Deuxième partie

Le système d'exploitation comme gestionnaire des ressources

Chapitre 3

Les processus de MS-DOS

3.1 Philosophie des processus de MS-DOS

Le système d'exploitation MS-DOS n'est ni un système orienté monoprogrammation, ni multiprogrammation (contrairement à UNIX, par exemple). Il se trouve quelque part entre les deux, avec une philosophie qui lui est propre.

3.1.1 Processus actif

Au démarrage de MS-DOS, le processus `command.com` (interpréteur de commande orienté clavier) est lancé et attend une entrée. Dès qu'une ligne est tapée, `command.com` lance un nouveau processus, lui passe le contrôle et attend qu'il se termine pour recommencer.

Jusqu'à ce point, il n'y a pas de différence avec UNIX, où l'interpréteur de commande (*shell*) attend également qu'une commande se termine pour afficher de nouveau l'invite. Mais, sous UNIX, il suffit de placer une esperluette '&' après la commande pour que l'interpréteur de commande affiche immédiatement l'invite (alors que le premier processus utilisateur se déroule), ce qui permet de créer un autre processus utilisateur et donc d'effectuer de la multiprogrammation. De façon générale, un processus père créant un processus fils n'a pas besoin, sous Unix, d'attendre la fin de celui-ci et peut continuer à être effectué en parallèle avec son fils.

Sous MS-DOS, un processus père et son processus fils ne s'exécutent pas en parallèle : lorsqu'un processus père crée un processus fils, il se met automatiquement en attente de la fin du processus fils. Il peut donc y avoir, à un instant donné, un nombre quelconque de processus en mémoire, mais un seul est **actif**, tous les autres étant en attente de la fin d'un fils.

3.1.2 Programme résident

À la vérité on a un embryon de multiprogrammation en utilisant les programmes **TSR** (*Terminate and Stay Resident*), dits aussi **programmes résidents**. En effet, normalement, lorsqu'un processus se termine, sa mémoire est désalouée et le processus disparaît à jamais. MS-DOS offre toutefois une autre possibilité : un processus peut en effet ordonner au système de *ne pas* récupérer sa mémoire lorsqu'il se termine.

À première vue, avoir un **processus mort** qui ne peut pas être activé n'a rien d'intéressant. Mais il faut se rappeler que, sous MS-DOS, les processus peuvent installer leurs propres gestionnaires d'interruption. Ainsi, par exemple, un processus peut installer un nouveau gestionnaire de clavier dont la routine de service sera exécutée à chaque interruption provenant du clavier, à la place de la routine du BIOS. Cette routine se trouve à l'intérieur d'un processus TSR qui, lui, demeure inaccessible.

Le gestionnaire examine rapidement si ce qui a été frappé au clavier est une touche spéciale ou une combinaison de touches, appelée parfois **touche rapide** (en anglais *hot key*), qui active le code TSR. Si ce n'est pas le cas, le caractère est placé dans la file d'attente de caractères du système d'exploitation et le programme TSR revient de l'interruption vers l'application. Si c'est bien une touche rapide, le TSR réactivé fait alors ce qu'il est censé faire.

3.2 Les programmes exécutables de MS-DOS

Il y a deux sortes de fichiers directement pris en compte sur la ligne de commande : les **fichiers de script** de commandes, d'extension `.bat`, et les **programmes exécutables**. Un programme exécutable comprend deux parties : la première est une simple traduction en langage machine d'un programme source ; mais il comprend également un certain nombre de paramètres pour déterminer l'adresse à laquelle il faut le placer en mémoire et la façon de revenir à l'interpréteur de commandes après son exécution, ce qui en constitue la seconde partie. C'est la raison pour laquelle un exécutable pour un système d'exploitation donné (par exemple PC avec Linux) n'est pas un exécutable pour un autre système (par exemple PC avec MS-DOS), même si on est capable, par ailleurs, de traduire l'un en l'autre.

3.2.1 Deux types d'exécutables sous MS-DOS

Sous MS-DOS il existe deux types de programmes exécutables : les héritiers du système CP/M (pour le microprocesseur `8085`), d'extension `.com`, et ceux qui sont spécifiques à MS-DOS, commençant par les deux premiers octets 'MZ' et en général d'extension `.exe`, bien que cela ne soit pas obligatoire.

Il est rare qu'un système d'exploitation ait deux types de programmes exécutables ; comme toujours, c'est pour une raison de compatiblité avec un système antérieur que MS-DOS se distingue : en effet, au moment de la conception de MS-DOS, le système d'exploitation le plus répandu sur les micro-ordinateurs était CP/M ; les programmes exécutables CP/M devaient donc pouvoir tourner sous MS-DOS (avec l'extension `.com`).

Remarquons qu'il est traditionnel de parler du deuxième type de programmes exécutables de MS-DOS comme programmes `.exe`, bien que cette extension ne soit pas obligatoire (mais conseillée) comme nous l'avons déjà dit plus haut.

3.2.2 Les programmes `.com`

Nous n'avons écrit jusqu'à maintenant que des programmes `.exe`. Nous pouvons aussi écrire des programmes `.com`, dont le plus célèbre est certainement `command.com`. L'intérêt des programmes `.com` est qu'ils ont une taille plus petite que les programmes `.exe` correspondants, puisqu'il n'y a pas d'en-tête, comme nous allons le voir. Par contre leur taille est limitée à 64 kiO.

