Troisième partie

Programmation système II : le mode protégé

Chapitre 8

Interprétation des adresses en mode protégé

Nous avons vu, dans le chapitre précédent, qu'avec l'architecture IA-32 sont introduites des instructions permettant, en particulier, d'utiliser des registres 32 bits et de manipuler des constantes de 32 bits. Cependant l'espace mémoire accessible reste limité à 1 MiO. La première raison de passer au *mode protégé* (une autre raison est évidemment celle qui a donné son nom à ce mode) est de pouvoir accéder aux 4 GiO de mémoire que permettent les adresses sur 32 bits, ou tout au moins à tous les emplacements mémoire dont dispose l'ordinateur sur lequel on se trouve.

8.1 Le mode protégé

Le microprocesseur `80286` d'*Intel* introduit, en 1982, une nouveauté importante : il est non seulement plus rapide et possède quelques instructions supplémentaires par rapport au `8086` et au `80186`, mais il possède surtout deux **modes** de fonctionnement. Le premier mode, appelé **mode réel**, est compatible avec le `8086`, en plus rapide. Le **mode protégé** permet de simplifier la conception des systèmes d'exploitation multi-tâches et multi-utilisateurs. Il faut cependant attendre plusieurs années avant que cette caractéristique soit utilisée, avec les systèmes d'exploitation OS/2 (introduit en décembre 1987), *Windows/NT* et *Linux* (1993).

8.2 Accès à la mémoire en mode protégé

8.2.1 Adresse physique et segmentation

Puisqu'on peut accéder à beaucoup de mémoire (à condition d'avoir suffisamment de barettes de mémoire, évidemment), il n'est peut-être pas utile de pouvoir accéder à l'intégralité de la mémoire à tout moment. Ceci conduit à ne pas utiliser l'adresse physique de façon abrupte.

Adresse physique.- D'un point de vue matériel, les cellules mémoires sont adressées, pour l'architecture IA-32, grâce à 32 broches. On parle d'**espace physique** : l'**adresse physique** est le mot binaire transmis aux broches.

Segmentation.- 1^o) Cependant, comme nous l'avons expliqué, le programmeur d'applications ne doit pas utiliser l'adresse physique. Il n'utilise pas l'espace physique adressable dans son intégralité, mais seulement une portion de celui-ci, appelée **segment** (sous-entendu *de mémoire*).

Un segment peut être constitué de l'intégralité de la mémoire physique si on y tient vraiment, mais c'est rarement le cas.

- 2^o) Un segment est constitué d'une portion connexe de l'espace physique. Son emplacement est donc entièrement caractérisé par son **adresse de base**, c'est-à-dire l'adresse physique de son premier élément mémoire, et sa **taille**, c'est-à-dire le nombre d'octets qui le constitue.

- 3^o) On repère une cellule mémoire dans un segment grâce à un index, appelé techniquement **décalage** (*offset* en anglais).

Le décalage varie de 0 à taille moins un, cette dernière quantité étant appelée **limite**, d'où le nom de décalage : l'adresse physique est la somme de l'adresse de base et du décalage.

- 4^o) En mode protégé, une **adresse** est donc constituée d'un **sélecteur** (*selector* en anglais), spécifiant le segment, sur la structure duquel nous reviendrons, et d'un décalage.

- 5^o) À chaque tentative d'accès à la mémoire vive, le microprocesseur vérifie que le décalage ne conduit pas en dehors du segment : le programme est interrompu et une **faute**, dite **de protection générale**, est levée si c'est le cas, plus précisément il appelle l'interruption 13 (dont la routine de service est conçue par le concepteur du système d'exploitation).

- 6^o) Le décalage, de 32 bits, spécifié par l'un quelconque des registres généraux étendus (`eax`, `ebx`, `ecx`, `edx`, `ebp`, `edi` et `esi`), permet d'accéder à des données dans un segment de taille pouvant aller jusqu'à 4 GiO.

8.2.2 Modes d'adressage

Nous avons vu divers types d'adressage, c'est-à-dire la façon de spécifier le décalage, à propos du mode réel. Les types d'adressage vus à propos du mode réel sont encore valables mais il en existe quelques autres :

— Dans l'**adressage indirect par registre**, le décalage est le contenu d'un registre, par exemple :

```
MOV [ECX],EDX
```

en notation *Intel*, où le décalage est le contenu du registre `ECX` et la taille est spécifiée par le second opérande (le registre `EDX`).

En notation AT&T, on écrit :

```
MOVL %EDX,(%ECX)
```

Dans le cas d'un adressage sur 32 bits, le décalage doit pouvoir être également de 32 bits. Les registres permis (on parle de **pointeur** pour leur contenu) sont les huit registres généraux `EAX`, `EBX`, `ECX`, `EDX`, `ESP`, `EBP`, `ESI` et `EDI`.

— Dans l'**adressage par index**, on utilise les huit registres généraux sauf `ESP` dans le cas d'un adressage sur 32 bits, par exemple en notation *Intel* :

```
MOV ECX,24[ESI]
```

— On trouve un nouveau mode d'adressage, valable uniquement en mode 32 bits. Dans l'**adressage par index et échelle**, l'index est multiplié par l'**échelle** (une constante égale à 1, 2, 4 ou 8, *scale* en anglais) et le tout est ajouté à un déplacement :

```
MOV ECX,24[ESI*4]
```

— Dans l'**adressage par base, index et échelle**, le décalage est obtenu comme somme des contenus d'un registre de base et d'un registre d'index, ce dernier contenu étant multiplié par l'échelle :

```
MOV ECX,[ESI*8][EBX]
```

— Dans l'**adressage par base, index et déplacement**, le décalage est obtenu comme somme du contenu d'un registre de base, du contenu d'un registre d'index et d'un déplacement :

```
MOV ECX,[ESI][EBX+80]
```

— Le cas général est celui de l'**adressage par base, index, échelle et déplacement**, où le décalage est obtenu comme somme du contenu d'un registre de base, du contenu d'un registre d'index, celui-ci étant multiplié par l'échelle, et d'un déplacement :

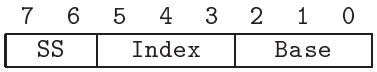
```
MOV ECX,24[ESI*4][EBX+80]
```

Pour résumer, un décalage est obtenu de la façon suivante :

$$\begin{bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ ESP \\ EBP \\ ESI \\ EDI \end{bmatrix} + \begin{bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ \\ EBP \\ ESI \\ EDI \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \text{déplacement (0, 8 ou 32 bits)}$$

8.2.3 Codage des instructions

Une nouveauté de l'architecture IA-32 est que l'octet `mod r/m` peut appeler un second octet d'adressage, appelé **SIB** (pour *Scale Index Base*, soit 'base et index avec échelle') :



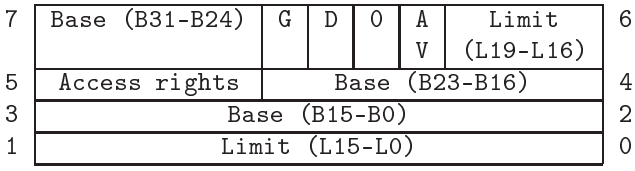
où le champ SS d'échelle signifie :

SS	Multiplication par :
00	0
01	2
10	4
11	8

8.2.4 Description d'un segment

Dans le contexte de la protection des données (origine du mode protégé que nous étudierons plus tard), un segment n'est pas seulement caractérisé par son emplacement mais aussi par des **droits d'accès**. Ces caractéristiques sont placées dans un **descripteur** (*descriptor* en anglais) de segment.

Le format d'un descripteur pour l'architecture IA-32 (celui-ci est un peu différent pour le `80286`), d'une de taille 8 octets, est le suivant :



— L'**adresse de base** est la partie du descripteur spécifiant l'adresse physique de début du segment. Cette adresse occupe 32 bits : le segment peut débuter à n'importe laquelle des adresses physiques des 4 GiO de l'espace physique.

— La **limite du segment** contient le dernier décalage du segment.

Par exemple, si un segment commence à l'adresse `0xF00000` et se termine à l'adresse `0xF000FF`, l'adresse de base est `0x00F00000` et la limite du segment est `0x000FF`.

La limite a une taille de 20 octets : la taille d'un segment est comprise entre 1 octet et un MiO, par pas de un octet, ou entre 4 KiO et 4 GiO, par pas de 4 KiO, suivant la **granularité**.

