Chapitre 11

Les tâches

Nous connaissons, en tant qu'utilisateur, l'intérêt des systèmes d'exploitation multitâches : ils permettent, par exemple, d'imprimer un document pendant qu'on continue à travailler sur un autre. Comment sont-ils implémentés ? On introduit la notion de *tâche*, on découpe le temps en tranches de temps et on commute rapidement d'une tâche à une autre à chaque tranche de temps pour donner l'impression que toutes les tâches se déroulent en même temps. Ceci peut se faire de façon entièrement logicielle mais les processeurs modernes simplifient ce travail.

11.1 Les systèmes d'exploitation multitâches

Nous connaissons les *systèmes d'exploitation*, du point de vue utilisateur : il s'agit du logiciel qui prend la main au démarrage de l'ordinateur et qui simplifie la vie de l'utilisateur.L'évolution des systèmes d'exploitation pour les micro-ordinateurs ayant connu une évolution analogue à celle des ordinateurs, retraçons-en l'évolution pour les micro-ordinateurs.

Ordinateur sans système d'exploitation.- Les premiers micro-ordinateurs, tels que l'*Apple II* ou même l'IBM-PC, pouvaient être vendus sans système d'exploitation. Un tel ordinateur démarre sur un programme par défaut, implémenté en mémoire ROM, en général un interpréteur du langage de programmation BASIC. Pour passer à un autre programme, il faut redémarrer l'ordinateur avec une disquette dans le lecteur de disquette, contenant le code de celui-ci. Ce programme prévoit éventuellement la possibilité de sauvegarder les résultats dans un fichier sur une autre disquette pour pouvoir les utiliser avec un autre programme, en consultant cette disquette : cette disquette intermédiaire résout le problème du passage des données d'un programme à un autre.

Système d'exploitation monotâche.- Un système d'exploitation tel que MS-DOS (*MicroSoft Diskette Operating System* pour système d'exploitation des disquettes de Microsoft), le système d'exploitation des premiers IBM-PC, est *mono-utilisateur*, c'est-à-dire qu'une seule personne peut l'utiliser à un moment donné, ce qui n'est pas dérangeant pour un ordinateur *personnel* (PC voulant dire *Personal Computer*, rappelons-le !). Il est également *monotâche*, c'est-à-dire qu'une fois que l'ordinateur a démarré sur ce système d'exploitation, l'utilisateur spécifie à celui-ci quel programme il veut voir exécuter, *via* la ligne de commande. Il attend que ce programme se termine pour en lancer un autre.

C'est une avancée importante par rapport à l'étape précédente : on n'a pas à redémarrer l'ordinateur à chaque fois que l'on veut changer d'application. Un tel système d'exploitation présente cependant un inconvénient majeur : si on lance, par exemple, l'impression d'un (gros) document, il faut attendre que celle-ci soit terminée avant de pouvoir à nouveau accéder à celui-ci ou passer à autre chose.

Système d'exploitation multitâche.- Apple introduit en 1987 un système d'exploitation *multitâche*, qu'il appelle *multifinder*. On peut lancer une impression et continuer à travailler sur son document, ou à autre chose. Comment cela fonctionne-t-il ? Grâce au partage de temps (*Time-Sharing* en anglais, introduit au début des années 1960) : le système d'exploitation attribue des tranches de temps du microprocesseur à chacune de ces deux activités (ou même à plusieurs), si bien qu'on a l'impression que les deux tâches se déroulent simultanément et même, le plus souvent, sans ralentissement. En effet, en monotâche le microprocesseur connaît beaucoup de temps mort : il doit attendre qu'un caractère soit imprimé avant d'en transmettre un autre ou qu'on appuie sur une touche.

Système d'exploitation multi-utilisateur.- Une fois le multi-tâche introduit, on peut passer au système d'exploitation *multi-utilisateur*, tel que *Unix*, ou *Windows for Workgroups*. Plusieurs utilisateurs peuvent utiliser le même ordinateur en partageant certains fichiers. Il y a un seul utilisateur à un moment donné s'il n'y a qu'une seule console (clavier plus moniteur), mais il peut aussi y avoir autant d'utilisateurs simultanés que de consoles, sans parler des tâches qui s'effectuent en arrière-plan.

Conception des systèmes multitâches.- Dans un système d'exploitation multitâches, plusieurs programmes (appelés **tâches**, *task* en anglais) peuvent s'exécuter en parallèle.

Comment exécuter plusieurs tâches en parallèle lorsqu'on ne dispose que d'un seul processeur ? Nous l'avons expliqué ci-dessus avec la notion de **pseudo-parallélisme** : chaque tâche avance un petit peu lorsqu'on lui attribue une tranche de temps.

11.2 Les tâches et leur commutation sur Intel IA-32

Au vu de l'intérêt du multitâche, les concepteurs des microprocesseurs ont mis en place des fonctionnalités de celui-ci en facilitant la mise en place. C'est le cas pour le mode protégé des microprocesseurs *Intel*.

Lors du passage d'une tâche à l'autre (on parle de **commutation**), on sauvegarde l'*environnement* de la tâche en cours avant de passer à la tâche suivante. Cet environnement est sauvegardée dans des *segments d'état de tâche*. La commutation de tâche fait appel à un type spécifique de porte : une *porte de commutation de tâche* (*task gate* en anglais).

11.2.1 Segment d'état de tâche

Notion.- Les microprocesseurs modernes prévoient une zone mémoire pour sauvegarder les données d'une tâche lorsque celle-ci est mise en veille, c'est-à-dire qu'il ne s'agit pas de la tâche en cours.

Dans le cas des microprocesseurs *Intel*, depuis le `80286`, d'une façon revue lors de la conception du `80386`, les concepteurs en sont arrivés à la conclusion que le contexte (environnement) d'une tâche peut être déterminé par :

- le contenu des registres internes du processeur au moment où la tâche est mise en veille ;
- une LDT spécifique ;
- une structure éventuelle de pagination ;
- un pointeur de pile pour chacun des quatre niveaux de privilège ;
- un lien éventuel avec une autre tâche.

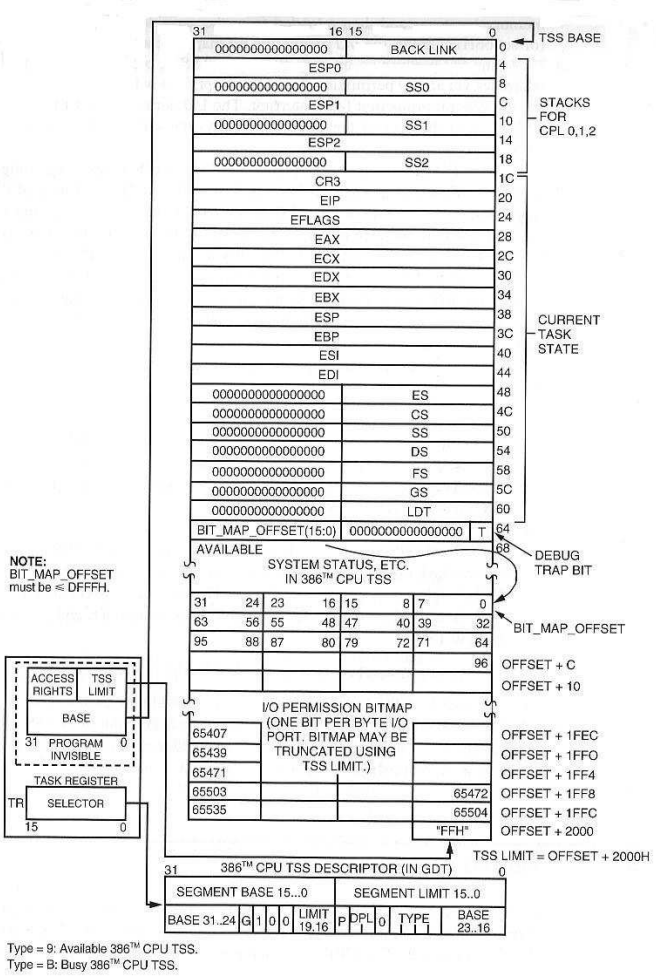
Ce contexte est décrit dans la mémoire par un segment, du type système, appelé **TSS** (pour *Task State Segment*, soit **segment d'état de tâche**).

