Chapitre 5

Les techniques de programmation en langage d'assemblage

Nous avons vu, dans le cours d'initiation à la programmation, quelles sont les structures de contrôle que l'on considère comme étant les plus naturelles. Comment les mettre en place en langage d'assemblage ?

5.1 Implémentation des sous-programmes

5.2 Implémentation des instructions conditionnelles

L'exemple typique d'utilisation d'une instruction conditionnelle est la fonction calculant le maximum de deux entiers. Commençons par écrire une telle fonction en langage C :
'long' est suffisant

```
long int max(long int x, long int y)
   {
   long int res;
   res=y;
   if (x>y)
       res=x;
   return res;
   }
```

puis un programme `callmax.c`, toujours en langage C, appelant cette fonction :
Commencer par fonction

```
int main(void)
   {
   long int res;
   res = max(5,3);
   exit(res);
   }
```

Compilons le code de `max` à la main :

```
long int max(long int x, long int y)
    {                                          max:
    long int res;                                   movq %rsi,%rax
    res=y;                                          cmpq %rdi,%rax
    if (res>x) goto L;                  ==>         jg   L
    res=x;                                          movq %rdi,%rax
 L: return res;                                 L:
    }
```

Faisons de même pour le programme principal :

```
_start:
        movq $3,%rsi
        movq $5,%rdi
        call max
        movq %rax,%rdi
        movq $60,%rax
        syscall
```

Reste à recoller toutes les parties ensemble. Pour que `max` soit traité correctement comme une fonction, on ajoute un « prologue » au début (deux lignes) et un « épilogue » à la fin (deux lignes également) :

```
max:
        pushq %rbp
        movq  %rsp,%rbp
        movq  %rsi,%rax
        cmpq  %rdi,%rax
        jg    L
        movq  %rdi,%rax
L:
        leave
        ret
```

Il faut aussi indiquer dans le programme principal que l'étiquette `max` marque le début d'une fonction (observez bien la syntaxe!) :

`callmax.s`

```
          .text
          .type  max, @function
          .globl _start

  _start:
          nop
          movq $3,%rsi
          movq $5,%rdi
          call max
          movq %rax,%rdi
          movq $60,%rax
          syscall
          nop

  max:
          pushq %rbp
          movq %rsp,%rbp
          movq %rsi,%rax
          cmpq %rdi,%rax
          jg   L
          movq %rdi,%rax
  L:
          leave
          ret
```

Assemblons, lions et testons le programme! Modifions les valeurs de `x` et de `y` et vérifions la valeur de retour ! Exécutons le programme pas à pas avec `gdb` en observant comment le contrôle passe du programme principal à la fonction `max` et comment on retourne au programme principal !

Question.- Que fait l'instruction `call` ?

Question.- Que font les deux lignes du prologue ?

Question.- Que fait l'instruction `leave` ?

Question.- Que fait l'instruction `ret` ?

Question.- Où la fonction `max` trouve-t-elle l'adresse de retour ?

[Pour trouver la réponse, inspectez la mémoire autour du sommet de la pile.]

Remarque.- On peut placer le code pour la fonction `max` et celui pour le programme principal dans les fichiers différents, les assembler séparément et les lier ensemble ensuite :

```
max.s
          .text
          .globl max
     max:
          pushq %rbp
          movq %rsp,%rbp
          movq %rsi,%rax
          cmpq %rdi,%rax
          jg   L
          movq %rdi,%rax
       L:
          leave
          ret
```

```
call-max.s
             .text
             .type  max, @function
             .globl _start
         _start:
             nop
             movq $3,%rsi
             movq $5,%rdi
             call max
             movq %rax,%rdi
             movq $60,%rax
             syscall
             nop
```

```
> as -o max.o max.s
> as -o call-max.o call-max.s
> ld -O0 max.o call-max.o
> ./a.out
> echo $?
5
```

5.3 Implémentation des boucles

Nous allons procéder d'une façon standardisée : convertir une boucle quelconque en une boucle `while` et exprimer cette dernière en utilisant des instructions conditionnelles (`if`) et des sauts (`goto`). La compilation pour obtenir le code en langage d'assemblage se fait alors comme ci-dessus. Lorsqu'on voit directement comment exprimer une boucle donnée par `if` et `goto` sans la convertir préalablement en une boucle `while` alors il ne faut pas hésiter !

