Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Traceur d'allocation

L'objectif de ce projet est de réaliser un outils d'aide à la programmation permettant d'analyser la gestion de la mémoire dans un code complet (éventuellement modulaire, mais exécutable).

Cette bibliothèque va agir comme un patch à la compilation en détournant les fonctions de gestion de mémoire vers des clônes permettant la récolte d'informations à l'éxécution.

Présentation générale

Alternative/complément à `Valgrind`

`Valgrind` est un outil très puissant – parfois trop – pour détecter les problèmes de mémoire.

Son principal défaut est qu'il agit directement sur l'*exécutable* `a.out` : il analyse donc **tout** ce que fait le process, c'est-à-dire notre code mais aussi tout les appels à des bibliothèques extérieures dont le code source n'est pas disponible (donc non corrigeable).

Il suffit d'utiliser `Valgrind` sur un programme très simple utilisant une `API` graphique (`OpenGl, SDL` ou autre) pour s'en rendre compte : des quantités de «problèmes» sont détectés même si notre code ne gère pas de mémoire directement.

L'outil proposé ici n'agit pas directement sur l'exécutable mais en amont, à la compilation, en tétournant certaines fonctions (ici, celles gérant la mémoire) vers des « clônes » qui vont collecter des informations, vérifier certains points, et tenter de surmonter les erreurs détectées.

Comme pour le debogueur `gdb`, l'activation de cet outil se fera **sans aucune intervention dans le code source**, mais nécessitera une **compilation paramétrée** (cf. l'option `-g` pour «brancher» `gdb`).

Fonctionalités

L'analyse portera au minimum sur les fonctions d'allocation/libération (`malloc, calloc, realloc, free`) – dans `<stdlib.h>` – mais pourra être étendue aux fonctions de manipulation de blocs mémoire (`memset, memcpy, memmove`) – dans `<string.h>`.

Comme toute "option" de *debug*, cette bibliothèque est destinée à être utilisée dans une phase de développement / correction de projet. A ce titre elle devra pouvoir être activée/désactivée à la compilation (en ligne de commande ou via un `Makefile`).

Son usage ne devra pas modifier le comportement du programme cible, sauf éventuellement pour surmonter quelques bugs et défauts connus de certaines de ces fonctions (cf. exemple 2 ci-dessous).

Son rôle sera de :

- ▷ repérer et comptabiliser les appels aux fonctions d'allocation / manipulation de mémoire,
- ▷ évaluer la quantité de mémoire globalement allouée / libérée,
- ▷ vérifier les libérations correspondantes,
- ▷ repérer et surmonter quelques *bugs* classiques.
- ▷ tester quelques débordements éventuels
- ▷ produire un "rapport" final

Quelques exemples illustratifs

Le court programme suivant alloue un bloc de 1 octet, affiche l'adresse et la valeur pointée, mais ne libère pas la zone alouée. Il se compile sans erreur ni `warning` et s'exécute sans problème, mais rien n'indique que la mémoire n'a pas été libérée.

utest_malloc.c

```c
# include <stdio.h>
# include <stdlib.h>

int main(void)
{
  char* ptr=(char*)malloc(1);
  fprintf(stderr,"adresse : %p -> valeur : %d \n",ptr,*ptr);
  return  0;
}
```

xterm

```
$> gcc -Wall utest_malloc.c -o
utest_malloc
$> ./utest_malloc
   adresse:0x55e852da42a0 -> valeur:0
$>
```

☞ moyennant une légère adaptation de la directive de compilation (détaillée plus loin), l'exécution de ce même programme (toujours compilé sans erreur ni `warning`) produira quelque chose comme :

xterm

```
$> ./utest_malloc
   in file <utest_malloc.c>
     in function <main>
       at line <006> : 0x5581bf72b2a0=malloc(1)
   adresse:0x55e852da42a0 -> valeur:0
     summary :
       @0x5581bf72b2a0 : 1 octet allocated (src/utest_malloc.c|main|6) not freed
```

Autre exemple :