Structure d'un segment de tâche.- Un TSS a une taille de 104 octets. Sa structure est représentée sur la figure 11.1 :

- Le premier mot, appelé **back-link**, est un sélecteur spécifant la tâche ayant fait appel à la tâche que l'on est en train de décrire, soit la **tâche parent**.

  Il est évidemment renseigné au moment de l'appel de cette tâche.

  Lorsque l'exécution de la tâche est complètement terminée, on revient à la tâche appelante : le contenu de *back-link* est à ce moment placé dans le registre TR, dont nous parlerons ci-dessous.
- Le mot suivant, réservé, contient 0.
- Les trois mots doubles suivants (2 à 7) contiennent les valeurs des registres `ESP` et `ESS` pour les trois niveaux de privilège 0 à 2. On en a besoin pour sauvegarder l'état de la pile au cas où la tâche en cours est interrompue à l'un de ces niveaux de privilège.
- Le huitième mot double (donc de décalage `0x1C`) contient une copie du registre `CR3`, stockant l'adresse de base du registre de répertoire de page de la tâche précédente.



The task state segment (TSS) descriptor. (Courtesy of Intel Corporation.)

FIGURE 11.1 – Structure du TSS

— Les dix-sept mots doubles suivants contiennent les valeurs des registres `EIP` à `EDI`, étendus par un mot nul dans le cas d'un registre de seize bits. À chaque fois que le commutateur de tâche donne la main à cette tâche, les registres sont initialisés à ces valeurs, conservées automatiquement lors de la mise en sommeil de la tâche.

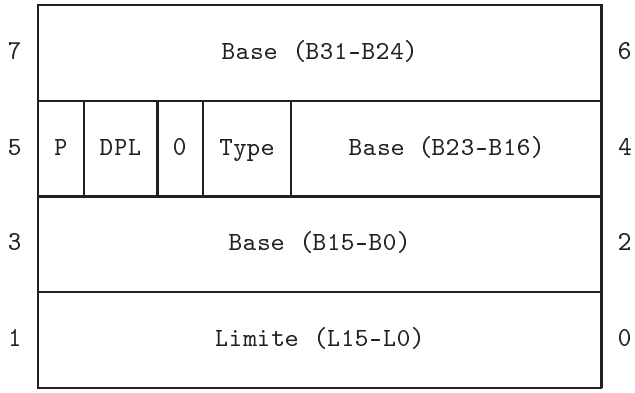
— Le mot suivant, donc de décalage `0x64`, contient le bit T (pour *debug Trap bit*) complété par des zéros.

— Le mot suivant, le dernier, celui de décalage `0x66`, contient l'adresse de base, appelée `BIT_MAP_OFFSET`, d'une structure spécifiant les droits d'accès aux 256 ports d'entrées-sorties. Ceci permet de bloquer les opérations d'entrées-sorties tout en levant une interruption de refus de permission des entrées-sorties : il s'agit de l'interruption 13, l'interruption de faute de protection générale.

Déporter cette structure des droits d'entrées-sorties permet à plusieurs TSS d'utiliser la même structure.

11.2.2 Descripteur de TSS

Comme tout segment, un segment de tâche est pointé par un descripteur, lui-même indexé par un sélecteur, dont la structure est la suivante :



— Puisqu'un TSS occupe 104 octets, le microprocesseur se contente de vérifier que la limite est supérieure à 103.

— L'adresse de base est celle du premier octet du TSS.

— Le bit P (pour l'anglais *Present*) spécifie si le segment est chargé en mémoire centrale (P = 1) ou ne se trouve que sur le disque (P = 0). Il sert à gérer la mémoire virtuelle.

— Les deux bits DPL (*Descriptor Privilege Level*), de niveau de privilège, spécifient le niveau de privilège maximum permettant d'accéder à ce descripteur (et donc à cette tâche).

— Les quatre bits de type pour un descripteur de TSS ont la signification suivante :

  — `0001b` pour un `TSS 80286` disponible,
  — `0011b` pour un `TSS 80286` occupé,
  — `1001b` pour un `TSS 80386` disponible,
  — `1011b` pour un `TSS 80386` occupé.

  Le bit 1 est appelé *Busy Flag* puisqu'il indique si la tâche est disponible (B = 0) ou est celle en cours (B = 1).

11.2.3 Commutation des tâches

L'instruction `LTR` permet d'initialiser la première tâche (appelée **tâche système**) puis les instructions `JMP` et `CALL` permettent d'appeler des tâches filles.

Tâche active.- La tâche active à un instant donné est pointée par le registre du processeur, de 16 bits, appelé **registre de tâche TR** (pour l'anglais *Task Register*) ou, de façon plus complète **TSSR** (pour l'anglais *Task State Segment Register*). Ce registre contient le sélecteur de la tâche en cours.

Initialisation du multitâche.- On peut ne pas utiliser le multitâche, ce que nous avons fait jusqu'à présent. Si on veut utiliser l'aide apportée par le microprocesseur, ce qui n'est possible qu'en mode protégé, il faut utiliser l'instruction **LTR** (pour *Load Task Register*) pour initialiser la première tâche, plus précisément pour charger son sélecteur.

Passage d'une tâche à une autre.- Deux phases interviennent pour passer d'une tâche à une autre :

— Il faut sauvegarder l'état en cours des registres du processeur. Cette sauvegarde s'effectue en mémoire, à partir de l'adresse pointée par le registre `TR`.

— Il faut modifier `TR` pour pointer sur la nouvelle tâche, donc le nouveau TSS, et appeler son contenu.

On peut utiliser à cet effet l'instruction `JMP`, l'instruction `CALL`, ou une interruption. Considérons d'abord le cas de `CALL` :

— On utilise un `CALL` lointain pour changer de tâche : le sélecteur est celui du descripteur de cette tâche ; le décalage est en général 0, c'est-à-dire qu'on ira à la valeur de `EIP` du TSS.

Lors de la commutation, le microprocesseur sauvegarde les valeurs adéquates dans le TSS de la tâche appelante et place le sélecteur de la tâche appelante dans le champ *Back-Link* de la tâche appelée.

— On quitte cette tâche par une instruction `IRET`.

