# LES SYSTÈMES INFORMATIQUES

## Vision cohérente et utilisation

---

Christian CARREZ

Professeur des Universités au CNAM

# Avant-propos

Ce polycopié de cours est une mise à jour du livre que j'ai publié en 1990, et qui maintenant n'est plus disponible. C'est le support de cours pour la partie systèmes informatiques de la valeur d'architecture des machines et des systèmes informatiques du cycle A informatique du CNAM.

Pour réaliser des programmes, il est évident qu'il faut maîtriser les concepts de base de la programmation. Mais ce n'est pas suffisant. En particulier, il faut aussi comprendre l'ensemble des mécanismes et outils nécessaires à leur mise en œuvre en machine. C'est le but recherché par ce cours. En ce sens, il s'adresse en priorité à ceux qui seront des informaticiens professionnels. Cependant, comme on ne peut faire de bonne programmation sans cette compréhension, il permettra à toute personne ayant abordé la programmation de parfaire ses connaissances dans les concepts de base de l'informatique.

Nous allons tout d'abord présenter, dans ce polycopié, les principaux outils qui sont utiles à la construction des programmes eux-mêmes. Nous verrons ensuite les relations entre les programmes et les objets externes, c'est-à-dire essentiellement les fichiers. Enfin, nous aborderons quelques unes des fonctionnalités des systèmes d'exploitation qui permettent d'adapter la machine aux véritables besoins des utilisateurs. Notre but essentiel est de faire comprendre l'importance des mécanismes mis en jeu, et leurs conséquences sur l'exécution des applications, quels que soient les langages utilisés pour les écrire. Nous prendrons le point de vue d'utilisateur d'un système d'exploitation et non celui du concepteur. Nous n'hésiterons pas cependant, à donner par endroit une description interne des mécanismes, là où il nous semble que cette description permet de mieux les comprendre.

Ce polycopié est construit sur quatre parties d'importances inégales: introduction, chaîne de production de programmes, environnement externe, et environnement physique. Il est complété par un deuxième polycopié qui contient un ensemble de problèmes et solutions.

Dans la première partie, nous commençons par rappeler rapidement l'évolution historique de l'utilisation des ordinateurs, ce qui permet de mettre en avant les différents concepts ou mécanismes, au fur et à mesure de leur apparition. Nous décrivons ensuite brièvement l'architecture du matériel, pour montrer les difficultés de l'utilisation du matériel “nu”, et justifier ainsi le rôle du système d'exploitation. Nous présentons alors les différents types de systèmes d'exploitation, leurs modes d'utilisation, les critères de choix, et ce qui caractérise un système général ou spécialisé. Cette introduction permet de présenter les services offerts par le système, et se termine par la description de l'architecture d'un système informatique.

La deuxième partie présente la chaîne de production des programmes, qui peut apparaître au programmeur sous forme d'une boîte à outils, ou sous la forme d'un système intégré. Dans tous les cas, elle comporte trois aspects: la traduction, l'édition de liens et des outils complémentaires. Nous présentons brièvement les constituants essentiels de la traduction: l'analyse lexicale, l'analyse syntaxique, l'analyse sémantique, la génération et l'optimisation de code et terminons par la description de la traduction croisée.

L'édition de liens est décrite plus en détail, en introduisant d'abord la notion de module translatable et la notion de lien, en montrant l'expression de la nature du lien dans le module source ainsi que la représentation des liens dans le module objet. L'étude du fonctionnement de l'éditeur de liens montre ensuite comment se passe l'édition simple d'une liste de module, la construction de la table des liens et l'utilisation de la notion de bibliothèque pour ajouter des modules de bibliothèques à la liste. Nous montrons alors que la liaison avec le système doit être traité différemment, par des instructions spéciales. Cet aspect se termine avec la notion de recouvrement, les références croisées et le chargement.

La présentation des autres outils de la chaîne de production permet d'évoquer les outils d'aide à la mise au point, avec, en particulier, la notion de trace, de point d'arrêt, de reprise et de pas à pas, conduisant aux metteurs au point symboliques multi-fenêtres. Elle se poursuit avec les préprocesseurs et les macrogénérateurs. Elle se termine avec le *make*, outil de définition et d'exploitation du graphe de dépendance entre des fichiers. L'idée est de montrer qu'il vaut mieux faire un outil pour réaliser une activité automatisable plutôt que de la faire à la main.

La troisième partie aborde la description des objets externes vus de l'utilisateur. Nous introduisons tout d'abord la notion de fichier logique comme notion abstraite qui permet la manipulation des objets externes par le programme (les fichiers physiques). Suivant le cas, les objets externes ont pour but d'échanger des informations, ou de les mémoriser à long terme. Ceci peut conduire à différentes formes de fichiers. Concrètement, l'implantation des objets externes sur disque peut se présenter soit sous forme d'une suite séquentielle de blocs avec éventuellement des extensions, soit sous forme d'un ensemble de blocs de taille fixe. La représentation de l'espace libre conduit à la notion de quantum, et peut utiliser une table de bits ou une liste des zones libres.

À l'exécution, il faut relier le fichier logique à un objet externe, et pour cela il faut disposer d'un mécanisme de désignation. Ceci conduit à introduire la notion de volume, en tant que désignation du support, et la notion de répertoire, pour désigner l'objet sur le support. L'intérêt de l'uniformisation de la désignation par une arborescence unique est mise en évidence, ainsi que l'utilisation de l'environnement pour abréger cette désignation.

La description des objets externes ne serait pas complète sans une introduction de la protection et de la sécurité. La protection, qui a pour but de définir des règles d'utilisation des opérations sur les objets, est obtenue soit par des droits d'accès soit par des mots de passe. La sécurité, qui a pour but de garantir que les opérations s'exécutent conformément à leurs spécifications même en cas de défaillances, est obtenue par redondance interne ou externe.

Enfin, nous terminons cette partie par une présentation rapide de quelques systèmes de gestion de fichiers courants.

Dans la quatrième partie, nous abordons l'interface du programme avec le système d'exploitation. La notion de processus est décrite, ainsi que la hiérarchie de processus. Cette notion est indissociable de la notion de ressources, et permet d'introduire les différents états d'un processus. Nous abordons ensuite les mécanismes habituels de synchronisation des processus que sont les verrous et les sémaphores, et montrons que le contrôle du respect d'une règle du jeu conduit souvent à proposer aux programmeurs d'application des mécanismes particuliers plus élaborés. Nous montrons alors les conséquences du blocage des processus en attente de ressources sur les temps de réponse, et introduisons la notion d'interblocage et le problème de la famine.

Nous présentons ensuite le partage de la mémoire centrale, en introduisant la notion de multiprogrammation. Nous montrons les limites du partitionnement de la mémoire. La notion de segmentation fournit au programmeur un espace mémoire à deux dimensions lui facilitant la gestion de son propre espace, lorsque cette segmentation est prise en compte par le matériel. La notion de pagination est complémentaire de la notion de segmentation, et permet l'implantation de la notion de mémoire virtuelle. Les principaux algorithmes de pagination à la demande sont alors présentés.

Le rôle du langage de commandes est décrit en conclusion: quelles sont les fonctionnalités et comment s'exécute une commande permettent d'assimiler l'interpréteur de commande à un programme comme un autre. C'est en fait un des constituants de la chaîne de production de programmes, mais qui permet entr'autre la manipulation des objets externes et des processus, ce qui justifie de ne l'aborder qu'à la fin.

Les chapitres des quatre premières parties comportent presque tous un paragraphe “conclusion” qui tente de résumer ce qui nous paraît essentiel dans ce chapitre.

Christian Carrez

# Table des matières

Première Partie: Introduction ... 1

1 Évolution historique de l'utilisation des ordinateurs ... 3

1.1. Les systèmes purement séquentiels (1950-1960) ... 3
1.1.1. Les programmes autonomes (1950) ... 3
1.1.2. Le moniteur d'enchaînement des travaux (1955) ... 4
1.1.3. L'ordinateur spécialisé d'entrées-sorties (1960) ... 5

1.2. L'introduction du parallélisme (1960-1965) ... 6
1.2.1. Les entrées-sorties tamponnées (1960) ... 6
1.2.2. La multiprogrammation (1965) ... 7
1.2.3. Le contrôle de procédés (1965) ... 8

1.3. L'amélioration de l'interface homme-machine (1965-1980) ... 9
1.3.1. Le temps partagé (1965) ... 9
1.3.2. Le transactionnel (1970) ... 10
1.3.3. Les stations de travail (1980) ... 10

1.4. Conclusion ... 11

2 Rappels d'architecture matérielle ... 13

2.1. Architecture générale ... 13

2.2. Architecture du processeur ... 14

2.3. Les entrées-sorties ... 15
2.3.1. Les entrées-sorties programmées ... 15
2.3.2. Les entrées-sorties par accès direct à la mémoire ... 16
2.3.3. Les entrées-sorties par processeur spécialisé ... 17

2.4. Les interruptions ... 17

2.5. Notion d'appel système ... 19
2.5.1. Mode maître-esclave ... 19
2.5.2. Déroutement ... 19

2.6. Les caractéristiques des principaux périphériques ... 19
2.6.1. Les périphériques de dialogue homme-machine ... 20
2.6.1.1. Les écrans claviers distants ... 20
2.6.1.2. Les écrans graphiques ... 20
2.6.1.3. Les imprimantes ... 20
2.6.2. Les périphériques de stockage séquentiel ... 21
2.6.3. Les périphériques de stockage aléatoire ... 22
2.6.4. Les périphériques de communications entre machines ... 22
2.6.5. Les périphériques et le système ... 23

2.7. Conclusion ... 23

3 Généralités sur les systèmes d'exploitation ... 25

3.1. Les différents types de systèmes d'exploitation ... 25

3.1.1. Les modes d'utilisation ... 25
3.1.2. Les critères de choix ... 26
3.1.3. Système général ou spécialisé ... 27

3.2. Les services offerts par le système ... 28
3.2.1. La chaîne de production de programmes ... 28
3.2.2. Le programme et les objets externes ... 29
3.2.3. Le programme et son environnement physique ... 29

3.3. Architecture d'un système ... 29

3.4. Conclusion ... 30

Deuxième Partie: Chaîne de production de programmes ... 31

4 La traduction des langages de programmation ... 33

4.1. L'analyse lexicale ... 34

4.2. L'analyse syntaxique ... 35

4.3. L'analyse sémantique ... 36

4.4. La génération et l'optimisation de code ... 37

4.5. La traduction croisée ... 38

4.6 Conclusion ... 39

5 L'édition de liens et le chargement ... 41

5.1. La notion de module translatable ... 41

5.2. La notion de lien ... 42
5.2.1. Expression de la nature du lien dans le module source ... 43
5.2.2. Représentation des liens dans le module objet ... 44

5.3. Fonctionnement de l'éditeur de liens ... 45
5.3.1. Édition simple de la liste *L* ... 45
5.3.2. Construction de la table des liens ... 46
5.3.3. Notion de bibliothèque ... 47
5.3.4. Adjonction de modules de bibliothèques ... 48
5.3.5. Edition de liens dynamique ... 50

5.4. Notion de recouvrement ... 50

5.5. Les références croisées ... 51

5.6. Le chargement ... 52

5.7. Conclusion ... 52

6 Autres outils de la chaîne de production ... 55

6.1. Les outils d'aide à la mise au point ... 55
6.1.1. La notion de trace ... 55
6.1.2. La notion de point d'arrêt, de reprise et de pas à pas ... 56
6.1.3. Les metteurs au point symboliques ... 56
6.1.4. Les metteurs au point symboliques multi-fenêtres ... 57
6.1.5. Les mesures de comportement dynamique ... 58

6.2. Les préprocesseurs et les macrogénérateurs ... 58

6.3. Le *make* ... 59
6.3.1. Graphe de dépendance ... 60
6.3.2. Exploitation du graphe de dépendance ... 60

6.3.3. Macro substitution ........ 61

6.4. Les outils complémentaires ........ 62

6.5. Conclusion ........ 62

Troisième Partie: Environnement externe ........ 61

7 La notion de fichier ........ 67

7.1. Le but des objets externes ........ 67
7.1.1. Les objets externes comme outils d'échange d'informations ........ 67
7.1.2. Les objets externes comme outils de mémorisation à long terme ........ 68

7.2. La manipulation des objets externes par le programme ........ 69
7.2.1. La notion de fichier ........ 69
7.2.2. Le fichier séquentiel ........ 69
7.2.3. Le fichier séquentiel de texte ........ 70
7.2.4. Le fichier à accès aléatoire ........ 70

7.3. La liaison entre le fichier et l'objet externe ........ 71
7.3.1. L'établissement de la liaison ........ 71
7.3.2. Représentation interne d'un fichier ........ 71

7.4. Conclusion ........ 72

8 Implantation des objets externes sur disque ........ 73

8.1. La linéarisation de l'espace disque ........ 73

8.2. Allocation par zone ........ 74
8.2.1. Implantation séquentielle simple ........ 74
8.2.2. Implantation séquentielle avec extensions fixes ........ 75
8.2.3. Implantation séquentielle avec extensions quelconque ........ 76

8.3. Allocation par blocs de taille fixe ........ 77
8.3.1. Implantation par blocs chaînés ........ 77
8.3.2. Implantation par blocs à plusieurs niveaux ........ 78

8.4. Représentation de l'espace libre ........ 79
8.4.1. Notion de quantum ........ 79
8.4.2. Représentation par table de bits ........ 80
8.4.3. Représentation par liste des zones libres ........ 81

8.5. Conclusion ........ 81

9 La désignation des objets externes ........ 83

9.1. La définition d'une liaison ........ 83

9.2. La notion de volume ........ 84
9.2.1. La structuration d'un volume ........ 84
9.2.2. Le montage de volume ........ 85
9.2.3. La relation entre volume et support ........ 86

9.3. La notion de répertoire ........ 86
9.3.1. Le répertoire sur bande magnétique ........ 86
9.3.2. Répertoire simple sur disque ........ 87
9.3.3. Arborescence de répertoires ........ 87

9.4. Construction d'une arborescence unique à la Unix ........ 88
9.4.1. Les fichiers spéciaux comme périphériques ........ 89
9.4.2. Le montage de volume dans Unix ........ 89
9.4.3. Les fichiers multi-répertoires ........ 90
9.4.4. La désignation par rapport à l'environnement ........ 91

9.5. Conclusion ... 92

10 La sécurité et la protection des objets externes ... 93

10.1. Les mécanismes de protection ... 93
10.1.1. La protection par droits d'accès ... 93
10.1.2. La protection par mot de passe ... 94

10.2. Les mécanismes de sécurité ... 95
10.2.1. La sécurité par redondance interne ... 95
10.2.2. La sécurité par sauvegarde périodique ... 95
10.2.3. Les disques à tolérance de panne ... 97

10.3. Conclusion ... 98

11 Quelques exemples de SGF ... 99

11.1. Les systèmes FAT et VFAT ... 99
11.1.1. Représentation de l'espace ... 99
11.1.2. Les répertoires ... 99

11.2. Les systèmes HFS et HFS Plus de MacOS ... 100
11.2.1. La représentation de l'espace. ... 100
11.2.2. Les répertoires ... 101
11.2.3. Cohérence du SGF ... 102
11.2.4. Le système HFS Plus ... 102

11.3. Le système NTFS ... 103
11.3.1. Les descripteurs de fichiers ... 103
11.3.2. Les Répertoires ... 104
11.3.3. Compression de données ... 104
11.3.4. Sécurité par fichier journal ... 105

11.4. Le système ext2fs de Linux ... 105
11.4.1. La représentation de l'espace ... 106
11.4.2. Les répertoires ... 106
11.4.3. Sécurité ... 107

11.5. Structuration en couche du système ... 107

Quatrième Partie: Environnement physique ... 105

12 La gestion des processus ... 111

12.1. La notion de processus ... 111

12.2. La hiérarchie de processus ... 112
12.2.1. Un nombre fixe de processus banalisés ... 113
12.2.2. La création dynamique de processus ... 113
12.2.3. L'exemple de Unix ... 113

12.3. La notion de ressources ... 114

12.4. Les états d'un processus ... 115

12.5. Conclusion ... 116

13 Synchronisation et communication entre processus ... 119

13.1. Les mécanismes de synchronisation ... 119
13.1.1. Les verrous ... 120
13.1.2. Les sémaphores ... 120
13.1.3. Les mécanismes plus élaborés ... 121

13.2. La communication entre processus ... 122

13.2.1. Le schéma producteur-consommateur ........ 122
13.2.2. Les tubes Unix ........ 123
13.2.3. La communication par boîte aux lettres ........ 124

13.3. Les conséquences sur les temps de réponse ........ 124

13.4. La notion d'interblocage ........ 125

13.5. La notion de famine ........ 127

13.6. Conclusion ........ 128

14 La gestion de la mémoire centrale ........ 129

14.1. La notion de multiprogrammation ........ 129
14.1.1. L'intérêt de la multiprogrammation ........ 129
14.1.2. Les conséquences de la multiprogrammation ........ 130
14.1.3. Les difficultés du partitionnement ........ 131

14.2. La notion de mémoire segmentée ........ 132

14.3. Le mécanisme de pagination ........ 133
14.3.1. La pagination à un niveau ........ 133
14.3.2. La pagination à deux niveaux ........ 135
14.3.3. La segmentation vis à vis de la pagination ........ 136
14.3.4. Le principe de la pagination à la demande ........ 136

14.4. La notion d'espace de travail ........ 138

14.5. Conclusion ........ 138

15 Le langage de commandes ........ 141

15.1. Le langage issu du traitement par lot ........ 141

15.2. Le langage interactif ........ 142
15.2.1. Le message d'invite ........ 142
15.2.2. Un langage orienté verbe ........ 142
15.2.3. La gestion de l'environnement ........ 142
15.2.4. Le traitement préliminaire de la commande ........ 143
15.2.5. L'exécution de la commande ........ 144

15.3. Les structures de contrôle du langage de commandes ........ 144

15.4. Le langage à base de menus ou d'icônes ........ 145

15.5. Conclusion ........ 145

Bibliographie ........ 147

Index ........ 149

# PREMIERE PARTIE

---

# INTRODUCTION

# 1

# Évolution historique de l'utilisation des ordinateurs

A l'origine, les ordinateurs étaient utilisés pour des besoins spécifiques, par une population réduite de personnes qui connaissaient tous les aspects de leur exploitation. Des raisons économiques, d'adéquation aux besoins, d'amélioration de performances, etc..., ont entraîné une évolution de cette exploitation. Cette évolution a conduit à l'introduction de plusieurs concepts successifs. Un rappel historique permet la présentation de ces concepts et leur justification.

## 1.1. Les systèmes purement séquentiels (1950-1960)

### 1.1.1. Les programmes autonomes (1950)

Initialement, les périphériques des ordinateurs se résumaient en un lecteur de cartes, un perforateur de cartes, et une imprimante. Le mode de fonctionnement était assez simple, vu de l'utilisateur, puisque l'exécution d'un programme consistait à mettre dans le lecteur de cartes un paquet contenant la forme “binaire” du programme suivie des données. Après lecture du programme, l'ordinateur en lançait l'exécution, les résultats étant obtenus sous forme d'un paquet de cartes perforées, ou de lignes imprimées (figure 1.1).

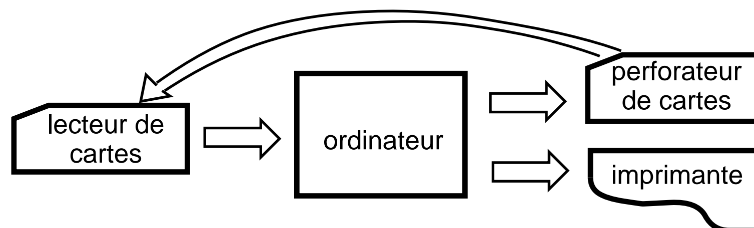



***Fig. 1.1.*** *Fonctionnement en programmes autonomes.*

La constitution de la forme binaire était peut-être un peu compliquée. Il est en effet apparu très vite que l'écriture de programmes devait être aidée de façon à éviter la conception directe en forme binaire. Les langages d'assemblage ou de haut niveau, FORTRAN par exemple, sont apparus très tôt. Le programmeur constituait une suite de cartes contenant le texte source de son programme, dont il obtenait une traduction sous forme binaire au moyen de l'exécution successive de divers programmes standards (les “passes” du compilateur). Ainsi l'exécution d'un programme FORTRAN, perforé sur un paquet de cartes *ps*, pouvait, par exemple, demander les étapes suivantes:

- exécution de la passe 1 du compilateur avec comme données *ps*, et fournissant un paquet de cartes *pi*, forme intermédiaire pour le compilateur,
- exécution de la passe 2 du compilateur avec comme données *pi*, et fournissant un paquet de cartes *pb*, forme binaire du programme,
- exécution de *pb* sur ses données.

A la fin de l'exécution de la passe 2, *pi* peut être jeté, car il ne sert plus à rien.

Le concept de base de ce mode de fonctionnement était la notion d'*amorçage*. La mise en route (ou la réinitialisation) de l'ordinateur lance l'exécution d'un programme cablé de quelques instructions (l'*amorce cablée*), qui met le matériel en position de lecture d'une carte, dont le contenu sera mémorisé dans un emplacement fixe de la mémoire. Dès que cette carte est lue, le matériel exécute l'instruction qui se trouve à cet emplacement fixe. Le contenu de cette première carte doit donc être un court programme qui assure le chargement en mémoire des cartes qui suivent, et le lancement du programme correspondant. D'où le terme d'*amorce logicielle* (*bootstrap* en anglais). Notons que ce concept est toujours utilisé sur les ordinateurs d'aujourd'hui, lors de leur mise en route ou de leur réinitialisation; le seul changement intervenu est de pouvoir amorcer la machine depuis n'importe quel périphérique, et non plus à partir de cartes perforées.

Ce mode de fonctionnement correspondait assez bien à la structure purement séquentielle de l'ordinateur. Le matériel était conçu pour exécuter les instructions du programme sans intervention extérieure possible, en dehors de la réinitialisation. Ainsi la lecture d'une carte commençait lorsque le programme faisait appel à l'instruction correspondante, cette instruction se terminant lorsque la carte avait été lue sans erreur en mémoire.

Ce mode de fonctionnement attire les remarques suivantes:

- Seul l'enchaînement des instructions d'un programme est automatique.
- Le matériel est en général sous employé, du fait des manipulations humaines entre deux activités.
- L'utilisation sous forme de vacations (réservation pendant une certaine durée par un utilisateur unique), accentue le sous-emploi, et entraîne la manipulation de paquets de cartes importants (les passes d'un compilateur) par de nombreuses personnes.
- Les programmes sont écrits pour fonctionner sur une “machine nue”, puisque la totalité du logiciel est chargé par l'amorce. On dit aussi que l'on a un fonctionnement en *autonome* (*stand alone* en anglais).

Notons que le mode d'utilisation des micro-ordinateurs individuels s'apparentent à celui-ci, avec remplacement des cartes par les disquettes. La fonctionnalité a néanmoins été améliorée, et les programmes ne fonctionnent plus tout à fait sur une machine nue.

## 1.1.2. Le moniteur d'enchaînement des travaux (1955)

L'émergence des supports magnétiques (bandes initialement) a tout d'abord permis la conservation des programmes binaires importants sur ce support. Il a fallu alors définir des outils spécifiques de gestion de ces périphériques, pour permettre aux utilisateurs de retrouver facilement ces programmes. Simultanément, pour améliorer la rentabilité des machines, on a imaginé l'introduction du *moniteur d'enchaînement des travaux*, qui est un programme particulier activé automatiquement à la fin de l'exécution de chaque programme utilisateur, et dont le but est d'assurer alors la lecture en mémoire et le lancement du programme utilisateur suivant (figure 1.2).

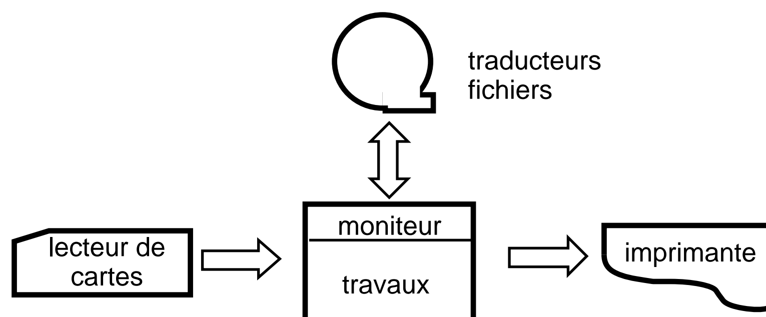


***Fig. 1.2.*** *Moniteur d'enchaînement des travaux.*

Dans ce contexte, l'utilisateur prépare un paquet de cartes représentant son *travail* (*job* en anglais), et qui est constitué d'une ou plusieurs *étapes* (*steps*). Chaque étape correspond à un programme dont l'exécution est demandée par l'utilisateur sur un jeu de données particulier. Ce programme peut être un des utilitaires conservés sur bande, tel qu'un compilateur, ou le résultat d'une étape précédente mémorisé temporairement sur bande.

Le moniteur d'enchaînement joue le rôle du programme de l'amorce logicielle vu en 1.1.1, et assure le chargement et l'exécution successifs de chacun des programmes définis par les étapes des travaux des différents utilisateurs. Pour cela trois concepts ont été introduits:

- Le *langage de commande* est interprété par le moniteur, et permet à l'utilisateur de préciser en quoi consiste le travail. C'est donc le moyen de communication entre l'utilisateur et le moniteur d'enchaînement des travaux.
- La *protection* des données et des instructions du moniteur est nécessaire pour éviter qu'un programme utilisateur ne vienne les détruire.
- Le *superviseur d'entrées-sorties* est un ensemble de sous-programmes résidents en machine, qui assurent le contrôle des opérations d'entrées-sorties des programmes des utilisateurs. Ce contrôle est nécessaire pour garantir que chaque travail d'un utilisateur ne sera pas perturbé par les actions intempestives des autres.

Ce mode de fonctionnement attire les remarques suivantes:

- L'utilisateur n'a plus accès directement à la machine, mais utilise les services d'un opérateur.
- Il y a enchaînement automatique des programmes.
- Le *débit des travaux* (*throughput* en anglais), c'est-à-dire le nombre de travaux par unité de temps, est amélioré.
- Le *temps de réponse*, c'est-à-dire le délai qui sépare le moment où l'utilisateur donne son programme et celui où il obtient le résultat, est augmenté.
- L'utilisateur ne peut plus agir sur son programme durant l'exécution.

### 1.1.3. L'ordinateur spécialisé d'entrées-sorties (1960)

Les opérations d'entrées-sorties relatives aux cartes et aux imprimantes sont, en général, longues par rapport aux performances des machines. Comme le moniteur d'enchaînement impose que ces opérations soient exécutées par le superviseur d'entrées-sorties, au lieu de l'être directement par le programme, celui-ci peut réaliser différemment ces opérations, pourvu que le résultat soit le même pour l'utilisateur. Aussi, pour améliorer la rentabilité de l'ordinateur de traitement, on a imaginé de transférer les cartes sur une bande, en utilisant un ordinateur spécifique et bon marché, et de faire lire cette bande, à la place des cartes, par le moniteur d'enchaînement des travaux. De même, les résultats d'impression sont transférés sur bande dans l'ordinateur principal, et la bande est ensuite lue par l'ordinateur spécifique pour être imprimée (figure 1.3).

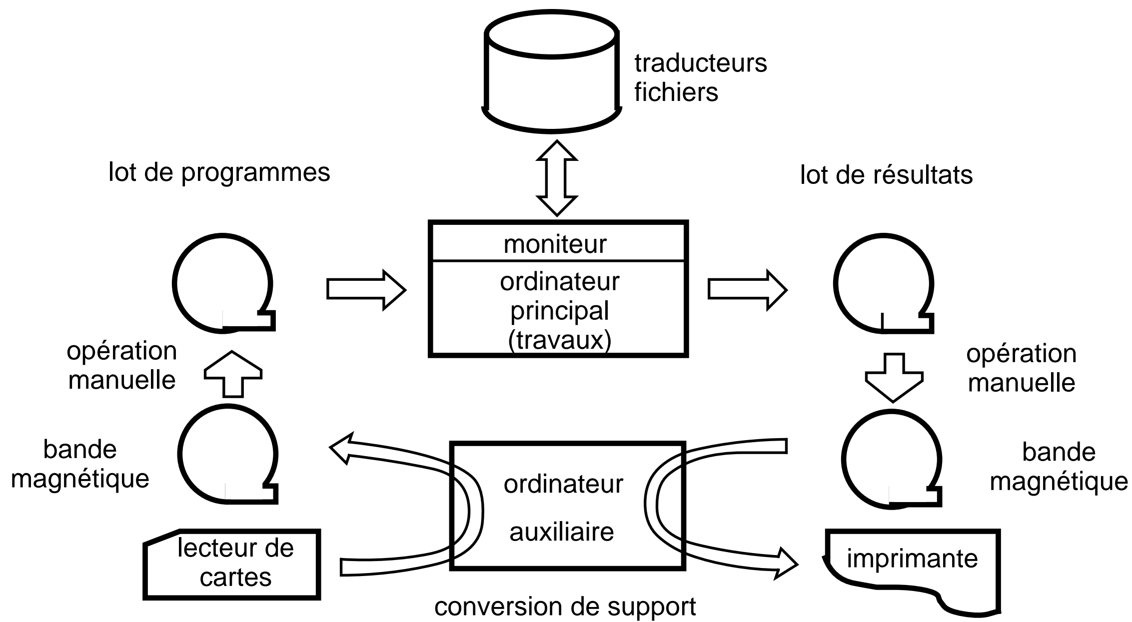



***Fig. 1.3.*** *Traitement par lot avec ordinateur spécialisé d'entrées-sorties.*

Ce mode de fonctionnement attire les remarques suivantes:

- Le débit des travaux est amélioré.
- Le temps de réponse est augmenté, car le transport des bandes n'est effectué que lorsqu'un nombre important de travaux y est mémorisé. On parle de *train de travaux*.

# 1.2. L'introduction du parallélisme (1960-1965)

## 1.2.1. Les entrées-sorties tamponnées (1960)

Il est apparu très vite que la réalisation matérielle d'une opération d'entrées-sorties par le processeur de calcul conduisait à une mauvaise rentabilité de la machine. Les concepteurs du matériel ont donc introduit des processeurs spécialisés qui prenaient en charge ces opérations de façon autonome (voir le chapitre suivant). Il était ainsi possible de poursuivre les traitements pendant l'exécution de l'opération. Ceci a permis de connecter de nouveau les périphériques de type lecteur de cartes ou imprimante sur l'ordinateur principal, et de supprimer l'ordinateur secondaire. Le superviseur d'entrées-sorties assure la lecture des cartes dans une zone dédiée de mémoire centrale, avant que le programme n'en ait effectivement besoin, permettant ainsi de satisfaire immédiatement sa demande ultérieure. De même, lorsque le programme demande une impression, celle-ci est remplacée par une recopie dans une zone dédiée de mémoire centrale, le superviseur assurant l'impression du contenu de cette zone ultérieurement.

Ce mode de fonctionnement n'a été rendu possible que par l'introduction du mécanisme d'*interruption*, permettant à un dispositif extérieur d'arrêter momentanément le déroulement normal d'un programme pour exécuter un traitement spécifique. Nous reviendrons plus en détail sur ce mécanisme dans le chapitre suivant. Par exemple, lorsque le lecteur de cartes a fini le transfert du contenu de la carte dans la mémoire centrale, il le signale au superviseur par le biais d'une interruption. Celui-ci peut alors commander la lecture de la carte suivante dans un autre emplacement mémoire. De même, l'imprimante signale au superviseur, par une interruption, la fin de l'impression d'une ligne. Celui-ci peut alors commander l'impression de la ligne suivante si elle est disponible.

Ce mode de fonctionnement attire les remarques suivantes:

- Le temps de réponse est amélioré, puisqu'il n'est plus nécessaire de remplir une bande pour pouvoir la transférer depuis (ou vers) l'ordinateur secondaire d'entrées-sorties.
- La rentabilité du système est améliorée par la récupération au niveau processeur de traitement des temps des opérations d'entrées-sorties devenues autonomes.

- La réservation de tampons d'entrées-sorties en mémoire centrale est une solution coûteuse (surtout à l'époque où les tailles de mémoire centrale étaient faibles par rapport à celles que l'on trouve aujourd'hui!). L'évolution naturelle est d'étendre ces zones sur disque (figure 1.4).

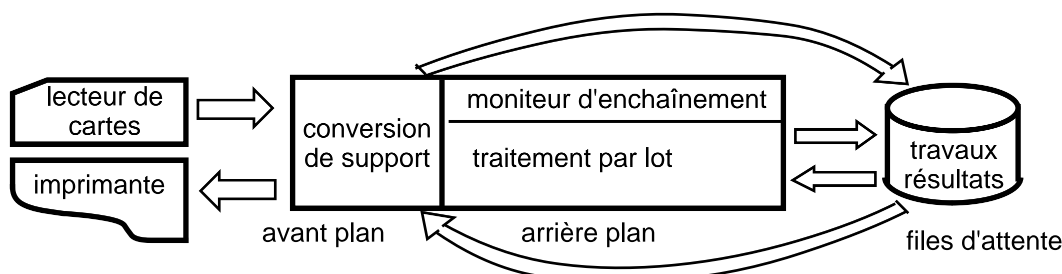



***Fig. 1.4.*** *Entrées-sorties tamponnées.*

## 1.2.2. La multiprogrammation (1965)

Les méthodes vues jusqu'alors, si elles améliorent la rentabilité des machines, atteignent cependant leurs limites. En effet, lorsqu'un programme s'exécute, il n'est pas possible de supprimer complètement les temps des opérations d'entrées-sorties. Le tamponnement, ainsi qu'il a été introduit ci-dessus, n'apporte qu'une solution partielle, du fait de la taille limitée du tampon. Par ailleurs, il n'est guère envisageable pour les périphériques à accès aléatoire tels que les disques. Les temps d'unité centrale laissés disponibles par un programme pendant ses entrées-sorties, peuvent être récupérés par d'autres programmes indépendants s'ils sont également présents en mémoire centrale à ce moment. C'est ce que l'on appelle la *multiprogrammation* (figure 1.5).

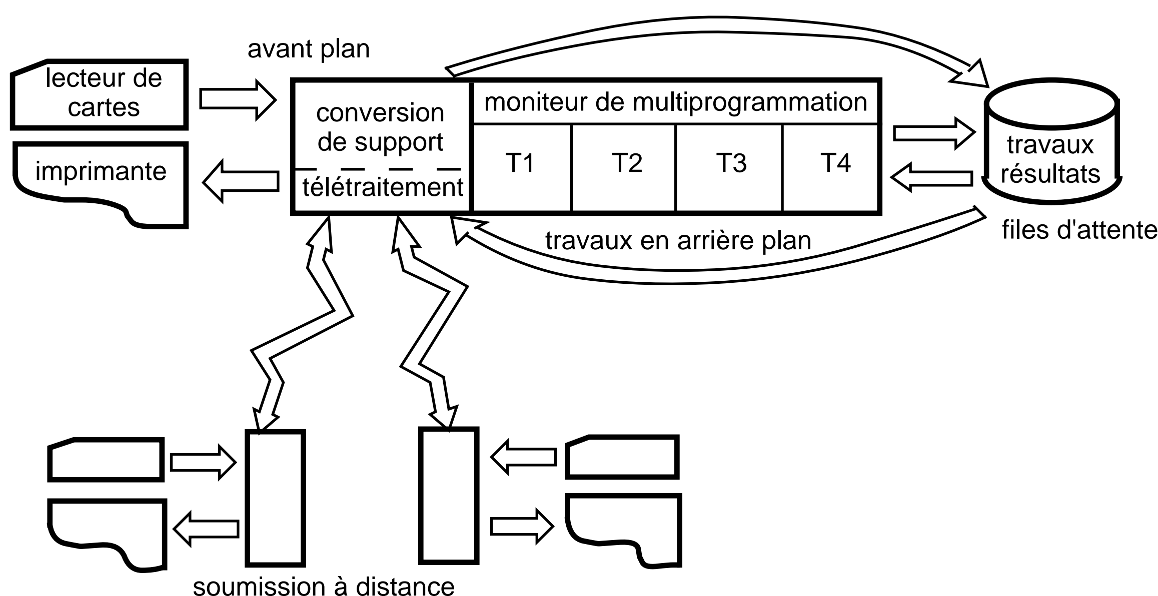


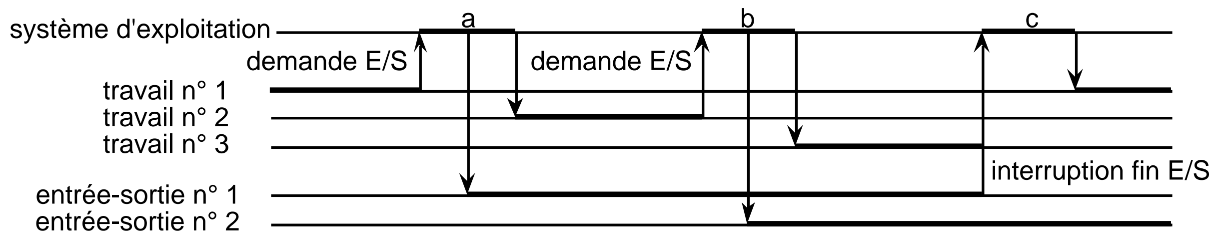

***Fig. 1.5.*** *Moniteur de multiprogrammation.*

Constatons tout d'abord que, pour mettre plusieurs programmes simultanément en mémoire, il faut avoir de la place. La multiprogrammation a pu être développée grâce à l'augmentation de la taille des mémoires centrales conjointement à une diminution considérable de leur coût.

La mise en œuvre de la multiprogrammation nécessite le renforcement de certains mécanismes qui ont déjà été présentés:

- La protection des données et des instructions du moniteur doit maintenant être étendue aux données et instructions d'un programme par rapport aux autres, de façon à permettre l'isolation de chacun des utilisateurs dans un univers partagé.
- Le superviseur d'entrées-sorties doit contrôler l'accès aux ressources de la machine, et assurer la gestion de ces ressources (allocation, contrôle, restitution).

- Le mécanisme d'interruption est au centre de l'allocation du processeur. Signalant la fin d'une opération d'entrées-sorties, l'interruption a en général pour effet de permettre au programme demandeur de poursuivre son exécution (figure 1.6).
- La notion de *système d'exploitation* apparaît avec la nécessité de gérer l'ensemble du matériel et du logiciel mis à la disposition d'utilisateurs simultanés.




En a, le système d'exploitation prend le contrôle pour lancer l'entrée-sortie demandée par le travail n° 1, et donner l'unité centrale au travail n° 2. Il en va de même en b pour l'entrée-sortie demandée par le travail n° 2. Lors de la fin de l'entrée-sortie n° 1, une interruption donne le contrôle au système d'exploitation qui peut en conséquence relancer le travail n° 1 qui était en attente.

***Fig. 1.6.*** *Fonctionnement simplifié de la multiprogrammation.*

Ce mode de fonctionnement attire les remarques suivantes:

- L'amélioration de la rentabilité de la machine est obtenue en la partageant entre plusieurs programmes simultanés, conduisant à une meilleure utilisation de l'ensemble des ressources.
- Il est possible de s'abstraire de la séquentialité originelle des travaux. Le système assure le transfert sur disque des programmes dès qu'ils sont disponibles. Lorsqu'il y a de la place libre en mémoire, il choisit l'un des travaux sur disque en fonction de paramètres variés, et non plus dans l'ordre d'arrivée. Il est ainsi possible de diminuer le temps de réponse pour les travaux courts.
- Il est possible de déporter les organes d'entrées-sorties du type cartes et imprimantes, en les reliant à l'ordinateur central par des lignes téléphoniques spécialisées. Les travaux qui sont entrés par ce moyen sont traités de la même façon que les autres, les résultats étant délivrés par le système sur l'imprimante déportée.

