TP 17 – Mini-memcheck

Le rendu de ce TP se fera sous forme de dépôt Git. Pour cela, créez un dépôt privé *sur Gitlab (`gitlab.com` ou `gitlab.istic.univ-rennes1.fr`) et ajoutez l'utilisateur `jmgorius` (pour Gitlab) ou `jgorius` (pour l'instance Gitlab de l'ISTIC) comme collaborateur.*

Rappel : Les anciennes versions de Git utilisent `master` comme nom de branche par défaut. Afin d'utiliser la nouvelle convention de nommage et de travailler sur la branche `main`, créez votre dépôt en utilisant la séquence de commandes

```
> git init <nom-du-dépôt> && cd <nom-du-dépôt>
> git checkout -b main
```

puis utilisez `git push -u origin main` lors de votre premier `push`.

1 Introduction

L'objectif de ce TP est d'implanter une version simplifiée de Valgrind que nous appellerons `mini-memcheck`. Cet outil affichera un résumé des fuites mémoires dans un programme C donné en argument.

L'idée directrice de ce TP est que l'on peut suivre les allocations en mémoire d'un programme en utilisant des méta-données dans une petite zone mémoire située avant chaque bloc de alloué. Le fichier `mini-memcheck.h` contient la définition de la structure `meta_data` correspondante. Les méta-données forment une liste chaînée en mémoire, chaque nœud enregistrant la taille du bloc mémoire qui le suit, le nom du fichier et l'adresse de l'instruction qui a effectué l'allocation ainsi qu'un pointeur vers le bloc suivant alloué en mémoire. Lorsque le programme termine, les nœuds restants dans la liste correspondent aux fuites mémoire.

2 Mini-memcheck

Le code fourni dans `mini-memcheck.c` remplace dynamiquement les appels à `malloc`, `calloc`, `realloc` et `free` dans le programme passé en argument par des appels à `mini_malloc`, `mini_calloc`, `mini_realloc` et `mini_free` respectivement. Vous devrez implanter ces quatres fonctions pour suivre les allocations réalisées par le programme en entrée.

Les fonctions `malloc` et cie. peuvent être appelées dans le fichier `mini-memcheck.c`. Vous ne devez pas réécrire votre propre version de `malloc` en utilisant des appels systèmes !

Le fichier `mini-memcheck.h` donne quelques indications additionnelles sur le code.

3 Variables globales

En plus des quatre fonctions de gestion de la mémoire, vous devrez veiller à mettre à jour les variables globales suivantes :

- `head` pointe vers le premier élément de la liste chaînée des méta-données.
- `total_memory_requested` contient le nombre total d'octets alloués par un programme pendant son exécution (sans méta-données).
- `total_memory_freed` contient le nombre total d'octets libérés par un programme pendant son exécution.
- `invalid_addresses` contient le nombre d'occurences de `realloc` ou `free` appelées avec un pointeur invalide.

Grâce à ces variables, `mini-memcheck` peut indiquer la quantité de mémoire allouée et libérée ainsi que la taille totale des fuites mémoires, comme le fait Valgrind.

4 Tests

Votre implantation devra être testée de manière appropriée. Les programmes de test peuvent ensuite être exécutés comme suit.

Tournez la page SVP...

```
./mini-memcheck ./test
```

Note : Les programmes de test devront tous être compilés avec les symboles de débogage pour permettre à `mini-memcheck` de retrouver le numéro de ligne correspondant à un appel de fonction causant une fuite mémoire.

4.1 Avertissement : `printf`

Soyez vigilants en appelant `printf` dans `mini-memcheck` ! Les fonctions d'affichage sur la sortie standard utilisent un tampon interne qui peut être alloué à l'aide de `malloc`.

Si vous appelez `printf` dans `mini_malloc` par exemple, le tampon sera alloué à l'aide de la version de la bibliothèque standard de `malloc`, mais la mémoire sera ensuite libérée par `mini_free` ! Le résultat peut donc être inattendu.