Le passage à la tâche positionne le bit 14 (appelé `NT` pour*Nested Task*) du registre des indicateurs. Ceci sert à l'instruction `IRET` (en fait `IRETD` puisque nous sommes en mode protégé) pour lui indiquer qu'on quitte une tâche et non une interruption.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	VM	RF	0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

Le microprocesseur utilise le champ *Back-Link* pour charger le contenu du registre `LDTR`.

11.2.4 Un exemple

Écrivons un programme s'exécutant sous MS-DOS qui passe en mode protégé, initialise une première tâche (appelée *tâche système*, ce qu'elle fait est tout simplement la suite du programme) qui appelle une autre tâche (appelée *tâche utilisateur*), affichant 'B' à la deuxième colonne de la seconde ligne. On revient ensuite au mode réel puis à MS-DOS.

```
#---------------------------------;
# task1.asm                       ;
# Example program switching tasks ;
#---------------------------------;
.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 and data segments in PM
#
        movw    %cs,%ax
        movzx   %ax,%eax
        shll    $4,%eax                 # eax=base for code segment
        movl    %eax,DESC1B
        movl    %eax,DESC6B
        movb    $0x9a,DESC1C            # set segment attribute
        movb    $0x92,DESC6C
#
# Fix up TSS entries : STSS
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        shll    $4,%eax
        movw    $stss,%bx
        addl    %ebx,%eax               # EAX = linear address of stss
        movw    %ax,DESC4B
        shrl    $16,%eax
        movb    %al,DESC4C
        movb    %ah,DESC4D
#
# and UTSS
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
```

```
        shll    $4,%eax
        movw    $utss,%bx
        addl    %ebx,%eax               # EAX = linear address of utss
        movw    %ax,DESC5B
        shrl    $16,%eax
        movb    %al,DESC5C
        movb %ah,DESC5D
#
# Setting of GDTA
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        movw    $DESC0,%bx
        shll    $4,%eax
        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    $0x30,%dx
        movw    %dx,%ds
        movw    %dx,%ss
        movw    %dx,%fs
        movw    %dx,%gs
        movw    $0x10,%dx
        movw    %dx,%es
#----------------------------;
# A sample in protected mode  ;
#----------------------------;
# Load task register. All registers from this task will be dumped
# into DESC4 after executing the CALL USER_TSS:0 (and restored
# from DESC4 when the user subroutine does iret)
# *** NOTE! ***
```

```
# ltr in real mode causes an illegal instruction interrupt
#
        movw    $0x20,%ax               # SYS_TSS
        ltr     %ax
#
#
addr32  incw %es:(0x60000)
#
# Initialize user TSS
#
        movl    $user,%eax
        movl    %eax,(utss_eip)
        movl    %esp,%eax
        movl    %eax,(utss_esp)
#
# Call user task
#
        call    essai
        call    $0x28,$0             # USER_TSS:0
#
# Return from user task
#
addr32  incw    %es:(0x60008)
        movb    $66,%al
addr32  movb    %al,%es:(0x0B80A2)
#-------------------------------------;
# End of the sample in protected mode ;
#-------------------------------------;
#
# Load 64kB segment descriptor from GDT into DS for return
#
        movb    $0x18,%dl
        movw    %dx,%ds
        movw    %dx,%ss
#
# 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
#
# User task
#
essai:
addr32  incw    %es:(0x60006)
        movb    $67,%al
addr32  movb    %al,%es:(0x0B80A4)
        ret
user:
addr32  incw    %es:(0x60004)
        movb    $65,%al
addr32  movb    %al,%es:(0x0B80A0)
#
# the task switch set the NT bit, so iret does a return-from-task
#
        iretl
#
#---------------------------------------;
#       GDT to be used in Protected Mode ;
#---------------------------------------;
#
# null descriptor
#
DESC0:  .long   0                       # null descriptor
        .long   0
#
# descriptor for code segment in PM and real Mode
#
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 absolute 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
#
# system TSS
#
DESC4:  .word   103
DESC4B: .word   0                       # base to set at stss
DESC4C: .byte   0
        .byte   0x89                    # present, ring 0, 32-bit available TSS
        .byte   0
DESC4D: .byte   0
#
# user TSS
#
DESC5:  .word   103
DESC5B: .word   0                       # base to set at utss
DESC5C: .byte   0
        .byte   0x89                    # present, ring 0, 32-bit available TSS
        .byte   0
DESC5D: .byte   0
#
# descriptor for data segment in PM beginning as address given by MS-DOS
#
DESC6:  .word   0x0FFFF                 # limit 64kB
DESC6B: .word   0                       # base to set
        .byte   0                       # base
DESC6C: .byte   0x92                    # R/W segment
        .byte   0x0CF                   # G = 1, limit
        .byte   0                       # base
#
# GDT table data
#
GDTL:   .word   0x37                    # limit
GDTA:   .long   0                       # base to set
#
#       task state segments
#
stss:   .word   0, 0                    # back link
        .long   0                       # ESP0
        .word   0, 0                    # SS0, reserved
        .long   0                       # ESP1
        .word   0, 0                    # SS1, reserved
        .long   0                       # ESP2
        .word   0, 0                    # SS2, reserved
        .long   0, 0, 0                 # CR3, EIP, EFLAGS
        .long   0, 0, 0, 0              # EAX, ECX, EDX, EBX
        .long   0, 0, 0, 0              # ESP, EBP, ESI, EDI
        .word   0, 0                    # ES, reserved
        .word   0, 0                    # CS, reserved
        .word   0, 0                    # SS, reserved
        .word   0, 0                    # DS, reserved
        .word   0, 0                    # FS, reserved
        .word   0, 0                    # GS, reserved
        .word   0, 0                    # LDT, reserved
        .word   0, 0                    # debug, IO perm. bitmap
```

```
#
utss:   .word   0, 0                    # back link
        .long   0                       # ESP0
        .word   0, 0                    # SS0, reserved
        .long   0                       # ESP1
        .word   0, 0                    # SS1, reserved
        .long   0                       # ESP2
        .word   0, 0                    # SS2, reserved
        .long   0                       # CR3
utss_eip:
        .long   0, 0                    # EIP, EFLAGS (EFLAGS=0x200 for ints)
        .long   0, 0, 0, 0              # EAX, ECX, EDX, EBX
utss_esp:
        .long   0, 0, 0, 0              # ESP, EBP, ESI, EDI
        .word   0x10, 0                 # ES, reserved
        .word   0x8, 0                  # CS, reserved
        .word   0x30, 0                 # SS, reserved
        .word   0x30, 0                 # DS, reserved
        .word 0x30, 0                 # FS, reserved
        .word   0x30, 0                 # GS, reserved
        .word   0, 0                    # LDT, reserved
        .word   0, 0                    # debug, IO perm. bitmap
```

Assemblons ce fichier `task1.s` sous Linux :

```
$ as task1.s --32 -o ./task1.o
$ objcopy -O binary ./task1.o ./task1.com
$ ls -l task1.com
-rwxr-xr-x 1 patrick patrick 824 oct.   2 09:46 task1.com
```

Il ne faut en garder que les 568 derniers octets du binaire :

```
$ objcopy -O binary -i 900 --interleave-width 568 -b 256 ./task1.o ./task1.com
$
```

Après avoir exécuté ce programme sous MS-DOS, on voit apparaître 'ABC' au début de la première ligne, ce qui correspond à ce que nous voulions.