3.2.2.1 Différence entre une programme `.exe` et un programme `.com`

Structure d'un fichier `.com`.- Un programme `.com` est tout simplement la traduction en langage machine du programme en langage d'assemblage correspondant. Il n'y a pas d'en-tête, contrairement à un programme `.exe`, ce qui explique pourquoi sa taille est inférieure (d'au moins 512 octets) à celle du programme `.exe` correspondant.

Taille maximum du programme.- Un programme `.exe` peut avoir pratiquement n'importe quelle taille alors qu'un programme `.com`, lorsqu'il est chargé, doit tenir sur un seul segment de code, et a donc une taille maximum de 64 kiO, y compris le préfixe PSP de 256 octets.

Les segments.- L'utilisation des segments pour les programmes `.com` est différente et plus facile que pour les programmes `.exe`.

Segment de pile.- Alors que l'on doit définir un segment de pile pour un programme `.exe`, celui-ci est généré automatiquement pour un programme `.com`. On ne doit surtout pas en parler. Si la taille de 64 KiO est insuffisante, le chargeur place la pile après le programme, en mémoire haute.

Segment des données.- Pour un programme `.exe`, on doit définir un segment des données et initialiser le registre `ds` avec l'adresse de ce segment. Pour un programme `.com`, il n'y a pas de segment de données indépendant : les données sont incluses dans le segment de code.

Segment de code.- C'est en fait le seul segment qui existe, dont la taille est d'au maximum 64 kiO, rappelons-le.

Initialisations.- Pour un programme `.exe`, on doit initialiser les registres `CS` et `DS`. Lorsqu'un programme `.com` est chargé en mémoire, les quatre registres de segment sont automatiquement initialisés avec l'adresse du préfixe PSP : on n'a donc pas à s'en occuper.

Par contre, puisqu'un PSP occupe les 256 premiers octets du code en mémoire centrale, on doit indiquer que le décalage du pointeur d'instruction `IP` se trouve à l'adresse `100h`. Ceci se fait, en langage d'assemblage, en plaçant la directive :

```
ORG 100h
```

immédiatement après le début du segment de code, disons :

```
.code
```

Le PSP est un concept hérité de CP/M. Dans le cas des exécutables `.com`, le PSP fait partie de l'espace d'adressage du processus, c'est pourquoi tous les processus `.com` commencent à l'adresse 256 et non pas à 0. En revanche, les exécutables `.exe` sont translatés au-dessus du PSP et leur adresse 0 correspond donc au premier octet au-dessus du PSP, ce qui évite de gâcher 256 octets de l'espace d'adressage.

Processus fils.- Même si un processus `.com` ne peut pas dépasser 64 kiO, toute la mémoire disponible lui est allouée. Si le processus n'a pas l'intention de créer de fils, il peut s'éxécuter sans problèmes. Mais s'il veut créer un fils, il doit restituer au système d'exploitation, au moyen d'un appel système, la partie de la mémoire qu'il n'utilise pas, faute de quoi sa tentative de créer un fils échouera en raison d'un manque de place mémoire.

3.2.2.2 Création d'un programme `.com`

Écriture du programme source.- Considérons l'un des premiers programmes en langage d'assemblage que nous avons écrit, disons le programme `affiche.asm` :

```
; affiche.asm

; affiche la lettre 'a' a l'ecran

           .model small
           .stack 100h

           .code
start :
           mov dl,'a' ;stocke le code ascii de 'a' dans dl

           mov ah,2h  ;appel de la fonction ms-dos d'affichage
                      ;d'un caractere

           int 21h    ;affiche le caractere qui se trouve en dl

           mov ax, 4c00h ;revient a ms-dos
           int 21h
           end start
```

Il ne peut pas servir de programme source pour un programme exécutable de type `.com` : il n'y a pas de déclaration de variable, ce qui est bien, mais il y a déclaration de la taille de la pile et il n'y a pas de directive d'origine. De plus seul le modèle `tiny` convient pour un programme `.com`.

Il n'est pas difficile, cependant, de le transformer, en un programme source qui convient :

```
; affiche1.asm

; affiche la lettre 'a' a l'ecran

           .model tiny

           .code
           ORG 100h
start :
           mov dl,'a' ;stocke le code ascii de 'a' dans dl

           mov ah,2h  ;appel de la fonction ms-dos d'affichage
                      ;d'un caractere

           int 21h    ;affiche le caractere qui se trouve en dl

           mov ax, 4c00h ;revient a ms-dos
           int 21h
           end start
```

Assemblage.- Rappelons que l'assemblage au sens fort comprend l'assemblage proprement dit, produisant un fichier `.obj`, et le liage. On fait appel à l'assemblage d'un programme `.com` de la même façon qu'un programme `.exe`, par exemple :

```
masm affiche1.asm
```

Liage.- On peut lier un programme `.com` de la même façon que pour un programme `.exe`. Le lieur envoie alors un message d'avertissement :

```
Warning: No STACK Segment.
```

dont on peut ne pas tenir compte. On obtient cependant ainsi un exécutable `.exe`.

La méthode pour obtenir un exécutable `.com` dépend de l'assembleur utilisé (essentiellement MASM et TASM). Avec TASM il suffit d'utiliser l'option `\T` :

```
TLINK \T affiche1.obj
```

On obtient alors `affiche1.com` que l'on peut faire exécuter.

