Module I3

## ALGORITHMIQUE ET PROGRAMMATION

# Outils pour la programmation en langage C
# GNU Debugger et Make

Camille Coti
camille.coti@iutv.univ-paris13.fr
IUT de Villetaneuse - Département R&T - 2011-2012

# Table des matières

# 1 Débugger avec gdb

GNU gdb exécute un programme dans un contexte particulier afin d'aider à diagnostiquer l'origine d'un problème dans un programme. Il permet d'exécuter le programme pas-à-pas (instruction par instruction) et d'afficher la valeur des variables du programme en accédant à la mémoire de celui-ci.

## 1.1 Compiler

Les programmes destinés à être débuggés doivent être compilés spécifiquement afin de contenir les informations nécessaires au débuggage. Il faut utiliser l'option `-g` dans la ligne de commande de compilation. Si la compilation est faite en plusieurs étapes, cette option doit être activée à toutes les étapes.

Attention aux options d'optimisation trop agressives : comme certaines techniques d'optimisation de code peuvent modifier le code lui-même (notamment les boucles), elles rendent l'analyse des instructions exécutées plus difficiles voire impossible. Si vous compilez pour débugger, privilégiez un niveau d'optimisation bas voire pas d'optimisation du tout. Le but de l'exécution n'étant pas ici la rapidité d'exécution, il est de toute manière inutile d'activer un niveau élevé d'optimisation.

## 1.2 Exécuter dans gdb

On lange gdb avec la commande `gdb` et on lui passe en paramètre l'exécutable que l'on veut débugger (éventuellement avec le chemin pour l'atteindre).

Le programme est alors chargé par gdb. On a alors une invite de commandes. On peut lancer l'exécution avec la commande `run`. Si des arguments doivent être passés à l'appel du programme, on les met à la suite de la commande `run`.

```
1  coti@abidjan:~$ gdb /bin/true
2  GNU gdb (GDB) 7.0.1-debian
3  Copyright (C) 2009 Free Software Foundation, Inc.
4  License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
5  This is free software: you are free to change and redistribute it.
6  There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
7  and "show warranty" for details.
8  This GDB was configured as "i486-linux-gnu".
9  For bug reporting instructions, please see:
10 <http://www.gnu.org/software/gdb/bugs/>...
11 Reading symbols from /bin/true...(no debugging symbols found)...done.
12 (gdb) run
13 Starting program: /bin/true
14
15 Program exited normally.
16 (gdb)
```

On peut alors relancer l'exécution en tapant à nouveau la commande `run`, ou quitter avec la commande `quit`.

Si le programme ne renvoie pas 0, gdb affiche la valeur de retour :

```
1  coti@abidjan:~$ gdb /bin/false
2  GNU gdb (GDB) 7.0.1-debian
3  Copyright (C) 2009 Free Software Foundation, Inc.
4  License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
5  This is free software: you are free to change and redistribute it.
6  There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
7  and "show warranty" for details.
8  This GDB was configured as "i486-linux-gnu".
9  For bug reporting instructions, please see:
10 <http://www.gnu.org/software/gdb/bugs/>...
11 Reading symbols from /bin/false...(no debugging symbols found)...done.
12 (gdb) run
13 Starting program: /bin/false
14
15 Program exited with code 01.
```

Si le programme plante, son exécution s'arrête à l'endroit où il a planté et gdb affiche le signal reçu ainsi que sa signification. On est alors arrêté à l'endroit précis de l'erreur et on a accès à la mémoire du programme à cet endroit précis de l'exécution.

```
1  coti@abidjan:~$ gdb plante
2  GNU gdb (GDB) 7.0.1-debian
3  Copyright (C) 2009 Free Software Foundation, Inc.
4  License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
5  This is free software: you are free to change and redistribute it.
6  There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
7  and "show warranty" for details.
8  This GDB was configured as "i486-linux-gnu".
9  For bug reporting instructions, please see:
10 <http://www.gnu.org/software/gdb/bugs/>...
11 Reading symbols from /users/coti/plante...done.
12 (gdb) run
13 Starting program: /users/coti/plante
14
15 Program received signal SIGSEGV, Segmentation fault.
16 0x080483af in fonction () at plante.c:8
17 8                     tab[i] = i;
18 (gdb)
```

On voit ligne 15 que le programme `plante` a reçu le signal SIGSEGV, correspondant à une erreur de segmentation : le programme a tenté d'accéder à une zone de mémoire qui ne lui a pas été allouée par le système s'exploitation.

