Programmation système IV
Débogage, gestion mémoire et performances

DUT 1re année

Université de Marne La vallée

Qu'est ce qu'un bogue ?

Techniques élémentaires

Le débogueur `gdb`

Analyseur mémoire `valgrind`

Le profiler `gprof`

Qu'est ce qu'un bogue ?

Un bogue désigne toute erreur de programmation qui rend un programme partiellement ou totalement inutilisable. Les exemples les plus courants sont :

- le programme s'interrompt brutalement (tel que l'erreur de segmentation),
- le programme ne se termine pas,
- le programme ne retourne pas le résultat escompté,
- le programme est anormalement lent,
- le programme présente de graves faiblesses de sécurité

Le bogue informatique pour le lancement 501 de la fusée Ariane 5 : coût de 500 millions de dollars

Protection par le système

Les systèmes d'exploitation modernes font leur possible pour limiter les conséquences des erreurs de programmation.

Un programme utilisateur n'accède jamais directement aux ressources matérielles, mais seulement par l'intermédiaire du système.

Le système `UNIX` fait de nombreuses vérifications sur les processus qu'il gère et interdit un grand nombre d'opérations dangereuses.

Armes du système :
Pour protéger l'ordinateur, `Unix` se permet de terminer brutalement des processus ou de rendre certaines commandes sans effet.

Protection par le système

Protection mémoire :
Un processus possède des segments mémoires réservés et ne doit pas accéder à la mémoire réservée aux autres programmes, au système ou encore à la mémoire réservée aux entrées / sorties des périphériques.

Protection des fichiers :
Les fichiers ont des droits et les processus aussi (suivant le programme et l'utilisateur qui lance le programme...). Si vous exécutez un programme censé modifier un fichier ne vous appartenant pas, le système nullifie l'action des modifications tentées.
(exemple : `fopen` retourne un pointeur `NULL` mais ne plante pas en cas d'échec.)

Bogues et sécurité

Trou de sécurité :
Il peut arriver qu'un bogue ne se déclenche que dans des conditions très particulières, jamais réunies en utilisation normale. De ce fait, le bogue est difficile à identifier. Toutefois, un pirate doué commence par réunir les conditions du bogue puis exploite le bogue et peut potentiellement rendre l'utilisation de la machine cible impossible ou prendre le contrôle de la machine cible.

Bogues du système :
Il peut arriver qu'un composant du système soit lui-même bogué. Les conséquences peuvent être très graves car le système a confiance en lui-même et opère sur le matériel avec niveau de privilège maximum (Il ne se bride pas lui-même). Il est important de maintenir son système à jour.

Messages d'erreur classiques

Voici quelques messages d'erreurs classiques envoyés par le système `UNIX` lors de l'exécution de programme écrit en C.

- **`Segmentation fault :`** accès mémoire illégal (interdit par le système). Souvent dû à un dépassement de tableau ou à un déréférencement de pointeur invalide.
- **`Bus error :`** accès mémoire non aligné. Erreur commune sur les processeurs Sparc (problème d'arithmétique sur les pointeurs ou violation du typage).
- **`Floating point exception :`** division (ou modulo) par zéro **dans les entiers**.
- **`Broken pipe :`** écriture dans un tube qui a déjà été fermé.

Bogue silencieux

Certains bogues sont complètement silencieux (exemple : dépassement de capacité des entiers). Voici un programme `test.c` :

```
 1  #include <stdio.h>
 2
 3  int factorielle(int n){
 4    if (n <= 1)
 5      return 1;
 6    return n*factorielle(n-1);
 7  }
 8
 9  int main(void){
10    printf("%d\n", factorielle(17));
11    return 0;
12  }
```

```
lancelot-~>gcc -o test test.c
lancelot-~>./test
-288522240
```

Non initialisation de la mémoire en C

Voici un programme `test.c` sans initialisation des variables :

```
1  #include <stdio.h>
2
3  int main(void){
4    int p;
5    char chaine[10];
6
7    printf("%d␣%s\n", p, chaine);
8
9    return 0;
10 }
```

```
lancelot-~>gcc -o test test.c
lancelot-~>./test
-1219132891 p
lancelot-~>./test
-1218772443 u
lancelot-~>./test
-1218637275 w
lancelot-~>./test
-1218239963 }
```

