Annexe
Petit Traité Pratique sur l'utilisation de Gcc sous Linux

par B.Albar

3 juin 2008

Résumé : Démystification de gcc sous linux. Le code source et l'ensemble du texte présenté dans ce document a été entièrement conçu par leur auteur, et sont libre de droits. Vous pouvez les utiliser, les modifier et les redistribuer à votre convenance.

Première partie
Introduction

GCC (GNU Compiler Collection) est un ensemble de compilateurs libre supportant de nombreux languages tel que le C, C++, ADA, Objective-C, D, Java, langages assembleur et d'autres. Nous nous intéresserons plus particulièrement au compilateur C et assembleur (Gas). On supposera comme préalable que le lecteur connaît correctement le language d'assemblage x86.

Deuxième partie
Compilation en C

1 Syntaxe

La commande la plus simple pour compiler un programme est la suivante :

```
1 gcc nomdufichiersource.c -o
    nomdufichierbinaire
```

De même pour un programme de deux fichiers, le plus simple est :

```
1 gcc nomsrc1.c nomsrc2.c -o nombinaire
```

Si le programme est composé d'un nombre important de fichiers, il est nécessaire pour des questions de commodités d'utilisé un `makefile`.

2 Rappel des principales options de compilation

A ces commandes ci-dessus, nous pouvons y ajouter plusieurs options dont voici les plus importantes :

-s : génère un fichier assembleur au lieu et place d'un fichier binaire

-o : génère un fichier objet (non lié)

-Wall : active tous les avertissements (warnings) au moment de la compilation

-g : génère les informations (essentielles) au bon débugage du programme

-O3 : active l'ensemble des optimisations (rapidité) (-O2 et -O1 pour les niveaux d'optimisation inférieur)

-Os : optimisation en taille

Troisième partie
Assembleur

Beaucoup de personnes sous Linux utilisent `nasm` ou d'autres assembleurs. Poutant GCC inclut son propre compilateur assembleur, j'ai nommé `gas` (GNU Assembler). Par défault, la syntaxe utilisé par `gas` (AT&T) est différente de celle traditionellement utilisé par d'autre compilateur asm comme `nasm` (syntaxe Intel) . Néammoins sur les versions récentes de `gas`, une directive de compilation permet de passer en syntaxe Intel.

1 Ecrire en assembleur en AT&T avec gas

Nous étudierons dans cette partie les différences entre les syntaxes AT&T et Intel, les directives de compilations propres à `gas` à partir du programme suivant :

```
1  .section .data
2  chaine:
3  .ascii "Les données sont %d et %d\n\0"
4  dat:
5  .long 5,4 # tableau de 2 long initialisé à 5 et 4
6  variable:
7  .lcomm var, 8   # 8 octets réservé sur le bss
      initialisé à 0
8
9  ############ Constantes ###########
```

```
10 .equ LINUX_SYSCALL, 0x80      # interruption
       système
11                               # pour quitter le
                                     programme
12 ############ Fin Constantes  ######
13
14 # Et maintenant le code :
15 # le but du jeu étant de déplacer les données
       dans le
16 # tableau var et de les afficher par un appel à
       printf
17
18 .section .text
19 .globl _start
20 _start:
21 call exemple
22 call affichage
23 call quiter
24
25 # Fonctions ####################
26 .globl exemple
27 .type exemple,@function
28 exemple:
29 pushl %ebp
30 movl %esp, %ebp
31 xorl %edx, %edx
32 movl $dat, %esi
33 movl $var, %edi
34 movl (%esi), %eax
35 movl %eax, (%edi, %edx, 4)
36 incl %edx
37 movl 4(%esi), %eax
38 movl %eax, (%edi, %edx, 4)
39 movl %ebp, %esp
40 popl %ebp
41 ret
42
43
44 .globl affichage
45 .type affichage,@function
46 affichage:
47 pushl %ebp
48 movl %esp, %ebp
49 pushl var+4
50 pushl var
51 pushl $chaine
```

```
52  call printf
53  movl %ebp, %esp
54  popl %ebp
55  ret
56
57
58  .globl quitter
59  .type quitter,@function
60  quitter:
61  movl $1, %eax # on place le numéro de l'
        interruption dans eax
62  movl $0, %ebx # code de retour pour l'
        interruption, ici 0
63  int $0x80      # appel système
64  ret
```