Commentaires.- 1^o) La GDT débute par les descripteurs habituels : le descripteur nul DESC0, le descripteur DESC1 de segment de code en mode protégé (utilisé également pour le retour au mode réel), le descripteur DESC2 de segment de données absolu, c'est-à-dire commençant à l'adresse 0, en mode protégé, et le descripteur DESC6 de segment de données en mode protégé commençant à l'adresse décidée par MS-DOS.

- 2^o) On déclare un TSS `stss` pour la tâche système. Il n'y a rien de spécial à initialiser.

- 3^o) On déclare un TSS `utss` pour la tâche utilisateur. La plupart des champs sont initialisés à 0, comme pour `stss`, mais d'autres doivent être renseignés :

— les champs registres de segment, ce que l'on peut faire de façon statique ;
— le champ `EIP` avec l'adresse à laquelle on commencera lorsqu'on fera appel à cette tâche.
Puisque nous ne savons pas à l'avance quelle en sera l'adresse absolue, décidée par MS-DOS, cette initialisation sera l'objet d'une partie du programme.
— le champ `ESP`, valeur du registre de pointeur de pile au moment d'accéder à la tâche.
Bien entendu ceci dépend de ce qui se passe dans le programme et sera donc également initialisé par une partie du programme.

- 4^o) On déclare un descripteur de segment DESC4 pour `stss` et un descripteur de segment DESC5 pour `utss` : la limite est 103 puisqu'un TSS occupe 104 octets ; l'adresse de base sera initialisée plus tard par une partie du programme ; l'octet des droits d'accès est `0x89` pour présent, `DPL = 0` et le type est égal à 9 pour 'TSS 386 disponible'.

- 5^o) La première partie nouvelle du programme consiste à initialiser les adresses de base des deux descripteurs de TSS.

- 6^o) La documentation, que ce soit celle d'*Intel* ou celle d'AMD, dit qu'après être entré dans le mode protégé, il faut initialiser les registres de segment et charger le registre des tâches, ce que nous n'avons pas fait jusqu'à présent.

Nous n'avons, en particulier, initialisé que les registres de segment dont nous avions besoin. Lorsqu'on utilise les tâches, même si on n'utilise pas explicitement certains registres de segment, il faut tous les initialiser. Vérifiez qu'il y a redémarrage du système si on ne le fait pas.

- 7^o) On charge le sélecteur de la tâche système dans le registre `TR` grâce à l'instruction `LTR`.

Comme nous l'avons déjà dit, rien de spécial ne se passe : le code exécuté est tout simplement ce qui suit cette instruction. On en a cependant besoin pour le retour depuis la tâche utilisateur.

- 8^o) On peut alors passer à la tâche utilisateur. Pour cela, il faut d'abord initialiser les champs `EIP` et `ESP` de son TSS. On peut le faire quand on veut : avant pour le champ `EIP`, évidemment juste avant pour le champ `ESP`.

L'appel de la tâche est un appel lointain dont les arguments sont : le sélecteur du descripteur de la tâche et le décalage 0.

- 9^o) Le code de la tâche utilisateur, commençant à l'étiquette `user`, se contente d'afficher 'A' au début de la deuxième ligne de l'écran.

On ne termine pas une tâche par une instruction `RET` mais par une instruction `IRETD`, comme pour une interruption. Rappelons que le code machine de ces instructions `IRET` et `IRETD` est le même, la différence se faisant suivant qu'on se trouve en mode réel ou en mode protégé.

Si on place l'instruction `here: jmp here` avant `IRETD` et qu'on enlève les initialisations des registres de segment `FS` et de `GS`, le programme se comporte de façon voulue : le 'A' est affiché et le programme est gelé. C'est seulement au moment de l'exécution de l'instruction `IRETD` qu'il y a redémarrage si on enlève les initialisations de `FS` et de `GS`.

- 10^o) Nous avons fait suivre l'appel de la tâche utilisateur par un affichage de 'B' pour bien montrer que le retour de la tâche (la commutation) s'effectue correctement. On termine le programme comme d'habitude.

11.3 Appel des tâches par un saut lointain ou une interruption

Nous venons de voir comment commuter d'une tâche à une autre avec `CALL`, c'est-à-dire en fait comment emboîter les tâches. Étudions maintenant les deux autres méthodes.

11.3.1 Principe

11.3.1.1 Saut lointain

Appel.- La tâche est définie par un descripteur de la GDT indexé par un sélecteur, disons `task`. L'appel de la tâche s'effectue par le saut lointain :

```
jmp task:0
```

permettant de spécifier à la fois le sélecteur et le décalage, mais c'est en fait le sélecteur qui est important.

En effet :

— Le décalage du code à exécuter la première fois que l'on fait appel à la tâche doit être renseigné dans le champ `EIP` du TSS de la tâche avant de la lancer.

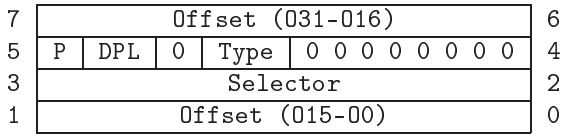
— Une tâche ne sera pas en général terminée en une tranche de temps. Lorsqu'on passe à une autre tâche, le microprocesseur place dans ce même champ `EIP` l'instruction suivante à exécuter (par rapport à l'emplacement du code où on quitte, momentanément, la tâche) : ceci permet de reprendre le déroulement de la tâche à l'endroit voulu lorsqu'on y reviendra.

Le bit B.- Nous avons vu que le type d'une tâche contient un bit (`B` pour *Busy*), positionné par le microprocesseur lorsque cette tâche est la tâche en cours. Il sert à éviter la récursivité, c'est-à-dire à appeler cette tâche à partir de cette tâche : appeler une tâche dont le bit `B` est positionné lève une exception.

Lorsqu'on fait appel à une autre tâche alors que la tâche en cours a été appelée par un saut lointain, le bit `B` n'est pas remis à zéro. Il faut donc le remettre à zéro si on veut qu'elle puisse être appelée à nouveau.

11.3.1.2 Interruption

Une tâche peut être appelée par une interruption (matérielle ou logicielle). Nous avons vu que, en mode protégé, l'interruption de numéro n fait appel au n-ième descripteur de porte de la table IDT. Rappelons la structure générale d'un descripteur de porte :

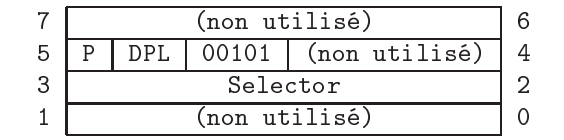


Il existe quatre types de descripteurs de porte :

Nom anglais	Nom français	Fonction
Call Gate	Porte d'appel	Changement de procédure
Task Gate	Porte de tâche	Commutation de tâche
Interrupt Gate	Porte d'interruption	
Trap Gate	Porte de piégeage	

Nous avons déjà étudié les portes d'interruption. Étudions ici le cas d'une porte de tâche.

La structure du descripteur se simplifie alors de la façon suivante :



c'est-à-dire que :

- le type est 5 (`0101b`) ;
- le décalage n'a pas d'intérêt pour la raison indiquée à propos du saut lointain (il est spécifié dans le TSS).

