Quatrième partie

Démarrage d'un système 64 bits

Chapitre 13

Démarrage d'un système 64 bits

Comme nous l'avons déjà dit dans l'introduction, le démarrage d'un microprocesseur *Intel* à architecture x86-64 ressemble au lancement d'une fusée à trois étages :

— Lors d'une réinitialisation, un tel microprocesseur se trouve en *mode réel*, dans lequel il se comporte comme un `8086` (avec quelques instructions supplémentaires).

— Une instruction `mov` sur le registre 32 bits `CR0`, non présent sur le `8086`, permet de passer au *mode protégé*, mode par excellence des microprocesseurs *Intel* d'architecture IA-32.

— Une fois en mode protégé on peut, si on le désire, pour un microprocesseur d'architecture x86-64 (mais pas avec l'architecture IA-32), dit aussi **IA-32e** (avec 'e' pour *Extended*), passer dans un nouveau mode, appelé **mode long**, comportant deux sous-modes, dont l'un d'entre eux est appelé **mode 64 bits**.

13.1 Le mode 64 bits

Pour le mode 64 bits, on revient au modèle plat de la mémoire, se passant avantageusement ainsi du modèle segmenté du mode réel.

13.1.1 Modèle plat de mémoire

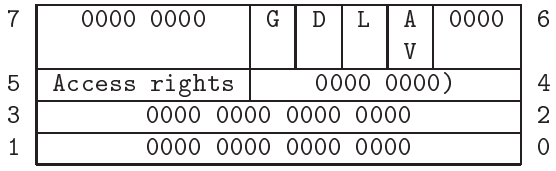
Bien qu'un microprocesseur 64 bits puisse *a priori* manipuler des adresses physiques sur 64 bits (c'est-à-dire avec 64 broches d'adresse), elles sont, tout au moins pour l'instant, restreintes à 36 bits (depuis le *Pentium Pro*, d'architecture IA-32 et non IA-32e d'ailleurs), permettant d'accéder à 64 GiO de mémoire vive, ou à 40 bits, permettant d'accéder à 1 TiO (un téra-octet) de mémoire, pour l'architecture IA-32e.

Le modèle de mémoire utilisé dans le sous-mode 64 bits est ce qui appelé **modèle plat** (*flat mode memory* en anglais) dans lequel il n'y a pas de segmentation : l'adresse physique du premier octet de la mémoire est `0x00 0000 0000` et celle du dernier est `0xFF FFFF FFFF`.

Les registres de segment n'ont donc plus d'utilité *a priori*. En fait le registre de segment de code `CS` est encore utilisé pour spécifier un descripteur précisant les droits d'accès (mais non son adresse de base et sa limite). Les données et la pile sont contenues dans le segment de code.

13.1.2 Descripteur du segment de code

La structure d'un descripteur (de segment de code donc) est celle de l'architecture IA-32 mais dont seuls quelques champs sont pris en compte :



L'adresse de base et la limite ne sont pas pris en compte. Il y a par contre un nouveau champ, le bit **L** (pour *Long* bien qu'*Intel* l'appelle le bit 64), permettant de choisir entre des adresses sur 64 bits (`L = 1`) ou sur 32 bits (`L = 0`), comme nous le verrons dans la sous-section suivante.

Le registre `CS` spécifie donc un segment de code : il débute nécessairement à l'adresse 0, il n'y a pas de limite mais il y a des droits d'accès et on choisit la taille des adresses par défaut.

13.1.3 Le mode long

Le mode long a deux sous-modes, ou **modes d'opération**, appelés **mode compatible** (*Compatibility Mode Operation*) et **mode 64 bits** (*64-bit Mode*, voir [Int], volume 3, §9.8.5.3). On utilise les deux bits `CS.L` et `CS.D` du descripteur du segment de code pour spécifier le mode d'opération, une fois le mode long initialisé :

— Lorsque `CS.L = 1` et `CS.D = 0`, le processeur opère en **mode 64 bits**. Dans ce mode d'opération, par défaut la taille d'opérande est de 32 bits et la taille d'adresse 64 bits. En utilisant les préfixes d'adresse, d'opérande et un nouveau préfixe, appelé `REX`, la taille d'opérande peut être changée en 64 bits ou 16 bits, la taille d'adresse peut être changée en 32 bits.

— Lorsque `CS.L = 0`, le processeur opère en **mode compatible**. Dans ce mode, `CS.D` contrôle les tailles d'opérande et d'adresse par défaut de la même façon qu'en mode protégé :

  — `CS.D = 1` spécifie les tailles d'adresse et d'opérande par défaut sur 32 bits.

  — `CS.D = 0` spécifie les tailles d'adresse et d'opérande par défaut sur 16 bits.

— La combinaison `CS.L = 1` et `CS.D = 1` est réservée.

13.1.4 Les registres

En mode 64 bits, les registres ont une taille de 64 bits, permettant ainsi d'effectuer des opérations arithmétiques sur 64 bits en une seule opération.

Registres généraux.- Les 64 bits des registres généraux ne peuvent être utilisés que dans ce sous-mode 64 bits. Les registres généraux sont désignés par **RAX**, **RBX** et ainsi de suite pour les huit premiers, **R8**, **R9**, ..., **R15** pour les huit suivants (avec 'R' pour *Re-extended*). On peut donc écrire en langage symbolique :

```
add rcx,rbx
```

Les huit derniers registres peuvent être adressés comme octet, mot, mot double ou **mot quadruple** (*quadword* en anglais), mais seulement ceux de poids faible. On utilise les suffixes **b**, **w** et **d** en langage symbolique pour les désigner. On a, par exemple :

```
mov r9b,r10b
mov r10w,ax
mov r14d,r15d
mov r13,r12
```

Si on veut vraiment accéder à d'autres parties, il y a la possibilité d'utiliser des rotations.

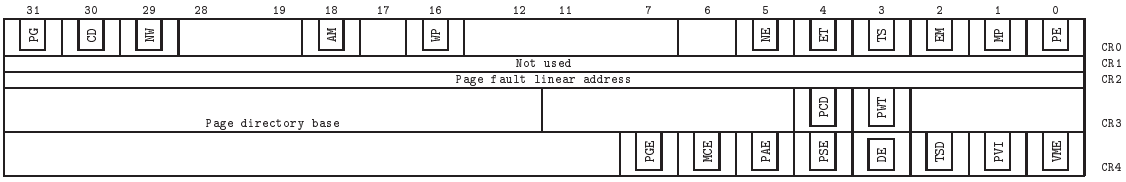
Registre des indicateurs et pointeur d'instructions.- Le registre des indicateurs a également une taille de 64 bits, appelé alors **RFLAGS**, mais seuls les 32 bits de poids faible ont une signification. On retrouve alors le registre `EFLAGS` de l'architecture IA-32.

Le pointeur d'instruction **RIP** est un registre de 64 bits mais seuls les 40 bits de poids faible sont pris en compte pour l'instant.

Registres de segment.- Il n'y a pas d'évolution par rapport à IA-32 : il y a six registres de segments (`CS`, `DS`, `ES`, `SS`, `FS` et `GS`) de 16 bits mais seul le contenu du registre `CS` a une incidence dans le sous-mode 64 bits. Les autres registres sont traités comme ayant un contenu nul.

