Compilation C sous *Unix*

Épisode 3: outils de mise au point
Debugging, traçage et optimisation

Outils de mise au point

Nous étudierons deux outils principaux:

Le debugger : `gdb` pour *gnu*-debugger.

Outil qui permet une analyse interactive des erreurs d'exécution et des comportements des programmes.

Intérêt: méthode rapide et fiable pour analyser les problèmes au moment où ils se produisent. Ceci a pour conséquence un débugging beaucoup plus rapide des programmes.

Note: la version sous `X` de `gdb` s'appelle `xxgdb`.

Le profiler : `gprof` pour *gnu*-profiler.

Outil qui permet une analyse fine des temps passés lors d'une exécution dans chaque fonction du programme avec détection des cycles.

Intérêt: Découvrir quelles sont les fonctions dont l'optimisation fera effectivement gagner sur le temps d'exécution du programme, ou certains problèmes d'exécution engendrant des temps d'exécution anormalement élevés.

La maîtrise de ces deux outils est **absolument nécessaire** pour **tout** informaticien, car elle permet un développement efficace d'un code de bonne qualité.

Outils de vérification du code

Les outils de vérification d'un programme tentent de détecter des sources d'erreur à l'exécution avant la compilation ou à la compilation.

La dénomination de ces sources d'erreur est plus connue sous le nom de **WARNINGS**.

Règle: Il **faut** comprendre le sens d'un **WARNING** pour se permettre de l'ignorer.

Vérification à la compilation

Les options de `gcc` commençant par `-W` permettent de fixer le niveau de vérification.

`-Wall` effectue le niveau de vérification maximal.

Vérification syntaxique `lint` (ou `lclint`)

`lint` est un analyseur syntaxique d'un source `C` plus rigoureux que `gcc`. Il détecte en plus certaines erreurs de sémantique et les constructions "machine dépendant" (i.e. permet d'assurer la portabilité).

Les flags les plus facilement utilisables pour fixer le niveau de vérification sont:

`-weak`	peu de vérifications.
`-standard`	mode par défaut.
`-checks`	vérifications modérément stricte.
`-strict`	vérifications absurdement stricte.

Règle: Toute partie de code n'utilisant pas des spécificités particulières d'une machine `doit` être portable. Un code non portable est un mauvais code.

Outils de vérification du code
exemple avec `gcc -Wall`

bad2.c

```
 1  main() {
 2    double v=0.0,u=0.0;
 3    int    i,n;
 4
 5    n=1;
 6    for(i=1;i<=10;i++) {
 7      v += exp(n);
 8      u += 10/i;
 9      n += 2;
10    }
11
12    printf("%d %lf %lf\n",n,v/9,u/7);
13  }
```

```
$ gcc -Wall bad2.c
bad2.c:2: warning: return-type defaults to ‘int’
bad2.c: In function ‘main’:
bad2.c:8: warning: implicit declaration of function ‘exp’
bad2.c:12: warning: implicit declaration of function ‘printf’
bad2.c:12: warning: use of ‘l’ length character with ‘f’ type character
bad2.c:12: warning: use of ‘l’ length character with ‘f’ type character
bad2.c:13: warning: control reaches end of non-void function
$ ./a.out
21 351.111111 3.857143
```

Outils de vérification du code
exemple avec `lclint`

```
$ lclint -weak bad2.c
LCLint 2.2a --- 04 Sep 96

bad2.c: (in function main)
bad2.c:7,14: Function exp expects arg 1 to be double gets int: n
  Types are incompatible. (-type will suppress message)

Finished LCLint checking --- 1 code error found
```

```
$ lclint -strict bad2.c
LCLint 2.2a --- 04 Sep 96

bad2.c:1,1: Function main declared without parameter list
  A function declaration does not have a parameter list.
  (-noparams will suppress message)
bad2.c: (in function main)
bad2.c:7,14: Function exp expects arg 1 to be double gets int: n
  Types are incompatible. (-type will suppress message)
bad2.c:12,3: Called procedure printf may access file system state,
       but globals list does not include globals fileSystem
  A called function uses internal state, but the globals list for
  the function being checked does not include internalState
  (-internalglobs will suppress message)
bad2.c:12,3: Undocumented modification of file system state possible
       from call to printf: printf("%d %lf %lf\n", n, v / 9, u / 7)
  report undocumented file system modifications (applies to unspecified
  functions if modnomods is set) (-modfilesys will suppress message)
bad2.c:13,2: Path with no return in function declared to return int
  There is a path through a function declared to return a value on
  which there is no return statement. This means the execution may fall
  through without returning a meaningful result to the caller.
  (-noret will suppress message)

Finished LCLint checking --- 5 code errors found
```

Outils d'indentation

Règle: un programme bien indenté (et bien commenté) facilite la lecture du code, et donc sa mise au point.