## 1.2.3. Le contrôle de procédés (1965)

Les ordinateurs peuvent aussi être spécialisés pour le contrôle de procédés. Par analogie avec ce qui précède, les interruptions vont permettre de déclencher des programmes spécifiques de lecture d'informations sur des capteurs, de mémorisation de ces valeurs, ou d'envoi de commandes à l'extérieur. Lorsque aucune activité de ce type (travaux en *avant plan* ou *foreground* en anglais) n'est à exécuter, l'ordinateur peut être utilisé pour des travaux en *arrière plan* (*background* en anglais). Les travaux en avant plan correspondent aux activités du superviseur vues au paragraphe précédent, alors que les travaux en arrière plan correspondent aux travaux des utilisateurs (figure 1.7).

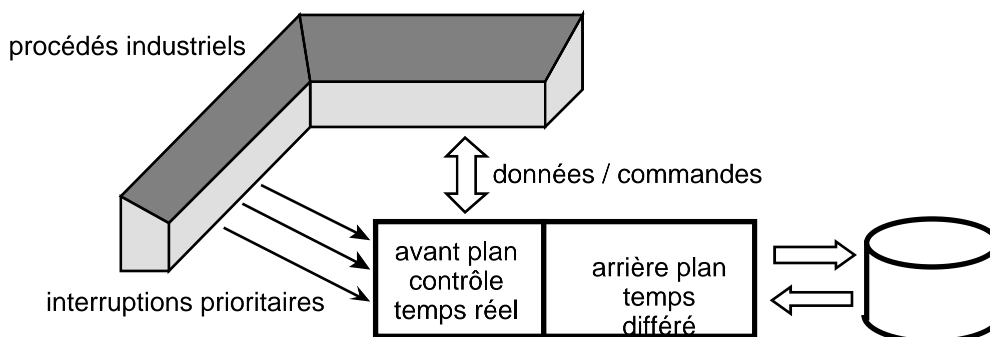



***Fig. 1.7.*** *Contrôle de procédés industriels.*

# 1.3. L'amélioration de l'interface homme-machine (1965-1980)

## 1.3.1. Le temps partagé (1965)

L'évolution qui ressort des étapes ci-dessus conduit à un éloignement entre l'utilisateur et la machine. Il était naturel que l'on cherche à retrouver certaines fonctionnalités intéressantes de l'utilisation en portes ouvertes, sans en avoir les inconvénients. Constatant qu'un utilisateur est satisfait s'il passe beaucoup plus de temps à réfléchir au travail qu'il va demander qu'à attendre le résultat de ce travail, ce temps de réflexion peut être mis à profit par l'ordinateur pour répondre aux besoins des autres utilisateurs. L'ensemble des ressources de la machine est partagé entre un ensemble d'utilisateurs, chacun d'eux ayant l'impression qu'il a la machine pour lui tout seul (figure 1.8).

La conception d'un système en temps partagé a nécessité l'introduction de l'horloge temps réel, en plus des mécanismes déjà vus.

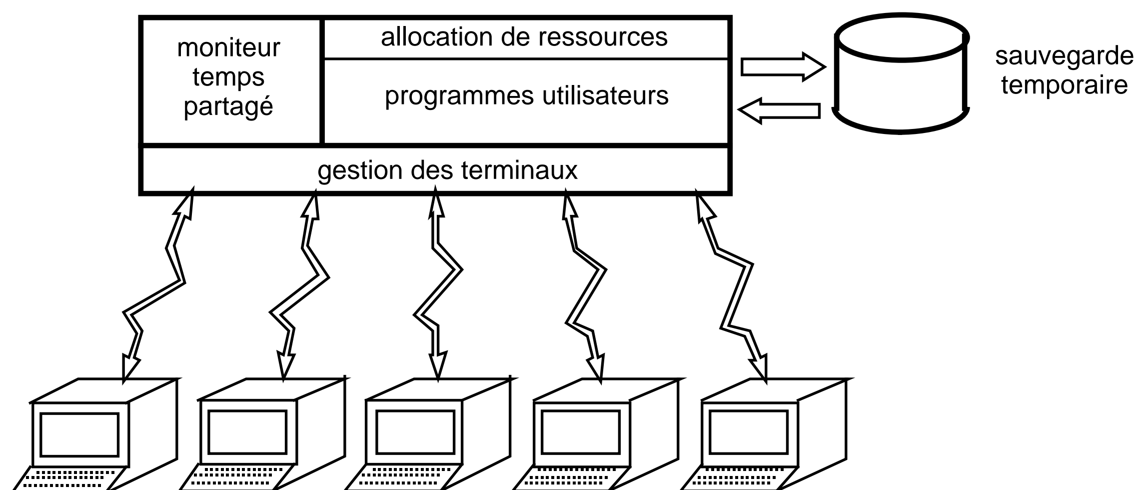



***Fig. 1.8.*** *Système en temps partagé.*

- L'horloge temps réel est un dispositif matériel externe qui provoque des interruptions à des instants réguliers. Ces interruptions sont au centre de l'allocation du processeur, et permettent de répartir équitablement le temps du processeur entre les différents programmes utilisateurs.
- Le mécanisme des interruptions et la notion de superviseur permet la gestion d'un ensemble de terminaux.
- La multiprogrammation permet d'optimiser l'utilisation de l'unité centrale pendant les opérations d'entrées-sorties des programmes des utilisateurs.

En réalité, c'est un peu plus complexe, dans la mesure où il n'est pas raisonnable de laisser un programme d'un utilisateur en mémoire pendant que cet utilisateur réfléchit, car ce temps de réflexion est souvent de l'ordre de 30 secondes. En général il est préférable de mémoriser ce programme sur disque pendant ce temps, et de le rappeler en mémoire lorsque nécessaire. Ceci a été rendu possible par des techniques de *va-et-vient* (*swapping* en anglais) ou de *mémoire virtuelle*.

Par ailleurs le grand nombre d'utilisateurs ayant des activités voisines implique qu'un même programme peut être en cours d'exécution pour plusieurs utilisateurs à la fois (un compilateur par exemple). Ceci a conduit à la conception de *programmes réentrants*: un seul exemplaire du programme capable de traiter en même temps plusieurs jeux de données.

## 1.3.2. Le transactionnel (1970)

Le temps partagé s'adresse plutôt à l'utilisateur qui désire concevoir, mettre au point et utiliser des programmes. Le développement des terminaux a conduit à s'intéresser aux utilisateurs qui désirent exploiter des applications spécifiques orientées vers la saisie et la consultation d'informations conservées dans des fichiers. Il ne s'agit plus alors de mettre à leur disposition une machine, mais de leur permettre d'effectuer sur des données, qu'ils ont mises en commun, des opérations

préprogrammées qui demandent un temps d'unité centrale assez bref. Ceci conduit à gérer l'ordinateur et les terminaux de façon spécifique pour améliorer les temps de réponses vus de l'utilisateur.

Par rapport aux concepts évoqués jusqu'alors, le *partage d'informations* est ici fondamental, alors qu'il était secondaire dans les modes précédents. Ce partage peut couvrir les données proprement dites, mais aussi les instructions qui décrivent les opérations. Pour gérer ce partage et garder la cohérence de ces données partagées, on introduit la notion de *transaction*.

De plus la conservation des données, quoi qu'il arrive, est aussi exigée par les utilisateurs. Il faut donc mettre en œuvre des mécanismes de *sauvegarde* appropriés.

### 1.3.3. Les stations de travail (1980)

L'abaissement des coûts des matériels a permis la conception d'ordinateurs individuels dont les performances avoisinnent celles des ordinateurs de moyenne puissance des années 1970-1980. Ceci n'a pas eu pour conséquence la remise en cause des concepts de base des systèmes d'exploitation. L'utilisateur, habitué à une fonctionnalité fournie sur les gros systèmes, désire retrouver cette fonctionnalité agrémentée d'une meilleure convivialité sur sa machine personnelle.

D'un point de vue général, le poste de travail a d'abord été conçu de façon autonome, utilisé par une seule personne. En ce sens, le système lui-même avait des fonctionnalités simplifiées proches de celles des premiers systèmes. L'exemple type est MSDOS. Les performances de mémoires centrales ont ensuite permis de définir directement par logiciel et point par point, le contenu de l'image qui est affichée à l'écran 25 fois par secondes. Les fonctionnalités du système se sont alors naturellement étendues vers les aspects graphiques et l'interactivité au moyen de la souris, avec l'apparition des premiers MacIntosh au début des années 1980. Ceux-ci étaient déjà organisés en un petit réseau local permettant le partage de fichiers et d'imprimantes entre les utilisateurs. Par la suite, les recherches poursuivies au MIT ont conduit à la sortie d'un produit X-Window, à la fin des années 1980, qui permettait la séparation entre le programme de gestion de l'écran graphique situé sur une machine et celui de l'application elle-même de l'utilisateur situé sur une autre machine, la coopération entre les deux programmes passant par le réseau, selon un mode *client-serveur*. On appelle ainsi le mode de fonctionnement où l'un des programmes, appelé client, ici l'application, demande un traitement spécifique (service) à l'autre, appelé le serveur, ici la gestion de l'écran.

La généralisation des postes de travail en réseau et l'interconnexion des réseaux ont conduit au *world wide web* ou toile mondiale, où un programme, appelé *navigateur*, installé sur le poste de travail de l'utilisateur, permet à celui-ci d'obtenir, depuis des serveurs situés n'importe où, des informations qui sont mises en page par le navigateur.

Deux aspects importants découlent de ces utilisations:

- L'*interface homme-machine* doit permettre une grande facilité d'utilisation par un non-spécialiste. La puissance de calcul locale permet de satisfaire les besoins d'affichage.
- La *communication* entre ordinateurs distants est une nécessité, pour assurer le partage d'un ensemble de ressources matérielles et logicielles entre les différentes stations de travail. Elle conduit à la conception de *systèmes répartis*, construits autour de *réseaux locaux*.

## 1.4. Conclusion

☞ L'amorce (bootstrap) est un petit programme qui est chargé en mémoire en une seule opération d'entrée lors de la mise en route de l'ordinateur, et dont l'exécution permet le chargement et le lancement d'un programme plus important.

☞ Un programme autonome (stand alone) est un programme capable de s'exécuter sur une machine nue.

☞ Un moniteur d'enchaînement des travaux est un programme spécifique qui assure, à la fin d'exécution d'un programme utilisateur, le chargement et l'exécution du suivant de façon automatique.

☞ Le langage de commande permet à l'utilisateur de définir la suite des programmes dont il veut voir l'exécution.

☞ Le superviseur d'entrées-sorties est un ensemble de sous-programmes qui assure le contrôle et la bonne exécution des entrées-sorties pour le compte des programmes des utilisateurs.

☞ Le mécanisme d'interruption est le mécanisme qui permet à un organe matériel externe d'interrompre le déroulement normal du processeur pour lui demander d'exécuter un travail spécifique et bref.

☞ La multiprogrammation consiste à mettre plusieurs programmes en mémoire centrale au même moment de façon à pouvoir exécuter l'un d'entre eux pendant que les autres sont en attente de la fin d'une entrée-sortie.

☞ La protection mémoire est un dispositif matériel qui permet d'empêcher un programme d'accéder en mémoire à des données qui ne lui appartiennent pas.

☞ Le temps partagé consiste à partager dans le temps les ressources de la machine entre les utilisateurs, de telle sorte que chacun d'eux ait l'impression d'avoir la machine pour lui seul.

☞ Le système d'exploitation est le logiciel chargé de gérer l'ensemble du matériel et du logiciel à la disposition des utilisateurs et qu'ils se partagent.

☞ Le débit des travaux est le nombre moyen de travaux exécutés par la machine en un temps donné.

☞ Le temps de réponse est le délai qui sépare le moment où l'utilisateur soumet sa commande et le moment où il obtient le résultat.

# 2

# Rappels d'architecture matérielle

Pour comprendre le rôle et les fonctionnalités des systèmes d'exploitation, il est nécessaire d'appréhender ce qu'est une “machine nue”, c'est-à-dire quels sont les constituants matériels d'un ordinateur, et quels en sont les principes de fonctionnement.

## 2.1. Architecture générale

En première approche, un ordinateur est constitué d'un processeur qui effectue les traitements, d'une mémoire centrale où ce processeur range les données et les résultats de ces traitements et de périphériques permettant l'échange d'informations avec l'extérieur. Tous ces constituants sont reliés entre eux par l'intermédiaire d'un *bus*, qui est l'artère centrale et leur permet de s'échanger des données (figure 2.1.). Pratiquement tous les ordinateurs actuels ont cette architecture, que ce soient les micro-ordinateurs personnels ou les gros ordinateurs des entreprises. Les différences résident essentiellement dans les performances des constituants.

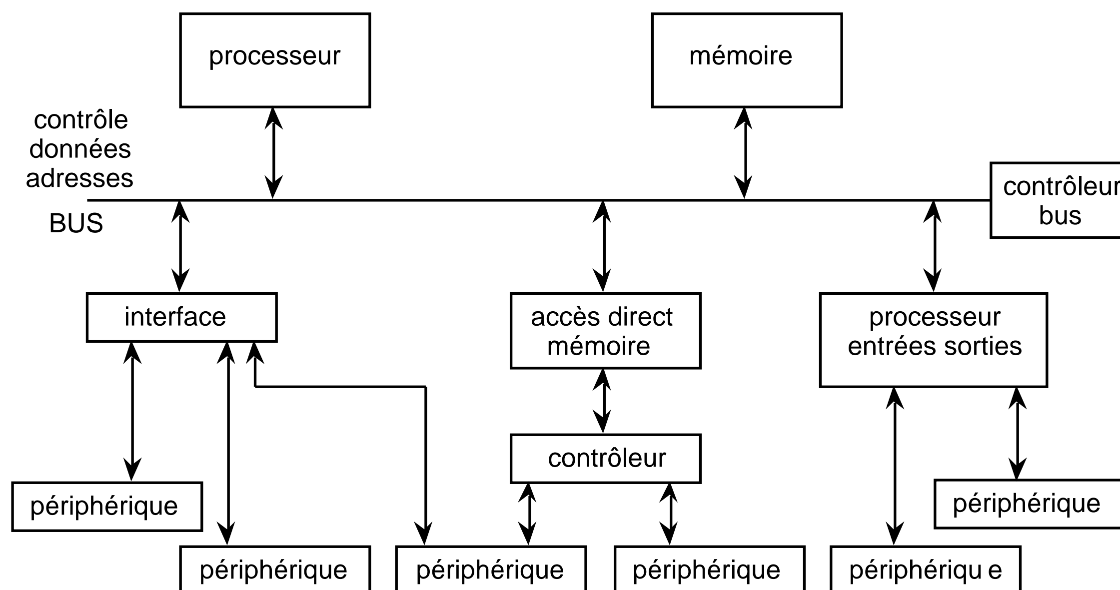



***Fig. 2.1.*** *Architecture générale d'un ordinateur.*

La mémoire est un organe passif, qui répond à des ordres indiqués par les fils de contrôle du bus. En réponse à un ordre d'écriture, elle range la valeur représentée par les fils de données du bus dans un emplacement défini par les fils d'adresse du bus. En réponse à un ordre de lecture, elle fournit sur

les fils de données du bus la valeur mémorisée à l'emplacement défini par les fils d'adresses. Le nombre de fils de données du bus définit le nombre de bits des emplacements mémoire. C'est une caractéristique importante pour les performances de l'ordinateur, puisqu'il détermine le nombre de bits pouvant être lus ou écrits en mémoire par une seule opération. Le nombre de fils d'adresse du bus définit la taille maximale de la mémoire centrale. Le bus est géré par un contrôleur, parfois intégré au processeur, qui empêche son utilisation simultanée par plusieurs organes.

## 2.2. Architecture du processeur

Le processeur est l'organe qui effectue les traitements suivant un “algorithme” défini par le programmeur. Il est constitué essentiellement de trois parties (figure 2.2):

- L'unité arithmétique et logique (U. A. L.) est capable d'effectuer les opérations élémentaires habituelles sur des valeurs binaires, telles que l'addition, la soustraction, le “ou” logique, les décalages, etc...
- Les registres permettent de mémoriser des résultats intermédiaires ou des états particuliers du processeur. Ils sont en général en petit nombre, mais d'accès très rapide. Certains ont un rôle particulier, comme l'*accumulateur*, le *compteur ordinal* ou le registre *instruction*.
- Le décodeur-séquenceur contrôle l'exécution des différentes phases des instructions.

Le principe de fonctionnement est assez simple. Le décodeur-séquenceur répète indéfiniment la séquence d'opérations suivante:

- lecture mémoire à l'adresse indiquée par le compteur ordinal, et rangement du résultat dans le registre instruction,
- décodage de cette instruction pour en exécuter les différentes phases.

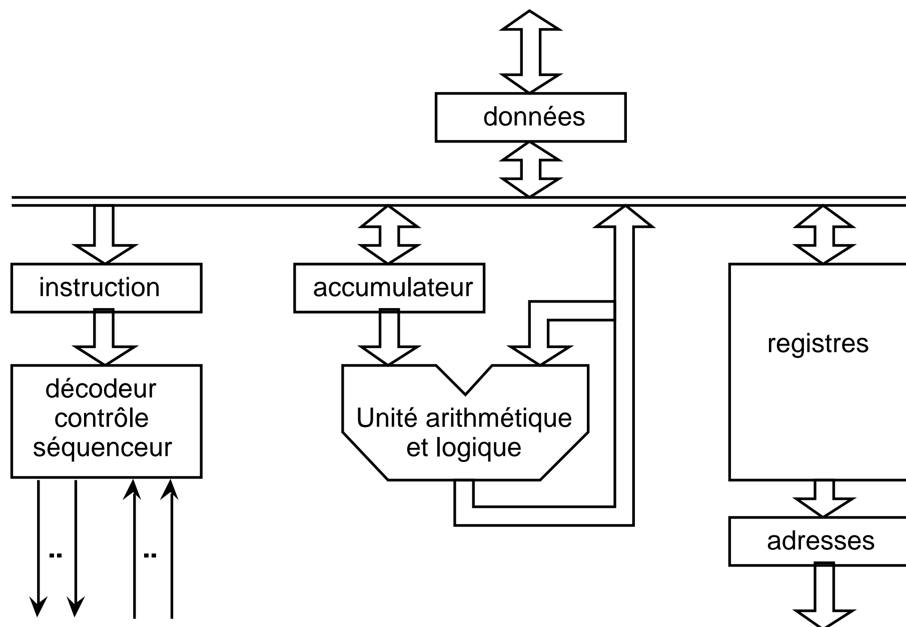



***Fig. 2.2.*** *Architecture générale d'un processeur.*

Les instructions sont en général assez rudimentaires. Ce sont essentiellement des opérations de transfert de données entre les registres et l'extérieur du processeur (mémoire ou périphérique), ou des opérations arithmétiques ou logiques avec un ou deux opérandes. Pour ces dernières opérations, un registre particulier, l'accumulateur, est souvent utilisé implicitement comme l'un des opérandes et comme résultat. En général le déroulement de l'instruction entraîne l'incrémentation du compteur ordinal, et donc l'exécution de l'instruction qui suit. Notons que le transfert d'une valeur dans le compteur ordinal entraîne un “branchement” à l'adresse correspondant à cette valeur. La tendance naturelle a été de construire des processeurs avec un jeu d'instructions de plus en plus large; on pensait alors que le programmeur utiliserait les instructions ainsi disponibles pour améliorer l'efficacité de ses programmes. Ceci a conduit à ce que l'on a appelé l'architecture *CISC* (*Complex Instruction Set Computer*). Cependant on a constaté que les programmes contenaient toujours les

mêmes instructions, une partie importante du jeu d'instructions n'étant utilisée que très rarement. Une nouvelle famille de processeurs a alors été construite, l'architecture *RISC* (*Reduced Instruction Set Computer*), qui offre un jeu réduit d'instructions simples mais très rapides.

La figure 2.3 donne un exemple de la représentation binaire d'une instruction en machine (cas 68000). Cette forme n'est pas très agréable pour l'homme, mais c'est le véritable langage de la machine. La forme mnémonique qui l'accompagne est déjà plus lisible, mais nécessite un programme (l'*assembleur*) pour pouvoir être interprêtée par la machine.

| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

MOVE.W D1,(A2)

transfert dans le registre D1 de la valeur située en mémoire, et dont l'adresse est dans le registre A2.

***Fig. 2.3.*** *Exemple d'instruction binaire et mnémonique.*

# 2.3. Les entrées-sorties

Le principe élémentaire mis en œuvre pour l'échange de données entre deux constituants physiques est représenté en figure 2.4. En dehors des données proprement dites, deux liaisons supplémentaires sont nécessaires, pour permettre d'une part à l'émetteur de la donnée de signaler la présence effective de cette donnée sur les fils correspondants, et d'autre part au récepteur de signaler qu'il a lu la donnée. On peut comparer ceci au ping-pong: celui qui a la balle a le droit d'accéder aux fils de valeur pour lire ou écrire la donnée. Suivant la capacité de traitement que l'on place entre le processeur et le périphérique, c'est-à-dire suivant la complexité du dispositif qui va prendre en compte cet échange élémentaire et le transformer en des échanges sur le bus, on trouvera les dispositions évoquées dans la figure 2.1.

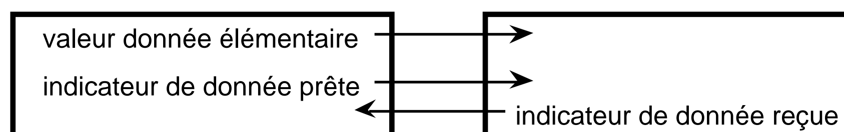



***Fig. 2.4.*** *Protocole élémentaire d'entrées-sorties.*

## 2.3.1. Les entrées-sorties programmées

La façon la plus simple d'assurer la liaison entre le bus et un périphérique, est de faire une simple adaptation des signaux évoqués ci-dessus. On parle alors d'une *interface*. Le processeur adresse directement le périphérique soit par les instructions habituelles d'accès à la mémoire centrale, l'interface jouant alors le rôle d'un (ou de plusieurs) emplacement de mémoire, soit par des instructions spécialisées qui assurent le transfert d'une donnée élémentaire avec un registre ou un emplacement mémoire. Dans tous les cas, le programmeur doit assurer le protocole élémentaire d'échanges évoqué plus haut; c'est pourquoi on parle d'entrées-sorties programmées.

```
tantque il_y_a_des_données_à_lire faire
     tantque donnée_suivante_non_prête faire fait; { attente de la donnée}
     lire_la_donnée;
     traitement_de_la_donnée;
fait
```

***Fig. 2.5.*** *Exemple d'entrée-sortie programmée.*

La figure 2.5 donne le schéma d'un tel programme de lecture. Noter la boucle d'attente de la donnée qui teste l'indicateur de donnée prête. Nous supposons ici que la lecture de la donnée entraîne le positionnement par l'interface de l'indicateur de donnée lue et sa remise à zéro par le positionnement de l'indicateur de donnée prête. Il est facile de constater qu'un tel échange a une vitesse (on dit encore un *débit*) limité par le nombre d'instructions machine qui constituent le corps de la boucle externe. Ce débit est souvent limité en conséquence à 50 Ko/s (Kilo-octets par seconde). Par ailleurs si le périphérique est lent, le processeur est monopolisé pendant toute la durée de l'échange. Dans ce cas, on ne lit que quelques octets à la fois pour éviter cette monopolisation. Comme il a été mentionné dans le précédent chapitre, cette forme d'échange était la seule possible dans les premières générations de machines.

## 2.3.2. Les entrées-sorties par accès direct à la mémoire

Pour accroître le débit potentiel des entrées-sorties, et diminuer la monopolisation du processeur dont nous avons parlé ci-dessus, la première solution a été de déporter un peu de fonctionnalité dans le dispositif qui relie le périphérique au bus, de façon à lui permettre de ranger directement les données provenant du périphérique en mémoire dans le cas d'une lecture, ou d'extraire directement ces données de la mémoire dans le cas d'une écriture. C'est ce que l'on appelle l'*accès direct à la mémoire*, ou encore le *vol de cycle*.

La figure 2.6 schématise le mécanisme d'accès direct à la mémoire. L'exécution d'un transfert se déroule de la façon suivante:

- Le processeur informe le dispositif d'accès direct à la mémoire de l'adresse en mémoire où commence le tampon qui recevra les données s'il s'agit d'une lecture, ou qui contient les données s'il s'agit d'une écriture. Il informe également le dispositif du nombre d'octets à transférer.
- Le processeur informe le contrôleur des paramètres de l'opération proprement dite.
- Pour chaque donnée élémentaire échangée avec le contrôleur, le dispositif demande au processeur le contrôle du bus, effectue la lecture ou l'écriture mémoire à l'adresse contenue dans son registre et libère le bus. Il incrémente ensuite cette adresse et décrémente le compteur.
- Lorsque le compteur atteint zéro, le dispositif informe le processeur de la fin du transfert par une ligne d'interruption (voir plus loin).

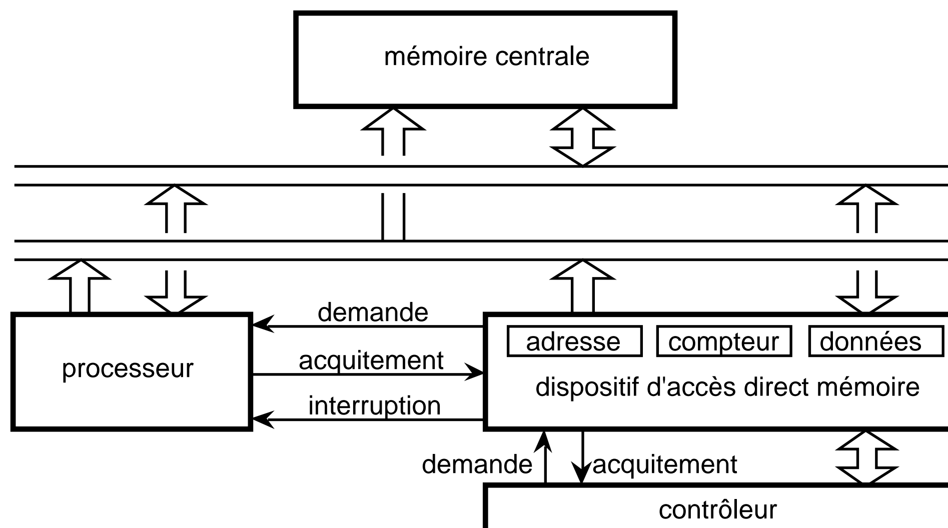



***Fig. 2.6.*** *Mécanisme d'accès direct à la mémoire.*

Pendant toute la durée du transfert, le processeur est libre d'effectuer un traitement quelconque. La seule contrainte est une limitation de ses propres accès mémoire pendant toute la durée de l'opération, puisqu'il doit parfois retarder certains de ses accès pour permettre au dispositif d'accès direct à la mémoire d'effectuer les siens (d'où le terme de vol de cycle évoqué plus haut).

La limitation du débit inhérent au mécanisme est en général lié au débit potentiel de la mémoire elle-même, c'est-à-dire à la durée d'un cycle de lecture ou d'écriture en mémoire. Si cette durée est de 500 ns., on peut théoriquement atteindre 2 Mo/s. Si la mémoire est organisée en mots de 4 octets, il est possible de lire ou d'écrire 4 octets en un seul cycle, et de porter ainsi le débit à 8 Mo/s. Dans ce cas le dispositif d'accès direct à la mémoire assure le regroupement ou l'éclatement des données échangées avec le périphérique de façon à minimiser les accès mémoire.

## 2.3.3. Les entrées-sorties par processeur spécialisé

La troisième façon de relier un périphérique avec la mémoire est d'utiliser un processeur spécialisé d'entrées-sorties, c'est-à-dire de déléguer plus d'automatisme à ce niveau. Dans le schéma précédent, le processeur principal avait en charge la préparation de tous les dispositifs, accès direct à la mémoire, contrôleur, périphérique, etc..., pour la réalisation de l'opération. L'utilisation d'un processeur spécialisé a pour but de reporter dans ce processeur la prise en charge de cette

préparation, mais aussi des opérations plus complexes, telles que par exemple le contrôle et la reprise d'erreurs. L'intérêt est de faire exécuter des tâches de bas niveau par un processeur moins performant, et donc moins coûteux à réaliser, tout en réservant le processeur principal à des tâches plus nobles.

## 2.4. Les interruptions

Nous avons déjà introduit ce mécanisme dans le chapitre précédent ainsi que dans le paragraphe ci-dessus. Ce mécanisme a été imaginé pour permettre à un dispositif extérieur d'interrompre le déroulement normal du processeur pour lui faire exécuter un traitement spécifique. Un programme en arrière plan peut ainsi être interrompu pour permettre d'activer un travail en avant plan; il sera ensuite poursuivi à la fin de ce dernier. Un périphérique peut signaler la fin de l'opération d'entrées-sorties demandée précédemment, permettant ainsi au système de temps partagé de reprendre l'exécution du programme qui avait demandé cette opération.

Le mécanisme est obtenu en modifiant légèrement le fonctionnement du décodeur-séquenceur du processeur, par introduction d'une troisième phase, et qui devient le suivant:

- lecture mémoire à l'adresse indiquée par le compteur ordinal, et rangement du résultat dans le registre instruction,
- décodage de cette instruction pour en exécuter les différentes phases,
- s'il y a une demande d'interruption, alors la prendre en compte.

La prise en compte de l'interruption peut se faire de différentes façons. Puisqu'il s'agit d'interrompre le déroulement normal des instructions, il “suffit” de modifier le compteur ordinal. Cependant comme on désire reprendre ultérieurement le programme interrompu, il faut aussi “sauvegarder” la valeur de ce compteur. Dans la plupart des processeurs, ce n'est pas jugé tout à fait suffisant. En général, le constructeur définit un *PSW* (*Program Status Word*), ou *mot d'état du programme*, contenant en particulier le compteur ordinal ainsi que certains indicateurs sur l'état courant du processeur. La prise en compte d'une interruption par le processeur consiste alors à ranger en mémoire ce PSW dans un emplacement déterminé ou repéré par un registre particulier utilisé comme un pointeur de pile[1], et à charger un nouveau PSW soit fixe, soit obtenu depuis la mémoire.

Pour éviter de ranger inutilement en mémoire des registres, ce qui serait coûteux en temps, le processeur ne range que le minimum lors de la prise en compte de l'interruption. Si on désire pouvoir reprendre ultérieurement le programme interrompu, les instructions qui constituent le traitement de l'interruption que nous appellerons *sous-programme d'interruption*, doit ranger en mémoire le contenu de l'ensemble des registres du processeur qui ne l'ont pas été par la prise en compte de l'interruption, pour permettre cette reprise ultérieure. C'est ce que l'on appelle la *sauvegarde du contexte*. Dans certains cas, on pourra ne faire qu'une sauvegarde partielle des seuls registres modifiés par le sous-programme d'interruption. Lorsque le contexte est sauvegardé, le sous-programme analyse la cause de l'interruption, et effectue le traitement approprié. Il peut ensuite restituer le contexte du programme interrompu (ou celui d'un autre programme lorsque l'interruption correspond à la fin d'une opération d'entrées-sorties pour celui-ci). Il se termine par une instruction spéciale qui demande au processeur de restituer le PSW du programme interrompu, entraînant sa reprise effective (figure 2.7).

[1] Une pile est une structure de données permettant la conservation d'informations, de telle sorte que les informations sont enlevées de la structure dans l'ordre inverse où elles y ont été mises.

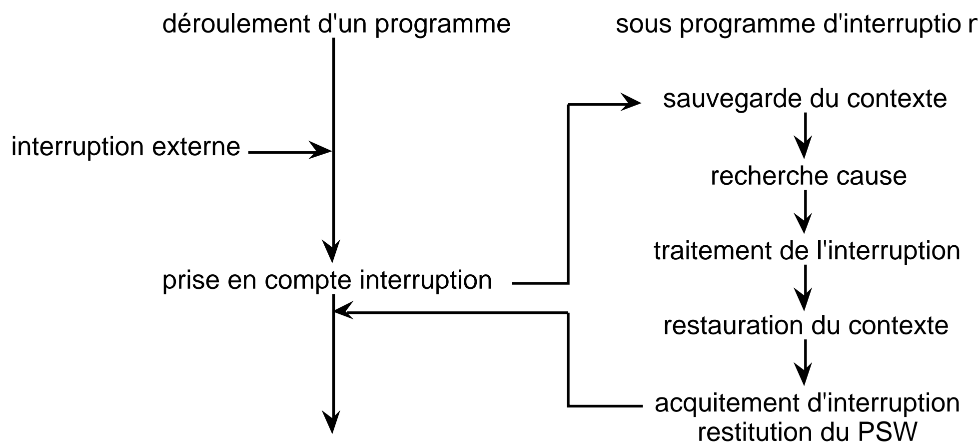



***Fig. 2.7**. Déroulement d'un sous-programme d'interruption.*

Le schéma de fonctionnement du mécanisme d'interruption ainsi décrit met en avant une difficulté si une nouvelle interruption survient alors que le traitement de la précédente n'est pas terminée. Si on ne prend pas de précaution, le sous-programme d'interruption est de nouveau appellé avant sa fin, entraînant la sauvegarde du contexte du sous-programme d'interruption et la perte du contexte du programme initialement interrompu[2]. Ce problème est résolu par le biais du *masquage des interruptions*, qui interdit au processeur la prise en compte d'une interruption si elle est masquée. Cela n'annulle pas la demande d'interruption proprement dite, qui n'est pas considérée par le processeur. Lorsque, ultérieurement, une instruction sera exécutée par le processeur pour démasquer cette interruption, celui-ci pourra alors la prendre en compte. Cela permet ainsi au logiciel (en général le système d'exploitation) de contrôler les moments où une telle prise en compte pourrait perturber son bon fonctionnement. En particulier, le masquage automatique des interruptions par le processeur lors de la prise en compte d'une interruption évite de le réexécuter une nouvelle fois avant qu'il ne soit terminé, à moins que le programmeur n'ait lui-même demandé le démasquage des interruptions dans le sous-programme après avoir pris les précautions nécessaires à cette nouvelle interruption (*sous-programme réentrant*).

Certains processeurs ont plusieurs “fils d'interruption” qui sont masquables individuellement. Ils correspondent alors à des interruptions hiérarchisées, c'est-à-dire qu'à chaque fil est attribué un numéro appellé *niveau de priorité d'interruption*, permettant au processeur de savoir laquelle il y a lieu de prendre en compte si plusieurs sont présentes au même moment. Ce dispositif est complété par un registre spécialisé du processeur (qui fait partie du PSW) qui contient le niveau de priorité de l'interruption en cours de traitement. Le processeur ne prend en compte une interruption que si elle est non masquée, et de priorité supérieure au niveau de priorité courant. Cela permet de satisfaire les demandes plus urgentes que celle que l'on est en train de traiter tout en évitant les problèmes de réentrance évoqués plus haut.

# 2.5. Notion d’appel système

## 2.5.1. Mode maître-esclave

Le chapitre précédent a introduit la nécessité de contrôler les actions d’un programme vis à vis de son environnement direct. En général, il est nécessaire d’empêcher le programme d’un utilisateur de perturber le fonctionnement global du système ou les programmes des autres utilisateurs. Cela implique qu’il n’ait pas accès à l’ensemble des ressources de la machine, mais seulement à celles qui lui ont été allouées en propre. La solution la plus couramment adoptée est de distinguer deux modes de fonctionnement du processeur, le mode *maître* et le mode *esclave*. Dans le mode maître, le processeur a accès à toutes les ressources de la machine. Dans le mode esclave, certaines instructions lui sont interdites. Pour cela, le fonctionnement du décodeur-séquenceur du processeur est légèrement modifié, par introduction d'une phase de contrôle intermédiaire:

[2] À moins d'utiliser pour la sauvegarde des contextes successifs, mais se pose alors le problème de la taille de cette pile.

- lecture mémoire à l'adresse indiquée par le compteur ordinal, et rangement du résultat dans le registre instruction,
- si le processeur est en mode esclave, vérification que l'instruction est autorisée,
- décodage de cette instruction pour en exécuter les différentes phases,
- s'il y a une demande d'interruption, alors la prendre en compte.

L'indicateur du mode de fonctionnement maître/esclave fait partie du mot d'état programme. De cette façon, ce mode est lié au programme en cours d'exécution. En particulier, la prise en compte d'une interruption chargeant un mot d'état programme spécifique à l'interruption, le sous programme correspondant s'exécutera dans le mode défini par ce mot d'état et non par celui du programme interrompu. La reprise de ce dernier restaurera le mode qui lui est attaché.

### 2.5.2. Déroutement

Si le programme d'un utilisateur fonctionne en mode esclave, sous un jeu d'instructions réduit, néanmoins, certaines actions nécessaires à son bon fonctionnement nécessitent de disposer du jeu complet, et doivent donc être exécutées en mode maître. Évidemment le changement de mode doit être contrôlé. En général, ceci se fait au moyen d'une instruction spéciale, qui va changer le mot d'état programme du processeur, selon un principe voisin de la prise en compte des interruptions, c'est-à-dire, rangement du mot d'état courant et chargement du mot d'état lié à la fonction demandée et restauration du mot d'état initial à la fin de la fonction. C'est pourquoi, ce mécanisme prend parfois la dénomination d'*interruption programmée*. On dit encore *déroutement* ou *appel superviseur*.

Un déroutement est aussi très voisin d'un appel de sous programme : sauvegarde du compteur ordinal, dont la valeur est remplacée par l'adresse du sous programme, et restauration de la valeur sauvegardée à la fin du sous programme. Il y a, cependant, deux différences essentielles :

- Le déroutement modifiant le mot d'état programme permet un changement de mode, un masquage des interruptions, ou un changement du niveau de priorité courant du processeur.
- L'adresse du sous programme exécuté n'est pas contenue dans l'instruction de déroutement elle-même, mais dans le mot d'état programme chargé. Ceci oblige le passage par des points de contrôles qui vérifieront les valeurs des paramètres de la fonction avant de l'exécuter. De plus, le programme n'a pas à connaître les adresses qui sont à l'intérieur du système et aux quelles il n'a pas accès.

## 2.6. Les caractéristiques des principaux périphériques

Les exemples suivants montrent la grande diversité des périphériques habituels.

### 2.6.1. Les périphériques de dialogue homme-machine

Trois types de périphériques principaux sont concernés par le dialogue entre l'homme et la machine.

#### *2.6.1.1. Les écrans claviers distants*

Il s'agit de terminaux relativement simples, qui ont vocations à être installés loin de l'ordianteur. La liaison doit donc être à faible coût, et on utilise 2 ou 4 fils, sur lesquels les bits sont "sérialisés", c'est-à-dire que les 8 bits d'un octet sont transmis les uns après les autres, à vitesse constante, suivant un protocole standard international (RS232) ils sont précédés par un bit de départ (*start*), et suivis d'un ou deux bits de fin (*stop*). C'est ce que l'on appelle la *ligne série asynchrone*. Elle est asynchrone, car chaque octet est transmis individuellement, le bit start permettant la synchronisation automatique du récepteur sur l'émetteur, pour la durée du transfert de l'octet correspondant.

