Programmer un secteur de boot qui passe en mode protégé

Sources

Le package contenant les sources est téléchargeable ici : bootsect_PMode.tgz .
Pour naviguer dans l'arborescence : BootSector_PMode

Pourquoi utiliser le mode protégé ?

Le mode protégé, c'est quoi ?

Le microprocesseur d'un PC possède trois modes de fonctionnement :

- Le **mode réel**, qui est le mode par défaut, fournit les mêmes fonctionnalités que le 8086. Mais cela a certaines conséquences comme l'impossibilité d'adresser plus de 1 Mo de mémoire.
- Le **mode protégé** (voir aussi sur Wikipédia) permet d'exploiter la totalité des possibilités du microprocesseur, avec notament l'addressage de toute la mémoire et le support pour l'implémentation de systèmes multi-tâches et multi-utilisateurs.
- Le **mode virtuel** est un mode particulier très peu utilisé.

Notre objectif étant de réaliser un noyau multi-utilisateurs, multi-tâches et pouvant adresser toute la mémoire, il nous faudra basculer le microprocesseur en mode protégé. Mais cela a d'importantes conséquences pour le programmeur :

- le mécanisme d'adressage en mode protégé est très différent de celui en mode réel
- le jeu d'instruction n'est plus sur 16 bits mais sur 32 bits
- il n'est pas possible avec ce mode de s'appuyer sur les routines du BIOS pour accéder aux péripheriques. Tous les drivers sont donc à réécrire !

Comment passer du mode réel au mode protégé

Passer du mode réel au mode protégé est très simple, il suffit juste de mettre le bit 0 du registre `CR0` à 1 :

```
; PE mis a 1 (CR0)
    mov eax, cr0
    or  ax, 1
    mov cr0, eax
```

Mais si cela suffit pour changer de mode, cela ne suffit pas à ce qu'un programme continue de fonctionner une fois le mode protégé établi. Pourquoi ? L'adressage en mode protégé, qui diffère de l'adressage en mode réel, s'appuie sur des structures qui doivent etre correctement initialisées lors du changement de mode pour que le processeur continue d'adresser correctement la mémoire (et notamment le segment de données, la pile et le pointeur d'instruction).

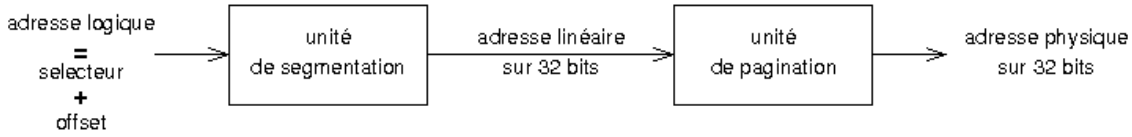
Addresser la mémoire en mode protégé

Différents types d'adresses

En mode protégé, il existe pour le programmeur trois types d'adresses :

- L'**adresse logique** est directement manipulée par le programmeur. Elle est composée à partir d'un **sélecteur de segment** et d'un **offset**.
- Cette adresse logique est transformée par l'unité de segmentation en une **adresse linéaire**, sur 32 bits.
- Cette adresse linéaire est transformée par l'unité de pagination en une **adresse physique**. Si la pagination n'est pas activée, l'adresse linéaire correspond à l'adresse physique.

Le schéma ci-dessous résume le principe de l'adressage en mode protégé :



Dans un premier temps, nous allons utiliser uniquement le mécanisme de segmentation sans le mécanisme de pagination, plus délicat à mettre en oeuvre.

