< Introduction | TutoOS | Réaliser un secteur de boot qui charge et exécute un noyau >

Réaliser un secteur de boot qui affiche un message

La programmation d'un noyau est difficile et ce tutoriel s'adresse en premier lieu aux programmeurs ayant déjà une bonne expérience de la programmation et une connaissance suffisante de la théorie des systèmes d'exploitation. L'ensemble des pré-requis nécessaires à la bonne compréhension de ce tutoriel est détaillé dans l'introduction. Si vous débutez en C, que vous ne connaissez pas l'assembleur ou bien que vous n'avez qu'une très faible idée des fonctionnalités d'un noyau, il vous sera donc très difficile de tout saisir. Quoiqu'il en soit, je vous souhaite une excellente lecture et une bonne compilation !

Qu'est-ce qu'un secteur de boot ?

Un **secteur de boot** est un programme situé sur le premier secteur d'une unité de stockage, et qui est chargé et exécuté au démarrage du PC. Le programme de ce secteur a en principe pour tâche de charger un noyau en mémoire et de l'exécuter. Ce noyau peut être présent au depart sur une disquette, un disque dur, une bande ou tout autre support magnétique. Ce chapitre détaille ce qui se passe quand on boote sur disquette mais les principes expliqués ici restent valables pour tout autre support.

Comment est chargé le secteur de boot au démarrage de l'ordinateur ?

Au démarrage, le PC commence par initialiser et tester le processeur, la mémoire et les périphériques. C'est le *Power On Self Test* (POST) . Ensuite, le PC charge et exécute un programme particulier résidant en ROM, le BIOS ("Basic Input/Output System" ou "Built In Operating System"). Le BIOS peut être vu comme un système d'exploitation minimal chargé automatiquement par le PC et dont l'un des rôles est de charger un véritable système d'exploitation en essayant de trouver un secteur de boot valide parmi les unités de stockage. Une fois trouvé, ce secteur est chargé en memoire à l'adresse `0000:7C00` puis le BIOS passe la main au programme de boot fraichement chargé.

Le secteur de boot de l'unité de stockage est appelé **Master Boot Record** (MBR). Il contient des données, dont la **table des partitions**, et du code exécutable. La table des partitions contient des informations sur les partitions primaires du disque (où elles commencent, leur taille, etc.). Le code exécutable d'un MBR standard cherche sur la table des partitions une partition active, puis, si une telle partition existe, ce code charge le secteur de boot de cette partition (appelé aussi VBR). C'est souvent ce deuxième secteur de boot qui charge le noyau et lui donne la main.

Créer un secteur de boot qui affiche un message

Dans notre cas, nous allons pour commencer créer un secteur de boot très simple qui ne fera qu'afficher un message de bienvenue. Celà n'est pas bien difficile et pour vous épargner un long suspense, voici le programme :

```
[BITS 16]  ; indique a nasm que l'on travaille en 16 bits
[ORG 0x0]

; initialisation des segments en 0x07C00
    mov ax, 0x07C0
    mov ds, ax
    mov es, ax
    mov ax, 0x8000
    mov ss, ax
    mov sp, 0xf000    ; stack de 0x8F000 -> 0x80000

; affiche un msg
    mov si, msgDebut
    call afficher


end:
    jmp end


;--- Variables ---
    msgDebut db "Hello world !", 13, 10, 0
;-----------------

;-------------------------------------------------------
; Synopsis: Affiche une chaine de caracteres se terminant par 0x0
; Entree:   DS:SI -> pointe sur la chaine a afficher
;-------------------------------------------------------
afficher:
    push ax
    push bx
.debut:
```

```
        lodsb           ; ds:si -> al
        cmp al, 0       ; fin chaine ?
        jz .fin
        mov ah, 0x0E    ; appel au service 0x0e, int 0x10 du bios
        mov bx, 0x07    ; bx -> attribut, al -> caractere ascii
        int 0x10
        jmp .debut

    .fin:
        pop bx
        pop ax
        ret

    ;--- NOP jusqu'a 510 ---
        times 510-($-$$) db 144
        dw 0xAA55
```

[Get Code]

Que fait exactement ce programme ?

```
    [BITS 16]  ; indique a nasm que l'on travaille en 16 bits
```

On indique tout d'abord que l'on travaille sur 16 bits pour le codage des instructions et des données, c'est le mode par defaut. Ceci n'est pas encore le début du programme, c'est juste une directive de compilation pour que l'exécutable obtenu soit bien sur 16 bits et pas sur 32 bits.

```
    [ORG 0x0]
```

Cette directive indique l'offset à ajouter à toutes les adresses référencées.

```
    ; initialisation des segments en 0x07C00
        mov ax, 0x07C0
        mov ds, ax
        mov es, ax
```

Le programme du secteur de boot est chargé par le BIOS en `0x7C00`, donc toutes les données internes au programme sont situées à partir de cette adresse. Le bloc ci-dessus initialise les registres `ds` et `es` qui servent à indiquer où débute le segment de données. En mettant la valeur `0x07C0` dans le sélecteur de données, on accède en fait à l'adresse `0x07C0:0000` = `0x07C00`.