Vu de l'ordinateur, un écran-clavier est donc en fait une ligne série. Les octets sont en général transmis un par un par des entrées-sorties programmées. Le débit en sortie est au plus de 1000 octets par seconde. Il ne sert à rien d'aller plus vite, puisque l'homme ne peut déjà plus lire le contenu de l'écran à cette vitesse. Le débit en entrée est en général beaucoup plus lent puisqu'il est

limité par la frappe manuelle sur le clavier, et donc quelques octets par seconde. Pour éviter que le processeur ne perde trop de temps lors de ces entrées-sorties programmées, on utilise le mécanisme d'interruption pour chaque caractère transmis. Si l'ordinateur doit comporter beaucoup de lignes, il peut s'en suivre une charge de traitement importante due à la prise en compte de ces interruptions. Dans ce cas, les constructeurs proposent souvent des processeurs d'entrées-sorties spécialisés dans ce travail; on les appelle alors des *frontaux*.

### *2.6.1.2. Les écrans graphiques*

Les écrans graphiques sont maintenant le mode d'affichage habituel des postes de travail individuels, qu'il s'agisse de poste bureautique ou de stations de travail haut de gamme. Une zone de mémoire dédiée contient l'image en point à point devant être affichée. Un processeur graphique spécialisé parcours cette mémoire et interprète la suite des emplacements comme décrivant les points successifs ou *pixels* de l'image, et les transforme en signal vidéo pour l'affichage. Ce balayage est effectué 25 fois par secondes. De plus, cette zone est également accessible par le processeur central qui peut donc définir le contenu de chaque pixel de l'écran. Comme un écran peut comporter jusqu'à 1300 lignes de 1600 points chacune, l'image peut donc être constituée d'environ 2 millions de pixels, chaque pixel étant défini généralement par trois couleurs (rouge, vert et bleu), le niveau de chacune d'elles étant défini par 8 bits. Une telle mémoire peut donc atteindre 6 Mo. Le processeur central n'a pas à redéfinir tout ou partie du contenu de ces 6 Mo, 25 fois par secondes, mais chaque fois qu'un changement intervient dans l'image à afficher à l'utilisateur. Dans le cas de la nécessité d'animation complexe, il peut être fait appel à un processeur spécialisé pour faire varier le contenu de cette mémoire.

Les écrans graphiques sont en général couplés à ce que l'on appelle une *souris*. Il s'agit en fait d'un dispositif très simple qui transmet les mouvements subis dans deux directions, ainsi que l'appui sur 1 à 3 boutons. Ces informations de mouvements peuvent être analysés par un logiciel spécialisé, qui en retour affiche une marque à une position de l'écran, permettant à l'utilisateur de voir l'effet de ses mouvements sur la souris. Ce logiciel doit être assez rapide pour que tout mouvement de l'utilisateur ait un effet visuel immédiat. Les programmes d'applications reçoivent en général les coordonnées de la position de la marque sur l'écran au moment de l'appui sur l'un des boutons.

### *2.6.1.3. Les imprimantes*

L'imprimante est un autre type de périphérique de dialogue homme-machine très courant. On peut distinguer les imprimantes caractères et les imprimantes graphiques.

Nous mettons dans la catégorie des imprimantes caractères celles à qui on envoie les caractères à imprimer et qui ont peu de fonctionnalités de mise en page. Une telle imprimante peut être connectée sur une ligne série comme les terminaux écran-clavier, et a alors les mêmes caractéristiques vues du processeur. L'intérêt est d'une part la standardisation vue de l'ordinateur qui ne fait pas de distinction entre ces imprimantes et les terminaux, et d'autre part la possibilité de les éloigner de l'ordinateur pour les rapprocher de l'utilisateur. Elle peut également être reliée par une interface directe en mode caractère, c'est-à-dire en utilisant les entrées-sorties programmées vues plus haut, la boucle assurant le transfert d'une ligne. Elle peut enfin être reliée sur un dispositif d'accès direct à la mémoire. Dans ces deux derniers cas, le débit instantané peut être important, car l'imprimante dispose en général d'un tampon correspondant à une ligne, ce qui justifie ces deux modes de liaison. Il faut noter cependant que l'impression proprement dite ralentit le débit moyen, qui varie entre 100 octets par seconde et 5000 octets par seconde (2400 lignes/minute).

Les imprimantes graphiques, par exemple les imprimantes laser, offrent des fonctionnalités complexes de dessin de caractères ou de figures comme de mise en page. Les plus simples doivent recevoir le contenu de la page à imprimer sous la forme d'une suite de bits décrivant les points (noir ou blanc) des lignes successives. Cependant, avec une finesse de 300 points par pouce, une page contient environ 8 millions de points, décrits par 1 Mo, qui doivent être transmis pendant le défilement de la feuille de papier devant le faisceau laser. Aussi cette méthode a tendance à être abandonnée au profit d'une méthode plus élaborée, dans la quelle le contenu de la page est décrit dans un langage standardisé, comme *postscript*. L'imprimante dispose d'un processeur rapide qui interprète cette description et la traduit en points sur le papier. Ce processeur peut être doté de plus ou moins de mémoire pour stocker les descriptions de page à imprimer et les dessins correspondants

à plusieurs fontes (une fonte est un assortiment de caractères). Par ailleurs, la disponibilité d'un processeur permet de connecter l'imprimante directement au réseau.

Notons que pour les écrans claviers et les imprimantes classiques, le débit en sortie n'est pas crucial en ce sens que si le débit est ralenti du fait du processeur, il n'y aura pas perturbation du fonctionnement, mais simplement perte de performance. Le débit en entrée des claviers doit être respecté sous peine de perte d'informations, mais ce débit est assez faible pour ne pas être trop contraignant, sauf, comme nous l'avons dit, dans le cas d'un nombre important de terminaux. Nous verrons que ces contraintes sont beaucoup plus fortes pour les autres types de périphériques.

### 2.6.2. Les périphériques de stockage séquentiel

Nous nous intéressons ici aux bandes magnétiques de 1/2 pouce[3] de large, soit 1.27 cm., et d'environ 2400 pieds de long, soit 730 mètres. L'enregistrement et la lecture sont obtenus par 9 têtes permettant de mettre transversalement un octet avec un bit de parité. Le transfert entre l'ordinateur et la bande ne peut s'effectuer que lorsqu'elle défile à vitesse constante, en général 75 ips (inches par seconde), soit 190.5 cm/s, mais peut atteindre 200 ips, soit 5 m/s. La *densité*, exprimée en nombre d'octets par pouce (bpi pour byte per inch), déterminera d'une part la capacité totale de la bande, d'autre part le débit du transfert. Les densités les plus courantes actuellement sont, d'une part 1600 bpi, donnant une capacité totale de 44 Mo, et un débit de 120 Ko/s, d'autre part 6250 bpi, donnant une capacité totale de 172 Mo, et un débit de 500 Ko/s, mais pouvant atteindre 1.25 Mo/s. La vitesse de défilement devant être constante pendant le transfert, il faut mémoriser les informations par bloc de taille variable, les blocs successifs étant séparés par un espace inutilisable (*gap*) réservé au freinage et à l'accélération. Cet espace est en moyenne de 3/4 de pouce, soit 1.905 cm.

Constatons que les débits interdisent les entrées-sorties programmées. Par ailleurs le débit doit impérativement être respecté sous peine d'écritures d'informations erronées si l'ordinateur ne fournit pas un octet à temps en écriture, ou de perte d'informations si l'ordinateur ne mémorise pas un octet à temps en lecture. C'est pourquoi on utilise les entrées-sorties par accès direct à la mémoire ou par processeur spécialisé. Ce dernier offre l'avantage de permettre d'assurer toutes les reprises d'erreurs qui pourraient se produire, en recommençant éventuellement plusieurs fois l'opération.

### 2.6.3. Les périphériques de stockage aléatoire

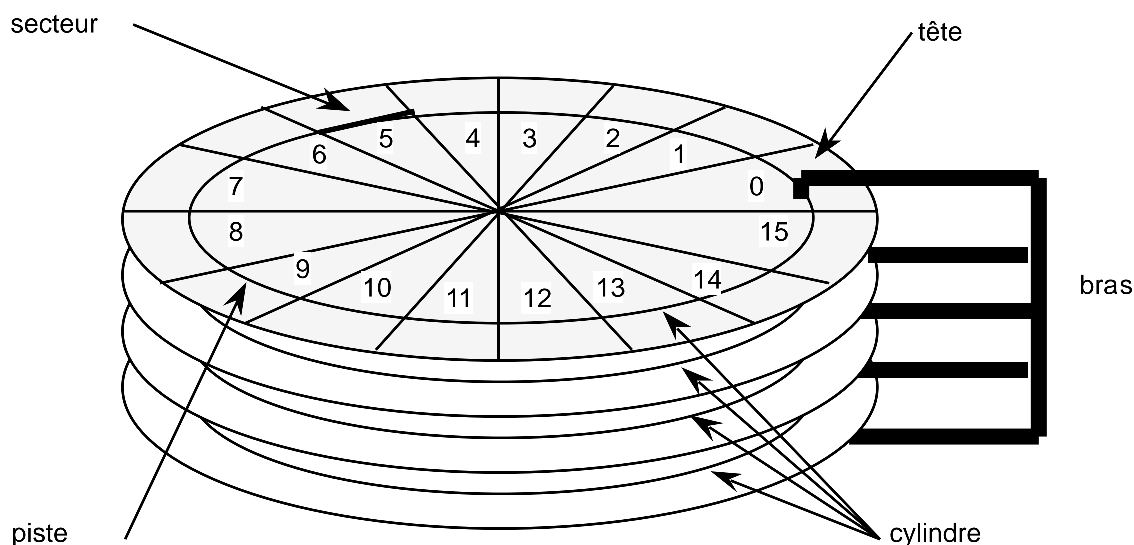



***Fig. 2.8.*** *Disque magnétique.*

Il s'agit essentiellement des disquettes, ou disques magnétiques. Bien des caractéristiques s'appliquent également aux disques optiques, mais nous n'en parlerons pas. Un disque est constitué d'un ou plusieurs plateaux tournant à vitesse constante autour de leur axe (on parle parfois d'une *pile*

---

[3] Rappelons que le pouce (*Inch*) est une mesure de longueur anglaise, qui vaut 2.54 centimètres, tout comme le pied, qui vaut 12 pouces, soit 30.48 centimètres.

*de disque*). Chaque plateau est recouvert sur ses deux faces d'une couche magnétique. Des têtes de lecture/écriture sont fixées, à raison de une par face, sur un bras qui peut se déplacer perpendiculairement à l'axe. Pour une position du bras donnée, chaque tête peut lire ou écrire sur une circonférence du plateau qui la concerne, délimitant ainsi ce que l'on appelle une *piste*. L'ensemble des pistes vues par chaque tête, pour une position de bras donnée, s'appelle un *cylindre*. Enfin chaque piste est découpée en *secteurs*, qui contiennent chacun le même nombre d'octets (figure 2.8).

La structure même du disque impose que les opérations se fassent par secteurs complets. Par ailleurs, le débit est constant pour un disque donné, et est actuellement compris entre 250 Ko/s et 5 Mo/s. Comme il doit impérativement être respecté, il est nécessaire d'utiliser les entrées-sorties par accès direct à la mémoire ou par processeur spécialisé. Contrairement aux bandes magnétiques, ce dernier n'est pas essentiel pour la récupération d'erreurs qui sont peu fréquentes (et en général plus irrémédiables) sur les disques, mais peut prendre en charge une partie de la gestion du disque lui-même et des accès.

Les disquettes ont la même structure que les disques durs, mais le débit est beaucoup plus faible, puisqu'il est de l'ordre de 30 Ko/s. Ce débit est donc parfois compatible avec le mode d'échange programmé. Il faut noter cependant que lors de la lecture ou de l'écriture d'un secteur, le délai qui sépare le lancement de l'opération du transfert du premier octet, est aléatoire, et compris entre 0 et 300 ms, temps pendant lequel le transfert programmé impose au processeur de rester en boucle d'attente.

## 2.6.4. Les périphériques de communications entre machines

Lorsqu'on veut faire communiquer entre elles plusieurs machines, différentes solutions sont possibles. Nous nous intéressons ici aux communications entre machines hétérogènes et à distance moyenne ou longue. En effet un constructeur peut construire des dispositifs particuliers, entre deux accès directs à la mémoire, ou même partager la mémoire entre les machines, pour échanger des informations entre des machines de sa fabrication qui seraient situées dans la même pièce.

La communication entre machines sur des distances dépassant par exemple 100 mètres, pose d'abord le problème du coût du cable. Comme dans le cas des liaisons avec les écrans-claviers, la connexion est effectuée en utilisant des cables à 2 ou 4 fils. Cette communication pose ensuite le problème du “langage” utilisé, et que l'on appelle le *protocole de communication*. Initialement, chaque constructeur avait son propre protocole, ne permettant que la communication entre ses machines. Sous la pression des utilisateurs, et de divers organismes, des protocoles variés ont été normalisés par des instances internationales. Nous ne tentons ici que de donner quelques caractéristiques importantes de ces communications.

La notion de protocole consiste à encadrer l'information à transmettre, c'est-à-dire le contenu, par des informations qui construisent le protocole, et servent à véhiculer le contenu. Dans la description de la ligne série asynchrone, nous avons vu un exemple de protocole simple où l'octet était encadré par un bit start et des bits stops. Dans la communication entre machines les protocoles sont plus complexes, de façon à permettre des échanges complexes d'informations.

La première caractéristique est la transmission par *paquet*. L'ensemble des octets représentant l'information à transmettre est tout d'abord découpé en morceau, par exemple 250 octets. Chaque morceau est encadré par les informations du protocole permettant de le véhiculer, constituant un paquet. Les différents paquets sont transmis individuellement. A la réception des paquets, le récepteur en extrait le contenu, et utilise les informations du protocole pour reconstruire l'information. Intuitivement, le protocole doit permettre d'identifier l'émetteur, le récepteur, et un numéro d'ordre du paquet.

La deuxième caractéristique est la transmission du paquet à vitesse constante. On dit que la transmission est *synchrone*, car d'une part les horloges de l'émetteur et du récepteur sont synchronisées entre elles, d'autre part l'émetteur commence par transmettre en début de paquet une information permettant au récepteur de reconnaître le premier bit réel du paquet, et de faire le regroupement des bits des octets 8 par 8.

La troisième caractéristique est la complexification de l'ensemble de la transmission entre deux machines qui résulte de l'abandon des liaisons point à point par des lignes spécialisées au profit de

l'interconnexion globale de l'ensemble des machines de la planète. Ceci a conduit à la définition du modèle OSI (Open Systems Interconnexion) qui découpe la communication en 7 couches, chacune d'elle étant chargée d'une tâche spécifique, comme par exemple, la couche réseau chargée du routage des paquets ou la couche transport chargée de segmenter les messages en paquet lors de l'émission et de recomposer les messages à partir des paquets lors de la réception.

Les débits actuels sont importants, puisqu'ils sont couramment de 200 Ko/s, mais ils peuvent atteindre plusieurs dizaines de Mo/s, sur des réseaux locaux comme sur des réseaux à longue distance. Il est évident que de tels débits nécessitent d'utiliser soit l'accès direct à la mémoire, soit le processeur spécialisé d'entrées-sorties, la complexité des protocoles et leur normalisation lui faisant souvent préférer ce dernier.

### 2.6.5. Les périphériques et le système

Il ressort de l'étude des caractéristiques des différents périphériques une très grande diversité de fonctionnement, de mode de liaison, de vitesse de transfert. Programmer sur une machine nue implique la connaissance de cette diversité, et la spécialisation du programme à une configuration particulière. L'un des rôles essentiels du système d'exploitation, et de son constituant particulier le superviseur d'entrées-sorties, est de prendre en compte cette diversité pour en décharger le programmeur. Le système fournira au programmeur un nombre limité d'*interfaces standards et uniformes*.

## 2.7. Conclusion

☞ Un ordinateur est constitué d'un bus qui relie les divers organes, d'un processeur, d'une mémoire centrale et de périphériques.

☞ Un processeur est constitué d'une unité arithmétique et logique, de registres et d'un décodeur-séquenceur qui le contrôle. Il lit en mémoire l'intruction située à l'adresse définie par le compteur ordinal, l'exécute, puis recommence. Son langage est binaire.

☞ Les entrées-sorties programmées sont obtenues par un programme qui effectue le transfert des informations entre la mémoire et un périphérique, une par une.

☞ L'accès direct à la mémoire est un dispositif qui assure le transfert des informations entre la mémoire et un périphérique par vol de cycle.

☞ La prise en compte d'une interruption par le processeur consiste à ranger en mémoire le mot d'état programme et à le remplacer par un nouveau. Le compteur ordinal fait partie de ce mot d'état programme. Le reste du contexte doit être sauvegardé par le sous-programme d'interruption, pour permettre sa restitution en fin d'exécution de ce sous-programme.

☞ Le masquage des interruptions est le mécanisme qui permet au logiciel de contrôler la prise en compte des interruptions par le processeur.

☞ Le processeur peut travailler selon le mode maître ou le mode esclave dans lequel certaines instructions sont interdites. Des instructions spéciales d'appel superviseur permettent de changer de mode de façon contrôlée.

☞ Les périphériques de dialogue homme-machine, lorsqu'ils sont simples, peuvent être gérés par des transferts programmés, éventuellement régis par interruption. Cependant lorsque leur nombre augmente sur une même machine, la charge de traitement peut nécessiter l'utilisation de processeurs spécialisés appelés frontaux. Par ailleurs, dans les stations de travail, l'affichage est le résultat de modifications du contenu de la mémoire partagée par le dispositif de balayage vidéo.

☞ Les périphériques magnétiques, bandes ou disques, nécessitent l'utilisation soit de l'accès direct à la mémoire, soit d'un processeur spécialisé d'entrées-sorties. Ce dernier a l'avantage de permettre de décharger le processeur principal d'une partie de la gestion du périphérique.

☞ La communication entre machines utilise des protocoles normalisés, qui permettent de s'affranchir des contraintes matérielles et prendre en compte la complexité du réseau

d'interconnexion. L'information est découpée par l'émetteur en paquets indépendants, et reconstruite par le récepteur à l'aide du protocole.

☞ L'une des fonctions du système est de décharger le programmeur du souci de la diversité des périphériques, par la fourniture d'un nombre limité d'interfaces standards et uniformes.

# 3

# Généralités sur les systèmes d'exploitation

Le premier chapitre a montré l'évolution qui s'est produite sur l'utilisation de l'ordinateur, et l'introduction progressive de programmes généraux permettant de contrôler son utilisation. Cette évolution a été parfois mal ressentie par les utilisateurs qui voyaient souvent les contraintes et non les avantages. Ceux-ci ont “espéré” de l'abaissement des coûts du matériel la suppression du système d'exploitation. Le deuxième chapitre a mis en évidence la complexité du matériel, ainsi que sa diversité. Il ne viendrait à l'esprit de personne, actuellement, de demander la suppression du système d'exploitation. Il s'agit de le rendre apte à répondre aux véritables besoins des utilisateurs et de privilégier les services dont ces derniers ont besoin.

## 3.1. Les différents types de systèmes d'exploitation

Lors de la phase de conception d'un système d'exploitation, il est nécessaire de faire de nombreux choix, choix d'algorithmes, choix de politique de gestion, choix de paramètres, etc... Ces choix sont guidés par la nature de l'utilisation du système, et conduisent à des solutions de compromis.

### 3.1.1. Les modes d'utilisation

Le premier chapitre a montré plusieurs modes d'utilisation de l'ordinateur.

- Le mode *interactif* correspond au cas où l'utilisateur dialogue avec l'ordinateur: il lance des commandes dont la durée d'exécution est brève, et dont il désire une réponse en quelques secondes, pour pouvoir décider de la suivante, après un temps de réflexion plus ou moins long. Ce mode se subdivise en plusieurs sous modes suivant les applications:
  - Le mode *temps partagé*: l'utilisateur doit pouvoir utiliser toutes les fonctionnalités de l'ordinateur, comme s'il l'avait pour lui tout seul. Il s'agit surtout de lui permettre de créer, tester, modifier, et exécuter des programmes quelconques.
  - Le mode *transactionnel*: l'utilisateur exécute des programmes prédéfinis, pour consulter ou mettre à jour des données partagées entre de nombreux utilisateurs.
  - Le mode *conception assistée*: l'utilisateur se sert de toute la puissance de l'ordinateur pour l'aider dans un travail complexe de conception. L'aspect graphique est souvent ici une caractéristique essentielle, et est une des raisons du besoin de puissance de calcul, le résultat d'une commande étant une modification d'un dessin sur l'écran.
- Le mode *différé* correspond au traitement par lot. Dans ce mode, l'utilisateur fournit un travail dont il viendra chercher le résultat quelques heures après. Le travail peut être conséquent, et nécessiter de disposer de plusieurs ressources pendant un temps important, comme par exemple, du temps de processeur central, de l'espace mémoire ou des bandes magnétiques.

- Le mode *temps réel* correspond au contrôle de procédés industriels. L'ordinateur a en charge de prendre des mesures au moyen de capteurs externes, pour surveiller un procédé, et lui envoyer en retour des commandes pour en assurer la bonne marche. Ces commandes doivent être envoyées dans des intervalles de temps qui sont imposés par le procédé lui-même, pour garantir son bon fonctionnement.

### 3.1.2. Les critères de choix

Les critères qui permettent de caractériser l'adéquation du système au mode de fonctionnement désiré, et donc d'induire les choix lors de la conception du système peuvent s'énoncer de la façon suivante:

- Le *temps de traitement* d'une commande pourrait se définir comme le temps que mettrait l'ordinateur pour exécuter la commande s'il ne faisait que cela. En général, ce temps est décomposé en un temps de processeur et un temps d'entrées-sorties. Il est normalement assez bref dans le mode interactif (inférieur à la seconde, par exemple); il est souvent connu dans le sous mode transactionnel, mais il est a priori imprévisible dans les sous modes temps partagé et conception assistée. Dans le mode différé, ce temps peut être beaucoup plus important, de l'ordre de plusieurs dizaines de minutes, mais sa valeur maximale est souvent définie par le demandeur. Enfin, dans le mode temps réel, il est en général très bref, quelques dizaines de millisecondes, et connu, de façon assez précise.
- Le *temps de réponse* est le délai qui sépare l'envoi de la commande par le demandeur, de la fin d'exécution de la commande. Ce temps ne peut, évidemment, être inférieur au temps de traitement. Ce n'est pas une conséquence de la demande seule, mais le résultat du comportement du système. C'est l'un des critères de jugement de l'utilisateur, et donc une contrainte que doit respecter le système pour satisfaire ce dernier. Si, par nature, un temps de réponse de plusieurs heures est acceptable pour le mode différé, par contre l'utilisateur, en mode interactif, attend normalement des temps de réponse de l'ordre de la seconde. Le sous-mode conception assistée peut parfois impliquer des temps de réponse beaucoup plus bref (suivi de souris par exemple). Le mode temps réel impose des contraintes souvent strictes sur le temps de réponse qui peut être aussi bien de quelques millisecondes, que de quelques minutes.
- Le *débit* (en Anglais *throughput*) est le nombre de commandes qui sont exécutées par unité de temps. C'est une mesure de l'efficacité de l'ensemble du système, matériel et logiciel, qui est surtout intéressant pour le responsable du service informatique, car il permet de calculer indirectement la rentabilité du système, tous utilisateurs confondus. Il dépend évidemment de la nature des commandes demandées par les utilisateurs. Dans le mode transactionnel, le débit est défini par le nombre de transactions par seconde ou *Tps*. Dans le mode différé, le débit est défini par le nombre de travaux à l'heure.
- Les *moments d'arrivée* des commandes déterminent la façon dont les commandes sont demandées au système. C'est une caractéristique de l'utilisateur. On a constaté, par exemple, qu'en mode temps partagé un utilisateur réfléchissait environ 30 secondes entre la réponse à une commande et l'envoi de la commande suivante. Cette durée peut être plus courte en mode transactionnel, lorsqu'il s'agit par exemple de la saisie d'un ensemble d'opérations comptables. A l'opposé, le mode différé signifie que chaque travail consiste lui-même en l'enchaînement d'une suite de commandes prédéfinies, permettant une planification des travaux.
- Le *mode de communication* entre le demandeur et l'ordinateur impose des contraintes supplémentaires sur le système, dès que l'on désire introduire une distance entre les deux. Nous avons vu dans le chapitre précédent que la communication entre machines faisait intervenir un protocole normalisé plus ou moins complexe.
- La *nature du partage des ressources* entre les demandeurs doit être clairement appréhéndée, car elle a une influence directe sur le comportement du système. Par exemple, le processeur est une ressource effectivement partagée entre les utilisateurs. Le système doit permettre une utilisation simultanée (à l'échelle de temps humaine) de cette ressource. Par contre une imprimante, ou un dérouleur de bandes magnétiques ne peuvent être attribués qu'à un seul utilisateur à la fois, pendant des périodes de temps assez longues. Le partage des ressources “logicielles” laisse souvent plus de liberté de choix suivant le mode d'utilisation. Notons tout d'abord que toute donnée peut être consultée par un nombre quelconque d'utilisateurs sans difficulté. Les problèmes surgissent avec les modifications d'une même donnée par plusieurs. En mode différé,

le partage de données est obtenu simplement en attribuant exclusivement au travail l'ensemble des données dont il a besoin. Il en va souvent de même en mode temps partagé ou en mode conception assistée. En mode transactionnel, il est par contre impératif de permettre à de nombreux utilisateurs de pouvoir consulter ou modifier une même donnée, qui doit alors apparaître comme utilisée simultanément par tous ces utilisateurs. Par ailleurs, dans ce mode, il y a souvent beaucoup d'utilisateurs exécutant le même “programme”, justifiant le partage de la mémoire centrale entre les utilisateurs.

### 3.1.3. Système général ou spécialisé

Certains systèmes ont été conçus pour satisfaire le plus grand nombre de besoins, tout en privilégiant légèrement l'un des modes.

Par exemple, Multics (Bull) ou VMS (Dec) privilégient le mode temps partagé, tout en étant acceptable pour le transactionnel ou le différé; Multics ne tournant que sur grosse machine (DPS-8 spécial) est mal adapté aux modes conception assistée ou temps réel, alors que VMS peut être configuré sur des machines de taille moyenne pour répondre à peu près aux besoins de la conception assistée ou du temps réel. De même UNIX privilégie sutout le temps partagé; sa taille (initialement petite!) et sa disponibilité sur de nombreux matériels permettent de l'adapter aux besoins de la conception assistée. On ne peut le considérer comme adapté au temps réel, même s'il est plus facile que dans bien d'autres systèmes, d'y incorporer des “pilotes de périphériques”, c'est-à-dire des sous-programmes de gestion de périphériques.

A l'opposé, GCOS3 (Bull) ou MVS (IBM) sont des systèmes orientés d'abord vers le mode différé. Sur MVS, les autres modes sont obtenus par le biais de sous systèmes spécialisés, tels que TSO pour le temps partagé, CICS ou IMS pour le mode transactionnel.

Le système VM (IBM) se place de façon différente. Il s'agit en fait d'un noyau de système qui émule un ensemble de machines sur lesquelles on peut faire exécuter des systèmes particuliers. Par exemple, CMS est un système tournant sous VM, qui privilégie le mode temps partagé; il est d'ailleurs plus apprécié que TSO sous VMS.

Pour certaines applications, cette universalité du système d'exploitation n'est pas adaptée, ce qui conduit alors à concevoir un système spécialisé pour l'un des modes d'utilisation. Ceci est particulièrement vrai pour le temps réel ou le contrôle de procédés, pour lequel il existe beaucoup de systèmes dédiés: iRMX86 de Intel, SPART sur SPS-7 de Bull, RT-11 de Dec, etc... Il en est ainsi également pour certaines applications transactionnelles où le nombre de terminaux est particulièrement important, comme, par exemple, pour les systèmes de réservation de places pour les transports aériens ou terrestres. Enfin, la conception assistée demande souvent une puissance de calcul importante, et conduit à un poste de travail qui est déjà un ordinateur doté d'un système d'exploitation; il s'agit souvent d'un Unix configuré spécialement pour ces besoins.

Sur les ordinateurs personnels, par définition, il n'y a qu'un utilisateur à la fois. Il est donc naturel que leur système mette l'accent sur le mode interactif. MS-DOS, Windows 95/98 ou MacOS en sont de bons exemples. Cependant, si à l'origine, la faible puissance de ces ordinateurs justifiait la simplicité du système, ce n'est plus le cas aujourd'hui, et ces systèmes se sont complexifiés dans les aspects interfaces. Cependant, les limites de ces systèmes, qui ne gèrent pas le partage du processeur entre plusieurs activités, sont mises en évidence par la concurrence des stations de travail, de prix très abordable, dotées souvent d'un système de type UNIX, LINUX ou Windows NT.

## 3.2. Les services offerts par le système

Nous nous intéressons ici au point de vue de l'utilisateur dans son travail de conception et de réalisation d'une application particulière.

### 3.2.1. La chaîne de production de programmes

Nous avons vu dans le chapitre précédent que le langage de l'ordinateur était binaire et rudimentaire. Il faut donc tout d'abord lui fournir un ensemble d'outils qui lui permette de construire cette application dans un langage plus agréable pour lui; ce sont les divers constituants de la *chaîne*

*de production de programmes*. Certains d'entre eux seront étudiés plus en détail dans les chapitres suivants.

- L'*interpréteur de commande* est le premier outil fondamental. C'est lui qui va permettre à l'utilisateur de se faire connaître de l'ordinateur (*login*), et de lui demander l'exécution des programmes préenregistrés, au fur et à mesure de ses besoins. Certains systèmes offrent plusieurs interprèteurs permettant de satisfaire des utilisateurs aux besoins différents.
- Les *éditeurs de texte* permettent la création et la modification du texte des programmes. Les premiers découpaient le texte en lignes, et permettaient les modifications des textes comme on le faisait avec des cartes perforées, en utilisant des terminaux de possibilités réduites. La deuxième génération découpe le texte en pages, affiche une page sur l'écran dont est doté le terminal et permet de se déplacer dans le texte à la position voulue pour faire les corrections visibles immédiatement sur l'écran. Dans chacun des cas, le texte manipulé est une suite de caractères quelconques, l'éditeur permettant parfois la manipulation par caractère, par mot, par ligne ou par paragraphe. La dernière génération tente de faciliter le travail du programmeur en tenant compte du langage utilisé: l'éditeur permet alors la manipulation d'entités structurelles du langage, telles que les instructions, les blocs, les sous-programmes, les déclarations, etc... On les appelle *éditeurs syntaxiques*, car les manipulations autorisées sont guidées par la syntaxe du langage de programmation de l'utilisateur.
- Les *compilateurs*, *interpréteurs* ou *assembleurs* sont des programmes particuliers, fournis en général par le constructeur de la machine, qui permettent la traduction d'un programme écrit dans un langage donné, plus adapté à l'homme, dans le langage de la machine.
- L'*éditeur de liens* et le *chargeur* sont également des programmes fournis par le constructeur. Les logiciels professionnels sont souvent de bonne taille, et ne peuvent être manipulés comme un tout. Il est nécessaire de les décomposer en *modules* qui sont des morceaux de programmes qui peuvent être compilés séparément les uns des autres. L'éditeur de liens permet de rassembler de tels modules déjà compilés et d'en faire un programme exécutable. Le chargeur est le programme qui assure la mise en mémoire centrale d'un programme exécutable, et d'en lancer l'exécution (figure 3.1).

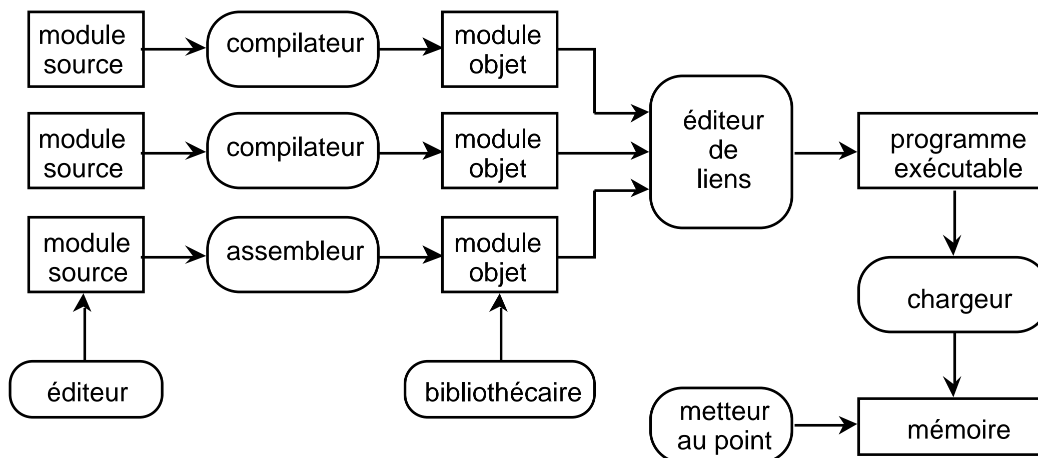



***Fig 3.1***. *La chaîne de production de programmes.*

- Des outils complémentaires sont souvent également fournis pour faciliter le travail du programmeur, tels que un *metteur au point* qui facilite le test des programmes, *paragrapheur* qui met en forme les textes sources de programmes, outil de *références croisées* qui permet de connaître les liens entre les modules, *bibliothécaire* qui gère les modules déjà compilés, etc...

## 3.2.2. Le programme et les objets externes

Nous avons vu également que les connexions avec les périphériques étaient complexes et variées. Bien des applications n'ont pas pour but d'accéder à ces périphériques en tant que tel, mais pour pouvoir mémoriser des informations dont la durée de vie excède la durée d'exécution du programme, ou pour pouvoir communiquer avec l'extérieur de la machine. Il faut donc lui fournir des moyens de simplifier les accès à ces périphériques, en lui proposant des mécanismes uniformes d'*accès à des objets externes* aux programmes.

Les chapitres 7 à 11 montreront quels sont les problèmes, et comment ils sont résolus habituellement. Il s'agit de fournir à l'utilisateur, d'une part, des outils de gestion de ces objets externes sous forme de programmes spécifiques, mais aussi, d'autre part, des sous-programmes d'accès au contenu de ces objets à l'intérieur de ses propres programmes.

### 3.2.3. Le programme et son environnement physique

Le matériel dont on dispose est figé et a certaines caractéristiques plus ou moins faciles à utiliser. Le programmeur a, par ailleurs, des besoins inhérents à l'application qu'il doit réaliser, et ne peut configurer strictement le matériel à ses besoins, car ceux-ci varient d'une application à l'autre. Le problème est alors de fournir à l'utilisateur une *machine virtuelle* adaptée à ses besoins. Le terme virtuel doit ici être compris essentiellement par opposition à réel, c'est-à-dire que notre machine virtuelle est obtenue par substitution totale ou partielle de composants matériels par des composants logiciels. L'un des premiers aspects est en particulier d'assurer l'indépendance des usagers de la machine réelle, en gérant implicitement les ressources communes (processeur, mémoire, etc...). Il est possible également de “simuler” des ressources qui n'existent pas ou qui sont insuffisantes, de façon à adapter la configuration de la machine virtuelle aux besoins. Ceci sera développé dans les chapitres 12 à 14.

## 3.3. Architecture d'un système

La figure 3.2 montre l'architecture habituelle d'un système informatique. Dans le premier niveau, on trouve la machine nue, c'est-à-dire les composants matériels. Le deuxième niveau assure les fonctions premières de gestion des ressources physiques, et sert d'interface entre le logiciel et le matériel. Le troisième niveau définit les fonctionnalités du système lui-même. Les niveaux supérieurs ne sont plus partie intégrante du système proprement dit. Ce sont donc des programmes plus ou moins fortement liés au système. En particulier le quatrième niveau rassemble l'ensemble des outils qui contribuent à la chaîne de production de programmes. Le cinquième niveau regroupe l'ensemble des programmes d'application, ainsi que des utilitaires courants qui sont en fait des programmes d'application suffisamment généraux pour ne pas être dédiés à une application donnée, comme par exemple les utilitaires de tri-fusion ou de copie de fichiers. Les utilisateurs se trouvent au niveau supérieur (6).

<table>
<tr><td>6</td><td colspan="6">utilisateurs</td></tr>
<tr><td>5</td><td colspan="2">programmes d'application</td><td colspan="2">programmes utilitaires</td><td colspan="2">ateliers de programmation</td></tr>
<tr><td>4</td><td>éditeur de texte</td><td>assembleur</td><td>compilateur</td><td>éditeur de liens</td><td>chargeur</td><td>metteur au point</td></tr>
<tr><td>3</td><td colspan="6">système d'exploitatio r</td></tr>
<tr><td>2</td><td>gestion processeur</td><td>gestion mémoire</td><td>gestion données</td><td>gestion périphériques</td><td colspan="2">gestion communications</td></tr>
<tr><td>1</td><td colspan="6">machine nue</td></tr>
</table>

***Fig. 3.2.*** *Architecture générale d'un système informatique.*

## 3.4. Conclusion

☞ Il y a plusieurs modes différents d'utilisation de l'ordinateur: le mode interactif, le mode différé, ou le mode temps réel. Le mode interactif peut lui-même se décomposer en mode temps partagé, mode transactionnel et mode conception assistée.

☞ Chacun des modes a ses propres caractéristiques, qui se distinguent par le temps de traitement, le temps de réponse, les moments d'arrivée, le mode de communication et la nature du partage des ressources.

☞ Les systèmes d'exploitation peuvent être conçus pour satisfaire le plus grand nombre de besoins, tout en privilégiant l'un des modes, ou être dédiés à un besoin spécifique, lorsque les contraintes sont trop fortes.

☞ La chaîne de production de programmes est l'ensemble des outils qui concourrent à la réalisation des programmes. Elle comprend principalement l'interpréteur de commandes, les éditeurs de texte, les traducteurs, l'éditeur de liens, le chargeur, etc...

☞ Les objets externes sont des objets manipulés par les programmes, mais dont la durée de vie s'étend au-delà de la durée d'exécution d'un programme unique.

☞ L'environnement physique des programmes doit être adapté par le système pour fournir à l'utilisateur une machine virtuelle dont la configuration correspond à ses besoins.

# DEUXIÈME PARTIE

---

# CHAINE DE PRODUCTION DE PROGRAMMES

# 4

# La traduction des langages de programmation

Le langage de la machine est en fait constitué de bits, ce qui le rend impraticable par l'homme. La première étape a été de concevoir des langages simples, assez proches du langage de la machine pour pouvoir être traduits facilement. Il s'agissait alors de remplacer les codes binaires des opérations de base par des codes mnémoniques plus faciles à se rappeler, la traduction consistant à retrouver les codes binaires correspondants. Ceci a conduit aux *langages d'assemblage* qui se sont perfectionnés, tout en restant proches de la machine, dont le programmeur devait avoir une connaissance approfondie.

La difficulté de mise en œuvre de ce type de langage, et leur forte dépendance avec la machine a nécessité la conception de *langages évolués*, plus adaptés à l'homme, et aux applications qu'il cherchait à développer. Faisant abstraction de toute architecture de machine, ces langages permettent l'expression d'algorithmes sous une forme plus facile à apprendre, et à dominer. On s'intéresse alors aux vrais besoins d'expression du programmeur plutôt qu'aux mécanismes fournis par la machine. Il s'ensuit une spécialisation du langage suivant la nature des problèmes à résoudre et non suivant la machine qui sera utilisée pour les résoudre. Ainsi, FORTRAN est un langage destiné aux applications scientifiques, alors que COBOL est destiné aux applications de gestion.

