Chapitre 9

Les interruptions en mode protégé

En mode protégé, les interruptions sont traitées d'une façon autre qu'en mode réel. Comme dans le mode réel, le processeur reconnaît 256 interruptions différentes et effectue le lien entre une interruption et le gestionnaire associé. Cependant, ce lien n'est plus réalisé par l'intermédiaire de la *table des vecteurs d'interruption*, qui occupait le premier kiO de la mémoire. L'analogue de cette table, la *table des descripteurs d'interruption*, peut désormais se trouver n'importe où dans la mémoire.

Ce changement est dû à la segmentation de la mémoire mise en place en mode protégé pour des raisons de protection. Sujet que nous n'avons pas encore abordé, mais qu'importe pour le moment !

9.1 Mise en place des interruptions en mode protégé

9.1.1 Les interruptions et exceptions de IA-32

On ne parle que d'*interruption* pour le `8086`. Pour l'architecture IA-32, on en distingue plusieurs types. Les *interruptions* et les *exceptions* sont des événements altèrant le déroulement normal d'un programme, afin de répondre immédiatement à une nouvelle situation, externe ou interne :

— Une **interruption** est déclenchée par un phénomène externe, asynchrone.

— Une **exception** provient du processeur lui-même, lorsque celui-ci détecte une situation spécifique exigeant un traitement immédiat au cours de l'exécution d'un programme.

Il existe deux catégories d'interruptions (comme pour le mode réel) et deux catégories d'exceptions :

Classe	**Catégorie**	**Signification**
Interruptions	Masquables	Arrivent sur la broche `INTR`
	Non masquables	Arrivent sur la broche `NMI`
Exceptions	Détectées par le processeur	3 sous-catégories : *fautes*, *pièges* et *avortement*
	Programmées	*Interruptions logicielles* : `INTO`, `INT 3`, `INT n` et `BOUND`

Les exceptions détectées par le processeur sont classées en trois sous-catégories :

— Les **fautes** (*fault* en anglais) sont les exceptions reconnues soit avant que l'instruction en étant à l'origine ne soit exécutée, soit pendant son exécution. En voici quelques exemples : tentative de violer une protection ; essai de dépassement des limites autorisées du segment de mémoire ; transfert de l'exécution dans un segment non exécutable ; essai d'écriture dans un segment en lecture uniquement.

— Les **pièges** (*trap* en anglais) sont les exceptions déclenchées après l'exécution de l'instruction. L'exemple typique est la division par zéro.

— L'**avortement** (*abort* en anglais) survient lorsque le microprocesseur n'arrive ni à connaître l'instruction qui en est à l'origine, ni à relancer le programme.

La **faute double** est une exception survenant lorsque le processeur est déjà en train de traiter une exception et qu'il reçoit une seconde exception. Seules peuvent intervenir dans le compte d'une faute double l'erreur de division (vecteur 0) et les exceptions concernant les segments, de vecteurs 10, 11, 12 et 13.

Intel réserve les 32 premièrs numéros d'interruption, dont les significations sont données dans le tableau suivant (où *Retour* signifie que l'adresse de retour pointe l'instruction fautive) :

Vecteur	**Cause**	**Classe/Type**	**Retour**
0	Erreur de division	Faute	Oui
1	Débogage	Piège	Oui
2	Interruption non masquable	Interruption	
3	Point d'arrêt (avec `INT 3`)	Piège	Non
4	Dépassement (avec `INTO`)	Piège	Non
5	Test de limite (`BOUND`)	Faute	Oui
6	Code opération non valide	Faute	Oui
7	Coprocesseur non disponible	Faute	Oui
8	Faute double	Avortement	Oui
9	Dépassement du segment coprocesseur	Avortement	Non
10	Segment d'état des tâches (TSS) non valide	Faute	Oui
11	Segment absent	Faute	Oui
12	Erreur sur la pile	Faute	Oui
13	Protection générale	Faute/Avortement	Oui
14	Faute sur page	Faute	Oui
15	(Réservée)		
16	Erreur du coprocesseur	Faute	Oui
17 à 31	(Réservées)		

L'ordre des priorités est résumé dans le tableau suivant, à partir des priorités de plus haut niveau, donc par ordre décroissant :

Fautes, exceptées celles de débogage Pièges `INTO`, `INT n` et `INT 3`
Pièges en débogage pour l'instruction en cours Fautes en débogage pour l'instruction suivante
Interruptions `NMI` Interruptions masquables

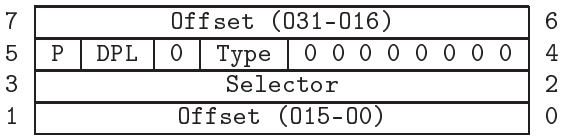
Enfin, les instructions pouvant être à l'origine de ces interruptions et exceptions sont les suivantes :

