Utilisation de gcc sous Linux

Frédéric Ferrandis (frederic.ferrandis@openwide.fr)

Juin 2010

MODIFICATIONS

VERSION	***DATE***	***AUTEUR(S)***	***DESCRIPTION***
1.0	05/07/10	F.Ferrandis	Création

Table des matières

1.Présentation du document

L'objectif de ce document est de présenter (succintement) les mécanismes permettant de compiler un ensemble de fichiers sources en des fichiers objets.

Le compilateur étudié sera GCC, qui est un standard utilisé sous Linux et Windows, et permettant également de faire de la cross-compilation.

Les différentes étapes de compilation seront analysées, ainsi que les différents paramètres qui peuvent être ajoutés. Enfin, seront présentées les options de compilation les plus usuelles, ou pratiques.

2.Présentation de GCC

GCC (Gnu C/C++ Compiler) est un ensemble de programme permettant de générer exécutables, librairies et fichiers objets, à partir d'un ensemble de fichiers sources d'un langage donné. Dans les exemples qui seront donnés ci-dessous, nous partirons du principe que les sources sont écrites en C, mais la plupart du raisonnement serait identique avec du C++.

GCC permet de compiler des programmes écrits en C. De la même façon, pour compiler du c++, le programme utilisé sera généralement G++.

Généralement, g++ et gcc correspondent aux mêmes programmes.

2.1. Quelques mots sur la cross-compilation

On appelle un cross_compilateur, un programme s'exécutant sur un pc appelé « host », d'une architecture donnée, et générant du code machine destiné à s'éxécuter sur un pc appelé « target » (ou cible en français).

Par exemple, il est possible de générer, à l'aide d'un cross_compilateur, du code pour une machine ayant un processeur à base de powerpc, depuis une machine de type x86.

La création d'un cross_compilateur est souvent longue et fastidieuse. Il faut en effet :

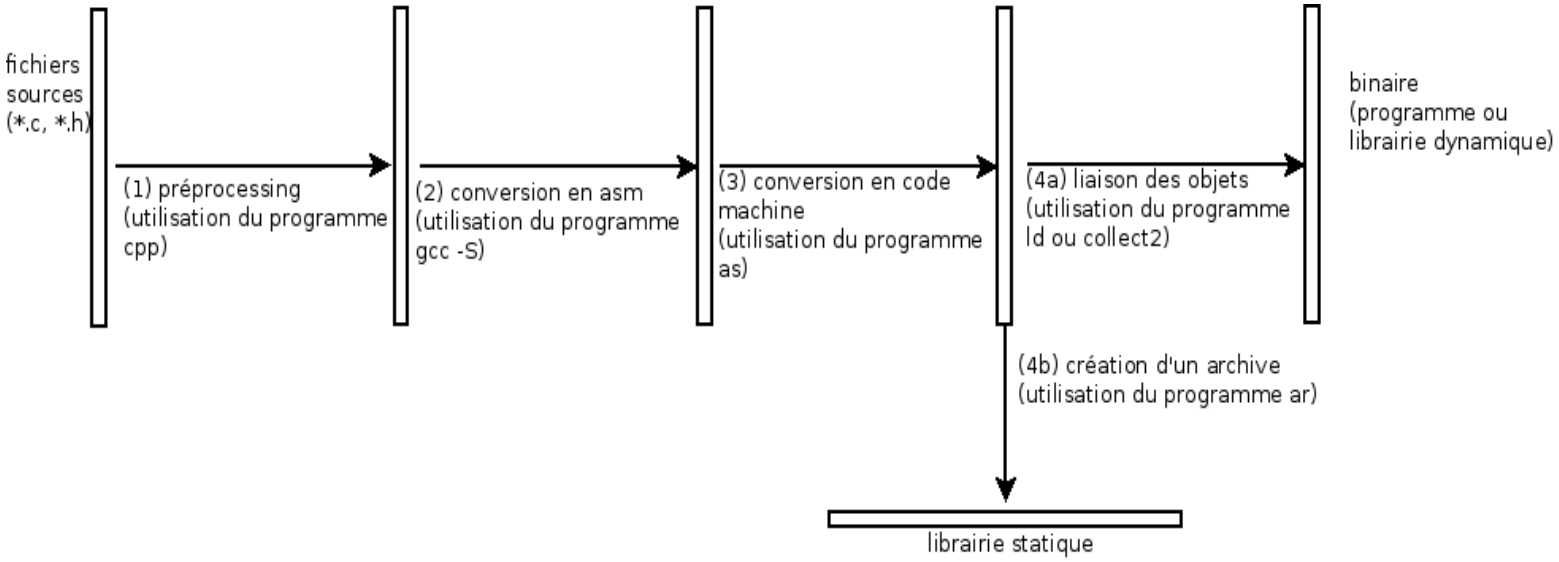
- choisir le type de libc utilisée (glibc, uclibc, eglibc)
- choisir la version de gcc
- choisir la version des fichiers d'entêtes du noyau (pour la compilation de la partie sysdep de la libc)
- choisir la version des binutils
- ...
- et compiler tout ça !