Pour l'ordinateur, un programme écrit dans un langage qui n'est pas le sien est en fait une simple suite de caractères plus ou moins longue. Si le langage est un langage d'assemblage, un programme spécialisé, l'*assembleur*, parcourt cette suite de caractères pour reconnaître les différents mnémoniques, et les remplacer par leur équivalent binaire dans le langage de la machine (c'est une approche très sommaire!). Si le langage est un langage évolué, le programme spécialisé, chargé de la traduction, peut se présenter sous l'une des deux formes suivantes:

- L'*interpréteur* parcourt en permanence la suite de caractères, analyse les actions définies par le programme, exprimées dans le langage évolué, et exécute immédiatement la séquence d'action de la machine qui produit les mêmes effets.
- Le *compilateur* parcourt la suite de caractères, analyse les actions du programme et la traduit en une suite d'instructions dans le langage de la machine.

Il s'ensuit évidemment que l'analyse des actions est une phase plus complexe et plus difficile que pour l'assembleur. Notons que l'interpréteur simule une machine qui comprend le langage évolué, alors que le compilateur fait une traduction dans le langage de la machine, et c'est cette traduction qui est exécutée. Il s'ensuit qu'un interpréteur est plus facile à utiliser pour le programmeur, mais le résultat est moins efficace que l'exécution du programme compilé, car les actions du programme sont analysées chaque fois qu'elles doivent être exécutées. Par exemple, une action située à l'intérieur d'une boucle exécutée 10000 fois, sera analysée 10000 fois.

Notons qu'il est possible de combiner les deux. Plusieurs environnements de programmation ont été ainsi construits autour du Pascal vers 1971, de Caml vers 1988 et de Java en 1998, chacun d'eux ayant un fonctionnement analogue. Nous décrivons celui de Java qui est le plus récent. On définit d'abord une machine virtuelle universelle, ayant son propre langage, appelé bytecode. En général ces machines n'existent pas physiquement, bien qu'il y ait eu des tentatives dans ce sens. On compense cette absence en la simulant par logiciel, c'est-à-dire en créant un interpréteur de bytecode. La compilation consiste alors à traduire un programme Java en bytecode qui peut donc être exécuté à l'aide de cet interpréteur. Nous reviendrons sur ce principe à la fin du chapitre.

Quels que soient le langage et la méthode de traduction utilisés, le programme est toujours une suite de caractères. C'est ce que nous appellerons un *programme source*. Le traducteur doit analyser cette suite de caractères pour retrouver les constituants essentiels, vérifier la conformité avec la définition du langage, et en déduire la sémantique. Un interpréteur doit ensuite exécuter les actions correspondant à cette sémantique, alors qu'un assembleur ou un compilateur doit traduire cette sémantique dans un autre langage. On appelle *programme objet*, le résultat de cette traduction. Par ailleurs, nous avons vu qu'il pouvait être intéressant de découper un programme en morceaux, traduits séparément, et rassemblés ultérieurement par l'éditeur de liens (évidemment, ceci n'est pas possible avec un interpréteur). On parle alors respectivement de *module source* et de *module objet*. Le langage utilisé doit permettre la prise en compte de ce découpage en module.

# 4.1. L'analyse lexicale

La suite de caractères représentant le module est la forme concrète de communication entre le programmeur et le traducteur. Mais chaque caractère pris individuellement n'a pas, en général, de signification dans le langage lui-même. Par exemple, dans un langage tel que Pascal, chaque caractère de la suite “**begin**” n'a pas de signification, alors que la suite elle-même est un *symbole* du langage. Il apparaît alors nécessaire de découper le module source, en tant que chaîne de caractères, en une suite de symboles du langage, chaque symbole étant représenté par une suite de caractères.

Considérons, par exemple, le petit bout de programme source suivant:

```
begin x := 23; end
```

Cette suite de caractères doit être découpée de la façon suivante:

| **Begin** | x | := | 23 | ; | **end** |
|---|---|---|---|---|---|

mettant en avant les six symboles dont est constitué le programme. Pour des raisons de commodité de traitement évidentes, on donne à chacun des symboles, une forme de représentation interne à la machine. Cette opération de transformation du module source en une représentation interne codée s'appelle l'*analyse lexicale*. Cette opération n'est évidemment possible que si la structure lexicale du langage est correctement définie[4]. Ceci implique quelques contraintes habituelles sur la définition des langages:

- Un identificateur commence par une lettre, et ne peut comporter que des lettres, des chiffres, ou éventuellement des caractères spéciaux qui ne sont pas utilisés ailleurs ("_" ou "$", par exemple). On peut décrire cela sous la forme suivante:
  ```
  identificateur ::= lettre { lettre | chiffre | "_" }
  ```
- Une constante numérique commence par un chiffre, même si elle est dans une base supérieure à 10. On peut décrire cela sous la forme suivante:
  ```
  constante ::= chiffre { chiffre | lettre }
  ```
- Les mots clés ont souvent la forme lexicale des identificateurs, et ne sont pas alors utilisables en tant qu'identificateur. C'est ce que l'on appelle encore les mots réservés.
- Les opérateurs peuvent être représentés soit par l'un des caractères spéciaux habituels, tels que par exemple "+", "-", "=", etc..., mais aussi par une combinaison de ces caractères spéciaux, comme par exemple ":=", pourvu que cette combinaison ne soit pas ambiguë.

---

[4] Notons que c'est l'absence d'une telle définition correcte pour le langage FORTRAN qui a été la cause de l'une des erreurs de programmation les plus coûteuses pour la NASA. Un “point” ayant été tapé à la place de la “virgule”, le compilateur a traduit l'instruction de boucle "DO 50 I = 1 . 5" par une simple affectation de la valeur 1.5 à la variable "DO50I". La sonde de Venus est passée à 500 000 km de sa cible.

- Les espaces, tabulations, fins de lignes sont sans signification, c'est-à-dire qu'ils sont ignorés au cours de l'analyse, si ce n'est qu'ils ne peuvent se trouver à l'intérieur de la représentation d'un symbole. En d'autres termes, ces caractères servent essentiellement à la présentation externe, mais sont aussi des séparateurs entre deux symboles. Ainsi dans le petit programme ci-dessus, l'espace permet à l'analyseur lexicale de séparer le symbole **`begin`** de l'identificateur `x`, et est donc nécessaire à cet endroit, alors que les autres espaces sont inutiles.

L'idée générale de l'analyseur lexical est, sachant où commence un symbole dans la chaîne de caractères du module source, de rechercher où il se termine, en suivant les règles de définitions lexicales du langage, puis de trouver où commence le symbole suivant, etc... Ces techniques sont suffisamment générales et bien maîtrisées pour permettre de construire des programmes capables de produire automatiquement des analyseurs lexicaux à partir de la définition lexicale d'un langage (par exemple, l'utilitaire *lex* disponible sur Multics ou sur Unix).

# 4.2. L'analyse syntaxique

Le résultat obtenu après analyse lexicale est une suite de symboles. Mais, toute suite de symboles ne constitue pas un programme. Il faut donc vérifier la concordance de la suite avec la structure du langage, sans se préoccuper, pour le moment, de la sémantique du programme. C'est ce que l'on appelle l'*analyse syntaxique*.

La plupart des langages de programmation modernes définissent une syntaxe du langage, indépendante du contexte, par des *règles de production*, qui décrivent la façon dont on peut construire une suite de symboles pour qu'elle constitue un programme. Le programmeur construit alors son programme en appliquant ces règles de façon répétitive. L'analyse syntaxique consiste à retrouver, à partir du programme source, ou en fait à partir de la suite de symboles, quelles sont les règles que le programmeur a appliquées. On utilise une syntaxe indépendante du contexte, pour permettre cette reconstruction.

Une règle de production est, en général, donnée sous une forme standard, dite *forme normale de Backus-Naur*. Cette forme a l'avantage d'être simple et facile à assimiler. Par exemple, la syntaxe d'une expression pourrait être donnée par l'ensemble des règles suivantes:

```
<expression>   ::= <facteur> | <facteur> <opérateur additif> <expression>
<facteur>      ::= <terme> | <terme> <opérateur multiplicatif> <facteur>
<terme>        ::= <identificateur> | <nombre> | ( <expression> )
<opérateur additif>         ::= + | -
<opérateur multiplicatif>   ::= * | /
```

La première précise qu'une `expression` est, soit un `facteur`, soit un `facteur` suivi d'un `opérateur additif` suivi d'une autre `expression`. De même la dernière indique qu'un `opérateur multiplicatif` est soit le symbole `*`, soit le symbole `/`. Plus généralement, l'interprétation est la suivante:

- Les structures syntaxiques s'écrivent sous la forme de mots encadrés par les signes `<` et `>`.
- Les symboles s'écrivent sous leur forme lexicale.
- Une règle de production comporte une structure syntaxique unique à gauche du signe `::=` et une ou plusieurs suites de symboles et de structures syntaxiques, les suites étant séparées par le signe `|` indiquant un choix parmi les suites.

Dans le contexte des règles ci-dessus, la suite de symboles "`23 * 2 + 5`" est une `expression` constituée du `facteur` "`23 * 2`", suivi de l'`opérateur additif` "`+`" et de l'`expression` "`5`", qui est aussi un `facteur`, puis un `terme` et enfin un `nombre`. Le `facteur` "`23 * 2`" est lui-même constitué du `terme` "`23`" qui est un `nombre`, suivi de l'`opérateur multiplicatif` "`*`" et du `facteur` "`2`", qui est un `terme`, et un `nombre`.

Cet exemple montre que, non seulement, l'analyse syntaxique permet de contrôler que le programme est correctement écrit, mais aussi, qu'elle permet de trouver la structure syntaxique de ce programme (figure 4.1). En particulier, si `2 + 5` est bien une sous-suite de la suite donnée, elle ne correspond à aucune structure syntaxique du langage dans cette `expression`, alors que la sous-suite `23 * 2` a été reconnue comme `facteur` de cette `expression`.

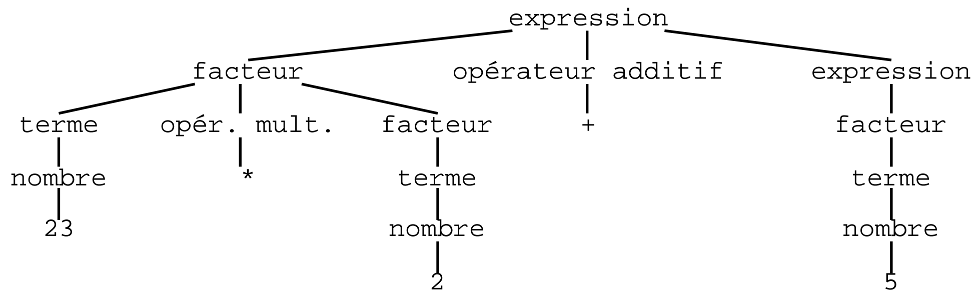



***Fig. 4.1***. *Structure syntaxique de l'expression 23 * 2 + 5.*

La rigueur de la définition des règles de production tient à la nécessité d'éviter toute ambiguïté dans la façon dont le programme est écrit. Il est en effet nécessaire d'avoir une structure syntaxique unique pour une suite donnée de symboles. Si plusieurs structures étaient possibles, il pourrait arriver que le compilateur choisisse une structure qui ne corresponde pas celle voulue par le programmeur, ce qui pourrait donner lieu ensuite à une interprétation sémantique différente.

# 4.3. L'analyse sémantique

Après avoir contrôlé la correction du programme, et en avoir obtenu la structure syntaxique, la traduction continue par la phase d'*analyse sémantique*, qui a pour but de trouver le sens des différentes actions voulues par le programmeur. Trois problèmes doivent alors être résolus:

- quels sont les objets manipulés par le programme,
- quelles sont les propriétés de ces objets,
- quelles sont les actions du programme sur ces objets.

Les objets manipulés sont en général désignés par le programmeur au moyen d'identificateurs. Certains langages imposent que tous les identificateurs soient explicitement “déclarés”, c'est-à-dire, que le programmeur doit donner des indications au traducteur sur la nature et les propriétés de l'objet désigné par l'identificateur. D'autres langages déduisent implicitement ces informations de la façon dont l'identificateur est utilisé dans le programme. Les propriétés importantes sont les suivantes:

- Le *type* de l'objet permet au traducteur de connaître la sémantique des opérations effectuées sur l'objet, et de déterminer ainsi la combinaison des opérations de la machine qui auront le même effet.
- La *durée de vie* de l'objet permet au traducteur de savoir s'il doit prévoir ou non d'engendrer des instructions permettant sa création ou sa destruction, et à quels moments de l'exécution du programme.
- La *taille* de l'objet permet au traducteur de savoir le nombre d'emplacements de mémoire occupés par sa représentation.
- L'*adresse* de l'objet permet au traducteur de désigner l'objet dans les intructions de la machine qui le manipulent.

L'analyse sémantique des actions du programme a pour but de permettre au traducteur de déterminer avec précision la suite des actions qu'il doit exécuter indépendamment de tout aspect langage. Prenons par exemple le morceau de programme suivant:

```
var i: entier;
   s: réel;
début
     s := 0;
     i := 0;
     tantque i < 10 faire
          s := s + sqrt (i);
          i := i + 1;
     fait;
fin;
```

Des déclarations, le traducteur peut conclure que l'affectation `s := 0` est une affectation de la valeur `0` à un nombre réel, alors que `i := 0` est une affectation entière. De même, l'addition dans `s := s + sqrt (i)` est une addition entre des nombres réels, alors que l'addition dans `i := i + 1` est une addition entre des nombres entiers.

Cette phase d'analyse comporte une part importante de contrôle, puisque le traducteur peut constater des éléments incompatibles ou ambigus, ou des actions qui n'ont pas de sens. Les erreurs qui peuvent être découvertes à cette étape, sont par exemple l'absence de déclaration ou des déclarations incompatibles, des identificateurs déclarés, mais inutilisés, des expressions sans signification (ajouter des réels et des chaînes de caractères, par exemple), etc...

Bien des contraintes que l'on trouve dans les langages de programmation modernes, et qui paraissent superflues a priori, n'ont en fait pour but que de garantir au programmeur que la sémantique déduite par le traducteur de son programme ne présente aucune ambiguïté. Si ce n'était pas le cas, cela impliquerait que le traducteur pourrait prendre une sémantique différente de celle voulue par le programmeur, et que deux traducteurs différents du même langage pourraient déduire des sémantiques différentes.

## 4.4. La génération et l'optimisation de code

La phase de génération de code est la phase de traduction proprement dite. Elle n'est entreprise que si aucune erreur n'a été trouvée. Elle consiste à construire un programme équivalent à la sémantique obtenue dans la phase précédente, dans un nouveau langage, et donc en particulier dans le langage de la machine lui-même. Le plus souvent, l'analyse sémantique produit une structure particulière qui regroupe l'ensemble des informations obtenues sur le programme. La génération exploite simplement cette structure pour produire le code.

La notion d'optimisation est plus intéressante. L'écriture de programmes dans un langage évolué permet de s'abstraire des particularités de la machine, mais a pour conséquence de ne pas toujours donner un programme traduit aussi efficace que si l'on avait directement programmé dans le langage d'assemblage. L'optimisation de code a pour but de compenser cet inconvénient. On peut dire que la qualité d'un compilateur peut se mesurer par l'efficacité du code machine produit, et repose en grande partie sur les techniques d'optimisation qui sont mises en jeu. Il n'est pas question d'en expliquer ici le fonctionnement, mais de donner quelques indications sur ses possibilités.

- Tout d'abord, le compilateur peut déterminer les expressions constantes, c'est-à-dire dont le résultat peut être connu à la compilation, et propager cette valeur le plus possible dans le programme.
- Il est facile également d'éliminer le code inaccessible. Le but essentiel est de diminuer l'espace mémoire du programme résultat. C'est une solution élégante au problème de la *compilation conditionnelle*. Par exemple, si le compilateur constate qu'une condition d'une instruction conditionnelle est toujours fausse, il peut éliminer toutes les instructions qui devraient être exécutées dans le cas où la condition est vraie.
- Il peut déplacer des parties de code pour faire sortir d'une boucle des instructions qui produiront le même résultat à chaque pas de la boucle.
- Il peut éliminer les calculs liés aux variables d'induction de boucle, pour les remplacer par d'autres plus essentiels. Par exemple, la progression d'un indice dans un tableau peut être remplacée par la progression d'une adresse sur les emplacements du tableau.

Il ne s'agit pas d'améliorer un programme mal écrit, mais d'atténuer certaines inefficacités dûes à l'écriture de programmes dans un langage évolué.

## 4.5. La traduction croisée

Un traducteur est un programme exécutable particulier qui prend pour donnée un module source, et fournit comme résultat un module objet. En tant que programme exécutable, il s'exécute sur une machine particulière qui comprend le langage dans lequel il est écrit. Si cette machine est bien souvent la même que celle à laquelle sont destinés les modules objets qu'il produit, ce n'est pas une obligation. Lorsque les deux machines sont différentes, on parle de *traduction croisée*. Le

traducteur s'exécute alors sur une *machine hôte*, et produit des modules objets pour une *machine cible* (figure 4.2).

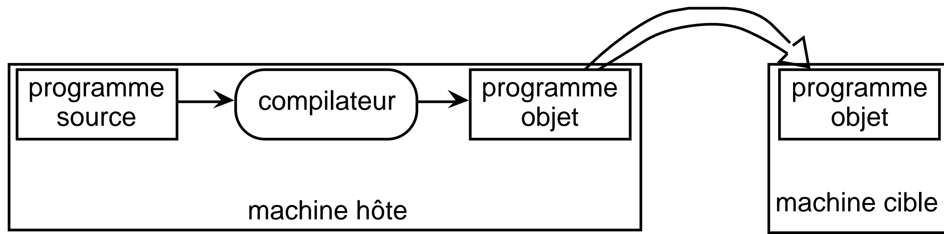



***Fig. 4.2.*** *Traduction croisée.*

Plus généralement, le traducteur est lui-même écrit dans un langage d'assemblage, ou plus souvent dans un langage évolué $L_1$ ; il accepte en entrée un module source écrit dans un langage L qu'il traduit en un module objet écrit en un langage LM. Par le biais d'un traducteur de $L_1$ vers $LM_2$ s'exécutant sur une machine $M_1$, on obtient un traducteur de L vers LM écrit en $LM_2$, qui peut s'exécuter sur une machine $M_2$ si le langage de $M_2$ est $LM_2$. Ce traducteur de L vers LM permettra de traduire sur la machine $M_2$ les programmes écrits en L en des programmes écrits en LM qui pourront s'exécuter sur les machines M dont le langage est LM (figure 4.3).

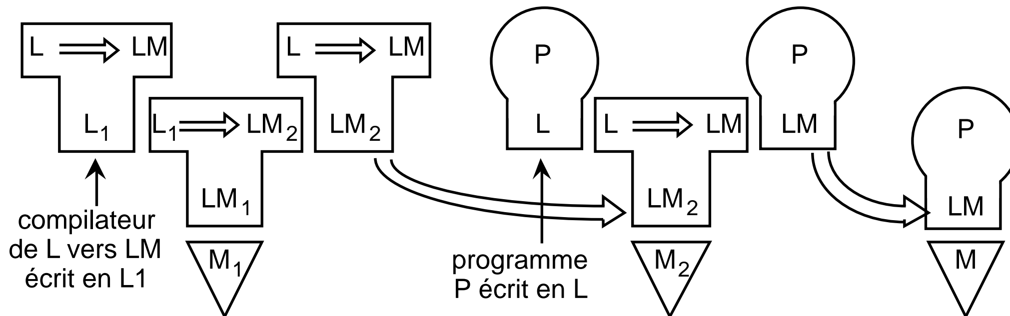



***Fig. 4.3.*** *Représentation schématique de la traduction croisée.*

Le schéma de traduction croisée a un cas particulier représenté par la figure 4.4. Le traducteur C du langage L vers le langage LM est écrit dans le langage L lui-même. On emprunte pendant quelque temps un traducteur C1 de L vers LM écrit en LM. Il est donc capable de traduire C en un traducteur équivalent C', écrit en LM. Les performances d'exécution de C' sont directement liées aux performances du traducteur C1. On peut alors utiliser notre traducteur C' de L en LM écrit en LM sur la machine M pour de nouveau traduire C en C''. Cette fois, les performances d'exécutions de C'' sont celles que nous avons définies dans notre traducteur C de L en LM, qui s'est donc autocompilé. En général C'' est différent de C', car ils sont tous deux produits par deux traducteurs différents. Par contre, si on recommence l'opération en traduisant C par C'', on réobtiendra C''. Ceci est parfois utilisé à des fins de contrôle de bon fonctionnement de la suite des opérations.

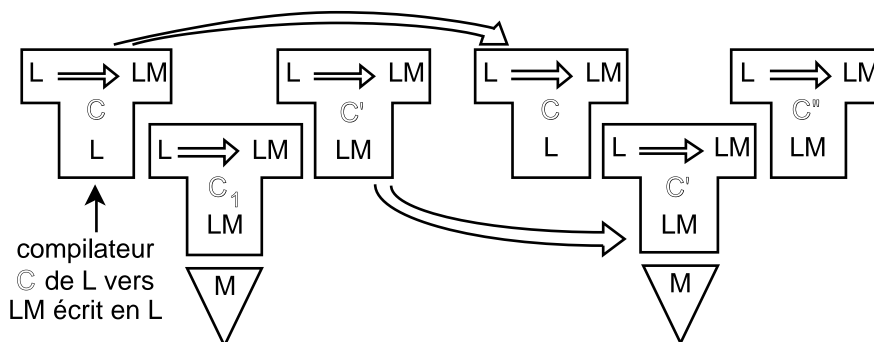



***Fig. 4.4.*** *Représentation schématique de l'autocompilation.*

Nous avons vu que, dans le cas du langage Java, il est habituel de combiner une phase de traduction de Java en bytecode et une interprétation du bytecode par une machine virtuelle (simulation par

logiciel de la machine). Ainsi tout programme java peut être traduit en bytecode sur une machine M, envoyé sur n'importe quelle autre machine M1 pour être exécuté pourvu que celle-ci dispose d'un interpréteur de bytecode. Cependant, pour pallier le manque de performance inhérent à l'interprétation, la machine réceptrice peut également traduire le bytecode en son langage propre et le résultat être exécuté directement sur M1. Un tel traducteur est encore appelé *just in time compiler* (figure 4.5).

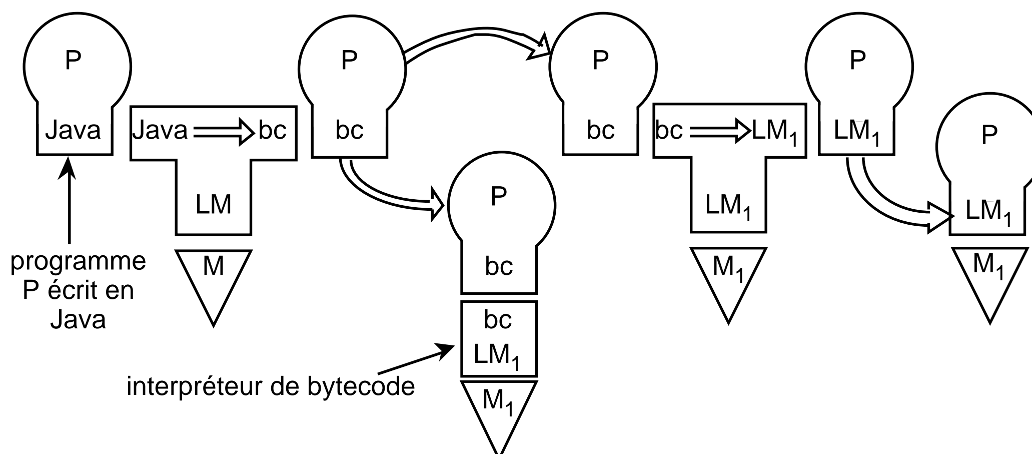



***Fig. 4.5.*** *Représentation schématique du système Java.*

# 4.6 Conclusion

☞ L'assembleur est un programme qui traduit un programme écrit en langage d'assemblage, à base de codes mnémoniques d'opérations d'une machine dans le langage binaire de cette machine.

☞ L'interpréteur est un programme qui simule une machine qui comprend un langage évolué.

☞ Le compilateur est un programme qui traduit un programme écrit dans un langage évolué dans le langage binaire d'une machine, pour que celle-ci puisse l'exécuter.

☞ L'analyse lexicale consiste à reconnaître dans une chaîne de caractères la suite des symboles du langage et à en donner une codification interne.

☞ L'analyse syntaxique consiste à retrouver la structure syntaxique d'une suite de symboles et à vérifier sa conformité aux règles du langage.

☞ L'analyse sémantique consiste à trouver une signification, indépendante de tout langage, aux actions définies par le programme.

☞ L'optimisation a pour but d'atténuer certaines inefficacités dûes à l'écriture en langage évolué.

☞ La génération de code consiste à traduire la sémantique du programme dans le langage binaire de la machine.

☞ La traduction croisée consiste à utiliser un traducteur qui s'exécute sur une machine hôte pour produire un résultat qui s'exécute sur une machine cible différente.

☞ *En fait, tout programme qui interprète des données venant d'un utilisateur devrait comporter les phases d'analyse lexicale, syntaxique et sémantique.*

# 5

# L'édition de liens et le chargement

Les programmes relativement complexes ne peuvent être écrits est nécessaire de pouvoir les découper en morceaux, chacun d'eux pouvant alors être modifié, compilé et testé séparément des autres. Les morceaux doivent ensuite pouvoir être rassemblés pour former le programme complet. Nous avons appelé *module source* un tel morceau de programme pouvant être traité de façon indépendante par un traducteur, et *module objet* le résultat de cette traduction. Nous nous intéressons maintenant à la reconstruction du programme complet à partir des modules objets. Nous avons appelé *édition de liens* cette opération. Elle est réalisée par un programme spécifique, appelé *éditeur de liens* (en Anglais *link editor*).

Pour préciser clairement ce dont on parle, on utilise parfois le terme *module éditable* ou *programme éditable* pour désigner le résultat d'une traduction qui nécessite le passage par l'édition de liens, et le terme *programme exécutable* pour désigner un programme complet prêt à être chargé et exécuté.

## 5.1. La notion de module translatable

Un module objet, lorsqu'il est inclus dans un programme exécutable, occupe un certain nombre d'emplacements en mémoire. Ces emplacements doivent être disjoints de ceux occupés par les autres modules objets du même programme. L'architecture matérielle (par exemple, Multics, iAPX432, etc...) permet parfois de conserver la modularité à l'exécution en attribuant des espaces mémoires disjoints à chaque module. Les adresses sont alors des couples `<n, d>` où `n` désigne l'espace attribué au module, et `d` un déplacement relatif dans cet espace. Dans ce cas, le traducteur est entièrement maître de la gestion interne, dans cet espace, des objets appartenant au module qu'il traite, et donc des valeurs des déplacements relatifs.

Par contre, dans la majorité des cas, il faut pouvoir mettre dans un espace unique plusieurs modules issus de traductions différentes. Les traducteurs doivent alors mettre dans les modules éditables qu'ils produisent les informations qui permettent de placer ces modules n'importe où en mémoire. En principe, il suffit qu'ils indiquent les différents emplacements où sont rangées des adresses internes au module. En fait, les traducteurs séparent souvent l'espace mémoire du module en plusieurs *sections* suivant la nature de leur contenu, comme par exemple:

- code des instructions,
- données constantes,
- données variables.

Ces différentes sections, gérées séparément par le traducteur peuvent être mises n'importe où en mémoire. Les informations de placement produites par le traducteur précisent alors pour chaque emplacement qui contient une adresse, la section à laquelle cette adresse se réfère. On dit alors que le module objet est *translatable*. L'opération de translation consiste à ajouter à chacun des

emplacements qui contient une adresse, la valeur de l'adresse effective d'implantation finale de la section.

Considérons, comme exemple, le petit programme suivant en assembleur 68000, dont la traduction en hexadécimal est donnée à côté, ( à droite du `:`) après l'adresse relative du premier octet dans la section (à gauche du `:`). Deux instructions, la deuxième et la cinquième font référence à un emplacement mémoire: l'adresse générée sur 4 octets a été soulignée. La première est relative à l'identificateur `C` qui repère une instruction, dont l'emplacement est à l'adresse relative `0E` dans la section du code des instructions. La seconde est relative à l'identificateur `MEMO` qui repère une donnée, dont l'emplacement est à l'adresse `02` dans la section des données variables.

```
AJOUT10:  move.w   #10,D0        0 : 30 3C 00 0A
          jmp      C             4 : 4E F9 00 00 00 0E
AJOUT16:  move.w   #16,D0        A : 30 3C 00 10
C:        add.w    D1,D0         E : D1 01
          move.w   D0,MEMO      10 : 33 C0 00 00 00 02
          rts                   16 : 2E 75
          .data
LOC:      ds.w     1             0 : 00 00
MEMO:     ds.w     1             2 : 00 00
          .end
informations de translation:
     en 6 section code, adresse 4 octets relative au code
     en 12 section code, adresse 4 octets relative aux données
```

Si l'éditeur de liens décide de mettre la section de code à l'adresse hexadécimale `012340`, et la section des données à l'adresse hexadécimale `023220`, le module doit être modifié en conséquence. A l'adresse relative `6` de la section de code, il faut ajouter l'adresse de début de la section de code, donc `012340`, (sur 4 octets), et à l'adresse relative `012` de la section de code, il faut ajouter l'adresse de début de la section des données, donc `023220`. On obtient alors le code suivant, où nous laissons figurer le texte assembleur pour permettre au lecteur de comparer avec ce qui précède.

```
AJOUT10:  move.w   #10,D0       012340 : 30 3C 00 0A
          jmp      C            012344 : 4E F9 00 01 23 4E
AJOUT16:  move.w   #16,D0       01234A : 30 3C 00 10
C:        add.w    D1,D0        01234E : D1 01
          move.w   D0,MEMO      012350 : 33 C0 00 02 33 22
          rts                   012356 : 2E 75
          .data
LOC:      ds.w     1            023320 : 00 00
MEMO:     ds.w     1            023322 : 00 00
          .end
```

# 5.2. La notion de lien

La construction d'un programme à partir d'un ensemble de modules n'est pas simplement la juxtaposition en mémoire de l'ensemble de ces modules. Il est nécessaire d'assurer une certaine forme de coopération entre les modules, et donc qu'ils puissent communiquer. Cette communication peut se faire par *variable globale*, c'est-à-dire qu'une variable appartenant à un module est accessible depuis un ou plusieurs autres modules. Elle peut se faire par *appel de procédure*, c'est-à-dire qu'une procédure ou une fonction appartenant à un module est accessible depuis un ou plusieurs autres modules. On appelle *lien* l'objet relais, qui permet à un module d'accéder à un objet appartenant à un autre module. L'établissement de ce lien passe par un intermédiaire, le *nom* du lien, permettant de le désigner dans les deux modules. Ce nom est en général une chaîne de caractères. Nous appellerons *lien à satisfaire* la partie du lien située chez l'accédant, et *lien utilisable* la partie située chez l'accédé (figure 5.1).

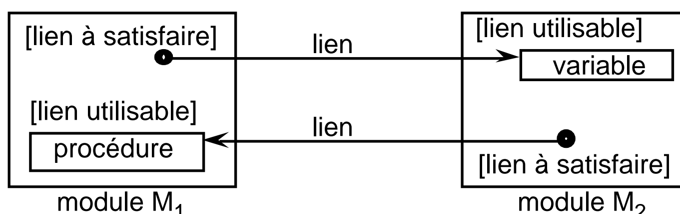


***Fig. 5.1.*** *La notion de lien.*

## 5.2.1. Expression de la nature du lien dans le module source

Lors de la traduction d'un module, le traducteur doit être capable de définir ces deux types de liens. Par ailleurs, le traitement, par le traducteur, des objets qu'ils désignent est différent, puisque un lien à satisfaire correspond à un objet pour lequel le traducteur ne réserve pas de place en mémoire. Certains de ces objets sont connus, a priori, du traducteur. Il s'agit, en particulier, de tous les sous-programmes de travail liés au langage, dont le traducteur décide d'engendrer l'appel au lieu de la séquence d'instructions correspondante: conversion de type, contrôle et transmission de paramètres des procédures du langage source, fonctions standards, etc... Pour ces objets, c'est le traducteur qui décide des liens à satisfaire.

Quant aux objets de l'utilisateur, le traducteur doit déterminer à quelle catégorie ils appartiennent:

- Les objets peuvent être internes au module, et inaccessibles de l'extérieur du module. Aucun lien n'est à engendrer dans le module objet.
- Les objets peuvent être internes au module, et accessibles de l'extérieur du module. On dit qu'ils sont *exportés*, ou encore *publics*. Le traducteur doit engendrer un lien utilisable, dont le nom est alors l'identificateur désignant l'objet dans le module.
- Les objets peuvent ne pas appartenir au module, mais être accédés depuis ce module. On dit qu'ils sont *importés*, ou encore *externes*. Le traducteur doit engendrer un lien à satisfaire, dont le nom est alors l'identificateur désignant l'objet dans le module.

Le langage de programmation doit préciser la façon dont le programmeur exprime cette catégorie au traducteur. S'il ne le fait pas, c'est alors au concepteur du traducteur de le préciser, mais ceci entraîne une divergence entre les traducteurs différents du même langage, et nuit à la portabilité des programmes.

La catégorie d'un objet peut être implicite, ou partiellement implicite, comme dans le cas du FORTRAN, qui considèrera, par exemple, comme sous-programme externe un objet désigné par un identificateur non déclaré, et dont une occurence dans le module source est suivie du symbole "(".

La catégorie peut être définie explicitement, c'est-à-dire que les objets sont considérés normalement comme uniquement internes, sauf si le programmeur les spécifient explicitement comme exportés ou importés. Ceci peut présenter diverses formes, comme le montrent les exemples suivants, où `A` est importé et `B` est exporté.

Pascal 6.02 Multics

```
program P (A);        (* importation de A               *)
var def B: integer;        (* exportation de B               *)
procedure A (x : integer); pascal;    (* déclaration du type de A *)
begin ...end.
```

Pascal Multics, versions ultérieures

```
program P;
$IMPORT 'A(pascal)': A$    (* importation de A               *)
$EXPORT B$                 (* exportation de B               *)
var B: integer;            (* déclaration de B               *)
procedure A(x: integer); external; (* déclaration du type de A *)
begin ...end.
```

Pascal Microméga

```
program P;
var B: integer;            (* exportation implicite              *)
procedure A(x: integer); external;(*déclaration et importation *)
begin ...end.
```

Certains langages récents orientés vers la modularité séparent le module source en deux parties, la partie définissant les exportations, et celle définissant l'implantation du module. Ainsi, en langage Ada, la partie *spécification* contient une liste de déclaration de types, d'objets ou de procédures qui appartiennent au module, et sont exportés par ce module, alors que la partie *corps* renferme l'implantation du module, et ce qui lui est propre. L'importation se fait en nommant dans une clause particulière située devant le module, la liste des modules dont on désire importer des objets. L'exemple ci-dessus se décrirait de la façon suivante:

```
package E is
     procedure A (x : integer);     -- procédure du module exportée
end E;
package P is
     B : integer;                   -- exportation de B
end P;
with E;                        -- importation des objets du module E
package body P is
     -- contenu de P
end P;
```

### 5.2.2. Représentation des liens dans le module objet

La représentation d'un lien utilisable dans un module objet ne pose guère de problèmes. Il s'agit d'un couple `<nom, valeur>`, où la valeur définit d'une part la section contenant l'objet, d'autre part l'adresse relative de l'objet dans la section. Lorsqu'on connait l'adresse effective d'implantation de cette section, il est ainsi possible de calculer la valeur finale du lien. En reprenant l'exemple du module assembleur, et en supposant que `AJOUT10` et `AJOUT16` sont déclarés publics, l'assembleur fournira les informations suivantes:

```
AJOUT10        00      relative au code
AJOUT16        0A      relative au code
```

Ceci permettra à l'éditeur de liens de calculer les adresses finales suivantes:

```
AJOUT10        012340
AJOUT16        01234A
```

La représentation d'un lien à satisfaire dans un module objet peut présenter diverses formes suivant les systèmes, dont voici quelques exemples:

1) C'est un couple `<nom, liste>` qui associe au nom du lien la liste des emplacements du module qui font référence au lien (nous notons `contenu[j]` le contenu de l'emplacement d'adresse `j` du module).

- Cette liste peut être obtenue par chaînage de ces emplacements entre eux, `liste` étant alors l'adresse du premier emplacement, et chacun d'eux contenant l'adresse du suivant. L'établissement des liaisons est alors obtenu par l'algorithme suivant:

```
procédure relier (liste, valeur);
début
     tant que liste <> nil faire
          x := contenu [liste];  { x repère le suivant de la liste }
          contenu [liste] := valeur;  { on établit le lien }
          liste := x;
     fait;
fin;
```

- Cette liste peut être un tableau des adresses des emplacements. L'établissement des liaisons est alors obtenu par l'algorithme suivant:

```
procédure relier (liste, n, valeur);
début
     pour i de 1 à n faire
          contenu [liste [i]] := contenu [liste [i]] + valeur;
     fait;
fin;
```

Contrairement à la deuxième représentation, la première ne permet pas de modifier statiquement le lien, comme, par exemple, dans `A + 5`.

2) C'est un couple `<nom, numéro>`, qui définit le numéro sous lequel est référencé dans le module ce lien à satisfaire. Les références au lien sont alors transcrites dans le module objet sous forme d'informations de translation: "`ajouter à l'emplacement e, la valeur du lien n`".

## 5.3. Fonctionnement de l'éditeur de liens

L'édition de liens peut se faire parfois de façon dynamique (Multics). Dans ce cas, le module correspondant au programme principal est lancé par l'utilisateur, sous sa forme objet éditable.

Lorsque ce module accède à un lien à satisfaire, le système (matériel et logiciel) appelle alors automatiquement l'éditeur de liens pour trouver le module contenant ce lien en tant que lien utilisable, et établir la liaison. Peu de systèmes offrent cependant cette possibilité. Dans tous les autres cas, l'édition de liens est faite de façon statique par appel de l'éditeur de liens (ou *relieur*, ou *linker*), en fournissant en paramètres la liste des modules à relier, $L = (M_1, M_2, \dots, M_n)$.

## 5.3.1. Édition simple de la liste *L*

L'édition de liens d'un ensemble fixe de modules donnés par une liste *L* peut se faire en trois étapes. Les deux premières étapes sont parfois fusionnées.

*Étape 1*: Construction de la table des adresses d'implantation des sections des différents modules, par acquisition de leur taille depuis les modules objets, et placement de ces sections les unes derrière les autres. En général ce placement tient compte de la nature des sections, c'est-à-dire regroupe ensemble les sections de même nature, ce qui permettra, à l'exécution, de protéger différemment ces sections, et donc d'en limiter les accès. Par exemple:

| nature | opérations autorisées |
|---|---|
| code instruction | exécution |
| données constantes | lecture |
| données variables | lecture, écriture |

*Étape 2*: Construction de la table des liens utilisables. Cette table associe aux noms des liens l'adresse effective en mémoire de l'objet que le lien désigne. Cette adresse est obtenue par transformation de l'adresse relative dans sa section, fournie par le module objet.

*Étape 3*: Construction du programme final lui-même. Chaque module est translaté, en tenant compte des informations de translation du module objet. Les noms des liens à satisfaire sont recherchés dans la table des liens utilisables. S'ils y sont trouvés, les emplacements du module, qui se réfèrent à ce lien à satisfaire, sont modifiés conformément à la valeur associée au lien dans la table. Si un lien à satisfaire n'est pas trouvé dans la table, il reste indéfini dans le programme (ou *non résolu*, ou *unresolved*), et un message d'anomalie est alors imprimé par l'éditeur de liens. Ceci ne veut pas dire que le programme ne peut s'exécuter, car une exécution particulière peut ne pas accéder à ce lien, par exemple, en phase de mise au point partielle. Évidemment un programme “opérationnel” ne devrait plus contenir de liens indéfinis.