Erreurs classiques de programmation

Accès mémoire invalide

- Lire une zone accessible au programme ⇒ *sans effet*
- Lire ou écrire dans une zone protégée par le système ⇒ *Segmentation fault*
- Écrire des choses aléatoires dans une zone réservée au programme mais plus utilisée par la suite ⇒ *sans effet*
- Écrire des choses aléatoires dans une zone réservée au programme et utilisée dans la suite du programme ⇒ *effet aléatoire, Segmentation fault possible*

Dans le dernier cas, l'erreur de segmentation est une chance car détecter la localisation du bogue quand tout fonctionne mais que les résultats sont aberrants est très difficile.

Un tour des vérifications à faire

- **Références non définies**
- **Erreur de syntaxe**
- **Variable non initialisée**
- **`Return` manquant**
- **Déclarations implicites**
- **Arithmétique en C**
- **Ordre d'évaluation**
- **Affectations multiples**
- **Opérateurs logiques**
- **Indices dans les tableaux**
- **Chaîne de caractères**
- **Fonctions dangereuses**
- **Pointeurs**
- **Code d'erreur**

Un tour des vérifications à faire

Références non définies : Donner au compilateur les informations suffisantes pour qu'il trouve les bibliothèques (exemple : `-lm` quand on utilise `math.h`)

Erreur de syntaxe : Erreur courante quel que soit le niveau du programmeur. Un bon éditeur peut aider à les éviter (en donnant des couleurs aux parenthèses et accolades par exemple).

Variable non initialisée : Penser à initialiser vos variables avant de les utiliser quand c'est nécessaire.

`Return` manquant : Une fonction dont le type de retour n'est pas `void` doit toujours renvoyer une valeur avec `return`. Sinon, la valeur de retour est aléatoire.

Un tour des vérifications à faire

Déclarations implicites : Ne pas oublier de faire les `#include` nécessaires pour les fonctions externes. Quand les types de fonctions sont inconnus, le type `int` est supposé.

Arithmétique en C : Il faut connaître les faiblesses de l'arithmétique en C comme par exemple la taille maximale des entiers ou flottants. Tous les symboles arithmétiques ont des spécificités (exemple : a % b est du signe de a).

Ordre d'évaluation : Bannir toute écriture dont l'effet dépends de l'ordre d'évaluation!
Exemple : `x = puts("a") + puts("b");`

Affectations multiples : Une instruction ne peut modifier la valeur d'une variable qu'une seule fois au plus!
A bannir : `x = (y=2) + (y=3);`

Un tour des vérifications à faire

Opérateurs logiques : Ordre de gauche à droite, évaluation feignante (on ne met jamais d'effet de bord).
A bannir : `if ((i = 2) || y++)`

Indices dans les tableaux : Un tableau de taille N a des indices définis de 0 à $N - 1$ inclus. Tout appel en dehors produit potentiellement une `Segmentation fault` ou un résultat aléatoire.

Chaîne de caractères : Les fonctions standards supposent que les chaînes de caractères se terminent par le caractère. La non présence de ce caractère final peut provoquer des erreurs.

Un tour des vérifications à faire

Fonctions dangereuses : Le manuel `man` précise que certaines fonctions des bibliothèques standards sont dangereuses. (exemple : `fgets`, `snprintf`, `strncat`, `strncpy` sont les versions sécurisées des fonctions `gets`, `sprintf`, `strcat`, `strcpy`). Elles ont souvent un argument supplémentaire : un entier donnant une taille maximale.

Pointeurs : Grande cause d'erreur surtout quand on utilise la mémoire dynamique (`malloc`, `realloc`, `free`). Ne jamais déréférencer (opérateur *) un pointeur `NULL` ou un pointeur non initialiser.

Code d'erreur : De nombreuses fonctions des bibliothèques retourne des codes d'erreur en cas d'échec (souvent -1 pour les retours de type `int` et `NULL` pour les pointeurs). Il faut utiliser ces codes d'erreurs pour éviter les erreurs (exemple : si un `malloc` échoue, il ne faut surtout pas déréférencer le pointeur qu'il a rendu).