Registres de tables de descripteurs.- Les registres `GDTR`, `LDTR`, `IDTR` et `TR` ont un champ d'adresse de base sur 64 bits (et non plus sur 32 bits).

Registres de contrôle.- Depuis le *Pentium*, il existe un cinquième registre de contrôle, dénommé **CR4**. La structure des registres de contrôle est alors la suivante :



— Le registre `CR0` a quatre bits ayant une incidence de plus :
  — Le bit 30, appelé **CD** (pour *Cache Disable*) contrôle le cache interne :
    — Lorsque `CD = 1`, le cache ne peut plus être surchargé par de nouvelles données mais il continue à fonctionner pour les données déjà en place.
    — Lorsque `CD = 0`, de nouvelles données peuvent être ajoutées ou peuevnt remplacer d'anciennes données.
  — Le bit 29, appelé **NW** (pour *Not Write-through*), permet de choisir le mode cache des données. Si `NW = 1`, le cache est inhibé.
  — Le bit 18, appelé **AM** (pour *Aligment Mask*), active la vérification de l'alignement lorsqu'il est positionné, mais uniquement pour le niveau de privilège 3 du mode protégé.
  — Le bit 5, appelé **NE** (pour *Numeric Error*), active la prise en compte des erreurs du coprocesseur arithmétique. Si `NE = 1`, la broche $\overline{\texttt{FERR}}$ devient active pour une erreur du coprocesseur numérique. Si `NE = 0`, toute erreur du coprocesseur est ignorée.
— Cinq bits seulement du registre `CR4` ont une signification :
  — Le bit 0, appelé **VME** (pour *Virtual Mode Extension*), active le support de l'indicateur d'interruption virtuelle du registre des indicateurs en mode protégé lorsque `VME = 1`.
  — Le bit 1, appelé **PVI** (pour *Protected Mode Virtual Interrupt*), active ? ?
  — Le bit 2, appelé **TSD** (pour *Time Stamp Disable*), contrôle l'instruction `RDTSC`.
  — Le bit 3, appelé **DE** (pour *Debugging Extension*), active les extensions de débogage au point d'arrêt des entrées-sorties lorsqu'il est positionné.
  — Le bit 4, appelé **PSE** (pour *Page Size Extension*), permet d'obtenir une taille de pages de 4 MiO, au lieu de 4 kiO, lorsqu'il est égal à 1.
  — Le bit 6, appelé **MCE** (pour *Machine Check Enable*), active l'interruption de vérification de la machine.

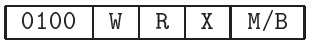
Depuis le *Pentium Pro*, pour lequel on peut passer d'un adressage mémoire de 4 GiO à un adressage sur 64 GiO, grâce à quatre broches d'adresse supplémentaires, il y a deux bits significatifs de plus pour le registre `CR4` :

— Le bit 7, appelé **PGE** (pour *PaGe Extension*), contrôle le nouveau mode d'adressage de 64 GiO lorsqu'il est positionné, conjointement avec les bits `PAE` et `PSE`.
— Le bit 5, appelé **PAE** (pour *Page Address Extension*), active les broches d'adresse `A35-A32` lorsque le nouveau mode spécial d'adressage, contrôlé par le bit `PGE`, est activé.
— Le bit 4, appelé **PSE**, a la même signification si `PGE` n'est pas positionné. Si `PGE` et `PSE` sont positionnés, la taille d'une page est de 2 MiO.

13.1.5 Codage des instructions

En mode 64 bits, on peut ajouter un nouveau préfixe, appelé **REX** (pour *Register EXtension*), devant être placé entre les préfixes déjà connus pour IA-32 et le code opération. Sa valeur, comprise entre `0x40` et `0x4F`, modifie la signification des champs `reg` et `r/m` du second octet (éventuel) du code opération de l'instruction. Il est, en particulier, nécessaire pour pouvoir accéder aux registres `R8` à `R15`.

La structure de `REX` est :



— Le bit **W** (pour *Word*) permet de choisir entre 64 bits (`W = 1`) et la taille spécifiée par le descripteur de CS (`W = 0`).

— Le bit **R** (pour *Register*)permet d'ajouter un quatrième bit, celui de poids fort, aux trois bits du champ `reg` du second octet du code opération. Le registre sélectionné est alors spécifié par le tableau suivant :

Code	Registre	Mémoire
0000	RAX	[RAX]
0001	RCX	[RCX]
0010	RDX	[RDX]
0011	RBX	[RBX]
0100	RSP	
0101	RBP	[RBP]
0110	RSI	[RSI]
0111	RDI	[RDI]
1000	R8	[R8]
1001	R9	[R9]
1010	R10	[R10]
1011	R11	[R11]
1100	R12	[R12]
1101	R13	[R13]
1110	R14	[R14]
1111	R15	[R15]

— Le bit **M** (pour *Mode*) permet d'ajouter un quatrième bit, celui de poids fort, aux trois bits du champ `r/m` du second octet du code opération. Le tableau précédent donne l'interprétation, sauf dans le cas `0100b`.

— En effet, si `r/m = 0100b`, le second octet du code opération est suivi d'un octet d'index avec échelle. Dans le cas de l'architecture IA-32 sa structure est `Scale-Index-Base`, les champs ayant respectivement 2, 3 et 3 bits. Dans le cas du mode 64 bits, le bit X donne le bit de poids fort de l'échelle, qui peut donc être 1×, 2×, 4× ou 8×. Le bit `B` est alors utilisé comme quatrième bit, celui de poids fort, pour déterminer la base.

Remarque.- Comment faire la différence entre l'octet `REX` et le code opération qui viendrait alors ?

Remarquons que, pour l'architecture IA-32, les codes opération `0x40-0x4F` sont ceux de certaines variantes, dites *courtes*, des instructions `INC` et `DEC`. En mode 64 bits, les valeurs `0x40` à `0x4F` ne sont pas des codes opérations (mais des valeurs de `REX`). Si on veut utiliser ces variantes de `INC` et `DEC`, on doit nécessairement en utiliser la version longue, la version courte n'existant plus en mode 64 bits.

13.1.6 Nouvelles instructions IA-32

Le *Pentium* a introduit de nouveaux registres et de nouvelles instructions dont deux nous sont utiles pour démarrer le mode 64 bits.

Compteur de temps.- Le compteur de temps **TSC** (pour l'anglais *Time-Stamp Counter*) contient le nombre de tops d'horloge depuis que le microprocesseur a été réinitialisé plus une constante, `TSC` étant initialisé de façon aléatoire à chaque redémarrage. Il s'agit d'un compteur de 64 bits si bien qu'un microprocesseur cadencé à 1 GHz peut accumuler un décompte effectué sur 580 ans sans revenir à sa valeur initiale.

L'instruction **RDTSC** (pour *ReaD TSC*) lit le `TSC` et en place son contenu dans `EDX:EAX`, `EDX` contenant le mot double de poids fort. On utilise deux registres de 32 bits puisque les registres de 64 bits n'existaient pas sur le *Pentium* et qu'il faut conserver la compatibilité.

