Programmation Avancée

TP n°5 : Débogage

Simon Forest

18 février 2021

`gdb` est un débogueur (*debugger*), un programme qui permet d'exécuter pas-à-pas un programme pour trouver la cause de bugs et de plantages. Il remplacera avantageusent les « *`printf` de débogage* ».

Pour pouvoir utiliser efficacement `gdb`, il faut avoir préalablement compilé votre programme avec `gcc -g`. Par exemple :
`gcc -Wall -g -o truc truc.c`
Ensuite on lancera `gdb` avec le nom du binaire, dans cet exemple :
`gdb truc`
Cela lance `gdb` mais l'exécution du programme n'a pas encore démarré. `gdb` est un programme interactif, on lui donne des commandes. Ainsi, celle qui lance l'exécution du programme est `run`. On écrit après `run` les arguments de la ligne de commande. Ainsi
`run machin chose`
lancera votre programme avec ces deux arguments, comme si on avait tapé
`./truc machin chose`
dans un terminal. Après la terminaison du programme, refaire un `run` relance le programme avec les mêmes arguments. On peut aussi faire `run > sortie.txt` pour rediriger la sortie standard.

Pour quitter `gdb` utiliser `quit` ou `q`. `help` donnera une aide sur une commande, par exemple : `help run`. Beaucoup de commandes `gdb` peuvent être abrégées ; ainsi la commande `run` est abrégée en `r`.

Point d'arrêt

Un point d'arrêt (*breakpoint* en anglais) est un endroit du programme où `gdb` arrête l'exécution et attend une action de votre part. On peut en placer :

- Sur une fonction, avec la commande `break`. Par exemple pour s'arrêter quand `f()` est appelée :
  `break f`
- Sur une ligne de code donnée, avec `break` aussi. Par exemple pour stopper ligne 10 :
  `break 10`
- Quand une variable est modifiée, avec `watch`. Par exemple pour être averti quand `x` est affectée :
  `watch x`

L'exécution peut être poursuivie par l'une des trois commandes suivantes :

- `cont` ou `c` : aller jusqu'au prochain point d'arrêt
- `step` ou `s` : exécuter juste une ligne du programme
- `next` ou `n` : exécuter aussi juste une ligne du programme, mais en restant dans la même fonction, même si un appel de fonction à lieu. Alors qu'avec `step`, si l'instruction contient un appel de fonction, on est arrêté à la première ligne de la fonction appelée.

On peut effacer tous les *breakpoints* avec `delete`. Pour effacer un seul *breakpoint*, on donne son numéro :
`delete 5`
pour le breakpoint numéro 5. On obtiendra la liste des *breakpoints* par :
`info breakpoints`

Affichage des valeurs

La commande `print` donne la valeur d'une variable ou d'une expression plus complexe, par exemple :
```
print cell->suiv
print argv[1]
print x >= y + 3
```
La commande `info` donnes diverses informations :

- `info line` : numéro de la ligne courante, et emplacement des instructions correspondantes (en langage machine) dans le segment de code
- `info source` : info sur le code source
- `info args` : arguments de la fonction courante (celle en cours d'exécution)
- `info locals` : variables locales de la fonction courante
- `info stack full` ou `bt full` : affiche tout le contenu de la pile, c'est-à-dire les arguments et variables locales des différentes fonctions en cours d'appel. La fonction courante est `#0`, celle qui l'a appelée est `#1`, elle-même ayant été appellée par `#2`, et ainsi de suite jusqu'à `main` qui est la dernière affichée, en fond de pile. La notion de pile sera vue au prochain cours.

Enfin, plus puissante, la commande `explore` est interactive et permet de donner les types et de descendre dans des structures ou des tableaux, par exemple si `u` est un `struct sommet` :
```
explore struct sommet
explore u
```

Exercice 1 : Premier contact avec `gdb`

Considérer le programme suivant :

```c
#include <stdio.h>
int main()
{
  int x;
  int y;
  int i = 0;
  i++;
  scanf("%d",&x);
  i++;
  scanf("%d",&y);
  i++;
  printf("%d",x+y);
  i++;
  return 0;
}
```

1. Compiler le programme et le lancer avec `gdb`. Placer un point d'arrêt sur `main` et lancer l'exécution avec la commande `r`. Le programme s'arrête alors immédiatement au début de `main`. Continuer l'exécution pas à pas jusqu'à la fin en affichant les valeurs de `x`, `y` et `i` régulièrement avec la commande `p`. Quelles sont les commandes possibles pour exécuter pas à pas ici ?
2. Relancer `gdb` en mettant un point d'arrêt cette fois-ci sur la ligne `int i = 0;`. Lancer l'exécution avec `r`. Le programme s'arrête immédiatement à la ligne souhaitée. Placer un watchpoint sur `i` avec la commande `wa i`. Exécuter le programme jusqu'à la fin avec la commande commmande `continue` (ou `c`). À quelles autres lignes le programme s'arrête avant de terminer ? Est-ce cohérent avec le watchpoint demandé ?