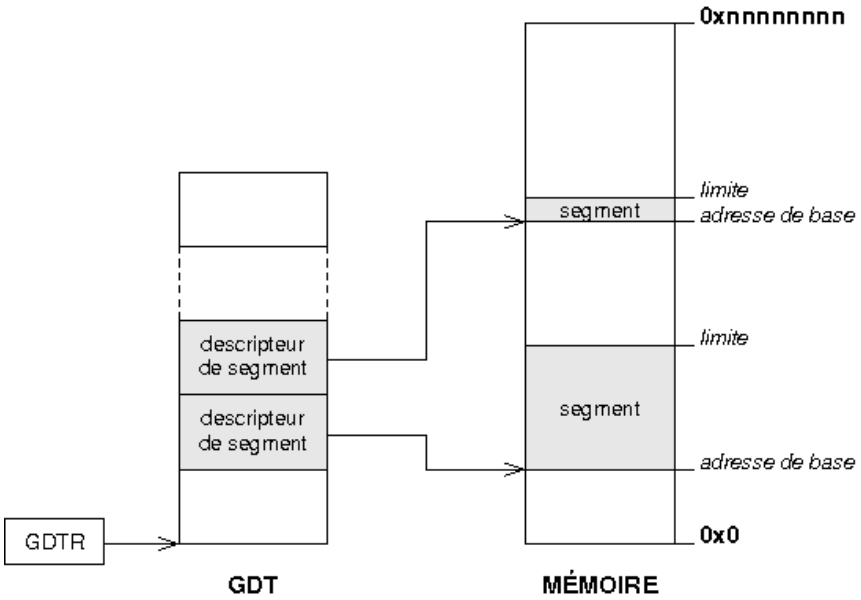
Le mécanisme de segmentation

Une adresse logique est constituée par un sélecteur de segment et un offset. Le sélecteur sélectionne un bloc mémoire d'une certaine taille, appelé **segment**, qui définit en quelque sorte l'espace de travail du programme. L'offset est un déplacement par rapport au début de ce bloc.

Un segment est décrit par une structure de 64 bits appelée **descripteur de segment** qui précise :

- sa **base**, l'endroit en mémoire où commence le segment (sur 32 bits)
- sa **limite**, la taille du segment exprimée en octets ou en blocs de 4 ko
- son type (code, données, pile ou autre)

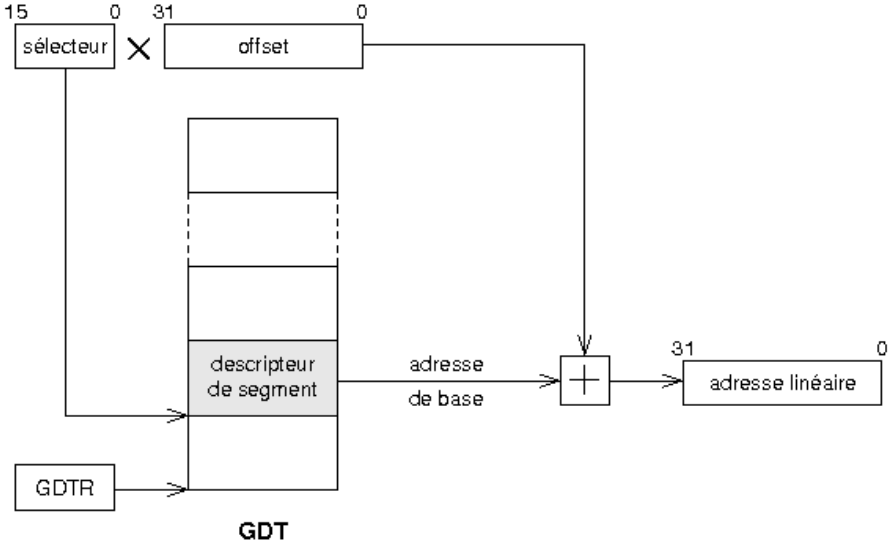
Les descripteurs sont stockés dans la **Global Descriptor Table (GDT)**. Cette table peut résider n'importe où en mémoire. Son adresse en mémoire physique est renseignée au processeur grâce à un registre particulier : le **GDTR** :



Le **sélecteur de segment** est un registre de 16 bits directement manipulé par le programmeur qui pointe sur un descripteur de segment dans la GDT et indique de ce fait dans quel segment on se situe. Ces registres sont bien connus de ceux qui ont déjà programmé en mode réel :

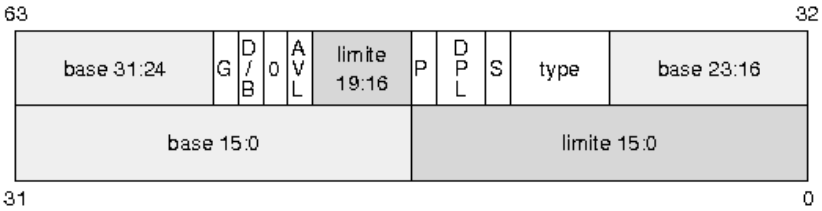
- `cs` est le sélecteur de segment de code
- `ds` est le sélecteur de segment de données
- `es`, `fs` et `gs` sont des sélecteurs de segments généraux
- `ss` est le sélecteur de segment de pile