Il existe pas mal d'outils permettant de créer un cross_compilateur de manière assez simple. Les plus connus sont cités ci-dessous, et feront l'objet d'un autre écrit :

- buildroot (uclibc)
- crosstool-ng (glibc, eglibc, uclibc)
- crosstool

Cette capacité à utiliser GCC en tant que cross_compilateur est un des intérêts majeurs de cet outil.

2.2.Schéma (simplifié) des étapes de compilation



2.2.1Étape de préprocessing

L'étape de préprocessing fait intervenir le programme CPP (C-PreProcessor). Ce programme a en charge la transformation du code avant la compilation.

Ce programme s'appelle ainsi car son principal (mais néanmoins importantissime) travail consiste à traiter toutes les directives de compilation telles que :

- #include
- #define
- #if, #ifdef ...

Concrétement, il va transformer le code en le nettoyant de ses directives de type macro.

Il est possible d'arrêter GCC après cette étape, en utilisant l'option -E.

```
gcc -E source.c
```

Cela revient au même que d'utiliser directement la commande cpp

```
cpp source.c
```

Le programme CPP a une dernière utilité. Il permet de récupérer les macros prédéfinies à la création de votre compilateur. Pour cela, il suffit d'utiliser la commande

cpp -dM /dev/null

Voici un extrait de la sortie sur un linux standard 32 bits, architecture x86.

```
fred@kdiv ~/ >cpp -dM /dev/null
#define __DBL_MIN_EXP__ (-1021)
#define __pentiumpro__ 1
#define __UINT_LEAST16_MAX__ 65535
#define __FLT_MIN__ 1.17549435082228750797e-38F
#define __UINT_LEAST8_TYPE__ unsigned char
#define __INTMAX_C(c) c ## LL
#define __CHAR_BIT__ 8
#define __UINT8_MAX__ 255
#define __WINT_MAX__ 4294967295U
#define __SIZE_MAX__ 4294967295U
#define __WCHAR_MAX__ 2147483647L
#define __GCC_HAVE_SYNC_COMPARE_AND_SWAP_1 1
#define __GCC_HAVE_SYNC_COMPARE_AND_SWAP_2 1
#define __GCC_HAVE_SYNC_COMPARE_AND_SWAP_4 1
#define __DBL_DENORM_MIN__ ((double)4.94065645841246544177e-324L)
#define __GCC_HAVE_SYNC_COMPARE_AND_SWAP_8 1
#define __FLT_EVAL_METHOD__ 2
#define __unix__ 1
...
```

Toutes ces macros sont très utilisés dans le code des bibliothèques C.

2.2.2Conversion en code assembleur

L'étape suivante consiste à transformer le code C pré-traité, en du code assembleur, spécifique à l'architecture cible.

GCC utilise le programme as pour réaliser cela. Il est possible, de la même façon, d'arrêter la compilation après la génération des fichiers assembleurs.

Ainsi, la commande

```
gcc -S test.c
```
générera un fichier test.s, correspondant à la transformation du fichier test.c, après le préprocessing, et après la transformation en fichier assembleur spécifique.

