 2 autres algorithmes fournies *(dont SBDM et LOSE LOSE)*, DJB2 est un algorithme renommée et certifié en tant que meilleur algorithme pour les chaînes de caractères par une grande population *(avis de l'auteur du site + de la grande communauté des développeurs lus sur Stackoverflow et d'autres forums)*.

L'utilisation de DJB2 garantit une excellente distribution et vitesse sur de nombreux jeux de clés et tailles de table différents. Pourtant, la magie du nombre 33 *(pourquoi il fonctionne mieux que de nombreuses autres constantes, premières ou non)* n'a jamais été suffisamment expliquée.

> **Note**: Le résultat de cette fonction peut notamment être négatif. Afin d'assurer une bonne insertion des couples `(key,value)` dans `Commit`, il est forcément recommandé d'utiliser la valeur absolue du `hash` renvoyé par DJB2.

Changement par rapport à l'algorithme fournie

1. Changement de **type du paramètre en entrée**

    Suivant l'algorithme fourni sur le site web mentionné dans l'énoncé, le type du paramètre pris en entrée sera un `unsigned char`. Pourtant, en testant cette fonction avec un tel paramètre, la boucle risque de s'arrêter immédiatement une fois le premier caractère de la chaîne en entrée parcouru au lieu d'avancer jusqu'à son dernier élément.

    Cela cause donc un grand problème de collision, vu que les résultats de hachage ne seront que les hachages des caractères à la place des chaînes de caractères. Donc, pour améliorer la complexité du programme et avoir une meilleure gestion de collisions, nous avons décidé de modifier ce paramètre en `char*`

2. Changement de **type retour de la fonction**

Pour simplifier la compréhension du programme, nous avons modifié le type de retour de cette fonction en `int` au lieu de `unsigned long`. En terme de résultat, après plusieurs assertions, nous avons trouvé que cela ne cause pas de problème à la simulation de l'ensemble du logiciel.

Commandes `myGit` simulées

Quelques notations importantes

Pour assurer une bonne poursuite tout au long du projet, nous notons ici quelques commentaires brefs et importants.

- **Nom du logiciel**: `myGit` *(a.k.a. `git` version Sorbonne)*
- Concernant les commandes, leurs noms et leurs fonctionnalités restent les mêmes que ceux de `git`

Fonction de base

1. `blob[File | WorkTree | Commit]`
2. `saveWorkTree`
3. `restoreWorkTree`

Simulation de `myGit`

```
./myGit <command> [<option> <arg1> <arg2> ...]
```

- `command`: commande indiquant l'action voulue de l'utilisateur, à choisir à partir de la liste ci-dessous
- `option`: facultatif, `-m`, `-f`, *etc*.
- `arg1`, `arg2`...: facultatif selon `command` utilisée, indiquant les données nécessaires pour la simulation des commandes

Liste de commandes

Cette liste est composée de **14 commandes primordiales** du logiciel simulées tout au long de ce projet semestriel.

1. `./myGit init`: initialise le répertoire de références
2. `./myGit list-refs`: affiche toutes les références existantes
3. `./myGit create-ref <name> <hash>`: crée la référence `<name>` qui pointe vers le commit correspondant au `<hash>` donné
4. `./myGit delete <name>`: supprime la référence `name`
5. `./myGit add <elem> [<elem2> <elem3> ...]`: ajoute un ou plusieurs fichiers/répertoires à la zone de préparation *(prochain commit)*
6. `./myGit list-add`: affiche le contenu de la zone de préparation
7. `./myGit clear-add`: vide la zone de préparation
8. `./myGit commit <branch-name> [-m <message>]`: effectue un commit sur une branche, avec ou sans mesage descriptif
9. `./myGit get-current-branch` : affiche le nom de la branche courante
10. `./myGit branch <branch-name>`: crée une branche qui s'appelle `<branch-name>` si elle n'existe pas déjà
11. `./myGit branch-print <branch-name>`: affiche le hash de tous les commits de la branche, accompagné de leur message descriptif éventuel
12. `./myGit checkout-branch <branch-name>`: réalise un déplacement sur la branche `<branch-name>`
13. `./myGit checkout-commit <pattern>`: réalise un déplacement sur le commit qui commence par `<pattern>`
14. `./myGit merge <branch> <message>`: met à jour la branche courante avec la branche `<branch>` en gérant les conflits

