

Introduction à Valgrind

Mechri Moncef aka Fireboot

7 Novembre 2008

1 Introduction

L'allocation dynamique de mémoire joue un rôle important dans la programmation en C/C++. Ces langages ne fournissant aucun mécanisme de vérification (Il n'y a par exemple pas de ramasses-miettes en C/C++), il incombe au programmeur de prendre soin de ne pas commettre d'erreurs de gestion de la mémoire.

Ces erreurs peuvent être de nature différente (Lecture/écriture dans une zone déjà désallouée ou non allouée, désallocation d'une zone déjà désallouée, fuite de mémoire, ...), ce qui les rend d'autant plus difficiles à débuserquer.

C'est ici qu'intervient Valgrind, un outil permettant le "débugage" de la mémoire, capable de trouver bon nombre d'erreurs et d'indiquer avec précision au programmeur leur nature et leur position. En fait, Valgrind permet bien plus que ça, puisqu'il offre un large panel d'outils pour aider le développeur à améliorer l'efficacité de son code, et même la possibilité de développer des modules.

Cependant, seul Memcheck, le module d'analyse de la gestion de la mémoire sera étudié dans cet article. Pour plus d'informations sur les autres modules, et même sur Memcheck, veuillez vous référer à la documentation officielle qui est très bien fournie.

Valgrind est disponible pour les plates-formes suivantes : x86/Linux, AMD64/Linux, PPC32/Linux et PPC64/Linux. Des ports pour *BSD existent, mais ne sont pas terminés. Pour l'obtenir, utilisez votre gestionnaire de paquet, ou rendez vous sur le site officiel pour le compiler depuis les sources.

Les exemples de cet article ont été faits sous GentooLinux x86, gcc 4.2.3, Valgrind 3.3.0. Ils sont pour la plupart écrits en C, mais le fonctionnement est strictement équivalent sur du code C++.

2 Utilisation de Valgrind

2.1 Préambule à l'utilisation de Valgrind

Pour que Valgrind soit le plus verbeux possible, votre programme doit être compilé en utilisant l'option `-ggdb` (ou `-g`) de GCC. De plus, il est fortement conseillé de désactiver toutes les optimisations qu'effectue le compilateur, en ajoutant `-O0` sur la ligne de compilation. Cette mesure permet de réduire le nombre de faux-positifs à une valeur proche de 0.

L'invocation de Valgrind s'effectue comme suit :

```
valgrind --leak-check=yes /chemin/vers/votre/executable
```

2.2 Détection des fuites de mémoire

Les fuites de mémoire (*memory leak*) sont des zones de mémoire allouées avec `malloc()` (ou `new/new[]` en C++) qui n'ont pas été désallouées à temps, et qui sont donc définitivement perdues. Etudions un exemple basique :

```
1 #include <stdlib.h> /* malloc() */
2
3 void foo(void)
4 {
5     /* Ou, en C++ : char* bar = new char[50]; */
6     char* bar = (char*) malloc(50);
7
8     /* Traitement... */
9 }
10
11 int main(void)
12 {
13     foo();
14     /* Traitement... */
15     return 0;
16 }
```

Analysons cet exemple :

- Le programme débute, la fonction `main()` est exécutée. Elle réalise un appel à la fonction `foo()`, qui alloue dynamiquement (sur le tas) 50 char's (soit 50 octets), et crée le pointeur "bar" qui permet d'accéder à cette zone.
- Au retour de la fonction `foo()`, le pointeur `bar`, objet local sur la pile, est automatiquement détruit. En revanche, la zone allouée dynamiquement n'est pas libérée, mais, faute de pointeurs, elle n'est définitivement plus accessible. Elle est donc perdue jusqu'à l'arrêt de l'ap-

plication. C'est précisément ce phénomène qui est appelé "fuite de mémoire" et qui a pour conséquence l'augmentation de l'espace mémoire qu'utilise le programme.

Analysons maintenant la sortie de Valgrind :

```
fireboot@My_World ~ $ gcc -g -O0 -o exemple exemple.c
fireboot@My_World ~ $ valgrind --leak-check=yes ./exemple
...
==4260== 50 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4260==    at 0x4022AB8: malloc (in /usr/lib/valgrind/
    x86-linux/vgpreload\_memcheck.so)
==4260==    by 0x80483C5: foo (exemple.c:6)
==4260==    by 0x80483E0: main (exemple.c:13)
```

Ce qui est entre "==" est le PID du processus (son identifiant au sein du système d'exploitation).

