MENCHERO Dorian
ROUQUIER Lucas
GIRET Maël

Etape 2 Bis

Nous allons expliquer brièvement la résolution des problèmes, les modifications apportées dans cette nouvelle version de notre programme par rapport à l'ancienne ainsi que les résultats.

Pour tester le programme, il suffit d'entrer coremain suivit du rulename de son choix pour tester les 12 requêtes du jeu de test.

Résolution des problèmes :

1. Résoudre le problème lié à la segmentation fault lors du traitement de multiples requêtes.

Nous avons détecté que le problème venait du fait que les requêtes étaient stockées dans un seul fichier « requête.txt » . Ainsi, soit on ouvrait avec la fonction parseur() le fichier «requete.txt » en écriture plusieurs fois sans l'avoir fermé ce qui conduisait à une segmentation fault, soit les requêtes s'ajoutaient les unes à la suite des autres et la requête était analysée avec les précédentes, ce qui créait un arbre immense contenant toutes les requêtes mélangées.

Pour résoudre ce problème , l'idée était de fermer le fichier « requête.txt » et de le rouvrir pour écraser les données des requêtes précédentes à chaque fois qu'une requête est traitée.

Pour cela, nous avons ajouter un pointeur de fichier « requete » dans un header, de ce fait il est accessible depuis tout les fichier source et on peut donc l'ouvrir avec parseur() de api.c mais aussi de le fermer à chaque passage dans la boucle for de « coremain.c » .
Ainsi à chaque fois que l'on reçoit une requête, on ouvre le fichier «requete.txt », on y recopie la requête, on la traite, puis on ferme le fichier et on peut recommencer avec une nouvelle requête.

2. Résoudre le segmentation fault de la fonction searchTree() :

Nous nous sommes rendu compte qu'après le deuxième appel à la fonction searchTree(), un SegFault était renvoyé.

Notre fonction récursive searchTree utilisait une variable static qui permettait de dire si le nœud créé devait être la racine (dans le cas où c'est le premier), ou sinon le dernier nœud de la chaîne, et la fonction renvoyait la racine. Ainsi cette variable vaut 0 au premier appel puis 1 pour tout les autres. On ne créé qu'une fois la racine et tout les autres nœuds de type _Token sont ajouté à la fin de la chaîne.

Le problème est qu'après un purgeElement(), la racine est supprimée et notre variable static nous indiquait que l'élément _Token allait être ajouté à la fin de la chaîne dont la tête était la racine supprimé, ainsi nous avions un SegFault.

Nous avons donc défini une variable « actualisation » dans un header pour les avantages énoncés dans le problème précédent. Ainsi à chaque passage dans le boucle for de «coremain.c» , on mettait

actualisation à 0 de façon à créer une nouvelle chaîne de _Token avec une racine, ainsi, il n'y a plus eu de SegFault.

3. Fuite de mémoire :

Nous avons redécouvert valgrind après avoir rendu notre première version, nous nous sommes rendu compte que 1Mo de fuite de mémoire pour le traitement d'une seule requête est quand même assez important.

Avec valgrind, nous avons localisé les allocations qui n'ont pas été libérées, mais nous avons ajouté beaucoup de free() là où il fallait afin qu'il n'y ait plus de fuite de mémoire. Cet objectif est atteint dans cette nouvelle version de notre projet.

Modification apporté :

La principal modification apporté est l'ajout du fichier source coremain.c qui est quasiment le même que celui se trouvant sur Chamilo au détail près qu'il est adapté à notre code pour ce qui est du type de getRootTree (évoqué dans la documentation précédente).

Le main exécuté sera donc celui de coremain.c , mais nous avons commenté le main de api.c au cas où car il peut être utile en cas de débogage ou pour observer l'arbre complet.

Résultat :

Les résultats sont excellent, le score sur le jeu de test est de 12/12, tout les rulename valables peuvent être traités.
Les erreurs sont détectées, que ce soit les requête non valable ou les rulename non valable.

En terme de mémoire, il n'y a aucune fuite de mémoire.

Pour ce qui est de la rapidité, nous n'avons pas mesuré exactement le temps d'exécution, les 12 requêtes sont traités en environ une seconde. Mais tout dépend des capacités de la machine sur laquelle on lance le programme, de plus l'affichage consomme souvent du temps d'exécution.