Voici un extrait du resultat, à savoir le fichier test.s

```
  .file   "test.c"
  .comm   toto,4,4
  .section    .rodata
  .LC0:
      .string "hello"
  .LC1:
      .string "%p\n"
```

```
      .text
      .type   f1, @function
f1:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $24, %esp
    movl    $.LC0, (%esp)
    call    puts
    movl    4(%ebp), %eax
    movl    %eax, toto
    movl    toto, %edx
    movl    $.LC1, %eax
    movl    %edx, 4(%esp)
    movl    %eax, (%esp)
    call    printf
    leave
    ret
    .size   f1, .-f1
    .section    .rodata
.LC2:
    .string "unable to open procfile"
    .align 4
.LC3: apitre 2
```

Il n'est pas nécessaire de comprendre tout ce qui est écrit, néanmoins, c'est à cette étape que le découpage en section commence. Une section peut être vue comme un sous-ensemble d'un fichier objet. Il existe plusieurs sections prédéfinies dans un fichier objet :

- section .rodata contenant les chaines de caractères constantes
- section .data contenant les données globales initialisées
- section .bss contenant les données globales non-initialisées (valant la valeur 0 par défaut)
- section .text contenant le code

Beaucoup d'autres existent (.plt, .got, .ref.plt, .init, .initdata) et seront étudiés dans le chapître sur le format ELF.

Pour en revenir à l'extrait de code ci-dessus, les instructions « .section .rodata » et « .text », introduisent respectivement la section rodata(contenant la chaîne « hello » et « unable to open procfile ») et text (contenant le code assembleur correspondant à la fonction C f1).

2.2.3Conversion en code machine

L'étape suivante est la conversion de ces fichiers en code machine, d'extension .o.

Ces fichiers, appelées fichiers objets, correspondent à la transformation du code assembleur en code machine, spécifique à l'architecture cible.

Ces fichiers sont qualifiés de « relocalisables ». Cela signifie que toutes les sections (voir ci-dessus) d'un fichier objet, contiennent des données référencées de manière relative.

Pour arrêter la compilation à cette étape, la ligne de commande suivante suffit :

```
gcc -c fic.c
```
Cela générera par défaut un fichier fic.o.

2.2.4Création d'une librairie statique

Après l'étape précédente, il est possible de générer un fichier de type librairie statique. Nous verrons un peu plus tard ce qu'est une librairie statique, et comment l'utiliser. Pour le moment, il est possible de comprendre ce chapître en considérant une librairie comme un fichier contenant un ensemble de fonctions. La plus connu est évident la libc, qui est utilisé automatiquement par tous vos programmes C (sauf option de compilation particulière).

Pour créer une librairie statique, il suffit de créer une archive contenant tous les fichiers d'extensions « .o », correspondant au code objet des fonctions.

```
ar cr libname.a fic1.o fic2.o fic3.o
```

Une librairie statique n'est donc qu'une archive, contenant du code relocalisable.

Il est possible de lister l'ensemble des membres d'une archive. L'exemple le plus intéressant demeure sur la version statique de la libc.

```
fred@kdiv ~ >ar t /usr/lib/libc.a |grep printf
vfprintf.o
vprintf.o
printf_fp.o
reg-printf.o
printf-prs.o
printf_fphex.o
printf_size.o
fprintf.o
printf.o
snprintf.o
sprintf.o
asprintf.o
```

2.2.5Création d'un binaire(ou d'une librairie dynamique)

Cette étape fait intervenir le programme LD (GNU Linker). L'objectif de cette étape qui est de loin la plus complexe du processus de compilation, va être de lier les différents fichiers objets pour générer soit une librairie dynamique, soit un programme. Le mécanisme de liaison se fait donc en plusieurs étapes :

- concaténation des différentes section des différents objets
- référencement des symboles undéfinis, correspondant à des symboles définies dans d'auter librairies.

- création d'une tableau de liaison, appelée .plt (procedure linkage table), indiquant la manière dont devront être résolus les symboles référencés, à l'exécution

Création d'une librairie dynamique

La création d'un librairie dynamique se fait généralement en deux étapes distinctes :

- une étape où les fichiers objets sont créés
- une étape où ils sont assemblés pour former une librairie dynamique

La première étape a déjà été vue précédemment. Elle correspond à la ligne de commande suivante :

```
gcc -c fic.c
```

Néanmoins, cette commande va devoir être légèrement modifiée pour permettre de passer à l'étape suivante. Il faut en effet utiliser l'option -fpic. Cette option va demander au programme cc de générer un code de type PIC (position-independant-code).