Le sélecteur pointe sur un descripteur qui donne l'adresse où commence le segment. En ajoutant l'offset à cette base, on obtient une adresse linéaire sur 32 bits :



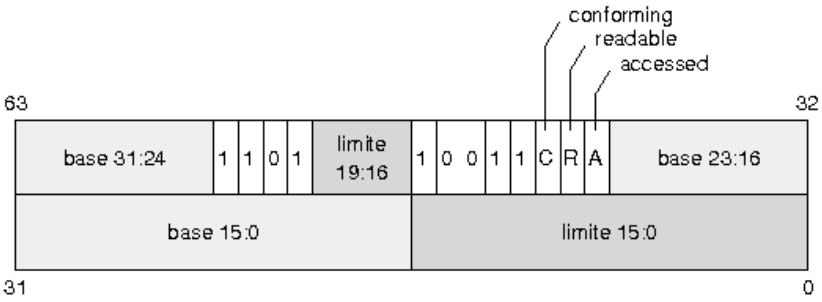
Les descripteurs de segment en détail

Le schéma ci-dessous décrit la structure générale d'un descripteur de segment :



- la **base**, sur 32 bits, est l'adresse linéaire ou débute le segment en mémoire.
- la **limite**, sur 20 bits, définit la longueur du segment.
- si le **bit G** est à 0, la limite est exprimée en octets, sinon, elle est exprimée en nombre de pages de 4 ko.
- le **bit D/B** précise la taille des instructions et des données manipulées. Il est mis à 1 pour 32 bits.
- le **bit AVL** est librement disponible.
- le **bit P** est utilisé pour déterminer si le segment est présent en mémoire physique. Il est à 1 si c'est le cas.
- le **DPL** indique le niveau de privilège du segment. Le niveau 0 correspond au mode super-utilisateur.
- le **bit S** est mis a 1 pour un descripteur de segment et a 0 pour un descripteur système (un genre particulier de descripteur que nous verrons plus tard).
- le **type** définit le type de segment (code, données ou pile).

Descripteur d'un segment de code

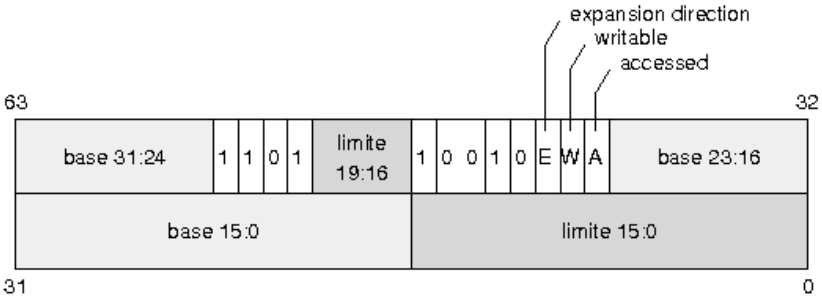


- le bit G est mis à 1 (limite exprimee en pages)
- le bit D/B est mis à 1 (code sur 32 bits)
- le bit P est mis à 1 (page présente en mémoire)
- le niveau de privilege est mis à 0 (mode super-utilisateur)
- le bit S est mis à 1 (descripteur de segment)
- le premier bit à 1 pour le type indique que l'on a affaire à un segment de code
- pour pouvoir adresser toute la mémoire, la base du segment doit être à `0x0` et sa limite doit être à `0xFFFFF` avec le bit de granularité à 1.
- Le bit C indique si le segment de code est "conformant" ou non. Pour le moment, ce bit sera mis à 0.

Ce flag est complexe à manipuler. Il est en lien avec les différents niveaux de privilèges et de protection mis en oeuvre par les microprocesseurs de type i386. La plupart des segments de code sont *non conformant*, ce qui signifie qu'ils peuvent transférer le contrôle (via un `call` ou un `jmp`) seulement à des segments de même privilège. La gestion des niveaux de protection sur architecture i386 est rendue très complexe par un foisonnement de mécanismes impossible à résumer ici. Pour une étude approfondie, l'étude de la documentation de référence est indispensable...

- Le bit R est mis à 1 pour indiquer que le segment de code est accessible en lecture (en plus de l'être en exécution).
- Le bit A est mis à 1 par le processeur quand le segment est utilisé.