### 1.3 Diagnostiquer avec des affichages

La grande utilité de gdb pour chercher des erreurs dans des programmes vient principalement du fait qu'il permet d'accéder à l'espace mémoire du programme qui s'exécute.

On peut afficher la pile d'appels pour savoir dans quelle fonction on se trouve et où cette fonction a été appelée avec la commande `bt` (backtrace) :

```
1  (gdb) bt
2  #0  0x080483af in fonction () at plante.c:8
3  #1  0x080483c5 in main () at plante.c:13
4  (gdb)
```

On voit ici que le problème se trouve dans la fonction `fonction()` du fichier `plante.c`, à la ligne 8. Cette fonction est appelée par la fonction `main()` à la ligne 13 du fichier `plante.c`.

On peut monter ou descendre dans la pile d'appels des fonctions avec les commandes `up` et `down`

On peut regarder le bout de code dans lequel on est, en affichant quelques lignes avant et après la ligne à laquelle on se trouve avec la commande `list`.

```
(gdb) list
3
4       void fonction(){
5               int* tab = NULL;
6               int i;
7               for( i = 0 ; i < 40 ; i++ ) {
8                       tab[i] = i;
9               }
10      }
11
12      int main(){
```

On a vu que l'erreur se situait dans la fonction `fonction()` (à a ligne 8). Regardons le code situé autour de l'appel à cette fonction en montant d'un niveau dans la pile d'appels des fonctions :

```
(gdb) up
#1  0x080483c5 in main () at plante.c:13
13              fonction();
(gdb) list
8                       tab[i] = i;
9               }
10      }
11
12      int main(){
13              fonction();
14              return EXIT_SUCCESS;
15      }
```

On peut retourner dans la fonction en descendant d'un niveau dans la pile :

```
(gdb) down
#0  0x080483af in fonction () at plante.c:8
8                       tab[i] = i;
(gdb)
```

On est maintenant à nouveau dans l'appel de la fonction et on peut accéder aux variables qui sont visibles à l'endroit précis où on se trouve (ligne 8). On peut notamment regarder la valeur de quelques variables avec la commande `print` :

```
(gdb) print i
$1 = 0
(gdb) print tab
$2 = (int *) 0x0
(gdb) print tab[i]
Cannot access memory at address 0x0
```

On voit que `tab`, qui est un pointeur sur entier, a une valeur particulière : l'adresse 0. En effet, le pointeur a été initialisé à NULL et on n'a pas alloué de mémoire entre son initialisation et l'accès au tableau tab[0] = 0;. C'est pourquoi on ne peut pas accéder à tab[0].

### 1.4 Utilisation de points d'arrêt

On peut vouloir examiner l'exécution du programme avant le moment où il plante. Pour cela on définit des points d'arrêt (breakpoints) sur lesquels gdb suspendra son exécution pour, par exemple, nous laisser accéder à la mémoire du programme.

On utilise la commande `break` suivi du numéro de ligne où doit se situer le point d'arrêt. Si le programme provient de plusieurs fichiers, on précise le nom du fichier comme suit : `fichier.c:ligne`. Attention : le point d'arrêt est situé *au début* de la ligne. Elle n'est pas exécutée au moment où le programme est interrompu.

```
(gdb) break 7
Breakpoint 1 at 0x80483a1: file plante.c, line 7.
```

On lance l'exécution avec la commande `run` : elle s'arrête sur le point d'arrêt.

```
Breakpoint 1, fonction () at plante.c:7
7               for( i = 0 ; i < 40 ; i++ ) {
```

On peut regarder la valeur des variables du programme à l'endroit précis du point d'arrêt.

```
(gdb) print i
$1 = -1208270860
```

Comme précisé plus haut, la ligne où est situé le point d'arrêt n'a pas encore été exécutée. La variable i n'a donc pas encore été initialisé.

On peut passer à l'instruction suivante avec la commande `next` :

```
(gdb) next
8                       tab[i] = i;
(gdb) print i
$2 = 0
(gdb) print tab[0]
Cannot access memory at address 0x0
```

La ligne 7 a été exécutée : i a pris la valeur 0. La ligne 8 n'a pas été exécutée, donc on n'a pas encore essayé d'accéder à tab[0]. Cependant, on voit que l'on ne peut pas accéder à tab[0] (situé à l'adresse 0).

