INF301 – Debug de segfaults

Théophile Bastian <theophile.bastian@inria.fr>

Une *segfault* – *segmentation fault*, ou *erreur de segmentation* – apparaît quand un programme essaye d'accéder à une zone mémoire qui ne lui est pas allouée[1]. C'est une erreur fréquente dès lors qu'on manipule des pointeurs, et elle est plus difficile à corriger qu'un bug classique – en tout cas quand on n'a pas les bons réflexes !

Elle arrive par exemple dans les cas suivants :

```c
int *test = NULL;
*test = 42;
```

```c
int *test = NULL;
printf("%d\n", *test);
```

```c
int *test = 215342565;
*test = 42;
```

Dans le premier cas, on essaye d'assigner une valeur à l'adresse mémoire `NULL` – c'est-à-dire, l'adresse 0, ce qui est une erreur. De même, dans le second cas, on essaye de lire cette valeur – c'est tout autant interdit. Dans le troisième cas, on fait de même, mais à une adresse au hasard – les chances que cette adresse corresponde à de la mémoire qui nous est réservée sont infimes !

Prenons l'exemple suivant :

```c
struct liste {
    struct liste* suivant;
    int valeur;
};
typedef struct liste liste_t;

// Alloue une nouvelle cellule de
// valeur `val`
liste_t* alloc_cell(int val) {
    liste_t* cell =
        malloc(sizeof(liste_t));
    cell->valeur = val;
    cell->suivant = NULL;
    return cell;
}
```

```c
// Affiche les `taille` 1ères valeurs de `liste`
void afficher_liste(liste_t* liste, int taille) {
    liste_t* cur = liste;
    for(int pos=0; pos < taille; ++pos) {
        printf("%d\n", cur->valeur);
        cur = cur->suivant;
    }
}
int main(int argc, char** argv) {
    int n = atoi(argv[1]); // premier argument
    liste_t* tete = alloc_cell(1);
    tete->suivant = alloc_cell(2);
    afficher_liste(tete, n);
    return 0;
}
```

Ce programme n'est pas correct si on lui passe `n > 2` : on tente d'afficher plus de valeurs de la liste qu'il n'y en a. Et en effet :

```
$ ./test 3
1
2
Segmentation fault (core dumped)
```

Premier réflexe : `valgrind`

Valgrind est un logiciel analysant les accès mémoire d'un programme. Il peut servir de débuggeur rudimentaire, mais analyse également les *fuites mémoire*. Il s'utilise très facilement : il suffit de rajouter `valgrind` en tête de sa ligne de commande.

```
$ valgrind ./test 3
[...]
==862386== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==862386==  Access not within mapped region at address 0x8
==862386==    at 0x1091CE: afficher_liste (exemple.c:20)
==862386==    by 0x109253: main (exemple.c:29)
[...]
```

[1] À noter que si une segfault est toujours due à un accès à de la mémoire non allouée, l'inverse n'est pas vrai : parfois, un tel accès ne résulte pas en une segfault, et peut par exemple corrompre une autre variable du programme.

Valgrind commence par nous rappeler qu'il s'agit d'une *segfault* : `SIGSEGV` est encore un de ses autres noms[2].

Dans les lignes *avant* ce message (éludées ici), valgrind liste les erreurs non-critiques rencontrées. Bien souvent, ces erreurs correspondent se traduisent par des problèmes incompréhensibles plus tard, et mieux vaut les inspecter de plus près !

Pour une *segfault*, la partie la plus importante est celle recopiée ici. Elle nous indique, de bas en haut, que :

- dans la fonction `main`, fichier `exemple.c`, ligne 29, …
- on appelle la fonction `afficher_liste`, fichier `exemple.c`, et ligne 20…
- on crash (segfault)

La première ligne nous indique la position du bug, les lignes suivantes nous indiquent d'où on vient dans le programme. C'est certes peu d'informations (la ligne de l'erreur seulement – pas la variable concernée, etc), mais c'est déjà mieux que juste `Segmentation fault (core dumped)`, et souvent suffisant.

Si c'est plus compliqué : gdb

Le debugger `gdb` est plus compliqué à utiliser, mais plus puissant. Lorsqu'on l'appelle

```
$ gdb ./test
[...]
(gdb)
```

on entre en *mode interactif* : gdb attend des commandes. Notez qu'on ne **passe pas nos arguments à gdb** : `gdb ./test` et non `gdb ./test 3`.

Il faut tout d'abord dire à gdb de lancer le programme (c'est ici qu'on passe le 3 !)

```
(gdb) run 3
[...]
Program received signal SIGSEGV, Segmentation fault.
0x00005555555551ce in afficher_liste (tete=0x5555555592a0, taille=3) at exemple.c:20
20              printf("%d\n", cur->valeur);
(gdb)
```

`gdb` nous dit qu'on a une *segfault* dans `afficher_liste`, ligne 20 de `exemple.c` – rien de nouveau. Il nous donne les valeurs des arguments de cette fonction (`taille=3`, et une adresse pour `tete`). Mais surtout, *on reste en mode interactif* ! On peut lui demander ce que `valgrind` affichait spontanément, la *backtrace* (savoir d'où on vient) :

```
(gdb) backtrace
#0  0x00005555555551ce in afficher_liste (tete=0x5555555592a0, taille=3) at exemple.c:20
#1  0x0000555555555254 in main (argc=2, argv=0x7fffffffe4b8) at exemple.c:29
```

On peut lui demander d'afficher des valeurs *au moment du crash* :

```
(gdb) print pos
2
(gdb) print cur
(liste_t *) 0x0
(gdb) print &(liste->suivant) # On peut écrire des expressions C compliquées
(struct liste **) 0x5555555592a0
(gdb) print liste->suivant->valeur + 42 - taille  # aussi compliquées qu'on veut
41
```

…ou la même chose, mais dans `main`, au moment de l'appel de fonction :

```
(gdb) frame 1  # le même 1 qu'en début de ligne de `backtrace`
(gdb) print n
3
(gdb) frame 0  # et de retour dans `afficher_liste`
```

Le debugger `gdb` est *beaucoup* plus puissant que ça, avec des dizaines de fonctionnalités : renseignez-vous sur les *breakpoints* par exemple. Il est également très pratique pour débugger des erreurs classiques, et pas seulement des *segfaults*.

[2] C'est en réalité plutôt le nom de la réaction de Linux face à une segfault – *cf.* les *signaux UNIX* si vous êtes curieux · se.