Comparaison des tailles.- Il est intéressant de comparer les tailles des deux exécutables, pour `affiche.exe` et pour `affiche1.com`. On trouve que `affiche.exe` a une taille de 523 octets et `affiche.com` de 11 octets, ce qui s'explique bien par la présence de l'en-tête de 512 octets pour l'exécutable `.exe`.

3.2.2.3 Création d'un programme `.com` possédant des données

Voyons maintenant comment créer un programme `.com` possédant des données. Rappelons qu'il n'y a pas de segment de données, nous devons donc procéder autrement.

Écriture du programme source.- Écrivons un programme permettant de calculer la somme de deux entiers placés dans `A` et `B`, somme à placer dans `C`.

Pour un exécutable `.exe`, on écrirait :

```
; sum.asm

      .model small
      .stack 100h
      .data
A     DW         15
B     DW         23
C     DW         ?
      .code
start :
      mov ax, @data
      mov ds, ax     ; charge le segment des donnees

      mov ax, A
      add ax, B
      mov C,ax

      mov ax, 4C00h
      int 21h
      end start
```

Pour un exécutable `.com`, on écrira :

```
; sum1.asm

      .model tiny
      .code
      ORG     100H
start :
      JMP     MAIN
A     DW      15
B     DW      23
C     DW      ?

MAIN PROC     NEAR
      mov ax, A
      add ax, B
      mov C,ax

      mov ax, 4C00h
      int 21h
MAIN  ENDP
      end start
```

Assemblage et liage.- On commence par assembler ce programme :

```
tasm sum1.asm
```

puis on le lie :

```
TLINK \T sum1.obj
```

On obtient alors `sum1.com` que l'on peut faire exécuter.

Remarques.- 1^o) Comme nous venons de le voir sur notre exemple, un programme `.com` commence en général par un saut inconditionnel, pour se placer au début effectif du code, ce qui permet de laisser de la place pour les données. Une autre façon de faire est, si l'on préfère, de placer les données à la fin.

- 2^o) Les deux dernières lignes de code :

```
mov ax, 4C00h
int 21h
```

permettent de retourner au système d'exploitation (en général à l'interpréteur de commande). On pourrait les remplacer par :

```
RET
```

qui renvoie alors à `INT 20h` à l'octet `01h` du PSP, ancienne façon de quitter un programme.

3.2.2.4 Structure d'un exécutable `.com`

La structure d'un exécutable `.com` est simple : il s'agit tout simplement de la traduction en langage machine du programme source correspondant.

Exemple.- Considérons le programme `sum1.com` ci-dessus. Engendrons le fichier `sum1.lst` correspondant :

```
tasm sum1.asm,,,
```

Nous obtenons :

```
Turbo Assembler  Version 2.01      11/30/99 10:43:37     Page 1
sum1.asm



      1                             ; sum1.asm
      2
      3 0000                              .model tiny
      4 0000                              .code
      5                                   ORG     100H
      6 0100                        start :
      7 0100  EB 07 90                    JMP           MAIN
      8 0103  000F                  A     DW      15
      9 0105  0017                  B     DW      23
     10 0107  ????                  C     DW      ?
     11
     12 0109                        MAIN  PROC    NEAR
     13 0109  A1 0103r                   mov ax, A
     14 010C  03 06 0105r                add ax, B
     15 0110  A3 0107r                   mov C,ax
     16
     17 0113  B8 4C00                    mov ax, 4C00h
     18 0116  CD 21                      int 21h
     19 0118                             MAIN    ENDP
     20                                  end start

Turbo Assembler  Version 2.01      11/30/99 10:43:37     Page 2
Symbol Table



Symbol Name             Type   Value                      Cref (defined at #)

??DATE                  Text   "11/30/99"
??FILENAME              Text   "sum1    "
??TIME                  Text   "10:43:37"
??VERSION               Number 0201
@CODE                   Text   DGROUP                     #3
@CODESIZE               Text   0                          #3
@CPU                    Text   0101H
@CURSEG                 Text   _TEXT                      #4
@DATA                   Text   DGROUP                     #3
@DATASIZE               Text   0                          #3
```

```
@FILENAME               Text   SUM1
@MODEL                  Text   1                          #3
@WORDSIZE               Text   2                          #4
A                       Word   DGROUP:0103                #8  13
B                       Word   DGROUP:0105                #9  14
C                       Word   DGROUP:0107                #10 15
MAIN                    Near   DGROUP:0109                7  #12
START                   Near   DGROUP:0100                #6  20

Groups & Segments       Bit Size Align  Combine Class      Cref
                                                 (defined at #)

DGROUP                  Group                             #3  3
  _DATA                 16  0000 Word   Public  DATA       #3
  _TEXT                 16  0118 Word   Public  CODE       #3  #4
```

Nous obtenons un certain nombre de renseignements. dont le seul qui nous intéresse ici est la traduction en langage machine.

Examinons le fichier `sum1.com` en hexadécimal, par exemple grâce à DOSSHELL :

```
|  000000  |  EB07900F   00170000   00A10301   03060501   |  ................  |
|  000010  |  A30701B8   004CCD21                         |  .....L=!          |
```

On retrouve bien le programme en langage machine, sans autre élément.

3.2.3 Création d'un exécutable `.com` à la main

Principe.- Nous venons de voir la structure d'un exécutable `.com` : il s'agit d'un programme en langage machine. Nous pouvons donc créer un tel programme sans avoir recours à un assembleur, simplement en utilisant un éditeur de texte, si possible hexadécimal.