On peut continuer l'exécution jusqu'au prochain point d'arrêt avec la commande `continue` :

```
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x080483b6 in fonction () at plante.c:8
8                       tab[i] = i;
```

Notre programme a donc planté sur une erreur de segmentation en tentant d'exécuter l'instruction située à la ligne 8.

On peut lister les points d'arrêt qui ont été définis sur le programme avec la commande `info breakpoints` :

```
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x080483a1 in fonction at plante.c:7
        breakpoint already hit 1 time
```

On voit que l'on a défini un point d'arrêt et qu'il porte le numéro 1. On peut le supprimer avec la commande `delete numéro` :

```
(gdb) delete 1
(gdb) info breakpoints
No breakpoints or watchpoints.
```

# 2 Compiler avec make

GNU make est un outil permettant d'automatiser des suites de commandes, notamment pour compiler des programmes. Les séries de commandes et leurs dépendances sont décrites dans un fichier lu par make lors de l'appel à la commande make.

Les commandes sont exécutées les unes après les autres tant qu'aucune ne retourne d'erreur. Lorsqu'une commande retourne une erreur, la série de commandes est interrompue et la commande make retourne une valeur non nulle (1 ou 2 selon les options choisies).

## 2.1 Appel à make

La commande make lit un fichier où sont décrites les séries de commandes à exécuter. Ce fichier s'appelle le plus souvent `makefile` ou `Makefile` et se trouve dans le répertoire courant. Dans ce cas, on appelle simplement la commande

```
1 $ make
```

Si le nom du fichier est différent ou qu'il se trouve dans un autre répertoire, on précise le chemin vers le fichier avec l'option `-F` :

```
1 $ make -f ../fichier
```

Pour profiter des possibilités offertes par les processeurs multi-cœurs, make peut lancer un processus par commande exécutée. On utilise l'option `-j`, seule pour utiliser autant de processus que de cœurs disponibles sur la machine, ou en précisant le nombre de processus à lancer.

```
1 $ make -j
2 $ make -j 5
```

On peut demander à make d'afficher les commandes qu'il va exécuter si tout se passe bien, sans toutefois les exécuter. Cela permet par exemple de vérifier qu'une compilation va bien se dérouler comme on le souhaite. Pour cela, on utilise l'option `-n` :

```
1 $ make -n
```

## 2.2 Le Makefile

Le Makefile est le fichier décrivant les actions à effectuer. Bien qu'il puisse s'appeler autrement (comme nous l'avons vu dans la section précédente), nous le désignerons ainsi dans la suite.

Un Makefile est constitué d'une série de règles dépendant les unes des autres. Chaque règle a la syntaxe suivante :

```
1 cible: dependances
2         commandes
```

Les dépendances sont les cibles dont dépend la commande à exécuter. Lorsqu'il exécute une cible, make vérifie si les dépendances sont présentes. Si elles le sont toutes, il exécute la ou les commande(s). Si des dépendances sont manquantes, il va d'abord exécuter les règles des dépendances manquantes, puis une fois qu'il les a toutes exécutées (ainsi que leurs dépendances, l'algorithme étant récursif) et que toutes les dépendances sont donc disponibles, il exécute les commandes de la cible.

La première cible exécutée est la cible `all`. C'est le point d'entrée dans le Makefile. On lui donne généralement comme dépendances les exécutables ou les fichiers à produire (par exemple, des bibliothèques).

```
1 all: executable liblib.so
```

On peut avoir plusieurs commandes à effectuer dans une règle donnée. Les commandes sont alors données les unes après les autres, en écrivant une commande par ligne. Chaque commande doit être écrite sur une seule ligne. Si, pour des raisons de lisibilité par exemple, vous avez besoin d'écrire une commande sur plusieurs lignes, les fins de lignes doivent être terminées par \(sauf la dernière).

Make fonctionne également en vérifiant les dates de dernière modification des fichiers, afin de ne pas recompiler des fichiers déjà compilés et dont les dépendances n'ont pas été modifiées depuis la dernière compilation. Pour cela, il compare les dates de dernière modifications des dépendances et de la cible.

Ce fonctionnement encourage la modularité du code. Il est facile de découper le code en plusieurs fichiers sans que la compilation devienne fastidieuse.

