Documentation OS

September 15, 2024

Chapter 1

Introduction

On suppose pour toute la suite que le système utilise un processeur Intel 32 bits.

1.1 Émulation

Pour compiler :

```
./building.sh
```

Pour émuler en disquette :

```
qemu-system-x86_64 -boot a -drive file=floppyA,format=raw,index=0,media=disk
```

1.2 Normes de programmation

Les normes décrites ci-dessous ont comme principalement but d'assurer la stabilité des programmes écrits, la lisibilité des codes et ainsi la facilitation du déboguage.

L'unique langage de programmation utilisé est l'assembleur NASM (`https://www.nasm.us/`).

1.2.1 Conventions pour la modularité

Tout module est composé de deux fichiers :

- un fichier de code d'extension *.ASM*
- un fichier d'entête d'extension *.INC*

Le fichier d'entête est systématiquement inclu par directive de préprocesseur au début du fichier de code. Après l'écriture des directives de préprocesseur et avant le code lui-même, le fichier de code doit avoir une ligne *[BITS 32]* spécifiant à l'assembleur que l'assemblage doit être fait en 32 bits.

Le fichier d'entête doit se charger en totalité de l'import des fonctions externes nécessaires au fichier de code, de la définition des constantes, et de la globalisation des fonctions exportés par le module (et aucune autre).

Remarque : Toutes les fonctions externalisées doivent être nommés selon un espace de nom sous la forme : *ESPACE__FONCTION*.

Exemple :

module.inc

```
%define CONST_INUTILE 0x1234

EXTERN screen__print
GLOBAL module__helloworld ; void -> void
```

module.asm

```
%include "module.inc"

[BITS 32]

msg db "Hello World !", 10, 0

; Fonction helloworld : void -> void
; Affiche le message "Hello World !" sur l'écran
module__helloworld:
        push ebp
        mov ebp, esp
                push msg
                call screen__print
        mov esp, ebp
        pop ebp
ret
```

Dans le code ci-dessus, *screen* et *module* sont des espaces de nom. **Import de fonctions externes :** Toutes les fonctions externes d'un même espace de nom sont importés sur la même ligne :

```
EXTERN espace__f1, espace__f2, espace__f3 ; etc...
```

Globalisation : Toutes les fonctions globalisées le sont sur des lignes séparées, toujours suivies par leur signature en commentaire :

```
GLOBAL math__add ; int a, int b -> int
GLOBAL math__exp ; double x, unsigned int n -> double
GLOBAL math__abs ; int n -> unsigned int
```

Commentaire des fonctions : Le fichier de code doit comprendre la description de chaque fonction en préambule de celle-ci :

```
; Fonction nom_de_la_fonction_sans_espace_de_nom : TYPE a1, ..., TYPE an -> TYPE
; Description de l'action effectuée par cette fonction
; Parametres
; - a1 : sens de a1
; ...
```

```
; - an : sens de an
; Retour
; Sens du retour
espace__nom_de_la_fonction_sans_espace_de_nom:
        ; CODE DE LA FONCTION
ret
```

1.2.2 Conventions d'appel de fonction

Soient $n, k \in \mathbb{N}$ et f une fonction *n-aire* de paramètres $(p_1, \ldots, p_n)$.
Les arguments lors de l'appel de cette fonction doivent être poussés sur la pile d'exécution du processus en cours dans l'ordre inverse des paramètres, de sorte que les adresses en pile correspondant à chaque paramètre conservent l'ordre de ceux-ci.

La valeur de renvoi de la fonction suit la norme du langage C :

- Si la valeur de renvoi est stockée sur au plus 32 bits, elle est contenu dans EAX
- Si la valeur de renvoi est stockée sur au plus 64 bits, elle est contenu dans $(EDX : EAX)$

Exemple :

```
section .text

_start:

        push dword 0 ; Argument 2
        push dword 0 ; Argument 1
        call fonction
        ; EAX contient le résultat

end:
        jmp end
```

1.2.3 Conventions d'écriture de fonction

Avant de créer une nouvelle trame d'exécution, on pousse EAX en mémoire pour avec les 4 octets nécessaires à la valeur de renvoi. L'adresse en pile d'exécution de ce bloc de 4 octets contiendra la valeur de renvoi.