## 5.3.2. Construction de la table des liens

Posons $LU_M$ l'ensemble des noms des liens utilisables d'un module *M*, et $LAS_M$ l'ensemble des noms des liens à satisfaire du module *M*.

Remarquons d'abord que les ensembles $LU_M$ devraient être deux à deux disjoints pour les modules d'une même liste, dont on veut faire l'édition de liens. Si ce n'est pas le cas, un même lien possède plusieurs définitions, parmi lesquelles l'éditeur de liens fera, en général, un choix, qui ne sera pas forcément le bon; il signalera cette anomalie. Les définitions multiples peuvent se décrire par l'ensemble suivant:

$$\bigcup_{M_i, M_j \;\; L, M_i \;\; M_j} \left( LU_{M_i} \quad LU_{M_j} \right)$$

Par ailleurs, nous avons vu que certains liens à satisfaire pouvaient ne pas être trouvés. Nous pouvons définir, pour une liste de modules *L* :

$$LAS_L = \bigcup_{M \;\; L} LAS_M - \bigcup_{M \;\; L} LU_M$$

La deuxième étape de l'édition de liens peut être modifiée pour permettre la connaissance de $LAS_L$ dès la fin de cette étape, et avant d'entamer la troisième étape. Il suffit, pour cela, de remplacer la table des liens utilisables par une table plus générale, dite *table des liens* (figure 5.2), qui associe à chaque nom de lien:

- l'état du lien considéré comme *utilisable*, *à satisfaire* ou *non résolu*,

- la valeur du lien, qui n'a de signification que lorsque le lien est dans l'état *utilisable*.

| nom | état | valeur |
|---|---|---|
| A | à_satisfaire | |
| B | utilisable | 3281 |

***Fig. 5.2.*** *Exemple du contenu de la table des liens après traitement du module donné en §5.2.1.*

La construction de la table des liens peut alors être obtenue en parcourant les modules de la liste, et en effectuant pour chacun d'eux l'algorithme de la figure 5.3. Cet algorithme lui-même met dans la table des liens les noms de tous les liens rencontrés dans ces modules, en tenant compte de leur état. En particulier, tous les noms qui correspondent à un lien utilisable dans un module de la liste, sont dans la table avec l'état *utilisable*, à la fin de l'étape. De même, tous les noms qui correspondent à un lien à satisfaire dans un des modules de la liste, sont dans la table avec l'état *à_satisfaire* si aucun des modules de la liste ne contient de lien utilisable de même nom. Après avoir exécuté l'algorithme de la figure 5.3 sur tous les modules de la liste, on a:

$$LAS_L = \{ n \mid n \quad \text{table} \quad \text{état}(n) = \textit{à_satisfaire} \}$$

```
procédure lire_les_liens_du_module;
début pour tous les liens du module faire
          soit n son nom, recherche de n dans la table
          si n a été trouvé alors
               si n   LUM alors
                    si état (n) = utilisable alors double_définition (n)
                    sinon état (n) := utilisable; valeur (n) := sa_valeur;
                    finsi;
               finsi       { s'il est lien à satisfaire dans le module,
                         et déja dans la table, on ne fait rien }
          sinon
               ajouter n dans la table;
               si n   LUM alors état(n) := utilisable; valeur(n) := sa_valeur;
               sinon état (n) := à_satisfaire;
               finsi;
          finsi;
       fait;
fin;
```

***Fig. 5.3.*** *Procédure de lecture des liens d'un module et de construction de la table des liens.*

## 5.3.3. Notion de bibliothèque

La notion de bibliothèque de modules objets a pour but de compléter la liste *L* avec des modules provenant de la bibliothèque, et qui, par les liens utilisables qu'ils contiennent, permettraient de satisfaire les liens de $LAS_L$. On peut distinguer trois fonctions qui justifient le regroupement de modules objets dans une bibliothèque:

- La *bibliothèque de calcul* contient des sous-programmes standards tout faits, tels que `sinus`, `cosinus`, `racine_carrée`, etc... On peut y adjoindre également les sous-programmes de travail liés à un langage, dont nous avons déjà parlé, et dont le traducteur génère l'appel au lieu de la séquence d'instruction correspondante.
- La *bibliothèque d'interface système* contient les sous-programmes qui exécutent les appels systèmes. Si un tel appel peut paraître équivalent à un appel de sous-programme, nous avons vu dans le chapitre 2 deux raisons militent pour les traiter différemment. La première est le besoin éventuel de faire passer le processeur en mode maître et lui permettre de disposer du jeu complet d'instructions. La deuxième est qu'un contrôle doit être effectué au moment de l'appel, et il ne faut pas qu'un programmeur mal intentionné puisse contourner ce contrôle, or la connaissance de l'adresse A de la fonction dans le système permettrait au programmeur de faire un appel au sous programme en A+5 par exemple. Notons que, par ailleurs, si les programmes exécutables sont reliés directement par l'éditeur de liens avec les sous-programmes du système, un changement de système nécessiterait de refaire l'édition de liens de tous les programmes exécutables de l'installation. Aussi, comme nous l'avons déjà dit, un appel système est en général réalisé par une instruction particulière de la machine (par exemple, *SVC* pour *supervisor call* sur IBM370, ou

*INT* pour *interrupt* sur MS-DOS). Ces instructions déroutent le processeur vers un emplacement de mémoire connu du matériel, où se fera le contrôle et la redirection vers les instructions du système qui réalisent la fonction demandée. Par ailleurs, le passage des paramètres se fait souvent en utilisant les registres du processeur et non en utilisant une pile comme le font beaucoup d'implantations des langages de programmation. Les sous-programmes de la bibliothèque d'interface système ont pour but de préparer les paramètres et d'exécuter l'instruction d'appel, en évitant ainsi aux traducteurs d'être trop dépendants du système.

- La *bibliothèque de l'utilisateur* contient les sous-programmes écrits par un utilisateur ou un groupe d'utilisateurs pour les besoins courants de leurs applications. Évidemment il peut en exister plusieurs qui satisfont des besoins spécifiques.

Une telle bibliothèque est, en général, une structure de données constituée, au minimum de la collection des modules objets qu'elle contient, et éventuellement d'une table dont les entrées sont les noms des liens utilisables de ces modules, et qui associe à chacune de ces entrées le module qui le contient comme lien utilisable. Cette table permet d'éviter de devoir parcourir l'ensemble des modules de la bibliothèque pour savoir celui qui contient un lien utilisable de nom donné (figure 5.4).

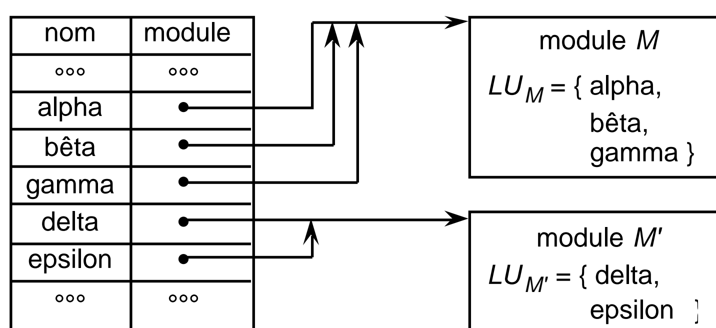



***Fig. 5.4.*** *Exemple de structure de bibliothèque de modules objets.*

En général, les ensembles des liens utilisables des modules situés dans une même bibliothèque B sont deux à deux disjoints, c'est-à-dire que:

$$M, M' \quad B, LU_M \quad LU_{M'} = \emptyset$$

ce qui garantit l'unicité du module associé à une entrée de la table.

Par contre, une édition de liens particulière peut demander l'utilisation de plusieurs bibliothèques, dont les ensembles de liens utilisables ne sont pas forcément disjoints. Les bibliothèques doivent alors être fournies dans l'ordre où l'on désire que l'éditeur fasse la recherche. Ainsi en UNIX, la demande d'édition de liens suivante:

```
ld A.o B.o C.o -lpc -lm -lc
```

est interprétée comme l'édition de liens, tout d'abord, de la liste des trois modules `A.o`, `B.o` et `C.o`, et qui peut être complétée par des modules recherchés dans les bibliothèques `pc`, `m` ou `c`, dans cet ordre.

## 5.3.4. Adjonction de modules de bibliothèques

Plus généralement, considérons l'édition de liens d'une liste L avec les bibliothèques B1, B2, ..., Bp. L'éditeur de liens commence par construire la table des liens à partir de la liste L. Il doit ensuite chercher à étendre cette liste initiale pour y ajouter des modules provenant des bibliothèques, et qui permettent de satisfaire des liens qui ne le sont pas encore. Chaque fois qu'il trouve un tel module, il l'ajoute à la liste L, et exécute l'algorithme de lecture de ses liens présenté plus haut. Pour éviter de rechercher indéfiniment un lien à satisfaire qui n'existe, comme lien utilisable, dans aucun module de bibliothèques, on se sert de l'état inexistant. Nous utiliserons les notations suivantes:

- *LAS* = { n | n  table  état (n) = *à_satisfaire* }
- *LU* = { n | n  table  état (n) = *utilisable* }
- *LIN* = { n | n  table  état (n) = *inexistant* }

Deux méthodes peuvent être utilisées par l'éditeur de liens. Ces deux méthodes peuvent ne pas donner le même résultat.

i) Chaque lien est recherché successivement dans les bibliothèques. On obtient l'algorithme de la figure 5.5. On notera qu'à la fin du traitement, on a:

$$LIN = LAS_L - \bigcup_{M \quad L,B_i} LU_M$$

```
tant que LAS   Ø faire
     soit n   LAS
     i := 1;
     tant que état (n) = à_satisfaire faire
          si n est une entrée dans la table de Bi alors
               soit M le module qui lui est associé
               ajouter M à L
               lire_les_liens_du_module (M)
          sinon si i < p alors i := i + 1
               sinon état (n) := inexistant
               finsi
          finsi
     fait
fait
```

***Fig. 5.5.*** *Première méthode de recherche en bibliothèque.*

En d'autres termes, si un lien est inexistant à la fin du traitement, c'est qu'il n'existe effectivement comme lien utilisable dans aucun module de l'une des bibliothèques.

ii) La recherche est effectuée pour tous les liens en parcourant les bibliothèques successives. C'est la méthode utilisée habituellement dans l'éditeur de liens de UNIX. L'algorithme est donné en figure 5.6. On notera que dans ce cas, il est possible qu'à la fin du traitement on ait:

$$LAS_L \cap \bigcup_{M \quad B_i} LU_M$$

Il suffit qu'un tel lien à satisfaire soit introduit par un module $M$ provenant d'une bibliothèque $B_i$ , et qu'il soit lien utilisable dans $M'$ appartenant à $B_j$ , avec j<i, sans l'être dans aucun $M''$ appartenant à $B_k$ , avec i  k.

```
pour i de 1 à p faire
     tant que LAS   Ø faire
          soit n   LAS
          si n est une entrée dans la table de Bi alors
               soit M le module qui lui est associé
               ajouter M à L
               lire_les_liens_du_module (M)
          sinon état (n) := inexistant
          finsi
     fait
     tant que LIN   Ø faire  { préparer pour la bibliothèque suivante }
          soit n   LIN
          état (n) := à_satisfaire
     fait
fait
```

***Fig. 5.6.*** *Deuxième méthode de recherche en bibliothèque.*

L'intérêt de cette deuxième méthode est de permettre de prendre en compte des bibliothèques non structurées, qui sont simplement une suite de modules (sans table), par simple parcours de cette suite. Il est évident que l'ordre dans lequel les modules sont placés dans cette suite est important. Si un module $M$ d'une bibliothèque contient un lien à satisfaire qui est lien utilisable d'un module $M'$ de la même bibliothèque, il faut que le module $M$ précède le module $M'$ pour que la sélection de $M$ entraîne celle de $M'$.

Il va sans dire que, lorsque l'algorithme d'adjonction de modules de bibliothèques est terminé, l'éditeur de liens exécute l'étape 3 de construction du programme exécutable final, en utilisant la liste *L* ainsi augmentée par la recherche des modules provenant de bibliothèques.

### 5.3.5. Edition de liens dynamique

Certains systèmes offrent la possibilité de retarder l'édition de liens, soit jusqu'au moment du chargement du programme, soit jusqu'au moment où la liaison est utilisée au moment de l'exécution. Ceci peut se faire soit module par module (c'est la méthode par défaut utilisée dans Multics en 1970), soit au niveau global d'une bibliothèque complète (c'est la méthode utilisée par windows 95). Une des méthodes utilisées consiste à remplacer les modules ainsi reliés par un module particulier, que nous appelons module relais, qui va servir d'intermédiaire aux liaisons. Ce module est en fait une table où chaque entrée est un relais vers un lien utilisable de l'un des modules effectifs et tient lieu de lien utilisable pour l'édition de lien statique. Lors du chargement effectif des modules correspondants, les entrées de cette table assureront le relais vers le lien effectif du module. La figure 5.7 montre la liaison entre un lien à satisfaire du module M1 vers un lien utilisable du module M2, par l'intermédiaire du module relais. L'éditeur de lien ne se préoccupe que du lien entre le module M1 et le module relais, l'établissement du lien dynamique étant reporté à un instant ultérieur (chargement ou exécution).

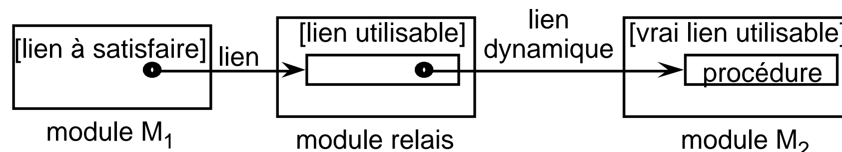



***Fig. 5.7.*** *Liaison dynamique.*

## 5.4. Notion de recouvrement

Lorsqu'on construit un programme de grande taille, il est possible que la mémoire centrale réelle dont on dispose soit insuffisante. Certains systèmes résolvent ce problème en offrant une mémoire virtuelle de grande taille (par exemple 2 Go), et se chargent de ne mettre en mémoire réelle, à un instant donné, qu'une partie du programme et des données, celle qui est nécessaire à son exécution à cet instant. D'autres systèmes n'offrent pas cette possibilité qui nécessite des dispositifs matériels. Dans ce cas, l'éditeur de liens propose souvent une solution par le biais de ce que l'on appelle le *recouvrement* (en Anglais *overlay*).

La mémoire virtuelle entraîne, sans doute, la disparition de cette notion dans un avenir proche. Le principe de base reste cependant, même si l'utilisateur ne s'en aperçoit pas: à un instant donné, un programme n'utilise qu'une petite partie du code et des données qui le constituent. Les parties qui ne sont pas utiles en même temps peuvent donc se “recouvrir”, c'est-à-dire occuper le même emplacement en mémoire réelle. Dans le cas de la mémoire virtuelle, ce recouvrement sera assuré dynamiquement par le système, au fur et à mesure de l'exécution du programme. S'il n'y a pas de mémoire virtuelle, l'utilisateur doit assurer lui-même un découpage de son programme, indiquer ce découpage à l'éditeur de liens et préciser les parties qui peuvent se recouvrir, et prévoir explicitement dans son programme des opérations de chargement de ces parties lorsqu'elles sont nécessaires. L'éditeur de liens tiendra compte de ces informations lors de la définition des adresses d'implantations des différents modules du programme. En général, le découpage et le recouvrement sont indiqués sous la forme d'un arbre, dont la figure 5.8 est un exemple. L'éditeur de liens déduira de cet arbre que, entre autre, les modules `E` et `H` peuvent être implantés au même endroit, de même que les modules `C`, `J` et `Q`.

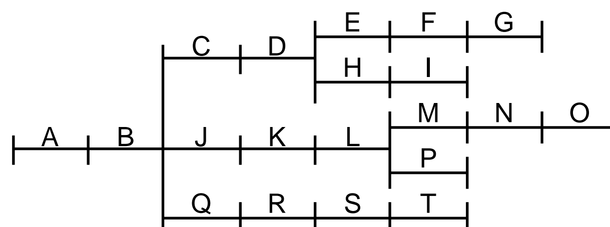


***Fig. 5.8.*** *Exemple d'arbre de recouvrement.*

# 5.5. Les références croisées

L'éditeur de liens, au cours de son travail, dispose de certaines informations qui peuvent être intéressantes pour le programmeur (en dehors des anomalies rencontrées).

- La carte d'implantation des modules peut aider l'utilisateur dans la mise au point ou la maintenance du programme, lorsqu'une erreur est détectée à l'exécution à une adresse donnée. Nous verrons, cependant, que les metteurs au point peuvent apporter une aide plus intéressante de ce point de vue. Notons, pour le moment, qu'ils exploiteront cette carte d'implantation si l'éditeur de liens la leur fournit dans le programme exécutable.
- Au cours du travail, l'éditeur de liens associe les liens à satisfaire d'un module aux liens utilisables des autres modules. Il peut être intéressant qu'il fournisse cette information à l'utilisateur. Il s'agit alors d'indiquer à cet utilisateur, pour chaque lien utilisable, la liste des modules (et éventuellement les emplacements à l'intérieur de ces modules) qui ont ce lien comme lien à satisfaire. En d'autres termes, on peut ainsi savoir “qui utilise quoi”. C'est ce que l'on appelle les *références croisées*.

Ces références croisées peuvent être obtenues en sous-produit de l'édition de liens. Il est aussi possible de réaliser un programme spécifique pour cela, dont le fonctionnement sera assez voisin de l'éditeur de liens, du point de vue de la gestion de la table des liens. Évidemment, un tel outil est alors plus simple, puisqu'il n'a pas à construire le programme exécutable. Il offre aussi l'avantage de pouvoir être appliqué sur un ensemble de modules qui ne constituent pas un programme complet, sans avoir de messages d'erreurs inutiles. Voici un exemple d'édition de telles références croisées:

| lien | module où il est "utilisable" | modules et emplacements où il est "à satisfaire" |
|---|---|---|
| alpha | $M_1$ | $M_2$(10, 50, 86), $M_3$(26) |
| bêta | $M_2$ | $M_1$(104), $M_4$(34) |
| gamma | $M_3$ | $M_4$(246, 642) |
| delta | $M_3$ | $M_5$(68, 94), $M_6$(4), $M_7$(456, 682, 624) |

L'intérêt de disposer de ces réferences croisées est multiple, suivant que l'on regarde les variables ou les sous-programmes, et que l'on se place du point de vue de la mise au point ou de la maintenance des applications.

- Par exemple, si un identificateur exporté par un module désigne une variable, on peut ainsi savoir la liste des modules qui accèdent directement à cette variable. Outre l'aspect contrôle ainsi obtenu, cela permet, en phase de mise au point, si la valeur n'est pas conforme à ce qu'elle devrait être, de savoir quels sont les modules qui peuvent être en cause. Cela permet également, en phase de maintenance, si on change l'interprétation du contenu de cette variable, de savoir quels sont les modules qui sont concernés par ce changement.
- De même, si un identificateur exporté par un module désigne un sous-programme, on peut connaître la liste des modules où se trouvent les appels de ce sous-programme. En phase de mise au point d'une application, lorsque l'on constate que les variables permanentes internes au module ne sont pas conformes, il faut déterminer quels sont les appels précédents qui ont conduit à cette non-conformité. On peut alors savoir quels sont les modules qui peuvent être en cause. En phase de maintenance, en cas de changement de la spécification ou du fonctionnement de ce sous-programme, on peut savoir quels sont les modules qui sont concernés par ce changement.

Enfin, la connaissance des relations entre les modules que l'on peut ainsi obtenir, peut permettre d'améliorer le découpage de ces modules en regroupant ensemble les parties de module ou les modules entiers qui ont de fortes interactions. Ces relations peuvent être également utiles pour faciliter la construction des bibliothèques de modules lorsque l'ordre des modules a de l'importance pour l'éditeur de liens.

# 5.6. Le chargement

Plusieurs raisons militent pour que le résultat de l'édition de lien ne soit pas le programme installé quelque part en mémoire centrale prêt à être exécuté. La première est que cela limiterait

arbitrairement la taille des programmes, puisqu'elle ne pourrait atteindre que l'espace total moins l'espace occupé par l'éditeur de liens lui-même. La deuxième raison est que l'édition de liens consomme du temps machine, et qu'il n'est pas économique de la refaire chaque fois que l'on désire faire exécuter le programme lui-même. Il s'ensuit que l'éditeur de liens produit plutôt un fichier contenant le programme exécutable.

Pour exécuter un programme, il faut donc prendre sa version exécutable, et la mettre en mémoire (réelle ou virtuelle). On appelle *chargement* cette opération, et *chargeur* le programme qui l'effectue. Cette opération pourrait consister en une simple lecture du contenu du fichier dans un emplacement mémoire fixé a priori par l'éditeur de liens. Cette méthode est simple, mais restrictive, puisqu'il faut décider de l'endroit définitif en mémoire où le programme s'exécutera au moment où l'édition de liens est faite. Elle n'est souvent utilisée que pour les programmes autonomes (*stand alone*), car elle permet d'avoir des programmes d'amorçage très simples.

Il est facile de concevoir que, puisque l'éditeur de liens a disposé des informations qui lui ont permis de translater les modules, il laisse ces informations dans le fichier qui contient le programme exécutable. Le chargeur peut alors les utiliser pour translater le programme complet, et le mettre ainsi n'importe où en mémoire. Il est évident que les informations de translations provenant des modules doivent être modifiées légèrement pour être maintenant relatives au programme. En général, l'éditeur de liens fournira également la taille mémoire centrale totale nécessaire à l'exécution du programme, qu'il déduira des tailles de l'ensemble des modules qui le composent.

Enfin, une dernière information minimale est nécessaire au chargeur pour terminer le travail, et lancer l'exécution du programme, c'est l'*adresse de lancement*, ou l'adresse de l'emplacement où se trouve la première instruction à exécuter. Pour l'utilisateur, c'est la première instruction d'un module particulier, qu'est son *programme principal*. Pour les langages évolués, le compilateur est à même de déterminer si un module est ou non un programme principal, et si oui quelle est l'adresse dans le module objet de la première instruction à exécuter. Pour le langage d'assemblage, cette information est, en général, précisée explicitement par le programmeur à l'assembleur. Les traducteurs doivent donc informer l'éditeur de liens de l'adresse de lancement éventuelle du module. Comme l'éditeur de liens est en présence de plusieurs modules, il ne devrait trouver qu'un et un seul module contenant une telle adresse de lancement, qu'il peut alors transmettre au chargeur, après éventuelle modification pour la rendre relative au programme et non plus au module dont elle provient.

Par ailleurs, le programme exécutable a été conçu pour être exécuté dans un certain environnement. S'il est prévu pour fonctionner en autonome, son environnement est alors simplement la machine nue. S'il est prévu pour fonctionner sur un système particulier, il peut être nécessaire d'établir les liaisons avec ce système. Nous avons déjà mentionné l'existence de bibliothèque d'interface système qui peut résoudre complètement ces liaisons. Nous avons également mentionné (§5.3.5) que certains systèmes prévoient de retarder les liaisons effectives avec certains sous-programmes standards, ce qui permet en particulier de les partager entre l'ensemble des programmes présents en mémoire, de façon à économiser de la place (VMS ou Windows NT par exemple). Le chargeur doit alors remplir la table contenue dans le module relais avec les adresses effectives actuelles de ces sous-programmes. On peut considérer que le programme en mémoire est prévu pour s'exécuter sur une machine abstraite, et que le chargeur doit non seulement mettre le programme en mémoire, mais aussi préparer la machine abstraite nécessaire à son exécution.

# 5.7. Conclusion

☞ La translation d'un module consiste à modifier son contenu pour qu'il puisse s'exécuter à un endroit différent de celui pour lequel il était prévu initialement.

☞ Un lien est l'information qui permet à un module d'accéder à un objet appartenant à un autre module. Il est désigné par un nom. On appelle lien à satisfaire la partie du lien située chez l'accédant, et lien utilisable celle située chez l'accédé.

☞ L'expression de la nature d'un lien dépend du langage de programmation et du traducteur. Elle peut prendre des formes variées.

☞ L'édition de liens se fait en deux phases. La première construit la table de l'ensembles des liens trouvés dans les modules à relier. La seconde construit le programme exécutable en translatant les modules et en remplaçant les liens à satisfaire par la valeur du lien utilisable correspondant.

☞ Entre les deux phases, l'éditeur de liens peut compléter la liste des modules à relier avec des modules provenant de bibliothèques, en tenant compte des liens restant à satisfaire.

☞ Une bibliothèque peut être une simple collection de modules parcourue séquentiellement par l'éditeur de liens, mais l'ordre des modules est important. Elle peut être structurée de façon à savoir rapidement quel module, s'il existe, contient un lien donné comme lien utilisable.

☞ Une bibliothèque regroupe en général des modules de fonctionnalité voisine. On peut avoir, par exemple, une bibliothèque de calcul, une bibliothèque d'interface système, et des bibliothèques spécifiques aux utilisateurs.

☞ Le recouvrement est une technique qui permet de diminuer l'encombrement total en mémoire centrale d'un programme, en décidant que des parties du programme peuvent se trouver au même endroit, car elles ne sont pas utiles au même moment pour la bonne exécution du programme.

☞ L'édition des références croisées consiste à éditer une table qui, pour chaque lien trouvé dans une liste de module, donne le module dans lequel il est utilisable, et ceux dans lesquels il est à satisfaire. Elle peut être obtenue en sous-produit de l'édition de liens, ou par un outil spécifique. Elle est utile pour la mise au point et pour la maintenance.

☞ Le chargement est l'opération qui consiste à mettre en mémoire centrale un programme exécutable, avec une éventuelle translation, et à en lancer l'exécution. Il consiste également à construire la machine abstraite demandée par le programme.

# 6

# Autres outils de la chaîne de production

Nous allons étudier ici quelques uns des autres outils qui participent à la chaîne de production des programmes. Ils ne sont pas aussi indispensables que les traducteurs, l'éditeur de liens ou le chargeur, mais ils facilitent grandement le travail du programmeur lorsqu'ils sont disponibles. Leur présence sur un système particulier dépend du bon vouloir du constructeur de la machine, ou plus souvent, de la reconnaissance de leur intérêt en fonction des applications.

## 6.1. Les outils d'aide à la mise au point

### 6.1.1. La notion de trace

La mise au point d'un programme est en général une tâche difficile, et diverses méthodes ont été proposées pour aider le programmeur dans cette tâche. La première, et la plus simple, a été de lui permettre de suivre l'évolution de l'état du programme au cours du temps, c'est ce que l'on, appelle la *trace* du programme. Pratiquement, il n'est pas possible de suivre cette évolution sur l'état complet du programme. Il est nécessaire de restreindre ce suivi sur une petite partie de cet état. Les principales informations utiles sont les suivantes:

- tracer les appels de sous-programmes avec les paramètres correspondants, et leurs retours avec la valeur du résultat,
- tracer le déroulement des instructions, en particulier pour reconnaître les ruptures de séquences,
- tracer les valeurs prises successivement par des variables,
- connaître l'état des données en certains points précis du programme.

L'obtention de ces traces peut être définie en introduisant dans les modules sources des instructions spéciales qui sont ou non prises en compte par le traducteur en fonction d'options décidées lors de la demande de traduction. C'est alors le traducteur qui insère dans le module objet des instructions, en général d'impression, qui permettent de suivre le déroulement de l'exécution.

La difficulté essentielle de cette méthode est qu'il faut prévoir ces options au moment de la traduction du module. Il est possible que lors de l'analyse du résultat de l'exécution, il soit nécessaire de connaître d'autres informations pour déterminer l'erreur. Il faut alors refaire la traduction du module en adaptant les options de mise au point en conséquence. Pour éviter cela, la tendance est de prendre le maximum d'options, mais alors il faut dépouiller une quantité importante d'informations pour trouver celles qui sont effectivement utiles.

L'introduction des instructions de trace par l'éditeur de liens lui-même, ou par le chargeur, a été très peu utilisée, car ce n'est pas d'un emploi commode.

### 6.1.2. La notion de point d'arrêt, de reprise et de pas à pas

L'inconvénient majeur des options de traces traitées par le traducteur est le manque de souplesse pendant l'exécution. Lors de la recherche d'une erreur, il est souvent nécessaire de disposer de beaucoup d'informations sur l'évolution de l'état du programme au voisinage de l'endroit où se produit cette erreur. Pour faciliter le suivi de cette évolution, il suffit de pouvoir arrêter temporairement cette exécution, pour permettre l'observation de l'état complet du programme. Ceci est obtenu par le biais de *points d'arrêt*, qui sont des adresses d'emplacements particuliers du programme où on désire que l'exécution soit interrompue, avant que l'une des instructions situées à ces emplacements ne soit exécutée.

Lorsqu'un programme est ainsi interrompu en un point d'arrêt, le programmeur doit avoir le moyen de consulter l'état du programme, en regardant le contenu de l'espace mémoire qu'il occupe. Il doit pouvoir éventuellement modifier (à ses risques et périls) le contenu de cet espace mémoire, pour en corriger certaines valeurs. Il doit pouvoir également en reprendre l'exécution, ce que l'on appelle la *reprise*. On appelle *metteur au point* ou *débogueur* (en Anglais *debugger*) un outil interactif qui permet de créer des points d'arrêt dans un programme pour en consulter ou modifier le contenu mémoire et en reprendre ensuite l'exécution.

Le *pas à pas* est un cas particulier du couple <point d'arrêt, reprise>, puisqu'il s'agit, sur un programme interrompu, d'en reprendre l'exécution pour une instruction avant de l'arrêter de nouveau. Ce pas à pas peut être au niveau machine, c'est-à-dire que la reprise n'a lieu que pour exécuter une seule instruction de la machine. Pour un programmeur en langage évolué, ceci n'est guère agréable, car il n'a souvent que peu de connaissance des relations entre les instructions de la machine et les instructions en langage évolué. C'est pourquoi, on lui préfère le pas à pas au niveau langage évolué. Cependant, il faut alors que le metteur au point ait une connaissance du découpage de la suite des instructions machines en instructions en langage évolué.

### 6.1.3. Les metteurs au point symboliques

La définition d'un point d'arrêt doit localiser un emplacement particulier de l'espace mémoire du programme. La consultation de l'état du programme demande également de localiser les emplacements dont on veut connaître ou modifier la valeur. Cette localisation peut évidemment être donnée par une valeur numérique dans une base adaptée, octal, décimal, hexadécimal, etc... Cela demande au programmeur de connaître ces valeurs numériques. Ce sont en fait des valeurs connues de façon dispersée par les traducteurs, l'éditeur de liens et le chargeur. Pour le programmeur, elles correspondent à des identificateurs de ses modules sources. Une coopération entre l'ensemble permet d'exprimer ces adresses par les identificateurs eux-mêmes. C'est ce que l'on appelle un *metteur au point symbolique*.

Pour permettre cette localisation, le traducteur doit tout d'abord compléter le module objet qu'il produit par une table de symboles, qui associe aux identificateurs internes aux modules (en plus des liens) les adresses relatives des emplacements attribués aux objets correspondants. Cette table a la même signification que les liens utilisables du module. L'éditeur de liens n'utilise pas cette table, si ce n'est pour translater les adresses pour les rendre relatives au programme exécutable et non plus au module objet. Cette table associe également à chaque identificateur des informations sur le type de l'objet, pour permettre au metteur au point d'éditer (ou saisir) les valeurs des objets conformément à leur type. Cela permet aussi d'accéder aux champs d'une structure par la notation pointée habituelle aux langages de haut niveau.

Le traducteur produit enfin une deuxième table qui associe aux numéros de lignes contenant des instructions du langage de haut niveau, l'adresse relative de la première instruction en langage machine correspondant. Après translation des valeurs de cette table par l'éditeur de liens, il sera ainsi possible de définir les points d'arrêt ou le pas à pas en terme d'instructions de haut niveau, clairement identifiées par le programmeur.

Le traducteur, puis l'éditeur de liens, mettra les informations nécessaires à l'identification du module source. Le metteur au point pourra ainsi lister les lignes du texte source à la demande de l'utilisateur, lui permettant de contrôler effectivement où il en est dans le programme.

Disposant de toutes ces informations dans le fichier du programme exécutable, le metteur au point peut offrir les opérations suivantes à l'utilisateur:

1. *Possibilités de trace*. Il est possible de mettre en route ou d'arrêter à tout instant des options de trace comme indiquées ci-dessus.
2. *Consultation et modification des variables du programme*. Ceci permet éventuellement de corriger des valeurs erronnées et poursuivre la mise au point sans être obligé de corriger le texte source et le recompiler.
3. *Création et suppression de points d'arrêt.* On peut ainsi lancer des parties d'exécution longues et s'arrêter aux endroits où l'on désire effectuer des contrôles plus précis.
4. *Exécution en pas à pas*. Elle permet de contrôler plus finement le comportement du programme à certains endroits.
5. *Consultation des appels de procédure en cours*. Lorsqu'un programme est interrompu, on peut ainsi savoir si on est dans le programme principal ou dans une procédure, et dans ce dernier cas, quels sont les appels imbriqués depuis le programme principal qui ont conduit à la procédure courante.
6. *Lister le fichier source*. En particulier on peut ainsi savoir quelles sont les instructions de haut niveau qui seront exécutées à la reprise.
7. *Éditer le texte sourc*e. Le metteur au point peut permettre de faire appel à l'éditeur de texte pour porter des corrections au fur et à mesure dans les modules sources, et poursuivre ensuite la mise au point. Évidemment, ceci ne modifie pas le programme exécutable, les modules ainsi modifiés devant être recompilés, et l'édition de liens refaite, mais permet d'avoir un «aide-mémoire» des corrections à apporter.

Les metteurs au point offrent toujours au moins les quatre premières opérations. S'ils sont simples (on dira parfois qu'ils sont *binaires*), ils ne connaissent que l'organisation de la machine physique, et ne prennent pas en compte le langage d'écriture des modules qui constituent le programme. Éventuellement ils sont dotés d'un *désassembleur* qui permet de lister les instructions de la machine sous forme mnémonique. Ils peuvent aussi exploiter une table des symboles dans le programme exécutable, évitant à l'utilisateur de devoir définir des adresses uniquement sous forme numérique. Les metteurs au point symboliques par contre exploitent un minimum d'informations provenant du langage de haut niveau utilisé pour écrire les modules. C'est pourquoi, ils peuvent prendre en compte les structures principales de ces langages pour les quatre premières fonctions et fournir les trois fonctions supplémentaires.

### 6.1.4. Les metteurs au point symboliques multi-fenêtres

Lorsqu'on dispose d'un système multi-fenêtres, permettant de découper l'écran en plusieurs zones distinctes, les metteurs au point symboliques peuvent en tenir compte pour faciliter le travail de l'utilisateur. Trois fenêtres sont alors le plus souvent utilisées:

- fenêtre des variables, où le metteur au point affiche les valeurs des variables dont la consultation ou la trace sont demandées,
- fenêtre de commandes, qui sert à l'échange des commandes de l'utilisateur au metteur au point,
- fenêtre de source, où le metteur au point affiche en permanence les quelques lignes du module source qui entourent le point d'arrêt sur lequel on est.

La manipulation du metteur au point par l'utilisateur en est alors grandement facilitée, puisque certaines commandes sont implicites et qu'il a devant les yeux les informations essentielles pour cette mise au point.

### 6.1.5. Les mesures de comportement dynamique

Certains systèmes offrent la possibilité de faire des mesures sur le comportement dynamique d'un programme à l'exécution. En général, deux types de mesures sont possibles:

- compter le nombre de fois où chaque instruction ou suite d'instructions est exécutée,
- mesurer le temps passé, à l'exécution, dans chaque instruction ou suite d'instructions du programme.

Ces mesures peuvent être effectuées sur la totalité du programme, ou sur certains modules spécifiques. Elles sont obtenues, au moyen d'une option de traduction, qui demande au traducteur d'introduire des instructions aux endroits judicieux du programme. Par exemple, l'incrémentation d'un compteur avant l'exécution d'une instruction en langage évolué permet de compter le nombre de ses exécutions; l'appel à une fonction système spécifique permet d'obtenir les mesures de temps d'exécution. Ces mesures peuvent être plus ou moins précises, suivant les caractéristiques du matériel. Les résultats sont mémorisés dans un fichier, au fur et à mesure, et peuvent être exploités par un outil spécifique qui rattache les mesures aux instructions des modules sources correspondants.

Multics permet de faire ces mesures au niveau de chaque instruction des langages évolués, grâce à une horloge très précise (de l'ordre de la microseconde). Mais on les trouve plus souvent au niveau des appels de sous-programmes, car la durée d'exécution plus importante exige une précision moins grande des mesures de durées.

L'intérêt de ces mesures réside surtout dans l'amélioration ultérieure de l'efficacité du programme par l'utilisateur, qui sait alors exactement où il doit porter son effort. Il est, en effet, inutile de gagner quelques microsecondes sur une instruction qui est exécutée une seule fois, alors que l'on peut être surpris de la durée effective d'une instruction banale, mais qui fait intervenir des conversions coûteuses.

# 6.2. Les préprocesseurs et les macrogénérateurs

Il arrive souvent que l'on ait besoin de disposer d'un module source sous différentes versions assez proches les unes des autres, sans que la fonctionnalité et les spécifications du module soient changées. C'est le cas, par exemple, lorsque le module utilise des constantes d'adaptation qui permettent de configurer les tailles des zones de données internes. C'est le cas également lorsque le module est intégré à une application qui doit pouvoir être portée sur des machines et systèmes différents, entraînant des variantes mineures de ce module. Si on dispose d'autant de copies de ce module source qu'il y a de versions, on rencontre deux problèmes: le premier est le nombre de ces copies presque identiques, et le second est la maintenance de toutes ces copies, lorsqu'une modification doit être apportée à toutes les versions.

Certains langages de programmation, ou leur traducteur, intègrent la prise en compte de ces différentes versions. Cependant une autre solution consiste à disposer d'un seul exemplaire du module source qui est alors paramétré pour permettre la construction d'une version particulière immédiatement avant la traduction. C'est le rôle d'un programme spécifique appellé *préprocesseur*. L'intérêt de reporter au niveau d'un préprocesseur ce travail, au lieu de le laisser au traducteur, est que, d'une part, le préprocesseur n'est pas attaché à un langage particulier, et peut donc servir à beaucoup de langages, d'autre part, cela simplifie le traducteur.

Nous allons nous baser sur le préprocesseur construit par les concepteurs du langage C, pour expliquer succintement le fonctionnement général. L'idée est qu'un préprocesseur prend en entrée un module source, et fournit en sortie un module source transformé. Il s'agit donc bien d'un outil qui travaille au niveau texte source. Il doit être capable de distinguer dans ce texte source les commandes qui lui sont destinées du reste du texte. Dans le cas du préprocesseur de C, on les distingue par la présence du caractère "`#`" en début de ligne. Ces commandes doivent permettre des modifications simples du reste du texte. En voici les principales:

`#include "nom de fichier"` demande au préprocesseur d'inclure à cet endroit le fichier correspondant.

`#define TOTO 2456` demande au préprocesseur de remplacer systématiquement, à partir de maintenant la chaîne de caractères `TOTO` par `2456` partout où elle se trouve.

`#undef TOTO` demande au préprocesseur d'arrêter le remplacement systématique de la chaîne `TOTO`.