— Le **bit de granularité** G (pour *Granularity*) vaut 0 si la limite spécifie un segment dont la limite est comprise entre `00000h` et `FFFFFh` et 1 si la valeur de cette limite doit être multipliée par 4 Ki.

— Le bit AV (pour l'anglais *AVailable*) est mis à disposition du système d'exploitation pour indiquer *a priori* que le segment est disponible (AV = 1) ou non (AV = 0).

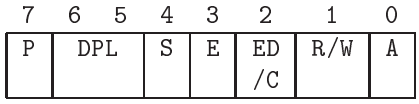
Le système d'exploitation l'utilise à sa guise.

— Le bit D (pour *Default mode*) indique si les instructions ont accès par défaut aux registres et à la mémoire en 32 bits ou en 16 bits.

Si D = 0, on accède, comme en mode réel, par défaut aux registres et aux décalages de 16 bits. On utilise les préfixes `0x66` et `0x67` pour accéder aux registres et aux décalages 32 bits, comme nous l'avons vu. On parle du **mode d'instructions 16 bits**.

Si D = 1, on accède par défaut aux registres et aux décalages de 32 bits. On utilise les préfixes `0x66` et `0x67` pour accéder aux registres et aux décalages 16 bits. On parle du **mode d'instructions 32 bits**.

— L'**octet des droits d'accès** contrôle l'accès au segment de mémoire. Les champs sont indiqués sur la figure ci-dessous :



— Le bit 0, noté A (pour *Accessed*), indique si on a eu accès au segment (A = 1) ou non (A = 0).

Il est mis à disposition du système d'exploitation pour savoir si on doit le recopier sur le disque avant de le remplacer en mémoire vive par un autre segment.

— Les bits 1 (noté R/W pour *Read/Write*), 2 (noté ED/C pour *Enable Decrementation / Controlled*) et 3 (noté E) spécifient la nature du segment.

— Si E = 0, il s'agit d'un segment de données :

— Si ED = 0, les adresses du segment doivent être incrémentées (cas typique d'un segment de données). Si ED = 1, elles doivent être décrémentées (cas typique d'un segment de pile).

— Si W = 0, on ne peut pas écrire (autrement dit surcharger les données) alors que si W = 1, on le peut.

— Si E = 1, il s'agit d'un segment de code :

— Si C = 0, on ignore le niveau de privilège du descripteur alors que si C = 1, on en tient compte.

— Si R = 0 on ne peut pas lire le segment de code alors qu'on le peut si R = 1.

— Le bit 4, noté S (pour *System*), permet de savoir s'il s'agit d'un descripteur système (S = 0), sur lequel nous reviendrons plus tard, ou du descripteur d'un segment de données ou de code (S = 1), dont le rôle est clair dès à présent.

— Les bits 5 et 6, notés DPL (pour *Descriptor Privilege Level*), spécifient le niveau de privilège du descripteur, compris entre 0 et 3, sur lequel nous reviendrons dans le chapitre consacré aux niveaux de privilège.

— Le bit 7, noté P (pour *Present*), indique que le segment n'est pas actuellement présent (P = 0) en mémoire vive ou qu'il l'est et contient donc une base et une limite valides (P = 1).

Deux tables de descripteurs.- Tout descripteur est placé dans une **table de descripteurs**. La **table globale des descripteurs** GDT (pour *Global Descriptor Table*) contient la description des segments s'appliquant à tous les programmes alors que les **tables locales des descripteurs** LDT (pour *Local Descriptor Table*), dont une seule est utilisable à un moment donné, contiennent les descriptions des segments utilisés par une tâche donnée.

Chaque table de descripteurs peut contenir jusqu'à 8 192 descripteurs. Il y a donc 16 384 segments de mémoire disponibles pour chaque application (tâche). Un descripteur ayant une taille de 8 octets, une table de descripteurs occupe au plus 64 KiO.

Le descripteur de numéro 0, appelé **descripteur nul**, ne peut pas être utilisé pour accéder à la mémoire.

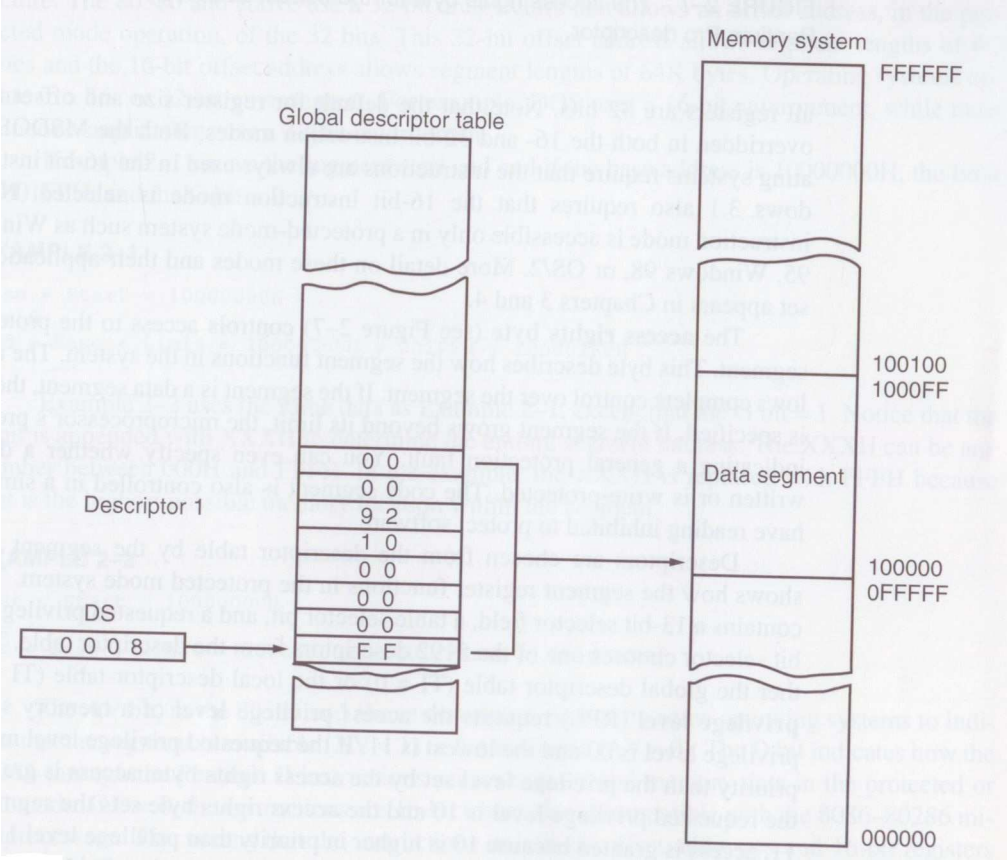
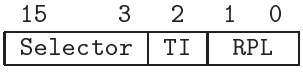


FIGURE 8.1 – Utilisation de DS pour sélectionner un descripteur de la GDT

Accès aux descripteurs.- Les descripteurs sont choisis dans une des deux tables de descripteurs disponibles à l'aide du contenu de l'un des registres de segment (`cs` à `gs`). La figure ci-dessous montre la structure d'un tel registre en mode protégé, dont la signification est donc différente de celle qu'elle a en mode réel :



— Le champ **sélecteur** (bits 3 à 15, *selector* en anglais) permet de choisir l'un des 8 192 descripteurs de la table concernée.

— Le bit TI de table de sélecteur (bit 2) permet de spécifier la table : la table globale des descripteurs si TI = 0, la table locale des descripteurs si TI = 1.

— Le champ RPL (pour *Resquested Privilege Level*, bits 0 et 1) spécifie le niveau de privilège requis pour accéder à ce descripteur. Si RPL est supérieur ou égal au niveau de privilège DPL du descripteur, l'accès sera accordé ; sinon, le système indique un essai de violation de privilège.

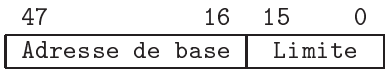
Remarquons que, pour TI = 0 et RPL = 0, le contenu d'un registre de segment correspond au décalage dans la table de descripteurs du descripteur voulu, puisque la taille d'un descripteurs est de huit octets.

Les registres de segment ne sont en général manipulables qu'au niveau de privilège 0 (c'est à ce niveau qu'on le décide, en tous les cas).

