TP 1: Compilation et exécution pas à pas

Vous pouvez utiliser un des programmes faits en TD. On supposera ici que votre programme s'appelle *sum.s*.

•1.1. Compilation

Dans notre contexte, la compilation se décompose en 3 phases successives :

1. **Le traitement par le préprocesseur :** le fichier source est analysé par un programme appelé *préprocesseur* qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source, etc.) ;
2. **L'assemblage :** cette opération transforme le code assembleur en un fichier *binaire*, c'est à dire en instructions directement compréhensibles par le processeur. Le fichier produit par l'assemblage est appelé fichier *objet* ;
3. **L'édition de liens :** un programme est souvent séparé en plusieurs fichiers source (ceci permet d'utiliser des librairies de fonctions standard déjà écrites comme les fonctions d'affichage par exemple). Une fois le code source assemblé, il faut donc *lier* entre eux les différents fichiers objets. L'édition de liens produit alors un fichier *exécutable.*
4. Pour assembler un fichier source assembleur — disons `sum.s` — vous pouvez utiliser la commande suivante dans votre interpréteur de commande :

```
as -a --gstabs -o sum.o sum.s
```

6. Nous venons d'utiliser un certains nombre d'options

- `-o` indique que le nom du fichier objet — dans notre cas `sum.o` ;
- `-a` permet d'afficher le code source que nous venons d'assembler parallèlement aux codes assemblés et aux offsets correspondant ;
- `-gstabs` n'est utilisé que pour l'exécution pas à pas de notre exécutable.

•Pour obtenir un fichier exécutable `sum`, il nous faut mener à bien l'édition de liens en utilisant la commande :

- `ld -o sum sum.o`

- Voila, nous pouvons exécuter ce programme en invoquant le nom de l'exécutable dans votre interpréteur de commandes favori… si ce n'est que le résultat du programme (stocker une somme dans le segment de données d'un processus) ne nous est pas visible. Pour voir ce qui se passe, nous allons utiliser un outil d'exécution pas à pas.

•1.2 Exécution pas à pas dans l'environnement gnu debugger

- Pour ce faire nous allons utiliser le gnu debugger `gdb` dans un premier temps puis l'interface graphique `ddd`[1]. Un récapitulatif des principales commandes de gdb est disponible à la fin de ce TP.
- L'environnement `gdb` permet d'exécuter des programmes pas à pas et d'examiner la mémoire. Il dispose d'un guide utilisateur accessible par la commande info gdb. Pour utiliser gdb, il suffit de l'invoquer dans son interpréteur de commandes en lui indiquant le fichier à examiner :

•[soumam.lifl.fr-meftali-/home/.../Sources] gdb sum

```
GNU gdb 5.3-22mdk (Mandrake Linux)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are
welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "i586-mandrake-linux-gnu"...
(gdb)
```

- Ce programme propose une aide en ligne accessible par la commande help :
- `(gdb) help help`

```
Print list of commands.
(gdb) help quit
Exit gdb.
```

- **Exécution du programme**
- Le programme considéré peut être exécuté dans l'environnement `gdb` en utilisant la commande run.
- `(gdb) run`

```
Starting program: /home/.../Sources/sum
Program exited normally.
(gdb)
```

- Comme lors de l'exécution par le biais d'un interpréteur de commande, le fonctionnement de notre programme ne nous est pas apparent. Pour ce faire, nous allons forcer le programme à stopper et contrôler son exécution.
- ***Points d'arrêt***
- Lorsque le code source de l'exécutable est disponible la commande `list` permet d'afficher le code source avec chacune de ces lignes numérotées. Dans notre cas :
- `(gdb) list`

```
1       .data
2
3       UnNom :
4             .long 43
```

- La commande `break` permet de placer un point d'arrêt sur une instruction du programme source de manière à ce qu'à la prochaîne exécution du programme dans `gdb`, l'invite du dévermineur soit disponible avant l'exécution de cette instruction.
- Une instruction du programme source peut être repérée par le numéro de ligne correspondant ou par un label. Ainsi, la suite de commande :
- `(gdb) break 26`

```
Breakpoint 1 at 0x8048079: file sum.s, line 26.
(gdb) break top
Breakpoint 2 at 0x8048083: file sum.s, line 29.
```

- permet de placer deux points d'arrêts à deux endroits différents. On peut avoir la liste des points d'arrêts en utilisant dans `gdb` la commande `info` :

- (gdb) info break
```
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048079 sum.s:26
2   breakpoint     keep y   0x08048083 sum.s:29
```

- ***Exécution pas à pas***
- Une fois ceci fait, on peut exécuter notre programme dans l'environnement `gdb` :
- (gdb) run
```
Starting program: /home/.../Sources/sum

Breakpoint 1, _start () at sum.s:26
26              movl $0, %ebx      /* EBX va contenir la
somme de ces entiers  $  */
Current language:  auto; currently asm
(gdb)
```