Cet autre programme aligne plusieurs instructions "fausses" ou "mal fichues" (lignes `08,09,11`) mais se compile toujours sans erreur ni `warning`. En revanche, cette fois l'éxécution se passe très mal.

utest_free.c

```c
# include <stdio.h>
# include <stdlib.h>

int main(void)
{
  int*  ptr1=malloc(1);
  int** ptr2=&ptr1;
  free(*ptr2);
  free(ptr1);
  *ptr2=NULL;
  free(ptr2);
  return  0;
}
```

xterm

```
$> gcc -Wall utest_free.c -o utest_free
$> ./utest_free
   free(): double free detected in tcache 2
   Abandon (core dumped)
$>
```

Là encore l'usage de notre bibliothèque permettra de mieux cerner les problèmes mais également d'éviter l'erreur d'exécution :

xterm

```
$> ./utest_free
     - (libmtrack) activation automatique > stderr -
   ------------------------------
   Thu Jul 11 12:07:01 2024
   USER : moi
   HOST : d311-5530
   DIR. : /home/machin/truc/Code/
   ------------------------------
   ligne 006, fonction <main> de <utest_free.c> malloc(1)->0x559bd6078750
   ligne 008, fonction <main> de <utest_free.c> free(0x559bd6078750)
   ligne 009, fonction <main> de <utest_free.c> free(0x559bd6078750) - ERREUR : "double free" détecté
   ligne 011, fonction <main> de <utest_free.c> free(0x7fffb8e7b118) - ERREUR : adresse d'entrée illégale
   ------------------------------
   bilan
   malloc : 1 appel
   free   : 1 appel correct
   free   : 2 appels incorrects (ignorés)
   total alloue 1 octet
   total libere 1 octet (100.0%)
```

Le format et le contenu des informations affichées est largement adaptable et rien n'est imposé du moment que les fonctionalités attendues sont présentes et que les informations données sont pertinentes.

Une autre version de cette bibliothèque, sur le même programme, pourra donner :

xterm

```
$>./utest_free
  in file <utest_free.c>
    in function <main>
      at line <006> : 0x56053af6c2a0=malloc(1)
      at line <008> : free(0x56053af6c2a0) -> OK
      at line <009> : free(0x56053af6c2a0) -> error : address already freed (src/utest_free.c|08|main) → ignored
      at line <011> : free(0x7ffdeb00eef8) -> error : address not listed → ignored
    summary :
      0x5581bf72b2a0 : 1 octet allocated (src/utest_min.c|main|7) -> freed(src/utest_free.c|main|08)
```

Ces deux versions, déjà assez élaborées, annoncent en plus les noms de fichier et de fonction, ainsi que le numéro de ligne où l'appel est réalisé. Elles sont également capables de détecter et surmonter certaines "fausses manoeuvres".

Cahier des charges

Il s'agit de développer une bibliothèque partagée utilitaire et "transparente". En conséquence, si les choix de développement sont entièrement à votre charge, le *livrable* doit respecter un cahier des charges assez rigoureux pour assurer un bon fonctionnement de la bibliothèque.

Conditionnement

- interface : fichiers d'en-tête (`*.h`) et bibliothèque partagée (`lib*.so`) accessibles
- branchement : automatique via fichier `Makefile` et/ou commande `shell` (compilation directe).
- installation : par script `shell`
- portée de l'analyse : l'ensemble des dépendances compilées et assemblées
- rapport d'analyse : sur sortie standard et/ou sur fichier texte (pas de format précis).
- tests unitaires : un jeu de courts programmes de test permettant d'illustrer les fonctionalités.
- README : un fichier expliquant comment utiliser votre lib. (installation, compilation, exécution, tests).

