Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Traceur d'allocation - «patch» à la compilation

L'objectif de ce mini-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» en détournant les fonctions de gestion de mémoire vers des clônes permettant la récolte d'informations à l'éxécution.

Il ne s'agit pas de redéfinir les protocoles de gestion de mémoire de `malloc` *et* `free`*, mais simplement de les surcharger : les clônes (ou* patch*) feront bien appel aux fonctions* `malloc` *et* `free` *de la* `libC`*.*

Le branchement se fera sans aucune intervention dans le code à analyser : c'est à la compilation que ce que se fera le détournement des fonctions à tracer. C'est donc le process de compilation (pré-processeur → compilation → édition de liens) qui va prendre un rôle essentiel.

Pour en arriver là, quelques étapes seront nécessaires :

① *mise en place d'une architecture de programme adaptée : branchement sans intervention dans le code.*
② *développement des fonctions cônes et table d'enregistrement, activation / arrêts automatiques*
③ *finalisation et déploiement en* librairie partagée *(*`shared object`*)*

Présentation générale du projet

Des outils comme `Valgrind` ou `gprof` sont des outils très puissants – parfois trop – pour détecter les problèmes de mémoire. Leur principal défaut est qu'ils agissent directement sur l'*exécutable* `a.out` : ils analysent 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) et toujours documenté.

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 détournant 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. 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 de gestion de mémoire,
- ▷ évaluer la quantité de mémoire globalement allouée / libérée,
- ▷ 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

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  int main(void)
05  {
06    char* ptr=(char*)malloc(1);
07    *ptr = 'x';
08    fprintf(stderr,"adresse : %p -> valeur : %d \n",ptr,*ptr);
09    return 0;
10  }
```

xterm

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

☞ moyennant une re-compilation adaptée, 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 : x
     summary :
       @0x5581bf72b2a0 : 1 octet allocated (src/utest_malloc.c | main() | l.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

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  int main(void)
05  {
06    int*  ptr1=malloc(1);
07    int** ptr2=&ptr1;
08    free(*ptr2);
09    free(ptr1);
10    *ptr2=NULL;
11    free(ptr2);
12    return 0;
13  }
```

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 (ici avec un autre format de sortie) :

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 (version finale)

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

Fonctionalités

- statistiques : comptabiliser les appels aux fonctions de gestion mémoire, évaluer les quantités de mémoire allouée et libérée ...
- contextuelle : pour chaque appel comptabilisé, identifier le module, la fonction et la ligne d'appel dans le code cible.
- 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`).
- 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 ...

Conditionnement (livrable)

- interface : fichiers d'en-tête (`*.h`) et bibliothèque partagée (`lib*.so`) accessibles
- branchement : automatique via fichier `Makefile` et/ou commande `shell` (compilation directe).
- activation : automatique sans action sur le code à analyser : simple re-compilation paramétrée.
- 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).