Debugging

François Trahay

Contents

Debugging

But: comprendre l’exécution d’un programme

- Pourquoi le programme *crash* ?
- Pourquoi le résultat est `12` ? Ca devrait être 17, non ?
- Pourquoi le programme est bloqué ?

Selon une étude[1], un développeur passe 50 % de son temps à debugger des programmes. Apprendre à debugger efficacement est donc nécessaire si vous souhaitez réduire la durée de cette activité pénible.

[1] T. Britton et al. *Reversible debugging software*. University of Cambridge-Judge Business School, 2013, Technical Report.

Debugging "manuel"

- On part d'un état (supposé) correct du programme
- Ajout de `printf("entrée dans foo(n=%d)\n", n);` dans le code source
- But: suivre le flot d'exécution et l'évolution des variables
- Avantages:
  - Facile à mettre en oeuvre
- Inconvénients
  - Beaucoup de code à modifier
  - Besoin de recompiler
  - Besoin de nettoyer le code après le debugging

Utilisation d'un Debugger

- Exécution du programme controllée par un debugger
- Permet:
  - d'examiner la valeur d'une variable
  - d'exécuter le programme en mode pas à pas
  - de mettre en pause le programme
  - d'insérer des *points d'arrêt*
- Exemple: GDB - The GNU Project Debugger

Exemple d'utilisation de GDB

```
$ ./sigsegv
Debut du programme
Erreur de segmentation

$ gdb ./sigsegv
[...]
(gdb) run
Starting program: ./sigsegv
Debut du programme

Program received signal SIGSEGV, Segmentation fault.
0x0000000000040050b in main (argc=1, argv=0x7fffffffdd68) at sigsegv.c:7
7     *ptr=5;
(gdb) print ptr
```

```
$1 = (int *) 0x0
```

Utiliser GDB

- Pré-requis: compiler le programme avec l'option `-g`
  - Inclue les noms de fonctions/variables dans le binaire pour faciliter le débugging
- Charger le programme dans `gdb`: `gdb ./programme`
- Lancer le programme: `r` (ou `run`)
- Arrêter le programme: `Ctrl+C`
- Quitter gdb: `q` (ou `quit`)

Vous trouverez sur https://www-inf.telecom-sudparis.eu/COURS/CSC4103/Supports/?page=annexe-gdb un récapitulatif des principales commandes `gdb`.

Reverse debugging

`gdb` est capable de faire du *reverse debugging*. En enregistrement ce que fait le programme, `gdb` peut "exécuter" le programme en arrière. Cela peut être utile pour détecter quand une variable a été modifiée.

Si vous terminez le TP rapidement, prenez le temps de regarder les vidéos suivantes qui ouvrent plein de perspectives : Reverse debugging.

Autres debuggers

Il existe d'autres debuggers que `gdb`, mais les principes restent les même. Si vous utilisez un IDE comme VS Code, celui-ci propose sans doute un debugger intégré.

`gdb` peut être étendu avec des sur-couches comme gef, ou avec des interfaces graphiques

---

Examiner l'état du programme

La commande `bt` (ou `backtrace`)

- affiche la pile d'appel
- pour chaque niveau
  - *callsite*: emplacement de l'appel de fonction
  - possibilité d'inspecter les variables locales
- sélection de la *stack frame* `#x` avec la commande `frame [x]`.

```
  (gdb) bt
#0  baz (a=2) at backtrace.c:7
#1  0x0000000000400581 in bar (n=5, m=3) at backtrace.c:15
```

```
#2  0x00000000004005ae in foo (n=4) at backtrace.c:21
#3  0x0000000000400559 in baz (a=5) at backtrace.c:9
[...]
```