Registres spécifiques à un modèle.- À partir du *Pentium*, les microprocesseurs *Intel* possèdent un certain nombre de registres, *a priori* spécifiques à chaque modèle de microprocesseur (**MSR** pour *Model Specific Register*). Le chapitre 35 du volume 3 de [Int] documente certains de ces deux milliers de registres.

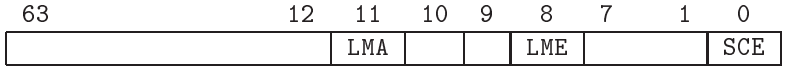
Les instructions **RDMSR** (pour *ReaD Model Specific Register*) et **WRMSR** (pour *WRite Model Specific Register*), exécutables en mode réel ainsi qu'en mode protégé au niveau 0 seulement, permettent de lire ces registres, et, pour certains d'entre eux, d'y écrire. Le numéro (*adresse*) du registre `MSR` doit d'abord être placé dans le registre `ECX` avant une opération ; avant une écriture, la donnée à écrire doit se trouver dans `EDX:EAX`.

Par exemple le registre d'adresse `0x10` permet de retrouver `TSC`, qu'on peut lire, comme avec `RDTSC`, mais également écrire. La plage d'adresses pour un *Pentium* va de `0x00` à `0x13`. Les microprocesseurs *Pentium 4* et *Core 2* contiennent 1 743 MSR, d'adresses allant de `0x00` à `0x6CF`.

L'instruction `RDMSR` est codée par `0000 1111 0011 0010b`, soit `0xF32`. alors que `WRMSR` l'est par `0000 1111 0011 0000b`, soit `0xF30`.

Registre EFER.- *Intel* a décidé que certains `MSR`, registres *a priori* spécifiques à un modèle, seront présents avec les mêmes fonctions sur tous les microprocesseurs suivants le modèle dans lequel ils ont été introduits (d'où une compatibilité ascendante). Leurs noms commencent alors par `IA32_`.

Il en est ainsi, depuis le *Core2*, du `MSR` d'adresse `0xC0000080`, appelé **IA32_EFER** (le suffixe signifiant *Extended FEatuRes enables*), dont la structure est :

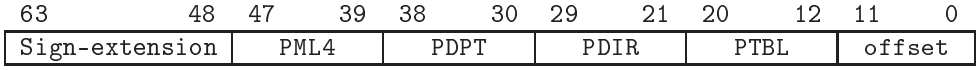


- On positionne le bit 0 (**SCE** pour *System Call Enable*), pour activer les instructions `SYSCALL` et `SYSRET` en mode 64 bits.
- On positionne le bit 8 (**LME** pour *Long Mode Enable*), pour passer au mode long.
- Le bit 11 (**LMA** pour *Long Mode Active*), en lecture uniquement, montre, lorsqu'il est égal à 1, que le microprocesseur opère en mode 64 bits.

13.1.7 Pagination

La mise en place de la pagination évolue, par rapport à celle de l'architecture IA-32, pour le mode 64 bits de façon à ce que l'unité de pagination puisse traduire des adresses virtuelles sur 64 bits (donc sur une plage de 16 exa-octets) en adresses physiques sur 52 bits (donc sur une plage de 4 péta-octets). Dans une première étape, *Intel* utilise des adresses virtuelles sur 48 bits (donc sur une plage de 256 téra-octets) et des adresses physiques sur 40 bits (donc sur une plage d'un téra-octets).

Quatre niveaux de tables.- Pour le mode 64 bits, on utilise quatre niveaux de tables de pages. L'indicateur `PAE` (*Page Address Extension*) du registre `CR4` doit être positionné. On peut choisir, grâce aux indicateurs `PSE`, `PAE` et `PGE` de `CR4`, une taille de page de 4 KiO, 2 MiO, 4 MiO ou 4 GiO. Par défaut, c'est-à-dire si l'on ne positionne aucun de ces indicateurs, la taille d'une page reste de 4 KiO (comme pour le `80386`) et une adresse virtuelle a alors la structure suivante :



— Les bits 63 à 48 doivent avoir la même valeur que le bit 47 (on parle d'**adresse canonique** lorsque c'est bien le cas). Remarquons que pour les « petites » valeurs d'adresse que nous utiliserons, nous n'avons pas à nous en préoccuper : il s'agit de 0.

— Les 9 bits de 47 à 39 spécifient l'index dans la table de niveau 4 (appelée **PML4** pour *Page Map Level-4 Table*).

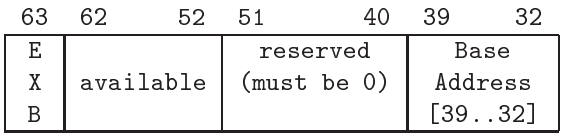
— Les 9 bits de 38 à 30 spécifient l'index dans la table de niveau 3 (appelée **PDPT** pour *Page Directory Pointer Table*, c'est-à-dire table de pointeur de répertoire).

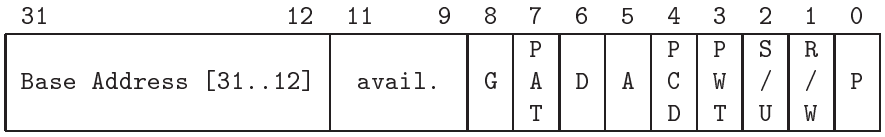
— Les 9 bits de 29 à 21 spécifient l'index dans la table de niveau 2 (on retrouve **PDIR**, pour *Page DIRectory*, répertoire de tables de pages, de l'architecture IA-32).

— Les 9 bits de 20 à 12 spécifient l'index dans la table de niveau 1 (on retrouve **PTBL** pour *Page TaBLe*, table de pages).

— Et enfin les 12 derniers bits spécifient le décalage dans la page, comme pour la pagination de IA-32.

Structure d'une entrée dans une table.- Les entrées dans les tables des différents niveaux, toutes de 64 bits (au lieu des 32 bits dans le cas de IA-32), ont la structure suivante :





Les 32 premiers bits reprennent la structure des entrées pour IA-32 :

— Le bit 0 (**P** pour *Present*) indique si la table de pages de niveau 1 à 3 ou la page est présente en mémoire vive. La table PML4, devant toujours être présente dans une zone de mémoire « identity mapped », ce bit n'a pas de signification pour elle.

— Le bit 1 (**R/W** pour *Read/Write*) spécifie les droits de lecture et d'écriture lorsque que le bit 2 vaut 1 : lorsqu'il est égal à 0, la table de pages ou la page peut être lue et écrite au niveau de privilège 3 ; lorsqu'il est égal à 1, elle ne peut qu'être lue à ce niveau.

— Le bit 2 (**U/S** pour *User/System*) spécifie les privilèges : lorsqu'il est égal à 0, la table de pages ou la page n'est accessible qu'aux niveaux de privilège 0 à 2.

— Les bit 3 (**PWT** pour *Page-level Write-Through*) et 4 (**PCD** pour *Page Caching Disable*) contrôlent la mise en cache et ne nous intéresseront pas dans une première étape.

— Le bit 5 (**A** pour *Accessed*) indique si on a eu accès à cette table de page ou à cette page (en lecture ou en écriture).

— Le bit 6 (**D** pour *Dirty*, c'est-à-dire *modifié*) est positionné par le microprocesseur chaque fois qu'une opération d'écriture est réalisée sur la table de page ou la page.