Exemple.- La figure 8.1 montre comment le registre de segment `DS`, dont la valeur est `0x0008`, choisit une zone de mémoire. On a `RPL = 0`, donc tous les droits. On a `TI = 0`, donc on sélectionne un descripteur de la table GDT globale des descripteurs. Le champ sélecteur est égal à 1, on choisit donc le premier descripteur de la table. Admettons que ce descripteur ait la valeur `0x00 00 92 10 00 00 00 FF`, comme indiqué sur le figure. L'adresse de base est alors `0x00 10 00 00` et la limite `0x0 00 FF`. Ainsi le microprocesseur utilisera-t-il les emplacements mémoire `0x00100000 - 0x001000FF`.

8.2.5 Registres spéciaux

Registre de la GDT.- Le **GDTR** (*Global Descriptor Table Register*) est un registre de 48 bits contenant l'adresse de base, bits 16 à 47, de la table globale des descripteurs et sa limite, bits 0 à 15. La limite de cette table tient sur 16 bits puisque la longueur maximum de la table est de 64 KiO.



On doit initialiser la table globale des descripteurs avant de passer au mode protégé : on doit charger son adresse et sa limite dans le registre GDTR, en mode réel donc, grâce à une instruction que nous verrons ci-dessous.

Registre de la LDT.- La table locale des descripteurs est spécifiée par l'un des segments de la table globale des descripteurs. Pour avoir accès à cette table locale des descripteurs, on place un sélecteur dans le registre **LDTR** (*Local Descriptor Table Register*), en mode protégé de niveau 0, grâce à une instruction que nous verrons plus loin.

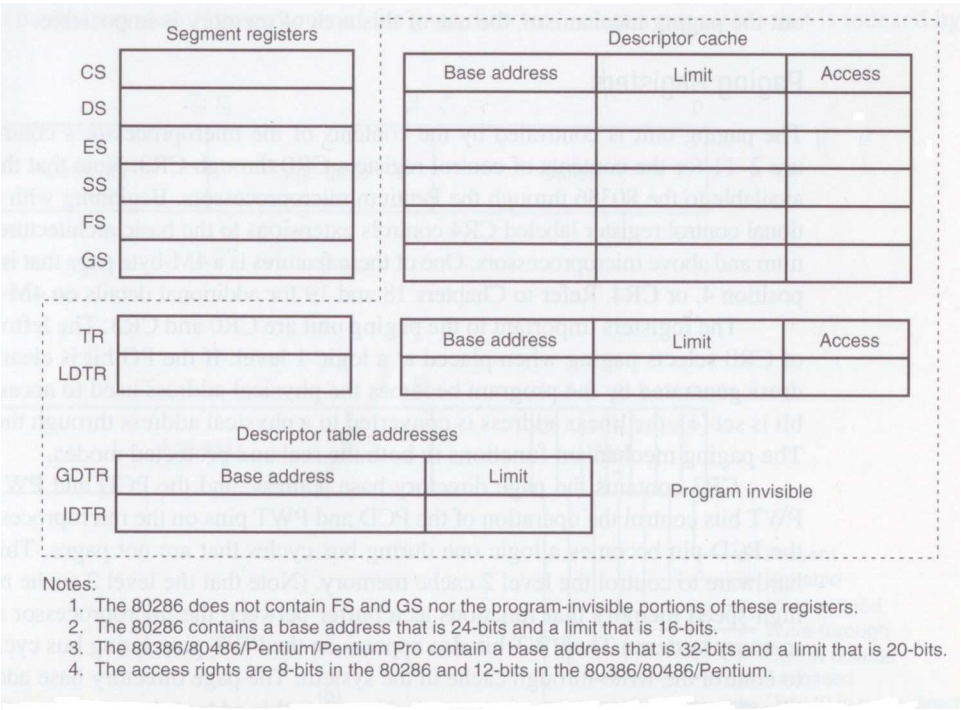


FIGURE 8.2 – Les registres invisibles au programmeur

Registres invisibles au programmeur.- Les tables de descripteurs, locale et globale, sont situées en mémoire vive. Pour accéder et spécifier les adresses de ces tables, le microprocesseur contient des **registres invisibles** : ils ne sont pas directement accessibles par des instructions, d'où leur nom. La figure 8.2 montre les registres invisibles contrôlant le microprocesseur lorsqu'il opère en mode protégé.

— Chaque registre de segment contient une partie invisible, utilisée en mode protégé.

La partie invisible de ces registres est quelquefois appelée **mémoire cache**, puisqu'il s'agit d'un cache stockant de l'information. Ce type de mémoire cache ne doit cependant pas être confondue avec les mémoires caches de niveau 1 et 2 trouvées sur le microprocesseur (et non en mémoire centrale).

La partie invisible d'un registre de segment contient l'adresse de base, la limite et les droits d'accès de celui-ci. Elle est modifiée chaque fois que la valeur du registre de segment l'est : lorsqu'un nouveau numéro de segment est placé dans un registre de segment, le microprocesseur accède à la table adéquate de descripteurs et charge le descripteur dans la partie cache du registre de segment. Ce descripteur est utilisé pour l'accès au segment de mémoire jusqu'au prochain changement de numéro de segment. Ceci permet au microprocesseur d'avoir accès au segment de mémoire sans recourir à la table adéquate de descripteurs à chaque accès.

— Le LDTR accède à la GDT et charge l'adresse de base, la limite et les droits d'accès de la LDT dans la partie cache du LDTR.

Registre de contrôle CR0.- Quatre registres de contrôle, notés CR0 à CR3, existent sur l'architecture IA-32, chacun d'une longueur de 32 bits, dont seul le premier nous intéresse pour l'instant.

La figure ci-dessous donne la structure de ce registre CR0, dont seul le bit 0 nous intéresse pour l'instant : noté **PE** (pour *Protected Enabled*), il sert à choisir le mode protégé lorsqu'il est positionné à 1 ; il sert également à revenir au mode réel en le positionnant à 0.

		MSW					
P G	0000000000000000	00000000000	E T	T S	E M	M P	P E

Les 16 premier bits du registre de contrôle CR0 étaient déjà présents sur le 80286, sous le nom de **mot de statut de la machine** (MSW pour *Machine Status Word*).

8.2.6 Instructions de contrôle du système

Description.- Appelons **instructions de contrôle du système** les instructions de l'architecture IA-32 (la plupart déjà présentes sur le 80286) utilisables au mode réel pour passer en mode protégé ; elles sont également utilisables en mode protégé, mais au niveau de privilège 0 uniquement, en particulier pour revenir au mode réel :

- 1^o) (**Chargement de la GDT**) On charge la table globale des descripteurs grâce à l'instruction :

```
LGDT S
```

(pour *Load the Global Descriptor Table Register*), où `S` spécifie l'adresse physique de l'emplacement mémoire contenant le premier octet des 6 octets à placer dans le `GDTR`.

- 2^o) (**Manipulation du mot de statut**) Pour le 80286, on écrit le mot de statut de la machine `MSW` grâce à l'instruction :

```
LMSW S
```

(pour *Load the Machine Status Word*), où `S` est le mot que l'on veut écrire. On lit ce mot grâce à l'instruction :

```
SMSW D
```

(pour *Store the Machine Status Word*), où `D` spécifie la destination du mot.

Pour passer en mode protégé, c'est-à-dire pour mettre le bit 0 (`PE` de ce mot à 1, on utilise en général la suite d'instructions suivante :

```
SMSW   AX       ; lit le MSW
OR     AX,1     ; met PE a 1
LMSW   AX       ; ecrit le MSW
```

- 3^o) (**Accès au registre CR0**) Pour l'architecture IA-32, on utilise plutôt :

```
MOV    EAX,CR0  ; lit CR0
OR     EAX,1    ; met PE a 1
MOV    CR0,EAX  ; ecrit sur CR0
```

Code opération des nouvelles instructions.- Les codes machine de ces nouvelles instructions sont :

Instruction	Format
MOV	
CR0/CR2/CR3 from register	0000 1111 | 0010 0010 | 00 eee reg
Register From CR0-3	0000 1111 | 0010 0000 | 00 eee reg
LGDT	0000 1111 | 0000 0001 | mod 010 r/m
LMSW	0000 1111 | 0010 0001 | mod 110 reg
SMSW	0000 1111 | 0010 0001 | mod 100 reg

8.3 Passage au mode protégé

8.3.1 Principe du passage en mode protégé

Les étapes.- Les microprocesseurs *Intel* démarrent toujours en mode réel, compatibilité oblige. Le passage au mode protégé s'effectue, comme nous venons de le voir, en mettant à 1 le bit `PE` du registre système `CR0`. Cependant un certain nombre d'initialisations doivent être accomplies autour de ce passage :

— On doit initialiser la table globale des descripteurs, dont on a vu le rôle dans la gestion de la mémoire en mode protégé.