Structure de projet

- Répertoire **desc**:
  - 5 **fichiers de type Markdown** *(l'extension* `.md` *)*, représentant 5 partie du projet, utilisés pour noter les énoncés et les points importants requis au fur et à mesure du projet principal
  - Sous-répertoire **sub_img**, contenant les images illustrant les schémas contenus dans les fichiers Markdown
- Répertoire **src**: contenant les **sources de codes primordiaux du projet**, dont les fichiers suivants:
  - 1 fichier **myGit.c**: simulateur principal de la commande `./myGit` du projet, fait appel à **bash.h** et **src.h**
  - 1 fichier **Makefile**: contenant les commandes configurées pour compiler et exécuter le programme
  - 1 fichier **bash.h**: contenant les fonctions de manipulation qui utilisent certaines commandes de Bash et les fonctions de simulation de **myGit**
  - 1 fichier **src.c**: contenant les fonctions de base qui sont utilisées dans les fonctions du fichier **bash.h**
  - 2 fichiers cachés **.current_branch** et **.add**: contribuent au contrôle de version et de branches de **myGit**
  - 2 sous-répertoires cachés: **.refs** et **.** :
    - **.refs**: contenant les références *(ou les branches)* pour le contrôle de version du projet
    - **.** : utilisé pour la restauration et sauvegarde des fichiers afin de ne pas rendre ambigü le projet suite à la création de plusieurs fichiers et répertoires suivant son hash
- Répertoire *secondaire* **main-version**: contenant les versions du **main** pour tester ce projet si souhaité *(pour une bonne exécution, modification des path recommandée)*
- 1 fichier **readme** de type Markdown: utilisé pour la représentation du projet sur Github + fournir une meilleure visualisation
- 1 fichier **report** de type Markdown et PDF: le rapport du projet

Analyse de Warning

> Ainsi déjà bien gérés avant le rendu de projet

Concernant les Data Leak

Gestion de valgrind

Jusqu'à présent, **ce programme a atteint 0 bytes de fuites de mémoire**, qui satisfait l'exigence d'un bon projet à rendre. Pourtant, avant d'arriver à un tel niveau, le nombre total de bytes de fuites de mémoire était incroyable.

Au départ, nous avons séparé notre projet en 6 programmes, dont: ***myGit, test, commit, bash, cell et work***; afin d'assurer que les tests ne nous paraissent pas trop longs et incompréhensibles. Pour chaque programme, sauf celui de ***myGit***, le nombre de bytes de fuites de mémoire après exécution de valgrind pouvait atteindre environ 25 milliards, incroyable pour un projet qui ne dure que 2 mois depuis son lancement.

Au fur et à mesure, nous avons fait beaucoup d'efforts pour trouver la source du problème, et de désallouer les variables initialisées lors des ***main*** et ceux internes des fonctions. Malheureusement, cela n'améliore pas la situation, et nous nous trouvions coincées avec nos instructeurs à chaque séance de TP.

Enfin, nous avons remarqué que cette catastrophe ne survient pas des allocations, mais du programme et du fichier lui-même. À chaque test, lors de l'initialisation des variables, ou de l'instruction d'affichage: `printf("test de ltos: %s\n", ltos(L));` par exemple, on a déjà enregistré une donnée de base dans le programme, dont ici c'est la fonction `ltos`. Chaque enregistrement comme celui-là cause une fuite de mémoire, et les données liées à cette fonction seront diffusées hors du programme. Donc, pour gérer une telle fuite, la seule solution est simple: enlever tous les tests et les programmes hors de celui principal.

Pour le programme principal, dont **myGit**, aucun byte de fuite de mémoire est reporté, avec une seule erreur lors de l'utilisation de `strcmp` pour comparer les chaînes de caractères de `argv[1]` avec les chaînes de commandes. Cette

erreur est venue du fait que `argv[1]` est une chaîne qui n'est initialisée qu'avec nos propres commandes dans le Terminal, pas dans le programme lui-même. Donc, il n'est pas forcément un problème qui attire trop de souci.

Lors des tests

Les fuites de mémoire ne peuvent pas seulement être trouvées lors des ***valgrind***, mais possiblement lors de l'exécution du programme lui-même sans être visible lors de sa compilation.