— Le bit 7 (**PAT** pour *Page-Attribute Table-index*) est nouveau.

— Le bit 8 (**G** pour *Global page*), nouveau également, est égal à 1 s'il s'agit d'une page globale.

13.1.8 Passage au mode long

13.1.8.1 Initialisation du mode long

Sur les microprocesseurs d'architecture x86-64, le registre spécifique `IA32_EFER` est mis à zéro lors de la réinitialisation. On doit se trouver en mode protégé avec pagination IA-32e, c'est-à-dire avec des structures de pagination à quatre niveau, activée avant d'essayer d'initialiser le mode long.

Pour initialiser le mode long, on doit :

— 1. Partir du mode protégé, désactiver (éventuellement) la pagination (du mode IA-32) en baissant le bit `PG` de `CR0` (`CR0.PG = 0`), en utilisant une instruction `MOV CR0`, évidemment dans une page mappée à l'identique.

— 2. Activer les extensions d'adresses physiques (PAE pour *Physical-Address Extensions*) en levant le bit `PAE` de `CR4` (`CR4.PAE = 1`), sinon une faute de protection générale est levée lorsqu'on essaie d'initialiser le mode long (ceci est également vrai pour les étapes 3 à 5).

— 3. Charger `CR3` avec l'adresse physique de base de la table de pages de niveau 4 (PML4).

Ces étapes 2 et 3 s'effectuent dans l'ordre que l'on veut, celui-ci n'ayant pas d'importance.

— 4. Activer le mode long en levant `IA32_EFER.LME` :

```
movl   0xC0000080,%ecx
rdmsr
orl    0x100,%eax
wrmsr
```

— 5. Activer la pagination (en mode IA-32e) en positionnant `CR0.PG = 1`. À ce moment, le microprocesseur positionne le bit `IA32_EFER.LMA` à 1.

— 6. Exécuter un saut lointain pour passer au sous-mode désiré, mode compatible ou mode 64 bits, la nature de celui-ci étant déterminé par le descripteur du code de segment. Le décalage de ce saut ne doit pas dépasser 32 bits.

— 7. C'est seulement alors que l'on peut, si on veut, se placer sur des pages non identiquement mappées.

Les tables de pagination du mode 64 bits doivent se trouver dans les 4 premiers GiO de la mémoire physique avant l'activation du mode long : puisque `MOV CR3` est exécuté en mode protégé, seuls les 32 bits de poids faible du registre sont écrits, ce qui limite l'emplacement des tables aux 4 premiers GiO de mémoire.

On peut déplacer les tables de pages où l'on veut une fois le mode long activé.

— 8. Après activation du mode long, les registres des tables de descripteurs système (GDTR, LDTR, IDTR, TR) continuent à faire référence aux tables de descripteurs du mode protégé. Les tables référencées par ces descripteurs résident toutes dans les 4 premiers GiO de l'espace d'adresse linéaire.

Après avoir activé le mode long, utiliser les instructions `LGDT`, `LLDT`, `LIDT` et `LTR` pour charger les registres de tables de descripteurs système pour référencer des tables de descripteurs 64 bits.

13.1.8.2 Retour au mode protégé IA-32

Pour revenir du mode long au mode protégé IA-32, il faut ([Int], §9.8.5.4) :

— 1. Passer au sous-mode compatible (si on se trouve dans le sous-mode 64 bits).

— 2. Désactiver la pagination IA-32e en mettant à zéro le bit `CR0.PG`. Ceci a pour effet de mettre le bit `IA32_EFER.LMA` à zéro. La plupart des instructions suivantes doivent évidemment se trouver dans une page identiquement mappée.

— 3. Charger (éventuellement) `CR3` avec l'adresse physique de base du répertoire de pagination IA-32.

— 4. Désactiver le mode long en mettant le bit `IA32_EFER.LME` à 0.

— 5. Préparer (éventuellement) la pagination en mode protégé en mettant le bit `CR0.PG` à 1.

— 6. Effectuer une instruction de saut après l'instruction `MOV CR0` pour activer la pagination.

Remarque.- On effectue en général des passages d'un mode à l'autre pour pouvoir utiliser les programmes 16 et 32 bits écrits antérieurement. Les contenus des registres seulement disponibles en mode 64 bits (`R8-R15` et `XMM8-XMM15`) sont préservés lors des transitions du mode 64 bits au mode compatible et retour. Par contre, les valeurs de `R8-R15` et de `XMM8-XMM15` sont indéfinies après transitions du mode 64 bits mode au modes compatible puis protégé ou au mode réel et retour.

13.2 Exemples

13.2.1 Utilisation des registres spécifiques à un modèle en mode protégé

Donnons un exemple d'utilisation d'un registre spécifique à un modèle. Afin qu'il ne soit pas « trop » spécifique, choisissons un de ceux présents sur tous les microprocesseurs à partir du *Pentium*.

Écrivons un programme permettant de lire le compteur de tops d'horloge `TSC` et d'en placer le contenu à l'emplacement `6000:0`. Puisque le programme n'effectue aucune vérification, il faut s'assurer par ailleurs que le PC utilisé ait au minimumun un *Pentium* comme microprocesseur :

```
#----------------------------;
# boot1.s       ;
# Example program reading TSC ;
#----------------------------;
.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
        movw    $DESC0,%bx
        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
#-----------------------------;
# A sample in protected mode: ;
#-----------------------------;
#
# Enable physical-address extensions
#
        movl    $0x20,%eax
        movl    %eax,%cr4
#
# MSR
#
        movl    $0x10,%ecx
        movl    $0,%eax
        movl    $0,%edx
        wrmsr
        movl    $0x12345678,%eax
        rdmsr
addr32  movl    %eax,%ds:(0x60000)
#-----------------------------------;
# 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
```

Assemblons ce fichier `boot1.s` sous Linux :

```
$ as boot1.s --32 -o ./boot1.o
$ objcopy -O binary ./boot1.o ./boot1.com
$ ls -l boot1.com
-rwxr-xr-x 1 patrick patrick 449 oct.  12 10:08 boot1.com
$
```

Il ne faut en garder que les 193 derniers octets du binaire :

```
$ objcopy -O binary -i 1500 --interleave-width 193 -b 256 boot1.o ./boot1.com
$
```

Exécutons sous MS-DOS :

```
C:\BOOT>boot1
C:\BOOT>debug
-D 6000:0 L 10
7000:0 D2 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-q
```