Une des raisons pour lesquelles le microprocesseur commence à opérer en mode réel, outre la compatibilité, est que cette table est bien trop complexe pour être renseignée lors de la réinitialisation du microprocesseur ou pour opérer temporairement avec quelques valeurs par défaut.

La table GDT globale des descripteurs doit contenir un descripteur nul comme descripteur de numéro 0 et des descripteurs valides pour au moins un segment de code et un segment de données.

— On doit charger dans le registre `GDTR` l'adresse physique de base de la GDT ainsi que sa limite.

— On peut seulement alors passer au mode protégé en mettant le bit `PE` de `CR0` à 1 : sans l'initialisation ci-dessus de la GDT, le microprocesseur redémarre.

On peut, pour cela, utiliser à sa guise une instruction `MOV` de transfert vers `CR0` ou l'instruction `LMSW` datant du `80286` : la différence est qu'on ne peut pas, avec cette dernière, mettre simultanément en place la pagination (sur laquelle nous reviendrons dans le chapitre qui lui est consacré).

— On se trouve alors en mode protégé. Il faut effectuer un saut dans le segment (`JMP` proche), par exemple tout simplement à l'instruction suivante, pour vider la file d'attente des instructions internes.

Ceci sert à nous assurer que les instructions suivantes seront bien interprétées comme du code en mode protégé et non en mode réel.

— Les six registres de segment continuent à contenir les valeurs qu'ils avaient en mode réel. Il faut donc charger au plus vite les sélecteurs de données (registres de segment) avec leurs valeurs de sélecteur initiales.

On commence par initialiser `DS`, `ES`, `FS` et `GS`.

Il est plus difficile d'initialiser `CS` : la seule façon de changer la valeur de `CS` en mode réel étant d'effectuer un saut long (inter-segmentaire), il suffit par exemple d'effectuer un `FAR JMP` à l'instruction suivante.

On initialise ensuite les registres `SS` et `ESP` : en effet, si on ne change pas le pointeur de pile `ESP` après le changement de `SS`, le microprocesseur utilise le pointeur de pile `SP` du mode réel. On utilise l'instruction `LSS` pour cela.

— Le microprocesseur opère alors en mode protégé, en utilisant les descripteurs de segment définis dans GDT.

8.3.2 Principe du Retour au mode réel

Si on veut revenir au mode réel sans redémarrer le microprocesseur, on met à zéro le bit `PE` du registre `CR0`, tout au moins à partir du `80386` (cette option n'étant pas prévue pour le `80286`). Cependant, comme dans le cas du passage au mode protégé, ceci nécessite plus qu'une simple instruction de transfert `MOV` :

— Il faut utiliser des descripteurs les plus proches possibles des segments en mode réel. On doit donc charger les sélecteurs avec des descripteurs ayant les attributs suivants pour les quatre registres de segment (`DS`, `ES`, `FS` et `GS`) et le registre de segment de pile (`SS`) :

Attribut	**Valeur**	
Limite	`FFFF`	(exactement 64 KiO)
Granularité	0	(granularité par octet)
Direction	0	(en avant)
Écriture	1	(on peut écrire)
Présent	1	(segment présent)

— On doit également charger `CS` avec un sélecteur vers un descripteur de code ayant ces attributs, sauf en ce qui concerne l'attribut d'écriture. On effectue ceci en effectuant une instruction `FAR JMP`.

— On positionne le bit `PE` de `CR0` à 0 grâce à une instruction `MOV`.

— On exécute une instruction `FAR JMP` pour vider la file d'attente des instructions.

— On se retrouve alors en mode réel.

8.3.3 Exemple

La difficulté de donner un exemple probant de passage en mode protégé est de faire exécuter quelque chose de « visible », une fois que l'on se trouve en mode protégé. Rappelons, en effet, que la lecture au clavier ou l'affichage se fait en général en mode réel grâce à des interruptions du BIOS ou du DOS, qui ne sont plus disponibles *a priori* lorsqu'on est passé en mode protégé.

Il est très difficile de trouver dans la littérature un programme qui fonctionne. Il existe beaucoup de descriptions sur ce qu'il faut faire, reprenant toutes ce qui est dit dans [Int], volume 3, section 9, donnant lui-même le début d'un exemple sans plus. On trouve également beaucoup d'exemples en utilisant un moteur de recherche sur le Web, dont la principale caractéristique est qu'aucun ne fonctionne. Les deux exceptions sont un programme simple de Jerzy TARASIUK [Tar-95] et l'exemple PRO386.ASM donné dans [Sch92], ce dernier ne revenantt pas à MS-DOS. Nous allons adapter ici le premier programme.

Utilisons un descripteur de segment de données et un descripteur de segment de code pour le mode protégé, configuration la plus simple possible en mode protégé. Nous prenons comme segment de code celui choisi par MS-DOS en mode réel, de longueur 64 kiO. Le segment des données occupera, quant à lui, toute la mémoire possible, c'est-à-dire que son adresse de base sera 0 et sa longueur 4 GiO. On ajoute également un segment de données pour le retour au mode réel, confondu avec le segment des données choisi par MS-DOS. Le niveau de privilège est initialisé à 0, le plus haut niveau de privilège. Une telle configuration convient lorsqu'un seul utilisateur a accès au microprocesseur et qu'il a besoin de toute la mémoire.

Programme source.- Utilisons le programme `pm1.s` écrit en langage d'assemblage `gas` :

```
.section .text
.globl   _start
.code16
.org     0x100
_start:
#
# Save flags and DS for return to real mode
#
        movw    %cs,%dx
        movw    %dx,%ds
        movw    %dx,%ss
        pushf
        push    %ds
        movw    %ds,%dx
        movw    %dx,seg
#
#Setting base for code segments
#
        movw    %cs,%ax
        movzx   %ax,%eax
        shll    $4,%eax                 # eax=base for code segment
        movl    %eax,DESC1B
        movb    $0x9a,DESC1C            # set segment attribute
#
# Setting of GDTA
#
        movl    $0,%eax                 # deja fait mais prudence
        movl    $0,%ebx
        movw    %cs,%ax                 # deja fait mais prudence
```

```
        .byte   0x0bb                   # movw  DESC0,%bx
        .word   DESC0
        shll    $4,%eax                 # deja fait mais prudence
        addl    %ebx,%eax
        movl    %eax,GDTA
#
# Make sure no ISR will interfere now
#
        cli
#
# LGDT is necessary before switch to PM
#
        lgdtl   GDTL
#
# Switch to PM
#
        movl    %cr0,%eax
        orb     $1,%al
        movl    %eax,%cr0
#
# Far jump to set CS & clear prefetch queue
#
        .byte   0x0ea
        .word   pm_in
        .word   8
#
# Load long segment descriptor from GDT into DS
#
pm_in:  movw    $0x10,%dx
        movw    %dx,%ds
        movw    %dx,%ss
#-----------------------------;
# A sample in protected mode: ;
# A beep for each key pressed ;
#   except ESC, meaning end   ;
#-----------------------------;
        movb    $2,%al
        outb    %al,$0x21               # disable IRQ1
attend:
        inb     $0x64,%al               # wait for a key pressed
        andb    $1,%al
        cmpb    $1,%al
        jnz     attend
        inb     $0x60,%al               # get the character
        movb    %al,%bl
#
# make a beep
#
        inb     $0x61,%al
        orb     $3,%al
        outb    %al,$0x61
        movl    $0x0ff,%ecx
loop2:  movl    $0x0ffff,%edx
loop1:  decl    %edx
```

```
        cmpl    $0,%edx
        jnz     loop1
        loop    loop2
        inb     $0x61,%al
        andb    $0x0fc,%al
        outb    %al,$0x61
#
# end of beep
#
        cmpb    $1,%bl
        jnz     attend                  # again if no key ESC
        movb    $0,%al
        outb    %al,$0x21               # active IRQ1
#-------------------------------------;
# End of the sample in protected mode ;
#-------------------------------------;
#
# Load 64kB segment descriptor from GDT into DS for return
#
        movb    $0x18,%dl
        movw    %dx,%ds
#
# Return from PM ro real mode
#
        andb    $0x0fe,%al
        movl    %eax,%cr0
#
# Far jump to restore CS & clear prefetch queue
#
        .byte   0x0ea
        .word   pm_out
seg:    .word   0
pm_out:
#
# Restore DS and flags
#
        movw    %cs,%dx
        movw    %dx,%ss
        pop     %ds
        popf
#
# Exit to MS-DOS
#
        ret
#
#--------------------------------------;
#       GDT to be used in Protected Mode ;
#--------------------------------------;
#
# null descriptor
#
DESC0:  .long   0                       # null descriptor
        .long   0
#
```

```
# descriptor for code segment in PM
#
DESC1:  .word   0x0FFFF                 # limit 64kB
DESC1B: .word   0                       # base = 0 to set
        .byte   0                       # base
DESC1C: .byte   0x9A                    # code segment
        .byte   0                       # G = 0
        .byte   0
#
# descriptor for data segment in PM
#
DESC2:  .word   0x0FFFF                 # limit 4GB
        .word   0                       # base = 0
        .byte   0                       # base
        .byte   0x92                    # R/W segment
        .byte   0x8F                    # G = 1, limit
        .byte   0
#
# descriptor for data segment return to real mode
#
DESC3:  .word   0x0FFFF                 # limit 64kB
        .word   0                       # base = 0 to set
        .byte   0                       # base
        .byte   0x92                    # R/W segment
        .byte   0                       # G = 0, limit
        .byte   0                       # base
#
# GDT table data
#
GDTL:   .word   0x1F                    # limit
GDTA:   .long   0                       # base to set
```