1.1 Compilation

La compilation d'un fichier en assembleur nécessite deux étapes la création d'un fichier objet avec `as` puis la phase d'étidier de lien avec `ld` :

```
1  as -o exemple.o exemple.c
2  ld exemple.o -o exemple -lc -dynamic-
       linker /lib/ld-linux.so.2
3  ./exemple
```

Les options `-lc` et `-dynamic-linker` permettent de lier dynamiquement le programme à une librairie externe en l'occurence la glibc, celle-ci sera chargée en mémoire dans une zone spécifique au démarage du programme.

1.2 Différence entre syntaxe AT&T et syntaxe Intel

La principale différence entre les deux syntaxes qui nécessite une periode d'adaptation est l'ordre des argumemts, inversé en syntaxe AT&T. Par exemple :

$$\textit{mov eax, 5} \Longleftrightarrow \textit{movl \$5, \%eax}$$

D'autre part, l'ensemble des registre (`eax`, `ebx`, `ecx` , ...) sont précédés par le caractère % et les instructions sont suivit par la taille sur laquelle elles travaillent (l pour 32 bits, w pour 16b, b pour 8 bits) . On remarque aussi que l'adressage immédiat est précédé du signe \$. Voyons les autres modes d'adressage :

L'adressage indirect qui en syntaxe Intel s'écrit par exemple :

$$\textit{mov eax, [ebx]} \Longleftrightarrow \textit{movl (\%ebx), \%eax}$$

De même *mov eax, [ebx+4]* devient *movl 4(%ebx), %eax*. Plus compliqué :

$$\textit{mov [8 + \%ebx + 1 * edi], eax}$$

$\Longleftrightarrow$

movl %eax, 8(%ebx,%edi,4)

Regardons de plus près le contenu de la parenthèse de droite `8(%ebx, %edi,4)` : 8 représente le décalage par rapport à l'adresse de base stocké dans `ebx`. Le contenu de `edi` représente aussi un déplacement par rapport à l'adresse de base mais non plus statique, celui-ci peu varier au cours de l'éxécution. 4 représente la taille du déplacement (les valeurs acceptés sont 4 ,8 et 16 pour ce dernier paramètre).

Le mode d'addressage direct est quand à lui on ne peut plus simple puisqu'il n'est précédé d'aucun symbole particulier (*movl Addresse, %eax* par exemple)

1.3 Directives de compilation

1. La directive *.section* marque le début d'une section et est toujours suivi du type de section, Généralement les types les plus utilisés sont *.section .data* pour le segment de données et *.section .text* pour le segment de code.
2. La trop peu connue directive *.intel_syntax noprefix* qui permet de passer en syntaxe Intel ! (et sa réciproque *.att_syntax noprefix)*
3. Les étiquettes comme `chaine :` ou encore `_start` : sont traité comme un symbole représentant l'adresse de la donnée ou de l'instruction qui la suit immédiatement. En l'occurence, *chaine :* représente l'adresse de la chaine de caractère qui suit et *_start :* est un symbole particulier qui définit le point d'entrée du programme. Ces symboles sont utilisable dans le code : *movl chaine, %eax* est correct tout comme *jmp _start.*
4. Les directives définissant les points d'entrée des fonctions ou du programme : *.globl etiquette* (par exemple *.globl _start* pour le point d'entrée). Dans le cas d'une fonction, on l'indique par la directive suivante : *.globl etiquettefct* suivi de *.type etiquettefct, @function* puis plus tard dans le code un *etiquettefct :* définit la première instruction de la fonction.
5. La directive *.equ* pourrait presque être considérée comme une macro simplifiée en C. Elle définit un symbole représentant un nombre. Prenons l'exemple suivant : *tab : .long 5, 4,3,2,1 tab_fin :* qui représente un tableau de 5 caractère de type long, on pourrait définir la taille du tableau par *.equ tab_taille, (tab_fin - tab)*
6. Les directives de données représentent un type spécifique de donnée :
   - *.ascii* pour une chaîne de caractères (doit se terminer par `\0`)
   - *.long init* pour un entier sur 32b (modifiable) initialisé à la valeur *init* (equivalent d'un type `long` en C).
   - *.byte* pour un entier sur 8b (type `char` en C)
   - *.int* pour un entier sur 16b (type `short` en C)
   - *.lcomm symb, taille* définit un symbole représentant une adresse pointant vers une zone mémoire de taille donnée (par le 2ème paramètre) alloué à l'éxécution sur le segment `bss` et initialisé à 0. Pratique pour un buffer, un tableau de données, ou encore une variable globale !