Exercice 2 : Segfault

Le programme suivant segfault (probablement) :

```c
#include <stdlib.h>
int f(int x,int *ptr)
{
  int idx = rand();
  ptr[idx] = 42;
}
int main()
{
  int y = rand();
  int tab[4];
  f(y,tab);
  return 0;
}
```

1. Lancer `gdb` avec ce programme et exécuter la commande `r`. Que s'affiche-t-il ? Est-ce que la ligne de code indiquée est bien une ligne problématique ?
2. Afficher la valeur de `x`, `idx` et `ptr` au moment du segfault. Afficher aussi les valeurs des quatre cases de `tab`. Afficher complètement la pile pour voir les variables locales et les fonctions appelées. Est-ce que les informations paraissent cohérentes avec l'endroit où le programme s'est arrêté ?
3. Est-ce que les valeurs de `x` et `idx` sont différentes à une autre exécution ? Le vérifier avec `gdb`. Le cas échéant, que faut-il rajouter pour que les valeurs de `x` et `idx` soient en général différentes à chaque exécution ? Vérifier votre réponse avec `gdb`.

Exercice 3 : Pointeurs et tableaux

Considérer le code suivant :

```c
int main()
{
  int *ptr = malloc(sizeof(int));
  int tab[10];
  *ptr = 0;
  for(int i = 0; i < 10; i++)
  {
    if(i % 3)
```

```
      (*ptr)++;
    tab[i] = i;
  }
  return 0;
}
```

1. Exécuter le programme avec `gdb` en ayant mis un watchpoint sur `ptr` dès que possible avec la commande `wa ptr`. Combien de fois le programme s'arrête ? Est-ce logique ?
2. À la place, mettre un watchpoint dès que possible avec la commande `wa *ptr`. Combien de fois le programme s'arrête cette fois-ci ? Pour quelles valeurs de `i` ?
3. Exécuter le programme avec `gdb` en ayant mis un watchpoint sur `tab` dès que possible avec la commande `wa tab`. Que constate-t-on ?
4. À la place, mettre un watchpoint dès que possible avec la commande `wa *tab@10`. Combien de fois le programme s'arrête cette fois-ci ? Observer l'évolution de `tab` qui s'affiche à chaque tour de boucle.

Exercice 4 : Fonction récursive

1. Écrire un fichier `.c` qui contient une fonction `int fibo(int n)` qui calcule la suite de Fibonacci de façon récursive, et un `main` qui affiche la valeur de `fibo(6)`.
2. Exécuter le programme avec `gdb` en ayant mis un point d'arrêt sur `fibo`. Observer l'évolution de la pile avec la commande `bt full`. Est-ce que l'évolution de cette pile vous paraît cohérent avec votre code ?

Exercice 5 : `valgrind`

Considérer le code suivant :

```
#include <stdlib.h>
int main()
{
  int *ptr = malloc(4);
  for(int i = 1; i <= 4; i++)
  {
    ptr[i] = i;
  }
  return 0;
}
```

1. Exécuter ce code. Est-ce qu'il segfault ?
2. Utiliser `valgrind` pour détecter et enlever toutes les erreurs de corruption mémoire. Le programme doit toujours initialiser un tableau d'`int` de quatre cases à la fin.
3. Est-ce qu'il y a des fuites de mémoire ? Comment cela est indiqué dans `valgrind` ?
4. Rajouter dans le code un appel à une fonction `f` qui fait juste un `malloc` de 4 octets. Relancer le programme avec `valgrind` en rajoutant l'option `--leak-check=full`. Comment le fait qu'il y a une fuite mémoire dans `f` est rapporté ?

Exercice 6 : Petit aperçu de l'ampleur de la bibliothèque standard

1. Lors de l'execution de n'importe quel programme, tapez `info types` pour voir tous les types définis (ils sont définis par des `typedef` des fichiers d'entête inclus, vous voyez ainsi l'effet d'un simple `#include <stdio.h>`).
2. Tapez également `info variables` pour avoir toutes les variables globales. Dur de repérer `stdin` et `stdout` au milieu de *beaucoup* d'autres ! Comment avoir leur type ?
3. La commande `ptype` permet de connaître la définition d'un type. Trouvez quel est le «vrai» nom du type `size_t`. Trouvez quel est le «vrai» nom du type `FILE` et affichez la structure correspondante.