Valgrind ¶

Plusieurs outils en informatique peuvent vous aider à localiser des bugs dans vos programmes. Parmi ceux-ci, voici deux outils particulièrement utiles pour les problèmes liés à la mémoire :

> **valgrind(1)** vous permet de détecter des erreurs liés à la gestion de la mémoire (**malloc(3)**, **free(3)**,…)
> **gdb(1)** permet de voir ce qui se passe « à l'intérieur » de votre programme et comment les variables évoluent.

Le site **https://www.cprogramming.com/debugging/** vous donne des techniques de débuggage plus détaillées et explique **valgrind(1)** et **gdb(1)** plus en détails.

Valgrind permet de détecter des erreurs liées à la gestion de la mémoire dans vos programmes. Pour utiliser valgrind, lancez la commande **valgrind(1)** avec votre exécutable comme argument:

Parmi les erreurs que valgrind est capable de détecter nous avons:

> Mémoire non-désallouée: Lors d'un appel à **malloc(3)**, vous obtenez un pointeur vers une zone de mémoire allouée. Si vous « perdez » la valeur de ce pointeur, vous n'avez plus le moyen de libérer cette zone de mémoire. Essayez **valgrind(1)** avec le petit programme **`src/nofree.c`**
> Désallouer deux fois la même zone de mémoire: Si vous appelez deux fois **free(3)** sur la même zone de mémoire, **valgrind(1)** va détecter cette erreur. Essayez-le avec le petit programme **`src/twofree.c`**
> Accès en dehors des limites d'une zone mémoire: Si vous allouez une zone de mémoire d'une certaine taille (par exemple un table de 10 chars) et que vous accédez à une adresse qui excède cette zone (par exemple vous accédez au 11ième élément) vous aurez probablement une `Segmentation fault`. Valgrind permet de détecter ces erreurs et indique l'endroit dans votre code où vous faites cet accès. Essayez-le avec le petit programme **`src/outofbounds.c`**

On vous encourage à lancer **valgrind(1)** sur votre projet pour vérifier que vous n'avez pas introduit de memory-leaks sans le vouloir. **valgrind(1)** ne remplace pas une écriture attentive du code mais peut permettre de détecter rapidement certaines erreurs courantes. Vous trouverez plus de détails sur les liens suivants:

> **https://www.cprogramming.com/debugging/valgrind.html**
> **https://valgrind.org**

Les bases de `valgrind`

Commençons par le programme le plus simple possible que nous allons tester à l'aide de **valgrind(1)**:

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

int main(void)
{
   printf("Hello, 1252 !\n");
   return EXIT_SUCCESS;
}
```

Après compilation et l'exécutions avec **valgrind(1)** nous obtenons :

```
$ gcc -o hello hello.c
$ ./hello
Hello, 1252 !
$ valgrind ./hello
==13415== Memcheck, a memory error detector
==13415== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==13415== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==13415== Command: ./hello
==13415==
Hello, 1252 !
==13415==
==13415== HEAP SUMMARY:
==13415==     in use at exit: 0 bytes in 0 blocks
==13415==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==13415==
==13415== All heap blocks were freed -- no leaks are possible
==13415==
==13415== For counts of detected and suppressed errors, rerun with: -v
==13415== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

importantes

Nous pouvons lire dans ce rapport plusieurs informations importante comme le `HEAP SUMMARY` qui résume l'utilisation du tas. Dans notre cas particulier, on peut voir que rien n'a été alloué (en effet, il n'y a pas eu de malloc) et rien n'a été libéré.

L'`ERROR SUMMARY` indique le nombre d'erreurs détectées.

La phrase que nous voulons voir après chaque exécution de **valgrind(1)** est:

```
All heap blocks were freed -- no leaks are possible
```

Ce qui indique qu'aucun memory leak ne peut avoir lieu dans notre programme.

Détecter les memory leaks

A présent nous allons montrer comment détecter des fuites de mémoire dans un programme à l'aide de **valgrind(1)**. Testons le programme **`src/nofree.c`**:

```
$ gcc -o nofree nofree.c
$ valgrind ./nofree
==13791== Memcheck, a memory error detector
==13791== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==13791== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==13791== Command: ./nofree
==13791==
==13791==
==13791== HEAP SUMMARY:
==13791==     in use at exit: 6 bytes in 1 blocks
==13791==   total heap usage: 1 allocs, 0 frees, 6 bytes allocated
==13791==
==13791== LEAK SUMMARY:
==13791==    definitely lost: 6 bytes in 1 blocks
==13791==    indirectly lost: 0 bytes in 0 blocks
==13791==      possibly lost: 0 bytes in 0 blocks
==13791==    still reachable: 0 bytes in 0 blocks
==13791==         suppressed: 0 bytes in 0 blocks
==13791== Rerun with --leak-check=full to see details of leaked memory
==13791==
==13791== For counts of detected and suppressed errors, rerun with: -v
==13791== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Nous remarquons directement que cette fois ci des leaks ont été trouvés par **valgrind(1)**. Celui ci indique en effet la perte de 6 bytes de mémoire sur le tas qui ont été alloués par 1 **malloc(3)** et qui n'ont pas été libérés avant le `return`.

Maintenant nous savons que nous avons un memory leak, mais **valgrind(1)** peut faire plus que seulement les détecter, il peut aussi trouver où ont ils lieu. Nous remarquons dans le rapport qu'il conseil de relancer le test avec cette fois ci l'option `--leak-check=full` pour avoir plus de détails sur notre fuite. Nous avons dés lors de nouvelles informations dans `HEAP SUMMARY` :