En d'autres termes, le code généré avec cette option va être plus volumineux que précédemment. Il est possible de comparer le code généré pour une fonction, compilée avec fpic et sans fpic.

```
gcc -c fic.c -o fic_without_fpic.o
gcc -fpic -c fic.c -o fic_with_fpic.o
```

L'utilisation du programme de désassemblage objdump, permet de comparer les deux codes générés :

```
objdump -D fic_without_fpic.o > desassemble_1.txt
objdump -D fic_with_fpic.o > desassemble_2.txt
vimdiff desassemble_1.txt desassemble_2.txt
```

Bref, ... Ce qu'il faut pour l'instant retenir, c'est la nécessité d'utiliser l'option -fpic. Cela permettra de générer du code dont le positionnement en mémoire est indépendant, ce qui est important dans le cas de librairies dynamiques, puisqu'elles sont chargées (à des adresses aléatoires) en mémoire, à l'exécution du programme.

La deuxième étape va être la formation de la librairie dynamique. Cela se fait à l'aide de la commande suivante :

```
gcc fic.o -o libname.so -shared
```

Cette ligne de commande demande à gcc de former une librairie à partir du fichier objet PIC fic.o. Il est possible de former une librairie avec plusieurs fichiers objets.

```
gcc *.o -o libname.so -shared
```

Il existe un dernier point à souligner : lorsque vous compilez une librairie, vous n'aurez pas d'erreurs pour des symboles undéfinis. Supposons par exemple que vous ayez créer une librairie, qui sera appelée dans notre exemple libsimple.so.

Dans cette librairie, vous avez du code qui utilise des fonctions qui ne sont pas définies dans la libc. Par exemple, vous utilisez la fonction « dlopen » qui est définit dans la librairie /usr/lib/libdl.so.

Clairement, lorsque vous allez créer votre librairie dynamique, vous n'aurez pas d'erreur de « linkage » à la compilation. Ce symbole sera néanmoins considéré comme indéfini. Il est possible d'utiliser le programme « nm » pour analyser les symboles d'un binaire (librairie dynamique ou programme).

Par contre, un programme utilisant la librairie libsimple.so sera forcé d'indiquer au compilateur qu'il a besoin de la librairie libdl.so pour que le programme LD puisse résoudre les références. En d'autres termes, vous pouvez avoir des symboles indéfinis dans une librairie dynamique, nous ne pourrez pas pour un programme.

Exemple :

Soit le fichier sym_resolv.c. Ce dernier va servir pour créer une librairie libsym_resolv.so. Il utilise des fonctions non-définies dans la libc, tel que dladdr, qui appartient à la librairie libdl.

Il est possible de compiler cette librairie comme suit :

```
cc -fpic *.c -o libsym_resolver.so -shared   -Wextra -Wall -W
```

Nous constatons que nous ne précisons par que la librairie utilise la librairie libdl. Le symbole dladdr va donc rester undéfini.

```
fred@kdiv ~ >nm  -A -S -l libsym_resolver.so  |grep dladdr
libsym_resolver.so:         U dladdr
```

La ligne ci-dessous signifie que le symbole dladdr est indéfini (U) dans la librairie libsym_resolver.so

Il est possible d'utiliser le programme readelf pour obtenir plus d'informations :

```
readelf -a libsym_resolver.so |less
Relocation section '.rel.plt' at offset 0x56c contains 19 entries:
…
00002444  00000a07 R_386_JUMP_SLOT   00000000   dladdr
…
Symbol table '.dynsym' contains 30 entries:
     9: 00000000     0 FUNC    GLOBAL DEFAULT  UND close@GLIBC_2.0 (2)
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND dladdr
```

Les extraits précédent montre que le symbole dladdr est de type inconnu, et qu'il est dans la section dynsym, contenant les symboles dynamiques.

Si nous souhaitons que le symbole soit référencé dans la librairie elle-même, il est possible d'utiliser la compilation suivante :

```
gcc -fpic *.c -o libsym_resolver.so -shared   -ldl -Wextra -Wall -W
```

L'unique différence est que nous informons le compilateur d'utiliser la librairie /usr/lib/libdl.so (grâce à l'option -ldl).

Si nous utilisons les commandes précédents pour étudier les symboles, nous obtenons les résultats suivants :

```
fred@kdiv ~>nm -A -S -l libsym_resolver.so |grep dladdr
libsym_resolver.so:         U dladdr@@GLIBC_2.0

fred@kdiv ~>readelf -a libsym_resolver.so |less
Relocation section '.rel.plt' at offset 0x598 contains 19 entries:
0000248c  00001207 R_386_JUMP_SLOT   00000000   dladdr
...
Symbol table '.dynsym' contains 30 entries:
18: 00000000     0 FUNC    GLOBAL DEFAULT  UND dladdr@GLIBC_2.0 (4)
```

Ce coup-ci, à la différence de toute à l'heure, le symbole est reconnu en tant que fonction. De plus, des informations supplémentaires permettront à l'exécution de la librairie, de résoudre les symboles dynamiques (voir cours sur ELF, section : résolution de symboles dynamiques).