11.3.2 Un exemple

Écrivons un programme qui, partant du mode réel, passe au mode protégé puis à la tâche système. Celle-ci fait appel dix-huit fois à chacune des deux tâches utilisateurs : en fait deux fois suffisent pour illustrer ce que nous venons de dire.

La première tâche utilisateur affiche 'C' à la troisième colonne de la deuxième ligne la première fois qu'elle est appelée ('D' la seconde fois et ainsi de suite en incrémentant). Elle retourne à la tâche système grâce à une interruption (l'interruption 8).

La seconde tâche utilisateur est calquée sur la première, sauf qu'elle commence par afficher 'B' à la première colonne de la deuxième ligne.

```
#--------------------------------;
# task3.asm                     ;
# Example showing task switching  ;
# using far jump and interrupt    ;
#--------------------------------;
.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 and data segments in PM
#
        movw    %cs,%ax
        movzx   %ax,%eax
        shll    $4,%eax               # eax=base for code segment
        movl    %eax,DESC1B
        movl    %eax,DESC4B
        movb    $0x9a,DESC1C          # set segment attribute
        movb    $0x92,DESC4C
#
# Fix up TSS entries : STSS
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        shll    $4,%eax
        movw    $stss,%bx
        addl    %ebx,%eax             # EAX = linear address of stss
        movw    %ax,DESC5B
        shrl    $16,%eax
        movb    %al,DESC5C
        movb    %ah,DESC5D
```

```
#
# UTSS1
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        shll    $4,%eax
        movw    $utss1,%bx
        addl    %ebx,%eax               # EAX = linear address of utss
        movw    %ax,DESC6B
        shrl    $16,%eax
        movb    %al,DESC6C
        movb %ah,DESC6D
#
# and UTSS2
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        shll    $4,%eax
        movw    $utss2,%bx
        addl    %ebx,%eax               # EAX = linear address of utss
        movw    %ax,DESC7B
        shrl    $16,%eax
        movb    %al,DESC7C
        movb    %ah,DESC7D
#
# Setting of GDTA
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        movw    $DESC0,%bx
        shll    $4,%eax
        addl    %ebx,%eax
        movl    %eax,GDTA
#
# Setting of IDTA
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        movw    $idt,%bx
        shll    $4,%eax
        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
        movw    %dx,%fs
        movw    %dx,%gs
        movw    %dx,%es
#----------------------------;
# A sample in protected mode  ;
#----------------------------;
# Load task register. All registers from this task will be dumped
# into DESC4 after executing the CALL USER_TSS:0 (and restored
# from DESC4 when the user subroutine does iret)
# *** NOTE! ***
# ltr in real mode causes an illegal instruction interrupt
#
        movw    $0x28,%ax               # SYS_TSS
        ltr     %ax
#
#
        movw    $0x20,%dx
        movw    %dx,%ds
#
# Initialize user TSS 1
#
        movl    $user1,%eax
        movl    %eax,(utss1_eip)
        movl    %esp,%eax
        movl    %eax,(utss1_esp)
#
# Initialize user TSS 2
#
        movl    $user2,%eax
        movl    %eax,(utss2_eip)
        movl    %esp,%eax
        movl    %eax,(utss2_esp)
```

```
#
        movb    $0x42,%bl
addr32  movb    %bl,%es:0xb80a2
#
# 18 times...
#
        movl    $18,%ecx
sched:
        ljmp    $0x30,$0            # USER1_TSS:0
#
# clear busy bit of user1 task
#
        movb    $0x89,DESC6E
#
        ljmp    $0x38,$0            # USER2_TSS:0
# clear busy bit of user2 task
        movb    $0x89,DESC7E
#
        loop    sched
#-----------------------------------;
# End of the sample in protected mode ;
#-----------------------------------;
#
# Load 64kB segment descriptor from GDT into DS for return
#
        movb    $0x18,%dl
        movw    %dx,%ds
        movw    %dx,%ss
#
# 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
#------------;
# User tasks ;
#------------;
# task 1
#
user1:
        movb    $66,%bl
suite1: incb    %bl
addr32  movb    %bl,%es:(0x0B80A4)
        movl    $0xFF,%ecx          # 0FFFFh
        int     $8
        jmp     suite1              # infinite loop (until timer interrupt)
#
# task 2
#
user2:
        movb    $64,%bl
suite2: incb    %bl
addr32  movb    %bl,%es:(0x0B80A0)
        movl    $0xFF,%ecx          # 0FFFFh
        int     $8
        jmp     suite2              # infinite loop (until timer interrupt)
#-----------------------------------------------------------------;
#       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
#
#--------------------------------------;
#       GDT to be used in Protected Mode ;
#--------------------------------------;
#
# null descriptor
#
DESC0:  .long   0                   # null descriptor
        .long   0
#
# descriptor for code segment in PM and real Mode
#
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 absolute 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 in PM beginning as address given by MS-DOS
#
DESC4:  .word   0x0FFFF                 # limit 64kB
DESC4B: .word   0                       # base to set
        .byte   0                       # base
DESC4C: .byte   0x92                    # R/W segment
        .byte   0x0CF                   # G = 1, limit
        .byte   0                       # base
#
#
# system TSS
#
DESC5:  .word   103
DESC5B: .word   0                       # base to set at stss
DESC5C: .byte   0
        .byte   0x89                    # present, ring 0, 32-bit available TSS
        .byte   0
DESC5D: .byte   0
#
# user TSS 1
#
DESC6:  .word   103
DESC6B: .word   0                       # base to set at utss
DESC6C: .byte   0
DESC6E: .byte   0x89                    # present, ring 0, 32-bit available TSS
        .byte   0
DESC6D: .byte   0
#
# user TSS 2
#
DESC7:  .word   103
DESC7B: .word   0                       # base to set at utss
DESC7C: .byte   0
DESC7E: .byte   0x89                    # present, ring 0, 32-bit available TSS
        .byte   0
DESC7D: .byte   0
#
# GDT table data
#
GDTL:   .word   0x3F                    # limit
GDTA:   .long   0                       # base to set
```

```
#--------------------------;
#       task state segments ;
#--------------------------;
# System TSS
#
stss:   .word   0, 0                    # back link
        .long   0                       # ESP0
        .word   0, 0                    # SS0, reserved
        .long   0                       # ESP1
        .word   0, 0                    # SS1, reserved
        .long   0                       # ESP2
        .word   0, 0                    # SS2, reserved
        .long   0, 0, 0                 # CR3, EIP, EFLAGS
        .long   0, 0, 0, 0              # EAX, ECX, EDX, EBX
        .long   0, 0, 0, 0              # ESP, EBP, ESI, EDI
        .word   0, 0                    # ES, reserved
        .word   0, 0                    # CS, reserved
        .word   0, 0                    # SS, reserved
        .word   0, 0                    # DS, reserved
        .word   0, 0                    # FS, reserved
        .word   0, 0                    # GS, reserved
        .word   0, 0                    # LDT, reserved
        .word   0, 0                    # debug, IO perm. bitmap
#
# User TSS 1
#
utss1:  .word   0, 0                    # back link
        .long   0                       # ESP0
        .word   0, 0                    # SS0, reserved
        .long   0                       # ESP1
        .word   0, 0                    # SS1, reserved
        .long   0                       # ESP2
        .word   0, 0                    # SS2, reserved
        .long   0                       # CR3
utss1_eip:
        .long   0, 0                    # EIP, EFLAGS (EFLAGS=0x200 for ints)
        .long   0, 0, 0, 0              # EAX, ECX, EDX, EBX
utss1_esp:
        .long   0, 0, 0, 0              # ESP, EBP, ESI, EDI
        .word   0x10, 0                 # ES, reserved
        .word   0x8, 0                  # CS, reserved
        .word   0x10, 0                 # SS, reserved
        .word   0x20, 0                 # DS, reserved
        .word 0x10, 0                  # FS, reserved
        .word   0x10, 0                 # GS, reserved
        .word   0, 0                    # LDT, reserved
        .word   0, 0                    # debug, IO perm. bitmap
#
# User TSS 2
#
utss2:  .word   0, 0                    # back link
        .long   0                       # ESP0
        .word   0, 0                    # SS0, reserved
        .long   0                       # ESP1
```

```
        .word   0, 0                    # SS1, reserved
        .long   0                       # ESP2
        .word   0, 0                    # SS2, reserved
        .long   0                       # CR3
utss2_eip:
        .long   0, 0                    # EIP, EFLAGS (EFLAGS=0x200 for ints)
        .long   0, 0, 0, 0              # EAX, ECX, EDX, EBX
utss2_esp:
        .long   0, 0, 0, 0              # ESP, EBP, ESI, EDI
        .word   0x10, 0                 # ES, reserved
        .word   0x8, 0                  # CS, reserved
        .word   0x10, 0                 # SS, reserved
        .word   0x20, 0                 # DS, reserved
        .word 0x10, 0                 # FS, reserved
        .word   0x10, 0                 # GS, reserved
        .word   0, 0                    # LDT, reserved
        .word   0, 0                    # debug, IO perm. bitmap
#
#---------------------------------------;
#       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
# INT 8 is IRQ0 (timer interrupt). The 8259's can (and should) be
# reprogrammed to assign the IRQs to higher INTs, since the first
# 32 INTs are Intel-reserved. Didn't IBM or Microsoft RTFM?
        .word   0
        .word   0x28                    # SYS_TSS
        .byte   0
        .byte   0x85                    # Ring 0 task gate
        .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 `task3.s` sous Linux :