```
==13818== 6 bytes in 1 blocks are definitely lost in loss record 1 of 1
==13818==    at 0x4A05FDE: malloc (vg_replace_malloc.c:236)
==13818==    by 0x4004DC: main (nofree.c:5)
```

La fuite a donc lieu à la ligne 5 de notre programme qui correspond à:

```
char *ptrChars = (char *)malloc(6 * sizeof(char));
```

On sait maintenant quel est le **malloc(3)** responsable du leak, et il est facile de l'éviter en écrivant `free(ptrChars);` avant le `return`.

Double free

valgrind(1) ne se contente pas seulement de trouver des memory leaks, il est aussi capable de détecter des doubles free qui peuvent engendrer des corruptions de mémoire. Pour montrer cette fonction de **valgrind(1)** nous utilisons le petit programme **`src/twofree.c`**.

```
$ valgrind ./twofree
==13962== Memcheck, a memory error detector
==13962== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==13962== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==13962== Command: ./twofree
==13962==
==13962== Invalid free() / delete / delete[]
==13962==    at 0x4A0595D: free (vg_replace_malloc.c:366)
==13962==    by 0x40053F: main (in twofree.c:8)
==13962==  Address 0x4c2d040 is 0 bytes inside a block of size 6 free'd
==13962==    at 0x4A0595D: free (vg_replace_malloc.c:366)
==13962==    by 0x400533: main (in twofree.c:8)
==13962==
==13962==
==13962== HEAP SUMMARY:
==13962==     in use at exit: 0 bytes in 0 blocks
==13962==   total heap usage: 1 allocs, 2 frees, 6 bytes allocated
==13962==
==13962== All heap blocks were freed -- no leaks are possible
```

```
==13962==
==13962== For counts of detected and suppressed errors, rerun with: -v
==13962== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

Ici **valgrind(1)** nous indique qu'il a trouver une erreur et qu'il s'agit d'un `Invalid free()` à la ligne 8 de notre programme. Facilement trouvé et corrigé!

Segmentation Fault

Les segmentation faults sont des erreurs courantes lors de la programmation en C/C++. Elles ont lieu lors de l'accès à des zones de mémoire non-allouées. **valgrind(1)** permet de facilement trouver l'origine des segfaults et de les corriger. Démonstration avec **`src/outofbounds.c`**:

```
$ gcc -g -o outofbounds outofbounds.c
```

Il est important de compiler avec le drapeau -g pour dire au compilateur de garder les informations de débuggage.

```
$ ./outofbounds
Segmentation fault
$ gcc -g -o outofbounds outofbounds.c
$ ./outofbounds
$ valgrind ./outofbounds
==14236== Memcheck, a memory error detector
==14236== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==14236== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==14236== Command: ./outofbounds
==14236==
==14236== Invalid write of size 1
==14236==    at 0x400530: main (outofbounds.c:7)
==14236==  Address 0x4c2d04c is 6 bytes after a block of size 6 alloc'd
==14236==    at 0x4A05FDE: malloc (vg_replace_malloc.c:236)
==14236==    by 0x40051C: main (outofbounds.c:5)
==14236==
==14236==
==14236== HEAP SUMMARY:
==14236==     in use at exit: 0 bytes in 0 blocks
==14236==   total heap usage: 1 allocs, 1 frees, 6 bytes allocated
==14236==
==14236== All heap blocks were freed -- no leaks are possible
==14236==
==14236== For counts of detected and suppressed errors, rerun with: -v
==14236== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

valgrind(1) trouve bien une erreur, à la ligne 7 de notre petit programme.

Détecter les deadlocks avec `valgrind`

valgrind(1) contient des outils qui vont au-delà des simples tests de l'allocation de la mémoire. Notamment l'outil `helgrind` permet de détecter des deadlocks. Utilisez `helgrind` sur le petit programme **`./src/thread_crash.c`** en faisant:

```
$ valgrind --tool=helgrind [my binary]

==24314== Helgrind, a thread error detector
==24314== Copyright (C) 2007-2010, and GNU GPL'd, by OpenWorks LLP et al.
==24314== Using Valgrind-3.6.1-Debian and LibVEX; rerun with -h for copyright info
==24314== Command: ./thread_crash
==24314==
==24314== Thread #2 was created
==24314==    at 0x512E85E: clone (clone.S:77)
==24314==    by 0x4E36E7F: do_clone.constprop.3 (createthread.c:75)
==24314==    by 0x4E38604: pthread_create@@GLIBC_2.2.5 (createthread.c:256)
==24314==    by 0x4C29B23: pthread_create_WRK (hg_intercepts.c:257)
==24314==    by 0x4C29CA7: pthread_create@* (hg_intercepts.c:288)
==24314==    by 0x400715: main (in /home/christoph/workspace/SINF1252/SINF1252/2012/S6/src/thread_crash)
==24314==
==24314== Thread #2: Exiting thread still holds 1 lock
==24314==    at 0x4E37FB6: start_thread (pthread_create.c:430)
==24314==    by 0x512E89C: clone (clone.S:112)
```

Plus d'informations sur:

https://valgrind.org/docs/manual/hg-manual.html