Exemple 1.- Considérons un exemple simple :

```
int sloop(int llim, int rlim)
  {
  int cntr;
  for (cntr = llim; cntr <= rlim; cntr++)
      {
        <corps de la boucle>
      }
  return <resultat>;
  }
```

Ce que le corps de la boucle doit faire n'est pas tellement important pour notre objectif. Soit, par exemple :

```
int sloop(int llim, int rlim)
  {
  int x = 0;
  int cntr;
    for (cntr = llim; cntr <= rlim; cntr++)
        {
        x=x+cntr;
        }
    return x;
  }
```

Ajoutons :

```
int main(void)
  {
  int res
  res = sloop(1, 3);
  exit(res);
  }
```

On compile le programme principal à la main exactement comme dans l'exemple précedent. Le code de la fonction `sloop` peut d'abord être transformé de la façon suivante :

```
int sloop(int llim,int rlim)
  {
  int x = 0;
  int cntr;
  cntr = llim;
Loop:
  if (cntr > rlim)
    goto Fin;
  x = x + cntr;
  cntr++;
  goto Loop;
Fin:
  return x;
}
```

Les valeurs des variables `llim`, `rlim` sont les arguments de la fonction `sloop`. Elles sont placées dans les registres `%rdi` et `%rsi` respectivement. Attribuons les registres pour les autres variables (il faut avouer que l6 registres généraux nous laisssent une certaine liberté) :

`%rax` pour `x` et `%rcx` pour `cntr`.

Après la compilation à la main on obtient :

```
sloop.s
        .text
        .globl sloop
sloop:
        pushq %rbp
        movq %rsp,%rbp
        xorl %eax,%eax
        movl %edi,%ecx
Loop:
        cmpl %esi,%ecx
        jg Fin
        addl %ecx,%eax
        incl %ecx
        jmp Loop
Fin:
        leave
        ret
```

```
call-sloop.s
        .text
        .type sloop, @function
        .globl _start
_start:
        nop
        movl $1,%edi
        movl $3,%esi
        call sloop
        movl %eax,%edi
        movq $60,%rax
        syscall
        nop
```

```
> as -o sloop.o sloop.s
> as -o call-sloop.o call-sloop.s
> ld -O0 nnsloop.o call-sloop.o
> ./a.out
> echo $?
6
```

Question.- Pourquoi a-t-on utilisé `%eax`, `%ecx`, `%edi` et `%esi` et non `%rax`, `%rcx`, `%rdi` et `%rsi` comme dans l'exemple précedent ?

Remarque.- Au lieu de compiler à la main un programme en langage C, on peut demander à `gcc` d'engendrer le code en langage d'assemblage puis de s'arrêter après avec l'option `-S`.

Soit le programme `c-call-max.c` suivant :

```
#include <stdlib.h>
long int max(long int x, long int y)
  {
  long int res;
  res = y;
  if (x > y)
    res = x;
  return res;
  }
int main(void)
  {
  long int res;
  res = max(5,3);
  exit(res);
  }
```

```
> gcc -S c-call-max.c
```

Le compilateur `gcc` place alors le résultat de la compilation dans le fichier `c-call-max.s`. Comparez-le avec `callmax.s` ci-dessus ! Qu'observez-vous ?