Au final, la différence entre ces deux méthodes est que, dans le premier cas, un programme utilisant la librairie libsimple.so devra indiquer sur la ligne de compilation qu'il utilise la libdl (car contenant des symboles de types indéfinis), alors que ce ne sera pas nécessaire dans le deuxième cas (car contenant des symboles externes, avec suffisamment d'informations pour que la librairie puisse les résoudre elle-même).

Création d'un binaire

La création d'un programme se fait de manière très simple, avec la commande suivante :

```
gcc *.c -o le_nom_du_programme
```

Votre programme peut utiliser des librairies externes, différentes de la libc, auquel cas il faudra ajouter des options de compilation. Cela sera étudié plus bas.

De plus, lorsque votre programme est compilé, il ne peut y avoir, au contraire d'une librairie dynamique, de symboles undéfinies (il peut y'avoir des symboles externes cependant).

3.Options de GCC

GCC comporte un grand nombre d'options de compilations : ces options vont servir aussi bien à ajouter des répertoires de recherche de fichiers d'entêtes .h, de librairies, où de la norme de langage C à utiliser. Les chapîtes ci-dessous présentent les plus utilisées.

3.1.Option de recherche de fichier d'entête

Par défaut, les fichiers d'entêtes sont recherchés dans le répertoire /usr/include. Il arrive néanmoins que l'on souhaite utiliser des fichiers d'entêtes situés dans un autre répertoire. Si vous utilisez par exemple l'instruction suivante :

#include <lmd.h>

le préprocesseur cpp, appelé lors de la première étape, va chercher dans le répertoire /usr/include ce fichier, afin de remplacer l'instruction par le contenu. S'il ne le trouve pas, la compilation échouera dés la première étape. Il est donc important de préciser des répertoires de recherches supplémentaires au compilateur GCC. Cela se fait se fait avec l'option -I.

Supposons que le fichier lmd.h se situe dans le répertoire /usr/local/perso/include.

Pour que cpp puisse trouver ce fichier, il faut donc utiliser la commande suivante :

```
gcc -I/usr/local/perso/include -E fic.c
```

Evidemment, si vous compilez directement le programme sans passer par toutes les sous-étapes, l'option -I est également obligatoire.

Il est également possible d'utiliser une variable d'environnement, dans laquelle seront indiqués des répertoires supplémentaires pour rechercher des fichiers d'entêtes.

```
export C_INCLUDE_PATH=/usr/local/perso/include:/usr/local/perso2/include
```

L'ordre de recherche d'un fichier d'entête est alors le suivant :

- recherche dans les répertoires renseignés dans C_INCLUDE_PATH
- recherche dans les répertoires renseignés via -I
- recherche dans le répertoire /usr/include

3.2.Option de recherche de librairie/Utilisation de librairie

Par défaut, le compilateur va chercher les librairies utilisées par le programme dans les répertoires suivants :

- /lib
- /usr/lib
- /usr/local/lib

Pratiquement dans tous les cas de figure, vous utilisez sans le savoir la librairie libc, située dans le répertoire /lib/libc.so. Cette librairie contient l'ensemble standard du langage C.

Il peut arriver cependant qu'on souhaite utiliser une librairie qui ne se situe pas dans un de ces répertoires. Par exemple, nous voudrions utiliser la librairie libsimple.so, située dans le répertoire /usr/local/perso/lib.

Pour cela, deux options supplémentaires sont nécessaires :

- une pour indiquer que le programme doit être compilé avec la librairie libsimple.so
- une autre pour indiquer que le compilateur doit rechercher les librairies dans les répertoires standard, mais également dans /usr/local/perso/lib

```
gcc fic.c -L/usr/local/perso/lib -lsimple -o prog
```

L'option -L permet d'indiquer des répertoires de recherche supplémentaire.

L'option -lname permet d'indiquer quelles librairies doivent être utilisées pour la compilation.

De la même manière, il existe une variable d'environnement, LIBRARY_PATH, permettant d'ajouter des répertoires de recherche supplémentaire.