Création de l'exécutable.- Assemblons ce fichier `pm1.s` sous Linux :

```
$ as pm1.s --32 -o ./pm1.o
$
```

On peut vérifier le code obtenu :

```
$ objdump -mi8086 -d ./pm1.o
```

Si on le transforme en code brut :

```
$ objcopy -O binary ./pm1.o ./pm1.com
$ ls -l pm1.com
-rwxr-xr-x 1 patrick patrick 463 sept. 13 09:31 pm1.com
$
```

on a 256 octets en trop à cause de l'origine à 256. Il ne faut donc garder que les 206 derniers octets :

```
$ objcopy -O binary -i 512 --interleave-width 206 -b 256 ./pm1.o ./pm1.com
$
```

Exécution.- Créons un répertoire '`protect`' sur la clé USB contenant MS-DOS et plaçons-y l'exécutable `pm1.com`.

Lorsqu'on lance le programme :

```
C:\PROTECT> pm1.com

C:\PROTECT>
```

on entend un bip (il faut bien écouter pour l'entendre) à chaque fois qu'on appuie sur une touche. Le programme se termine lorsqu'on appuie sur la touche d'échappement.

On revient alors en mode réel et même à MS-DOS, avec un prompteur à la ligne suivant la ligne à partir de laquelle on a lancé le programme.

Commentaires.- 1^o) La table globale des descripteurs comprend quatre descripteurs : le descripteur nul DESC0 obligatoire, le descripteur DESC1 du segment de code en mode protégé, le descripteur DESC2 du segment des données en mode protégé et le descripteur DESC3 du segment des données pour le retour au mode réel.

Ce sont les descripteurs 0 à 3, dans cet ordre, d'où les adresses (multiples de 8) : 0, 8, `0x10` et `0x18`. Les sélecteurs sont égaux à ces adresses.

2^o) La limite du segment de code est `0x0FFFF`. On a G = 0, d'après la valeur nulle du deuxième demi-octet de l'avant-dernier octet, donc une valeur de 64 Ki pour la limite.

Son adresse de base est pour l'instant 0, d'où le deuxième mot à 0 ainsi que le deuxième octet du troisième mot (n'oublions pas qu'ils sont inversés avec le choix petit-boutien des microprocesseurs *Intel*)) et le premier octet du quatrième mot.

Le bit D a la valeur 0, on sera donc dans le mode de programmation 16 bits. On reste pour l'instant, par prudence, dans le mode que l'on connaît.

L'octet des droits d'accès est `0x9A`, c'est-à-dire qu'il s'agit d'un descripteur valide (P = 1), de niveau de privilège 0 (DPL = 0), d'un segment de code (S = 1 et E = 1) que l'on peut lire (R = 1).

3^o) Le segment des données en mode protégé a une valeur du champ de limite égale à `0xFFFFF` avec une granularité de 1 (G = 1), donc une limite effective de 4 Gi.

Son adresse de base est également 0, mais elle ne sera pas modifiée.

L'octet des droits d'accès est `0x92`, c'est-à-dire qu'il s'agit d'un descripteur valide (P = 1), de niveau de privilège 0 (DPL = 0), d'un segment de données (S = 1 et E = 0), sur lequel on peut écrire (W = 1).

4^o) Le segment des données de retour au mode réel a les mêmes caractéristiques que le segment des données en mode protégé, à part sa granularité à 0 et la valeur du champ de limite égal à `0xFFFF`. Sa limite est donc de 64 Ki, le maximum pour le mode réel.

5^o) Le registre GDTR de GDT doit être renseigné avec la taille GDTL de la GDT et son adresse de base GDTA. On peut initialiser GDTL avec `0x1F` (= 4*8 - 1), puisque la GDT comprend 4 fois 8 octets. On ne connaît pas son adresse de base pour l'instant (puisque l'adresse du segment de code en mode réel sera choisie par MS-DOS).

6^o) Le programme commence au début puisqu'on veut un exécutable brut.

On est dans un modèle *tiny*, le segment des données et le segment de pile doivent donc être confondus avec le segment de code (choisi par MS-DOS).

7^o) On commence par sauvegarder le registre des indicateurs et le registre du segment des données sur la pile, puisqu'on en aura besoin lorsqu'on reviendra au mode réel.

8^o) On détermine ensuite l'adresse de base du descripteur du segment de code en mode protégé : elle doit être égale à la valeur du segment de code du mode réel (adresse choisie par le système d'exploitation MS-DOS).

Puisque nous sommes en mode réel, l'adresse physique est égale au contenu du registre `CS` multiplié par 16. On la place dans le premier mot double de l'adresse de base puisqu'on est sûr qu'elle occupe moins de 20 octets, mode réel oblige.

9^o) Le changement du mot double surcharge cependant l'octet des droits d'accès à zéro. Il faut donc replacer cette valeur, bien qu'elle ait déjà été initialisée avec la bonne valeur.

10^o) On détermine ensuite l'adresse de base de la GDT, qui est le décalage de DESC0 auquel on ajoute l'adresse de base du descripteur du segment de code, précédemment calculée (en fait on la recalcule ici).

On a un petit problème avec l'assembleur `gas` pour la traduction de l'instruction :

```
movw DESC0,%bx
```

ce qui n'arrive pas avec d'autres assembleurs. On la traduit donc à la main et on la place à l'endroit adéquat.

11^o) On inhibe les interruptions masquables avant le passage au mode protégé.

12^o) On initialise GDTR avec les valeurs préparées en GDTL et GDTA.

13^o) On peut alors passer au mode protégé en changeant le bit `PE` du registre de contrôle `CR0`.

14^o) On effectue ensuite un saut inter-segmentaire pour initialiser le registre de segment et vider la file d'attente de pré-extraction :

```
JMP CS:pm_in
```

Puisque l'assembleur, essayant d'optimiser, le traduirait en un saut court, on le traduit nous-même en code machine :

```
EA xx yy 08 00
```

`0xea` étant le code opération du saut inter-segmentaire, `0x0008` la valeur du sélecteur du segment de code en mode protégé (premier index de la GDT, TI = 0 et RPL = 0), la valeur du décalage `xx yy` étant celle de `pm_in`, à déterminer.

15^o) On initialise, en mode protégé, les registres de segment des données et de segment de pile à `0x10` (second index de la GDT, TI = 0 et RPL = 0).

16^o) On place ensuite ce que l'on veut en mode protégé, sur lequel nous reviendrons plus tard pour celui pris en exemple.

17^o) On veut ensuite revenir au mode réel : pour cela, on change de segment des données (troisième index de la GDT, TI = 0, RPL = 0) ; on change le bit `PE` du registre de contrôle `CR0` et on effectue un saut inter-segmentaire à l'instruction suivante (qu'il faut, comme ci-dessus, traduire à la main).