Vecteur	**Exception**	**Origine**
0	Erreur de division	`DIV`, `IDIV`
1	Débogage	Toutes
2	Interruption non masquable	`INT 2` ou broche `NMI`
3	Point d'arrêt	`INT 3`
4	Dépassement	`INTO`
5	Test de limites	`BOUND`
6	Code opération non valide	Tout code illégal
7	Coprocesseur non disponible	`ESC`, `WAIT`
9	Dépassement du segment coprocesseur	Le coprocesseur
10	Segment d'état des tâches (TSS) non valide	`JMP`, `CALL`, `IRET`, `INT`
11	Segment absent	Instructions sur registres de segment
12	Erreur sur pile	Références à la pile
13	Protection générale	Toute référence mémoire
14	Faute sur page	Tout accès mémoire ou recherche de code
16	Erreur du coprocesseur	`ESC`, `WAIT`

9.1.2 La table des descripteurs d'interruption

Lorsqu'une interruption est déclenchée, le processeur considère le numéro de l'interruption comme indice d'accès à la *table des descripteurs d'interruption* et y lit le *descripteur de porte* correspondant.

Structure d'un descripteur de porte.- Un **descripteur de porte** (*Gate Descriptor* en anglais), qui est un **descripteur système**, a une structure différente de celle d'un descripteur de segment, mais a toujours une taille de huit octets :



— Le sélecteur (octets 2 et 3) est celui du segment de code, se trouvant dans l'une des tables de descripteurs (la GDT ou la LDT), dans lequel se trouve le gestionnaire.

— Le décalage (octets 0, 1, 6 et 7), de 32 bits, spécifie le point d'entrée, dans ce segment de code, du gestionnaire d'interruption.

— L'octet 5 est l'octet de droits d'accès :

  — Le bit 7, noté P (pour *Present*), indique que le descripteur se trouve en mémoire (P = 1), c'est-à-dire qu'il contient une base et une limite valides, ou non (P = 0).

  — Les bits 5 et 6, notés DPL (pour *Descriptor Privilege Level*), spécifient le niveau de privilège du descripteur, compris entre 0 et 3.

  — Le bit 4 est toujours nul.

  — Les trois bits du type (bits 0 à 3) spécifient le type du descripteur de porte. Il est égal à `1110` pour une **porte d'interruption** IA-32, à `0100` pour une porte d'interruption `286` et à `01111` pour une **porte de piège**.

La différence entre les portes d'interruption et de piège réside dans la façon de traiter l'indicateur IF (d'annihilation des interruptions masquables). Il est mis à 0 ou à 1 avant le traitement de l'interruption suivant le cas :

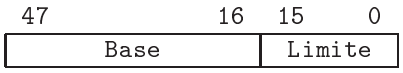
Type de porte	**Indicateur IF**
Interruption	Mis à 0. `IRET` le remettra à 1
Piège	Non modifié

Table des descripteurs d'interruption.- Les descripteurs d'interruption se placent dans la **table des descripteurs d'interruption** (**IDT** pour *Interrupt Descriptor Table*).

La table IDT étant limitée à 256 entrées, sa taille maximum est de 4 kiO.

On n'a accès (en écriture) à cette table qu'au niveau de privilège 0 en mode protégé. Contrairement à ce qui se passe en mode réel, une application n'est donc pas en mesure d'influencer l'exécution des interruptions. C'est là une condition importante pour assurer la stabilité de fonctionnement d'un système multitâches.

Registre de l'IDT.- Le système doit savoir où se trouve la table des descripteurs d'interruption. Il existe donc un **registre de la table des descripteurs d'interruption**, appelé **IDTR** pour *Interrupt Descriptor Table Register*, de 48 bits, contenant l'adresse de base, bits 16 à 47, de l'IDT et sa limite, bits 0 à 15.



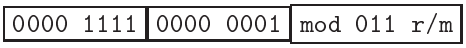
Chargement de l'IDT.- On charge la table des descripteurs d'interruption grâce à l'instruction :

```
LIDT S
```

(pour *Load the Interrupt Descriptor Table Register*), où `S` désigne l'emplacement mémoire du premier des 6 octets à placer dans le IDTR.

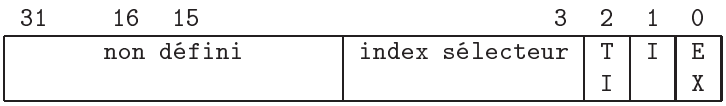
Cette instruction doit nécessairement être effectuée en mode réel avant de passer en mode protégé. Elle peut également être utilisée en mode protégé.