La recherche de librairie se fait dans l'ordre suivant:

- recherche dans les répertoires renseignés dans LIBRARY_PATH
- recherche dans les répertoires renseignés dans -L
- recherche dans /lib, /usr/lib, /usr/local/lib

Le point à ne pas oublier est qu'un programme, sauf option particulère, utilise /lib/libc.so sans qu'il ne soit nécessaire d'indiquer d'option supplémentaire.

3.3.Option de compilation pour debug

Il arrive fréquemment qu'on souhaite débugguer un programme. Pour faciliter cela, il est possible d'utiliser des options de compilations supplémentaires ajoutant des sections de debug, utilisée par le debugguer GDB.

gcc prog.c -o prog -ggdb

A noter qu'il est intéressant, dans le cas d'un programme qui plante, de lui permettre de générer un fichier core, contenant une image mémoire au moment du plantage. Pour cela, il est nécessaire d'utiliser dans le shell la commande suivante :

ulimit -c 8190

Cela autorisera au système d'exploitation de générer un fichier core de taille maximum 8190, lorsqu'un programme, lancé depuis votre shell, a planté.

3.4.Option de compilation pour MUDFLAP

Mudflap est une option de compilation permettant de détecter les buffer overflow, et plus généralement les manipulations incorrectes de pointeurs.

MUDFLAP est une extension de GCC agissant à deux niveaux :

- Elle intervient au moment de la compilation en transformant le code généré.
- Elle effectue des tests sur l'utilisation des pointeurs pendant l'exécution.

Le premier niveau est réalisé de la manière suivante :

L'utilisation de MUDFLAP conduit à l'ajout d'une étape à la compilation. Cette dernière intervient juste avant la phase d'optimisation effectuée par GCC. Pour cela, un arbre de syntaxe est fournit par GCC à MUDFLAP. L'analyse de cet arbre a pour objectif de trouver un ensemble de modèles correspondant à des accès mémoires dangereux. Ces modèles sont alors surcouchés par des expressions évaluant la même valeur, mais ajoutant un ensemble de test fournit par la bibliothèque MUDFLAP. Ces tests surviendront à l'exécution du programme.

Le second niveau n'est que le résultat d'utilisation de fonctions de la bibliothèque MUDFLAP, lesquelles ont été ajoutées à l'étape précédente.

MUDFLAP conduit donc à une modification de l'ABI et du code généré.

La mise en oeuvre de MUDFLAP se fait via l'utilisation d'option de compilation et de linkage.

Deux cas de figures seront distingués :

- compilation d'un programme sans thread
- compilation d'un programme avec thread

Dans le premier cas de figure, il suffit de compiler le programme avec les options -fmudflap -lmudflap. L'option -lmudlfap spécifie que le programme doit être lié à la librairie MUDFLAP.

L'option -fmudflap spécifie à GCC qu'une passe doit être ajoutée à la compilation.

Le deuxième cas de figure est similaire, excepté que l'option -fmudflapth doit remplacer -fmudflap.

Un exemple de compilation serait donc :

```
gcc -fmudflap fic.c -lmudflap -o fic
```

Il n'est pas nécessaire d'utiliser -L, car la libmudflap se situe dans un répertoire de base (/usr/lib)

Le comportement de MUDFLAP, à l'exécution, peut également être paramétré au moyen de la variable d'environnement MUDFLAP_OPTIONS.

Notamment, il est possible de spécifier le comportement de l'application lors d'une violation d'accés mémoire :

- -viol-nop : ne rien faire, si ce n'est afficher une trace
- -viol-abort : appeler la fonction abort
- -viol-sigsegv : générer un signal SIGSEGV au programme
- -viol-gdb : créer une session de débuggage avec ce programme

Il est également possible de spécifier le mode de fonctionnement de mudflap, au niveau de son analyse :

- -mode-nop : désactiver mudflap
- -mode-check : comportement par défaut

Diverses options permettent de modifier la manière dont les traces s'affichent :

- -backtrace= X : afficher une pile d'appel de profondeur X
- -abbreviate : afficher des rapports plus concis

En tout état de cause, la page du projet [http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging] présente toutes ces options.

