Première séance machine
Notions de C, compilation, débuggage

Ensimag 1A Apprentissage, Logiciel de Base

mai 2013

Tous les programmes et morceaux de code sont disponibles dans le répertoire `c/1b-gdb-valgrind/` de l'archive Git.

1 Premiers pas en C

On commence avec un programme simple, `hello.c` :

```
#include <stdio.h>

int main (void) {
        int age;
        puts("Bonjour,");
        puts("Quel age avez vous ?");
        scanf ("%d", &age);
        printf("Vous avez %d ans.\n", age);
        return 0;
}
```

(la fonction `puts` affiche une chaîne à l'écran, puis passe à la ligne. La fonction `printf` affiche la chaîne donnée en argument, dans laquelle on remplace les `"%d"` par les arguments suivant la chaîne)

Compilez-le, puis exécutez-le :

```
telesun> gcc hello.c -o hello
telesun> ./hello
[...]
```

2 Exploration de la chaîne de compilation

2.1 Compilation

Il s'est en fait passé beaucoup de choses dans les deux commandes précédentes. La première étape est la compilation proprement dite, on peut arrêter `gcc` après cette étape avec l'option `-S` :

```
telesun> gcc -S hello.c
```

Ceci va générer un fichier `hello.s`. Ouvrez-le dans un éditeur de texte. Vous avez devant vous un programme en langage d'assemblage. On trouve dedans :

– Des définitions de données (les chaînes qui apparaissent dans le programme)
– Des instructions. En particulier, on reconnait les appels à la fonction `puts`, qui sont sous la forme `call puts`, entourées d'incantations magiques.
– Des étiquettes, par exemple, `main:`.

A ce stade, le programme ne dit rien sur l'implémentation des fonctions `puts`, `printf` et `scanf`. On se contente de supposer qu'elles existent (on cherchera leur implémentation plus tard, c'est l'édition de liens).

2.2 Assemblage

Le langage d'assemblage décrit précisément chaque instruction à exécuter, mais elles ne sont pas dans un format directement exécutable par le processeur. L'étape suivante est d'assembler les instructions, on peut le faire avec la commande :

```
telesun> gcc -c hello.s
```

Ce qui va créer un fichier `hello.o`. Essayez de l'ouvrir dans un éditeur de texte pour vous convaincre que c'est du binaire illisible. On reconnaît quand même quelques chaînes de caractères. On peut afficher le fichier de manière un peu raisonnable avec des commandes comme :

```
telesun> hexdump hello.o
[...]
telesun> hexdump -c hello.o
[...]
```

Mais des outils spécialisés peuvent aussi extraire quelques données intéressantes du fichier. Par exemple, la commande `nm` liste les symboles définis ou utilisés dans le fichier (l'équivalent des étiquettes, mais après assemblage) :

```
$ nm hello.o
00000000 T main
         U printf
         U puts
         U scanf
```

On voit que le symbole `main` est défini (« T » veut dire « Text », on verra plus tard pourquoi). En revanche, les symboles `printf`, `puts` et `scanf` sont toujours indéfinis (« U » pour « Undefined »).

La traduction du langage d'assemblage vers le binaire garde la quasi-totalité des informations du fichiers assembleur, mais change de représentation. On peut retrouver le texte assembleur avec un désassembleur, par exemple :

```
$ objdump -d hello.o
```

(objdump n'est pas installé par défaut sous Mac OS X. Si vous ne pouvez pas faire cette manipulation, ce n'est pas très grave ...)

2.3 Édition de lien

L'étape suivante est surtout intéressante lorsqu'on fait de la compilation séparée, mais il se passe aussi des choses avec un seul fichier : on « prépare » le fichier, pour qu'il soit prêt à être exécuté :

```
telesun> gcc hello.o -o hello
```

Le fichier généré est un peu plus gros. La commande `nm` donne beaucoup plus de symboles. Par exemple, il y a maintenant un symbole `_start`, qui correspond au point d'entrée de l'exécution (c'est lui qui appellera la fonction `main`, au tout début de l'exécution). L'éditeur de lien a également repéré où se trouveront les fonctions `printf`, `puts`, `scanf` (et il aurait donné une erreur si il ne les avait pas trouvés). On peut voir ça avec la commande `ldd` (ou `otool -L` sous Mac) :

```
telesun> ldd hello
        libc.so.6 => /lib/tls/libc.so.6 (0x40018000)
        /lib/ld-linux.so.2 (0x40000000)
```

On n'utilise que des fonctions de la bibliothèque C standard, donc il y a une référence à `libc.so` (ne vous occupez pas du `ld-linux.so`). Mais sur un exécutable de la vraie vie, c'est un peu plus long. Par exemple, pour Emacs :

```
telesun> ldd /usr/bin/emacs
[...]
```

Il y a plus de monde !