La première ligne nous informe que 50 octets ont été perdus.

La deuxième ligne est un "détail d'implémentation" (dixit le manuel), et est à ignorer.

La suite nous informe que la zone non désallouée a été allouée dans le fichier exemple.c, à la ligne 6, dans la fonction foo(), appelée par la fonction main(), dans le fichier exemple.c, à la ligne 13.

Attention : Valgrind reporte simplement l'endroit où la zone non désallouée a été allouée. Il ne dit pas à partir de quel endroit cette zone est définitivement perdue (i.e. : à quel endroit le pointeur permettant d'y accéder est détruit).

Bien, maintenant que nous savons où est allouée la future fuite de mémoire, il faut encore corriger le problème, et la désallouer avant la destruction du pointeur. Pour ceci, on utilise respectivement la fonction free(), delete ou delete[] pour une zone allouée avec malloc(), new ou new[].

```
1 #include <stdlib.h>
2
3 void foo(void)
4 {
5     /* Ou, en C++ : char* bar = new char[50]; */
6     char* bar = (char*) malloc(50);
7
8     /* Traitement... */
9
10    /* Ou, en C++ : delete [] bar; */
11    free(bar);
```

```

12 }
13
14 int main(void)
15 {
16     foo ();
17     /* Traitement... */
18     return 0;
19 }

```

Regardons maintenant la nouvelle sortie de Valgrind :

```

==4695== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 7 from 1)
==4695== malloc/free: in use at exit: 0 bytes in 0 blocks.
==4695== malloc/free: 1 allocs, 1 frees, 50 bytes allocated.

```

Elle nous dit qu'aucune erreur n'a été rencontrée, qu'il n'y a eu aucune fuite de mémoire durant l'exécution, qu'il y a eu une allocation, autant de désallocation, et que 50 octets ont été alloués. L'exécution de Valgrind se termine sur ce message glorifiant pour un programmeur :

```

==4695== All heap blocks were freed -- no leaks are possible.

```

2.3 Détection des mauvaises utilisations des pointeurs sur tableaux

Un autre problème récurrent en C/C++ est le dépassement de capacité de tableaux. Concrètement, il s'agit d'une tentative d'écriture au delà de la capacité du tableau. Analysons un exemple de ce type d'erreurs :

```

1 #include <stdlib.h>
2
3 int main (void)
4 {
5     int* foo = (int*) malloc(5 * sizeof(int));
6     foo[5] = 10;
7     return 0;
8 }

```

A la ligne 5, nous créons un tableau dynamique de 5 éléments de type int. Les indices de ce tableau vont donc de 0 à 4 (et non pas 5!).

A la ligne suivante, nous essayons d'atteindre l'élément d'indice 5 (soit le 6ème élément). Or, nous venons de dire que ce tableau ne contient que 5 éléments. Nous tentons donc d'écrire dans une zone mémoire qui n'appartient plus au tableau.

Le comportement suscité par cette action est indéfini : il varie selon la donnée qui a été écrasée.

Voyons maintenant comment Valgrind nous aide à détecter cette erreur. Nous l'invoquons avec les mêmes options que dans le sous-chapitre précédent, soit :

```
fireboot@My_World ~ $ valgrind --leak-check=yes ./exemple
[...]

==11461== Invalid write of size 4
==11461==    at 0x80483DA: main (foo.c:6)
==11461== Address 0x416e03c is 0 bytes after a block of size 20 alloc'd
==11461==    at 0x4022AB8: malloc (in /usr/lib/valgrind/
    x86-linux/vgpreload\_memcheck.so)
==11461==    by 0x80483D0: main (foo.c:5)
```

La troisième ligne de la sortie de Valgrind nous informe que nous tentons d'écrire immédiatement après notre tableau de 20 octets. Si nous avons tenté d'écrire au rang 6 de ce tableau, nous aurions eu un message du type "... is 4 bytes after a block of size 20 alloc'd", qui signifie que nous tentons d'écrire 4 octets plus loin après la fin du tableau - rappelons qu'un int, sur une machine x86, occupe 4 octets.

Les deux premières lignes nous informent ensuite qu'à la ligne 6 du fichier foo.c, nous avons tenté d'écrire 4 octets (si nous avons simplement tenté de lire cette zone mémoire, nous aurions évidemment eu "Invalid read" au lieu de "Invalid write") après la fin du tableau.