Recherche à la main

La recherche à la main se fait en disposant des drapeaux (texte affiché avec `printf`) aux endroits clés du code qui bogue.

```
1  printf("foo\n");
```

Cette approche est formatrice! On comprend mieux le langage C quand on met vraiment les mains dedans. Ne pas oublier de mettre un retour à la ligne dû à la bufferisation de la sortie standard.

Toutefois, pour de grandes quantités de code, cette approche peut être très fastidieuse. C'est alors qu'on a recours à des outils plus performants de débogage.

Macros de débogage

On peut facilement utiliser des macros du préprocesseur pour déboguer un programme. Notamment, `__LINE__` et `__FILE__` sont deux macros systématiquement remplacer par le préprocesseur par le numéro de la ligne pour la première et le nom du fichier pour la seconde.

Une technique courante consiste à définir une macro `DEBUG_MODE` dont la valeur est ajustée suivant que le programmeur souhaite visionner ou pas les informations de débogage.

Macros de débogage

Exemple de macro d'information pour un fichier `test.c` :

```
1  #include <stdio.h>
2  #define DEBUG_MODE 1
3
4  int main(int argc, char* argv[]){
5    if (DEBUG_MODE == 1){
6      printf("main ligne %d dans le fichier %s ", __LINE__, __FILE__);
7      printf("(%d arguments recus)\n", argc - 1);
8    }
9
10   return 0;
11 }
```

```
lancelot-~>./test foo bla 3.14
main ligne 6 dans le fichier test.c (3 arguments recus)
lancelot-~>
```

Vérification par assertions

En utilisant la bibliothèque `assert.h`, on peut disposer des tests de vérification un peu partout dans son code avant les opérations critiques.

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int main(int argc, char* argv[]){
5    int nominateur, denominateur=0;
6
7    /* assert(denominateur != 0); */
8    printf("%d", nominateur / denominateur);
9
10   return 0;
11 }
```

Sans assertion :

```
lancelot-~>gcc -o test test.c
lancelot-~>./test
zsh: floating point exception (core dumped)  ./test
```

Avec Assertions :

```
lancelot-~>gcc -o test test.c
lancelot-~>./test
test: test.c:7: main: Assertion ‘denominateur != 0’ failed.
zsh: abort (core dumped)  ./test
```

Le Débogueur gdb

Wikipedia :
Un débogueur (ou débugueur, de l'anglais debugger) est un logiciel qui aide un développeur à analyser les bugs d'un programme. Pour cela, il permet d'exécuter le programme pas-à-pas, d'afficher la valeur des variables à tout moment, de mettre en place des points d'arrêt sur des conditions ou sur des lignes du programme

Notre débogueur pour `UNIX` et le langage C sera `gdb`.

On l'utilise pour un programme `text` en compilant `text` avec `gcc` et l'option `-g`. Puis on exécute dans un terminal :

```
gcc -o test test.c -g
gdb ./test
```

Un programme bogué

```
 1  #include <stdio.h>
 2
 3  int factorielle(int n){
 4    if (n <= 1)
 5      return (n/(n/2)); /* Division par zero ici! */
 6    return n*factorielle(n-1);
 7  }
 8
 9  int main(void){
10    printf("%d\n", factorielle(8));
11
12    return 0;
13  }
```

```
lancelot-~>gcc -o test test.c
lancelot-~>./test
zsh: floating point exception (core dumped)  ./test
lancelot-~>gcc -o test test.c -g
```

Un programme bogué

```
lancelot-~>gdb ./test
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/nicolas/test...done.
(gdb)
```

La compilation avec `-g` a introduit plein de drapeaux que le débogueur peut utiliser pour se promener dans l'exécution du programme.

Ici le prompt de `gdb` attends nos instructions.