Emacs

Le mode `C` normalement par défaut pour tout fichier `C`.
indentation automatique avec la touche <TAB>
chargement de mode avec: <TAB>`-x` c-mode

Note: l'utilisation des couleurs contextuelles facilite également beaucoup la lecture.

`indent`

commande `Unix` permettant la mise en forme automatique d'un source.

options:

`-gnu` style *gnu*project.
`-kr` style *Kernighan & Ritchie.*
`-orig` style original *Berkeley.*

De nombreuses options permettent de gérer précisément la mise en forme du code.

Exemple: `bad2.c`

```
main() {
  double v=0.0,u=0.0; int i,n=1;
  for(i=1;i<=10;i++) { v+=exp(n); u+=10/i; n+=2; }
  printf("%d %lf %lf\n",n,v/9,u/7);
}
```

```
$ ident -original bad2.c; cat bad2.c
main()
{
    double          v = 0.0,
                    u = 0.0;
    int             i,
                    n = 1;
    for (i = 1; i <= 10; i++) {
        v += exp(n);
        u += 10 / i;
        n += 2;
    }
    printf("%d %lf %lf\n", n, v / 9, u / 7);
}
```

Méthodes de débuggage

Ces deux méthodes sont utilisables mais **complémentaires**. Ne négliger aucune des deux approches.

Méthode des traces (la plus répandue)

Consiste à parsemer le code de `printf` et de `getchar`, afin de comprendre l'origine du dysfonctionnement.

Inconvénient: peut engendrer un volume de trace très élevé, donc difficile à analyser.

Mise-en-oeuvre: Penser à utiliser les commandes du précompilateur (`#define` et `#ifdef`) pour ajouter/supprimer les traces du code à compiler.

Exemple: la trace activée lorsque le symbole `TRACE` est défini (soit par un `#define`, soit avec `gcc -DTRACE`).

```
int func1(int a) {
   int i,b;
   for(i=b=0;i<100;i++) {
      b += func2(a,b)
#ifdef TRACE
      printf("func1: i=%d b=%d\n",i,b);
#endif
   }
}
```

Debugger

Outil d'aide à la recherche d'erreurs.

Fonctionnalités:

- Indique précisément la ligne du programme qui a provoqué l'erreur d'exécution.
- Analyse des valeurs des variables, des paramètres d'appel des fonctions qui ont conduit à l'erreur.
- Insertion de points d'arrêt, exécution pas à pas du programme.

Mise en oeuvre de gdb

On considère le programme `bugged.c`.

```
int func(int n) {
    int i, a[5]={1,2,3,4,5};
    for(i=0;i<n;i++)
       a[i] += a[i-1]+a[i+1];
    return a[10];
}
```

```
main () {
    int c;
    scanf("%d",c);
    printf("func(%d) = %d\n",
           c,func(c));
}
```

Étape 1

Compiler les modules et le programme principal avec l'option `-g` de `gcc`, en supprimant toutes les options d'optimisation.

```
$ gcc -g -o bugged bugged.c
```

Étape 2

Lancer `gdb` sur l'exécutable. Puis à l'invite `(gdb)`, taper `run` avec les éventuels arguments du programme `C` à débugger.

```
$ gdb ./bugged
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License.
There is absolutely no warranty for GDB.
(gdb) run
Starting program: ./bugged
2
Program received signal SIGSEGV, Segmentation fault.
0x80484a9 in func (n=134518256) at bugged.c:3
3           for(i=0;i<n;i++) a[i] += a[i-1]+a[i+1];
```

Étape 3

Analyse et traçage de l'erreur (cf les principales commandes ci-après).

```
(gdb) print i
$1 = -1073743910
```

Étape 4

Sortie.