`#ifdef TOTO`
`ooo`
demande au préprocesseur de transmettre les lignes qui suivent la commande `#ifdef`, s'il y a une demande de remplacement en cours pour la chaîne `TOTO`, et de les supprimer s'il n'y en a pas. La

```
#else
 ooo
#endif
```

prochaine commande `#else`, si elle existe, entraînera l'action opposée (suppression ou transmission) pour les lignes qui suivent, et la commande `#endif` reprendra le cours de fonctionnement normal. Ces commandes peuvent être imbriquées, avec la signification habituelle des langages de haut niveau.

Il faut remarquer que le langage de définition des commandes ainsi que les opérations de remplacement dans le texte n'impliquent aucune relation avec le langage C lui-même. Ce préprocesseur est d'ailleurs communément utilisé pour le langage Pascal dans les systèmes Unix.

Les remplacements définis ci-dessus sont rudimentaires. En fait ils peuvent être paramétrés. Ainsi la commande

```
#define NUMERO(A, B) 25 * A + B
```

demande le remplacement des chaînes de la forme `NUMERO (u, v)`, où `u` et `v` sont des chaînes quelconques ne contenant pas de virgules, par la chaîne `25 * u + v`. On dit parfois que l'on a une *macrodéfinition*. Il est important de rappeler que le préprocesseur travaille au niveau chaîne de caractères, sans avoir de connaissance du langage représenté. C'est à l'utilisateur de prendre garde que la chaîne obtenue après remplacement soit conforme à ce qu'il désire. Par exemple, dans le contexte de la commande précédente,

| | | |
|---|---|---|
| `NUMERO(x + 1, y)` | deviendra | `25 * x + 1 + y` |
| `NUMERO((x + 1), y)` | deviendra | `25 * (x + 1) + y` |

Il est probable que le premier remplacement n'est pas celui désiré. Il serait en fait préférable d'écrire:

```
#define NUMERO(A, B) 25 * (A) + (B)
```

pour que l'interprétation du remplacement obtenu conserve la structure syntaxique `25 * A + B`, quels que soient `A` et `B`.

Plus généralement on parlera de *macrogénérateur* pour un outil qui a pour but de produire un module source dans un langage de programmation donné, à partir d'un module source qui mélange des commandes dans le langage du macrogénérateur, parfois appellé *méta-langage*, et des parties dans ce langage de programmation. L'une des utilisations les plus courantes est de faciliter l'interfaçage d'un programme d'application avec une base de données. Le programmeur exprimera ses requêtes à la base de données dans le méta-langage, le macrogénérateur traduira ces requêtes en une suite d'instructions du langage de programmation, souvent constituées d'appels à des sous-programmes spécifiques.

## 6.3. Le *make*

Nous avons vu qu'un programme était constitué de plusieurs modules. Mais chacun de ces modules peut lui-même servir à construire plusieurs programmes. Lorsque le nombre de modules impliqués dans un ensemble de programmes est important, il devient difficile de maîtriser complètement les conséquences d'une modification d'un module particulier sur les programmes exécutables. Plus généralement, le problème est d'être certain que les programmes exécutables reflètent bien toutes les modifications qui ont été apportées dans les modules sources. Une solution est évidemment de refaire la traduction de tous les modules, et de refaire ensuite l'édition de liens de tous les programmes. Ceci peut entraîner des traductions inutiles, et des éditions de liens inutiles. Le *make* est un programme qui résout ce problème en ne faisant que ce qui est nécessaire, sous réserve que les informations qu'il exploite soient cohérentes. Initialement écrit pour le système Unix, de nombreuses versions existent maintenant pour des systèmes variés.

### 6.3.1. Graphe de dépendance

Pour déterminer ce qui doit être fait, *make* utilise trois sources d'informations:

- un fichier de description, appellé *makefile*, construit par l'utilisateur,
- les noms et les dates des dernières modifications des fichiers,
- des règles implicites liées aux suffixes des noms de fichiers.

À partir de ces informations, il construit d'abord un *graphe de dépendance* sur l'ensemble des fichiers qui servent à la construction du fichier cible que l'on cherche à obtenir, et qui est fourni en paramètre à *make*. Considérons par exemple un programme exécutable appellé `prog`, obtenu par édition de liens des modules objets situés dans les fichiers `x.o`, `y.o` et `z.o`. On dit que `prog` dépend de ces fichiers. De même, si le module objet `x.o` est obtenu par compilation d'un module source situé dans le fichier `x.c`, avec inclusion du fichier source `defs`, on dit que `x.o` dépend de `x.c` et de `defs`. Remarquons que la compilation de `x.c` demandant l'inclusion de `defs` n'entraîne pas de dépendance entre `x.c` et `defs`, puisque `x.c` n'est pas construit à partir de `defs`. Par contre, il y a bien dépendance de `x.o` sur `defs`, puisqu'une modification de `defs` nécessite de reconstruire `x.o`. La figure 6.1 représente le graphe de dépendance complet.

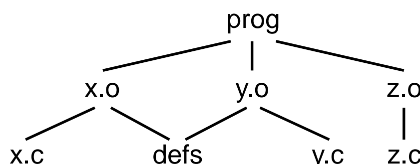



***Fig. 6.1.*** *Graphe de dépendance.*

Les dépendances peuvent être directement décrites dans le fichier *makefile*, et c'est un de ses rôles. On pourrait ainsi mettre dans ce fichier les lignes suivantes:

```
prog : x.o y.o z.o
x.o y.o : defs
```

Certaines dépendances peuvent être aussi obtenues au moyen de règles implicites simples et naturelles. Prenons comme convention (c'est assez habituel, et nous l'avons déjà fait), que les modules objets, résultats d'une traduction, sont dans des fichiers dont le nom est le nom du module suffixé par `.o`; ces fichiers dépendent évidemment du fichier qui contient le module source. Il est naturel, et également courant, de donner au fichier source le nom du module avec un suffixe lié au langage dans lequel il est écrit, comme par exemple `.c` pour le langage C, `.p` pour la langage pascal, ou `.f` pour le Fortran. *Make* peut alors chercher à utiliser les noms des fichiers du répertoire courant pour compléter les règles de dépendances. Ainsi, pour le fichier `x.o`, il recherche dans le répertoire un fichier dont le nom commence par `x` et suffixé par l'un de ceux correspondant à un langage source. Le fait de trouver le fichier `x.c`, lui permet d'établir la dépendance correspondante.

## 6.3.2. Exploitation du graphe de dépendance

Pour déterminer si un fichier est à jour, c'est-à-dire reflète les dernières modifications, il suffit de contrôler que chaque fichier existe et a une date de dernière modification supérieure ou égale à celle des fichiers dont il dépend. Si c'est le cas pour tous les fichiers du graphe, rien n'est à faire, et le fichier cible est à jour. Si ce n'est pas le cas, il est nécessaire de reconstruire ceux qui ne vérifient pas cette propriété. Il est nécessaire de faire cette reconstruction depuis le bas du graphe en remontant vers le haut, puisque la reconstruction d'un fichier entraîne celle de tous ceux qui dépendent de lui et qui sont donc situés au dessus.

Le problème est alors de savoir comment on peut reconstruire un fichier à partir de ceux dont il dépend. Pour cela, il est possible d'attacher à une règle de dépendance dans le fichier *makefile* un ensemble de commandes dont l'exécution sera demandée au système par *make* lorsque la règle ne sera pas respectée. De même, aux règles implicites sont attachées une ou plusieurs commandes qui jouent le même rôle. Pour notre exemple, le fichier *makefile* pourrait associer à la règle relative à `prog` la commande d'appel de l'édition de liens avec l'utilisation de la bibliothèque de nom `s`, et dont le résultat serait mis dans le fichier `prog`:

```
prog : x.o y.o z.o
     ld x.o y.o z.o -ls -o prog
x.o y.o : defs
```

La commande associée à la règle implicite de dépendance du suffixe `.o` sur `.c` pourrait être l'appel du traducteur correspondant sur le fichier source.

### 6.3.3. Macro substitution

Pour faciliter l'écriture du fichier *makefile*, *make* propose un mécanisme simple de *macro substitution*. Une définition de macro est simplement une ligne contenant un nom suivi du signe "=" et du reste de la ligne. Une référence à une macro est constituée du signe "$" suivi du nom de la macro entre parenthèses s'il fait plus d'un caractère. Par ailleurs, avant de lancer des commandes relatives à une règle de dépendance, *make* initialise certaines macros, comme par exemple `$?` qui représente la chaîne des noms de fichiers qui ont été trouvés plus récents que le fichier cible, et qui sont donc la cause de cette exécution.

Dans le contexte du fichier *makefile* de la figure 6.2, l'exécution de

```
make prog
```

entraîne la construction du graphe de dépendance donné en figure 6.1, et les recompilations éventuellement nécessaires ainsi que l'édition de liens. Le nom de l'éditeur de liens est paramétré par `$(LD)`, implicitement identifié à `ld`, pour permettre sa redéfinition lors de l'appel de *make*. De même, la définition des bibliothèques utilisées lors de cette édition de liens est paramétrée explicitement par la définition de la macro `LIBES`.

```
OBJECTS = x.o y.o z.o
FILES = x.c y.c z.c defs
LIBES = -ls
P = imprint -Plplaser
prog : $(OBJECTS)
     $(LD) $(OBJECTS) $(LIBES) -o prog
x.o y.o : defs
print : $(FILES)
     $P $?
     touch print
arch :
     ar uv /usr/src/prog.a $(FILES)
```

***Fig. 6.2**. Exemple de fichier makefile.*

Pour dater certaines opérations, il est possible de créer des fichiers vides, et de demander au système de modifier la date de leur dernière modification lorsque l'opération est exécutée (commande `touch` avec Unix, par exemple). Supposons qu'il existe un tel fichier de nom `print`, dans le contexte du même fichier *makefile*, l'exécution de

```
make print
```

entraînera l'impression des fichiers qui sont plus récents que ce fichier. Les commandes associées à la règle de dépendance non seulement impriment ceux qui sont plus récents, mais force la date du fichier `print` pour mémoriser la date à laquelle cette opération a été faite.

Enfin, après construction du graphe de dépendance, si un fichier n'existe pas, le *make* exécute les commandes associées à la règle de dépendances qui lui correspond. En général, ces commandes entraînent sa construction effective. Cependant, si le fichier n'est pas créé, le *make* considère qu'il a la date actuelle vis-à-vis des règles de dépendances au-dessus de lui. Il n'est donc pas obligatoire qu'un fichier soit effectivement attaché aux opérations. Ainsi, dans le contexte de l'exemple, l'exécution de

```
make arch
```

entraîne toujours l'exécution des commandes associées, puisque le fichier `arch` n'existe pas. Lorsque la commande est exécutée, le fichier n'est toujours pas créé, mais *make* considère qu'il est daté du moment présent, et donc que les propriétés d'antériorité sont respectées.

## 6.4. Les outils complémentaires

Il est courant de trouver sur la plupart des installations, des outils d'aide à la gestion des modules.

Ce peut être, par exemple, un *archiveur* qui permet d'archiver des modules sources, ou plus généralement des fichiers quelconques. Leur intérêt est d'une part de regrouper ensemble des fichiers qui forment un tout cohérent, d'autre part de diminuer l'encombrement sur support

secondaire de cet ensemble, par le biais de techniques complexes de compression d'informations. Le résultat est souvent au moins deux fois moins encombrant.

Pour faciliter le suivi de l'évolution d'un module source, on peut disposer d'un programme, appellé *diff*, qui détermine la différence entre deux fichiers de texte. Le résultat est la suite de commandes d'édition qui permet à un éditeur orienté lignes, de reconstruire le deuxième fichier à partir du premier. On voit que ceci permet à un programmeur de diffuser les modifications à apporter à une version du source pour passer à la version suivante. Celui qui reçoit le résultat du *diff*, n'a pas à faire les modifications à la main, mais à les faire exécuter par l'éditeur orienté lignes correspondant.

La présentation externe d'un module source est souvent très importante pour la mise au point et la maintenance. Un *paragrapheur*, encore appellé *indenteur*, est un programme qui donne une forme standard à un fichier source écrit dans un langage donné. Il est adapté au langage lui-même, car cette forme ne peut être obtenue que par une analyse lexicale et surtout syntaxique du texte source, pour en reconnaître les structures fondamentales.

La gestion des modules objets et des bibliothèques de ces modules est souvent assurée par un *bibliothéquaire*. Nous avons vu qu'une telle biblothèque de modules pouvait nécessiter une structure de données pour rendre plus efficace le travail de l'éditeur de liens.

De nombreux autres outils existent. Nous ne pouvons les décrire tous, certains étant dédiés à des tâches très précises. La raison est que souvent lorsque l'on doit faire ponctuellement une activité automatisable, il est préférable de faire l'outil qui réalisera cette activité plutôt que de faire l'activité soi-même, car l'outil pourra resservir.

# 6.5. Conclusion

☞ La trace d'un programme est une suite d'informations qui permettent de contrôler l'évolution de l'état d'un programme au cours de son exécution. En fait on ne cherche bien souvent à connaître qu'une partie de cette trace.

☞ Un point d'arrêt dans un programme est un endroit où l'on désire arrêter l'exécution du programme pour consulter son état interne. La reprise est la possibilité de poursuivre l'exécution du programme après qu'il ait été interrompu. Le pas à pas est la possibilité d'exécuter le programme une instruction à la fois.

☞ Un metteur au point est un outil qui permet de contrôler interactivement l'exécution d'un programme. Il est symbolique lorsqu'il permet l'expression des commandes en utilisant les structures du langage évolué dans lequel le programme a été écrit.

☞ Les mesures de comportement dynamique d'un programme permettent de savoir où il faut porter son effort pour améliorer son temps d'exécution.

☞ Un préprocesseur ou un macrogénérateur est un programme qui permet de faire un traitement simple sur le texte source avant de le donner à un traducteur.

☞ Il existe un graphe de dépendance entre l'ensemble des fichiers utilisés dans la chaîne de production de programmes: un fichier `A` dépend d'un fichier `B` si `B` est utilisé pour construire `A`.

☞ Un fichier contenant un programme exécutable est à jour si pour tous les fichiers `A` et `B` de son graphe de dépendance, tels que `A` dépende de `B`, la date de dernière modification de `A` est postérieure à celle de `B`.

☞ Le *make* est un outil qui, à l'aide d'un fichier parmètre, construit le graphe de dépendance d'un programme exécutable, contrôle qu'il est à jour et exécute les commandes minimales pour assurer cette mise à jour, si ce n'est pas le cas.

☞ Le *make* peut être utilisé chaque fois qu'il existe un graphe de dépendance entre des fichiers, pour contrôler que ces fichiers sont à jour, et exécuter les commandes nécessaires à cette mise à jour, si ce n'est pas le cas.

☞ Beaucoup d'autres outils existent, car il est souvent préférable de faire un outil pour une activité automatisable, plutôt que de réaliser l'activité à la main.

# TROISIÈME PARTIE

---

# ENVIRONNEMENT EXTERNE

# 7

# La notion de fichier

Nous avons vu dans les premiers chapitres que l'un des rôles du système d'exploitation était de fournir à l'utilisateur une machine virtuelle adaptée à ses besoins. Les connexions avec les périphériques présentant des formes variées, il est nécessaire d'étudier quels sont les vrais besoins des utilisateurs pour réaliser cette adaptation.

Rappelons, tout d'abord, que nous avons distingué quatre sortes de périphériques:

- les périphériques de dialogue homme-machine, pour lesquels le transfert se fait caractère par caractère, ou par écriture directe en mémoire pour les écrans graphiques,
- les périphériques de stockage séquentiel, pour lesquels le transfert se fait par paquet de caractères, de taille variable,
- les périphériques de stockage aléatoire, pour lesquels le transfert se fait par paquet de caractères, de taille fixe,
- les périphériques de communication de machine à machine, pour lesquels le transfert se fait par paquet de caractères, de taille variable mais bornée, un protocole assez complexe étant utilisé pour assurer ces transferts.

## 7.1. Le but des objets externes

Rappelons que lors de l'exécution d'un programme, le processeur ne manipule que des objets en mémoire centrale, dont les caractéristiques générales imposent que de tels objets ne puissent, en général, exister avant le chargement du programme, ou après la fin de son exécution. Leur durée de vie est limitée à la durée de présence du programme en mémoire. On dit que ces objets sont *internes* au programme. Par opposition, on appelle *objet externe* ceux qui ne sont pas internes.

### 7.1.1. Les objets externes comme outils d'échange d'informations

On peut distinguer deux objectifs dans l'utilisation des objets externes par un programme. Le premier est l'échange d'informations, qu'il soit avec l'homme ou avec une autre machine. La conséquence de cet échange est la nécessité de trouver un langage commun entre les deux partenaires de l'échange.

Avec l'homme comme partenaire, cela implique la transformation de la représentation interne des données en une forme compréhensible par l'homme, et réciproquement.

Lors de l'échange entre machines, plusieurs solutions peuvent être adoptées, suivant la nature des machines qui communiquent. Notons tout d'abord que l'échange de base est réalisé sous forme d'une suite d'octets. À l'émission, la représentation interne des données doit donc être convertie en une

suite d'octets, et l'opération inverse effectuée à la réception. Les données déjà sous forme de suite d'octets, comme par exemple les chaînes de caractères, ne posent en général pas de difficulté, du moins si les deux machines utilisent le même codage des caractères (il en existe deux principaux: `EBCDIC` et `ASCII`). Les données dont la représentation interne regroupe plusieurs octets posent plus de difficultés.

- Lors du découpage d'un mot, deux ordres des octets sont possibles. Certains constructeurs mettent d'abord les bits de poids faibles, c'est-à-dire, ceux de droite en premier. D'autres font l'inverse, c'est à dire mettent les bits de poids forts d'abord.
- L'interprétation sémantique de la représentation varie d'un constructeur à l'autre, en particulier, pour les nombres flottants.

Si les représentations des données sont les mêmes pour les deux machines, en particulier si elles sont du même modèle, la communication peut se faire en *binaire*, c'est-à-dire, dans la représentation interne commune. Si ce n'est pas le cas, on peut faire, sur l'une des machines ou sur l'autre, la conversion entre les représentations binaires. Une dernière solution plus courante est d'utiliser une représentation normalisée des données pour assurer l'échange d'informations entre ces machines. L'avantage, dans ce cas, est que le producteur de l'information n'a pas à se préoccuper du destinataire, et réciproquement. L'inconvénient est, bien entendu, la perte de temps machine qui résulte de ces conversions.

Notons que cette conversion de données est considérée comme suffisamment importante pour que certains langages de programmation comme le COBOL permettent de les réduire au minimum. Les données conservent alors dans le programme leur représentation externe. La conversion éventuelle n'intervient alors que lorsque les traitements le nécessitent. Le programmeur doit évidemment avoir conscience du compromis qu'il doit choisir:

- Soit il convertit les données lors des entrées-sorties, les rendant plus coûteuses, mais les traitements sont efficaces car les opérations s'effectuent sur des données en représentation interne.
- Soit il conserve les données en mémoire sous leur forme externe, rendant les entrées-sorties plus efficaces, mais les traitements sur les données doivent les interpréter ou faire à chaque fois les conversions correspondantes.

### 7.1.2. Les objets externes comme outils de mémorisation à long terme

Un programmeur a aussi besoin des objets externes pour disposer d'informations dont la durée de vie s'étend au-delà de la durée de vie du programme. C'est le cas lorsque ces données existent avant le chargement du programme, ou doivent être conservées après la fin de son exécution.

Contrairement au cas précédent, c'est la même machine qui crée ces données et les reprend ultérieurement[5]. On peut dire effectivement qu'il s'agit encore de pouvoir échanger des informations mais cette fois entre des programmes qui s'exécutent sur la même installation. Le problème de conversion énoncé ci-dessus ne se pose plus. Les données, en général, peuvent et doivent être mémorisées dans leur représentation interne.

Un problème peut cependant se poser pour les données de type *pointeur*. Rappelons qu'un pointeur est une information binaire qui permet de désigner un emplacement dans un espace mémoire; c'est donc une adresse. Si cette mémoire est la mémoire centrale, cela signifie que le pointeur désigne un objet interne au programme, et n'a de signification que pour cette exécution. Le conserver sous cette forme après la fin de l'exécution de ce programme n'a pas sens. Sa valeur doit être remplacée par une information qui désigne l'emplacement dans l'objet externe où sera rangé la copie de la donnée interne pointée.

## 7.2. La manipulation des objets externes par le programme

Le mot fichier est parfois utilisé de façon ambiguë, car il désigne tantôt l'objet externe mémorisé sur un support magnétique (bande ou disque), tantôt l'entité manipulée par le programme. Pour éviter

[5] au moins dans le cas des systèmes centralisés. Dans un réseau local, avec un serveur de fichier, ce peut être une autre machine.

cette ambiguïté, le premier pourrait être appelé fichier physique et le second fichier logique. En général nous n'utiliserons le terme fichier que pour désigner la façon dont le programme manipule l'objet indépendamment de l'objet lui-même et des contraintes physiques d'implantation comme des caractéristiques du périphérique qui le supporte.

## 7.2.1. La notion de fichier

Le *fichier* est la façon dont un programme voit un objet externe. C'est d'abord une collection d'enregistrements logiques éventuellement structurée, sur laquelle le programme peut exécuter un ensemble d'opérations. Un enregistrement logique, encore appelé *article*, est l'ensemble minimum de données qui peut être manipulé par une seule opération élémentaire du fichier. Il est souvent constitué de diverses données élémentaires, que l'on peut décrire dans le langage de programmation utilisé, comme le montre la figure 7.1. Ces données élémentaires ne sont accessibles individuellement par le programme que lorsque l'enregistrement logique est recopié dans un objet interne au programme. Dans cet exemple, le fichier est constitué d'une collection d'enregistrements qui ont la structure d'EMPLOYE. Les opérations sur le fichier permettent de transférer le contenu d'un tel enregistrement entre la mémoire interne du programme et l'objet externe qui lui est associé.

```
01 EMPLOYE
    02 NOM PICTURE X(30)
    02 PRENOM PICTURE X(20)
    02 NUM-S-S
        03 SEXE PICTURE 9
        03 DATE-NAISSANCE
            04 ANNEE PICTURE 99
            04 MOIS PICTURE 99
        03 LIEU-NAISSANCE
            04 DEPARTEMENT PICTURE 99
            04 CANTON PICTURE 999
        03 ORDRE PICTURE 999
    02 CODE-INTERNE PICTURE 9999
```

***Fig. 7.1****. Exemple de description en COBOL d'un enregistrement logique.*

## 7.2.2. Le fichier séquentiel

La caractérisation des fichiers est déterminée par la nature des opérations que l'on peut effectuer sur le fichier. Le plus simple est le fichier séquentiel. Les opérations se résument, essentiellement, en la lecture de l'enregistrement suivant ou l'écriture d'un nouvel enregistrement en fin du fichier séquentiel. Bien souvent, pour affirmer le caractère séquentiel du fichier, il n'est pas possible de mélanger les lectures et les écritures. Le fichier est alors un fichier séquentiel en lecture ou un fichier séquentiel en écriture.

Le fichier séquentiel en lecture est initialement positionné sur le premier enregistrement logique. Chaque opération de lecture (*lire* ou *read*) transfère dans une zone interne au programme un enregistrement du fichier, et prépare le fichier pour le positionner sur l'enregistrement suivant. On dispose souvent d'une opération complémentaire qui permet de savoir s'il y a encore des enregistrements à lire dans le fichier (*fin_de_fichier* ou *end_of_file*), et éventuellement une opération permettant le retour au début du fichier (*rembobiner* ou *rewind*).

Le fichier séquentiel en écriture peut être initialement vide, ou positionné après le dernier enregistrement déjà dans le fichier. Chaque opération d'écriture (*ecrire* ou *write*) rajoute un enregistrement dans le fichier depuis une zone interne au programme, et positionne le fichier après cet enregistrement. L'écriture d'un enregistrement se fait donc toujours après ceux qui sont déjà dans le fichier.

Noter que la caractéristique essentielle d'un fichier séquentiel est que les enregistrements sont lus dans l'ordre où ils ont été écrits. Aucune conversion des données n'est appliquée, soit parce qu'elle est faite par le programme, soit parce qu'elle n'est pas nécessaire.

### 7.2.3. Le fichier séquentiel de texte

Le fichier séquentiel de texte est analogue à un fichier séquentiel, mais avec conversion du format des données entre leur représentation interne et une représentation accessible à l'être humain. L'expression des opérations nécessitant la définition d'un format de conversion, la forme est très dépendante du langage de programmation. On peut assimiler un fichier séquentiel de texte à un fichier séquentiel où les enregistrements logiques sont en fait des caractères individuels.

En lecture, les opérations sur un tel fichier sont complétées par des opérations qui permettent de lire une donnée élémentaire, comme un entier, un nombre flottant, etc..., par conversion d'une suite de caractères lus depuis le fichier, suivant un certain format, implicite ou explicite.

En écriture, les opérations sur un tel fichier sont complétées par des opérations qui permettent d'écrire une donnée élémentaire, comme un entier, un nombre flottant, etc..., par conversion en une suite de caractères écrits dans le fichier, suivant un certain format, implicite ou explicite.

Certains langages de programmation, comme le FORTRAN, présentent le fichier séquentiel comme une collection d'enregistrements logiques correspondant à une ligne de texte. En lecture, on peut alors lire l'enregistrement complet, par le biais d'un format, et d'un ensemble de variables élémentaires. Le format est alors utilisé pour découper l'enregistrement en chaînes de caractères et les convertir dans la représentation interne de ces variables. De même, en écriture, on peut écrire un enregistrement complet, par le biais d'un format et d'un ensemble de valeurs élémentaires. Le format permet de convertir ces valeurs en leur représentation externe et de les placer dans l'enregistrement.

### 7.2.4. Le fichier à accès aléatoire

Le but du fichier aléatoire est de permettre l'accès à un enregistrement quelconque de la collection sans avoir à parcourir tous ceux qui ont été écrits avant lui. En général, il n'y a pas de restriction du type lecture seule ou écriture seule, comme dans les fichiers séquentiels, mais au contraire définition de trois opérations principales: lecture, écriture et mise à jour d'un enregistrement après sa lecture.

Pour fournir l'accès à l'un quelconque des enregistrements, il faut que le programmeur puisse le désigner. Cette désignation peut se faire par un *numéro*. Dans ce cas, le fichier est vu comme une collection d'enregistrements numérotés. Pour permettre d'implanter efficacement l'accès à un enregistrement de numéro donné, la taille d'un enregistrement, c'est-à-dire, le nombre d'octets de sa représentation, est en général fixe. Pour le programmeur, le fichier à accès aléatoire par numéro peut s'assimiler à un tableau à une dimension. C'est évidemment lui qui fixe l'attribution des numéros aux enregistrements lors des écritures.

La désignation d'un enregistrement peut se faire par l'intermédiaire d'une clé qui est en général incluse dans l'enregistrement. Lors d'une lecture, le programmeur peut, par exemple, donner d'abord une valeur aux champs correspondant à la clé dans la zone mémoire interne de l'enregistrement, et en demander la lecture. L'opération du fichier recherchera l'enregistrement correspondant et en rangera la valeur dans la zone. L'opération de *mise à jour* ou *update*, ultérieure réécrira cet enregistrement sans avoir besoin de le rechercher de nouveau. Une opération d'écriture, avec en paramètre le contenu de l'enregistrement dans une zone en mémoire interne, consistera à placer l'enregistrement dans l'objet externe, et à mettre à jour la structure de données qui permet de le retrouver (fichier séquentiel indexé ou B-arbre).

On trouve parfois un troisième type de fichier à accès aléatoire construit sur un fichier séquentiel, mais qui ne doit pas être confondu avec ce dernier. On dispose dans ce cas des deux opérations de lecture séquentielle et d'écriture séquentielle (au bout), ainsi que de deux opérations supplémentaires permettant l'accès aléatoire aux enregistrements du fichier:

- *noter* est une fonction qui retourne une valeur binaire représentant la position de l'enregistrement courant dans le fichier. Elle n'a de sens que pour le fichier.
- *positionner* avec en paramètre une valeur binaire précédemment obtenue par l'opération *noter* ci-dessus, repositionne le fichier sur l'enregistrement sur lequel il était au moment de cette opération.

L'utilisation de ces opérations permet au programmeur de construire des structures de données quelconques sur un fichier. La valeur retournée par *noter* peut s'assimiler à un pointeur externe et n'a de sens que sur le fichier dont il provient.

# 7.3. La liaison entre le fichier et l'objet externe

## 7.3.1. L'établissement de la liaison

Un fichier (logique) peut être vu comme un objet interne au programme. Pour pouvoir effectuer des opérations sur le fichier, il faut qu'il soit *relié* à un objet externe. Il n'est pas intéressant que cette liaison soit prise en compte par l'éditeur de liens. En effet cela figerait cette liaison au moment de cette édition de liens, le programme exécutable ne pouvant alors s'exécuter que sur des objets externes définis à ce moment. Par ailleurs l'éditeur de liens se préoccupe essentiellement des liaisons internes au programme, alors qu'il s'agit de liaisons externes. Pour les mêmes raisons, il n'est pas judicieux que la liaison soit fixée au moment du chargement du programme. Au contraire elle peut être établie et rompue dynamiquement au cours de son exécution.

Nous reviendrons ultérieurement sur la définition de cette liaison, qui doit permettre d'associer un fichier interne au programme à un objet externe. En supposant que cette définition existe, deux opérations sur un fichier quelconque permettent d'exploiter cette définition:

- L'opération d'*ouverture* (on dit aussi *open*) sur un fichier permet d'établir la liaison avec un objet externe suivant les informations contenues dans sa définition.
- L'opération de *fermeture* (on dit aussi *close*) sur un fichier ouvert, permet de rompre temporairement ou définitivement la liaison avec l'objet externe.

Les opérations que nous avons indiquées dans les paragraphes précédents ne peuvent être appliquées que sur des fichiers ouverts, puisqu'elles ont pour conséquence des accès à un objet externe qui doit donc avoir été associé au fichier.

Il y a toutefois une exception à cette règle d'établissement dynamique de la liaison. La plupart des systèmes établissent trois liaisons spécifiques préalablement à l'exécution du programme, pour les trois fichiers standards:

- `SYSIN` pour l'entrée des données provenant de l'utilisateur,
- `SYSOUT` pour la sortie des données vers l'utilisateur,
- `SYSERR` pour la sortie des messages d'erreurs vers l'utilisateur.

## 7.3.2. Représentation interne d'un fichier

La représentation interne d'un fichier logique dans le programme est une structure de données (appelée parfois *bloc de contrôle de données* ou encore *Data Control Bloc*). Elle contient quatre types d'informations:

- Les *attributs de la liaison* permettent de savoir quel est l'état de la liaison (ouverte ou fermée), les variables internes de gestion de la liaison ainsi que les opérations autorisées sur le fichier.
- Le *descripteur de l'objet externe* permet de localiser l'objet externe. S'il s'agit d'un périphérique, il faut connaître sa nature et son *adresse*. S'il s'agit d'un fichier physique, il faut connaître en plus la localisation de l'objet externe sur le périphérique qui le supporte.
- Les *procédures d'accès* sont en fait la concrétisation des opérations du fichier sur l'objet externe qui lui est relié. Il faut noter que, vu du programmeur, il s'agit toujours d'un ensemble d'opérations bien définies, mais l'implantation proprement dite de ces opérations dépend de la nature de l'objet externe et de son support.
- Les *tampons d'entrées-sorties* permettent d'assurer l'interface entre les enregistrements logiques et les enregistrements physiques. Nous avons dit que les opérations du fichier manipulaient des enregistrements logiques tels que les définit le programmeur. Par contre les entrées-sorties physiques ont des contraintes de taille et d'organisation qui dépendent du support. Par exemple, sur un disque, les enregistrements physiques occupent un nombre entier de secteurs. De même, sur une bande, les enregistrements physiques doivent avoir une taille minimum si on veut avoir une bonne occupation de l'espace. Les tampons permettent de regrouper plusieurs

enregistrements logiques par enregistrement physique lors des écritures, et de les dégrouper lors de la lecture. Ils servent aussi à améliorer les performances du transfert entre le support externe et la mémoire centrale.

Certaines des informations énoncées ci-dessus sont statiques, c'est-à-dire, connues à la compilation. Ce sont celles qui sont directement liées à la nature du fichier tel que l'utilise le programmeur. La plupart sont en fait dynamiques, car elles dépendent du fichier et de l'objet externe avec lequel il est relié. La figure 7.2 montre une telle représentation d'un fichier logique lorsqu'il est ouvert ou fermé.

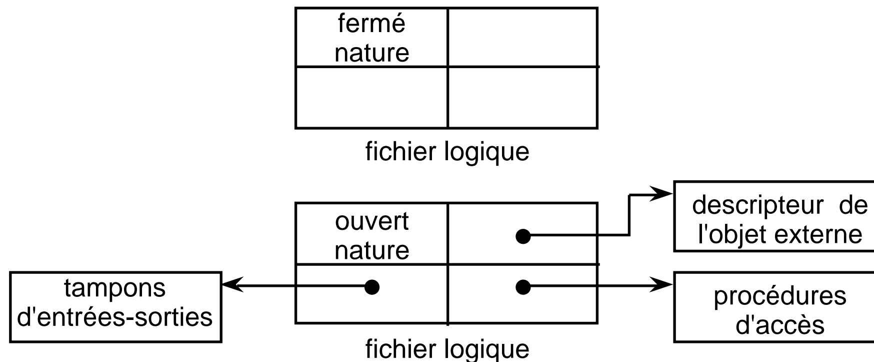



***Fig. 7.2**. Représentation d'un fichier ouvert ou fermé.*

# 7.4. Conclusion

☞ Les objets externes peuvent être utilisés pour échanger des informations. Cet échange peut nécessiter de convertir les données dans une représentation commune.

☞ Les objets externes peuvent être utilisés pour mémoriser des données à long terme. Aucune conversion n'est nécessaire sauf pour les pointeurs vers la mémoire centrale qui perdent toute signification.

☞ Le fichier (logique) est le point de vue qu'a un programme d'un objet externe. On distingue en général le fichier séquentiel d'enregistrements, le fichier séquentiel de texte, et le fichier à accès aléatoire.

☞ Un fichier doit être relié à un objet externe pour pouvoir être utilisé. La liaison est établie par l'ouverture, et supprimée par la fermeture.

☞ En mémoire centrale, un fichier logique est représenté par une structure de données (DCB) qui contient les attributs de la liaison, le descripteur de l'objet externe, les procédures d'accès et les tampons d'entrées-sorties.

8

# Implantation des objets externes sur disque

Avant d'étudier les différentes caractéristiques d'une liaison entre un fichier interne et un objet externe, nous allons présenter quelques méthodes actuelles de représentation des objets externes. Deux problèmes se posent: d'une part comment allouer de l'espace disque à un objet externe, d'autre part comment représenter l'espace alloué à l'objet externe. En fait ces deux aspects sont fortement liés. Par ailleurs, l'indépendance des objets externes implique de pouvoir donner une numérotation propre aux secteurs d'un objet. Nous appèlerons *numéro logique* le numéro relatif du secteur dans un objet externe. La représentation de l'espace alloué à l'objet doit permettre la transformation entre les numéros logiques et les adresses disques. Nous allons tout d'abord linéariser l'espace disque pour pouvoir nous ramener à des problèmes de gestion dans un espace à une dimension.

## 8.1. La linéarisation de l'espace disque

Nous avons dit qu'un disque est constitué d'un certain nombre de plateaux revêtus d'une couche magnétique. Des têtes magnétiques, à raison de une par face, sont fixées sur un bras qui permet de se déplacer perpendiculairement à l'axe. Pour une position de bras donnée, chaque tête délimite une circonférence sur son plateau, appelé piste. Chaque piste est découpée en un nombre fixe de secteurs de taille fixe. Nous avons donc un espace à trois dimensions. L'identification d'un secteur élémentaire est obtenue par trois nombres:

- Le numéro de cylindre $nc$ (on dit aussi le numéro de piste) détermine la position du bras. Il est compris entre 0 et le nombre de cylindres $nbc$ moins 1.
- Le numéro de face $nf$ détermine la tête devant faire le transfert. Il est compris entre 0 et le nombre de faces $nbf$ moins 1.
- Le numéro de secteur $ns$ détermine le secteur concerné sur la piste. Il est compris entre 0 et le nombre de secteurs par piste $nbs$ moins 1.

Il est habituel de transformer cet espace à trois dimensions en un espace à une dimension, plus facile à gérer. En appelant numéro virtuel de secteur $nv$, la désignation d'un secteur dans ce nouvel espace, on peut définir:

$$nv = ns + nbs * ( nf + nbf * nc )$$

Il s'ensuit que $nv$ est compris entre 0 et $nbs * nbf * nbc - 1 = N - 1$, en notant $N$ le nombre total de secteurs du disque. Cette transformation est bijective, et il est possible de retrouver $ns$, $nf$ et $nc$ à partir de $nv$ :

$ns = nv$ **mod** $nbs$

$nf = ( nv$ **div** $nbs )$ **mod** $nbf$

$nc = ( nv$ **div** $nbs )$ **div** $nbf$

où **mod** représente l'opération modulo, et **div** la division entière. L'avantage de cette numérotation par rapport aux autres que l'on pourrait faire, est que des numéros de secteurs virtuels voisins impliquent des cylindres voisins; c'est donc une numérotation qui minimise les mouvements de bras. A noter que cette transformation est effectuée automatiquement par les contrôleurs de disque SCSI, qui gèrent également les secteurs erronés. Dans ce cas, le disque est vu directement comme une suite de secteurs numérotés.

# 8.2. Allocation par zone

## 8.2.1. Implantation séquentielle simple

La première idée pour implanter un objet externe sur disque, est de lui attribuer un ensemble de secteurs contigus dans l'espace à une dimension de la numérotation virtuelle. S'il est de taille T, et commence au secteur virtuel de numéro D, il occupe alors les secteurs dont les numéros virtuels sont tels que:

$$D \quad nv \quad D + T - 1$$

L'information de localisation de l'objet externe est donc constituée du couple <D,T> (figure 8.1). L'accès au secteur de numéro logique *i* dans l'objet externe, se fait de la façon suivante:

$$0 \quad i \quad T - 1 \text{ alors } i \quad nv = D + i \quad <ns, nf, nc>$$

Le premier inconvénient de cette méthode est lié à la difficulté d'agrandir la taille de l'objet externe ultérieurement. En effet, ceci n'est possible que si un ensemble suffisant de secteurs virtuels, commençant en D + T, sont libres. Si un autre objet externe commence précisément en D + T, il n'est pas possible d'agrandir la taille de celui qui commence en D.

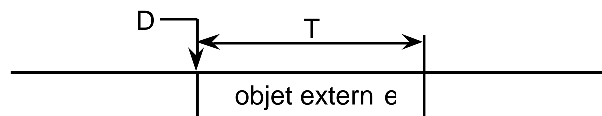



***Fig. 8.1.*** *Implantation séquentielle d'un objet externe localisé par <D,T>.*

Le deuxième inconvénient est la conséquence du premier: il faut connaître la taille exacte de l'objet externe au moment de sa création, et avant toute autre opération. Ceci conduit souvent à prévoir une taille supérieure à ce qui est en fait nécessaire, de façon à ne pas se trouver dans l'impossibilité de terminer un traitement parce que l'espace attribué à l'objet externe est plein.