Descripteur d'un segment de données



- Le segment de données se distingue du segment de code par le champ *type* avec le premier bit qui est mis à 0.
- Le bit E indique le sens d'expansion des données. Il est mis à 1 pour un segment de type "pile" dans lequel les données s'accumulent vers le début de la mémoire (*expand-down segment*).

  Selon que le bit E est à 0 ou à 1, la *limite* s'interprète différement. Pour un segment sur 32 bits, si le bit E est à 1, la limite supérieure de la plage de données est en `0xFFFFFFFF` et la limite inférieure est à l'adresse indiquée par le champ *limite*. *Note : à confirmer, mais il semble que la base soit dans ce cas purement ignorée*.
- Le bit W est mis à 1 pour indiquer que le segment est accessible en écriture (en plus de l'être en lecture).
- Le bit A est mis à 1 par le processeur quand le segment est utilisé.

Un boot loader qui passe en mode protégé

Quand passer en mode protégé ?

Il est possible de passer en mode protégé à plusieurs moments : lors de l'exécution du secteur de boot ou du noyau. Notre noyau va utiliser un jeu d'instructions sur 32 bits, seulement utilisable en mode protégé. Le plus simple est donc de passer en mode protégé pendant le boot, avant que le noyau ne s'exécute. Il est cependant possible que ce soit le noyau qui effectue la commutation mais celà complique inutilement son écriture car le code du noyau doit alors être en partie sur 16 bits et en partie sur 32 bits.

Le programme du boot loader

bootsect.asm <- cliquer pour afficher le code

```
%define BASE    0x100  ; 0x0100:0x0 = 0x1000
%define KSIZE   50     ; nombre de secteurs a charger

[BITS 16]
[ORG 0x0]

jmp start
%include "UTIL.INC"
start:

; initialisation des segments en 0x07C0
    mov ax, 0x07C0
    mov ds, ax
    mov es, ax
    mov ax, 0x8000    ; stack en 0xFFFF
    mov ss, ax
    mov sp, 0xf000

; recuparation de l'unite de boot
    mov [bootdrv], dl

; affiche un msg
    mov si, msgDebut
    call afficher

; charger le noyau
    xor ax, ax
    int 0x13

    push es
    mov ax, BASE
```

```
    mov es, ax
    mov bx, 0
    mov ah, 2
    mov al, KSIZE
    mov ch, 0
    mov cl, 2
    mov dh, 0
    mov dl, [bootdrv]
    int 0x13
    pop es

; initialisation du pointeur sur la GDT
    mov ax, gdtend    ; calcule la limite de GDT
    mov bx, gdt
    sub ax, bx
    mov word [gdtptr], ax

    xor eax, eax      ; calcule l'adresse lineaire de GDT
    xor ebx, ebx
    mov ax, ds
    mov ecx, eax
    shl ecx, 4
    mov bx, gdt
    add ecx, ebx
    mov dword [gdtptr+2], ecx

; passage en modep
    cli
    lgdt [gdtptr]    ; charge la gdt
    mov eax, cr0
    or  ax, 1
    mov cr0, eax        ; PE mis a 1 (CR0)

    jmp next
next:
    mov ax, 0x10        ; segment de donne
    mov ds, ax
    mov fs, ax
    mov gs, ax
    mov es, ax
    mov ss, ax
    mov esp, 0x9F000

    jmp dword 0x8:0x1000    ; reinitialise le segment de code

;-----------------------------------------------------------------------
bootdrv:  db 0
msgDebut: db "Chargement du kernel", 13, 10, 0
;-----------------------------------------------------------------------
gdt:
    db 0, 0, 0, 0, 0, 0, 0, 0
gdt_cs:
    db 0xFF, 0xFF, 0x0, 0x0, 0x0, 10011011b, 11011111b, 0x0
gdt_ds:
    db 0xFF, 0xFF, 0x0, 0x0, 0x0, 10010011b, 11011111b, 0x0
gdtend:
;-----------------------------------------------------------------------
gdtptr:
    dw 0  ; limite
    dd 0  ; base
;-----------------------------------------------------------------------

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

[Get Code]

Que fait exactement ce programme ?

Ce programme est identique à celui du chapitre précédent avec en plus des instructions qui basculent le microprocesseur en mode protégé. Pour résumer, le numéro de périphérique de boot est placé dans une variable, les registres relatifs aux segments de code et de données sont initialisés, puis le noyau est chargé en mémoire à l'adresse `0x1000` (on note que la variable `KSIZE` a été augmentée afin de charger un noyau plus volumineux). Ensuite, la GDT est initialisée et chargée en mémoire, puis on bascule en mode protégé. Enfin, le noyau est exécuté.

> Ce secteur de boot peut charger seulement des noyaux d'une taille limitée et bien inférieure à la capacité maximale d'une disquette. Nous verrons plus tard comment charger notre noyau à l'aide de GRUB.

Passer en mode protégé

Avant de passer en mode protégé, il faut initialiser la GDT de façon à ce qu'il n'y ait pas de problème d'adressage après le changement de mode. La GDT doit contenir des descripteurs pour les segments de code, de données et de pile. Les directives ci-dessous déclarent et initialisent la GDT :

