Project Fractales 2018

Liliya Semerikova Gilles Peiffer

11 mai 2018

1 Introduction

Dans le cadre du cours de Systèmes Informatiques donné en 2018 à l'EPL par Olivier Bonaventure, il nous a été demandé d'écrire un programme multi-threadé capable d'effectuer quelques opérations sur des fractales fournies sous forme de texte dans des fichiers.

2 Architecture

L'architecture générale de notre projet est composée d'une couche de « producteurs-consommateurs ». L'utilisateur entre sur la ligne de commande, en argument de son appel du programme, des fichiers d'entrée, ainsi qu'un fichier de sortie en dernier argument.

Le programme va alors lire ces noms de fichiers d'entrée (dont fait partie l'entrée standard si un des fichiers est « - ») et créer pour chaque fichier qui n'est pas l'entrée standard un *thread*, qui aura pour tâche de lire, ligne par ligne, le contenu de ces fichiers. Chaque ligne a la forme `name width height a b`.

Ensuite, la fractale lue sur chaque ligne est interprétée et ajoutée à une structure de pile, de taille maximale supérieure au nombre de fichiers d'entrée et protégée par un mutex et deux sémaphores. Justement, avant de pouvoir ajouter une fractale sur la pile, les producteurs doivent vérifier que la pile n'a pas encore atteint sa taille maximale. On utilise pour cela un des deux sémaphores.

Les consommateurs ont aussi accès à cette pile ; elle fait donc office de *buffer* entre les producteurs et les consommateurs dans le programme. Eux aussi ont besoin de passer par un sémaphore, pour vérifier que la pile n'est pas vide.

Ensuite, autant pour les producteurs que pour les consommateurs, il y a un risque de violation de section critique. Il a donc fallu passer par un mutex pour l'accès à la pile.

Le programme traite ainsi toutes les fractales en entrée, et ignore les lignes vides ou les commentaires (lignes dont le premier charactère est un « # »). Les consommateurs prennent les fractales dans la pile, les enlèvent et calculent la valeur du nombre d'itérations pour chaque pixel. Ils utilisent alors un *setter* pour sauvegarder ces valeurs dans le tableau des valeurs de cette fractale.

Tout au cours de ces calculs de valeurs, le programme retient la valeur moyenne la plus élevée parmi les fractales déjà calculées. Dans le cas où plusieurs fractales ont la même valeur moyenne maximale, elle sont ajoutées à une *linked list*

sous forme de pile et le programme stocke un pointeur vers la tête de cette pile.

Selon que l'option `-d` soit fournie lors de l'appel du programme, on décide combien de fichiers `bmp` sont à fournir. Si elle l'est, chaque fractale sera convertie en image, sinon, uniquement les fractales dans la liste chaînée des meilleures moyennes sont dessinées.

3 Choix de conception

Pour sauvegarder tous les fichiers `bmp`, nous avons créé un dossier appellé « *outputs* ». Dans le cas où nous avons l'option `-d`, nous trouvons dans ce dossier toutes les fractales sous le nom `fractalName.bmp`. Les fractales avec les meilleures moyennes seront converties une seconde fois, dans des fichiers nommés `fichierOut_fractalName`. Si on n'a pas cette option, nous ne trouvons dans ce dossier que les fractales avec la meilleure moyenne sous le nom `fichierOut_fractalName`.

Un autre choix fait lors de l'implémentation est d'utiliser un double pointeur lors de l'allocation des tableaux de valeurs. Bien que cela rend l'allocation et la libération plus compliquées, les *getters* et *setters* deviennent alors plus simples.

4 Débogage et tests

Pour vérifier l'exactitude de notre programme, deux outils vus au cours ont été très utiles :

— Valgrind et ses modules ont été inestimables dans la détection des fuites de mémoire ;

— le débogueur `gdb` a permis de faire des pas dans le code et d'analyser les *Segmentation Faults* dans notre programme.

En plus de ces outils extérieurs, nous avons également écrit des tests avec la librairie *CUnit*, pour tester le fonctionnement correct de nos méthodes là où cela était possible.

Finalement, nous avons utilisé le logiciel `git` pour gèrer la gestion de versions, afin de pouvoir travailler à plusieurs sur le même code, sans se soucier de la cohérence des versions locales.

5 Évaluation

En faisant les tests avec Valgrind, nous nous sommes rendus compte qu'il y a encore des fuites de mémoire dans notre programme. La commande utilisée pour cela est :

```
$ valgrind -leak-check=full -show-leak-kinds=all -track-origins=yes -v ./main - -d output
```

Cependant, ces fuites sont de l'ordre de quelques centaines de *bytes* et ne sont donc pas réellement importantes.

6 Conclusion

Ce projet nous a permis de nous familiariser avec le langage C ainsi qu'avec les *threads*, les sémaphores, les mutexs ou encore Linux.

En plus, il s'agissait d'une opportunité d'utiliser la gestion de versions avec `git` ou encore les outils de débogage tels que Valgrind et `gdb`.