Debugging

François Trahay

CSC4103 – Programmation système

2019–2020

1 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 ?

1.1 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 œuvre
- Inconvénients
  - Beaucoup de code à modifier
  - Besoin de recompiler
  - Besoin de nettoyer le code après le debugging

2 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

2.1 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.
0x000000000040050b in main (argc=1, argv=0x7fffffffdd68) at sigsegv.c:7
7           *ptr=5;
(gdb) print ptr
$1 = (int *) 0x0
```

2.2 Utiliser GDB

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

2.3 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
[...]
```

2.4 État 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`)

2.5 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)

2.6 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)

2.7 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 ?"*

3 Valgrind

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

4 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 [a] 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`

a. 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 :-)