3.5.Option de compilation pour vérification d'erreur

Dés la phase de génération de fichier objet, il est possible de demander à GCC de faire des vérifications supplémentaires dans le code. Généralement, sur une architecture standard, les options -Wall et -W suffisent. Il est possible d'ajouter également -Wextra.

```
gcc -W -Wall -Wextra fic.c -o prog
```

3.6.Option de compilation pour génération de couverture

Très souvent, sans les entreprises développant des algorithmes complexes, il est important de s'assurer que tout le code d'une librairie ou d'un programme à été tester. Il n'est pas rare en effet que l'exécution d'une branche conditionnelle (via un if-else), ne soit presque jamais effectuée tant les conditions y menant sont rarement satisfaites.

Pour analyser le taux de couverture d'un programme, il suffit de le compiler avec les options suivantes :

```
gcc -fprofile-arcs -ftest-coverage fic.c -o prog
```
La deuxième étape est d'exécuter le programme

```
./prog
```
Son exécution conduit à la création de fichier (*gcno et *gcda)

Ensuite, ces fichiers peuvent être analysés par le programme gcov.

```
gcov fic.c
```

Cette commande affichera la fréquence avec laquelle chaque ligne a été appelée, le code mort, etc etc.

3.7.Option de profiling

Pour des soucis de performance, il est fréquent de faire une analyse métrique de l'exécution de code. Cela se fait généralement dans le but de trouver les points du programme dans lesquels l'application passe le plus de temps.

Pour cela, il est nécessaire de compiler un programme avec les options de profiling :

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

Une fois ceci fait, il ne reste plus qu'à exécuter le programme.

```
./prog
```

Cette exécution conduit à la création d'un fichier gmon.out. Ce fichier peut être utilisé par le programme gprof pour générer un rapport de métrique.

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

4.Librairies statiques et dynamiques

4.1.Différences

Une librairie statique est une archive de type ar, contenant plusieurs fichiers d'extension .o.

```
fred@kdiv >ar t /usr/lib/libc.a |head
init-first.o
libc-start.o
sysdep.o
version.o
check_fds.o
libc-tls.o
elf-init.o
dso_handle.o
errno.o
errno-loc.o
```

Une librairie dynamique est un fichier d'extension .so, générée à partir de plusieurs fichiers d'extension .o de type PIC.

```
fred@kdiv ~>file /lib/libc-2.12.so
/lib/libc-2.12.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.18, stripped
```

Une librairie statique voit ses fichier d'extensions .o directement incorporé dans le binaire l'utilisant.

```
fred@kdiv ~ >nm --print-armap /usr/lib/libc.a |tail -n 20
dl-tlsdesc.o:
00000020 T _dl_tlsdesc_resolve_abs_plus_addend
         U _dl_tlsdesc_resolve_abs_plus_addend_fixup
00000080 T _dl_tlsdesc_resolve_hold
         U _dl_tlsdesc_resolve_hold_fixup
00000040 T _dl_tlsdesc_resolve_rel
         U _dl_tlsdesc_resolve_rel_fixup
00000060 T _dl_tlsdesc_resolve_rela
         U _dl_tlsdesc_resolve_rela_fixup
00000000 T _dl_tlsdesc_return
00000010 T _dl_tlsdesc_undefweak
```

Une librairie dynamique n'est jamais incorporée. A la place, le binaire contient des références vers la librairie et les symboles qu'elle définit.

4.2.Utilisation de librairie statique

Par défaut, un programme n'utilisera jamais la version statique d'une librairie, sauf dans deux cas de figure :

- s'il n'a pas trouvé l'équivalent en librairie dynamique
- s'il lui a été explicitement ordonné d'utiliser la version statique

Pour le deuxième cas de figure, il faut utiliser une option de compilation particulière pour le programme:

```
gcc prog.c -o prog -static
```

L'option -static permet d'utiliser la libc (ou tout autre librairie qui pourrait être utilisée par le programme) de manière statique.

Cela signifie concrétement que, si le fichier prog.c utilise uniquement les fonctions printf et scanf de la librairie C, les fichiers printf.o et scanf.o seront extraits de l'archive libc.a et intégré directement dans le code généré, sans références.

Pour savoir à quel fichier .o correspond une fonction, on peut utiliser la méthode suivante :

```
fred@kdiv ~>nm --print-armap /usr/lib/libc.a 2>/dev/null |grep "^printf "
printf in printf.o
```

4.3.Utilisation de librairie dynamique

C'est le comportement par défaut. Dans ce cas, le fichier libc.so (ou tout autre librairie utilisée) est utilisé.