Son code machine est :



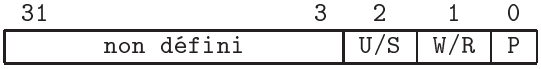
9.1.3 Code d'erreur

Les exceptions 8 et 10 à 14 engendrent et sauvegardent sur la pile un code d'erreur de 32 bits. Il est toujours égal à 0 pour l'exception 8. Pour les autres, son format est le suivant :



- Un sélecteur.
- Le bit EX (pour *EXtern*) est positionné à 1 par le processeur si une interruption externe au programme survient lors de la prise en compte de l'exception.
- Le bit I (ou bit de IDT) est positionné à 1 si l'index du code d'erreur concerne un descripteur de porte dans l'IDT.
- Le bit TI : lorsque le bit I est égal à 0, le bit TI spécifie si le code d'erreur se réfère à la GDT (TI = 0) ou à la LDT (TI = 1).

Dans le cas des erreurs de pagination (que nous étudierons en son temps), le code d'erreur engendré a le format suivant :



- Le bit U/S (pour *User/System*) spécifie si l'accès à l'origine de la faute est survenu alors que le processeur était en mode utilisateur (U/S = 1), c'est-à-dire au niveau de privilège 3, ou en mode système (U/S = 0), c'est-à-dire à l'un des niveaux de privilège 0 à 2.
- Le bit W/R (pour *Write/Read*) indique si l'accès s'est effectué en lecture (W/R = 0) ou en écriture (W/R = 1).
- Le bit P (pour *Page*) précise si la faute a été causée par une page absente (P = 0) ou par une tentative de violation du niveau de protection (P = 1).

9.1.4 Passage en mode protégé avec interruptions

Nous avons vu comment passer du mode réel au mode protégé lorsqu'on n'utilise pas d'interruption en mode protégé. Cependant les exceptions peuvent toujours survenir, ce qui conduit alors à un redémarrage du système. Si on veut tenir compte des interruptions, on a quelques étapes en plus à effectuer (en italique dans la description ci-dessous) :

— *Il faut initialiser la table* IDT *des descripteurs d'interruption pour qu'elle contienne des portes d'interruption valides pour au moins les 32 premiers numéros d'interruption.*

— Il faut initialiser la table GDT globale des descripteurs de façon à ce qu'elle contienne le descripteur nul comme descripteur de numéro 0 et des decripteurs valides pour au moins un segment de code et un segment des données.

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

— Charger ensuite IDTR de façon abrupte peut causer quelques problèmes. On sait à quoi sert IDTR en mode protégé ; en mode réel, il contient l'adresse de base de la table des vecteurs d'interruption. Entre le moment du chargement et le passage au mode protégé, on se retrouve donc en train d'utiliser une table de vecteurs d'interruption qui est inconsistante avec le mode d'opération en cours.

Il faut donc annihiler les interruptions matérielles (y compris `NMI`*) et les interruptions logicielles.*

— *On charge alors dans le registre* IDTR *l'adresse linéaire de base de l'*IDT *ainsi que sa limite.*

— On passe au mode protégé en mettant le bit `PE` de `CR0` à 1.

— 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 se placer dans la file des instructions internes.

— Les six registres de segment continuant à contenir les valeurs qu'ils avaient en mode réel, il faut 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` mais il suffit d'effectuer un saut long à l'instruction suivante.

On initialise ensuite les registres `SS` et `ESP` grâce à l'instruction `LSS`.

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

Retour au mode réel.- Lorsqu'on revient au mode réel, la table des interruptions (du mode protégé) est inconsistante avec ce mode. Il faut donc construire une table des vecteurs d'interruption et désactiver les interruptions lors de la transition entre les deux modes.

9.2 Un exemple

Écrivons un exemple de passage en mode protégé, avec retour au mode réel, mettant en place les 32 premières interruptions ainsi qu'une interruption logicielle. Nous nous inspirons ici de l'exemple `pm2.asm` de Christopher GIESE [Gie-98].