La *backtrace* permet d'examiner l'état actuel du processus, ainsi que l'enchaînement d'appels de fonctions qui a mené à cet état.

```
  (gdb) bt
#0  baz (a=2) at backtrace.c:7
#1  0x0000000000400581 in bar (n=5, m=3) at backtrace.c:15
#2  0x00000000004005ae in foo (n=4) at backtrace.c:21
#3  0x0000000000400559 in baz (a=5) at backtrace.c:9
[...]
(gdb) frame
#0  baz (a=2) at backtrace.c:7
7     if(a<=2)
(gdb) print a
$1 = 2
(gdb) frame 1
#1  0x0000000000400581 in bar (n=5, m=3) at backtrace.c:15
15    return baz(m-1);
(gdb) print m
$2 = 3
```

Ici, `gdb` nous indique que le programme est arrêté dans la fonction `baz`, à la ligne 7 du fichier `backtrace.c`. Cette fonction a été appelée (`frame #1`) par la fonction `bar` à la ligne 15. La fonction `bar` a été appelée par `foo` à la ligne 31 (cf la `frame #2`).

En sélectionnant une `frame`, on peut examiner l'état des variables locales au site d'appel.

Etat des variables d'un processus

- commande `p [var]` (ou `print [var]`)
  - affiche la valeur de `[var]`
- commande `display [var]`
  - affiche la valeur de `[var]` à chaque arrêt du programme
- `[var]` peut être une variable (eg. `p n`)
- `[var]` peut être une expression (eg. `p ptr->next->data.n`)

Il est possible de choisir le format d'affichage:

- `p/d [var]` affiche la valeur décimale de `[var]`
- `p/u [var]` affiche la valeur décimale (non signée) de `[var]`
- `p/x [var]` affiche la valeur hexadécimale de `[var]`
- `p/o [var]` affiche la valeur octale de `[var]`

- `p/t [var]` affiche la valeur binaire de `[var]`
- `p/a [var]` affiche la valeur `[var]` sous forme d'adresse
- `p/c [var]` affiche la valeur `[var]` sous forme de caractère
- `p/f [var]` affiche la valeur `[var]` sous forme de flottant

`gdb` peut également afficher la valeur d'un registre. Par exemple `p $eax` affiche la valeur du registre `eax`.

Exécution pas à pas

Une fois le programme lancé, possibilité d'exécuter les instructions une par une:

- `n` (ou `next`) : exécute la prochaine instruction, puis arrête le programme.
- `s` (ou `step`) : idem. Si l'instruction est un appel de fonction, ne descend pas dans la fonction.
- `c` (ou `continue`) : continue l'exécution du programme (arrête le mode pas à pas)

Points d'arrêt

- *"Arrête le programme dès qu'il atteint cette ligne de code"*
  - `b fichier.c:123`
- Permet d'examiner l'état du programme à certains points (exemple: entrée de la fonction buggée)

Après avoir définis les points d'arrêt, on laisse le programme s'exécuter (avec la commande `continue`). Lorsque le programme atteint un des points d'arrêt, le débugger le met en pause et donne la main au développeur afin qu'il puisse examiner l'état du programme.

Par exemple:

```
$ gdb ./programme
[...]
(gdb) b bar
Breakpoint 1 at 0x400569: file programme.c, line 13.
(gdb) b backtrace.c:9
Breakpoint 2 at 0x40054c: file programme.c, line 9.
(gdb) r
Starting program: programme
Debut du programme

Breakpoint 1, bar (n=11, m=9) at backtrace.c:13
13    if(m<2)
(gdb) p n
```

```
$1 = 11
(gdb) p m
$2 = 9
(gdb) c
Continuing.

Breakpoint 2, baz (a=8) at backtrace.c:9
9     return foo(a-1);
(gdb) p a
$3 = 8
(gdb) c
Continuing.

Breakpoint 1, bar (n=8, m=6) at backtrace.c:13
13    if(m<2)
(gdb)
```

Il est également possible de définir des points d'arrêt conditionnels. Par exemple la commande

```
(gdb) b bar if n == 0
```

n'arrêtera l'exécution du programme en entrant dans la fonction `bar` que si `n` est égal à `0`.