```
(gdb) break factorielle
Breakpoint 1 at 0x80483ea: file test.c, line 4.
(gdb) run
Starting program: /home/nicolas/test
Breakpoint 1, factorielle (n=8) at test.c:4
4   if (n <= 1)
(gdb) print n
$1 = 8
(gdb) continue
Continuing.
Breakpoint 1, factorielle (n=7) at test.c:4
4   if (n <= 1)
(gdb) continue 5
Will ignore next 4 crossings of breakpoint 1.  Continuing.
Breakpoint 1, factorielle (n=2) at test.c:4
4   if (n <= 1)
(gdb) continue
Continuing.
Breakpoint 1, factorielle (n=1) at test.c:4
4   if (n <= 1)
(gdb) p n
$2 = 1
(gdb) continue
Continuing.
Program received signal SIGFPE, Arithmetic exception.
0x08048407 in factorielle (n=1) at test.c:5
5     return (n/(n/2));
(gdb) quit
```

Commandes usuelles de gdb

commandes	**effet**
`help`	aide intégrée
`run`	commence l'exécution du programme
`continue`	reprend l'exécution du programme (après une pause)
`continue` nb	exécute nb fois continue
`next`	exécute la ligne courante, y compris les appels de fonctions
`next` nb	exécute nb fois next
`step`	exécute la ligne courante, s'arrête aux appels de fonctions
`step` nb	exécute nb fois step
`finish`	exécute jusqu'au retour de la fonction courante
`up`	remonte d'un cran dans la pile d'appels
`down`	descend d'un cran dans la pile d'appels
`bt full`	affiche la pile d'appels complète
`print` expr	affiche la valeur d'une expression expr C
`break` fonction	place un point d'arrêt au début d'une fonction
`break` ligne	place un point d'arrêt au début d'une ligne
`delete`	supprime tous les points d'arrêt
`delete break` n	supprime le point d'arrêt numéro n

Savoir où se trouve une erreur de segmentation

A partir de maintenant, vous devez **absolument** savoir comment trouver la ligne de code qui produit une erreur de segmentation. (2 commandes suffisent...)

```
1  #include <stdio.h>
2
3  int main(void){
4    char* nom="Pierre";
5
6    nom[12] = (char) 123;
7    printf("%s\n", nom);
8
9    return 0;
10 }
```

```
lancelot-~>gcc -o test test.c -g
lancelot-~>gdb ./test
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
...
Reading symbols from /home/nicolas/test...done.
(gdb) run
Starting program: /home/nicolas/test
Program received signal SIGSEGV, Segmentation fault.
0x080483ec in main () at test.c:6
6   nom[12] = (char) 123;
```

L'analyseur de mémoire `valgrind`

Wikipedia :
Valgrind est un outil de programmation libre pour déboguer, effectuer du profilage de code et mettre en évidence des fuites mémoires.

L'objectif principal de `valgrind` est de vérifier que toute la mémoire allouer dynamiquement dans le programme est bien libérée en fin d'exécution.

A chaque `malloc` doit correspondre un `free` plus tard dans le programme. `valgrind` recherche les différences et liste les objets qui n'ont pas été libérés. A votre niveau, on ne fera attention qu'a vérifier que tout à bien été libéré.

L'analyseur de mémoire `valgrind`

Voici un programme `test.c` :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(void){
6     char* nom;
7
8     if ((nom = (char *)malloc(sizeof(char) * 9)) != NULL)
9        strcpy(nom, "Bonjour!");
10    else{
11       fprintf(stderr, "Probleme d'allocation");
12       exit(1);
13    }
14
15    printf("%s\n", nom);
16
17    return 0;
18 }
```

L'analyseur de mémoire `valgrind`

Mon programme `test.c` fonctionne bien :

```
lancelot-~>gcc -o test test.c
lancelot-~>./test
Bonjour!
```

Mais la mémoire allouée n'est pas libérée...

```
lancelot-~>valgrind ./test
==18896== Memcheck, a memory error detector
==18896== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==18896== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==18896== Command: ./test
==18896==
Bonjour!
==18896==
==18896== HEAP SUMMARY:
==18896==     in use at exit: 9 bytes in 1 blocks
==18896==   total heap usage: 1 allocs, 0 frees, 9 bytes allocated
==18896==
==18896== LEAK SUMMARY:
==18896==    definitely lost: 9 bytes in 1 blocks
==18896== Rerun with --leak-check=full to see details of leaked memory
```