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$
```

gdb: commandes d'analyse d'erreur

Listing du code: `list` ou `l`

Syntaxe: `list`
`list` *num*
`list` *fichierC*:*num*

Affichage de la valeur d'une variable: `print` ou `p`

Syntaxe: `print` *nomvariable*

- tout modificateur correct en `C` (i.e. `*a`, `a[i]`, `a+i+1`, `a->val`) est accepté et évalué sur la variable si celle-ci est d'un type compatible.
- le flag `/p` permet d'afficher la variable en tant que chaîne de caractères.

Affichage de la pile des fonctions

`where`	affichage de l'état actuel de la pile.
`up`	monte d'un niveau dans la pile (vers le `main`).
`down`	descend d'un niveau dans la pile (vers la dernière appelée)
`frame` n	se place sur le niveau n ne la pile

```
(gdb) list 2,4
2           int i, a[10]={1,2,3,4,5,6,7,8,9,10};
3           for(i=0;i<n;i++) a[i] += a[i-1]+a[i+1];
4           return a[10];
(gdb) print i
$1 = -1073743910
(gdb) print a
$2 = {4, 9, 16, 25, 36, 49, 64, 81, 100, 119}
(gdb) print a[7]
$3 = 81
(gdb) where
#0  0x80484a9 in func (n=134518256) at bugged.c:3
#1  0x8048510 in main () at bugged.c:10
(gdb) up
#1  0x8048510 in main () at bugged.c:10
10          printf("func(%d) = %d\n",c,func(c));
(gdb) frame 0
#0  0x80484a9 in func (n=134518256) at bugged.c:3
3           for(i=0;i<n;i++) a[i] += a[i-1]+a[i+1];
```

gdb: commandes de traçage

Points d'arrêt

un point d'arrêt (ou **breakpoint**) est un arrêt de l'exécution du programme à un endroit précis permettant de revenir à un mode interactif, et d'analyser ses valeurs et son déroulement.

`break` *nom*	place un breakpoint à l'entrée de la fonction *nom*.
`break` *fichierC*:*num*	place un breakpoint à la ligne *num* du fichier *fichierC*.
`info break`	affiche la liste des breakpoints.
`disable` *i*	désactive le breakpoint numéro *i*.
`enable` *i*	réactive le breakpoint numéro *i*.

Déplacements dans l'exécution

`continue`	continue l'exécution (jusqu'au breakpoint suivant s'il y en a un).
`finish`	termine l'exécution de la fonction courante.
`next` *i*	avance de *i* lignes dans la fonction courante (*i* est optionnel).
`step` *i*	idem `next` mais `step` entre dans les fonctions rencontrées.

Affichage automatique à chaque arrêt

On utilise la commande `disp` dont le fonctionnement est identique à `print`.

```
(gdb) break main
Breakpoint 1 at 0x80484f6: file bugged.c, line 9.
(gdb) run
Starting program: ./bugged

Breakpoint 1, main () at bugged.c:9
9           scanf("%d",c);
(gdb) n
3
10          printf("func(%d) = %d\n",c,func(c));
(gdb) print c
$1 = 134518256
(gdb) n
Program received signal SIGSEGV, Segmentation fault.
0x80484a9 in func (n=134518256) at bugged.c:3
3           for(i=0;i<n;i++) a[i] += a[i-1]+a[i+1];
```

debuggage : le fichier `core`

Comportement par défaut

Lorsqu'un programme est compilé avec l'option `-g`, si le programme "plante" lors de son exécution, un fichier nommé par défaut `core` est créé sur le disque. Celui-ci contient:

- le code exécutable (`text`)
- les données statiques (`data`) et non initialisées (`BSS`).
- la totalité de la mémoire allouée par le programme lors de son exécution jusqu'au plantage.

avec l'ensemble des valeurs qui s'y trouvaient au moment du plantage.

Avantage

inutile de relancer l'exécution sous **gdb** pour tenter de réobtenir le même plantage:

```
$ ./bugged
Segmentation fault (core dumped)
$ gdb ./bugged core
```

permet d'obtenir le même résultat qu'en lançant le programme sous `gdb`. L'avantage principal est que le fichier `core` peut être analysé sur plusieurs sessions.

Inconvénient

le fichier peut être très très gros (attention aux quotas).

Gestion du fichier `core`

La commande `ulimit` permet[1] (entre autre, faire `ulimit -a`) de régler l'utilisation du fichier `core`:

`ulimit -c 0`	ne pas créer de fichier `core`.
`ulimit -c 1000000`	limiter sa taille à 1 millions de blocs.
`ulimit -c unlimited`	ne pas limiter la taille du fichier `core`.

Dès le débuggage terminé, ne pas hésiter à effacer vos fichiers `core`.

[1] Sur certains shell (`sh`, `csh`), ce réglage se fait avec la commande:

```
limit coredumpsize 1000000
```

Optimisation d'un code

Différents outils sont à disposition pour optimiser un code ou étudier les temps d'exécution de celui-ci:

Compilation optimisée

Trois options d'optimisation du code compilé sont disponibles pour `gcc`: `-O`, `-O2` et `-O3` (par ordre de l'importance de l'optimisation).

Notes:

- Il est nécessaire de recompiler tous les modules avec ces options.
- Le gain en temps est loin d'être négligeable (typiquement de 3 à 5).
- Attention, le code optimisé est possiblement très différent du code écrit (disparition de fonctions, changement de l'ordre des opérations); d'où une précaution nécessaire de validation pour les applications sensibles.

Profiler

Il permet d'analyser la fréquence et le temps d'utilisation des fonctions lors de l'exécution d'un programme.

Utilisations principales:

- Détermination des fonctions à optimiser en priorité.
- Détection des comportements anormaux.

La commande `time`

Elle renvoie des informations intéressantes sur le temps d'exécution d'un processus, notamment le temps effectif d'exécution (qui dépend de la charge de la machine) et le temps que le système a effectivement passé à résoudre la tâche.

```
$ time (ls > /dev/null)
real    0m0.015s
user    0m0.010s
sys     0m0.010s
```

Mise en oeuvre du profiler `gprof`

On considère le programme `prog.c`

```
int func3(int n) {
  int i,v=1;
  for(i=0;i<n;i++)
    v += func2(n-1)-func1(n-2);
  return v;
}

int func2(int n) {
  return func3(n)+func1(n-1);
}
```

```
int func1(int n) {
  if (n<0) return 1;
  return func3(n)+func2(n);
}

main(int na, char **av) {
  printf("Val=%d\n",
         func3(atoi(av[1])));
}
```

Étape 1

Compiler les modules et le programme principal avec l'option `-pg` de `gcc`. Cette option permet d'ajouter un suivi du comportement du code.

```
$ gcc -pg -o prog prog.c
```

Étape 2

Lancer le programme de façon habituelle. Une fois le programme terminé, un fichier `gmon.out` contenant le suivi a été créé sur le disque.

```
$ ./prog 8
Val=109601
$ ls
gmon.out  prog      prog.c
```

Étape 3

Analyse du résultat en sortie avec la commande `gprof`. Le résultat de la commande est présenté ci-après.

```
$ gprof ./prog gmon.out
```

Note: L'option `-b` de `gprof` permet de se débarrasser des légendes (signification de chaque champ).

Sortie de `gprof`

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ps/call  ps/call  name
 55.56      0.45     0.45  5188865 86724.17 86724.17  func3
 25.93      0.66     0.21  3489104 60187.37 60187.37  func2
 18.52      0.81     0.15  5278448 28417.44 28417.44  func1

                     Call graph

granularity: each sample hit covers 4 byte(s) for 1.23% of 0.81 seconds

index % time    self  children    called     name
[1]    100.0    0.81    0.00       1+13956416 <cycle 1 as a whole> [1]
                0.21    0.00 3489104             func2 <cycle 1> [4]
                0.15    0.00 5278448             func1 <cycle 1> [5]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00    0.81                 main [2]
                0.81    0.00       1/1           func3 <cycle 1> [3]
-----------------------------------------------
                             1699760             func1 <cycle 1> [5]
                             3489104             func2 <cycle 1> [4]
                0.81    0.00       1/1           main [2]
[3]     55.6    0.45    0.00 5188865         func3 <cycle 1> [3]
                             1789344             func2 <cycle 1> [4]
                             1789344             func1 <cycle 1> [5]
-----------------------------------------------
                             1699760             func1 <cycle 1> [5]
                             1789344             func3 <cycle 1> [3]
[4]     25.9    0.21    0.00 3489104         func2 <cycle 1> [4]
                             3489104             func3 <cycle 1> [3]
                             3489104             func1 <cycle 1> [5]
-----------------------------------------------
                             1789344             func3 <cycle 1> [3]
                             3489104             func2 <cycle 1> [4]
[5]     18.5    0.15    0.00 5278448         func1 <cycle 1> [5]
                             1699760             func3 <cycle 1> [3]
                             1699760             func2 <cycle 1> [4]
-----------------------------------------------

Index by function name
   [5] func1                   [3] func3
   [4] func2                   [1] <cycle 1>
```