Surveiller une variable

- *"Arrête toi dès qu'on modifie la variable [x]"*
  - `watch x`
- Permet de trouver les endroits où une variable est modifiée
  - *"Mais `ptr` ne devrait pas être `NULL` ! Qui a fait ça ?"*

Voici un exemple d'utilisation de la commande `watch`

```
$ gdb ./watch
  [...]
(gdb) watch n
 Hardware watchpoint 2: n
(gdb) c
Continuing.

Hardware watchpoint 2: n

Old value = 0
New value = 1
main (argc=1, argv=0x7fffffffdd68) at watch.c:7
7     for(i=0; i<1000; i++) {
```

```
(gdb) c
Continuing.

Hardware watchpoint 2: n

Old value = 1
New value = 2
main (argc=1, argv=0x7fffffffdd68) at watch.c:7
7     for(i=0; i<1000; i++) {
(gdb) p i
$1 = 17
[...]
```

Valgrind

- Outils de débugging et de profilage
  - Détection de fuites mémoire
  - Utilisation de variables non initialisées
- `valgrind [programme]`

Valgrind peut détecter l'utilisation de variables non initialisées. Par exemple, la non initialisation de `n` dans instructions suivantes est détectée par valgrind:

```c
int n;
printf("%d$\n", n);
```

```
  $ valgrind ./exemple_valgrind
==1148== Memcheck, a memory error detector
==1148== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==1148== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
==1148== Command: ./exemple_valgrind
==1148==
==1148== Conditional jump or move depends on uninitialised value(s)
==1148==    at 0x4E7F2D3: vfprintf (vfprintf.c:1631)
==1148==    by 0x4E86AC8: printf (printf.c:33)
==1148==    by 0x400504: foo (exemple_valgrind.c:6)
==1148==    by 0x400529: main (exemple_valgrind.c:13)
==1148==
==1148== Use of uninitialised value of size 8
==1148==    at 0x4E7C06B: _itoa_word (_itoa.c:179)
==1148==    by 0x4E7F87C: vfprintf (vfprintf.c:1631)
==1148==    by 0x4E86AC8: printf (printf.c:33)
==1148==    by 0x400504: foo (exemple_valgrind.c:6)
==1148==    by 0x400529: main (exemple_valgrind.c:13)
==1148==
```

```
[...]
==1148==
5
==1148==
==1148== HEAP SUMMARY:
==1148==     in use at exit: 0 bytes in 0 blocks
==1148==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==1148==
==1148== All heap blocks were freed -- no leaks are possible
==1148==
==1148== For counts of detected and suppressed errors, rerun with: -v
==1148== Use --track-origins=yes to see where uninitialised values come from
==1148== ERROR SUMMARY: 8 errors from 8 contexts (suppressed: 0 from 0)
```

Valgrind détecte également les *fuites mémoire*. Lorsqu'une zone mémoire allouée avec `malloc()` n'est pas libérée (avec `free()`), la zone mémoire peut être "perdue". L'effet peut être grave si la fuite mémoire survient fréquemment. Par exemple, un serveur web qui perdrait quelques octets lors du traitement d'une requête web, pourrait perdre plusieurs gigaoctets de mémoire après le traitement de millions de requêtes.