```
$ as task3.s --32 -o ./task3.o
$ objcopy -O binary ./task3.o ./task3.com
$ ls -l task3.com
-rwxr-xr-x 1 patrick patrick 1332 oct.   5 09:56 task3.com
```

Il ne faut en garder que les 1 076 derniers octets du binaire :

```
$ objcopy -O binary -i 1500 --interleave-width 1076 -b 256 task3.o ./task3.com
$
```

Après avoir exécuté ce programme sous MS-DOS, on voit apparaître 'RBT' au début de la première ligne, ce qui correspond à ce que nous voulions.

Commentaires.- 1^o) Outre les descripteurs habituels, la GDT comprend un descripteur pour la tâche système et un descripteur pour chacune des deux tâches utilisateur.

- 2^o) On déclare des TSS pour la tâche système, `stss`, et pour chacune des deux tâches utilisateur, `utss1` et `utss2`.

- 3^o) Puisqu'on va utiliser une interruption, on déclare une table `idt` des descripteurs d'interruption pour les 32 premières interruptions. Les gestionnaires d'interruption seront tous le gestionnaire générique sauf pour l'interruption 8. Dans ce dernier cas, on a une porte de tâche correspondant à la tâche système.

Nous verrons dans la section suivante pourquoi nous avons choisi l'interruption 8 et non, de façon plus naturelle, l'interruption 32.

- 4^o) Le code commence comme le code de l'exemple précédent. Bien entendu, on initialise les adresses de base de chacune des deux tâches système et non plus d'une seule.

- 5^o) Puisqu'on utilise des interruptions il faut, comme nous l'avons vu dans le chapitre consacré aux interruptions en mode protégé, initialiser l'adresse de base de l'IDT et charger le registre `IDTR`, sans oublier de passer à l'IDT adéquat avant de revenir au mode réel.

- 6^o) Une fois en mode protégé avec lancement de la tâche système, il faut initialiser les deux tâches utilisateur.

- 7^o) On fait alors appel dix-huit fois à chacune des tâches utilisateur grâce à une boucle `LOOP`. Il ne faut pas oublier d'affacer le bit `B` à chaque retour d'une des tâches utilisateur.

- 8^o) Le code de chacune des taches utilisateur est simple : on affiche ce qui est voulu puis on fait appel à l'interruption 8. Lorsqu'on reviendra dans la tâche utilisateur, on exécutera l'instruction suivante, c'est-à-dire le saut à l'étiquette qui permet d'incrémenter le caractère à afficher.

11.4 Principe de la mise en place du pseudo-parallélisme

Principe.- Un microproceseur tel que le `80386` d'*Intel* facilite la mise en place du multitâche : il permet de conserver l'environnement d'une tâche (grâce à un TSS) et la commutation d'une tâche à une autre. Il ne permet pas cependant de lancer plusieurs tâches puis de les gérer automatiquement.

C'est au système d'exploitation qu'il appartient de gérer, de façon logicielle et non matérielle, plusieurs tâches en même temps. Nous avons vu le principe du pseudo-parallélisme. Voyons comment le mettre en place.

Le pseudo-parallélisme est pris en charge par la première tâche : la tâche système, celle lancée par l'instruction `LTR`. On lance alors les tâches « utilisateur » l'une après l'autre dans le code de la tâche système.

Si on ne fait rien de plus, la première tâche « utilisateur » est exécutée puis la seconde et ainsi de suite. Avant de lancer la première tâche, on lance donc un minuteur (nécessairement externe puisque nous ne disposons pas du multitâche pour l'instant). On le conçoit de façon à ce que, une fois arrivé à zéro, il lève une interruption (nécessairement matérielle puisqu'il est externe). Nous avons vu, dans la section précédente, comment une interruption peut appeler une tâche. Si cette tâche est la tâche système et que l'exception est levée alors que, par exemple, la première tâche utilisateur est en train d'être exécutée, on se retrouve à l'instruction suivant l'instruction ayant lancée la première tâche. Si cette instruction est le lancement de la seconde tâche « utilisateur » on exécutera, de même, un petit bout de code de la seconde tâche.

On voit donc comment faire exécuter un petit bout de chaque tâche. Mais il faut bien que les tâches finissent par être complètement exécutées. On utilise des stratégies de choix des tâches à reprendre pour cela.

Si on a deux tâches utilisateur, par exemple, il suffit que le code de la tâche système soit une boucle (infinie) dont le corps consiste à appeler une tâche puis l'autre.

Exemple.- Considérons le cas de deux tâches utilisateur : la première affiche les caractères à partir de 'B' à la troisième colonne de la seconde ligne ; la seconde affiche les caractères à partir de 'A' à la première colonne de la seconde ligne.

Par compatibilité avec le premier IBM-PC, tout micro-ordinateur PC est muni, outre d'un microprocesseur, d'un circuit intégré annexe appelé PIC (pour *Programmable Interrupt Controller*) permettant de gérer les interruptions externes (par exemple décider ce qu'il faut faire lorsque deux interruptions externes apparaissent en même temps). Le PIC utilisé dans tous les PC qui ont suivi le premier reprend au minimum les fonctionnalités du 8259 du PC d'origine (pour raison de compatibilité). Une fois initialisé, ce qui est effectué par le BIOS au moment du démarrage, le PIC est contrôlé par les deux ports d'adresses `0x20` et `0x21` : le port `0x21` permet de spécifier les interruptions matérielles à inhiber, IRQ0 à IRQ15. Toujours par compatibilité avec le premier PC, IRQ0 correspond à un minuteur : on le lance en envoyant `0x20` sur le port `0x20` ; lorsqu'il est arrivé à zéro, il lève l'interruption 8. L'interruption 8 était déjà « réservée » par *Intel* lors de la conception du premier PC mais alors non utilisée.

Utilisons ce minuteur, présent sur tous les PC, pour gérer le multi-tâche.