L'analyseur de mémoire `valgrind`

Je modifie mon programme `test.c` en ajoutant un `free` :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(void){
6     char* nom;
7
8     if ((nom = (char *)malloc(sizeof(char) * 9)) != NULL)
9        strcpy(nom, "Bonjour!");
10    else{
11       fprintf(stderr, "Probleme d'allocation");
12       exit(1);
13    }
14
15    printf("%s\n", nom);
16    free(nom);
17
18    return 0;
19 }
```

L'analyseur de mémoire `valgrind`

Mon programme `test.c` fonctionne bien et se comporte bien avec la mémoire :

```
lancelot-~>valgrind ./test
==18998== Memcheck, a memory error detector
==18998== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==18998== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==18998== Command: ./test
==18998==
Bonjour!
==18998==
==18998== HEAP SUMMARY:
==18998==     in use at exit: 0 bytes in 0 blocks
==18998==   total heap usage: 1 allocs, 1 frees, 9 bytes allocated
==18998==
==18998== All heap blocks were freed -- no leaks are possible
==18998==
==18998== For counts of detected and suppressed errors, rerun with: -v
==18998== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Le profiler `gprof`

Wikipedia : En informatique, le profilage de code (ou code profiling en anglais) consiste à analyser une application afin de connaître la liste des fonctions appelées et le temps passé dans chacune d'elles, ce afin d'en identifier les parties de code qu'il faut optimiser.

Nous utiliserons avec `UNIX` et le langage C, le profiler `gprof`. Son utilisation implique de compiler les sources en donnant l'option `-pg` au compilateur `gcc`.

Le profiler gprof

Voici un programme méchamment récursif (il créé potentiellement 2 appels à chaque appel de la fonction `fibonacci`) :

```
1  #include <stdio.h>
2
3  int fibonacci(int n){
4    if (n <= 1)
5      return 1;
6    return fibonacci(n-1) + fibonacci(n-2);
7  }
8
9  int main(void){
10   printf("Fibonacci de 20 : %d\n", fibonacci(20));
11
12   return 0;
13 }
```

Le profiler gprof

Le programme fonctionne :

```
lancelot-~>gcc -o test test.c
lancelot-~>./test
Fibonacci de 20 : 10946
```

On voudrait savoir combien d'appels ont été fait en tout pour calculer fibonacci de 20. Pour cela, on utilise `gprof`. On recompile avec l'option `-pg` et on exécute une fois l'exécutable produit.

```
lancelot-~>gcc -o test test.c -pg
lancelot-~>./test
Fibonacci de 20 : 10946
lancelot-~>ls gmon.out -l
-rw-rw-r-- 1 nicolas nicolas 408 mars  13 15:37 gmon.out
```

L'exécution a générée un fichier `gmon.out` qui contient les informations de profilage.

Le profiler gprof

On demande ensuite à gprof de faire un résumé des informations.

```
lancelot-~>gprof test gmon.out > prof
lancelot-~>cat prof
Flat profile:
...
granularity: each sample hit covers 4 byte(s) no time propagated

index % time    self  children    called     name
                                  21890             fibonacci [1]
                0.00    0.00        1/1             main [5]
[1]      0.0    0.00    0.00        1+21890     fibonacci [1]
                                  21890             fibonacci [1]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.
...
```

Pour Fibonacci de 20, la fonction récursive a été appelée 21890 fois.

Le profiler gprof

Pour le projet du répertoire qui charge et affiche un répertoire de 8 fiches.

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        8     0.00     0.00  affiche_date
  0.00      0.00     0.00        8     0.00     0.00  affiche_fiche
  0.00      0.00     0.00        8     0.00     0.00  ajoute_fiche
  0.00      0.00     0.00        1     0.00     0.00  affiche_repertoire
  0.00      0.00     0.00        1     0.00     0.00  charge_repertoire
...
```

Outils de développement en C

Tous les outils présentés dans ce cours (`gdb`, `valgrind` et `gprof`) se comportent très bien avec les grands projets.

Plus le code est long et plus le code est divisé, plus les recherches d'erreur sont fastidieuses et alors les outils deviennent incontournables.

Pour faciliter l'utilisation de ces outils, les `Makefile` avec définition de drapeau pour `gcc` avec `CFLAGS=...` sont donc très important.