Par exemple, lors de l'exécution d'un de nos programmes précédents, l'affichage est comme suit:

```
test de ltos sur L:
(*master) {
        appendWorkTree(wt,(*master)->data,sha256file((*master)->data),getChmod((*master)->data));
        appendWorkTree(wt1,(*master)->data,sha256file((*master)->data),getChmod((*master)->data));
        *master = (*master)->next;
}
List* conflicts = initList();
WorkTree* wt2 = mergeWorkTree(wt,wt1,&conflicts);
printf("conflicts mergeWT: %s\n",
.|..|main.o|66|src.txt|.DS_Store|.current_branch|test|bash.c|Makefile|src.h|src.o|myGit.c|main.c|bash.
h|bash.o|.add|test.txt|.refs|file_test.txt)
```

dont le programme de départ ne va produire que `test de ltos sur L: .|..|main.o|66|src.txt|.DS_Store|.current_branch|test|bash.c|Makefile|src.h|src.o|myGit.c|main.c|bash.h|bash.o|.add|test.txt|.refs|file_test.txt` , et toute la partie qui apparaît au milieu de cet affichage appartient au corps du fichier **src.c** *(c'est juste un exemple pour démontrer le dysfonctionnement du programme)*.

Ce problème est lié au fait que certaines variables ne sont pas allouées suffisamment, le plus souvant quand on utilise `sprintf` et `strcat` ou `strtok`. Il est recommandé d'allouer avec le maximum possible, et puis les désallouer si nécessaire pour assurer le fonctionnement global.

Concernant les variables redondant

En écrivant les instructions de compilation dans un ***Makefile***, on écrit souvent:

```
gcc -g -Wno-unused-parameter -c bash.c
```

dont l'option `-g` pour débugger les informations dans le fichier de sortie. Malheureusement, avec une telle commande, il sera difficile de pouvoir trouver les variables non utilisées tout au long du projet. La syntaxe suivante est plus recommandée:

```
gcc -Wall -g -Wno-unused-parameter -c bash.c
```

L'option `-Wall` est utilisé pour lister les ***Warning*** concernant les variables redondantes du programme par exemple.

Ces variables ne causent pas de problème en terme de compilation et d'exécution, mais elles vont augmenter la complexité globale et temporelle pour leur allocation ou initialisation, et donc rallonger le temps d'exécution. Il vaut mieux éviter au maximum possible leur présence dans le projet.

Concernant les instructions de test `if-else`

Ces instructions ne paraissent pas comme un tel problème pour certaines fonctions. Pourtant, il est strictement recommandé de respecter l'ordre des `if-else if-else`.
Par exemple, en écrivant comme suit, le code compile mais ce sera impossible d'exécuter:

```
if (instructions1) {
        ...
}
if (instructions2) {
        ...
}
...
else {
        ...
}
```

En suivant une avalanche d'instructions de test `if`, le programme se trouve dans une situation coincée et aura une confusion lors de l'entrée et de la sortie des tests.

Pour remédier à ce problème, il faut simplement modifier le code comme suit:

```
if (instructions1) {
        ...
}
else if (instructions2) {
        ...
}
...
else {
        ...
}
```

Conclusion et Remerciement

Tout au long du projet, nous avons appris beaucoup de choses. Grâce aux manipulation des chaînes de caractères, nous retenons l'importance d'un bon choix de taille d'allocation, et l'impact des pointeurs sur l'espace de mémoire en appliquant les fonctions globales sur certaines variables. Parallèlement, nous comprenons mieux la différence entre l'utilisation d'un tableau et celle d'un pointeur, et leur impact sur l'espace de mémoire global après l'exécution d'un programme. De plus, nous sommes capables de mieux gérer les fuites de mémoire, à trouver les erreurs minuscules dans les codes, et à comprendre la source des fautes de segmentation.

Nous tenons à exprimer nos sincères remerciements auprès des responsables, nos instructeurs et professeurs qui nous ont accompagnés tout au long de ce projet.

De Duong à Halimatou: Merci d'avoir été ma première et meilleure binôme jusqu'à maintenant.

Références

1. DJB2 Algo explained | https://thealgorithms.github.io/C/d4/de3/hash__djb2_8c.html#details
2. Hash Functions for Strings | https://youtu.be/jtMwp0FqEcg
3. Options de fonction de hachage | http://www.cse.yorku.ca/~oz/hash.html