On crée ensuite une trame d'exécution :

```
push ebp
mov ebp, esp
        ; CODE DE LA FONCTION
mov esp, ebp
pop ebp
```

Cependant, dans l'ordre actuel des choses il y a plusieurs problèmes :

- Le bloc de renvoi n'a pas été réassigné à EAX
- Les blocs de renvoi et d'arguments en pile d'exécution n'ont pas été pop et surchargent donc la pile d'exécution

On règle ces deux problèmes d'un coup à la suite de la sortie de la trame :

```
mov eax, [esp + 4]
mov [esp + N], eax ; N = 4 * n
pop eax
add esp, M ; M = 4 * (n - 1) = N - 4 ; Cette ligne peut être supprimée si M = 0
```

Remarque : Ces problèmes n'existent que si $n > 1$. D'autre part, l'ajout des lignes juste ci-dessus implique un bug dans le cas où $n = 0$ (pas de paramètres). Pour cette raison, dans le cas où $1 \geq n \geq 0$, on écrira seulement :

```
pop eax
```

Remarque : Si la fonction ne renvoie rien, il n'est pas *tout le temps* nécessaire de créer un espace pour la valeur de renvoi. Cependant, il est très régulier que les fonctions écrites utilisent le registre EAX. À ce moment là, la création d'un espace pour la valeur de renvoi est équivalent à la mémorisation de EAX.

En effet, il est souhaitable que la fonction écrite ne modifie pas les registres qu'elle utilise. Ceux-ci sont donc sauvegardés puis rechargés respectivement au début et à la fin de la trame nouvellement créée.

Remarque : Il n'est pas utile de mémoriser une nouvelle fois la valeur du registre EAX, voir remarque précédente.

Exemples :

```
section .text

_start:

        push dword 20
        push dword 10
        call fonction_2_parametres_ou_plus

        push dword 42
        call fonction_1_parametre_ou_moins

end:
        jmp end

fonction_2_parametres_ou_plus:
        push eax ; crée un espace sur la pile pour la valeur de renvoi
        push ebp
        mov ebp, esp
        push ebx ; sauvegarde du registre EBX
                mov eax, [ebp + 12] ; Le premier argument vaut 10
                mov ebx, [ebp + 16] ; Le second argument vaut 20
                add eax, ebx
                mov [ebp + 4], eax ; Valeur de retour modifiée
        pop ebx ; chargement du registre EBX
        mov esp, ebp
        pop ebp
        mov eax, [esp + 4]
```

```
28          mov [esp + 8], eax
29          pop eax
30          add esp, 4
31  ret
32
33  fonction_1_parametre_ou_moins:
34          push eax
35          push ebp
36          mov ebp, esp
37                  mov eax, [ebp + 12]
38                  inc eax
39                  mov [ebp + 4], eax
40          mov esp, ebp
41          pop ebp
42          pop eax
43  ret
```

Ainsi, comme les registres sont de 32 bits soit 4 octets, les arguments sont aux adresses suivantes :

- $[ebp + 12]$, pour p_1
- $[ebp + 16]$, pour p_2
- ...
- $[ebp + 8 + 4 * i]$, pour p_i le i^e argument donné à la fonction

1.3 BootSector

Chapter 2

Kernel

2.1 Global Descriptor Table

2.1.1 Descripteur de segment

2.1.2 Table des descripteurs

2.2 Interruptions Descriptor Table

2.3 Microcontrôleurs PIC

2.4 Pagination

2.4.1 Buddy Memory Allocation

Minimal :

Nombre minimal de bits nécessaire pour le tableau de mapping : (où p désigne le nombre de niveaux)

$$N_{bits}(p) = \lceil log_2(p+2) \rceil + \sum_{i=0}^{p} \left(2^{p-i} \lceil log_2(i+1) \rceil \right)$$

En particulier comme on manipule 2^{20} blocs, il faut $N_{bits}(20) = 1712160$, soit $N_{octets}(20) = \dfrac{N_{bits}(20)}{8} = 214020$

En effet, il doit être précisé pour chaque case quel est le niveau de réservation. La première case peut réserver jusqu'à 20 niveaux. La case du milieu jusqu'à 19 niveaux, etc... Il faut donc stocker chacune des cases sur un nombre de bits différents.

Utilisé :

On note l'ordre d'un noeud nd d'un arbre t :

$$o(nd) = h(t) - h(nd), \text{ où } h \text{ est la fonction hauteur}$$

Propriété 1. Si l'arbre est complet, $2^{o(nd)}$ est le nombre de feuilles découlant de ce noeud.

On construit un arbre pour les 2^{20} blocs. On a donc 2^{21} noeuds. Chaque noeud contient le plus grand entier p tel que les pages d'un sous-noeud d'ordre p soient libres.

On a donc besoin de (où 2^p est le nombre de pages) :

$$N_{bits}(2^p) = 2^p + \sum_{i=1}^{p} 2^{p-i} \lceil log_2(i+1) \rceil$$

En particulier, pour 2^{20} pages, on a besoin de 2760731 bits, soit 345092 octets.

2.5 Appels systèmes