- On constate que cette fois l'invite `(gdb)` se présente avant la fin normal du programme.
- Pour provoquer uniquement l'exécution de l'instruction `movl $0,%ebx`, on peut utiliser la commande `step` :
- (gdb) step
```
27              movl $UnNom, %ecx  /* ECX va << pointer >>
sur l'\'el\'ement $
(gdb)
```

- Pour provoquer l'exécution de l'ensemble des instructions comprises entre la position courante et le prochain point d'arrêt, on peut utiliser la commande `continue`. Ainsi, si on reprend le processus ci-dessus depuis le début :
- [soumam.lifl.fr-meftali-/.../Sources] gdb -q sum
```
(gdb) break 26
Breakpoint 1 at 0x8048079: file sum.s, line 26.
(gdb) break top
Breakpoint 2 at 0x8048083: file sum.s, line 29.
(gdb) run
Starting program: /home/.../Sources/sum
c
Breakpoint 1, _start () at sum.s:26
26              movl $0, %ebx      /* EBX va contenir la
somme de ces entiers     $  */
Current language:  auto; currently asm
(gdb) continue
Continuing.

Breakpoint 2, top () at sum.s:29
29      top:   addl (%ecx), %ebx
(gdb)
```

- En combinant ces commandes, on peut pas à pas exécuter toutes les instructions du programme jusqu'à sa fin naturelle. Il ne nous reste plus qu'à jeter un coup d'oeil dans les entrailles de la machine.
- Affichage du contenu des registres et de la mémoire

- Pour afficher le contenu d'un registre — disons %ecx, il faut déjà se placer en mode pas à pas et puis utiliser la commande `print` :
-
```
[soumam.lifl.fr-meftali-/home/.../Sources] gdb -q sum
(gdb) break 27
Breakpoint 1 at 0x804807e: file sum.s, line 27.
(gdb) run
Starting program: /home/.../Sources/sum

Breakpoint 1, _start () at sum.s:27
27              movl $UnNom, %ecx  /* ECX va << pointer >>
sur l'\'el\'ement
Current language:  auto; currently asm
(gdb) print $ecx
$1 = 0
(gdb) step
29      top:   addl (%ecx), %ebx
(gdb) print $ecx
$2 = 134516896
(gdb)  print /x $ecx
$3 = 0x80490a0
(gdb) printf "registre = %x en hexa et %d en decimale
\n", $ecx, $ecx
registre = 80490a0 en hexa et 134516896 en decimale
(gdb) info register
eax            0x5      5
ecx            0x80490a0        134516896
edx            0x0      0
ebx            0x2b     43
esp            0xbffff710       0xbffff710
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048085        0x8048085
eflags         0x200306 2097926
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
fctrl          0x37f    895
fstat          0x0      0
ftag           0xffff   65535
fiseg          0x0      0
fioff          0x0      0
foseg          0x0      0
fooff          0x0      0
fop            0x0      0
(gdb)
```

- Remarquez que l'on peut aussi utiliser la commande `printf` qui nous permet d'afficher le contenu du registre comme en C (ici en hexadécimal dans une chaîne de caractères). La commande `info register` fournie la liste des registres ainsi que leurs contenus.
- En utilisant les mêmes commandes, il nous est possible de déterminer le contenu du segment de données en mémoire à l'aide des labels :
-
```
(gdb) print UnNom
$4 = 43
(gdb) print &UnNom
$5 = (<data variable, no debug info> *) 0x80490a0
(gdb) print &UnNom
$6 = (<data variable, no debug info> *) 0x80490a0
(gdb) print *0x80490a0
$7 = 43
(gdb) print *(&UnNom)
$8 = 43
(gdb) print *(&UnNom+1)
$9 = 54
(gdb)
```

- Le préfixe & permet d'avoir accès à l'adresse de l'objet et le préfixe * au contenu d'une adresse. On peut ainsi parcourir l'espace mémoire — notez que l'arithmétique est propre aux pointeurs i.e. additionner 1 à notre adresse consiste à additionner le nombre d'octets qui codent l'objet en mémoires (ici 4 octets).
-

Remarque 8. Dans notre code source, le label `UnNom` du segment de données pointait sur une suite de données de type différents (des long et une chaîne de caractères). Ce choix s'avère ici peu judicieux car :

-
```
(gdb) print *(&UnNom+2)
$10 = 23
(gdb) print *(&UnNom+3)
$11 = 32
(gdb) print *(&UnNom+4)
$12 = 76
(gdb) print *(&UnNom+5)
$13 = 1819043176
(gdb)
```