Écrivons un programme mettant en place les 33 premiers gestionnaires d'interruption : pour les 32 premiers, on se contente d'afficher '!' au coin supérieur droit de l'écran et de geler le système ; pour l'interruption logicielle `20h`, on affiche '20' sur la seconde ligne de l'écran. Le programme `int2.s` teste également l'interruption `20h` :

```
#-----------------------------------------;
# int2.s                                  ;
# Example showing Interrupt in Protected 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
        .byte   0x0bb
        .word   DESC0
        shll    $4,%eax                 # deja fait mais prudence
        addl    %ebx,%eax
        movl    %eax,GDTA
#
# Setting of IDTA
#
        movl    $0,%eax                 # pas deja fait
        movl    $0,%ebx
```

```
        movw    %cs,%ax                 # pas deja fait
        #movw   idt,%bx
        .byte   0x0bb
        .word   idt
        shll    $4,%eax                 # pas deja fait
        addl    %ebx,%eax
        movl    %eax,IDTA
#
# Make sure no ISR will interfere now
#
        cli
#
# LGDT is necessary before switch to PM
#
        lgdtl   GDTL
#
# LIDT is necessary before switch to PM
#
        lidtl   IDTL
#
# 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: ;
#-----------------------------;
#
# try an interrupt
#
        int     $0x20
#-------------------------------------;
# 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
#
# Point to real-mode IDTR
#
        lidtl   ridtr
#
# Exit to MS-DOS
#
        ret
#-----------------------------------------------------------------;
#       default handler for interrupts/exceptions                 ;
#       just puts '!' in upper right corner of screen and freezes ;
#-----------------------------------------------------------------;
unhand: cli
addr32  movb    $0x21,%ds:0x0B809E
here:   jmp     here
#---------------------------------------------------------;
#       handler for INT 0x20                              ;
#       prints '20' on second line of screen and returns ;
#---------------------------------------------------------;
isr20:
        movl    $0x0B80A0,%ebx
        movb    $0x32,%ds:(%ebx)
        movb    $0x30,%ds:2(%ebx)
        iretl
#
#--------------------------------------;
#       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
#
#---------------------------------------;
#       IDT to be used in Protected Mode ;
#---------------------------------------;
#
# 32 reserved interrupts:
#
idt:    .word   unhand                # entry point 15:0
        .word   0x8                   # selector
        .byte   0
        .byte   0x8E                  # type (32-bit Ring 0 interrupt gate)
        .word   0                     # entry point 31:16 (XXX - unhand >> 16)

        .word   unhand
        .word   0x8                   # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                   # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
```

```
.word   0x8                 # selector
.byte   0
.byte   0x8E
.word   0

.word   unhand
.word   0x8                 # selector
.byte   0
.byte   0x8E
.word   0

.word   unhand
.word   0x8                 # selector
.byte   0
.byte   0x8E
.word   0

.word   unhand
.word   0x8                 # selector
.byte   0
.byte   0x8E
.word   0

.word   unhand
.word   0x8                 # selector
.byte   0
.byte   0x8E
.word   0

.word   unhand
.word   0x8                 # selector
.byte   0
.byte   0x8E
.word   0

.word   unhand
.word   0x8                 # selector
.byte   0
.byte   0x8E
.word   0

.word   unhand
.word   0x8                 # selector
.byte   0
.byte   0x8E
.word   0

.word   unhand
.word   0x8                 # selector
.byte   0
.byte   0x8E
.word   0

.word   unhand
```

```
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
```

```
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
```

```
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

        .word   unhand
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0

#
# user interrupt handler
#
        .word   isr20
        .word   0x8                     # selector
        .byte   0
        .byte   0x8E
        .word   0
idt_end:
#
# IDT table data in Protected Mode
#
IDTL:   .word   0x107               # limit
IDTA:   .long   0                   # base to set
#
# an IDTR 'appropriate' for real mode
#
ridtr:  .word   0x120               # limit=0xFFFF
        .long   0                   # base=0
```

Assemblons ce fichier `int2.s` sous Linux :

```
$ as int2.s --32 -o ./int2.o
$ objcopy -O binary ./int2.o ./int2.com
$ ls -l int2.com
-rwxr-xr-x 1 patrick patrick 749 sept. 25 15:20 int2.com
$
```

Il ne faut garder que les 493 derniers octets du binaire :

```
$ objcopy -O binary -i 800 --interleave-width 493 -b 256 ./int2.o ./int2.com
$
```

On s'aperçoit alors que le programme s'exécute sous MS-DOS et que, lors du retour à MS-DOS, on voit '20' surchargeant la première ligne (et non la seconde puisqu'on a eu un retour chariot depuis qu'on est sorti du prpgramme).

Test des 32 premières interruptions.- Si on remplace `iretl` par `iret` dans `isr20`, on aura un problème au moment de récupérer `DS`. On devrait donc avoir un redémarrage du système. Avec nos gestionnaires d'interruption, on a '!' au coin supérieur droit et le système est gelé.

9.3 Bibliographie

[Gie-98] GIESE, Christopher, **Protected-Mode demo code**, July 1998. Code en ligne, initialement sur :

`http://www.execpc.com/~geezer/os`

que l'on trouve toujours en utilisant un moteur de recherche.

[Série de programmes écrits en langage d'assemblage `NASM` qui fonctionnent (il manque juste `-f bin` dans l'indication d'assemblage), ce qui est suffisamment rare pour être signalé.]