```
gdt:
    db 0, 0, 0, 0, 0, 0, 0, 0
gdt_cs:
    db 0xFF, 0xFF, 0x0, 0x0, 0x0, 10011011b, 11011111b, 0x0
gdt_ds:
    db 0xFF, 0xFF, 0x0, 0x0, 0x0, 10010011b, 11011111b, 0x0
gdtend:
```

L'étiquette `gdt:` est un pointeur sur le début du tableau qui contient trois descripteurs :

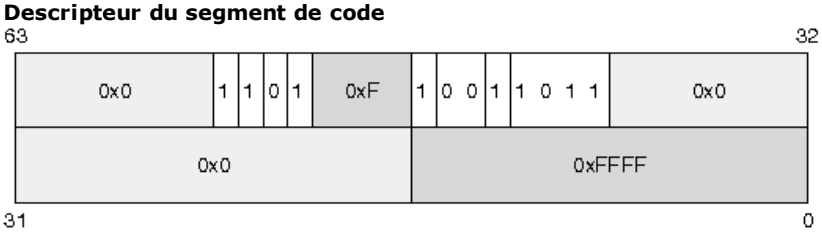
- le premier descripteur ne doit pas être utilisé et les données sont mises à zéro. Il s'agit du descripteur NULL.

- le deuxieme descripteur, avec l'étiquette `gdt_cs:`, décrit le segment de code.
- le troisième descripteur, avec l'étiquette `gdt_ds:`, décrit le segment de données.

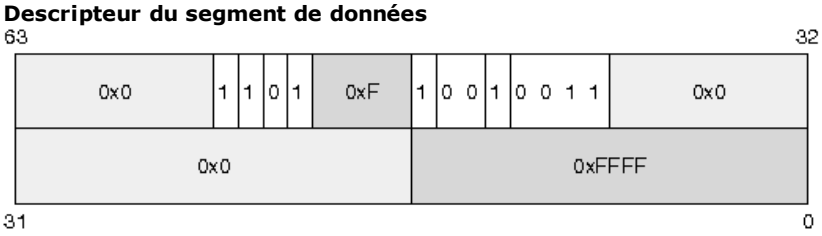
Chaque descripteur est initialié de façon à pouvoir adresser l'ensemble de la RAM. La base de ces segments est à `0x0` avec une limite de `0xFFFFF` pages (le bit G est à 1).

Les schémas ci-dessous résument la façon dont sont initialisés les descripteurs :

Descripteur du segment de code



Descripteur du segment de données



La GDT est directement initialisée, mais avant de basculer en mode protégé, il faut renseigner le processeur pour qu'il prenne en compte la GDT. Celà se fait en mettant à jour le registre **GDTR**, de 6 octets, qui contient l'adresse de la GDT et sa taille (on parle aussi de limite). On charge ce registre spécial avec l'instruction `lgdt`.

Dans le programme, `gdtptr` est un pointeur sur une structure qui contient les informations à charger dans le registre GDTR. La structure `gdtptr` est d'abord declarée et initialisée à zéro (comme une variable classique en C) :

```
gdtptr:
    dw 0  ; limite
    dd 0  ; base
```

Ensuite, on calcule les valeurs pour mettre dans cette structure. Le code suivant calcule la taille de la GDT et stocke la valeur dans le premier champs de `gdptr` :