5.4 Saisie des paramètres donnés sur la ligne de commande

Quand on programme en langage C c'est simple : on utilise les arguments de la fonction `main` dont les désignations « traditionnelles » sont `argc` (nombre de paramètres) et `argv` (tableau de chaînes de caractères reprenant les paramètres). Comment le faire en langage d'assemblage ? Ces chaînes de caractères doivent résider quelque part en mémoire. Chaque programme a deux repères principaux pour utiliser la mémoire : le sommet de la pile et la base du « *frame* ». Inspectons la région de la mémoire autour du sommet de la pile. Le débogueur `gdb` va nous y aider. Reprenons le programme `callmax.s` ci-dessus, assemblons-le et lions-le :

```
> as -gstabs -o callmax.o callmax.s
> ld -O0 callmax.o
> gdb a.out
(gdb) l 1,25
1
2 .text
3 .type  max, @function
4 .globl _start
5
6 _start:
7 nop
8 movq $3,%rsi
9 movq $5,%rdi
10 call max
11 movq %rax,%rdi
12         movq $60,%rax
13         syscall
14         nop
15
16 max:
17 pushq %rbp
18 movq %rsp,%rbp
19 movq %rsi,%rax
20 cmpq %rdi,%rax
21 jg   L
22 movq %rdi,%rax
23 L:
24 leave
25 ret
(gdb) b 8
Note: breakpoint 1 also set at pc 0x400079.
Breakpoint 2 at 0x400079: file callmax.s, line 8.
```

Question.- Pourquoi a-t-on placé le point d'arrêt à la ligne 8 ?

Pour démarrer le programme avec, par exemple, les paramètres `97` et `foo` sur la ligne de commande, on écrit :

```
(gdb) run 97 foo
```

ou bien on appuye sur la touche `F2` si on utilise `ddd`. On obtient alors :

```
Breakpoint 1, _start () at callmax.s:8
8 movq $3,%rsi
```

Notez que nous avons arrêté le programme *avant* qu'il fasse quoi que ce soit – c'est important. Regardons ce qu'il y a dans la pile à cet instant :

```
(gdb) i r rsp
 rsp         0x7fffffffdff0
(gdb) x /8gx 0x7fffffffdfe0
  0x7fffffffdfe0: 0x0000000000000000 0x0000000000000000
  0x7fffffffdff0: 0x0000000000000003 0x00007fffffffe2db
  0x7fffffffe000: 0x00007fffffffe30b 0x00007fffffffe30e
  0x7fffffffe010: 0x0000000000000000 0x00007fffffffe312
```

Le sommet de la pile contient alors la valeur 3, qui est la valeur de `argc`. Regardons ce qu'on trouve à partir de l'adresse `0x00007fffffffe2db`, c'est-à-dire la valeur de l'élément précédant le sommet :

```
(gdb) x /64bx 0x00007fffffffe2db
0x7fffffffe2db: 0x2f 0x68 0x6f 0x6d 0x65 0x2f 0x76 0x65
0x7fffffffe2e3: 0x72 0x6b 0x6f 0x2f 0x41 0x72 0x63 0x68
0x7fffffffe2eb: 0x69 0x2d 0x74 0x65 0x61 0x63 0x68 0x2f
0x7fffffffe2f3: 0x36 0x34 0x62 0x69 0x74 0x73 0x2f 0x41
0x7fffffffe2fb: 0x73 0x6d 0x36 0x34 0x47 0x75 0x69 0x64
0x7fffffffe303: 0x65 0x2f 0x61 0x2e 0x6f 0x75 0x74 0x00
0x7fffffffe30b: 0x39 0x37 0x00 0x66 0x6f 0x6f 0x00 0x43
0x7fffffffe313: 0x50 0x4c 0x55 0x53 0x5f 0x49 0x4e 0x43
```

Cela ressemble à des codes ASCII. Vérifions-le :

```
(gdb) x /4bs 0x00007fffffffe2db
0x7fffffffe2db:  "/home/verko/Archi-teach/64bits/Asm64Guide/a.out"
0x7fffffffe30b:  "97"
0x7fffffffe30e:  "foo"
0x7fffffffe312:  "CPLUS_INCLUDE_PATH=/usr/lib64/qt/include:/usr/lib64/qt/include"
```

Gagné ! Ce sont exactement `argv[0], argv[1], argv[2]` suivis de la liste des variables d'environnement. Nous pouvons maintenant conclure que, juste après le démarrage d'un programme en langage d'assemblage, `Cont((%rsp))` est égal à `argc`, `argv[0]` est la suite d'octets commençant à l'adresse `8(%rsp)` et se terminant par l'octet `0x00`, `argv[1]` est la suite d'octets commençant à l'adresse `16(%rsp)` et se terminant par l'octet `0x00`, etc.