Fonctionalités

- statistiques : comptabiliser les appels aux fonctions de gestion mémoire, évaluer les quantités de mémoire allouée et libérée, le taux de recyclage des blocs d'allocation, leur taille moyenne ...
- bilan : dresser un bilan des statistiques précédentes, détecter les blocs mémoires non libérés en indiquant leur emplacement
- debug : repérer et signaler systématiquement les échecs et *bugs* classiques liés à l'utilisation de la fonction `free` : libération sur adresse nulle, illégale, ou déjà libérée (`"double free corruption"`) et faire en sorte que le programme puisse surmonter ces problèmes (éviter la `seg. fault`).
- mode *verbeux* : tracer tous les appels en indiquant le fichier d'appel, la fonction d'appel, le numéro de ligne, ainsi que :
  - pour les fonctions d'allocation : la taille du bloc, l'adresse de retour, le taux de recyclage
  - pour la fonction de libération : l'emplacement de l'appel d'allocation, le signalement des bizarreries éventuelles (allocation et libération dans des fichiers différents, par exemple).
  - pour les fonctions de déplacement de bloc : les tailles et adresses des blocs source et destination.
- sortie : par défaut la sortie (affichages) se fera sur le fichier d'erreur standard (`stderr`) mais pourra être redirigé vers un fichier texte conservé en mémoire.

  Dans ce cas, on devra y faire figurer quelques informations supplémentaires :
  - date de création du programme hôte (date de compilation)

- date d'exécution
- nom de l'utilisateur (`USER`) et de la machine (`HOST`)
- condition d'utilisation : programme hôte avec la liste d'arguments
- paramètres du traceur
- ... et tout ce que vous jugerez utile ...

Mise en œuvre

Les fonctions clônes

Le plus facile dans ce projet est la réécriture des fonctions clônes. A ce niveau, la seule difficulté réside dans la structure de données à adopter pour engranger les informations collectées : il est impossible de prévoir la quantité de données qui devra être stockée (imaginez le cas extrême d'une allocation dans une boucle infinie... ou testez vos projets précédents...). Il faudra donc prévoir un mécanisme de "protection" pour gérer ce genre de situation.

Il faudra donc disposer d'une structure de données suffisament souple pour croître facilement sans multiplier les allocations (ça serait contre-productif !!!) et prévoir une limite au-delà de laquelle le traceur se désactive de lui-même (s'il doit en arriver là c'est sans doute que le programme hôte est à revoir !).

> Il faudra bien entendu faire en sorte que le traceur ne s'auto-analyse pas : s'il fait usage de fonctions de manipulation de mémoire, celles-ci ne **doivent pas** être remplacées par leur clônes à la compilation.

Les fonctions `free` et `realloc` nécessiteront une attention particulière du fait de leurs défauts de conception (source de nombreuses `Seg.Fault`):

- `free(ptr);` ne fait aucune vérification sur la validité de l'adresse passée en paramètre.
- `free(ptr);` libère la zone pointée par `ptr` mais garde l'adresse (devenue invalide) dans `ptr`,
- `dst=realloc(src,size);` ne fait pas de vérification sur la validité de l'adresse contenue dans `src` et si `dst≠src`, le pointeur `src` contient en sortie une adresse devenue invalide (même défaut que `free`).

La portée des données, variables, fonctions...

C'est bien sûr un point crucial de ce projet :

- le traceur doit pouvoir extraire un maximum d'information du programme hôte mais ne doit en aucun cas perturber son fonctionnement (sauf pour surmonter certains problèmes et éviter les erreurs).
- son fonctionnement doit être transparent et son architecture invisible de l'extérieur et il doit rester facilement paramétrable.

Tests unitaires

Un test unitaire correspond généralement à la mise en œuvre d'un environnement d'exécution le plus simple possible permettant de tester une fonctionnalité bien précise du programme.

Puisque la bibliothèque développée ici à pour but d'analyser l'exécution d'un autre programme, ces tests devront évidemment être de petits programmes compilables et exécutables.

Par exemple, le code test minimal donné en p.1 peut servir de test unitaire pour :

- l'activation automatique du traceur
- le traçage de la fonction `malloc`
- la détection des blocs non libérés.

Environnement / Installation

Cette partie dépasse un peu le cadre de la simple programmation en langage `C` et nécessitera quelques manipulations supplémentaires comme l'écriture de `Makefile` et la définition de variables d'environnement et commandes de compilation adaptées (fichier de configuration `~/.bashrc`).