Exemple.-

3.2.4 Les programmes `.exe`

3.2.4.1 Structure d'un programme `.exe`

En-tête et module chargeable.- Un programme `.exe` est stocké sur disque en deux parties (formant un seul fichier) : l'**en-tête** (en anglais *header record*), contenant les paramètres de chargement, et le **module chargeable** (en anglais *load module*).

Le module chargeable est du langage machine, il n'y a donc de spécial rien à en dire.

L'en-tête comprend au moins 512 octets mais peut être de taille plus importante s'il y a beaucoup d'items d'adresses relatives. L'en-tête doit donc spécifier sa taille réelle, à un emplacement déterminé, pour qu'on puisser déterminer où commence le module chargeable.

L'en-tête contient des informations sur la taille du module exécutable, sur l'emplacement où il doit être chargé en mémoire vive, sur l'adresse de la pile et sur les décalages de transfert (*relocation* en anglais) à insérer dans les adresses machine incomplètes.

Structure de l'en-tête.- La structure détaillée de l'en-tête est la suivante (un *bloc* représentant 512 octets) :

- `00-01h` : Il s'agit du code identifiant un fichier `.exe`, soit le nombre magique `4D5Ah` ('MZ').
- `02-03h` : Nombre d'octets utiles du dernier bloc du fichier `.exe`.
- `04-05h` : Taille du fichier, y compris l'en-tête, en nombre de blocs entiers. Ainsi si la taille est 1 025, ce champ contient 2 et le champ précédent 1 pour des blocs de 512 octets.
- `06-07h` : Nombre d'items dans la table de transfert (*relocation*), celle-ci commençant en `1Ch`, comme nous le verrons. Ce champ permet de déterminer la taille de l'en-tête.
- `08-09h` : Taille de l'en-tête, en paragraphes de 16 octets. Ceci permet au chargeur de savoir où commence le module exécutable. Ce nombre est au minimum de `20h`, soit $32 \times 16 = 512$ octets.
- `01-0Bh` : Nombre minimum de paragraphes devant résider au-dessus de la fin du programme lorsqu'il est chargé.
- `0C-0Dh` : Interrupteur de chargement haut/bas. On peut décider que le programme doit être chargé, à fin d'exécution lorsqu'il sera lié, soit en mémoire basse (ce qui est le cas en général), soit en mémoire haute. La valeur `0000h` spécifie la mémoire haute. Sinon cet emplacement contient le nombre maximum de paragraphes devant résider au-dessus de la fin du programme chargé.
- `0E-0Fh` : Emplacement du décalage dans le module exécutable du segment de pile.
- `10-11h` : Taille de la pile, plus précisément décalage que le chargeur doit insérer dans le registre `SP` lorsqu'il transfère le contrôle au module exécutable.
- `12-13h` : Valeur de la *somme de vérification*, à savoir la somme de tous les mots du fichier (faisant fi des dépassements), utilisée pour vérifier qu'aucune donnée n'a été perdue.
- `14-15h` : Décalage (en général égal à `00h`) que le chargeur doit insérer dans le registre `IP` lorsqu'il tranfère le contrôle au module exécutable.
- `16-17h` : Décalage du segment de code que le chargeur doit insérer dans le registre `CS` au moment du transfert du contrôle au module exécutable. Ce décalage est relatif aux autres segments, donc si le segment de code est le premier alors ce décalage doit être nul.
- `18-19h` : Décalage de la table de transfert (*relocation*), l'item commençant à `1Ch`.
- `1A-1Bh` : Numéro d'*overlay*, le numéro 0 (cas général) signifiant que le fichier `.exe` contient le programme principal.

— `1Ch-fin` : Table de transfert (*relocation*), contenant un nombre variable d'items de transfert, ce nombre étant spécifié au décalage `06-07h`, comme nous l'avons vu. Chaque tel item, commençant à l'emplacement `1Ch` de l'en-tête, comprend une valeur de décalage, sur deux octets, et une valeur de segment, également sur deux octets.

3.2.4.2 Exemple d'en-tête

Considérons un programme simple en langage d'assemblage, examinons l'en-tête généré par l'assembleur et commentons-le.

Le programme.- Considérons le programme suivant, dont l'effet est d'écrire 'Bonjour' :

```
; Bonjour.asm

; affiche 'Bonjour' a l'ecran et va a la ligne

           .model small
           .stack 100h

           .code
start :
;affiche 'B'
           mov dl, 'B'
           mov ah, 2h
           int 21h
;affiche 'o'
           mov dl, 'o'
           mov ah, 2h
           int 21h
;affiche 'n'
           mov dl, 'n'
           mov ah, 2h
           int 21h
;affiche 'j'
           mov dl, 'j'
           mov ah, 2h
           int 21h
;affiche 'o'
           mov dl, 'o'
           mov ah, 2h
           int 21h
;affiche 'u'
           mov dl, 'u'
           mov ah, 2h
           int 21h
;affiche 'r'
           mov dl, 'r'
           mov ah, 2h
           int 21h
;passe a la ligne
           mov dl, 13
           mov ah, 2h
           int 21h
           mov dl, 10
```

```
           mov ah,2h
           int 21h
;revient a ms-dos
           mov ax, 4c00h
           int 21h
           end start
```

Fichier `.map` associé.- Lors de l'assemblage de ce programme, on obtient le fichier `bonjour.map` suivant :

```
 Start  Stop   Length Name               Class

 00000H 0003AH 0003BH _TEXT              CODE
 00040H 00040H 00000H _DATA              DATA
 00040H 0013FH 00100H STACK              STACK

Program entry point at 0000:0000
```