3 Makefile

Lorsqu'il n'y a qu'un seul fichier à compiler, on a vu plus haut qu'une commande suffisait à tout faire (`gcc fichier.c -o executable`), mais les choses se compliquent sur des gros projets : si le projet met 1 heure à compiler, on ne veut pas *tout* recompiler à chaque fois, on veut un outil qui recompile seulement ce qui est nécessaire.

L'archive disponible sur le site du cours contient un répertoire `compilation/`.

Le répertoire contient 3 fichiers sources `.c`, et deux fichiers d'en-têtes `.h`. Un Makefile très simple est fourni, pour l'utiliser, il suffit de taper `make` dans le répertoire du projet. `make` va lire le fichier `Makefile`, regarder les dates de dernière modifications des fichiers sur le disque, et en déduire ce qu'il doit faire. Tapez `make` deux fois. La première fois recompile tout, la seconde est presque instantance.

Maintenant, modifiez, par exemple, le fichier `bonjour.c` pour changer le message affiché à l'utilisateur. Sauvez ce fichier, puis recompilez avec `make` : le fichier est recompilé, et l'édition de lien est refaite. Essayez la même chose en modifiant `bonjour.h` : tous les fichiers qui en dépendent sont recompilés.

4 Utilisation de GDB pour déboguer des programmes

Déboguer des programmes en C est bien plus difficile qu'en Ada ou en Java (en langage d'assemblage, ça sera encore pire ...). D'une part parce qu'il y a plus d'occasion de faire des erreurs, mais aussi parce qu'un

programme écrit en C qui plante ne donne pas le numéro de ligne ou le nom de la fonction dans lequel s'est produit le problème (typiquement, le programme dit juste « Segmentation Fault »).

Un mauvais programmeur ajouterait des `printf` un peu partout dans son code pour essayer de comprendre son exécution, mais cette technique est très inefficace. Heureusement, il existe des outils pour éviter ce calvaire : les débogueurs. Nous allons apprendre à nous servir de `gdb` : « GNU DeBugger ».

Ouvrez le fichier `gdb/gdb-tutorial.c` dans votre éditeur de texte favori, et suivez les instructions pas à pas.

5 Valgrind : Un autre outil bien pratique pour l'aide au débogage

Valgrind est un debogueur mémoire. Il permet d'exécuter un programme en vérifiant les accès mémoires, et permet donc d'identifier des erreurs de programmations qui seraient passées inaperçues « par chance », ou d'expliquer certains comportements étranges d'un programme. Une introduction courte est disponible sur EnsiWiki : `https://ensiwiki.ensimag.fr/index.php/Valgrind`

On vous fournit aussi un module implantant une table de hachage et un programme de test (très incomplet !). Compilez ce programme et exécutez le : il plante avec une segmentation fault.

Vous allez devoir étudier le code du module table de hachage pour comprendre comment il fonctionne et localiser les erreurs. Il s'agit d'erreurs en liaison avec la gestion mémoire : vous avez donc tout intérêt à utiliser le logiciel valgrind pour vous aider !

Indication : il y a 3 erreurs volontaires (et un nombre indéfini de bugs non voulus ;)) dans le module table de hachage.

6 Compilation et exécution des exemples vus en TD

Si vous ne l'avez pas déjà fait, téléchargez les exemples vus en TD puis compilez et exécutez les programmes.

Si le temps le permet, essayez d'exécuter pas-à-pas les programmes dans Gdb. Vous pouvez aussi jouer avec le fichier `gdb/bug.c` dans l'archive, et chercher à corriger le (ou plutôt les) bugs qu'il contient.