Valgrind détecte ce type de fuites mémoire. Pour obtenir des informations sur l'origine de la fuite, on peut utiliser l'option `--leak-check=full` :

```
$ valgrind  --leak-check=full ./exemple_valgrind2
==1572== Memcheck, a memory error detector
==1572== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==1572== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
==1572== Command: ./exemple_valgrind2
==1572==
85823552
==1572==
==1572== HEAP SUMMARY:
==1572==     in use at exit: 1,024 bytes in 1 blocks
==1572==   total heap usage: 2 allocs, 1 frees, 2,048 bytes allocated
==1572==
==1572== 1,024 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1572==    at 0x4C2BBCF: malloc (vg_replace_malloc.c:299)
==1572==    by 0x40054E: main (exemple_valgrind2.c:7)
==1572==
==1572== LEAK SUMMARY:
==1572==    definitely lost: 1,024 bytes in 1 blocks
==1572==    indirectly lost: 0 bytes in 0 blocks
==1572==      possibly lost: 0 bytes in 0 blocks
==1572==    still reachable: 0 bytes in 0 blocks
==1572==         suppressed: 0 bytes in 0 blocks
==1572==
```

```
==1572== For counts of detected and suppressed errors, rerun with: -v
==1572== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Sanitizers

Outre valgrind, vous pouvez utiliser des *sanitizers* pour repérer des soucis dans votre code:

- `AddressSanitizer`: détecte les accès hors-tableau, et l'utilisation des pointeurs après la libération
  - Compilez et linkez votre code avec `-fsanitize=address`
- `Undefined`: détecte divers comportements non-définis, notamment avec les pointeurs
  - Compilez et linkez votre code avec `-fsanitize=undefined`
- `Memory`: détecte des accès à de la mémoire non initialisée
  - Compilez et linkez votre code avec `-fsanitize=memory`

Ces sanitizers sont généralement implémentés dans les principaux compilateurs (gcc, clang, visual C/C++). En fonction de votre compilateurs ou plateforme, il est possible que certains sanitizers ne soient pas disponibles. Consultez le manuel (`man gcc`) !

Pointeurs de fonction

- Toute fonction a une adresse
  - `foo` désigne l'adresse de la fonction `int foo(int a, char b);`
    - l'adresse correspond à l'endroit où est situé le code dans la mémoire
- Un pointeur de fonction[1] désigne l'adresse d'une fonction
- Exemple de déclaration:
  - `int (*function_ptr)(int a, char b) = foo;`
- Utilisation:
  - `function_ptr(12, 'f'); // appelle la fonction foo`

[1] Oui, c'est sans rapport avec le debugging, mais pour équilibrer les séances, nous avons préféré ne pas aborder cette notion lors du cours sur les pointeurs :-)

- La déclaration d'un pointeur de fonction ressemble à la déclaration du prototype d'une fonction dont le nom serait `(*nom_pointeur)`
- Comme un pointeur "normal", un pointeur de fonction peut être initialisé à `NULL`
- Comme pour un pointeur sur `int` qui ne peut pointer que sur une valeur de type `int`, un pointeur sur `int (*f)(double, char, int)` ne peut pointer que sur une fonction dont le prototype est `int f(double, char, int);`

- Une fois initialisé, un pointeur de fonction peut s'utiliser comme une fonction "normale".

Exemple:

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

double add(double a, double b) {
  return a+b;
}

double substract(double a, double b) {
  return a-b;
}

int main(int argc, char**argv) {
  double n, m;
  scanf("%lf", &n);
  scanf("%lf", &m);
  // declare a function pointer named "operation"
  double (*operation)(double, double) = NULL;

  if(n < m) {
    /* operation points to the add function */
    operation = add;
  } else {
    /* operation points to the substract function */
    operation = substract;
  }

  /* call the function pointed to by operation */
  double result = operation(n, m);

  printf("Result of the operation: %lf\n", result);

  return EXIT_SUCCESS;
}
```

On peut définir un type (à l'aide du mot-clé `typedef`) correspondant à un pointeur de fonction. Par exemple:

```c
typedef double (*op_function)(double, double);
```

définit le type `op_function`. On peut donc ensuite déclarer un pointeur de fonction de ce type en faisant:

```c
op_function operation;
```

- Les pointeurs de fonctions sont utilisés pour faire de composants logiciels ou des plugins. Une interface est définie, par exemple:
  - il faut une fonction qui initialise la structure x
  - il faut une fonction qui calcule la structure x
  - il faut une fonction qui affiche la structure x Cela prend généralement la forme d'une structure contenant des pointeurs de fonction nommés *callbacks*:

```
struct component {
  char* plugin_name;
  void (*init_value)(struct value*v);
  void (*compute_value)(struct value*v);
  void (*print_value)(struct value*v);
};
```

Un plugin implémentant ce service allouera la structure et désignera ses fonctions comme callback pour le service.