Ce fichier indique les positions relatives de chacun des trois segments du programme. Ici le segment de code (de nom traditionnel `_TEXT`) commence à la position relative `00000h` par rapport au début du module exécutable, et sa longueur est de `0003Bh`, soit 59, octets. Le segment des données (de nom `_DATA`) commence à la position relative `00040h` et a une longueur nulle, ce qui est normal puisque nous n'avons défini aucune donnée. Remarquons que `00040h` est le première adresse suivant l'espace occupé par le code et correspondant au début d'un paragraphe, c'est-à-dire divisible par `10h`. Le segment de pile (de nom `STACK`) commence à l'adresse `00040h`, c'est-à-dire à la première adresse suivant `_DATA` commençant au début d'un paragraphe. Sa longueur, `100h`, est celle que nous avons spécifiée dans le programme.

Examen de l'en-tête du fichier `.exe`.- Nous pouvons examiner le contenu du fichier `bonjour.exe` avec un éditeur de texte hexadécimal, par exemple `debug` ou `dosshell`, qui était livré avec la version 5 de MS-DOS.

Utilisons `dosshell`. Nous obtenons les trois premières lignes suivantes de l'affichage du fichier en hexadécimal :

```
|  000000  |  4D5A3B00   02000000   20001100   FFFF0400   |  MZ;..... ...  ..  |
|  000010  |  00010000   00000000   3E000000   0100FB30   |  ........>.....%0  |
|  000020  |  6A720000   00000000   00000000   00000000   |  jr..............  |
```

Commentons ces lignes :

- Comme attendu, les deux premiers octets sont `4D54h`, soit 'MZ' en ASCII.
- Le nombre d'octets dans le dernier bloc est `3Bh00h`, soit `003Bh`, ou 59.
- La taille du fichier, y compris l'en-tête, est de `02h00h`, soit `0002h`, blocs de 512 octets, soit 1024 octets.

  En fait la commande `dir` indique que le fichier occupe 571 octets, soit 512 plus les 59 octets du segment de code indiqués dans le fichier `bonjour.map`.
- Le nombre d'items relogeables est de `00h00h`, soit de `0000h`. Il n'y en a donc pas.
- La taille de l'en-tête, en paragraphes de 16 octets, est de `20h00h`, soit `0020h`, ou 32. L'en-tête occupe donc 512 octets, le minimum permis.
- Le nombre minimum de paragraphes devant résider après le programme, lorsque celui-ci sera chargé, est de `11h00h`, soit `0011h` ou 17.

— Les deux octets suivants sont `FFh` et `FFh`. Le programme sera donc chargé en mémoire basse et pourra occuper tout l'espace si nécessaire.

— Le segment de pile sera placé à l'adresse `04h00h`, soit `0004h`.

— Les deux premiers octets de la deuxième ligne, à savoir `00h01h` spécifient le décalage à insérer dans `SP`, ici `0100h`.

— La valeur de la somme de vérification est ici `00h00h`, soit `0000h`.

— Le décalage pour `IP` est `00h00h`, soit `0000h`.

— Le décalage pour `CS` est `00h00h`, soit `0000h`.

— Le décalage pour la table d'items relogeables est `3Eh 00h`, soit `003Eh`.

— Le numéro d'*overlay* est `00h 00h`, soit `0000h`. Il s'agit donc du programme principal.

— La table d'items relogeables contient quelques valeurs, non prises en compte puisqu'il est indiqué précédemment qu'elle ne contient aucune valeur (pertinente). Les autres valeurs sont nulles jusqu'à `0001FF`.

Examen du module chargeable.- Le module chargeable commence à l'adresse `000200`, puisque l'en-tête occupe un bloc de 512 octets. On obtient le résultat suivant :

```
|  000200  |  B242B402   CD21B26F   B402CD21   B26EB402   |  &B{.=!&o{.=!&n{.  |
|  000210  |  CD21B26A   B402CD21   B26FB402   CD21B275   |  =!&j{.=!&o{.=!&u  |
|  000220  |  B402CD21   B272B402   CD21B20D   B402CD21   |  {.=!&r{.=!&.{.=!  |
|  000230  |  B20AB402   CD21B800   4CCD21                |  &.{.=!..L=!
```

Remarquons qu'il s'agit de la simple transcription du programme en langage machine.

3.2.4.3 Autre exemple d'en-tête

Donnons maintenant un exemple d'en-tête de programmes ayant des données ainsi que des items relogeables.

3.2.4.4 Création d'un fichier `.exe` à la main

Jusqu'à maintenant, nous avons utilisé des fichiers `.exe` qui nous ont été transmis tels quels ou que nous avons obtenus grâce à un compilateur ou un assembleur. Construisons un tel fichier à la main, en utilisant seulement un éditeur de texte hexadécimal. Ceci est la première étape à effectuer avant, par exemple, l'écriture de notre propre assembleur.

Exemple.- Commençons par une petite variation de notre programme précédent. Disons que nous sommes dans une situation dans laquelle nous disposons seulement de l'exécutable d'un programme, par exemple le programme `bonjour.exe`. Nous voudrions le transformer en un programme `welcome.exe`, affichant 'Welcome' au lieu de 'Bonjour'.