Question.- Comment charger l'adresse de début de `argv[1]` dans le registre `%rbx` ?

Question.- Comment calculer la longueur de `argv[1]` (et placer le résulat dans le registre `%rcx`) ?

5.5 Saisie et affichage d'un entier

5.5.1 Passage de la représentation ASCII à la représentation machine

Dans l'exemple ci-dessus `argv[1]` est une suite de chiffres décimaux représentant un entier, à savoir 97. Comment obtenir la représentation machine de cet entier sur un registre, par exemple `%rax` ? En langage C, on utilise dans ce but les fonctions `strtul()`, `strtoll()`, `strtoq()` ou bien le bon vieux `atoi()`. Écrivons une version simplifiée de cette dernière fonction en langage d'assemblage (appelons-la `readint()`).

On commence toujours par décrire *un algorithme*. En quoi cela consiste-t-il ? Supposons (pour simplifier les choses) que l'entier sur la ligne de commande soit une entier naturel. Nous pouvons lire la représentation ASCII de cet entier chiffre par chiffre (cest-à-dire octet par octet) *à partir du chiffre de plus fort poids* (on connaît l'adresse de début) en s'arrêtant dès qu'on voit l'octet `0x00`. Rappelons que la valeur numérique d'un chiffre, entre '`0`' et '`9`' est égale à son code ASCII hexadécimal moins `0x30` (vérifiez-le en consultant la table ASCII !). Soit une représentation $c_1 c_2 \dots c_{n-1} c_n \ \ (n \geq 1)$ en décimal. Considérons la suite (méthode de Hörner) :

$$S_0 = 0,\ S_i = S_{i-1} \times 10 + c_i \ (i = 1, \dots, n).$$

On voit que S_n est la valeur numérique du nombre dont on considère la représentation ASCII.

Pour écrire un programme, il ne reste plus qu'à désigner les rôles des registres (et de la pile !) et à indiquer où la fonction prend son argument (l'adresse d'une chaîne de caractères). Faites-le !

Exercice.- Écrire en langage d'assemblage :

1. une fonction `readint.s` implémentant l'algorithme ci-dessus ;
2. un programme « principal » `tst-readint.s` saisissant la représentation d'un entier en ASCII sur la ligne de commande et appelant `readint` pour le convertir en représentation machine. Le résultat doit se trouver dans le registre `%rax`.

5.5.2 Passage de la représentation machine à la représentation ASCII

Pour compléter notre bibliothèque sur les entiers, il nous faut une fonction convertissant un entier se trouvant dans un registre (disons `%rax`) en sa représentation ASCII *décimale* (une chaîne de caractères) et l'affichant.

L'algorithme est simple. Il suffit d'effectuer la division euclidienne du nombre à convertir par dix (la base). Le reste dans cette division euclidienne nous donne le chiffre *de plus faible poids*. On lui ajoute `0x30` pour obtenir son code ASCII. On prend le quotient dans la division euclidienne (s'il est différent de `0`) comme nouveau nombre à convertir et on continue le processus. On s'arrête lorsque le quotient est `0`.

On va, donc, accumuler une chaîne caractère par caractère. On ne peut pas l'afficher à la volée car elle est dans l'ordre inverse de ce que nous voulons afficher.

Où pouvons-nous placer temporairement cette chaîne de caractères ? Par exemple sur la pile.

Notons qu'on ne peut pas utiliser `push` pour mettre un code ASCII sur la pile.

Question.- Pourquoi ?

Question.- Existe-t-il un autre emplacement raisonnable pour cette chaîne des caractères qui s'accumule ? Lequel ?

Rappelons-nous que l'objectif est d'afficher la représentation décimale d'un entier naturel.

Question.- Quel appel système utiliser pour l'affichage ?

Question.- Quels sont les arguments de cet appel et où faut-il les placer ?

Exercice.- Écrire en langage d'assemblage :

1. Une fonction `printint.s` implémentant l'algorithme ci-dessus, affichant en décimal le contenu du registre `%rax`.
2. Un programme « principal » `tst-printint.s` plaçant un entier dans `%rax` et appelant `printint`.