Exercice.- *Vérifier que si on n'active pas PAE, on a un redémarrage.*

13.2.2 Passage au mode compatible

Écrivons un programme qui passe au mode compatible, affiche 'A' à la seconde ligne de l'écran, puis revient au mode réel.

```
#-------------------------------------------------;
# boot2.s                                         ;
# Example program switching to compatibility mode ;
#-------------------------------------------------;
.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
        movw    $DESC0,%bx
        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
#----------------------------;
# A sample in protected mode: ;
#----------------------------;
#
# Clear the page tables
#
        movl    $0x70000,%edi
        movl    $0x1000,%ecx
        xorl    % eax,%eax
REPETER:
        movl    %eax,%es:(%edi)
        addl    $4,%edi
        decl    %ecx
        jecxz   FIN
        jmp     REPETER
FIN:
#
# First entry of PML4
#
addr32  movl    $0x71007,%ds:(0x70000)
#
# First entry of PDP table
#
addr32  movl    $0x72007,%ds:(0x71000)
#
# First entry of Directory
#
addr32  movl    $0x73007,%ds:(0x72000)
#
# Fill up 256 first page tables
#
        movl    $0x73000,%edi           # address of first page table
        movl    $7,%eax                 # content of first page table
        movl    $256,%ecx               # number of pages to map (1 MB)
 make_page_entries:
        movl    %eax,%es:(%edi)
        addl    $8,%edi                 # address next entry, 8 bytes
        addl    $0x1000,%eax            # content next entry
        decl    %ecx
        jecxz   FIN2
```

```
        jmp     make_page_entries
FIN2:
#
# Load page-map level-4 base (without enabling)
#
        movl    $0x70000,%eax
        movl    %eax,%cr3
#
# Enable long mode
#
        movl    $0xC0000080,%ecx         # EFER MSR
        rdmsr
        orl     $0x100,%eax              # enable long mode
        wrmsr
#
# Enable physical-address extensions
#
        movl     $0x20,%eax
        movl     %eax,%cr4
#
# Enable paging
#
        movl    %cr0,%eax
        orl     $0x80000000,%eax
        movl    %eax,%cr0               # enable paging
#
# Far jump to Comptaibility Mode (Selector 0x8)
#
        ljmp    $0x8,$suite
suite:
#
# Sample in Compatibility Mode
        movb    $65,%al
addr32  movb    %al,%ds:(0x0B80A0)
#
# Disable paging
#
        movl    %cr0,%eax
        btrl    $31,%eax
        movl    %eax,%cr0
#
# Disable long mode
#
        movl    $0xC0000080,%ecx        # EFER MSR
        rdmsr
        btrl    $8,%eax                 # enable long mode
        wrmsr
#-----------------------------------;
# 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 and long mode
#
DESC1:  .word   0x0FFFF                 # limit 64kB
DESC1B: .word   0                       # base 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 code segment in Long Mode
#
DESC4:  .word   0x0FFFF                 # limit 64kB
        .word   0                       # base = 0
        .byte   0                       # base
        .byte   0x9A                    # code segment
        .byte   0xAF
        .byte   0
#
# GDT table data
#
GDTL:   .word   0x27                    # limit
GDTA:   .long   0                       # base to set
```

Assemblons ce fichier `boot2.s` sous Linux :

```
$ as boot2.s --32 -o ./boot2.o
$ objcopy -O binary ./boot2.o ./boot2.com
$ ls -l boot2.com
-rwxr-xr-x 1 patrick patrick 605 oct.  13 10:26 boot2.com
$
```

Il ne faut en garder que les 349 derniers octets du binaire :

```
$ objcopy -O binary -i 1500 --interleave-width 349 -b 256 boot2.o ./boot2.com
$
```

Lorsqu'on exécute `boot2.com` sous MS-DOS, on voit apparaître un 'A' à la première ligne, une fois revenu au prompteur.

Remarques.- 1^o) On commence par se placer en mode protégé sans pagination, de la façon habituelle.

- 2^o) On prépare les structures pour la pagination :

— la PML4, qui ne contient qu'une seule entrée, est placée à l'adresse physique `0x70000`, multiple de `0x1000` ;

— la PDPT, qui ne contient également qu'une seule entrée, est placée à l'adresse physique `0x71000`, également multiple de `0x1000` ;

— La PDIR, qui ne contient aussi qu'une seule entrée, est placée à l'adresse physique `0x72000`, toujours multiple de `0x1000` ;

— La table de pages, qui contient 256 entrées de façon à « mapper » le premier MiO de mémoire vive, est placée à l'adresse physique `0x73000`, toujours multiple de `0x1000`.

- 3^o) Par précaution, on met à zéro les entrées qu'on n'initialisera pas. Elles ne devraient cependant pas être utilisées.

- 4^o) On remplit la première entrée des trois premiers niveaux de pagination ainsi que les 256 premières entrées de la table de page. L'adresse physique de PML4 est placée dans le registre `CR3`.

- 5^o) On active le mode long en mettant à 1 le bit 8 (`LME`) du MSR `EFER`.

- 6^o) On active la PAE en mettant à 1 le bit 6 du registre `CR4`.

- 7^o) On active la pagination IA-32e en mettant à 1 le bit 31 (`PGE`) de `CR0`.

- 8^o) Il faut alors effectuer un saut inter-segmentaire, par exemple à la ligne suivante. Utilisons le même code de segment que pour le mode protégé. On a `CS.L = CS.D = 0`, nous nous retrouvons donc avec un mode compatible en mode d'instructions 16 bits.

- 9^o) Affichons, ce qui est maintenant classique, 'A' au début de la deuxième ligne de l'écran.

- 10^o) Pour quitter le mode long, il faut d'abord désactiver la pagination IA-32e (sinon on aura un redémarrage lors de la prochaine action) puis mettre à zéro le bit 8 du MSR `EFER`.

- 11^o) On se retrouve alors en mode protégé, depuis lequel on sait revenir au mode réel.

13.2.3 Passage au mode 64 bits

Écrivons un programme passant au mode compatible avec instructions 32 bits, puis au mode 64 bits et exécutant le programme (en mode 64 bits) situé à l'adresse absolue `6000:0`.

```
#-----------------------------------------;
# boot3.s                          ;
# Example program switching to 64-bits mode ;
#-----------------------------------------;
.section .text
.globl   _start
.code16
.org     0x100
_start:
#
#Setting base for code segment in PM
#
        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
        movw    $DESC0,%bx
        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
#
```

```
.code32
pm_in:  movw    $0x10,%dx
        movw    %dx,%ds
        movw    %dx,%ss
        movw    %dx,%es
#----------------------------;
# A sample in protected mode: ;
#----------------------------;
#
# Clear the page tables
#
        movl    $0x70000,%edi
        movl    $0x1000,%ecx
        xorl    %eax,%eax
REPETER:
        movl    %eax,%es:(%edi)
        addl    $4,%edi
        decl    %ecx
        jecxz   FIN
        jmp     REPETER
FIN:
#
# First entry of PML4
#
        movl    $0x71007,%ds:(0x70000)
#
# First entry of PDP table
#
        movl    $0x72007,%ds:(0x71000)
#
# First entry of Directory
#
        movl    $0x73007,%ds:(0x72000)
#
# Fill up 256 first page tables
#
        movl    $0x73000,%edi           # address of first page table
        movl    $7,%eax                 # content of first page table
        movl    $256,%ecx               # number of pages to map (1 MB)
make_page_entries:
        movl    %eax,%es:(%edi)
        addl    $8,%edi                 # address next entry, 8 bytes
        addl    $0x1000,%eax            # content next entry
        decl    %ecx
        jecxz   FIN2
        jmp     make_page_entries
FIN2:
#
# Load page-map level-4 base (without enabling)
#
        movl    $0x70000,%eax
        movl    %eax,%cr3
#
# Enable long mode
```

```
#
        movl    $0xC0000080,%ecx        # EFER MSR
        rdmsr
        orl     $0x100,%eax             # enable long mode
        wrmsr
#
# Enable physical-address extensions
#
        movl     $0x20,%eax
        movl     %eax,%cr4
#
# Enable paging
#
        movl    %cr0,%eax
        orl     $0x80000000,%eax
        movl    %eax,%cr0               # enable paging
#
# Far jump to Compatibility Mode (Selector 0x8)
#
        ljmp    $0x8,$suite
suite:
#
# Far jump to 64-bit Mode
#
        .byte   0xea
        .long   0x60000
        .word   0x18
#-------------------------------------;
# End of the sample in protected mode ;
#-------------------------------------;
#
#-----------------------------------------;
#       GDT to be used in Protected Mode ;
#-----------------------------------------;
#
# null descriptor
#
DESC0:  .long   0                       # null descriptor
        .long   0
#
# descriptor for code segment in PM and long mode
#
DESC1:  .word   0x0FFFF                 # limit 64kB
DESC1B: .word   0                       # base to set
        .byte   0                       # base
DESC1C: .byte   0x9A                    # code segment
        .byte   0xC0                    # G = 0, D = 1
        .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 code segment in 64-bits Mode
#
DESC3:  .word   0                       # no limit
        .word   0                       # no base
        .byte   0                       # no base
        .byte   0x9A                    # code segment
        .byte   0x20                    # L = 1 (64-bit mode)
        .byte   0
#
# GDT table data
#
GDTL:   .word   0x1F                    # limit
GDTA:   .long   0                       # base to set
```