Il suffit, bien entendu de remplacer la constante 'B' par 'W' et ainsi de suite. Nous avons même intérêt ici à nous servir d'un éditeur de texte ordinaire (et non pas d'un éditeur de texte hexadécimal).

Cherchons la première lettre à remplacer : '`B`'. On laremplace par '`W`'. On remplace de même '`o`' par '`e`', et ainsi de suite. On sauvegarde le fichier modifié sous le nom `welcome.exe`.

On peut s'assurer qu'il s'exécute correctement en affichant 'Welcome'.

Remarquons que nous avons eu de la chance car 'Welcome' et 'Bonjour' comportent tous les deux sept lettres.

3.2.4.5 Deuxième création d'un fichier `.exe`

Plaçons-nous maintenant dans le cas où nous avons à créer entièrement notre fichier `.exe` à partir d'un programme en langage d'assemblage.

3.3 Chargement des programmes

Nous avons vu la notion de **programme exécutable** (en abrégé **exécutable**), par opposition aux *programmes source*. Nous avons vu qu'il ne s'agit, en général, pas simplement du programme source traduit en langage machine mais qu'il contient aussi des renseignements sur la façon dont le **chargeur** (de programme) doit placer ce programme en mémoire vive et lui donner la main.

Le chargeur est l'un des constituants fondamentaux du système d'exploitation. Il dépend de celui-ci, ce qui explique qu'un exécutable pour Unix, par exemple, ne peut pas tourner sous DOS.

Le chargeur de programmes est appelé à partir du processeur de commandes `command.com` ou d'un autre programme.

3.3.1 Fonction du processeur de commande

Le processeur de commande `command.com` interprète chaque commande écrite après le prompteur en effectuant séquentiellement les tâches suivantes :

1. Il vérifie s'il s'agit d'une **commande interne**, telle que `dir`, `ren`, `erase`. Si c'est le cas, celle-ci est immédiatement exécutée par la routine DOS correspondante, résidant en mémoire vive.

2. Sinon il cherche un fichier correspondant d'extension `.com` dans le répertoire actif. S'il le trouve, celui-ci est exécuté.

3. Sinon il cherche un fichier correspondant d'extension `.exe` dans le répertoire actif. S'il le trouve, celui-ci est exécuté.

4. Sinon il cherche un fichier correspondant d'extension `.bat` dans le répertoire actif. S'il le trouve, les commandes qui s'y trouvent sont interprétées. Rappelons qu'un **fichier batch** (fichier de traitement par lot) est un fichier texte contenant une suite de commandes du DOS.

5. Sinon il recherche la même chose pour le premier répertoire indiqué par la variable d'environnement `path` au lieu du répertoire actif. Sinon il le fait pour le second et ainsi de suite.

6. S'il n'a rien trouvé, il indique une erreur.

Il y a donc soit une commande interne à exécuter, soit un programme `.com` ou `.exe`. Ces programmes sont dits **transitoires** (en anglais *transient program*) puisqu'ils sont chargés en mémoire seulement le temps de leur exécution.

3.3.2 Préfixe d'un programme transitoire (PSP)

Un programme transitoire `.com` est chargé en mémoire vive dans un segment, un programme `.exe` dans un ou plusieurs segments. Les 256 premiers octets du segment de code sont occupés par des paramètres de MS-DOS (ce ne sont pas des instructions) et constituent le **préfixe de programme transitoire** (**PSP** pour l'anglais *Program Segment Prefix*). Il s'agit d'un reliquat du système CP/M (compatibilité oblige).

3.3.2.1 Structure du PSP

Introduction.- La structure non détaillée du PSP est la suivante :

Offset	Commentaires
00-15	Pointeurs MS-DOS et adresses des vecteurs
16-2B	Réservé par MS-DOS
2C-2D	Adresse du segment du mot de l'environnement actuel
2E-5B	Réservé par MS-DOS
5C-7F	Blocs de contrôle des fichiers 1 et 2, jusqu'à MS-DOS 2.0
80-FF	Aire de transfert du disque par défaut et copie de l'en-tête de la commande MS-DOS

Structure détaillée.- La structure détaillée du PSP est la suivante :

Offset	Commentaires
00-01	L'instruction INT 20h (CD20h) pour faciliter le retour au système
02-03	L'adresse de segment du dernier paragraphe de la mémoire allouée au programme, de la forme xxxx0. Par exemple 640 kiO est indiquée 00A0, pour A0000[0]
04-09	Réservé par le système
0A-0D	Adresse de fin (adresse de segment pour INT 22h)
OE-11	Adresse de sortie pour CTRL-Break (adresse de segment pour INT 23h)
12-15	Adresse de sortie sur erreur critique (adresse de segment pour INT 24h)
16-17	Réservé par le système
18-2B	Table des numéros des fichiers par défaut
2C-2D	Adresse de segment de l'environnement du programme
2E-31	Réservé par le système
32-33	Longueur de la table des numéros de fichier
34-37	Pointeur lointain (*far*) à la table des numéros de fichier
38-4F	Réservé par le système
50-51	Appel à la fonction INT 21h
52-5B	Réservé par le système
5C-6B	Aire de paramètre 1, au départ FCB non ouvert
6C-7F	Aire de paramètre 2, au départ FCB non ouvert
80-FF	Tampon pour un DTA par défaut

Détaillons certaines de ces zones.

