Chapter 1

Compilation et Linkage en Assembleur x86

1.1 Introduction

En assembleur Linux 32 bits, un programme passe par plusieurs étapes avant de devenir exécutable :

1. **Ecriture du code source** (`programme.asm`)
2. **Assemblage** : NASM produit un fichier objet (`.o`)
3. **Linkage** : le linker (`ld`) relie les fichiers objets et crée l'exécutable.

> **Note.** petite analogie :
>
> - **Code source** = manuscrit
> - **Fichier objet (.o)** = pages imprimées
> - **Linkage** = reliure des pages pour former le livre (exécutable)

1.2 NASM : l'assemblage

Pour assembler un fichier source, on utilise la commande :

```
nasm -f elf32 programme.asm -o programme.o
```

Explications :

- `-f elf32` : indique à NASM de produire un fichier objet au format ELF 32 bits pour Linux
- `programme.asm` : fichier source assembleur
- `-o programme.o` : nom du fichier objet généré

1.3 Linker : création de l'exécutable

Pour transformer le fichier objet en programme exécutable :

```
ld -m elf_i386 programme.o -o programme
```

Explications :

- `-m elf_i386` : architecture cible 32 bits Intel x86
- `programme.o` : fichier objet à lier
- `-o programme` : nom de l'exécutable final

> **Note. Remarques importantes :**
>
> - Les options `-f` et `-m` doivent correspondre à la même architecture (32 bits ou 64 bits)
> - Pour Linux 64 bits, utiliser `-f elf64` et `-m elf_x86_64`
> - Sans linker, le fichier objet `.o` ne peut pas etre executé directement

1.3.1 Exécution

```
./programme
```

1.4 Schéma pédagogique

```
Code source (programme.asm)
        |
        v
nasm -f elf32 -> programme.o (fichier objet 32 bits)
        |
        v
ld -m elf_i386 -> programme (executable)
```

1.5 Résumé

- **NASM** : traduit le code assembleur en code machine dans un fichier objet
- **ld (linker)** : relie les fichiers objets, résout les adresses et symboles, produit un exécutable
- **-f** (NASM) et **-m** (ld) : options pour définir le format et l'architecture

Chapter 2

Structure d'un programme assembleur x86 (format ELF 32 bits) sous Linux.

2.1 Introduction

Ce rapport explique étape par étape comment écrire, assembler, lier et exécuter un programme en assembleur x86 (format ELF 32 bits) sous Linux.

2.2 1. Structure d'un programme assembleur

Un programme assembleur x86 contient généralement :

- une section **.data** : pour les variables statiques
- une section **.bss** : pour les variables non initialisées
- une section **.text** : pour le code exécutable
- une étiquette **_start** : point d'entrée du programme
- **global _start** : indique au linker le point d'entree du programme.

Exemple minimal :

```
section .text
global _start

_start:
    ; instructions ici
```

Pour quitter proprement :

```
mov eax, 1      ; syscall exit
mov ebx, 0      ; code retour
int 0x80
```

2.3 Commentaire en Assembleur

Dans le langage assembleur NASM, un commentaire commence par le caractere ; et se termine a la fin de la ligne.

```
; Ceci est un commentaire
mov eax, 4   ; Ceci aussi
```

2.4 La directive DB

La directive `db` signifie "define byte". Elle permet de reserver des octets en memoire et d'y placer du contenu.

```
msg db "hello", 0x0A
```

Ici :
- les caracteres de "hello" sont places en memoire
- `0x0A` represente le caractere saut de ligne

2.5 Le calcul de longueur avec EQU

La valeur `$` represente "l'adresse courante".

```
msg db "hello", 0x0A
msg_len equ $ - msg
```

Cela calcule : longueur = adresse courante - adresse de msg.

Remarques Si tu ecris une autre variable juste apres msg, alors $ sera place apres cette nouvelle variable. La longueur ne sera plus uniquement poour msg.

2.6 Appel systeme avec INT 0x80

2.6.1 Principe

L'instruction `int 0x80` demande au noyau Linux d'executer une fonction interne appelee "appel systeme" (system call).

Le numero du syscall est place dans EAX.

Les parametres sont places dans EBX, ECX, EDX (respectivement 1er, 2eme et 3eme argument).

2.6.2 Table des appels systeme utilises ici

- write = 4
- exit = 1
- read = 3

2.7 Les registres x86

Les registres sont des petites zones mémoire ultra rapides, utilisées pour stocker temporairement des valeurs.

- **EAX** : numéro d'appel système (syscall) + retour
- **EBX** : 1er argument
- **ECX** : 2^{e} argument
- **EDX** : 3^{e} argument
- **ESI**, **EDI** : pointeurs
- **ESP**, **EBP** : pile (stack)

2.8 Premier programme : afficher un message

Nous utilisons `sys_write` (numéro 4).

```
; sys_write(user: eax=4, ebx=1, ecx=adresse, edx=longueur)

section .data
msg db "Bonjour, monde!", 0x0A
len equ $ - msg

section .text
global _start

_start:
    mov eax, 4      ; syscall write
    mov ebx, 1      ; stdout
    mov ecx, msg    ; adresse
    mov edx, len    ; longueur
    int 0x80        ; appel syst me

    mov eax, 1      ; sys_exit
    xor ebx, ebx    ; code retour
    int 0x80
```

2.9 Lire une chaîne depuis stdin

Utiliser `sys_read` (numero 3).

```
section .bss
input_buf resb 128   ; reserver 128 octets

section .text
    mov eax, 3        ; sys_read
    mov ebx, 0        ; fd = 0 (stdin)
```

```
    mov ecx, input_buf
    mov edx, 127       ; lire au max 127 octets
    int 0x80
    ; eax = nombre d'octets lus
    ; mettre les instructions de retour
```

Apres l'appel, `eax` contient le nombre d'octets lus (incluant " si l'utilisateur a presse Entree). Tu peux ensuite utiliser ce buffer et sa longueur pour l'affichage (sys_write).

2.10 Exemple : lire puis afficher

```
section .bss
input resb 128

section .text
global _start

_start:
    ; lire
    mov eax, 3
    mov ebx, 0
    mov ecx, input
    mov edx, 128
    int 0x80
    mov esi, eax        ; esi = nb octets lus

    ; afficher (utilise eax=nb octets)
    mov eax, 4
    mov ebx, 1
    mov ecx, input
    mov edx, esi
    int 0x80

    ; exit
    mov eax, 1
    xor ebx, ebx
    int 0x80
```