Par exemple, imaginons un projet de bibliothèque. Les fonctionnalités de la bibliothèque sont découpées en plusieurs fichiers sources : un fichier pour les fonctionnalités de manipulations de fichiers, un fichier pour des algorithmes, un fichier pour les affichages. Cette bibliothèque a également un fichier d'en-têtes. On souhaite également compiler un programme de test qui appelle des fonctions de la bibliothèque. Le Makefile correspondant ressemblera au suivant :

```
1  all: progtest libmalib.so
2
3  progtest: progtest.o malib.h libmalib.so
4          gcc -o progtest progtest.o -lmalib
5
6  progtest.o: progtest.c malib.h
7          gcc -c progtest.c
8
9  libmalib.so: fichiers.o algos.o affichage.o malib.h
10         gcc  -shared -Wl,-soname,libmalib.so -o libmalib.so \
11                 fichiers.o algos.o affichage.o
12
13 fichiers.o: fichiers.c
14         gcc -c fichiers.c
15
16 algos.o: algos.c
17         gcc -c algos.c
18
19 affichage.o: affichage.c
20         gcc -c affichage.c
21
22 clean:
23         rm *.o progtest libmalib.so
```

La première règle exécutée est à la ligne 1 avec la cible `all`. Elle dépend de `progtest` et de `libmalib.so`. On va à la ligne 3 exécuter la règle correspondant à cette cible.

On voit que `progtest` dépend du fichier objet `progtest.o`, d'un fichier d'en-têtes `malib.h` et de la bibliothèque `libmalib.so`.

Le fichier objet `progtest.o` est compilé à la ligne 6. Il dépend des fichiers sources `progtest.c` et `malib.h`. Si le fichier `progtest.o` est présent et que sa date de dernière modification est postérieure à cette de `progtest.c` et `malib.h`, il n'est pas recompilé. Sinon, la ligne 7 est exécutée pour le compiler.

Selon le même principe, on compile la bibliothèque `libmalib.so` (ligne 9) et ses dépendances (lignes 13, 16 et 19) si nécessaire.

On ne compile que ce qui a été modifié depuis la dernière modification : par exemple, si seul le fichier algos.c a été modifié, on recompile `algos.o` à la ligne 17, `libmalib.so` aux lignes 10-11, et l'exécutable `progtest` (ligne 4). On ne recompile pas `fichiers.o`, `affichage.o` ni `progtest.o` s'ils n'ont pas été modifiés depuis la dernière modification.

On constate que `libmalib.so` étant une dépendance de `progtest`, on la compile si nécessaire en compilant l'exécutable (ligne 3). On n'exécute donc pas la seconde dépendance de la cible `all`, puisqu'elle a déjà été exécutée.

Si une compilation échoue, la suite ne sera pas exécutée et la commande `make` retournera la main à l'interpréteur de commandes.

À la fin de ce Makefile on a une cible particulière : la cible `clean`. Elle n'a pas de dépendance : la commande est exécutée quoi qu'il arrive. On l'utilise pour supprimer les résultats de la compilation. On l'appelle de la façon suivante :

```
1 $ make clean
```

De manière générale, on peut appeler une cible en particulier en la passant en argument de la commande make :

```
1 $ make nomDeLaCible
```

On peut rendre le Makefile encore plus flexible en utilisant des variables. Cela permet de modifier rapidement des paramètres comme le nom du compilateur, les options de compilation, les chemins vers les bibliothèques...

On définit une variable sous la forme `NOM = ...`. On l'appelle entre parenthèses en utilisant le signe $ (dollar), afin d'indiquer qu'il s'agit de la valeur d'une variable.

```
1  CC = gcc
2  OPT = -O1 -g -Wall
3
4  all: programme
5
6  programme: programme.o outils.o programme.h
7          $(CC) $(OPT) -o proggramme programme.o outils.o
8
9  programme.o: programme.c programme.h
10         $(CC) $(OPT) -c programme.c
11
12 outils.o: outils.c programme.h
13         $(CC) $(OPT) -c outils.c
14
15 clean:
16         rm *.o programme
```

On définit ici le compilateur (ligne 1) et les options qui lui sont passées (ligne 2 : -O1 pour utiliser le premier niveau d'optimisation, -g pour activer les fonctionnalités de débuggage, -Wall pour afficher tous les avertissements). On appelle ces variables dans les commandes de compilation, lignes 7, 10 et 13.

Si plus tard on veut modifier le nom du compilateur utilisé ou les options qui lui sont passées, on modifie uniquement la ligne correspondant à la définition de ces variables.