PSP 18-2B : Table des numéros des fichiers par défaut.- Chacun des 20 octets de cette table fait référence à une entrée dans la table définissant les pilotes associés. Au départ la table contient `0101010002FF ... FF`, c'est-à-dire :

Table	Pilote	Numéro	Pilote
01	Console	0	Clavier (entrée standard)
01	Console	1	Écran (sortie standard)
01	Console	2	Écran (erreur standard)
00	COM1 (port série)	3	Auxiliaire
02	Imprimante	4	Imprimante standard
FF	Non assigné	5	Non assigné

Cette table de vingt numéros explique pourquoi le système ne permet que 20 fichiers ouverts à la fois au maximum. Normalement le mot du PSP de décalage `32h` contient la longueur de la table (`14h`, soit 20), et `34h` contient son adresse de segment sous la forme `IP:CS`, où `IP` est `18h` (le décalage dans le PSP) et `CS` l'adresse de segment du PSP.

Les programmes nécessitant plus de 20 fichiers ouverts doivent mettre à jour la mémoire (`INT 21h`, fonction `4Ah`) et doivent utiliser la fonction `67h` (positionner le nombre maximum de fichier) :

```
MOV AH,67h          ; necessite plus de numeros
MOV BX,count        ; nouveau nombre (de 20 a 65 535)
INT 21h             ; appel du service d'interruption
```

PSP 2C-2D : Adresse du segment d'environnement.- Tout programme chargé à fin d'exécution possède un **environnement**, que le système stocke en mémoire, débutant à la frontière du paragraphe précédant le segment du programme. La taille par défaut est de 160 octets, avec un maximum de 32 kiO. L'environnement contient des commandes du système telles que COMSPEC, PATH, PROMPT et SET.

PSP 5C-6B : FCB standard non ouvert numéro 1.- L'exécution du programme peut exiger un paramètre, par exemple `c:essai.asm` pour l'instruction `masm c:essai.asm`. Le chargeur de programme **formate** ce paramètre dans le FCB numéro 1, avec `03h` (pour le lecteur C), suivi du nom de fichier (8 caractères) et de l'extension (3 caractères). S'il n'y a pas de lecteur et de nom de fichier, le chargeur positionne le premier octet à `00h` (valeur par défaut) et le reste du FCB avec des espaces (`20h`).

PSP 6C-7F : FCB standard non ouvert numéro 2.- L'exécution du programme peut exiger deux paramètres. C'est le cas de l'instruction :

```
copy c:fich1.txt a:fich2.txt
```

Le chargeur de programme formate alors le deuxième paramètre dans le `FCB #2`.

PSP 80-FF : Tampon DTA par défaut.- Le chargeur de programme initialise le **tampon par défaut** pour le DTA avec le texte entier (s'il existe) qui est saisi après le nom du programme. Le premier octet contient le nombre de touches (s'il y en a) qui ont été pressées immédiatement après le nom du programme. Il peut rester des caractères du programme précédent.

3.3.2.2 Exemples

Voyons quelques exemples et les octets `2C-FF` correspondants du PSP.

Premier exemple : commande sans opérande.- Supposons que nous demandions l'exécution du programme `essai.exe` en tapant `essai` <`retour`>. Lorsque le chargeur de programme construit le PSP, il initialise FCB #1, FCB #2 et DTA de la façon suivante :

```
5C FCB #1 : 00 20 20 20 20 20 20 20 20 20 20 20 ...
6C FCB #2 : 00 20 20 20 20 20 20 20 20 20 20 20 ...
80 DTA : 00 0D ...
```

Le premier octet des FCB est `00h`, c'est-à-dire qu'il n'y a pas de référence à un disque. Le nom du fichier et l'extension sont constitués d'espaces.

Le premier octet du DTA contient le nombre d'octets tapés après le nom du programme, hormis la touche retour. Ici on a `00h` puisqu'on n'a appuyé que sur la touche retour. Le second octet contient `0Dh` pour la touche retour. Les autres octets dépendent de ce qui s'y trouvait là auparavant.

Deuxième exemple : commande avec un opérande texte.- Supposons que nous demandions l'exécution du programme `color.exe` en passant le paramètre `BY` (pour la couleur bleue sur un fond jaune). On tape `color BY` <`retour`>. Lorsque le chargeur de programme construit le PSP, il initialise FCB #1, FCB #2 et DTA de la façon suivante :

```
5C FCB #1 : 00 42 59 20 20 20 20 20 20 20 20 20 ...
6C FCB #2 : 00 20 20 20 20 20 20 20 20 20 20 20 ...
80 DTA : 03 20 42 59 0D ...
```

Le premier octet du FCB #1 est `00h`, c'est-à-dire qu'il n'y a pas de référence à un disque. Par contre le nom du fichier (présumé) est `4259h` pour BY et l'extension est constituée d'espaces.

Le premier octet du DTA contient cette fois-ci 3 puisque trois caractères ont été tapés après le nom du programme, suivi des codes ASCII de ces caractères, à savoir un espace, 'B' et 'Y'.

Troisième exemple : commande avec un opérande nom de fichier.- Supposons que nous demandions l'exécution du programme `del.com` en passant le paramètre `D:CALCIT.OBJ`. On tape `del D:CALCIT.OBJ` <`retour`>. Lorsque le chargeur de programme construit le PSP, il initialise FCB #1, FCB #2 et DTA de la façon suivante :

```
5C FCB \#1 : 04 43 41 4C 43 49 54 20 20 4F 42 4A ...
                C  A  L  C  I  T        O  B  J
6C FCB \#2 : 00 20 20 20 20 20 20 20 20 20 20 20 ...
80 DTA :     0D 20 44 3A 43 41 4C 43 49 54 2E 4F 42 4A 0D ...
                 D  :  C  A  L  C  I  T  .  O  B  J
```

Le premier octet du FCB #1 indique `04h`, c'est-à-dire le numéro du lecteur D, suivi du nom du fichier, 'CALCIT'. Deux espaces suivent pour compléter le nom du fichier. Il y a ensuite le nom de l'extension, 'OBJ', sans le point.