```
#---------------------------------;
# task4.asm                       ;
# Example showing two ring 0 tasks ;
# multitasked via timer interrupt  ;
#---------------------------------;
.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 and data segments in PM
#
        movw    %cs,%ax
        movzx   %ax,%eax
        shll    $4,%eax                 # eax=base for code segment
        movl    %eax,DESC1B
        movl    %eax,DESC4B
        movb    $0x9a,DESC1C            # set segment attribute
        movb    $0x92,DESC4C
#
# Fix up TSS entries : STSS
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        movzx   %ax,%eax #
        shll    $4,%eax
        movw    $stss,%bx
        addl    %ebx,%eax               # EAX = linear address of stss
        movl    %eax,DESC5B
        movb    $0x89,DESC5E
#
# UTSS1
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        shll    $4,%eax
        movw    $utss1,%bx
        addl    %ebx,%eax               # EAX = linear address of utss
        movw    %ax,DESC6B
        shrl    $16,%eax
        movb    %al,DESC6C
```

```
        movb    %ah,DESC6D
#
# and UTSS2
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        shll    $4,%eax
        movw    $utss2,%bx
        addl    %ebx,%eax               # EAX = linear address of utss
        movw    %ax,DESC7B
        shrl    $16,%eax
        movb    %al,DESC7C
        movb    %ah,DESC7D
#
# Setting of GDTA
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        movw    $DESC0,%bx
        shll    $4,%eax
        addl    %ebx,%eax
        movl    %eax,GDTA
#
# Setting of IDTA
#
        movl    $0,%eax
        movl    $0,%ebx
        movw    %cs,%ax
        movw    $idt,%bx
        shll    $4,%eax
        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
        movw    %dx,%fs
        movw    %dx,%gs
        movw    %dx,%es
#----------------------------;
# A sample in protected mode  ;
#----------------------------;
# Load task register. All registers from this task will be dumped
# into DESC4 after executing the CALL USER_TSS:0 (and restored
# from DESC4 when the user subroutine does iret)
# *** NOTE! ***
# ltr in real mode causes an illegal instruction interrupt
#
        movw    $0x28,%ax               # SYS_TSS
        ltr     %ax
#
#
        movw    $0x20,%dx
        movw    %dx,%ds
#
# Initialize user TSS 1
#
        movl    $user1,%eax
        movl    %eax,(utss1_eip)
        movl    %esp,%eax
        subl    $512,%eax
        movl    %eax,(utss1_esp)
#
# Initialize user TSS 2
#
        movl    $user2,%eax
        movl    %eax,(utss2_eip)
        movl    %esp,%eax
        subl    $1024,%eax
        movl    %eax,(utss2_esp)
#
# shut off interrupts at the 8259 PIC, except for timer interrupt.
# The switch to user task will enable interrupts at the CPU.
#
        movb    $0xFE,%al
        out     %al,$0x21
#
# Displaying
#
        movb    $65,%bl
```

```
addr32  movb    %bl,%es:0xb80a2
#
# 18 times...
#
        movl    $18,%ecx
        movb    $0x20,%al
sched:
#
# Running user1 task
#
        ljmp    $0x30,$0             # USER1_TSS:0
#
# Timer interrupt returns us here
# clear busy bit of user1 task
#
        movw    $0x20,%dx
        movw    %dx,%ds
        movb    $0x89,DESC6E
#
# Reset 8259 PIC
#
        movb    $0x20,%al
        out     %al,$0x20
#
# Running user2 task
#
        ljmp    $0x38,$0             # USER2_TSS:0
#
# Timer interrupt returns us here
# clear busy bit of user2 task
#
        movw    $0x20,%dx
        movw    %dx,%ds
        movb    $0x89,DESC7E
#
#
# Reset 8259 PIC
#
        movb    $0x20,%al
        out     %al,$0x20           # send the EOI to the PIC
#
# End of loop
#
        loop    sched
#-------------------------------------;
# End of the sample in protected mode ;
#-------------------------------------;
#
# Load 64kB segment descriptor from GDT into DS for return
#
        movb    $0x18,%dl
        movw    %dx,%ds
        movw    %dx,%ss
#
```

```
# Return from PM ro real mode
#
        movl    %cr0,%eax
        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
#
# Re-enable interrupts (at CPU and at 8259)
# XXX - read old irq mask from port 0x21, save it, restore it here
#
       sti
       mov     $0xB8,%al
       out     %al,$0x21
#
# Exit to MS-DOS
#
       ret
#------------;
# User tasks ;
#------------;
# task 1
#
user1:
        movb    $64,%bl
suite1: incb    %bl
addr32  movb    %bl,%es:(0x0B80A0)
        movl    $0xFFFF,%ecx           # 0FFFFh
here1:  loop    here1                  # infinite loop (until timer interrupt)
        jmp     suite1
#
# task 2
#
user2:
        movb    $66,%bl
suite2: incb    %bl
addr32  movb    %bl,%es:(0x0B80A4)
        movl    $0xFFFF,%ecx           # 0FFFFh
```

```
here2:  loop    here2                   # infinite loop (until timer interrupt)
        jmp     suite2
#-----------------------------------------------------------------;
#       default handler for interrupts/exceptions                 ;
#       just puts '!' in upper right corner of screen and freezes ;
#-----------------------------------------------------------------;
unhand: cli
addr32  movb    $0x21,%es:(0x0B80A0) # '!'
here:   jmp     here
#
#---------------------------------------;
#       GDT to be used in Protected Mode ;
#---------------------------------------;
#
# null descriptor
#
DESC0:  .long   0                       # null descriptor
        .long   0
#
# descriptor for code segment in PM and real Mode
#
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 absolute data segment in PM
#
DESC2:  .word   0x0FFFF                 # limit 4GB
        .word   0                       # base = 0
        .byte   0                       # base
        .byte   0x92                    # R/W segment
        .byte   0xCF                    # 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 in PM beginning as address given by MS-DOS
#
DESC4:  .word   0x0FFFF                 # limit 64kB
DESC4B: .word   0                       # base to set
        .byte   0                       # base
DESC4C: .byte   0x92                    # R/W segment
        .byte   0x0CF                   # G = 1, limit
        .byte   0                       # base
```