La documentation d'*Intel* n'est pas suffisamment claire sur le mode dans lequel on se trouve avant le saut. Il s'agit en fait déjà du mode réel puisque `CR0` a été changé. Il est facile d'obtenir le décalage pour ce saut. Il est un peu moins facile d'obtenir la valeur du registre de segment pour ce saut, d'où l'utilisation de la variable `seg`, initialisée (avec la valeur choisie par MS-DOS) au début du programme.

18^o) On restaure les valeurs des registres du segment de pile, du segment des données, des indicateurs et on rend la main à MS-DOS.

8.4 Test d'accès à la mémoire en mode protégé

Puisque le système n'a pas redémarré, on peut penser que nous sommes bien passés au mode protégé. Comment le vérifier et, sutout, vérifier qu'on a bien accès à toute la mémoire, et non plus au seul premier MiO ?

8.4.1 Accès à la mémoire en mode protégé

Contexte.- On peut vérifier que, après avoir démarré MS-DOS, la zone mémoire se situant après l'adresse `6000:0` n'est occupée que par des 0 :

```
C:\PROTECT>debug
-D 6000:0 L 10
6000:0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-q
```

Programme.- Écrivons un programme accèdant à la mémoire en mode protégé. Le programme `pm2.s` suivant est exactement le même que le programme `pm1.s` précédent à part le début, évidemment :

```
#-----------------------------------------------------;
# pm2.s                                               ;
# Example program in Protected Mode to access memory ;
#-----------------------------------------------------;
```

et la partie en mode protégé, maintenant réduite à une seule ligne :

```
#-----------------------------;
# A sample in protected mode: ;
#-----------------------------;
addr32  incw    %ds:0x60000
#-------------------------------------;
# End of the sample in protected mode ;
#-------------------------------------;
```

Création de l'exécutable.- Assemblons ce fichier `pm2.s` sous Linux :

```
$ as pm2.s --32 -o ./pm2.o
$ objcopy -O binary ./pm2.o ./pm2.com
$ ls -l pm2.com
-rwxr-xr-x 1 patrick patrick 410 sept. 15 09:40 pm2.com
$
```

Il ne faut donc garder que les 154 derniers octets du binaire :

```
$ objcopy -O binary -i 512 --interleave-width 154 -b 256 ./pm2.o ./pm2.com
$
```

Exécution.- Plaçons l'exécutable `pm2.com` dans le répertoire '`protect`' de la clé USB contenant MS-DOS. Après avoir fait exécuter le programme :

```
C:\PROTECT> pm1.com

C:\PROTECT>debug
-D 6000:0 L 10
6000:0 01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-q
```

on vérifie que l'emplacement mémoire a bien été modifié.

Commentaire.- Nous avons utilisé la directive `addr32` de `gas` pour spécifier, en mode d'instruction 16 bits, que l'on veut utiliser des adresses sur 32 bits.

8.4.2 Accès à la mémoire située au-delà du premier MiO

Le programme précédent nous a montré que l'on peut accéder à la mémoire en mode protégé. Il n'y avait en vérité aucun doute à cela. Nous avons également aussi montré comment passer des paramètres entre le mode réel et le mode protégé.

Comment montrer de façon probante que l'on peut accéder à la mémoire vive située au-delà du premier MiO ? Voyons d'abord l'état du processeur au démarrage.

État du processeur au démarrage.- Rappelons qu'un `8086` démarre avec `CS = 0x000F` et `IP = 0xFFF0` de façon à ce que la première instruction soit située 16 octets avant la fin de la mémoire atteignable. De la mémoire ROM doit donc se trouver à cet emplacement dans un ordinateur à base d'un tel processeur.

Pour l'architecture IA-32, on a toujours `EIP = 0x0000FFF0` mais le descripteur de CS ne reprend pas exactement les mêmes conditions que celles du `8086`. Le sélecteur est `0xF000`, l'adresse de base est `0xFFFF0000` et la limite `0xFFFF` ([Int], vol. 3, chapitre 9, section 1, Table 9-1 et section 9.1.4).

Par contre les autres registres (`SS`, `DS`, `ES`, `FS` et `GS`) ont tous un sélecteur égal à `0x0000`, une base de `0x00000000` et une limite de `0xFFFF`.

La limite initiale explique qu'on ne peut pas accéder à la mémoire se situant au-delà de 1 MiO.

La première instruction se trouve à l'adresse physique `0xFFFFFFF0`, c'est-à-dire toujours 16 octets avant la fin de la mémoire atteignable, qui est maintenant de 4 GiO et non plus de 1 MiO. On doit donc y trouver de la mémoire ROM dans un ordinateur à base d'un microprocesseur IA-32. Il doit donc y avoir deux blocs de ROM si on veut une compatibilité ascendante : un à la fin, avant 4 GiO, et un avant 1 MiO.

Contenu de la ROM extrême.- Qu'y a t-il à l'emplacement mémoire `0xFFFFFFF0` (en ROM) sur un ordinateur à base d'un processeur IA-32 ?

Écrivons un programme `pm3.asm` pour le découvrir. C'est le même programme que le précédent, sauf que la seule instruction en mode protégé du programme `pm2.asm` est remplacée par :

```
#-----------------------------;
# A sample in protected mode: ;
#-----------------------------;
        addr32  movl      %ds:0xFFFFFFF0,%eax
        addr32  movl      %eax,%ds:0x60000
        addr32  movb      %ds:0xFFFFFFF8,%al
        addr32  movl      %eax,%ds:0x60008
#-----------------------------------;
# End of the sample in protected mode ;
#-----------------------------------;
```

Assemblons ce fichier `pm3.s` sous Linux :

```
$ as pm3.s --32 -o ./pm3.o
$ objcopy -O binary ./pm3.o ./pm3.com
$ ls -l pm3.com
-rwxr-xr-x 1 patrick patrick 430 sept. 15 10:18 pm3.com
$
```

Il ne faut garder que les 174 derniers octets du binaire :

```
$ objcopy -O binary -i 512 --interleave-width 174 -b 256 ./pm3.o ./pm3.com
$
```

Après exécution du programme sous MS-DOS, on obtient :

```
C:\PROTECT>debug
-D 6000:0 L 10
6000:0 E9 1F E3 00 00 00 00 00-EA 1F E3 00 00 00 00 00 .................
-q
```

On a donc un saut inconditionnel :

```
jmp E31Fh
```

à l'instruction d'adresse `0xFFFFFFF3 + 0xE31F`, soit `0xFFFFE312`.

Explorons le contenu de la mémoire à cet emplacement `0xFFFFE312`, avec un programme analogue au précédent (en jouant sur `EAX`, `AX` ou `AL` si l'ordinateur « a du mal » à accepter `EAX`). On obtient sur mon système :

```
FA 66 C1 E2 10 8E ...
```

soit une instruction CLI suivie d'une rotation sur 32 bits (qui a peu de chances de se trouver dans le premier MiO de la mémoire). Notre propos n'est pas d'étudier ici le contenu du BIOS, aussi n'allons-nous pas aller plus loin dans l'exploration.

Exercice.- *Écrire un programme qui place un 1 à tous les emplacements mémoire au-delà de 1 MiO (pas avant pour éviter de détruire quoi que ce soit qui nous empêchera de revenir au mode réel) puis qui effectue la somme de tous ces emplacements mémoire.*

8.5 Les instructions en mode protégé

Nous avons déjà vu les noms symboliques des instructions, et donc la syntaxe, à propos du mode réel. Il n'y a pas de changement de syntaxe en mode protégé mais la sémantique, elle, change un peu.

8.5.1 Le mode d'instructions

Nous avons vu que le mode d'instructions spécifie si les opérandes ont une taille de 16 bits ou de 32 bits. Par défaut, le mode est de 16 bits en mode réel ; il est choisi par le bit D du descripteur du segment de code en mode protégé. Ce mode par défaut n'empêche pas qu'on puisse changer le mode d'une instruction en utilisant les préfixes `0x67` d'adresse et `0x66` d'opérande, conformément au tableau suivant (avec N pour non présent et O pour oui, c'est-à-dire présent) :

Bit D du descripteur	0				1			
Préfixe `0x66`	N	N	O	O	N	N	O	O
Préfixe `0x67`	N	O	N	O	N	O	N	O
Opérande sur (bits)	16	16	32	32	32	32	16	16
Adresse sur (bits)	16	32	16	32	32	16	32	16

La taille de l'adresse concernant la pile est déterminée par le bit B du descripteur de segment de donnée : 0 spécifie 16 bits (et donc l'usage du registre `SP`) et 1 une longueur de 32 bits (et donc l'utilisation de `ESP`).