1.4 Variables locales

Je consacre une petite partie aux variables locales en assembleur. C'est en fait très simple, on réserve un espace sur la pile de taille voulue par un `subl $4, %esp` par exemple pour un `long`. Lorsque la variable ne sert plus à rien, à la fin de la fonction, il ne faut pas oublier de la libérer par un `addl $4, %esp`. Et voilà !

Quatrième partie
Assembleur inline et outline

Dans cette section, nous allons traité de l'assembleur intégré dans un language de haut niveau (en l'occurence le C) ainsi que de la manière de méler fonctions asm et C.

1 Assembleur inline avec GCC

L'assembleur inline du compilateur C de GCC est peu connu et/ou utilisé, du fait d'une part d'un manque de documentation (au moins francophone) sur le sujet ainsi que d'une nette rébarbativité. Néanmoins, il offre des possibilités de contrôle sur le code (au niveau de l'utilisation des registres, etc.) bien plus grande que certains autre compilateur dont je ne citerai pas le nom !

1.1 Analyse de code

L'assembleur inline (c'est à dire directement intégré dans une fonction C) débute par la directive `asm(...)` ou `__asm__(...)` au choix. La syntaxe est la suivante :

```
1  asm ("instructions"
2       : opérandes de sorties            /*
           optionelles */
3       : opérandes d'entrées             /*
           optionelles */
4       : liste des registres modifiés    /*
           optionelles */ );
```

Voyons immédiatement un exemple en syntaxe Intel pour utiliser la belle directive vue précédemment :

```
1  int main(void)
2  {
3       int a=10;
4       asm(".intel_syntax noprefix\n\t"
```

```
5           "mov eax, 0x3\n\t"
6           ".att_syntax noprefix"
7            : "=r" (a)
8            :                  /* Pas de
                 registres d'entrée */
9            :"%eax");          /* Registres
                 modifiés : eax */
10      printf("a = %d", a);
11      return 0;
12  }
```

Ce programme modifie la varible a. Au niveau des registres de sortie "*=r*" représente une contrainte appliqué au compilateur, dans le cas de r, il indique au compilateur de passer l'un des registres laissé à sa convenance (en l'occurence de charger a = 10 dans un registre reg, de modifier le registre reg = 0x3, et pour finir de réecrire le registre dans a = reg), le "=" quant à lui indique une écriture. Nous verrons de plus près les contraintes dans la partie suivante. Lorsque vous voyez une erreur du type *"error : impossible constraint in 'asm'..."*, cherchez de ce côté là! Voyons maintenant un second programme :

```
1   int main(void)
2   {
3         int a=10, b=5;
4         asm("movl %0, %%eax\n\t"
5             "addl %1, %%eax"
6             : "=r" (a)
7             : "0" (a), "g" (b)    /* Pas de
                  registres d'entrée */
8             :"%eax");             /*
                  Registres modifiés : eax  */
9       printf("a += b :  %d", a);
10      return 0;
11  }
```