Il n'ya pas d'option particulière à utiliser.

4.4.Avantages/inconvénients

statique	dynamique
• Une compilation statique permet d'obtenir un programme autonome, ne nécessitant pas l'installation de librairie supplémentaire.	• Une compilation dynamique permet d'obtenir un programme plus petit, car il ne contient que des références • Le programme n'a pas besoin d'être recompilé lorsque la librairie change de version (sauf s'il ya changement d'interface) ou lorsqu'elle subit un correctif

• Lors d'un correctif de la librairie, il faut recompiler tous les programmes qui l'utilisent, afin que ces derniers profitent des corrections, car le code est directement intégré • Le programme généré est plus gros	• Auparavant, l'exécution pouvait être plus lente du fait de la résolution des références à la compilation (via le programme ld-linux.so), mais aujourd'hui, les symboles peuvent être résolues dés l'initialisation du programme

5.Exécution du programme

Pour exécuter un programme, il suffit d'indiquer dans le shell son chemin d'accés. Par exemple,

```
./monprog param1 param2
```

Il arrive néanmoins qu'à l'exécution, un programme s'arrête immédiatement avec l'erreur suivante :

```
error while loading shared libraries: libxxxx.so: cannot open shared object
file: No such file or directory
```

L'erreur est facilement explicable et corrigeable. Ce que ce message d'erreur signifie, c'est que le programme, à son exécution, va résoudre les symboles externes qu'il référence. Ces références ont pu être créé grâce aux diverses options -L et -l qui ont été utilisées.

Pour cela, dés le début de l'exécution de votre programme, un sous-programme va être (systématiquement) appelé par le système d'exploitation : ld-linux.so. Bien que son extension soit .so, il s'agit bien d'un programme. Son travail ressemble énormément au programme ld. Son objectif va être d'analyser les librairies dynamiques que votre programme utilise, et de les projeter en mémoire. Une fois qu'elles sont en mémoires, il va également résoudre les références de symboles externes que votre programme utilise en les remplaçant par l'adresse en mémoire du symbole.

```
fred ~ $ readelf -a test |grep ld-linux
      [Requesting program interpreter: /lib/ld-linux.so.2]
```

Le programme ld-linux.so va rechercher par défaut les librairies à projeter en mémoire, aux mêmes endroits que le linker ld à la compilation, à savoir :

- dans /lib, /usr/lib, /usr/local/lib

Il est également possible d'ajouter des répertoires de recherches supplémentaires en utilisant le fichier de configuration /etc/ld.so.conf.

Revenons-en à notre erreur. Globalement, l'erreur signifie que le programme ld-linux.so a détecté que le programme utilisait la libxxx.so, mais ne la pas trouvé dans les répertoires de recherches. Cela ressemble grandement au problème de compilation rencontré précédemment, sauf que dans le cas présent, nous n'avons pas une erreur de compilation mais d'exécution, et que le programme est ld-linux.so et non ld.

Pour résoudre cela, il existe trois manières différents :

- La première méthode consiste à copier la libxxx.so dans un des répertoires de recherches par défaut, ou au moins d'y faire un lien symbolique
- la deuxième méthode consiste à modifier le fichier de configuration /etc/ld.so.conf et d'utiliser le programme ldconfig pour regénérer la configuration
- la troisième méthode consiste à utiliser la variable d'environnement LD_LIBRARY_PATH (à ne pas confondre avec LIBRARY_PATH utilisée par ld)

par exemple, si on suppose que la librairie libxxx.so se situe dans /var/lib :

LD_LIBRARY_PATH=/var/lib ./programme

6.Exercices proposés

1. Programmer une librairie permettant de faire une gestion de pile (Last In First Out).
    1. Compiler-la en statique avec les options de vérification
    2. Compiler-la en dynamique avec les options de vérification
2. Programmer un applicatif testant votre librairie dynamique
    1. compiler le programme avec le support de mudflap
    2. compiler le programme avec le support de la mesure métrique
        1. faites une analyse
    3. compiler le programme avec le support de couverture
        1. faites une analyse
3. Entrainez-vous sur chaque fichier .o, .a, .so, à utiliser les programmes permettant :
    1. de lire le fichier binaire (readelf)
    2. de desassembler un programme (objdump)
    3. d'afficher les symboles (nm)
    4. de lister le contenu d'une archive (ar)