8.5.2 Les routines en mode protégé

Principe.- En mode protégé, comme en mode réel, on appelle un sous-programme par l'instruction `CALL`, le code du sous-programme devant se terminer par `RET`.

Exemple.- Écrivons un programme `call1.s`, qui reprend le programme `pm3.s` dans lequel la partie exemple en mode protégé n'est constitué que d'un appel à un sous-programme :

```
#-----------------------------;
# A sample in protected mode: ;
#-----------------------------;
        call    essai
#-------------------------------------;
# End of the sample in protected mode ;
#-------------------------------------;
```

et en ajoutant le code suivant entre l'instruction `ret` de retour à MS-DOS et la déclaration de la GDT :

```
#---------;
# Routine ;
#---------;
essai:  ret
```

autrement un sous-programme qui ne fait rien.

Assemblons ce fichier `call1.s` sous Linux :

```
$ as call1.s --32 -o ./call1.o
$ objcopy -O binary ./call1.o ./call1.com
$ ls -l call1.com
-rwxr-xr-x 1 patrick patrick 407 sept. 18 09:52 call1.com
$
```

Il ne faut garder que les 151 derniers octets du binaire :

```
$ objcopy -O binary -i 512 --interleave-width 151 -b 256 ./call1.o ./call1.com
$
```

On s'aperçoit alors que le programme s'exécute sous MS-DOS, ce qui est déjà ça.

Commentaire.- Les seuls changements par rapport au programme, maintenant standard, de passage au mode protégé et retour au mode réel, est un appel à un sous-programme comme exemple de code en mode protégé. Ce sous-programme `essai` se réduit à sa plus simple expression, puisqu'il se contente de l'instruction de retour.

8.5.3 Rappels sur l'affichage de l'IBM-PC

Le problème avec le mode protégé est que nous ne pouvons plus utiliser les interruptions de MS-DOS ou du BIOS (comme nous le verrons dans le chapitre suivant). Si on veut faire afficher quelque chose avant de revenir au mode réel, il faut se passer des interruptions. Heureusement ceci est relativement simple si on reste dans le mode graphique spécifié par le BIOS ou MS-DOS.

Jeu de caractères.- L'affichage du texte sur un IBM-PC s'effectue avec un jeu de caractères ASCII étendu, comprenant 256 caractères numérotés de 0 à 255.

Attributs d'un caractère.- Un caractère nécessite un octet pour le déterminer, puisqu'il y en a 256. Un second octet, dit **octet d'attribut**, est cependant consacré à chaque emplacement de l'écran, pour indiquer le mode d'affichage (souligné, clignotant, inversé, ...) du caractère à afficher. Le mot pour un caractère, défini par IBM avec la même structure pour chacune des deux premières cartes graphiques du PC (la carte MDA [pour *Monochrome Display Adapter*], et la carte CGA [pour *Color Graphic Adapter*]), conçues en même temps (l'une pour les entreprises, l'autre pour les utilisateurs individuels), a le format suivant :

- Les bits 0 à 7 indiquent le **code** du caractère. Ce code sert d'index dans la **table des caractères** se trouvant en ROM caractères, déterminant la graphie de ceux-ci.
- Le bit 15 (`BLNK`, pour *BLiNK*), indique que le caractère doit clignoter lorsqu'il vaut 1. Suivant la carte graphique et le constructeur, le caractère clignote à une fréquence comprise entre 1 Hz et 3 Hz.
- Les bits 12 à 14 (BAK_0, BAK_1, BAK_2, pour *BACKground*) spécifient la couleur de fond du caractère. Huit couleurs différentes sont donc possibles. La couleur dépend de la carte graphique, de la palette de couleur sélectionnée et du fait qu'il s'agisse d'un moniteur monochrome ou couleur.
- Le bit 11 (`INT` pour *INTensity*) fait apparaître le caractère plus brillant (plus intense) lorsqu'il vaut 1.
- Les bits 8 à 10 (FOR_0, FOR_1, FOR_2 pour *FOReground*) spécifient la couleur de premier plan du caractère.

En ce qui concerne la carte MDA, ces bits peuvent être définis librement mais seules certaines combinaisons de bits pour les couleurs de premier plan et de fond produiront un affichage de caractères valables :

7	6	5	4	3	2	1	0	Effet
	0	0	0		0	0	0	Pas d'affichage (noir sur noir, pour les mots de passe)
	0	0	0		0	0	1	Caractère souligné
	0	0	0		1	1	1	Caractère normal (blanc sur noir)
	1	1	1		0	0	0	Caractère inversé (noir sur blanc)
	1	1	1		1	1	1	Contour du caractère blanc (blanc sur blanc)

Emplacement et capacité de la mémoire graphique.- Puisque chaque caractère nécessite deux octets, pour un affichage de 25 lignes de 80 caractères, la mémoire requise s'élève à 4 000 octets.

La mémoire graphique de la carte MDA commence à l'adresse `B000:000` et occupe 4 KiO (soit 4 096 octets).

Adresse dans la mémoire graphique.- La mémoire graphique est vue comme un tableau linéaire. Le premier mot concerne le caractère du coin supérieur gauche, c'est-à-dire de la ligne 1, colonne 1, le second mot celui de la ligne 1, colonne 2, et ainsi de suite.

L'adresse du mot mémoire pour le caractère de la ligne i, colonne j est donnée par l'équation suivante :

$$adresse = sv + 2 * ncpl * i + 2 * j$$

où sv (pour *Segment Video*) désigne l'adresse de début de la mémoire graphique (donc `0xB000`) et $ncpl$ le nombre de caractères par ligne (le BIOS a choisi 80). Les variables i et j commencent à 0.

8.5.4 Un sous-programme d'affichage d'un caractère

Écrivons maintenant un programme `call2.s` contenant un sous-programme `wrchr` d'affichage du caractère dont le code ASCII est contenu dans le registre `AL`.