```
; initialisation du pointeur sur la GDT
    mov ax, gdtend    ; calcule la limite de GDT
    mov bx, gdt
    sub ax, bx
    mov word [gdtptr], ax
```

Ce code calcule l'adresse physique de la GDT en se basant sur les valeur du segment de données `ds` et de l'adresse de l'étiquette `gdt`. Le résultat de ce calcul est stocké dans le second champ de `gdptr` :

```
; calcule l'adresse lineaire de GDT
    xor eax, eax
    xor ebx, ebx
    mov ax, ds
    mov ecx, eax
    shl ecx, 4
    mov bx, gdt
    add ecx, ebx
    mov dword [gdtptr+2], ecx
```

Notre structure est donc maintenant correctement initialisée. Nous sommes maintenant presque prêt à passer en mode protégé. Avant celà, il faut inhiber les interruptions car comme le système d'adressage va changer, les routines appelées par les interruptions ne seront plus valides après la bascule (il faudra les reprogrammer) :

```
; passage en modep
    cli
```

Le registre GDTR est chargé avec l'instruction `lgdt` pour indiquer au microprocesseur où se trouve la GDT :

```
; charge la gdt
    lgdt [gdtptr]
```

On peut maintenant passer en mode protégé :

```
; PE mis a 1 (CR0)
    mov eax, cr0
    or  ax, 1
    mov cr0, eax
```

Enfin ! :-)

Notre tâche semble terminée. Mais au fait... il reste encore à réinitialiser les sélecteurs de segment de code et de données ! La commande qui suit doit impérativement être la suivante afin de vider les caches internes du processeur :

```
jmp next
next:
```

> En principe, il faudrait faire un *far jump* à la place du *near jump* ci-dessus pour réinitialiser le sélecteur de segment de code. Oui mais voilà, le manuel spécifie : *"When the processor is switched into protected mode, the original code segment base-address value of FFFF0000H (located in the hidden part of the CS register) is retained and execution continues from the current offset in the EIP register. The processor will thus continue to execute code in the EPROM until a far jump or call is made to a new code segment, at which time, the base address in the CS register will be changed."*

Ensuite, on réinitialise les sélecteurs de données :

```
; segment de donnees
    mov ax, 0x10
    mov ds, ax
    mov fs, ax
    mov gs, ax
    mov es, ax
```

Puis le segment de pile :

```
; la pile
    mov ss, ax
    mov esp, 0x9F000
```

L'instruction suivante réinitialise le sélecteur de code et exécute le noyau situé à l'adresse physique `0x1000`. Cette instruction est essentielle car elle permet, outre l'exécution du code du noyau, la réinitialisation correcte du sélecteur de code sur le bon descripteur (offset `0x8` dans la GDT) :

```
; reinitialise le segment de code
    jmp dword 0x8:0x1000
```

Ensuite, le code du noyau s'exécute...

Un noyau très simple

Ce noyau affiche juste un message de bienvenue et boucle ensuite indéfiniment. A ce stade, les routines du BIOS permettant d'afficher des caractères à l'ecran ne sont plus utilisables, il faut donc que nous gérions nous même l'affichage.

Le code du noyau

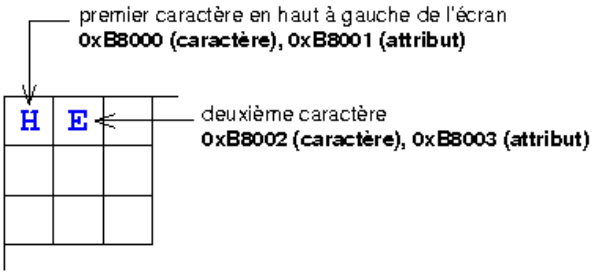
```
[BITS 32]
[ORG 0x1000]

; Affichage d'un message par ecriture dans la RAM video
    mov byte [0xB8A00], 'H'
    mov byte [0xB8A01], 0x57
    mov byte [0xB8A02], 'E'
    mov byte [0xB8A03], 0x0A
    mov byte [0xB8A04], 'L'
    mov byte [0xB8A05], 0x4E
    mov byte [0xB8A06], 'L'
    mov byte [0xB8A07], 0x62
    mov byte [0xB8A08], 'O'
    mov byte [0xB8A09], 0x0E