- On obtient directement le bloc de données `hell` constitué de 4 octets sans pouvoir les distinguer directement. Il faut ruser un peu :
-
```
(gdb) print &UnNom+5
$13 = (<data variable, no debug info> *) 0x80490b4
(gdb) printf "%s\n",(&UnNom+5)
hello world
(gdb) printf "%c\n", *0x80490b4
h
(gdb) printf "%c\n", *0x80490b5
e
(gdb) printf "%c\n", *0x80490b6
```

Remarque 9. Plus généralement, il est possible d'obtenir l'affichage d'une zone mémoire grâce à la commande x :

```
(gdb) x /5dw 0x80490a0
0x80490a0 <UnNom>:      43      54      23      32
0x80490b0 <UnNom+16>:   76
(gdb) x /12cb 0x80490b4
0x80490b4 <UnNom+20>:   104 'h' 101 'e' 108 'l' 108 'l'
111 'o' 32 ' '  119 'w' 111 'o'
0x80490bc <UnNom+28>:   114 'r' 108 'l' 100 'd' 0 '\0'
```

- Pour plus d'information sur cette commande, utilisez l'aide en ligne de gdb.

Dans le chapitre suivant, nous reviendrons sur la structure de la mémoire.

Exercice 1
Écrire un programme permettant d'afficher des étoiles dans la disposition suivante :

```
*****
*****
*****
*****
*****
```

Construisez un programme assembleur qui affiche cette disposition.

Exercice 2 — Conditionnelles plus complexes.

Recommencez l'exercice précédent avec les dispositions suivantes :

```
*          *****      *****          *
**          ****      ****          **
***          ***      ***          ***
****          **      **          ****
*****          *      *          *****
```

Exercice 3 — Calcul de factorielle.

En utilisant l'exemple donné précédemment, écrire un programme qui calcule et stocke dans le registre %rbx la factorielle de 4.
Vérifier l'exécution de votre programme avec `gdb`.
Modifiez votre programme pour calculer la factorielle de 5 puis celle de 6.
Modifiez votre programme pour que la valeur dont il faut calculer la factorielle soit en mémoire (de données) et le résultat stocké en mémoire également.

Commandes GDB

gdb fichier : lancement de l'environnement gdb
quit : sortie de l'environnement gdb
Remarque.

Le raccourci clavier CTRL-C ne provoque pas la terminaison de gdb mais interrompt la commande courante.
Commandes générales
run : lancement d'un programme dans l'environnement gdb
kill : arrêt définitif d'un programme
Manipulation des points d'arrêt
break FCT : placer un point d'arrêt au début de la fonction FCT
break *ADDR : placer un point d'arrêt à l'adresse ADDR
break NUML : placer un point d'arrêt à la ligne NUML
disable NUM : inactive le point d'arrêt NUM
enable NUM : réactive le point d'arrêt NUM
delete NUM : supprime le point d'arrêt NUM
delete : supprime tous les point d'arrêts
Exécution d'un programme pas à pas
step : exécute une instruction élémentaire
step NUM : exécute NUM instructions élémentaires
next : exécute une instruction (y compris les fonctions appelées)
next NUM : exécute NUM instructions (y compris les fonctions appelées)
until LOC : exécute les instructions jusqu'à ce que LOC soit atteint
continue : reprend l'exécution
continue NUM : reprend l'exécution en ignorant les points d'arrêt NUM fois
finish : exécute jusqu'à ce que la fin de la fonction en cours
where : affiche la position actuelle
Affichage du code et des données
list : affiche le code source par paquet de 10 lignes
list NUML : affiche le code source à partir de NUML
disas : affiche le code autour de la position courante
disas ADDR : affiche le code autour l'adresse ADDR
disas ADDR1 ADDR2 : affiche le code entre les adresses ADDR1 et ADDR2
print $REG : affiche le contenu du registre REG
print /x $REG : affiche le contenu du registre REG en hexadécimal
print /t $REG : affiche le contenu du registre REG en binaire
print /c $REG : affiche le contenu du registre REG sous forme de caractère
print /a $REG : affiche le contenu du registre REG sous forme d'adresse

printf "DESC",OBJ : affichage à la C
x /NFU ADDR : affichage du contenu de la mémoire à l'adresse ADDR
N est le nombre d'unité à afficher
F est le format d'affichage
U indique le groupement :
b pour 1 octet,
h pour 2 octets et
w pour 4 octets
Commandes d'aide
help : affiche l'aide
info program : affiche des informations sur le programme

info functions : affiche la liste des fonctions définies
info variables : affiche les variables et les symbols prédéfinies
info registers : affiche les informations sur les registres
info breakpoints : affiche les informations sur les points d'arrêts