Le premier octet du DTA contient la longueur, 13, suivi exactement par ce qui a été tapé, y compris la touche retour.

Quatrième exemple : commande avec deux opérandes nom de fichier.- Supposons que nous demandions l'exécution du programme `copy.com`, ce qui exige deux paramètres. On tape, par exemple :

`copy A:FILEA.ASM D/FILEB.ASM <retour>`.

Lorsque le chargeur de programme construit le PSP, il initialise FCB #1, FCB #2 et DTA de la façon suivante :

```
5C FCB \#1 : 01 46 49 4C 45 41 20 20 20 41 53 4A ...
                F  I  L  E  A           A  S  M
6C FCB \#2 : 04 46 49 4C 45 42 20 20 20 41 53 4D ...
                F  I  L  E  B           A  S  M
80 DTA :     10 20 41 3A 46 49 4C 45 41 2E 41 53 4D 20 etc ...
                  A  :  F  I  L  E  A  .  A  S  M    etc ...
```

Le premier octet du FCB #1 indique `01h`, c'est-à-dire le numéro du lecteur A, suivi du nom du premier fichier.

Le premier octet du FCB #2 indique `04h`, c'est-à-dire le numéro du lecteur D, suivi du nom du second fichier.

Le premier octet du DTA contient la longueur, `10h`, suivi exactement par ce qui a été tapé, y compris le retour chariot.

3.3.3 Chargement d'un programme exécutable `.com`

3.3.3.1 Les étapes

Pour charger un fichier `.com` en mémoire, le chargeur de programme effectue les opérations suivantes :

- 1^o) Il détermine un emplacement mémoire suffisant, qu'il choisit comme segment du programme.

- 2^o) Il crée un PSP (*Program Segment Prefix*) à partir du décalage `00h` de ce segment de programme et charge le programme exécutable proprement dit à partir de l'adresse `100h`.

- 3^o) Il donne aux quatre registres de segment `CS`, `SS`, `DS` et `ES` l'adresse du premier octet du PSP.

- 4^o) Il donne au pointeur de pile `SP` l'adresse de la fin du segment de 64 kiO, c'est-à-dire le décalage `FFFEh` (ou la fin de la mémoire s'il n'y a pas assez de place), et place un mot nul sur la pile.

- 5^o) Il donne au pointeur d'instruction `IP` la valeur `100h` (la taille du PSP) et passe le contrôle à l'adresse `CS:IP`, le premier emplacement mémoire suivant immédiatement le PSP. C'est le premier octet du programme utilisateur, devant donc contenir une instruction exécutable.

3.3.3.2 Un exemple

Reprenons notre exemple d'exécutable `sum1.com`.

3.3.4 Chargement d'un programme `.exe`

Les étapes.- Lorsqu'on demande le chargement d'un exécutable `.exe` depuis un disque vers la mémoire centrale à fin d'exécution, le chargeur de programme effectue les étapes suivantes :

- 1^o) Il accède au programme `.exe` du disque. Il lit l'en-tête.

- 2^o) Il calcule la taille du module exécutable (taille du fichier en son entier située à la position `04h` moins la taille de l'en-tête située à la position `08h`).

- 3^o) Il cherche un emplacement mémoire disponible de taille suffisante, il détermine une adresse qui est le début d'un paragraphe.

- 4^o) Il construit le PSP (*Program Segment Prefix*) de 256 octets en commençant à l'adresse déterminée au 3^o).

- 5^o) Il place le module exécutable du programme immédiatement après le PSP.

- 6^o) Il lit les items de la table des relogements dans une aire de travail et il ajoute la valeur de chaque item à la valeur du début du segment.

- 7^o) Il charge l'adresse du PSP dans les registres `DS` et `ES`.

- 8^o) Il charge l'adresse du segment de code dans le registre `CS`, à savoir l'adresse du PSP plus `100h` (la taille du PSP) plus la valeur de décalage contenue à l'adresse `16h` du PSP. Il positionne le registre `IP` au décalage de la première instruction (en général 0) du segment de code, située à l'adresse `14h` du PSP.

- 9^o) Il charge l'adresse de la pile dans le registre `SS`, à savoir l'adresse du PSP plus `100h` (la taille du PSP) plus la valeur de décalage située à l'adresse `0Eh` de l'en-tête. Il place dans le registre `SP` la taille de la pile, en général `10h`.

- 10^o) Le chargeur n'a plus besoin de la copie de l'en-tête, aussi la détruit-il.

- 11^o) Il transfère le contrôle au programme pour exécuter celui-ci, en commençant à `CS:IP` (en général la première instruction du segment de code).

Remarque.- Les registres `DS` et `ES` sont chargés avec l'adresse du PSP alors qu'on aurait besoin de l'adresse du segment de données. C'est la raison pour laquelle le programme doit initialiser le registre `DS` avec l'adresse du segment de données :

```
mov ax, datasegname
mov ds, as
mov es, ax
```