Assemblons ce fichier `boot3.s` sous Linux :

```
$ as boot3.s --32 -o ./boot3.o
$ objcopy -O binary ./boot3.o ./boot3.com
$ ls -l boot3.com
-rwxr-xr-x 1 patrick patrick 510 oct.  16 10:34 boot3.com
$
```

Il ne faut en garder que les 254 derniers octets du binaire :

```
$ objcopy -O binary -i 1500 --interleave-width 254 -b 256 boot3.o ./boot3.com
$
```

Remarques.- 1^o) Le descripteur DESC1 pour le segment de code en mode protégé a maintenant `0xC0` comme avant-dernier octet : `G = 1` mais surtout `D = 1` pour instructions 32 bits.

- 2^o) On n'a plus besoin du descripteur pour le retour au mode réel mais on a besoin d'un descripteur, DESC3, pour le segment de code en mode 64 bits : l'octet des droits d'accès est `0x9A` comme d'habitude pour un segment de code; l'avant-dernier octet est égal à `0x20`, c'est-à-dire que `L = 1`.

- 3^o) On ajoute la directive '`.code32`' lorsqu'on est parvenu dans le segment de code en mode protégé, puisque les instructions ne sont pas traduites en langage machine de la même façon suivant que l'on se trouve en mode d'instructions 16 bits ou 32 bits.

- 4^o) Pour passer du mode compatible au mode long, on effectue un saut long à l'adresse logique spécifiée par le sélecteur `0x18` (correspondant à DESC3) et le décalage `0x60000`.

Programme 64 bits.- Écrivons par ailleurs un programme 64 bits, disons `test.s`, plaçant, par exemple, 'LONG' à la deuxième ligne de l'écran puis gelant le programme :

```
.section .text
.globl   _start
.code64
.org     0x60000
_start:
        movq    $0x2047204E204F204C,%rax #'L O N G '
        movl    $0x0B80A0,%ebx
        movq    %rax,(%ebx)
here: jmp     here
```

Remarquons l'utilisation de la directive '`.code64`'.

Assemblons-le pour obtenir un programme binaire `test.bin`. Utilisons un éditeur de texte hexadécimal pour en observer le contenu :

```
48 B8 4C 20 4F 20 4E 20 47 20
48 89 05 8F 80 05 00
EB FE
```

— La première instruction commence par le préfixe REX `0x48` pour indiquer l'utilisation d'un registre de 64 bits. Le code opération `0xB8`, correspondant d'habitude à `mov ax,` suivi d'une constante sur deux octets, doit donc être traduit par `mov rax,` suivi d'une constante sur huit octets. Les huit octets qui suivent sont ceux des codes ASCII pour les caractères désirés.

— La seconde instruction commence également par le même préfixe REX. Le code opération, sur deux octets, `0x89 05`, signifiant d'habitude `mov [],ax`, signifie donc dans ce contexte `mov [],rax` avec un décalage (entre crochets) sur 32 bits. Les quatre octets suivants spécifient donc ce décalage, à savoir `0x0005808F`.

— La troisième instruction est un saut relatif court à l'adresse -2 par rapport à l'adresse de l'instruction qui la suit, c'est-à-dire qu'on revient à cette instruction, d'où une boucle infinie assurant le gel du système.

Plaçons `boot3.com` dans le répertoire BOOT de notre clé USB démarrant sur MS-DOS. Plaçons, à la main, le code du programme de test à l'emplacement `6000:0` et lançons `boot3.com` :

```
C:\BOOT>debug
-E 6000:0 48 B8 4C 20 4F 20 4E 20 47 20 48 89 05 8F 80 05 00 EB FE
-q

C:\BOOT>boot3
-
```

Le programme est gelé et on voit 'LONG' affiché à la deuxième ligne en noir sur fond vert.

13.2.4 Amélioration du passage au mode 64 bits

Ce n'est pas très convivial, mais ô combien instructif, d'avoir à écrire le programme 64 bits en code machine et à aller le placer à la main à l'emplacement mémoire `6000:0`. C'est en fait un peu plus facile lors de la conception d'un système d'exploitation car on ne part pas, et surtout on n'a pas l'intention de revenir, d'un système d'exploitation de lancement qui décide de l'endroit où placer le programme : on peut choisir certaines adresses de façon absolue.

Bien entendu, même lorsqu'on part de MS-DOS, on peut lancer le programme 64 bits automatiquement. C'est juste un peu plus compliqué car il faut sans cesse se préoccuper des adresses choisies par MS-DOS.