end:
    jmp end
```

[Get Code]

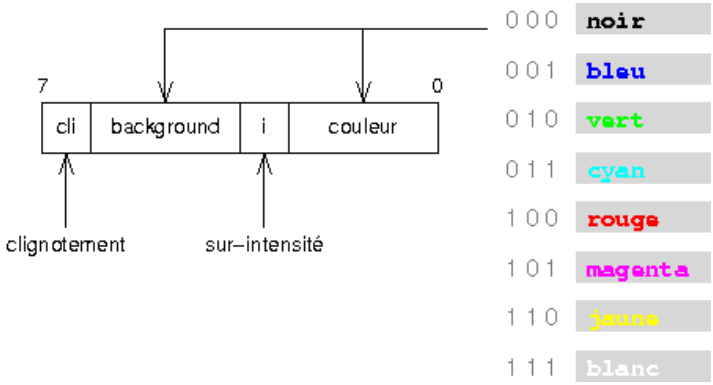
Afficher quelque chose à l'écran

La mémoire video est mappée en mémoire à l'adresse physique `0xB8000`. On peut donc afficher des informations en manipulant directement les octets débutant à cette adresse :

- La console d'affichage comprend 25 lignes et 80 colonnes.
- Chaque caractère est décrit par 2 octets. Le premier contient le code ascii du caractère à afficher et le suivant contient ses attributs (couleur, clignotement...)



Les attributs sont codés de la façon suivante :



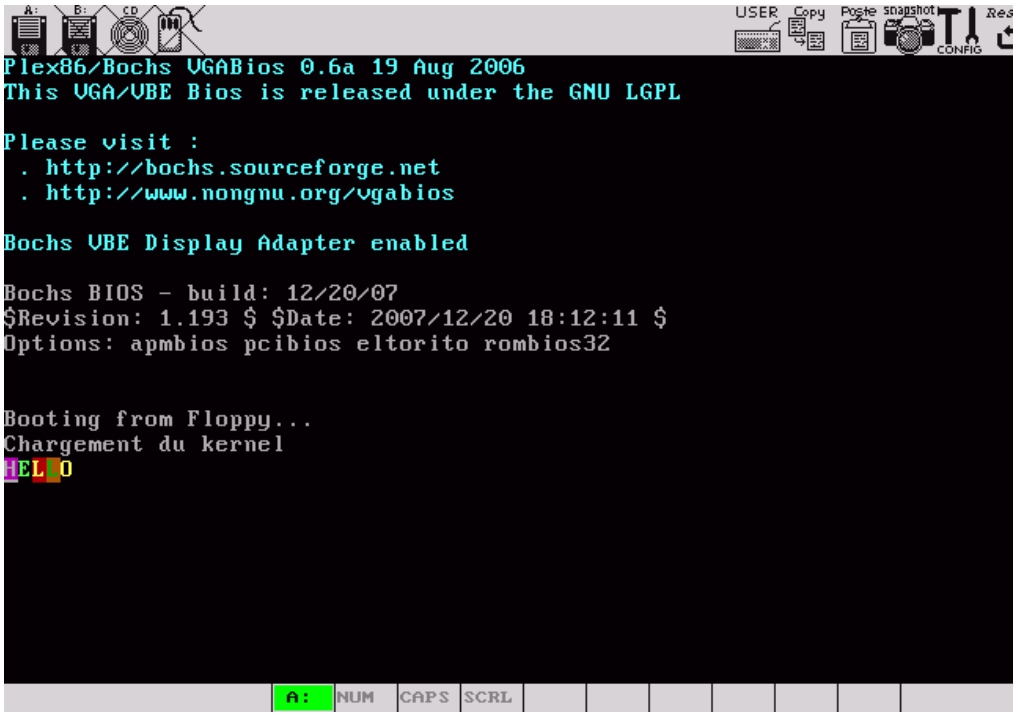
Par exemple, le code suivant affiche le caractère 'H' en blanc sur fond magenta en haut à gauche de l'écran :

```
mov byte [0xB8000], 'H'
mov byte [0xB8001], 0x57
```

Compiler et tester le boot loader et le noyau

On compile le boot loader et le noyau séparement puis on crée la disquette :

```
$ nasm -f bin -o bootsect bootsect.asm
$ nasm -f bin -o kernel kernel.asm
$ cat bootsect kernel /dev/zero | dd of=floppyA bs=512 count=2880
```



< Réaliser un secteur de boot qui charge et exécute un noyau | TutoOS | Écrire un noyau en C >