Le troisième inconvénient est lié à la méthode d'allocation qui doit être utilisée: elle est appelée *allocation par zone*. La création d'un objet externe de taille T, nécessite d'allouer un espace de T secteurs consécutifs dans l'espace unidimensionnel des numéros virtuels. Évidemment cet espace doit consister en des secteurs libres, c'est-à-dire, qui ne sont pas occupés par un autre objet externe. Il est possible que, par suite d'allocations et de restitutions successives, l'espace se présente comme une suite de zones libres et de zones occupées, aucune des zones libres ne pouvant satisfaire la demande de T secteurs, alors que la somme de toutes ces zones dépasse cette taille T. Il faut alors déplacer les différents objets externes, c'est-à-dire, tasser les objets, pour regrouper ensemble les zones libres dans un seul espace contigu. La difficulté est que le déplacement d'objets sur disque est une opération coûteuse en temps, puis qu'il faut lire son contenu en mémoire centrale et le réécrire.

La conséquence de ces trois inconvénients est en général une très mauvaise utilisation de l'espace disque, au bénéfice du constructeur qui vend donc plus de disques que ce qui est nécessaire. C'est pourquoi cette méthode tend à être abandonnée.

## 8.2.2. Implantation séquentielle avec extensions fixes

Pour éviter les inconvénients de la méthode précédente, on peut imaginer que l'espace occupé par un objet externe n'est pas un unique espace contigu, mais un ensemble d'espaces contigus. L'objet externe peut être rallongé à tout moment, si le besoin s'en fait sentir, en rajoutant un nouvel espace à l'ensemble. Pour éviter d'avoir à fournir, et à mémoriser, un paramètre de taille lors des rallongements successifs, on peut fixer une taille commune à tous les blocs, sauf peut-être le premier. Ainsi lors de la création de l'objet externe, on fixe les paramètres <P, I> qui déterminent la

taille de la première zone allouée (P), encore appelée partie *primaire* et la taille des zones en rallongement (I), encore appelée *incréments* ou *extensions*. La plupart des systèmes qui appliquent cette méthode imposent une limite au nombre d'incréments que peut avoir un objet externe, de façon à donner une représentation fixe à l'ensemble des informations qui localisent l'objet externe sur son support.

La figure 8.2 donne un exemple d'implantation séquentielle avec extensions. Les informations de localisation nécessitent de conserver les paramètres <P, I> de taille des zones primaires et extensions, mais aussi évidemment la table DEBUTS des numéros virtuels des premiers secteurs de chacune des zones allouées, ainsi que le nombre de zones allouées Q. La taille effective de l'objet externe à un instant donné est:

$$T = P + (Q - 1) * I.$$

Si le tableau DEBUTS contient au plus M entrées, la taille de l'objet externe est telle que:

$$P \quad T \quad P + (M - 1) * I$$

L'accès au secteur de numéro logique *i* dans l'objet externe, est défini comme suit:

0   *i*   P - 1, alors   *nv* = DEBUTS [0] + *i*

P   *i*   T - 1, alors   *nv* = DEBUTS [ (*i* - P) **div** I + 1 ] + (*i* - P) **mod** I

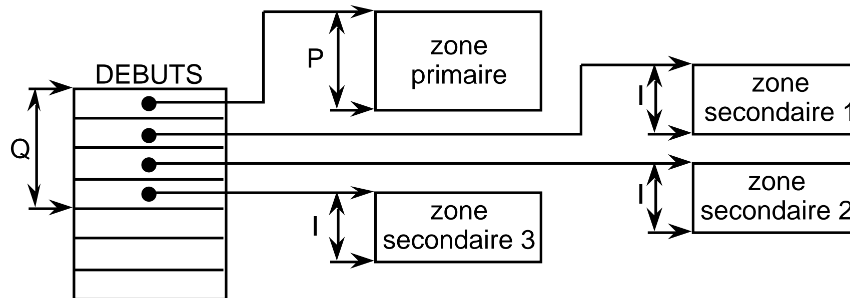



***Fig. 8.2.*** *Implantation séquentielle, avec extensions, d'un objet externe localisé par <P, I, Q, DEBUTS>.*

L'inconvénient de cette méthode est évidemment lié à l'allocation par zone, comme pour la méthode précédente. Pour créer l'objet externe, il faut disposer d'une zone contiguë de taille P, et pour le rallonger, il faut disposer de zones contiguës de taille I. Notons que ces valeurs peuvent être notablement inférieures à la taille totale, et donc le problème est ici moins important.

Par contre, il est possible d'agrandir la taille de l'espace attribué à l'objet externe, du moins dans certaines limites, et la taille totale maximale de l'objet ne doit pas être connue lors de sa création. Il faut néanmoins avoir une idée de cette taille pour définir les valeurs P et I. En général, P est toujours pris supérieur ou égal à I. Il est en effet inutile de tenter de créer l'objet avec un zone primaire plus petite que les zones secondaires simplement pour avoir plus de chance de réussir à l'allouer, puisque le problème se produira, de toute façon, lors de l'allocation de la première extension. Par ailleurs, l'objet externe n'occupe pas toujours la totalité de l'espace qui lui est alloué. Si sa taille effective est, à un instant donné, R, l'espace inoccupé sera P - R si R   P, et au pire I - 1 si R > P. On peut donc écrire:

$$T - R \quad \max(P - R, I - 1)$$

S'il existe une taille minimum pour l'objet, disons $R_{min}$, on obtient:

$$T - R \quad \max(P - R_{min}, I - 1)$$

Par ailleurs, appelons $R_{max}$ la taille maximale de l'objet. On a donc:

$$R_{max} = P + (M - 1) * I$$

Il s'ensuit que pour $R_{max}$ donné, une diminution de P entraîne une augmentation de I. Pour avoir le moins de perte, il faut donc prendre:

$$P - R_{min} = I - 1, \text{ donc } P = R_{min} + I - 1$$

En reportant dans l'équation précédente, on obtient:

$$R_{max} = R_{min} + I - 1 + (M - 1) * I$$

D'où on tire:

$$I = (R_{max} - R_{min} + 1) / M$$
$$P = R_{max} - (M - 1) * I$$

En fait ceci peut conduire à une valeur de P trop grande, et telle que l'allocation ne soit pas possible. Il est possible de diminuer cette valeur, mais, comme nous l'avons déjà dit, la condition P    I implique que P    $R_{max}$ / M. Si l'allocation de la zone primaire n'est pas possible dans ce cas, il est nécessaire de réorganiser le disque pour tasser les objets qui l'occupent.

La plupart des gros systèmes d'IBM utilisent ce type d'implantation. Certains offrent la possibilité d'avoir des extensions de tailles différentes, mais le nombre d'extensions d'un fichier est limité.

### 8.2.3. Implantation séquentielle avec extensions quelconque

Pour éviter les inconvénients de la méthode précédente, on peut imaginer que les différentes zones soient de taille quelconque, et qu'elles soient en nombre quelconque: <<$D_1$, $T_1$>, <$D_2$, $T_2$>, ..., <$D_n$, $T_n$>>. L'objet externe peut être rallongé à tout moment, si le besoin s'en fait sentir, soit en essayant de prolonger la dernière zone, s'il y a de l'espace libre derrière elle, soit en rajoutant un nouvel espace <$D_{n+1}$, $T_{n+1}$> à l'ensemble.

L'accès au secteur de numéro logique *i* dans l'objet externe, est défini comme suit:

$$i \quad nv = D_j + i - S_j$$

$$\text{où } S_j = \sum_{k=1}^{j-1} T_k \text{ et j tel que } S_j \quad i < S_{j+1}$$

Dans le cas d'un parcours séquentiel de l'objet externe, le système passera simplement d'une zone à la suivante, sans avoir à rechercher la zone à partir du numéro logique. Par contre, dans le cas d'un accès aléatoire, une boucle est nécessaire pour déterminer le numéro de zone concernée.

Notons que lorsque la demande d'allocation est implicite, comme par exemple lors de l'écriture au delà de la fin du fichier logique, le système déterminera souvent lui-même la taille nécessaire. Dans ce cas, il essayera souvent de prolonger d'abord la dernière zone, plutôt que d'en rajouter une nouvelle. Il est également possible de morceler la demande s'il n'y a pas de zone libre de taille suffisante. Cependant cela conduit à une fragmentation de l'espace, et une certaine perte d'efficacité.

Les systèmes de gestion de fichier de MacOS, NTFS (Windows NT) et VMS utilisent cette méthode.

## 8.3. Allocation par blocs de taille fixe

Une autre méthode d'implantation des objets externes est de leur attribuer une collection de zones de taille fixe, qui est la même quel que soit l'objet. On parle alors plutôt d'une *allocation par blocs de taille fixe*, chaque bloc étant constitué évidemment d'un nombre entier de secteurs. Les blocs sont appelés parfois des *clusters.* Leur numérotation découle directement du numéro du premier secteur virtuel qu'il contient. En effet, si on note *nbsb* le nombre de secteurs par blocs, le numéro virtuel du premier secteur du bloc de numéro *j* est $nv_j = nbsb * j$ (il faut parfois ajouter une constante).

### 8.3.1. Implantation par blocs chaînés

Le système de gestion de fichier FAT, initialement créé pour MS-DOS, utilise une telle méthode, en chaînant entre eux les blocs d'un même objet externe. Au lieu de réserver dans les blocs eux-mêmes la place pour mettre le numéro du bloc suivant de l'objet externe, ce système regroupe dans une table unique l'ensemble des chaînons de blocs pour tous les objets externes. Cette table est appelée la *File Allocation Table* (*FAT*), d'où le nom donné au système. L'entrée *j* de la table contient donc le

numéro du bloc suivant le bloc *j* dans l'objet externe à qui le bloc *j* est alloué. L'espace alloué à un objet externe peut alors être simplement repéré par le numéro de son premier bloc (figure 8.3).

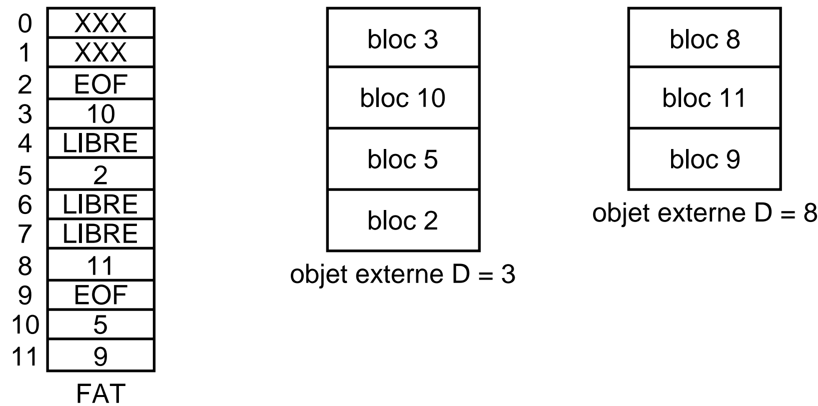



***Fig. 8.3.*** *Implantation par blocs chaînés des objets externes.*

La FAT est en général conservée en mémoire centrale de façon à ne pas pénaliser les accès. Pour obtenir le numéro virtuel *nv* du secteur logique *i* d'un objet externe, il faut dérouler l'algorithme suivant:

```
r := i div nbsb;          { rang du bloc dans l'objet }
j := D;                   { premier bloc de l'objet }
tant que r > 0 faire
     j := FAT [j];
     r := r - 1;
fait;
nv := j * nbsb + i mod nbsb;
```

Ceci montre que cette représentation est bien adaptée aux objets externes liés à des flots séquentiels. Les flots à accès aléatoire sont pénalisés par la durée de cet algorithme qui est proportionnelle au rang du bloc recherché. Si la FAT est entièrement en mémoire, les accès peuvent être de l'ordre de quelques microsecondes. Par exemple, avec 10 µs par pas, et un rang de l'ordre de 100, cela demande néanmoins 1 ms.

Cette méthode a été conçue initialement pour des disquettes, où le nombre de blocs est faible. Les numéros de blocs dans la FAT occupent 2 octets, mais sont sur 16 bits. Il ne peut donc y avoir que 65535 blocs au plus. Cela donne une FAT de 128 Ko, qui commence à occuper beaucoup de place, en particulier, lorsque la mémoire centrale est limitée à 640 Ko, comme dans les versions de MSDOS antérieure à 1991. Si on ne peut la mettre complètement en mémoire, cela rallonge d'autant la fonction de passage au bloc suivant. Notons que, avec une taille de blocs de 512 octets, cela permettait de gérer des disques de 32 Mo, ce qui était amplement suffisant à cette époque.

### 8.3.2. Implantation par blocs à plusieurs niveaux

Les systèmes UNIX et Linux utilisent également une allocation par blocs de taille fixe. Les problèmes de dimensionnement évoqués ci-dessus sont dûs au fait que les informations qui définissent l'espace alloué à un objet externe sont centralisées dans une seule table. On peut donc les éviter en les répartissant entre plusieurs tables. Il est alors logique d'attribuer une table par objet externe. La taille de cette table ne doit cependant pas être figée, car son encombrement serait prohibitif pour les objets de petite taille.

Unix répartit cette table sur plusieurs niveaux en fonction de la taille des objets eux-mêmes. La localisation d'un objet est définie par (figure 8.4):

- une table, que nous appelons TABDIRECT, donnant les numéros de blocs pour les 10 premiers blocs de l'objet externe,
- un numéro de bloc, que nous appelons INDIRECT_1, qui contiendra les numéros des p blocs suivants de l'objet externe,
- un numéro de bloc, que nous appelons INDIRECT_2, qui contiendra les p numéros de blocs contenant les numéros des $p^2$ blocs suivants de l'objet externe,

- un numéro de bloc, que nous appelons INDIRECT_3, qui contiendra les p numéros de blocs contenant, au total, $p^2$ numéros de blocs contenant les numéros des $p^3$ blocs suivants de l'objet externe.

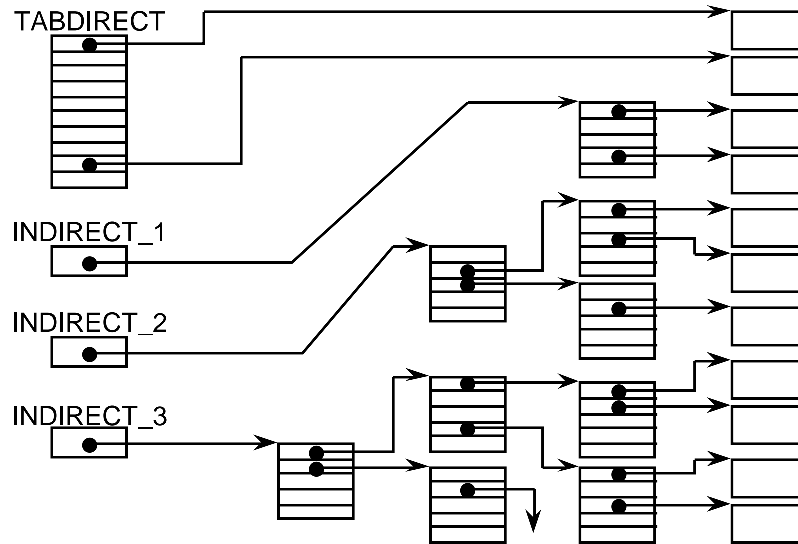



***Fig. 8.4.*** *Implantation par blocs de taille fixe sur plusieurs niveaux.*

Le numéro de secteur virtuel *nv* du secteur logique *i* de l'objet externe est obtenu par l'algorithme donné en figure 8.5. Un objet externe qui occupe moins de 10 blocs a des numéros de blocs indirects nuls (pas de bloc alloué), tous les accès disques seront immédiats. Si un bloc fait 1 Ko, cela signifie que cette méthode n'entraîne aucune pénalisation pour les objets externes de 10 Ko ou moins. Or cela représente en moyenne plus de 90% des fichiers d'une installation Unix. Si un objet externe occupe entre 10 et 10 + p blocs, il peut y avoir 1 accès disque supplémentaire pour chaque accès à un bloc de données. Si les numéros de blocs sont sur 4 octets, il s'ensuit que p = 256. Les objets externes qui font moins de 266 Ko entrent donc dans cette catégorie. De même il y a deux accès disque supplémentaires pour les objets externes qui ont entre 266 Ko et 64 Mo. Enfin, nous aurons 3 accès disques supplémentaires pour ceux qui ont entre 64 Mo et 16 Go. Ceci ne veut pas dire que la taille du disque soit elle-même limitée à 16 Go, puisque les numéros de blocs étant sur 32 bits, il peut y en avoir 4 milliards (ou 2 si ce sont des entiers signés) et le disque peut atteindre 4 To. Notons que la représentation de la longueur utile d'un fichier, si elle est sur 32 bits, limite de fait la taille des fichiers à 4 Go.

Pour éviter de relire un bloc sur le disque si on l'a déjà lu peu de temps auparavant, Unix implante une technique de cache des blocs disques, Cela veut dire que le décompte des accès supplémentaires est un maximum, qui peut en fait être beaucoup plus faible si les blocs contenant des numéros de bloc n'ont pas besoin d'être relus. Il en est souvent ainsi lorsque l'objet externe est relié à un fichier séquentiel.

Une telle structuration peut être rapprochée de celle vue en 8.2.3, où nous avions un nombre quelconque de zones de taille variable. Lorsque ces zones se morcellent, et deviennent de plus en plus petites, dans le cas d'un disque fragmenté, il y a augmentation de la taille de la représentation de l'espace alloué à l'objet externe et perte d'efficacité, en particulier lors des accès aléatoires. Dans la structuration abordée ici, au contraire, l'efficacité n'est pas influencée par le contexte et est prise en compte par le système dès sa conception.

```
r := i div nbsb;
si r < 10 alors
     j := TABDIRECT [r];
sinon
     r := r - 10;
     si r < p alors
          lire_bloc ( INDIRECT_1, TAB_LOCALE );
          j := TAB_LOCALE [r];
     sinon
          r := r - p;
          si r < p * p   alors
               lire_bloc ( INDIRECT_2, TAB_LOCALE );
               lire_bloc ( TAB_LOCALE [r div p], TAB_LOCALE );
               j := TAB_LOCALE [r mod p];
          sinon             { r supposé < p * p + p * p * p }
               r := r - p * p;
               lire_bloc ( INDIRECT_3, TAB_LOCALE );
               lire_bloc ( TAB_LOCALE [r div (p * p)], TAB_LOCALE );
               r := r mod (p * p);
               lire_bloc ( TAB_LOCALE [r div p], TAB_LOCALE );
               j := TAB_LOCALE [r mod p];
          finsi;
     finsi;
finsi;
nv := j * nbsb + i mod nbsb;
```

***Fig. 8.5.*** *Algorithme de calcul de nv du secteur logique i d'un objet externe.*

# 8.4. Représentation de l'espace libre

## 8.4.1. Notion de quantum

Les opérations élémentaires sur les disques doivent se faire par un nombre entier de secteurs. Il est normal que les secteurs soient entièrement alloués à un unique objet externe. Cependant, on peut noter qu'un disque de 20 Mo structuré en secteurs de 512 octets, en comporte 40000, alors qu'un disque de 5 Go structuré en secteurs de 4096 octets, en comporte 1.2 millions. Devant une aussi grande diversité, et de telles valeurs, il est nécessaire de définir ce que l'on appelle le *quantum d'allocation* qui est la plus petite unité d'espace qui peut être alloué lors d'une demande. L'espace est ainsi découpé en unités de même taille, constituées de secteurs consécutifs, ce que nous avons déjà appelé des blocs (rappelons que ces blocs sont parfois appelés des clusters).

La valeur du quantum a deux conséquences:

- La *perte d'espace*. L'espace effectivement nécessaire à un objet peut se mesurer en nombre de ses enregistrements logiques, ou plus généralement, en nombre effectif d'octets qu'occupe sa représentation. L'espace qui lui est alloué se mesure en nombre de quanta qu'il a reçu. Comme il reçoit un nombre entier de quanta, le dernier reçu n'est que partiellement occupé. Ceci induit une perte d'espace que l'on peut évaluer à 1/2 quantum par objet existant sur le disque. Pour un nombre donné d'objets, la perte sera d'autant plus grande que le quantum sera plus important.
- Le *nombre d'unités allouables*. Pour un disque de N octets, et un quantum de q octets, le nombre d'unités allouables est N/q. Il s'ensuit que, plus le quantum est petit, plus le nombre d'unités allouables est grand, ce qui a, en général, pour conséquence une augmentation de la taille de la représentation de l'espace libre, et de la durée d'exécution de l'algorithme d'allocation.

Pour les systèmes utilisant une allocation par zone de taille variable, une demande doit être exprimée en donnant le nombre de quanta d'espace contigu. Le quantum peut être une unité de base imposée par le système, et connue de l'utilisateur. Il s'ensuit alors que le nombre de quanta d'une unité de disque dépend du disque lui-même. Le système doit alors avoir un algorithme d'allocation qui en tienne compte. À l'opposé, certains systèmes imposent un nombre maximum de quanta quelle que soit la taille du support. Le quantum varie alors d'un disque à l'autre. Pour éviter que l'utilisateur ait à se préoccuper de cette taille, on fixe un quantum de base, et il exprime alors ses besoins en donnant le nombre de quanta de base qu'il désire. Le système déterminera alors le nombre de quanta de l'unité support qui donne une taille supérieure ou égale à ce nombre. L'utilisateur se verra ainsi alloué un espace supérieur ou égal à celui qu'il a demandé.

Par exemple, si on fixe le nombre maximum de quanta à 100 000, un disque de 256 Mo peut avoir 64 000 quanta de 4 Ko, et un disque de 5 Go peut avoir 78 000 quanta de 64 Ko. Le quantum de base pourrait être fixé à 4 Ko. Un utilisateur qui en veut 10, obtiendrait effectivement 10 quanta, soit 40 Ko, sur le disque de 256 Mo, et obtiendrait 1 quantum de 64 Ko sur celui de 5 Go. Il va sans dire que, dans tous les cas, l'objet peut utiliser la totalité de l'espace alloué. Ainsi dans le deuxième cas, si on a une implantation avec extension, cela veut dire que l'objet aura moins d'extensions. Le nombre maximum de quanta est également influencé par l'identification de chaque quantum. Ainsi, la FAT dans le cas de MSDOS, ou HFS dans le cas de MacOS limitaient à 16 bits cette identification, et donc interdisaient d'avoir plus de 65 536 quanta sur un disque, ce qui n'était pas un handicap pour les petits disques, mais l'est devenu avec les disques actuels. Pour pallier cet inconvénient, Microsoft a construit NTFS, et Apple HFS+.

Pour les systèmes utilisant une allocation par blocs individuels, le quantum se confond avec la taille du bloc. Pour l'utilisateur, la taille, et donc la valeur d'un quantum importe peu, puisqu'il n'exprime directement aucun besoin d'allocation, les allocations étant faites automatiquement lors des accès. Le quantum peut même dépendre du disque, et non pas être imposé par le système.

## 8.4.2. Représentation par table de bits

Plusieurs méthodes sont utilisées pour représenter l'espace libre. Il faut en fait savoir quelles sont les différentes unités d'allocation de un quantum, ou blocs, qui sont libres. La première consiste à utiliser une table de bits indicée par les numéros de blocs, et précisant leur état libre ou occupé. L'algorithme d'allocation consiste à parcourir cette table pour rechercher une entrée, ou plusieurs consécutives, ayant l'état libre. Au lieu de partir du début de la table, on peut considérer que l'on a une table circulaire, et commencer sur un numéro quelconque correspondant à un numéro de bloc au voisinage duquel on cherche à faire l'allocation de façon à minimiser ultérieurement les mouvements de bras.

En voici trois exemples, dans le cas d'implantation par blocs individuels:

- Minix, système voisin d'Unix sur micro-ordinateur écrit pour l'enseignement de système par Andrew Tanenbaum et son équipe, utilise une telle table de bits, et commence la recherche sur le numéro du premier bloc alloué à l'objet externe. NTFS utilise également cette méthode.
- Linux utilise également une table de bits, mais pour améliorer l'efficacité des accès, découpe le disque en groupes, chaque groupe ayant sa propre table de bits. Le système cherche à allouer les blocs prioritairement dans le groupe où est déjà situé l'objet externe, lui assurant ainsi une certaine localité.
- MS-DOS fusionne la table de bits avec la FAT dont nous avons déjà parlé. Lorsqu'un bloc est libre, l'entrée de la FAT qui lui correspond reçoit une valeur particulière (0). Lors d'une allocation, le système recherche une entrée ayant une valeur nulle, au voisinage du dernier bloc courant de l'objet externe concerné.

La méthode est également utilisable pour l'allocation d'une zone de p blocs. On recherche le premier bit à `LIBRE`, et on regarde s'il est suivi d'au moins p bits à `LIBRE`. Si ce n'est pas le cas, on recommence au delà. Voici l'algorithme:

```
i := 0; trouvé := faux; k := 0; { k = nombre de bits LIBRE trouvés successifs }
tant que i < N et non trouvé faire
     si état [i] = LIBRE alors
          si k = 0 alors j := i; { repére le début } finsi;
          k := k + 1; trouvé := k = p;  { on en a trouvé p successifs }
     sinon k := 0;
     finsi;
     i := i + 1;
fait; { si trouvé alors allouer de j à j + p - 1 }
```

Pour améliorer cette recherche, certains systèmes complètent la table de bits par un liste partielle de zones libres contiguës. On ne recherche dans la table que si on n'a pas trouvé satisfaction dans cette liste.

### 8.4.3. Représentation par liste des zones libres

La deuxième méthode consiste à avoir une ou plusieurs listes des zones libres, avec leur longueur. Il en existe de nombreuses variantes, suivant la façon dont les zones sont ordonnées dans la liste par tailles croissantes ou par adresses croissantes. On peut aussi éclater la liste en plusieurs listes, chacune d'elle correspondant à une taille donnée. Nous ne décrirons cette méthode que dans un cas particulier simple, le cas lié à l'implantation par blocs individuels, car alors toutes les zones peuvent être ramenées à des blocs individuels, c'est-à-dire des zones d'une seule taille.

Elle est souvent utilisée dans les systèmes Unix. Pour chaque disque, le système dispose d'une petite liste de ce type en mémoire centrale. L'allocation consiste à allouer le premier bloc de la liste. La libération d'un bloc consiste à le remettre en tête de la liste. Lorsqu'elle devient trop importante, l'un des blocs est extrait de la liste, et on y range une partie de la liste. Ces blocs de liste de blocs libres sont eux-mêmes chaînés entre eux. Lorsque la liste en mémoire centrale devient trop petite, on la complète avec celle contenue dans le premier bloc de la liste des blocs libres (figure 8.6). Cette méthode ne permet pas de regrouper un objet sur des blocs voisins sur disque, mais permet une allocation/libération immédiate, tout en ayant un nombre quelconque d'unités d'allocation. Si beaucoup de blocs sont libres, cette liste est répartie dans des blocs libres eux-mêmes. Lorsqu'il y en a peu, la liste occupe peu de place sur disque.

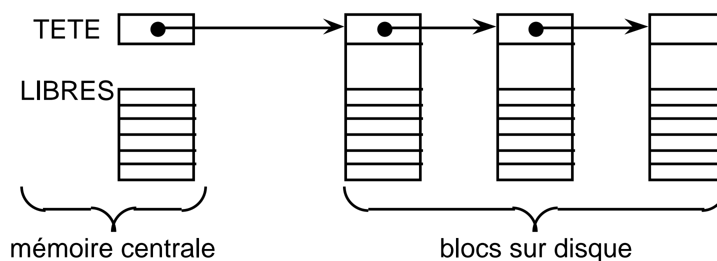



***Fig. 8.6.*** *Représentation des listes de blocs libres.*

## 8.5. Conclusion

☞ L'espace disque peut être linéarisé par les numéros virtuels de secteur.

☞ Les secteurs de l'espace appartenant à un objet peuvent recevoir un numéro logique propre à l'objet. La représentation de l'espace alloué à l'objet doit permettre de transformer ces numéros logiques en numéros virtuels.

☞ L'allocation par zone consiste à allouer des secteurs contigus. Elle présente l'inconvénient de ne pas toujours être possible, bien que l'espace libre soit suffisant.

☞ L'implantation séquentielle d'un objet consiste à l'implanter dans des secteurs contigus, en utilisant l'allocation par zone. Pour permettre les rallongements, on peut utiliser des extensions, chacune d’elles étant constituée également de secteurs contigus. Le nombre d'extensions est en général faible et borné.

☞ L'allocation par blocs de taille fixe consiste à découper l'espace en blocs, qui sont tous constitués du même nombre de secteurs, et qui sont alloués individuellement. L'avantage de l'allocation par blocs est qu'elle est impossible uniquement lorsque tout l'espace est occupé.

☞ L'implantation par blocs peut être obtenue en chaînant entre eux les blocs d'un même objet externe, ce qui peut être inefficace pour les accès aléatoires. Elle peut encore être obtenue par une table qui fait correspondre aux numéros logiques de blocs le numéro virtuel de celui qui a été alloué à cet endroit. La taille de cette table peut être adaptée à la taille de l'objet en utilisant plusieurs niveaux.

☞ Le quantum d'allocation est la plus petite quantité d'espace qui peut être allouée lors d'une demande. Si elle est petite, le nombre d'unités différentes est important. Si elle est grande, cela induit une perte d'espace.

☞ Le quantum peut être fixe pour un système, ou dépendre du type de disque. Dans ce dernier cas, l'utilisateur exprime ses besoins en quanta de base et le système en déduit le nombre de quanta du disque qui est nécessaire.

☞ L'espace libre peut être représenté soit par une table de bits indiquant l'état libre ou occupé de chaque unité individuelle soit par des listes de zones libres. Ces deux représentations sont utilisables pour les deux types d'algorithmes d'allocation.

# 9

# La désignation des objets externes

Nous avons dit précédemment que l'accès à un objet externe dans un programme se faisait par l'intermédiaire d'un fichier logique, point de vue du programmeur, qui devait être relié à l'exécution avec un tel objet externe. L'établissement du lien proprement dit est réalisé par l'opération d'ouverture du fichier. Les paramètres de cette opération doivent permettre de localiser l'objet externe, et d'en trouver les caractéristiques.

## 9.1. La définition d'une liaison

Les avis divergent sur l'endroit où doit se trouver cette définition. Dans certains systèmes, cette définition ne fait pas partie du programme, mais de son environnement. Aussi, c'est le langage de commande qui propose des commandes permettant cette définition. Ceci offre l'avantage de bien faire la séparation entre l'algorithme et les objets sur lesquels s'exécute l'algorithme. Par ailleurs, cela permet de prendre en compte un grand nombre de paramètres, avec des valeurs par défaut pour chacun d'eux, ce qui n'est pas toujours possible dans les langages évolués. Enfin cela évite de faire dépendre certaines fonctionnalités du système, de la possibilité de les exprimer dans les langages évolués. Cette méthode est souvent appliquée sur les gros systèmes qui privilégient le traitement par lots, et qui peuvent recevoir des périphériques divers, impliquant un nombre important de paramètres de la définition d'une liaison (près de 150 paramètres pour une commande `//DD` d'IBM). Cela permet également au système de savoir quels seront les besoins du programme avant d'en lancer l'exécution, et de la retarder si ces besoins ne peuvent être satisfaits. En général, la mise en correspondance entre le programme et le langage de commande est obtenue par un nom logique (identificateur) donné au fichier explicitement par le programmeur ou implicitement par le traducteur, et repris par l'utilisateur dans la commande de définition.

Par opposition, d'autres systèmes supposent que les paramètres de l'ouverture d'un fichier sont déterminés par le programme et sont partie intégrante de la demande d'exécution de l'opération. On évite de trop fortes contraintes entre les langages et le système en ayant assez peu de paramètres, dont au moins un est une simple chaîne de caractères qui n'est pas interprétée par le langage évolué, et qui joue ainsi le rôle de “commande”. Cette méthode offre l'avantage de permettre au programme d'avoir une certaine maîtrise sur la définition de la liaison. En particulier, lorsque le programme est utilisé de façon interactive, il est possible de cacher à l'utilisateur du programme la complexité de l'interface système concernant les objets externes, sans pour autant devoir toujours relier le fichier avec le même objet externe.

Lorsque l'objet externe est directement un périphérique physique, la définition de la liaison est relativement simple, car ces périphériques existent en nombre très limité, et leurs nombres ou caractéristiques varient très peu pour une installation donnée. On peut donc leur donner un nom mnémonique spécifique qui est connu de tous les utilisateurs. Ce nom est ensuite utilisé par le système pour en trouver la localisation et les caractéristiques au moyen d'une recherche dans ses

tables internes. Il trouve ainsi, non seulement le descripteur du périphérique, mais également les procédures de traitement des opérations qui le concernent. Par ailleurs ces tables permettent de savoir l'état libre ou occupé de ces périphériques, et d'en assurer ainsi la gestion.

Lorsque l'objet n'est pas un périphérique, mais est un fichier physique sur un support magnétique, la définition doit permettre de:

- localiser le support physique de l'objet,
- localiser l'objet sur le support,
- déterminer les caractéristiques de l'objet.

La notion de volume est souvent utilisée pour permettre la localisation du support, et la notion de répertoire pour permettre la localisation sur le support ainsi que la détermination des caractéristiques de l'objet lorsqu'il existe déjà.

# 9.2. La notion de volume

La localisation du support par localisation du périphérique n'est pas acceptable dès que le support lui-même est amovible. Le périphérique permet d'accéder au support qui est actuellement monté sur ce périphérique. Pour l'utilisateur, il importe plus d'accéder à un support donné que d'accéder au périphérique sur lequel il est monté. Pour la bonne exécution des opérations, il importe plus au système de savoir quel périphérique est concerné par les opérations. En première approche, on appelle *volume* un support d'informations, tel que, par exemple, une bande magnétique, une cartouche, une disquette, ou une pile de disque (une gamelle). Il est important de noter que le volume est quelque chose que l'on manipule comme un tout, et que, comme support d'objets externes, il a une structure qui lui est propre. En tant que support, il est de nature physique, mais en tant que structure, il est de nature logique.

Le fait qu'il puisse être amovible, implique qu'il peut ne pas être accessible, et c'est alors sa nature physique seule qui nous intéresse, puisqu'on ne peut lire alors son contenu. Au contraire, pour être accessible, il doit être disponible sur un périphérique particulier, et sa structure reconnue, et c'est donc sa nature logique seule qui nous intéresse. On appelle *montage d'un volume* l'opération qui rend accessible son contenu, et *démontage* l'opération qui le rend inaccessible.

## 9.2.1. La structuration d'un volume

Le fait qu'un volume puisse être amovible implique que l'ensemble des informations qui y sont mémorisées doivent être suffisantes pour en définir le contenu. Il est donc nécessaire que l'on y trouve deux types d'informations:

- les caractéristiques des objets externes qu'il contient, ainsi que leur localisation sur le volume,
- les informations permettant au système d'y créer de nouveaux objets externes sans écraser ceux qui y sont déjà.

Le permier type d'informations est directement lié à la localisation des objets sur le support. Nous verrons ultérieurement les structures permettant de résoudre ce problème. Le deuxième type d'informations est en fait la représentation de l'espace libre dont nous avons parlé dans le chapitre précédent.

Lorsqu'un volume n'est pas monté, c'est sa nature physique qui nous intéresse. Encore faut-il être capable de l'identifier parmi l'ensemble des volumes existants pour l'installation. Ceci passe en général par une étiquette collée directement sur le support, qui permet à l'opérateur d'accéder à cette identification. Pour permettre un contrôle par le système de cette identification, le contenu de cette étiquette est souvent recopié dans le volume lui-même. C'est ce que nous appellerons le *nom du volume* (en Anglais le *label* du volume).

L'ensemble de ces informations de gestion doivent évidemment être placées à des endroits prédéfinis par le système de façon à lui permettre de les retrouver. Ceci doit être fait avant de faire un accès quelconque au support. C'est ce que l'on appelle le *formatage du volume*. Il consiste à lui donner un nom, à construire la représentation de l'espace libre initial et à initialiser la structure de données qui permet de localiser les objets existants sur le volume.

Ce formatage peut être conforme à un standard international (ANSI) ou national (AFNOR), ce qui permettra aux utilisateurs d'échanger ces volumes pour communiquer des données. C'est souvent le cas pour les bandes magnétiques. Ce formatage peut également être un “standard de fait”, défini par un système et universellement adopté par d'autres systèmes, comme par exemple les disquettes MS-DOS. Ce formatage peut être propre à un constructeur pour tout ou partie de sa gamme. C'est souvent le cas pour les disques, car ils sont assez peu utilisés pour permettre la communication entre machines, mais essentiellement pour la mémorisation locale. La portabilité n'est pas alors un critère important, alors que l'efficacité de l'exploitation du support l'est.

### 9.2.2. Le montage de volume

L'opération de montage de volume est constituée de trois étapes: choix du périphérique, montage physique du volume sur le périphérique et modification des tables du système. Lorsque ces opérations sont terminées, le système connaît l'association entre le nom du volume logique et le périphérique. L'utilisateur n'a plus à s'en préoccuper et peut donc localiser le périphérique par le nom du volume logique. L'opération de démontage est elle en deux étapes: modification des tables du système puis démontage physique. Il est en effet important qu'il y ait accord préalable entre l'opérateur et le système pour qu'aucun lien ne soit établi au moment du démontage physique, ce qui pourrait nuire à la cohérence des objets sur le support.

Le choix du périphérique, que l'on appelle l'*allocation*, peut être fait par le système de façon automatique. En effet, le système connaît un besoin immédiat d'un programme, soit par la présence de la définition d'un lien dans les commandes qui précèdent la demande d'exécution d'un programme, soit par une demande d'établissement du lien par le programme. Si le volume désiré n'est pas monté, le système peut alors allouer lui-même un périphérique parmi ceux qui sont libres, c'est-à-dire, ceux qui ne sont pas associés à un volume, et demander à l'opérateur de faire le montage physique du volume concerné sur ce périphérique. Cette méthode est surtout appliquée lorsque la définition des liens est donnée dans le langage de commandes, car le système peut anticiper les allocations et faire les choix en fonction de l'ensemble des besoins de tous les programmes qui sont en attente d'exécution.

L'allocation peut être faite par l'opérateur, par anticipation, car il a connaissance du besoin avant qu'il ne soit effectivement exprimé au système. C'est souvent le cas lorsque l'on désire permettre les accès aux objets mémorisés sur un volume pendant une période définie de la journée. Notons que l'opérateur peut constater de visu l'état libre d'un périphérique par le fait qu'aucun support n'y est actuellement monté physiquement.

Le montage physique est une opération manuelle (sauf pour les robots spécialisés de montage de cartouches). L'opérateur retrouve dans un stock plus ou moins important de volumes, celui qui est demandé et qui est désigné par son étiquette, et le place sur le périphérique alloué. Durant cette opération, le périphérique est “déconnecté” de l'ordinateur, c'est-à-dire, que celui-ci ne peut y accéder. On dit qu'il est en mode *local* (ou encore *off line*). Lorsque le placement est terminé, l'opérateur le fait passer en mode *distant* (ou encore *on line*). Cette opération n'est en général pas instantanée, car elle demande, par exemple, un positionnement sur les dérouleurs de bande, et une mise en rotation sur les tourne-disques.

Lorsque le montage physique est terminé, le système doit mettre à jour ses tables. Ceci peut être fait manuellement par une commande de l'opérateur, ou automatiquement lorsque le système détecte le passage en mode distant d'un périphérique. Dans ce dernier cas, on dit qu'il y a *reconnaissance automatique de volume*. Le système peut alors lire le nom du volume sur le périphérique, et mémoriser dans ses tables ce nom associé au périphérique. Notons que même si l'allocation de périphérique a été faite par le système, l'opérateur peut néanmoins placer le volume sur un autre périphérique libre. C'est bien au moment où le système lit le nom du volume sur le périphérique que l'association est faite. Avant cet instant, il ne s'agit que de propositions.