```
#
#
# system TSS
#
DESC5:  .word   103
DESC5B: .word   0                       # base to set at stss
DESC5C: .byte   0
DESC5E: .byte   0x89                    # present, ring 0, 386 available TSS
        .byte   0
DESC5D: .byte   0
#
# user TSS 1
#
DESC6:  .word   103
DESC6B: .word   0                       # base to set at utss
DESC6C: .byte   0
DESC6E: .byte   0x89                    # present, ring 0, 386 available TSS
        .byte   0
DESC6D: .byte   0
#
# user TSS 2
#
DESC7:  .word   103
DESC7B: .word   0                       # base to set at utss
DESC7C: .byte   0
DESC7E: .byte   0x89                    # present, ring 0, 386 available TSS
        .byte   0
DESC7D: .byte   0
#
# GDT table data
#
GDTL:   .word   0x3F                    # limit
GDTA:   .long   0                       # base to set
#---------------------------;
#       task state segments ;
#---------------------------;
# System TSS
#
stss:   .word   0, 0                    # back link
        .long   0                       # ESP0
        .word   0, 0                    # SS0, reserved
        .long   0                       # ESP1
        .word   0, 0                    # SS1, reserved
        .long   0                       # ESP2
        .word   0, 0                    # SS2, reserved
        .long   0, 0, 0                 # CR3, EIP, EFLAGS
        .long   0, 0, 0, 0              # EAX, ECX, EDX, EBX
        .long   0, 0, 0, 0              # ESP, EBP, ESI, EDI
        .word   0, 0                    # ES, reserved
        .word   0, 0                    # CS, reserved
        .word   0, 0                    # SS, reserved
        .word   0, 0                    # DS, reserved
        .word   0, 0                    # FS, reserved
        .word   0, 0                    # GS, reserved
```

```
        .word   0, 0                    # LDT, reserved
        .word   0, 0                    # debug, IO perm. bitmap
#
# User TSS 1
#
utss1:  .word   0, 0                    # back link
        .long   0                       # ESP0
        .word   0, 0                    # SS0, reserved
        .long   0                       # ESP1
        .word   0, 0                    # SS1, reserved
        .long   0                       # ESP2
        .word   0, 0                    # SS2, reserved
        .long   0                       # CR3
utss1_eip:
        .long   0, 0x200                # EIP, EFLAGS (EFLAGS=0x200 for ints)
        .long   0, 0, 0, 0              # EAX, ECX, EDX, EBX
utss1_esp:
        .long   0, 0, 0, 0              # ESP, EBP, ESI, EDI
        .word   0x10, 0                 # ES, reserved
        .word   0x8, 0                  # CS, reserved
        .word   0x10, 0                 # SS, reserved
        .word   0x20, 0                 # DS, reserved
        .word   0x10, 0                 # FS, reserved
        .word   0x10, 0                 # GS, reserved
        .word   0, 0                    # LDT, reserved
        .word   0, 0                    # debug, IO perm. bitmap
#
# User TSS 2
#
utss2:  .word   0, 0                    # back link
        .long   0                       # ESP0
        .word   0, 0                    # SS0, reserved
        .long   0                       # ESP1
        .word   0, 0                    # SS1, reserved
        .long   0                       # ESP2
        .word   0, 0                    # SS2, reserved
        .long   0                       # CR3
utss2_eip:
        .long   0, 0x200                # EIP, EFLAGS (EFLAGS=0x200 for ints)
        .long   0, 0, 0, 0              # EAX, ECX, EDX, EBX
utss2_esp:
        .long   0, 0, 0, 0              # ESP, EBP, ESI, EDI
        .word   0x10, 0                 # ES, reserved
        .word   0x8, 0                  # CS, reserved
        .word   0x10, 0                 # SS, reserved
        .word   0x20, 0                 # DS, reserved
        .word   0x10, 0                 # FS, reserved
        .word   0x10, 0                 # GS, reserved
        .word   0, 0                    # LDT, reserved
        .word   0, 0                    # debug, IO perm. bitmap
#
#--------------------------------------;
#       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
# INT 8 is IRQ0 (timer interrupt). The 8259's can (and should) be
# reprogrammed to assign the IRQs to higher INTs, since the first
# 32 INTs are Intel-reserved. Didn't IBM or Microsoft RTFM?
        .word   0
        .word   0x28                  # SYS_TSS
        .byte   0
        .byte   0x85                  # Present ring 0 task gate
        .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
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 `task4.s` sous Linux :

```
$ as task4.s --32 -o ./task4.o
$ objcopy -O binary ./task4.o ./task4.com
$ ls -l task4.com
-rwxr-xr-x 1 patrick patrick 1368 oct.   7 10:41 task4.com
$
```

Il ne faut en garder que les 1 112 derniers octets du binaire :

```
$ objcopy -O binary -i 1500 --interleave-width 1112 -b 256 task4.o ./task4.com
$
```

Lorsqu'on exécute ce programme sous MS-DOS, on voit défiler un certain nombre de caractères de part et d'autre de '-B-' au début de la première ligne, ce qui correspond à ce que nous voulions.

Commentaire.- Il ne surtout pas oublier la valeur `0x200` de `EFLAGS` dans le descripteur des TSS utilisateur, c'est-à-dire de positionner l'indicateur `IF`, sinon l'interruption n'est pas prise en compte.

11.5 Historique

Dans le système dit des *transmissions multiples* de télégraphie, on fait usage de plusieurs *appareils transmetteurs*, manœuvrés simultanément, un par employé. La transmission en découpée en intervalles temporels réguliers et périodiques, dont chaque période est affectée à un appareil transmetteur distinct. Imaginé en 1860 par ROUVIER, inspecteur des lignes télégraphiques françaises, la première réalisation a lieu en 1871 par MEYER, employé de l'administration des télégraphes de Paris. Le télégraphe Meyer comprend trois appareils distincts : le *récepteur*, le *transmetteur* et surtout le *distributeur* qui permet d'envoyer tour à tour les dépêches de chacun des appareils.

Le concept de temps partagé (*Tine Sharing*) pour les ordinateurs a d'abord été décrit par Bob BEMER en 1957, dans un article de la revue *Automatic Control Magazine*. Le projet MAC (*Multi Access Computer*), dirigé par John MCCARTHY au MIT, a été l'une des premières (voire la première) implémentation. Plusieurs autres suivirent.

Après la démonstration du système *Compatible Time Sharing System* (CTSS) en novembre 1961, puis du DTSS, les principes ont montré leur efficacité. Ce système inspira en particulier le système d'exploitation *Multics*, du SDS Sigma 7, et plusieurs de ces principes furent utilisés par Ken THOMPSON lors de la conception de la première version de ce qu'il appela UNIX, à la suite d'une proposition de Brian KERNIGHAN.

Christopher STRACHEY est parfois crédité de l'invention du temps partagé. Toutefois, ce qu'il décrit est plus proche du multitâche. Temps partagé se réfère à l'utilisation d'un ordinateur par plusieurs utilisateurs, tandis que multitâche évoque plus largement le déroulement simultané de plusieurs processus sans accorder d'importance spéciale au nombre d'utilisateurs.

11.6 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.

[Le programme PM7 est celui dont nous nous sommes inspirés pour la commutation de tâche : il est cependant plus compliqué dans la mesure où les interruptions sont traitées.]