Cet exemple permet de voir plus précisément les registres d'entrée et de sortie : on a le même registre de sortie que précédemment avec la même contrainte (passage par un registre). Au niveau des contraintes d'entrées, le "*0*" signifie que cette entrée à les même contraintes que la 0eme sortie en l'occurence un registre("*=r*"). La contrainte "*g*" de la deuxième entrée signifie que tout est laissé à la convenance du compilateur (g correspond à un registre quelconque un emplacement mémoire, ou même un entier en adressage immédiat si possible). On notera de légers changement au niveau de la notation des registres qui prennent *%%* pour les distinguer des opérandes d'entrés notés *%0* et *%1*. Enfin la partie des registres modifiés contient naturellement *"eax"*.

1.2 Contraintes en assembleur inline

Nous énumèrerons ici l'ensemble des principales contraintes possibles (nous renvoyons le lecteur à la documentation de GCC pour une liste exhaustive) :

- "*m*" pour une opérande stocké en mémoire
- "*r*" pour un passage par un registre quelconque (eax, ax, ebx, bx, ecx, edx, esi, edi)
- "*a*", "*b*", "*c*", "*d*" pour un passage par un registre spécifique (resp. eax, ebx, ecx ,edx)
- "*g*" le choix est laissé à la convenance du compilateur
- "*f*" un registre flottant
- "*A*" pour un registre sur 64b composé de deux registres sur 32b edx : :eax

Et nous notterons aussi le "=" qui correspond à une écriture et le "&" , moins employé, pour les opérandes modifiés avant la fin de la fonction (registre d'index par exemple) ! Et pour finir quelques z'oli exemples :

```
1 __asm__ volatile ("RDTSC" : "=A" (x));
```

L'instruction *rdtsc* (Read Time-Stamp Counter) est un compteur 64b intégré au processeur à partir des P6 (Pentium et compatibles) qui est incrémenté à chaque cycle d'horloge et qui fait un tour complet en 1000 ans. On le récupère en passant par 2 registres 32b, e, x représentant une variable mémoire sur 64b. La directive __volatile__ signifie au compilateur qu'il ne doit pas modifier la place de l'instruction dans le code à la compilation (pour faire une optimisation par exemple).

```
1 int main(void)
2 {
3         int a = 10, b = 15;
4         __asm__ __volatile__("addl  %%ebx
              ,%%eax" :"=a"(a): "a"(a), "b"(
              b));
5         printf("a+b=%d\n", a);
6         return 0;
7 }
```

Et en voilà nettement plus compliqué tiré directement des sources de la Glibc de linux : La célèbre fonction strcpy(char*, char*) !

```
1 satic inline char *strcpy(char *dest,
      const char *src)
2 {
3     int d0, d1, d2;
4     __asm__ __volatile__(  "1:\tlodsb\n\t"
5                            "stosb\n\t"
```

```
6                                   "testb %%al,%%
                                        al\n\t"
7                                   "jne 1b"
8                                   : "=&S" (d0), "
                                        =&D" (d1), "
                                        =&a" (d2)
9                                   : "0" (src),"1"
                                         (dest)
10                                  : "memory");
11      return dest;
12  }
```

On note la présence d'une étiquette en début de code qui servira pour un saut (*1 :*), l'utilisation de registres 16b (`ax`) et les instruction `lodsb` (load string byte) (= Charge dans AL l'octet adressé par DS :SI. Si DF = 0, alors SI est ensuite incrémenté, sinon il est décrémenté) et `stosb` (store string byte) (= Stocke le contenu de AL dans l'octet adressé par ES :DI. Si DF = 0, alors DI est ensuite incrémenté, sinon il est décrémenté.)

