C Avancé

Rapport de Projet

Bastien CROCHEZ **Marwin DAMIS**

Introduction

Le projet consistait en la réalisation d'une macro et d'un programme afin de visualiser l'exécution d'un programme en C.

Le programme, appelé profiler, permet une visualisation claire des résultats, à la fois en console mais également avec une interface graphique. Le profiler offre la possibilité à l'utilisateur d'effectuer des tris sur les fonctions (par temps moyen, nombre d'appels, et temps total) mais aussi de se déplacer dans l'arbre et donc de choisir une racine différente du main pour un affichage personnalisé.

Dans ce rapport, nous décrirons donc l'architecture de notre projet, puis nous expliquerons nos différents choix de développement en détaillant les deux parties de ce projet : la macro et le profiler.

Architecture

Le profiler est découpé en deux modules bien distincts :

- tree.h : C'est dans ce module que sont réalisées les principales actions comme la création d'un arbre d'appels à partir d'un fichier de log ou encore les différents tris.
- gui.h : Module permettant l'affichage du projet dans une fenêtre et offrant ainsi une interface agréable pour consulter les résultats.

Choix de Développement

Nous avons développé ce projet à 2 en essayant d'utiliser efficacement Subversion.

Nous avons choisi de commencer par la macro afin d'obtenir des données que nous pourrions utiliser dans notre profiler, et nous avons ensuite travaillé sur le profiler, représentant la majeure partie de ce projet.

La Macro

Après plusieurs essais, nous avons finalement choisi de déclarer le fichier de log en global afin d'éviter d'éventuels problèmes liés à l'ouverture / fermeture de ce même fichier en chaîne.

Comme pour le profiler, la difficulté majeure aura été la gestion des temps. Nous avons choisi d'utiliser des double en limitant le nombre de chiffres après la virgule à 6. Nous nous sommes par la suite inspiré de cette solution pour résoudre notre problème de temps dans le profiler.

Le Profiler

Afin de représenter l'arbre des appels, nous avons choisi d'utiliser une structure d'arbre dit "fils gauche / frère droit" comme suggéré dans l'énoncé. Notre structure se présente de la façon suivante :

```
typedef struct node {
        char name[50];
```

```
        double start, end, callTime;
        struct node *sonL, *broR;
} Node, *Tree;
```

On conserve à la fois le temps de début, le temps de fin et le temps d'appel.

La principale difficulté que nous avons pu rencontrer était justement la gestion des temps. Les temps récupérés dans l'arbre au niveau de la fonction de chargement du fichier de log étaient les bons mais ils posaient ensuite problème lors des tris.
Au final notre choix s'est arrêté sur l'utilisation de double pour start, end et callTime.

Après avoir construit l'arbre, nous fabriquons un tableau contenant les données des fonctions appelées par le programme. Pour cela nous avons 2 structures :

- La structure du tableau, qui contient le tableau et la taille du tableau :

```
typedef struct table {
        DataFunction* tab;
        int size;
} TabOfFun;
```

- La structure de données, qui contient le nom de la fonction, son nombre d'appels et le temps total de son exécution. Nous ne gardons pas en mémoire le temps moyen d'appel car il peut facilement se calculer à partir du nombre d'appels et du temps total :

```
typedef struct dataFunction {
        char name[50];
        int nbCall;
        double totalTime;
} DataFunction;
```

Pour construire ce tableau nous parcourons l'arbre récursivement. Lorsque l'on rencontre une fonction qui n'est pas dans le tableau, nous créons un nouveau DataFunction et l'insérons dans le tableau. Sinon nous incrémentons le nombre d'appels de la fonction et ajoutons son temps d'exécution au temps total.

Ensuite, pour trier le tableau selon le choix de l'utilisateur, nous avons implémenté des algorithmes de tri rapide pour trier le tableau par nombre d'appels décroissant, par temps total d'exécution décroissant et par temps moyen décroissant. C'est pour ces tris que nous gardons en mémoire la taille du tableau.

L'affichage se fait à partir de la racine sélectionnée. La couleur des rectangles dépend du temps d'appel de la fonction par rapport au main. Nous avons défini une taille minimale; dans le cas où il n'y a plus de place pour afficher les fonctions restantes, on dessine un rectangle contenant "S" ou "B" selon qu'il reste des fils ou des frères.

Nous avons défini une structure Color pour la gestion des couleurs suite à un problème avec l'utilisation d'un MLV_Color. Cette structure est très basique :

```
typedef struct color {
```

```
        int red;
        int green;
        int blue;
        int alpha;
} Color;
```

Elle comporte 4 int : 3 pour le système rbg et 1 dernier pour l'opacité.
On se sert de la fonction MLV_rgba() pour effectuer la conversion vers un MLV_Color.

La taille des rectangles est calculée pour chaque fonction en comptant son nombre de frères. On utilise pour cela les 4 paramètres left, top, right, et bottom, qui correspondent aux 4 bordures du "cadre". Ce cadre se réduit au fur et à mesure que l'on progresse dans l'arbre. Une fois que la taille de celui-ci est inférieure à la taille minimale, on s'arrête.

Bugs

Nous avons finalement abandonné le profilage du profiler car il n'était pas à 100% fonctionnel. En effet, nous avons pu remarquer que la fonction main n'apparaissait pas dans le fichier de log produit en sortie. Comme nous n'étions pas certains qu'il s'agisse de la seule fonction à ne pas apparaître, nous avons préféré passer les balises de profilage en commentaire.

Collaboration

Comme demandé dans le sujet, nous avons procédé à un échange de fichiers de log avec le groupe de Maxime PETIT-HENGEN et Maxime PICHOU.
Leur fichier de log se trouve dans le dossier bin de notre projet sous le nom "larger_file.log", et les données sont exploitables. Nous leur avons également fourni un fichier de log.

Conclusion

En conclusion, nous pouvons donc dire que malgré quelques petits problèmes pour le profilage du profiler, le projet aura été mené à bien et fournit un résultat satisfaisant. En dehors de la gestion des temps dans l'arbre, nous n'avons pas rencontré de difficultés en particulier au niveau du code.
Enfin, ce projet aura également été l'occasion pour nous d'apprendre à utiliser de nouveaux outils comme le gestionnaire de versions ou encore doxygen pour la documentation.