Vous pouvez utiliser les alternatives suivantes.

```c
// Unbuffered
fprintf(stderr, /* ... */);

// Disable buffering on stdout
setvbuf(stdout, NULL, _IONBF, 0);
```

4.2 Avertissement : Appels supplémentaires à `free`

Il est possible que vous observiez des appels supplémentaires à `mini_free` à la fin de l'exécution du programme. Ceci est un effet secondaire des choix d'implantation internes de `mini-memcheck`. La fonction `__libc_freeres` est appelée dans le fichier `mini-utils.c` à la fin du programme : elle fait appel à `free` plusieurs fois pour désallouer les éventuels tampons alloués en interne par la bibliothèque standard. Ceci ne devrait pas avoir d'impact sur votre implantation de `mini_free`.

5 Exemple

On considère le programme en entrée suivant :

```c
#include <stdlib.h>
int main(void) {
  void *p1 = malloc(30);
  void *p2 = malloc(40);
  void *p3 = malloc(50);
  free(p2);
  return 0;
}
```

La sortie de `mini-memcheck` devrait ressembler à

```
==4723== Mini-Memcheck
==4723==
==4723== LEAK REPORT:
==4723==    Leak origin: main (test.c:5)
==4723==    Leak size: 50 bytes
==4723==    Leak memory address: 0x1009790
==4723==
==4723==    Leak origin: main (test.c:3)
==4723==    Leak size: 30 bytes
==4723==    Leak memory address: 0x10096f0
==4723==
==4723== Program made 0 bad call(s) to free or realloc.
==4723==
```

```
==4723== HEAP SUMMARY:
==4723==     Total memory requested: 120 bytes
==4723==     Total memory freed: 40 bytes
==4723==     Total leak: 80 bytes
```

Vous pouvez également exécuter `mini-memcheck` sur d'autres programmes, comme `echo` :

```
> ./mini-memcheck echo 'Hello, world!'
==19506== Mini-Memcheck
Hello, world!
==19506==
==19506== Program made 0 bad call(s) to free or realloc.
==19506==
==19506== HEAP SUMMARY:
==19506==     Total memory requested: 1068 bytes
==19506==     Total memory freed: 1068 bytes
==19506==     No leaks, all memory freed. Congratulations!
```

`mini-memcheck` devrait fonctionner sur la plupart des utilitaires Unix standards, mais il est probable que votre implantation ne fonctionne pas sur des entrées trop complexes. Il est également possible que la sortie obtenue ne soit pas la même que Valgrind si vous tentez d'exécuter des programmes comme `python`. Ceci est tout à fait attendu : écrire un outil comme Valgrind est une tâche complexe qui nécessite un peu plus de travail que 2h de TP !

6 Pour aller plus loin : Sentinelles

Supposons que les utilisateurs de `mini-memcheck` ne soient pas très attentifs. Ils ont dépassé la capacité d'un tampon qu'ils avaient alloué en mémoire et corrompu ce qui suivait le bloc qu'ils avaient alloué. Pouvons-nous détecter ce type d'erreur ? Oui, en utilisant des *sentinelles*.

Une *sentinelle* est une valeur ou une suite de valeurs connues placées à la fin d'un bloc alloué en mémoire. Si la valeur d'une sentinelle est modifiée pendant l'exécution du programme, il y a un *buffer overflow*.

Ajoutez des sentinelles de valeur `0xBAAAAAAD` à la fin des allocations. Si la valeur de la sentinelle a été modifiée lors d'un appel à `mini_realloc` ou `mini_free`, affichez un avertissement.

Vous pouvez tester votre implantation sur des exemples du type suivant :

```c
#include <stdlib.h>
int main(void) {
  char* ptr = malloc(10);
  *(ptr + 10) = 'a';
  free(ptr);
  return 0;
}
```

Remerciements

Ce sujet est basé sur un sujet du cours *CS 241 : System Programming* de l'Université d'Illinois.