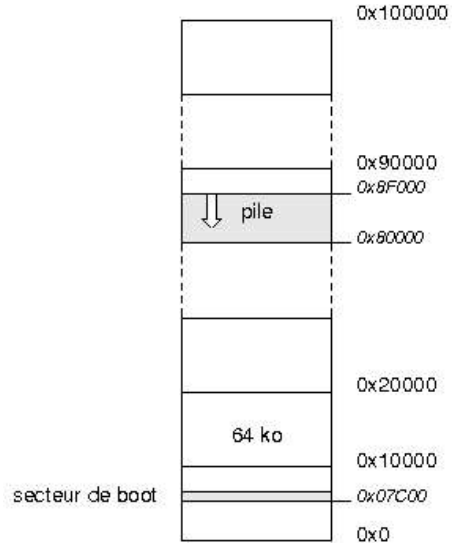
> L'adressage en mode réel est particulier. La valeur du sélecteur représente les 16 bits "hauts" d'une adresse linéaire sur 20 bits. Il faut donc multiplier par 0x10 la valeur du sélecteur pour obtenir la base. Par exemple, l'adresse `A000:1234` en mode réel correspond à l'adresse physique `A0000` + `1234` = `A1234`. Celà signifie aussi qu'un segment en mode réel occupe 64k, et pas un octet de plus !

```
    ; stack de 0x8F000 -> 0x80000
        mov ax, 0x8000
        mov ss, ax
        mov sp, 0xf000
```

Ensuite on initialise le segment de pile `ss` et le pointeur de pile `sp` en faisant commencer la pile en `0x8F000` et en la faisant finir en `0x80000`. Note : le choix de ces valeurs est arbitraire, dans le cas présent, la pile aurait pu être placée ailleurs.

À ce stade du programme :

- le secteur de boot est chargé en `0x07C00`
- le segment de données est initialisé pour couvrir les adresses de `0x07C00` à `0x17C00`
- la pile commence en `0x8F000` et finit en `0x80000`




```
    ; affiche un msg
        mov si,msgDebut
```

```
        call afficher
```

Une fois les principaux registres initialisés, la fonction `afficher` est appelée pour mettre à l'écran le message pointé par `msgDebut`. Cette fonction fait appel au BIOS pour gèrer l'affichage du message. Plus précisement, elle appelle le service `0x0e` de l'interruption logicielle `0x10` du BIOS qui permet d'afficher un caractère à l'écran en précisant ses attributs (couleur, luminosité...). Il aurait été possible d'écrire ce message à l'écran en se passant de cette facilité offerte par le BIOS mais cela aurait été beaucoup moins simple à réaliser (c'est d'ailleurs l'objet du chapitre suivant... patience !).

```
    end:
        jmp end
```

Une fois le message affiché, le programme boucle et ne fait plus rien.

```
    msgDebut db "Hello world !", 13, 10, 0
```

En fin de fichier, on définit les variables et les fonctions utilisées dans le programme. La chaîne de caractères affichée au demarrage à pour nom `msgDebut`.

Ensuite, la fonction `afficher` est définie. Elle prend en argument une chaîne de caractères pointée par les registres `DS` et `SI`. `DS` correspond à l'adresse du segment de données et `SI` est un déplacement par rapport au début de ce segment (un offset). La chaîne de caractère passée en argument doit se terminer par un octet égal à 0 (comme en C).

```
    ;--- NOP jusqu'a 510 ---
        times 510-($-$$) db 144
        dw 0xAA55
```

Cette directive ajoute du bourrage sous forme d'instructions NOP (opcode 0x90, ou 144), puis le mot `0xAA55` à la fin du code compilé afin que le binaire généré fasse 512 octets. Le mot `0xAA55` est crucial : en fin de secteur, il est une signature pour indiquer au BIOS que le secteur en question est un MBR !

Compiler et tester le programme

Le programme est écrit dans le fichier `bootsect.asm`. Pour le compiler avec `nasm` et obtenir le binaire (exécutable au boot) `bootsect`, il faut utiliser la commande suivante :

```
$ nasm -f bin -o bootsect bootsect.asm
```

Ensuite, pour tester notre secteur de boot, nous pourrions l'écrire sur une disquette et rebooter notre machine préférée avec la disquette dedans. Mais cette démarche est très fastidieuse car un PC met toujours du temps à rebooter et les possibilites de débogage sont très limitées à ce stade. Heureusement, il existe de très bons émulateurs de PC, tels que bochs ou qemu . Pour ma part, j'utilise *bochs* qui, contrairement à *qemu*, possède un mode debug puissant.

Bochs a la possibilité d'utiliser un fichier en faisant comme si c'était une vraie disquette. Sous Unix, pour générer une image de disquette :

```
$ cat bootsect /dev/zero | dd of=floppyA bs=512 count=2880
```

> Je ne sais pas trop comment tout cela peut être fait sous Windows (je ne travaille que sous Debian GNU/Linux). En revanche, vous trouverez d'autres développeurs Windows sur l'excellent forum Osdev.org .

Ensuite, la commande pour démarrer sur cette disquette virtuelle avec Bochs devrait ressembler à ça :

```
$ bochs 'boot:a' 'floppya: 1_44=floppyA, status=inserted'
```

> Il est assez peu probable que Bochs fonctionne du premier coup si vous ne vous êtes pas donné la peine de lire sa documentation pour créer un fichier de configuration qui lui soit intelligible. Je sais, ça n'est pas la partie la plus passionnante, mais elle est hélas indispensable !

Pour ceux qui préfèrent utiliser qemu , il faut taper la commande suivante :

```
$ qemu -boot a -fda floppyA
```

Le résultat obtenu avec bochs :

< Introduction **|** TutoOS **|** Réaliser un secteur de boot qui charge et exécute un noyau **>**