La partie exemple en mode protégé fait appel deux fois à cette routine, une fois pour afficher 'A', une fois pour afficher 'B' :

```
#-----------------------------------;
# call2.s                          ;
# Routine for displaying a character ;
#-----------------------------------;
.section .text
.globl   _start
.code16
.org     0x100
_start:
#
# Save flags and DS for return to real mode
#
        movw    %cs,%dx
        movw    %dx,%ds
        movw    %dx,%ss
        pushf
        push    %ds
        movw    %ds,%dx
        movw    %dx,seg
#
#Setting base for code segments
#
        movw    %cs,%ax
        movzx   %ax,%eax
        shll    $4,%eax                 # eax=base for code segment
        movl    %eax,DESC1B
        movb    $0x9a,DESC1C            # set segment attribute
#
# Setting base for data segment beginning as code segment
#
        movl    %eax,DESC4B
        movb    $0x92,DESC4C
#
# Setting of GDTA
#
        movl    $0,%eax                 # deja fait mais prudence
        movl    $0,%ebx
        movw    %cs,%ax                 # deja fait mais prudence
        #movw   DESC0,%bx
```

```
        .byte   0x0bb
        .word   DESC0
        shll    $4,%eax                 # deja fait mais prudence
        addl    %ebx,%eax
        movl    %eax,GDTA
#
# Make sure no ISR will interfere now
#
        cli
#
# LGDT is necessary before switch to PM
#
        lgdtl   GDTL
#
# Switch to PM
#
        movl    %cr0,%eax
        orb     $1,%al
        movl    %eax,%cr0
#
# Far jump to set CS & clear prefetch queue
#
        .byte   0x0ea
        .word   pm_in
        .word   8
#
# Load long segment descriptor from GDT into DS
#
pm_in:  movw    $0x10,%dx
        movw    %dx,%ds
        movw    %dx,%ss
        movw    %dx,%es
        movw    %dx,%gs
#-----------------------------;
# A sample in protected mode: ;
#-----------------------------;
        movb    $65,%al
        call    wrch
        incb    %al
        call    wrch
#-----------------------------------;
# End of the sample in protected mode ;
#-----------------------------------;
#
# Load 64kB segment descriptor from GDT into DS for return
#
        movb    $0x18,%dl
        movw    %dx,%ds
#
# Return from PM ro real mode
#
        andb    $0x0fe,%al
        movl    %eax,%cr0
#
```

```
# Far jump to restore CS & clear prefetch queue
#
        .byte   0x0ea
        .word   pm_out
seg:    .word   0
pm_out:
#
# Restore DS and flags
#
        movw    %cs,%dx
        movw    %dx,%ss
        pop     %ds
        popf
#
# Exit to MS-DOS
#
        ret
#---------;
# Routine ;
#---------;
#-------------------------------------------------;
# character-output video routine in Protected Mode ;
# Display character whose ASCII code is in al      ;
#-------------------------------------------------;
wrch:   push    %gs
        push    %ecx
        push    %ebx
        push    %eax
        movw    $0x20,%bx               # Data selector as Real
        movw    %bx,%gs
#
# (Y * 80 + X) * 2 --> EAX
#
        xorl    %eax,%eax
        movb    %gs:CsrY,%al
        movb    $80,%cl
        mulb    %cl
        addb    %gs:CsrX,%al
        adcb    $0,%ah
        shll    $1,%eax
#
# EAX + 0xB8000 --> EBX
#
        movl    $0x0B80A0,%ebx
        addl    %eax,%ebx
        pop     %eax
        push    %eax
#
# store char in video memory
#
        movb    %al,%ds:(%ebx)
#
# advance cursor
#
```

```
        movw    %gs:CsrX,%cx
        incb    %cl
        cmpb    $80,%cl                 # cursor off right side of screen?
        jb      wrch2
        xorb    %cl,%cl                 # yes, wrap to left side...
        incb    %ch                     # ...and down one line
        cmpb    $25,%ch                 # cursor off bottom of screen?
        jb      wrch2
        xorb    %ch,%ch                 # yes, wrap to top left corner
                                        # (no scroll)
wrch2:  movb    %cl,%gs:CsrX
        movb    %ch,%gs:CsrY
        pop     %eax
        pop     %ebx
        pop     %ecx
        pop     %gs
        ret
#
#---------------------------------------;
#       GDT to be used in Protected Mode ;
#---------------------------------------;
#
# null descriptor
#
DESC0:  .long   0                       # null descriptor
        .long   0
#
# descriptor for code segment in PM
#
DESC1:  .word   0x0FFFF                 # limit 64kB
DESC1B: .word   0                       # base = 0 to set
        .byte   0                       # base
DESC1C: .byte   0x9A                    # code segment
        .byte   0                       # G = 0
        .byte   0
#
# descriptor for data segment in PM
#
DESC2:  .word   0x0FFFF                 # limit 4GB
        .word   0                       # base = 0
        .byte   0                       # base
        .byte   0x92                    # R/W segment
        .byte   0x8F                    # G = 1, limit
        .byte   0
#
# descriptor for data segment return to real mode
#
DESC3:  .word   0x0FFFF                 # limit 64kB
        .word   0                       # base = 0 to set
        .byte   0                       # base
        .byte   0x92                    # R/W segment
        .byte   0                       # G = 0, limit
        .byte   0                       # base
#
```

```
# descriptor for data segment beginning as code segment
#
DESC4:  .word   0x0FFFF                 # limit 64kB
DESC4B: .word   0                       # base = 0 to set
        .byte   0                       # base
DESC4C: .byte   0x92                    # R/W segment
        .byte   0                       # G = 0, limit
        .byte   0                       # base
#
# GDT table data
#
GDTL:   .word   0x27                    # limit
GDTA:   .long   0                       # base to set
#
# Position of cursor
#
CsrX:   .byte   0
CsrY:   .byte   2
```

Assemblons ce fichier `call2.s` sous Linux :

```
$ as call2.s --32 -o ./call2.o
$ objcopy -O binary ./call2.o ./call2.com
$ ls -l call2.com
-rwxr-xr-x 1 patrick patrick 529 sept. 18 16:47 call2.com
$
```

Il ne faut garder que les 273 derniers octets du binaire :

```
$ objcopy -O binary -i 700 --interleave-width 273 -b 256 ./call2.o ./call2.com
$
```

On s'aperçoit alors que le programme s'exécute sous MS-DOS et qu'on surcharge alors par 'AB' le début de la troisième ligne.

Commentaires.- 1^o) Il ne suffit pas de faire appel au sous-programme et d'écrire le code de ce sous-programme. Nous avons également ajouté un segment de données, utilisé en mode protégé, dont l'adresse de base est la même que celle du segment de code (choisie par MS-DOS, là où il y a de la place, rappelons-le !). Ceci est indispensable car on fait référence à des emplacements mémoire, dont le décalage est relatif à cette adresse.

- 2^o) La limite de la GDT a donc changé, passant de `0x1F` à `0x27`.

- 3^o) Il faut donc, outre la déclaration de ce segment, ajuster son adresse de base, comme on le fait pour le segment de code.

- 4^o) Le code exemple en mode protégé demande d'afficher le caractère 'A' puis le caractère 'B'.

- 5^o) On déclare les emplacements mémoire `CsrX` et `CsrY` contenant l'abscisse et l'ordonnée du curseur, initialisées à 0 (première colonne) et 2 (troisième ligne).

On ne fait pas apparaître ces caractères à la première ligne car, après l'exécution du programme, on passe à la ligne suivante pour écrire le prompteur. Ce qu'on a écrit devrait encore se voir sur la deuxième ligne lorsqu'on est revenu au mode réel.

- 6^o) Pour la routine, les emplacements mémoire faisant intervenir les variables sont relatifs au segment dont l'adresse de base est celle donnée par MS-DOS. On peut utiliser `CS` pour la lecture mais pas pour l'écriture. On initialise donc `GS` avec le segment de données ajouté et on se réfère à ce segment.

- 7^o) La mémoire graphique, commençant à l'adresse `0xB8000`, exige l'utilisation d'une adresse absolue. On se sert donc du segment de données commençant à l'adresse 0.

8.5.5 Un sous-programme d'affichage d'une chaîne de caractères

Allons plus loin. Écrivons un sous-programme `wrstr` affichant la chaîne de caractère dont la suite de codes ASCII, se terminant par zéro, se trouve au décalage contenu dans le registre `ESI`. Pour cela, changeons l'exemple en mode protégé :

```
#----------------------------;
# A sample in protected mode: ;
#----------------------------;
        movl    $hi_msg,%esi
        call    wrstr
#-----------------------------------;
# End of the sample in protected mode ;
#-----------------------------------;
```

Ajoutons une routine, après la routine d'affichage d'un caractère :

```
#----------------------------------;
#       string-output video routine ;
#       Display string in esi       ;
#       ended by 0                  ;
#----------------------------------;
wrstr:  push    %esi
        push    %eax
        cld
        jmp     wrstr2
wrstr1: call    wrch
wrstr2: movw    $0x20,%bx
        movw    %bx,%ds
        lodsb
        movw    $0x10,%bx
        movw    %bx,%ds
        orb     %al,%al
        jne     wrstr1
        pop     %eax
        pop     %esi
        ret
```

et, évidemment, ajoutons le message quelque part, disons à la fin :

```
#
# Position of cursor
#
CsrX:   .byte   0
CsrY:   .byte   2
#
# Message
#
hi_msg: .ascii  "Bonjour"
.byte   0
```

Assemblons ce fichier `call3.s` sous Linux :

```
$ as call3.s --32 -o ./call3.o
$ objcopy -O binary ./call3.o ./call3.com
$ ls -l call3.com
-rwxr-xr-x 1 patrick patrick  566 sept. 22 09:44 call3.com
$
```

Il ne faut garder que les 310 derniers octets du binaire :

```
$ objcopy -O binary -i 700 --interleave-width 310 -b 256 ./call3.o ./call3.com
$
```

On s'aperçoit alors que le programme s'exécute sous MS-DOS et qu'on surcharge alors par 'Bonjour' le début de la troisième ligne.

Commentaire.- Remarquez dans l'écriture de la routine `wrstr` la nécessité, pour faire appel à l'instruction `LODSB`, d'utiliser comme segment de données, non pas celui absolu commençant à 0, mais celui commençant à l'adresse décidée par MS-DOS, puisque le décalage contenu dans `ESI` est relatif à celui-ci.

8.6 Bibliographie

[Int] **Intel 64 and IA-32 Architectures Software Developper's**, 3 volumes set.

[Sch92] SCHAKEL, Holger, Data Becker, 1992. Traduction française **Programmer en assembleur sur PC**, Éditions Micro Application, 1995, 512 p. + disquette.

[Tar-95] TARASIUK, Jerzy, **SIMPL_PM.txt**, 8 June 1995. Téléchargeable.