Réécrivons le programme précédent de façon à ce qu'on n'ait pas à aller placer le code machine à la main :

```
#-----------------------------------------;
# boot4.s                                 ;
# Example program switching to 64-bits mode ;
# and loading a sample                    ;
#-----------------------------------------;
.section .text
.globl   _start
.code16
.org     0x100
_start:
#
#Setting base for code segment in PM
#
        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
        movw    $DESC0,%bx
        shll    $4,%eax                 # deja fait mais prudence
        addl    %ebx,%eax
        movl    %eax,GDTA
#
# Address of beginning of program in 64-bits mode
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        movw    $suiteL,%bx
        shll    $4,%eax
        addl    %ebx,%eax
        movl    %eax,offL
#
# 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
#
        .code32
pm_in:  movw    $0x10,%dx
        movw    %dx,%ds
        movw    %dx,%ss
        movw    %dx,%es
#----------------------------;
# A sample in protected mode: ;
#----------------------------;
#
# Clear the page tables
#
        movl    $0x70000,%edi
        movl    $0x1000,%ecx
        xorl    %eax,%eax
REPETER:
        movl    %eax,%es:(%edi)
        addl    $4,%edi
        decl    %ecx
        jecxz   FIN
        jmp     REPETER
FIN:
#
# First entry of PML4
#
        movl    $0x71007,%ds:(0x70000)
#
# First entry of PDP table
#
        movl    $0x72007,%ds:(0x71000)
#
# First entry of Directory
#
        movl    $0x73007,%ds:(0x72000)
#
```

```
# Fill up 256 first page tables
#
        movl    $0x73000,%edi           # address of first page table
        movl    $7,%eax                 # content of first page table
        movl    $256,%ecx               # number of pages to map (1 MB)
make_page_entries:
        movl    %eax,%es:(%edi)
        addl    $8,%edi                 # address next entry, 8 bytes
        addl    $0x1000,%eax            # content next entry
        decl    %ecx
        jecxz   FIN2
        jmp     make_page_entries
FIN2:
#
# Load page-map level-4 base (without enabling)
#
        movl    $0x70000,%eax
        movl    %eax,%cr3
#
# Enable long mode
#
        movl    $0xC0000080,%ecx        # EFER MSR
        rdmsr
        orl     $0x100,%eax             # enable long mode
        wrmsr
#
# Enable physical-address extensions
#
        movl     $0x20,%eax
        movl     %eax,%cr4
#
# Enable paging
#
        movl    %cr0,%eax
        orl     $0x80000000,%eax
        movl    %eax,%cr0               # enable paging
#
# Far jump to Compatibility Mode (Selector 0x8)
#
        ljmp    $0x8,$suite
suite:
#
# Enable 64-bit Mode
#
        .byte   0xea
offL:   .long   0
        .word   0x18
#
# Sample in 64-bit Mode
#
        .code64
suiteL:
        movq    $0x2047204E204F204C,%rax #'L O N G '
        movl    $0x0B80A0,%ebx
```

```
        movq    %rax,(%ebx)
here: jmp     here
#
#-------------------------------------;
# End of the sample in protected mode ;
#-------------------------------------;
#
#-----------------------------------------------;
#       GDT to be used in Protected and long Mode ;
#-----------------------------------------------;
#
# null descriptor
#
DESC0:  .long   0                       # null descriptor
        .long   0
#
# descriptor for code segment in PM and long mode
#
DESC1:  .word   0x0FFFF                 # limit 64kB
DESC1B: .word   0                       # base to set
        .byte   0                       # base
DESC1C: .byte   0x9A                    # code segment
        .byte   0xC0                    # G = 0, D = 1
        .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 code segment in 64-bits Mode
#
DESC3:  .word   0                       # no limit
        .word   0                       # no base
        .byte   0                       # no base
        .byte   0x9A                    # code segment
        .byte   0x20                    # 64-bit mode
        .byte   0
#
# GDT table data
#
GDTL:   .word   0x1F                    # limit
GDTA:   .long   0                       # base to set
```

Assemblons ce fichier `boot4.s` sous Linux :

```
$ as boot4.s --32 -o ./boot4.o
$ objcopy -O binary ./boot4.o ./boot4.com
$ ls -l boot4.com
-rwxr-xr-x 1 patrick patrick 559 oct.  16 09:59 boot4.com
$
```

Il ne faut en garder que les 303 derniers octets du binaire :

```
$ objcopy -O binary -i 1500 --interleave-width 303 -b 256 boot4.o ./boot4.com
$
```

Lorsqu'on exécute `boot4.com` sous MS-DOS, on voit apparaître 'LONG' en noir sous fond vert à la deuxième ligne. Tout est gelé, ce qui est normal.

Commentaires.- 1^o) Lors du passage au mode 64 bits, nous avons toujours le même problème : le décalage doit être égal à l'adresse « absolue », c'est-à-dire à partir de 0. Appelons-là `OffL`. Ne sachant pas où MS-DOS va placer le programme, il faut l'initialiser, en mode réel, à partir de la valeur de `CS` choisie par MS-DOS.

- 2^o) Lorsque nous sommes en mode 64 bits, nous ne pouvons pas nous contenter de l'instruction :

```
movq %rax,(0xB80A0)
```

car l'adresse `0xB80A0` n'est pas traduite en code machine comme une adresse absolue mais comme un déplacement. D'où le recours à l'adressage indirect, en utilisant le registre `EBX`.

13.2.5 Passage au mode 64 bits et retour au mode réel

On peut partir du mode réel, passer au mode protégé, au mode long puis au mode 64 bits puis revenir, à l'inverse, en passant d'abord au mode de compatibilité, au mode protégé puis au mode réel, comme le motre l'exemple suivant.

```
#----------------------------------------;
# boot5.s                                ;
# Example program switching to 64-bit mode ;
# with return to real mode               ;
#----------------------------------------;
.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 segment in PM
#
        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
        movw    $DESC0,%bx
        shll    $4,%eax                 # deja fait mais prudence
        addl    %ebx,%eax
        movl    %eax,GDTA
#
# Address of beginning of program in 64-bits mode
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        movw    $suiteL,%bx
        shll    $4,%eax
        addl    %ebx,%eax
        movl    %eax,offL
#
```

```
# 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
#-----------------------------;
# A sample in protected mode: ;
#-----------------------------;
#
# Clear the page tables
#
        movl    $0x70000,%edi
        movl    $0x1000,%ecx
        xorl    %eax,%eax
REPETER:
        movl    %eax,%es:(%edi)
        addl    $4,%edi
        decl    %ecx
        jecxz   FIN
        jmp     REPETER
FIN:
#
# First entry of PML4
#
addr32  movl    $0x71007,%ds:(0x70000)
#
# First entry of PDP table
#
addr32  movl    $0x72007,%ds:(0x71000)
#
# First entry of Directory
#
addr32  movl    $0x73007,%ds:(0x72000)
```

```
#
# Fill up 256 first page tables
#
        movl    $0x73000,%edi           # address of first page table
        movl    $7,%eax                 # content of first page table
        movl    $256,%ecx               # number of pages to map (1 MB)
make_page_entries:
        movl    %eax,%es:(%edi)
        addl    $8,%edi                 # address next entry, 8 bytes
        addl    $0x1000,%eax            # content next entry
        decl    %ecx
        jecxz   FIN2
        jmp     make_page_entries
FIN2:
#
# Load page-map level-4 base (without enabling)
#
        movl    $0x70000,%eax
        movl    %eax,%cr3
#
# Enable long mode
#
        movl    $0xC0000080,%ecx        # EFER MSR
        rdmsr
        orl     $0x100,%eax             # enable long mode
        wrmsr
#
# Enable physical-address extensions
#
        movl     $0x20,%eax
        movl     %eax,%cr4
#
# Enable paging
#
        movl    %cr0,%eax
        orl     $0x80000000,%eax
        movl    %eax,%cr0               # enable paging
#
# Far jump to Compatibility Mode (Selector 0x8)
#
        ljmp    $0x8,$suite
suite:
#
# Enable 64-bit Mode
#
        .byte   0x66                    # 16-bit instruction mode
        .byte   0xea
offL:   .long   0
        .word   0x18
#
# Sample in 64-bit Mode
#
        .code64
suiteL:
```

```
        movq    $0x2047204E204F204C,%rax #'L O N G '
        movl    $0x0B80A0,%ebx
        movq    %rax,(%ebx)
#here: jmp     here
#
# Return to compatibility mode
#
# jmp [suiteC]
#
        .byte   0xFF
        .byte   0x2D
        .word   0
        .word   0
offC:   .long   suiteC
        .word   0x8
suiteC:
        .code16
        movb    $65,%al
addr32  movb    %al,%ds:(0x0B80A0)
#
# Disable paging
#
        movl    %cr0,%eax
        btrl    $31,%eax
        movl    %eax,%cr0
#
# Disable long mode
#
        movl $0xC0000080,%ecx       # EFER MSR
        rdmsr
        btrl    $8,%eax                # enable long mode
        wrmsr
#
# Sample in PM
#
        movb    $67,%al
addr32  movb    %al,%ds:(0x0B80A0)
#-------------------------------------;
# End of the sample in protected mode ;
#-------------------------------------;
#
# Load 64kB segment descriptor from GDT into DS for return
#
        movb    $0x20,%dl
        movw    %dx,%ds
        movb    $0x10,%dl
        movw    %dx,%es
#
# Return from PM to real mode
#
        andb    $0x0fe,%al
        movl    %eax,%cr0
#
# Sample
```

```
#
        movb    $69,%al             # 'E'
addr32  movb    %al,%es:(0x0B80A0)
#
# Far jump to restore CS & clear prefetch queue
#
        .byte   0x0ea
        .word   pm_out
seg:    .word   0
pm_out:
#
# Sample
#
        movb    $70,%al             # 'F'
addr32  movb    %al,%es:(0x0B80A0)
#
# 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 and long mode
#
DESC1:  .word   0x0FFFF                 # limit 64kB
DESC1B: .word   0                       # base to set
        .byte   0                       # base
DESC1C: .byte   0x9A                    # code segment
        .byte   0                       # G = 0, D = 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 code segment in 64-bits Mode
#
DESC3:  .word   0                       # no limit
        .word   0                       # no base
        .byte   0                       # no base
        .byte   0x9A                    # code segment
        .byte   0x20                    # 64-bit mode
        .byte   0
#
# descriptor for data segment return to real mode
#
DESC4:  .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   0x27                    # limit
GDTA:   .long   0                       # base to set
```