2.4 Détection de l'utilisation de variables non-initialisées

Lorsqu'une variable est créée sans être initialisée, sa valeur est totalement aléatoire. Par conséquent, l'utilisation d'une telle variable comme condition entraîne un résultat imprévisible. C'est pourquoi cette pratique est à bannir.

Comme vous vous en doutez, Valgrind est capable de détecter ce genre d'erreurs. Soit le code suivant :

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int foo;
6     if (foo == 1)
7     {
8         printf("Hello world!\n");
9     }
```

```

10     return 0;
11 }

```

Rien de complexe ici non plus. Le contenu du bloc if n'est exécuté qu'à condition que foo vaille 1. Or, foo n'a jamais été initialisée et prend donc pour valeur ce qui était précédemment dans cette zone mémoire (très probablement autre chose que 1 d'ailleurs).

Voyons ce que nous dit Valgrind :

```

==15552== Conditional jump or move depends on uninitialised value(s)
==15552==    at 0x80483F9: main (exemple.c:6)

```

Cette fois ci encore, Valgrind est très clair concernant le problème et son origine.

2.5 Utilisation de la mauvaise méthode de désallocation

En C++, en plus de la classique méthode `free()` héritée du C, deux opérateurs de désallocation de mémoire ont fait leur apparition : `delete` et `delete[]`. Il est aisé de savoir laquelle de ces 3 méthodes il faut utiliser :

- Toute zone mémoire allouée à l'aide de `malloc()` doit être libérée en utilisant la fonction `free()`,
- Toute zone mémoire allouée à l'aide de l'opérateur `new` doit être libérée en utilisant l'opérateur `delete`,
- Toute zone mémoire allouée à l'aide de l'opérateur `new []` doit être libérée en utilisant l'opérateur `delete[]`.

En cas de non concordance de méthode de désallocation, le comportement est indéfini. Là aussi, Valgrind peut nous avertir de la présence d'une telle erreur. Soit le code suivant :

```

1  #include <stdlib.h>
2
3  int main(void)
4  {
5      /* On cree avec malloc() un tableau de 5 int. */
6      int* foo = (int*) malloc(5 * sizeof (int));
7
8      /* Traitement... */
9
10     /* Tentative de désallocation a
11        l'aide de delete[] au lieu de free() */

```

```
12     delete [] foo;
13     return 0;
14 }
```

Ce programme compile parfaitement, même avec le niveau d'avertissement le plus élevé (-Wall sous gcc). Il est pourtant faux et peut ne pas avoir le résultat escompté.

Voici la sortie de Valgrind :

```
==4075== Mismatched free() / delete / delete []
==4075==      at 0x4021EFC: operator delete[](void*) (in /usr/lib/valgrind/
x86-linux/vgpreload\_memcheck.so)
==4075==      by 0x80484C4: main (exemple.c:12)
```

Notez que Valgrind est également capable de détecter les tentatives de désallocation d'un bloc déjà désalloué.

2.6 Limites

Valgrind n'est malheureusement pas l'outil parfait.

Tout d'abord, Valgrind est incapable de détecter les mauvaises utilisations de pointeur dans le cas d'un tableau alloué statiquement (i.e. : sur la pile). De plus, l'exécution de votre programme dans l'environnement de Valgrind sera considérablement plus lente et plus gourmande en mémoire qu'en environnement réel. Cela peut être handicapant si vous travaillez sur un gros programme.

3 Conclusion

Les fonctionnalités de Valgrind présentées ici sont les plus simples et les plus couramment utilisées. Mais il dispose de bien plus de possibilités, toutes documentées dans la documentation officielle (en anglais). Correctement utilisé, Valgrind deviendra un allié fidèle qui vous fera économiser beaucoup de temps, et qui améliorera significativement la qualité de votre programme.

Je remercie M. Didier Mathieu, professeur à l'IUT d'Aix-en-Provence pour sa relecture, ses corrections et ses conseils.

Je tiens à remercier également Deimos, pour sa relecture, ses conseils, la mise en page, et la traduction en L^AT_EX de cet article. Il n'aurait pas été

publié sans lui. Je remercie aussi Overcl0k[] pour sa relecture et ses conseils.

Tous les deux font partie de l'équipe FutureZone. Venez nous rejoindre sur [#futurezone@irc.worldnet.net](https://irc.worldnet.net/#futurezone), vous y trouverez une ambiance chaleureuse et des personnes compétentes.