2 Assembleur outline avec gcc

Un autre moyen de mêler C et assembleur est d'écrire entièrement une fonction en assembleur et d'y faire appel depuis une fonction en C. Examinons immédiatement un exemple :

```
1  //* Partie C - Affiche le maximum de 2
       nombres *//
2  #include <stdio.h>
3
4  extern int maximum(int a, int b)
5
6  int main(void)
7  {
8       int a = 10, b = 15;
9       printf("Le␣maximum␣est␣:␣%d", maximum
            (a,b));
10      return 0;
11 }
```

```
1  # Partie asm : Renvoie le maximum de deux
       nombre #
2
3  .section .text
4  .globl maximum
5  .type maximum,@function
```

```
 6  maximum:
 7  pushl %ebp
 8  movl  %esp, %ebp
 9  movl  8(%ebp), %eax      # première valeur
       dans eax
10  movl  12(%ebp), %ebx     # seconde valeur
      dans ebx
11  cmpl  %ebx, %eax
12  jge   fin
13  movl  %ebx, %eax
14  fin:
15  popl  %ebp
16  ret
17
18  # Fin partie asm #
```

Les paramètres de fonctions sont par défaut empilés, des directives du compilateur C permettent de les faire passer par registre en déclarant la fonction maximum comme ça :

```
extern __attribute__((fastcall)) int maximum(int a,
                        int b)
```

Voici les 3 directives possibles, la seconde étant celle par défaut sur gcc

- `cldecl` : la procédure appelante supprimera les paramètres mis dans la pile (directive *__attribute__((cdecl))*)
- `stdcall` : la fonction appelée supprime les paramètres de la pile (directive *__attribute__((stdcall))*)
- `fastcall` : utilisation des registres EAX, EDX, ECX au lieu de passer les paramètres dans la pile, si plus de registres disponible poursuit sur la pile (directive *__attribute__((fastcall))*)

La compilation d'un programme de ce genre nécessite plusieurs étapes :

```
1  gcc -o fichierc.o fichierc.c
2  as -o fichierasm.o fichierasm.s
3  gcc prog -o fichierc.o fichierasm.o
4  ./prog
```

Cinquième partie

Makefile

Les commandes ci-dessus sont utiles uniquement lorsque le nombre de fichiers à compiler est faible. Pour des projets de plusieurs dizaines de fichiers, comme c'est le cas pour le notre, elles seraient trop lourdes à gérer : ici intervient le makefile pour automatiser la compilation et l'edition de liens :

```
1  CC = gcc
2  AS = as
3  DEBUG = no
4  SRCC = $(wildcard *.c)
5  SRCS = $(wildcard *.s)
6  OBJ = $(SRCC:.c=.o) $(SRCS:.s=.o)
7  EXE = exemple
8
9  ifeq ($(DEBUG),yes)
10     CFLAGS = -Wall -g
11 else
12     CFLAGS =  -Wall -O3 -ffast-math
13 endif
14
15 all: $(EXE) .PHONY: all
16
17 # Compilation d'objets %.o: %.c
18 $(CC) -c $^ -o $@ $(CFLAGS) %.o: %.s
19 $(AS) $^ -o $@
20
21 # Compilation de l'exécutable
22 $(EXE): $(OBJ)
23 $(CC) $^ -o $@ $(CFLAGS)
24
25 clean:
26     rm $(EXE) *.o
```

L'éxécution de la commande *./make* compile l'ensemble des fichiers .c et .s situé à la racine du dossier et génère le programme. Il ne reste plus qu'à l'éxécuter avec *./exemple*. Je ne détaillerai pas plus, car ceci sort du cadre de l'annexe.

Conclusion

Voilà nous avons fait le tour des capacités de Gcc ! J'espère que cela aura été utile. D'autre sujets n'ont pas été abordés ici comme l'utilisation du debugger `gdb` mais le lecteur trouvera toute la documentation nécessaire sur Internet !

Références

[1] JONATHAN BARTLETT, Programming from the Ground Up

[2] JEAN-MICHEL RICHER, http ://www.info.univ-angers.fr/~richer/index.php

[3] http ://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html