Assemblons ce fichier `boot5.s` sous Linux :

```
$ as boot5.s --32 -o ./boot5.o
$ objcopy -O binary ./boot5.o ./boot5.com
$ ls -l boot5.com
-rwxr-xr-x 1 patrick patrick 702 oct.  18 16:41 boot5.com
$
```

Il ne faut en garder que les 446 derniers octets du binaire :

```
$ objcopy -O binary -i 1500 --interleave-width 446 -b 256 boot5.o ./boot5.com
$
```

Lorsqu'on exécute `boot5.com` sous MS-DOS, on voit apparaître 'LONG' en noir sous fond vert à la première ligne, une fois revenu au prompteur. Cette fois-ci, le programme n'est pas gelé.

Commentaires.- 1^o) Pour revenir au mode d'instructions 16 bits en mode protégé et en mode compatible, on utilise le descripteur DESC1, avec `0x00` comme octet des droits d'accès et non plus `0xC0`.

Remarquez, par conséquent, l'ajout du préfixe `0x66` lors du passage au mode 64 bits par rapport au programme précédent.

- 2^o) Pour revenir au mode compatible, il faut, comme d'habitude, utiliser un saut lointain. Le saut avec adressage direct n'existe pas en mode 64 bits, comme on peut le constater en parcourant la description des instructions dans le manuel, AMD ou *Intel*. Il faut donc utiliser un saut lointain avec adressage indirect :

```
jmp [displacement]
```

Ces mêmes manuels indiquent que le code machine du saut lointain est `FF /5`, c'est-à-dire :

```
| 0xFF | Mod 101 R/M |
```

avec comme premier octet d'opcode `0xFF`, la seconde partie de l'opcode se trouve dans l'octet suivant `ModR/M` : le champ `REG` constitue en fait la seconde partie du code opération, égale à `101b`.

Choisissons le cas le plus simple, celui où le déplacement est donné par une constante (sur 32 bits). Il faut choisir `mod = 00` et `R/M = 101`. Ceci conduit au code opération :

```
| 0xFF | 00 101 101 |
```

soit `0xFF 0x2D`. Ce code opération doit être suivi du déplacement sur 32 bits. Si nous regardons notre programme, nous avons choisi :

```
FF 2D 00 00 00 00
```

soit un déplacement de 0.

En effet, en mode 64 bits, un déplacement est relatif au `RIP`, c'est-à-dire à l'adresse qui suit l'instruction. Avec 0, ceci signifie que le saut lointain aura lieu par rapport aux données (un déplacement sur 32 bits suivi d'un sélecteur sur 16 bits) suivant cette instruction.

```
#
# jmp [suiteC]
#
        .byte   0xFF
        .byte   0x2D
        .word   0
        .word   0
offC:   .long   suiteC
        .word   0x8
suiteC:
```

On y trouve ce que nous aurions aimé placer en immédiat, c'est-à-dire le décalage de l'étiquette qui suit, puis le sélecteur du code de segment pour les modes réel, protégé et compatible.

Nous devrions être placé, en mode compatible, à l'étiquette `suiteC`, ce que le bon déroulement du programme confirme.

Exercice.- Pour confirmer ce qui vient d'être dit, prenons un déplacement de 1 au lieu de 0 et déplaçons les données d'un octet :

```
#
# jmp [suiteC]
#
        .byte   0xFF
        .byte   0x2D
        .word   1         # displacement of 1
        .word   0
        .byte   0         # extra byte
offC:   .long   suiteC
        .word   0x8
suiteC:
```

avec le même comportement du programme.

13.3 Bibliographie

[AMD-02-1] Advanced Micro Devices, **AMD64 Architecture Programmer's Manual, volume 1 : Application Programming**, publication 24592, 2002, March 2012 for Revision 3.19, xxii + 352 p.

[AMD-02] Advanced Micro Devices, **AMD64 Architecture Programmer's Manual, volume 2 : System Programming**, publication 24593, 2002, September 2012 for Revision 3.22.

[AMD-02-3] Advanced Micro Devices, **AMD64 Architecture Programmer's Manual, volume 3 : General-Purpose and System Instructions**, publication 24594, 2002, September 2012 for Revision 3.19, xxxii + 538 p.

[Bre-09] Brey, Barry B., **The Intel Microprocessors : Architecture, Programming, and Interfacing : 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4, and Core2 with 64-bit Extensions**, Pearson, Eight Edition, 2009, xviii + 926 p.

[Dernière édition d'un classique publié depuis 1987, n'abordant alors que le `8086/8088`. Des couches sont ajoutées à chaque édition sans que le texte antérieur ne soit touché (ni corrigé). Comme la plupart des livres, il s'agit d'une adaptation de la documentation officielle d'*Intel*, dont [Int], moins complète mais avec une formulation plus pédagogique, utile pour commencer. Cette dernière édition débrousaille le mode 64 bits, mais sans aucun exemple de programme spécifique.]

[Ceg-04] Patrick Cégielski, **Conception des systèmes d'exploitation : Le cas linux. Deuxième édition**, Eyrolles, XIII + 680 p., septembre 2004.

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

[À la fois la documentation officielle d'*Intel* en plus de 3 000 pages et la référence. La plupart des livres reprennent une partie de celle-ci. Tous les sujets sont abordés mais il n'y a aucun exemple complet.]