### 9.2.3. La relation entre volume et support

La notion de volume est évidemment liée au support, ce qui implique des contraintes de taille de l'espace total, qui peuvent être jugées trop fortes. D'une part, un support peut être de taille trop faible pour contenir un objet externe donné, qui pourtant doit se trouver sur un seul volume. D'autre part, nous avons vu dans le chapitre précédent qu'un support de très grande taille était parfois mal adapté pour recevoir beaucoup de petits objets.

Le premier cas est surtout celui des bandes magnétiques. Certains systèmes offrent la notion de *multi-volumes*. Elle peut être vue comme une facilité du logiciel, de définir la localisation du support non plus au moyen d'un volume unique, mais par une collection de volumes. Le système considèrera la collection comme une seule bande, les volumes étant dans l'ordre où la collection est définie.

Le deuxième cas est celui des disques. Dans ce cas, il est assez courant maintenant de découper l'ensemble de l'espace d'un disque en *partitions* qui sont des portions indépendantes du disque, et qui en tant que tel seront considérées comme des supports séparés. Chaque partition est ensuite structurée en volume. Lorsque le disque est monté, (passage en mode distant), le système explore son découpage en partition pour identifier chacun de ces supports permettant ensuite le montage des volumes qui les structurent.

Certains systèmes combinent les deux principes, en permettant d'agréger plusieurs partitions situées éventuellement sur des disques différents en un seul support qui peut ainsi être structuré en un volume unique. Cela permet d'obtenir un volume de grande taille à partir de plusieurs petits disques, ou d'utiliser au mieux de petites partitions restantes éparpillées sur plusieurs disques. Notons que la répartition des données d'un volume sur plusieurs disques peut améliorer les performances, car elle permet les accès simultanés. Nous verrons d'autres applications dans le chapitre suivant.

## 9.3. La notion de répertoire

Lorsque la localisation du support physique est obtenue, il faut localiser l'objet externe sur le support. La méthode qui consiste à le localiser explicitement par les informations numériques dont nous avons parlé dans le chapitre 8, a été utilisée pendant quelques années, mais s'est avérée très vite inappropriée car elle est beaucoup trop sujette à erreurs. Il est plus judicieux de donner, aux objets externes, un *nom symbolique* constitué d'une chaîne de caractères. Un *répertoire* (ou encore un *catalogue*, en anglais *directory*) est en fait une table qui associe le nom symbolique d'un objet externe et le *descripteur de cet objet externe*, c'est-à-dire, les informations de localisation de l'objet sur le support ainsi que ses caractéristiques. Il est naturel de conserver ces informations sur le support lui-même, puisqu'elles sont nécessaires si et seulement si le volume correspondant est monté.

### 9.3.1. Le répertoire sur bande magnétique

Sur une bande magnétique, la nature séquentielle du support conduit naturellement à faire précéder l'objet externe d'un bloc spécial (appelé *label de fichier*) qui contiendra son nom symbolique et ses caractéristiques. Aucune information de localisation n'est alors nécessaire, puisqu'il est situé immédiatement derrière, si ce n'est un autre bloc spécial (appelé *label de fin de fichier*) qui en indique la fin. Pour accélérer les accès lors de la recherche d'un objet externe par son nom, la suite des blocs est structurée à l'aide de *blocs drapeaux*: ce sont des blocs physiques sur la bande, qui sont écrits par des commandes particulières et qui sont reconnus par le matériel même lorsque le déroulement se fait à grande vitesse (défilement rapide avant ou arrière). La figure 9.1 donne un exemple d'une telle structuration. Notons que si le dernier fichier est multi-volumes, la marque de fin de fichier n'est pas présente, et on trouve directement la marque de fin de volume.

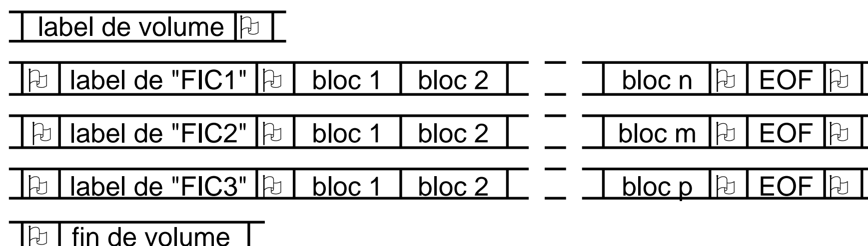



***Fig. 9.1.*** *Exemple de structuration d'une bande magnétique. ⚐ représente un bloc drapeau, EOF le label de fin de fichier.*

### 9.3.2. Répertoire simple sur disque

Sur disque, le secteur 0 est en général dédié pour donner les informations de base sur sa structure. Il faut en effet que ce soit au même endroit sur tous les volumes, et les fabricants garantissent que ce

secteur est exempt de tout défaut. Si le disque est partitionné, son secteur 0 définit la carte d'implantation des partitions, et le secteur 0 de chacune d'elles reçoit les informations de base sur la structure de la partition. Ce secteur peut donc en particulier contenir le nom du volume, mais aussi la localisation des informations nécessaires à la gestion de l'espace libre.

Une première solution consiste à réserver une partie dédiée de l'espace à la représentation du répertoire. Si elle est de taille fixe, cela limite le nombre d'objets externes que l'on peut mettre sur ce disque. Cette taille doit alors être judicieusement choisie. La localisation du répertoire peut être fixe ou variable en fonction du volume. Lorsque sa localisation est variable, elle est en générale repérée par des informations situées dans le secteur qui contient le nom du volume. Lorsque le disque doit recevoir des objets externes appartenant à plusieurs utilisateurs différents et indépendants, il peut être gênant d'avoir un seul répertoire, car les noms des objets doivent être distincts, et les utilisateurs doivent alors se mettre d'accord sur les noms qu'ils utilisent. Une solution consiste à ajouter implicitement devant les noms des objets les noms des utilisateurs.

On peut aussi découper le répertoire en tranche, chaque tranche étant attribuée à un utilisateur particulier. En fait, on dispose plutôt d'un *super-répertoire* qui permet la localisation des répertoires individuels des utilisateurs. Seul ce super-répertoire doit être localisé de façon absolue, les autres sont considérés comme des objets externes particuliers, qui peuvent être créés au fur et à mesure des besoins. MS-DOS, dans ses premières versions, mais aussi certains gros systèmes utilisent cette méthode.

## 9.3.3. Arborescence de répertoires

Lorsque le nombre d'objets externes devient important, les solutions précédentes ne sont plus viables. Dans un système Unix, par exemple, on a constaté qu'il y avait 47000 fichiers sur un disque de 400 Mo. Même si on suppose que ces fichiers appartiennent à 100 utilisateurs différents, cela donne 470 fichiers en moyenne par utilisateurs. Même sur un ordinateur personnel, il n'est pas rare d'avoir près de 5000 fichiers sur un disque de 500 Mo. Si tous ces fichiers sont repérés dans un répertoire unique, cela implique d'une part de bien choisir les noms pour n'avoir pas de confusion, d'autre part cela entraîne des temps de recherche importants dans le répertoire.

La solution qui est couramment adoptée actuellement est de partitionner l'espace des noms des objets externes. Notons que, dans l'approche du paragraphe précédent, nous avions introduit la notion de super-répertoire permettant de disposer de répertoires séparés pour les utilisateurs. Ceci nous a conduit à considérer qu'un répertoire était un objet externe particulier. La généralisation de cette idée consiste à considérer qu'un répertoire est une table qui associe un nom et un descripteur d'objet externe au sens large, c'est-à-dire fichier de données manipulé par un programme, ou répertoire. Au lieu d'un répertoire plat, on obtient une arborescence de répertoire, dont seule la racine est localisée de façon absolue.

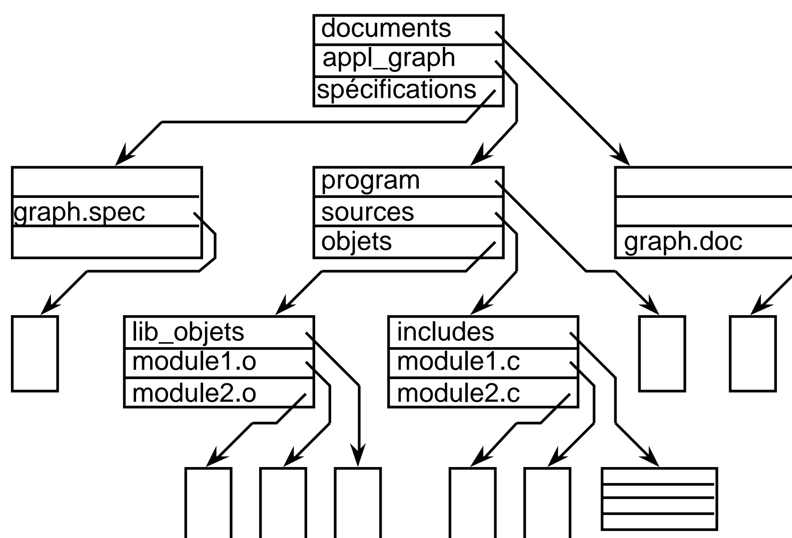



***Fig. 9.2.*** *Exemple d'arborescence de répertoire.*

La figure 9.2 donne un exemple d'une telle arborescence. Le répertoire racine contient trois entrées pour les répertoires de nom `documents`, `appl_graph` et `specifications`. Le répertoire

appl_graph contient aussi trois entrées, la première associe le nom program et un fichier, alors que les deux autres associent les noms sources et objets avec deux répertoires. Enfin, le répertoire sources contient une entrée pour le répertoire includes et deux entrées pour les fichiers de nom module1.c et module2.c. Cet exemple montre tout le bénéfice qu'un utilisateur peut retirer d'une arborescence, en regroupant dans un même répertoire des entités qui présentent une propriété de cohérence donnée. Ainsi, le répertoire documents pourrait regrouper les fichiers de documentation, et le répertoire sources regrouper les fichiers sources de l'application appl_graph, dont includes constitue le sous-groupe des fichiers qui peuvent être inclus dans les fichiers sources par les préprocesseurs.

La désignation d'un objet externe doit permettre d'une part d'identifier le volume qui contient l'objet et d'autre part de localiser l'objet sur ce volume. Lorsqu'on a une arborescence de répertoires, la localisation de l'objet sur le volume consiste donc à définir le chemin depuis la racine jusqu'à l'objet dans cette arborescence. C'est donc une suite de noms qui doivent être utilisés, pour désigner la suite des répertoires à parcourir pour atteindre l'objet recherché. Chacun de ces noms permet de désigner le répertoire suivant (ou l'objet recherché) dans le répertoire courant. Un nom étant une chaîne de caractères, il est courant de définir une structure lexicale rudimentaire qui permet de définir sous la forme d'une seule chaîne de caractères l'identification du volume et la suite de noms précédente. C'est ce que l'on appelle le *nom absolu de l'objet* ou encore le *chemin d'accès à l'objet* (en Anglais le *pathname*).

Par exemple, les versions récentes de MS-DOS utilisent le caractère “:” comme séparateur entre le nom du volume (en fait le nom symbolique du périphérique) et le nom de l'objet sur le volume, et le caractère “\” comme séparateur entre les noms de répertoires, ce que l'on peut énoncer de la façon suivante:

```
[nom_de_volume:]{\nom_répertoire}\nom_objet
```

Ainsi le nom absolu “C:\appl_graph\sources\module1.c” désigne un objet sur le volume C. Sur ce volume, il faut rechercher dans le répertoire racine le répertoire appl_graph, puis dans celui-ci le répertoire sources, et dans ce dernier l'objet module1.c.

# 9.4. Construction d'une arborescence unique à la Unix

La structure de désignation à laquelle nous venons d'aboutir dans le paragraphe précédent s'apparente à une arborescence unique: à la racine se trouve la désignation des volumes ou des périphériques, permettant l'accès à une sous-arborescence locale à chaque volume. Le système Unix présente à l'utilisateur l'ensemble des objets sous une véritable arborescence unique et une désignation uniforme dans cette arborescence.

## 9.4.1. Les fichiers spéciaux comme périphériques

On peut remarquer qu'une entrée particulière d'un répertoire associe un nom, local au répertoire, et un descripteur d'objet. Si l'objet est un fichier au sens classique du terme, le descripteur doit contenir les caractéristiques du fichier (organisation, par exemple) et les informations de localisation du contenu. Si l'objet est un répertoire, le descripteur doit contenir également les caractéristiques, en particulier, l'indication qu'il s'agit d'un répertoire, et les informations de localisation du contenu. Ce descripteur doit donc permettre de distinguer la nature fichier ou répertoire de l'objet associé.

L'idée des concepteurs d'Unix est que l'on peut créer un descripteur de nature différente des précédentes, appelé *fichier spécial*, pour désigner cette fois un périphérique de l'installation. Un tel descripteur contient, en particulier, un *numéro de périphérique majeur* qui détermine le type du périphérique (terminal, disque, bande, etc...) et un *numéro de périphérique mineur* qui détermine le périphérique particulier du type correspondant. Évidemment ces numéros n'ont de sens que sur une installation donnée.

Dans la plupart des installations, un répertoire particulier (/dev) regroupe les descripteurs de tous les périphériques, et les noms locaux qui leur sont associés. Par exemple /dev/lp désigne l'objet de nom lp dans le répertoire de nom dev situé à la racine. C'est la nature du descripteur de l'objet qui permet au système de constater qu'il s'agit de l'imprimante de l'installation. L'intérêt est donc essentiellement dans la désignation uniforme ainsi obtenue.

## 9.4.2. Le montage de volume dans Unix

Comme nous l'avons dit, un volume contient sa propre arborescence locale qui permet la désignation des objets sur le volume. Pour obtenir une arborescence unique lorsque l'on monte un volume, il faut raccrocher cette arborescence locale quelque part dans l'arborescence courante. L'option adoptée par le système est de permettre de la raccrocher n'importe où, à la place d'un répertoire. L'arborescence locale vient alors remplacer la sous-arborescence issue de ce répertoire. En général, on utilise un répertoire vide, mais ce n'est pas obligatoire. Si ce n'est pas le cas, les objets qu'il contenait sont momentanément inaccessibles.

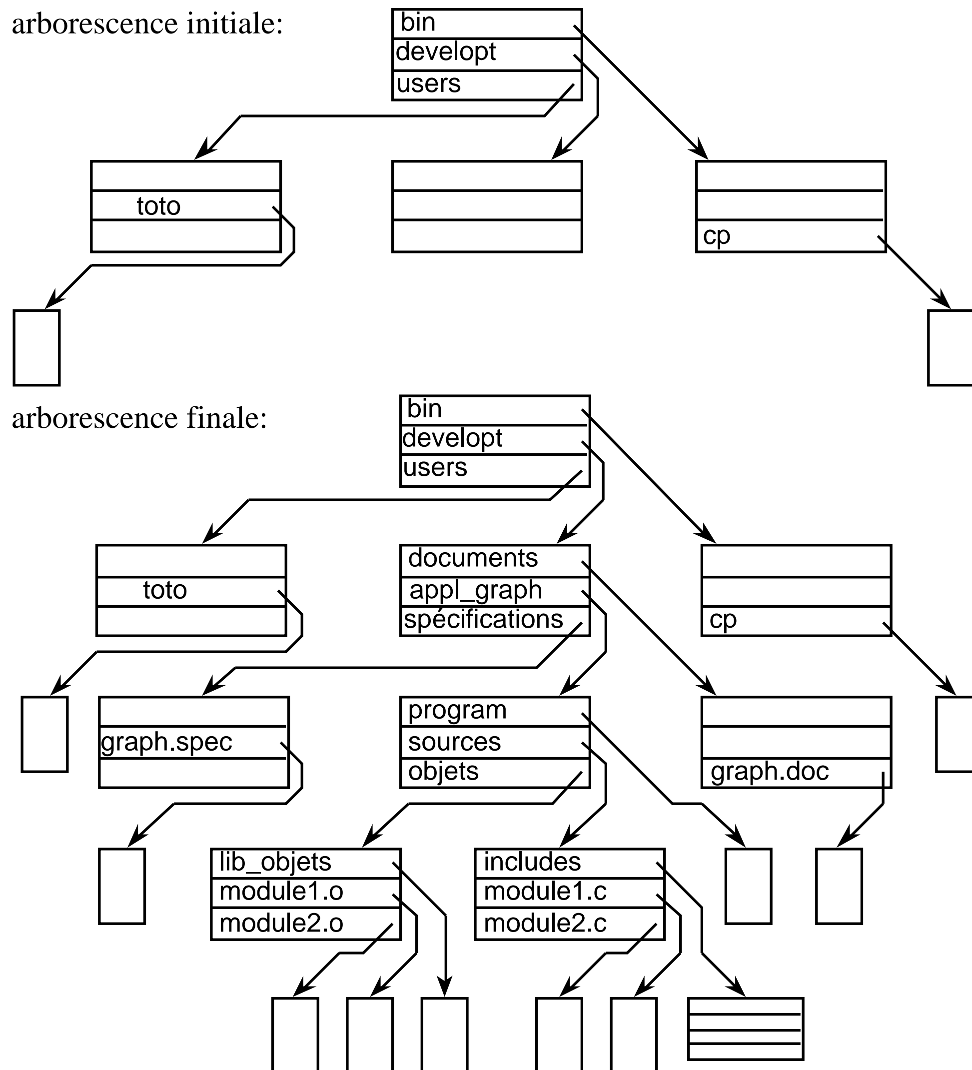



***Fig. 9.3***. *Exemple de montage d'un volume dans Unix. L'arborescence locale du volume remplace l'arborescence de /developt.*

La figure 9.3 montre le résultat du montage du volume dont l'arborescence a été présentée en figure 9.2, sur le répertoire de nom absolu `/developt` dans l'arborescence initiale. Lorsque le montage est terminé, on accède aux objets du volume en faisant précéder le nom local de l'objet sur le volume par le nom absolu du répertoire sur lequel le volume a été monté, comme, par exemple, dans:

```
/developt/appl_graph/sources/module1.c
```

En général les opérations de montage sont effectuées par l'opérateur qui choisit le périphérique (allocation manuelle) et le répertoire sur lequel les volumes sont montés. Pour l'utilisateur, il ne voit qu'une arborescence unique, et n'a plus à se préoccuper du support de ses objets. Comme les périphériques sont aussi désignés dans l'arborescence, il a donc un mécanisme uniforme de désignation de tous les objets externes.

### 9.4.3. Les fichiers multi-répertoires

Lorsque plusieurs utilisateurs travaillent sur un projet commun, ils doivent pouvoir accéder à des fichiers qu'ils ont en commun. En fonction de ce que nous avons dit ce n'est pas difficile, puisque tout objet a un nom absolu. Il arrive souvent que chacun aimerait le voir à un endroit précis de l'arborescence, qui n'est pas le même pour tous. Certains systèmes qui offrent une arborescence de fichiers (VMS, Unix, Multics) permettent de situer un même fichier à plusieurs endroits de cette arborescence. Évidemment, nous n'avons plus tout à fait un arbre, mais un *graphe orienté sans cycle*.

Deux méthodes sont utilisées pour cela. La première utilise ce que l'on appelle les *liens physiques*. Ils ne sont autorisés que pour les fichiers de données (et donc interdits pour les répertoires ou les fichiers spéciaux). Pour pouvoir prendre en compte les liens physiques, les descripteurs d'objets ne sont pas mis dans les répertoires, mais dans une zone particulière du disque, et sont repérés par un numéro appelé *i-nœud* dans le système Unix. Les répertoires associent les noms symboliques locaux et les numéros de descripteurs d'objets. Il est ainsi possible de créer une autre association entre un nom et ce numéro dans un autre répertoire. Le fichier possède alors deux (ou plus) noms absolus. L'utilisateur ne détruit plus les objets directement, mais les liens physiques vers ces objets. La destruction d'un objet externe intervient lorsque le système constate qu'il n'y a plus de lien physique pour cet objet. Évidemment si un utilisateur détruit un lien physique dans un répertoire vers un objet commun, et recrée un nouvel objet de même nom dans ce répertoire, l'objet ancien reste désigné par ses autres noms symboliques absolus.

La deuxième méthode utilise ce que l'on appelle les *liens symboliques*. Lorsque l'on crée un lien symbolique `n` dans un répertoire pour un objet `o` situé ailleurs, le système n'associe pas le nom local avec le descripteur de l'objet `o`, mais avec le nom symbolique absolu de l'objet `o`. L'objet `o` peut d'ailleurs ne pas exister au moment où l'on crée le lien symbolique. Par contre lorsque l'on cherche à accéder à l'objet par le nom `n`, le système interprètera à ce moment le nom symbolique absolu qui lui est associé pour trouver `o`. Évidemment comme le nom symbolique absolu est réinterprété à chaque désignation qui utilise `n`, cela est plus coûteux, mais on est sûr alors de ne pas trouver une ancienne version de l'objet. Par ailleurs, les liens symboliques sont autorisés pour les objets de toute nature, y compris les répertoires. Cette méthode se généralise bien aux systèmes répartis.

### 9.4.4. La désignation par rapport à l'environnement

L'arborescence de répertoire conduit à une désignation des objets relativement longue. On a donc cherché à alléger cette désignation. De fait, si une installation comporte plus de 100 000 objets externes, un utilisateur n'a besoin d'accéder qu'à une petite partie d'entre eux à un instant donné. Plusieurs solutions ont été proposées, en particulier, la notion de *répertoire de travail* et la notion de *règles de recherche*.

Pour tout utilisateur actif, le système privilégie un répertoire particulier de l'arborescence, et que l'on appelle le répertoire de travail. Des commandes permettent à l'utilisateur de changer son répertoire de travail à tout instant. Outre la désignation des objets par leur nom absolu, c'est-à-dire, en partant de la racine de l'arborescence, l'utilisateur peut désigner un objet en donnant son chemin d'accès à partir de ce répertoire courant. En Unix, tout chemin d'accès ne commençant pas par “`/`” est considéré comme relatif au répertoire courant. Il en va de même pour MS-DOS, si le chemin ne commence pas par “\”. Pour permettre de remonter dans l'arborescence, chaque répertoire de ces systèmes contient une entrée, dont le nom est prédéfini “`..`”, qui est associée au répertoire situé au-dessus dans l'arborescence. Pour être complet, signalons qu'il existe une deuxième entrée de nom prédéfini “`.`” qui est associée au répertoire lui-même. Ainsi, après la commande suivante qui définit le répertoire courant:

```
cd /developt/appl_graph/sources
```

le fichier de nom absolu `/developt/appl_graph/sources/module1.c` peut être désigné simplement par `module1.c`.

La notion de règles de recherche consiste à définir une liste de noms absolus de répertoires qui sont utilisés pour la recherche d'un objet dans un contexte particulier. Par exemple, on définit couramment une liste de répertoire pour la recherche du programme exécutable d'une commande, ce qui évite de devoir se rappeler dans quel répertoire il est situé. De même, on peut définir une liste de répertoire dans lequel il faut rechercher les fichiers d'un manuel en ligne du système. Ces règles

de recherche peuvent être définies implicitement par le système, et modifiables au gré de l'utilisateur qui peut ainsi adapter son environnement à sa convenance.

# 9.5. Conclusion

☞ La définition d'une liaison doit permettre de localiser le support physique de l'objet, de localiser l'objet sur le support et de déterminer les caractéristiques de l'objet. Elle est donnée soit par le langage de commande soit par les programmes.

☞ Un volume est un support d'informations. Pour être accessible il doit être monté physiquement sur un périphérique et logiquement reconnu par le système, c'est-à-dire que le système associe ce périphérique à son nom.

☞ Le formatage d'un volume consiste à lui donner une structure qui comporte en particulier son nom, la représentation de l'espace libre et les structures de données nécessaires à la localisation des objets qu'il contient.

☞ Un répertoire est une table qui associe un nom symbolique et le descripteur d'un objet externe. Mémorisé sur un volume, il permet de localiser les objets de ce volume. Lorsque le nombre d'objets est grand, on utilise une arborescence de répertoire.

☞ Le chemin d'accès à un objet est la concaténation du nom de volume sur lequel il se trouve et de la suite des noms dans les répertoires de l'arborescence qui permet de l'atteindre.

☞ Unix construit une arborescence unique pour tous les objets, y compris les périphériques, qui ont ainsi une désignation uniforme qui permet à l'utilisateur de s'abstraire des périphériques.

☞ Les liens physiques ou symboliques permettent de donner plusieurs noms absolus aux objets.

☞ La désignation peut être allégée en faisant référence à l'environnement, au moyen de la notion de répertoire de travail et de règles de recherche.

# 10

# La sécurité et la protection des objets externes

L'utilisateur qui confie des données à un système informatique s'attend à un certain service, et en particulier, désire une certaine garantie de retrouver ses données à tout moment, dans l'état exact où il les a laissées. Nous distinguerons d'une part, une altération due à une mauvaise utilisation des opérations sur les données et d'autre part, une altération de ces données par suite de défaillance du matériel ou du logiciel. Dans le premier cas, on parle de *protection* pour désigner l'ensemble des méthodes qui spécifient les règles d'utilisation des opérations. Dans le deuxième cas, on parle de *sécurité* pour désigner l'ensemble des méthodes qui assurent que les opérations se déroulent conformément à leur spécification, même en cas de défaillance. La sécurité concerne la spécification de chaque opération, alors que la protection concerne les règles d'utilisation des opérations.

## 10.1. Les mécanismes de protection

### 10.1.1. La protection par droits d'accès

Le premier mécanisme proposé par de nombreux systèmes est basé sur la notion de droits d'accès. Pour chaque objet externe et pour chaque utilisateur, on définit les opérations qui sont autorisées. Lors d'une demande d'établissement d'un lien entre un programme et un objet externe, les paramètres de la demande précisent les opérations que le programme désire effectuer sur l'objet externe, et le système vérifie que l'utilisateur, pour le compte de qui travaille le programme, est bien autorisé à effectuer ces opérations.

En général, on distingue peu d'opérations différentes sur les objets externes. Les opérations essentielles sont la *lecture*, l'*écriture* et éventuellement l'*exécution*, sauf pour les répertoires où les opérations essentielles sont la *recherche*, la *création* et la *modification* d'une entrée, en plus de l'opération de lecture de l'ensemble des entrées, pour en faire une liste. Dans Unix les droits d'un utilisateur sur un objet sont définis par trois bits appelés *rwx*:

- *r* correspond au droit de lire le contenu de l'objet quelle que soit sa nature,
- *w* correspond au droit de modifier le contenu de l'objet quelle que soit sa nature,
- *x* sur un répertoire permet d'y rechercher une entrée, alors que sur les autres objets, il permet de l'exécuter.

Par ailleurs, lorsque le nombre d'utilisateurs du système est important, il n'est guère envisageable de disposer pour chaque objet des droits d'accès à cet objet pour chacun des utilisateurs pris individuellement. D'une part ceci conduirait à une occupation d'espace qui ne serait pas négligeable, mais d'autre part ceci demanderait à être défini lors de chaque création d'objet. Les systèmes

proposent en général des restrictions sur cette définition, et des règles implicites de définitions. Il est courant, par exemple, d'établir des groupes d'utilisateurs, et de définir les droits par groupe.

- Multics a introduit la notion de projet, pour regrouper des utilisateurs; tout utilisateur lors de son identification par le système (login), indique implicitement ou explicitement le projet sous lequel il désire travailler. La protection peut alors être définie par projet, c'est-à-dire, globalement pour tous ceux qui travaillent sous ce projet, ou par utilisateur quel que soit le projet, ou enfin pour un utilisateur particulier lorsqu'il travaille sous un projet donné. Par exemple, la définition des droits

  ```
  <r, COMPTABLE.*>, <rw, COMPTABLE.Jean, *.Paul>
  ```

  permet à tous ceux qui travaillent sous le projet `COMPTABLE` de lire l'objet externe, alors que peuvent le lire et y écrire, d'une part `Jean` lorsqu'il travaille sous ce projet, et d'autre part `Paul` quel que soit le projet sous lequel il travaille.
- Dans Unix, on définit également la notion de groupe d'utilisateurs, mais pour simplifier la représentation de l'ensemble des droits d'accès à un objet externe, on ne peut les définir que pour trois catégories d'utilisateurs: le propriétaire, les membres d'un groupe donné attaché à l'objet et les autres. Dans l'exemple précédent, si `Jean` est propriétaire de l'objet et membre du groupe `COMPTABLE` attaché à l'objet, on peut définir les droits suivants:

  ```
  rw- r-- ---
  ```

  c'est-à-dire `rw` au propriétaire, `r` aux membres du groupe, et aucun accès aux autres. Remarquons que l'on ne peut fournir l'accès `rw` à Paul dans ce cas, à moins de devoir le fournir à tous.
- Windows NT utilise également la notion de groupe d'utilisateurs, un utilisateur pouvant appartenir à plusieurs groupes en même temps. A chaque objet est associée une *liste de contrôle d'accès*. Par exemple,

  ```
  <~r, Pierre>, <r, COMPTABLE>, <rw, Paul>
  ```

  interdit à Pierre la lecture de l'objet, même s'il appartient au groupe `COMPTABLE`, permet à tous ceux du groupe `COMPTABLE` de lire l'objet externe, et autorise `Paul` à lire et écrire dans l'objet. Notons que, dans ce système, la nature des accès contrôlés sont plus étendus que ceux vus jusqu'à maintenant. Certains sont standards quelque soit le type de l'objet, comme par exemple le droit de supprimer l'objet, d'autres sont définis en même temps que la définition de ce type.
- Dans beaucoup de systèmes les utilisateurs sont rattachés à un numéro de compte qui sert à la facturation du coût machine. Dans l'entreprise, ce numéro de compte désigne alors un service ou un département. La définition des droits d'accès se fait alors en donnant la liste des numéros de compte autorisés à lire et ceux autorisés à écrire.

### 10.1.2. La protection par mot de passe

Le deuxième mécanisme de protection est lié à l'utilisation de mots de passe. Lorsqu'un programme demande d'établir un lien avec un objet externe, le système recherche, dans une première étape, l'objet lui-même. S'il le trouve, et que son descripteur indique la présence d'un mot de passe, le système interrompt provisoirement l'opération d'ouverture, avec un code d'erreur particulier. Le programme peut récupérer cette erreur et exécuter une séquence d'instructions qui a pour but de fournir le mot de passe. Au retour de cette séquence d'instructions, le système contrôle la validité de ce mot de passe, et termine l'ouverture correspondante. Si le programme n'a pas prévu de récupérer cette erreur, ou s'il l'a récupérée mais n'a pas fourni le bon mot de passe, la liaison n'est pas établie, et le programme arrêté. Un mécanisme analogue est mis en place lors de la création d'un objet avec mot de passe. De cette façon, le mot de passe n'apparaît pas dans le langage de commande qui définit la liaison, mais est toujours fourni par le programme lui-même.

Notons que la protection par mot de passe n'est efficace que si les mots de passe ne sont pas divulgués, et ne peuvent être découverts qu'avec grande difficulté.

## 10.2. Les mécanismes de sécurité

Le problème de la sécurité est en grande partie lié aux défaillances du matériel ou du logiciel. Il est plus ou moins bien résolu par deux méthodes, la redondance d'informations et la sauvegarde périodique. Notons que la sauvegarde périodique est également une certaine forme de redondance.

## 10.2.1. La sécurité par redondance interne

La *redondance interne* consiste à structurer les données de telle sorte que toute information de structure puisse être déterminée de deux façons au moins, l'une étant l'information dite *primaire*, l'autre étant l'information *secondaire*. L'information primaire est l'information de base, l'information secondaire est souvent utilisée pour accélérer les accès. On vérifie en permanence la cohérence entre les informations primaires et les informations secondaires. En cas d'incohérence, ou de perte des informations secondaires on utilise les informations primaires pour reconstruire les informations secondaires.

On peut imaginer par exemple que chaque secteur (ou éventuellement bloc) du disque comporte deux parties, l'*en-tête* et le *contenu*. L'en-tête contient les informations primaires permettant de savoir à quel objet appartient ce secteur, quelle est sa position relative dans l'objet (numéro logique) et quelle est sa position absolue sur le disque. Le contenu est la partie de l'objet externe qui est mémorisée dans ce secteur. Les informations primaires ne sont pas utilisées pour localiser les objets ou les parties d'objets, mais pour contrôler que le secteur accédé est bien celui qui est recherché, c'est-à-dire, qu'il appartient bien à l'objet, qu'il correspond bien à la position physique demandée sur le disque et à la position relative demandée dans l'objet. L'objet lui-même peut être complété par un en-tête mémorisé dans un secteur dont le contenu contient la désignation symbolique absolue de l'objet ainsi que des informations complémentaires situées dans le descripteur (protection, identification du propriétaire, nature, etc...). Cet en-tête est également une information primaire. Les informations secondaires sont situées dans les divers répertoires et les descripteurs des objets, qui sont utilisés normalement pour les accès efficaces. En cas d'anomalie, on voit qu'il est possible (mais long) de parcourir tous les secteurs du disque pour accéder aux informations primaires, et reconstruire les informations secondaires correspondantes.

Un mécanisme de ce type a été mis en œuvre dans l'Alto et le système Pilot de Xerox. Certains contrôleurs de disques implantent une partie de cette méthode, en mémorisant dans l'en-tête des secteurs, lors du formatage, le numéro de disque, cylindre, face et secteur, et vérifient la concordance lors de la lecture ou l'écriture d'un secteur.

Cette méthode est en fait assez lourde, et occupe de la place. Aussi beaucoup de systèmes ne l'appliquent que partiellement. Par exemple, MS-DOS dispose de deux exemplaires de la FAT sur le disque. Un seul exemplaire est lu en mémoire centrale lorsque nécessaire, la recopie sur disque étant faite systématiquement dans les deux exemplaires. S'il s'avérait que le système soit incapable de relire le premier exemplaire, il accèderait alors au second, et le recopierait immédiatement dans le premier. De même, dans la plupart des systèmes, la représentation de l'espace libre sur disque est une information secondaire. Elle peut être reconstruite en recherchant les secteurs qui ne sont pas actuellement alloués à un objet externe.

## 10.2.2. La sécurité par sauvegarde périodique

La redondance peut être obtenue par sauvegarde périodique. Il s'agit alors de prendre une copie exacte de l'ensemble des objets externes et de leur structure. Cependant, l'important est plus de pouvoir reconstruire la structure que de pouvoir utiliser la copie. Il s'ensuit que la sauvegarde peut se faire sur un support moins coûteux, comme par exemple sur bandes magnétiques. Si l'on sauvegarde la totalité des objets, on dit que l'on a une *sauvegarde complète*. Il faut noter cependant que, avec un débit de 500 Ko/s, il faut 35 minutes et 7 bandes pour sauvegarder 1 Go. C'est pourquoi, on ne peut la faire trop souvent.

On évite cette difficulté tout d'abord en faisant la sauvegarde par volume. Pour avoir des volumes dont la taille soit raisonnable pour ces sauvegardes, certains systèmes (MS-DOS ou Unix) permettent de *partitionner* un disque physique en plusieurs volumes indépendants. La notion d'arborescence unique telle qu'on la trouve dans Unix permet de cacher ce partitionnement à l'utilisateur, tous les volumes d'un disque donné étant montés en même temps par l'opérateur. Il est alors judicieux de faire en sorte que les objets soient placés sur les volumes en fonction de leur plus ou moins grande évolution dans le temps. Par exemple, on peut dédier un volume à l'ensemble des objets externes des utilisateurs, ou un volume par groupe d'utilisateurs; un autre volume peut être réservé à la documentation en ligne du système, un volume à l'ensemble des outils courants de la chaîne de production de programmes, etc... La sauvegarde périodique ne doit alors concerner que

les volumes des utilisateurs, les autres n'étant pas modifiés pendant de longues périodes. On parle parfois de *sauvegarde de reprise* dans ce cas.

L'inconvénient de la sauvegarde par volume telle qu'elle vient d'être présentée est de sauvegarder systématiquement les objets du volume, même s'ils n'ont pas été modifiés depuis la dernière sauvegarde. On peut éviter cet inconvénient si pour chaque objet on connaît la date de sa dernière modification: il suffit de ne sauvegarder que ceux dont cette date est postérieure à la dernière sauvegarde. C'est ce que l'on appelle la *sauvegarde incrémentale*. Périodiquement, un programme spécial est activé, qui parcourt l'ensemble des objets de l'arborescence unique (ou par volume), et recopie sur bande ceux qui ont été modifiés depuis la dernière sauvegarde périodique. En général, on ne recopie que les objets qui ne sont pas ouverts en modifications, pour garantir que la copie soit intrinsèquement cohérente.

Lorsqu'on constate une incohérence ou une perte d'objets, il faut restituer leur état le plus récent à partir des sauvegardes. Ceci est obtenu en remontant les sauvegardes incrémentales successives, jusqu'à trouver l'objet concerné. Pour éviter de devoir remonter trop loin dans le temps, on effectue de temps en temps des sauvegardes de reprise, qui permettent d'arrêter cette recherche. Par ailleurs, lorsque la totalité du volume est perdu, on utilise alors la dernière sauvegarde de reprise de ce volume pour restaurer le volume dans un état initial, sur lequel on applique ensuite les modifications constatées dans les sauvegardes incrémentales postérieures, dans l'ordre des dates. Pour faciliter l'accès aux sauvegardes incrémentales, celles-ci sont parfois réalisées sur un disque dédié. Comme, en principe, la proportion des objets modifiés est faible par rapport à l'ensemble des objets, cela ne nécessite pas de doubler l'espace disque total de l'installation.

La périodicité et la durée de conservation des sauvegardes dépendent de leur nature. Le tableau suivant donne une idée de la valeur de ces paramètres.

| nature de la sauvegarde | période | conservation |
|---|---|---|
| incrémentale | 8 heures | 15 jours |
| reprise | 7 jours | 3 mois |
| complète | 1 mois | 1 an |

Notons que cela implique de conserver au total 45 sauvegardes incrémentales, 13 sauvegardes de reprise et 12 sauvegardes complètes. Si on suppose qu'une sauvegarde incrémentale représente 5% de l'ensemble des objets, cela ne représente qu'un peu plus de 2 sauvegardes complètes. La conservation représente donc, en capacité, 27 fois l'espace total. La sauvegarde complète doit être exécutée sans qu'aucun accès aux objets ne soit autorisé, de façon à représenter une véritable photographie de l'état de l'ensemble des objets. Cela prendra donc 6 heures pour 10 Go. Les sauvegardes de reprise ne doivent pas être effectuées pour l'ensemble des objets, mais par volume. On peut donc les répartir sur tous les jours de la semaine, en sauvegardant par exemple 1,4 Go par jour, ce qui prendra environ 50 minutes. Lors de la sauvegarde de reprise d'un volume, ce volume ne doit pas être modifié par les utilisateurs. Enfin les sauvegardes incrémentales porteront sur 500 Mo, et prendront environ 17 minutes. Notons que celle-ci est faite sans gêne pour les utilisateurs, l'ensemble des objets restant accessibles.

L'ensemble de la procédure peut paraître coûteuse, mais c'est le prix à payer pour offrir un minimum de service continu fiable aux utilisateurs. Par ailleurs, les procédures décrites n'offrent qu'une sécurité limitée, puisque la sauvegarde incrémentale n'est effectuée que toutes les 8 heures dans notre proposition. Si un incident survient juste à la fin de la période, le travail effectué dans la période est perdu. Dans un système temps partagé, le risque étant faible est considéré comme supportable. Dans un contexte transactionnel, ce n'est souvent pas acceptable. Les mécanismes mis en œuvre sont alors dédiés à la base de données, mais sont assez voisins dans le principe: un fichier journal mémorise les modifications successives et joue le rôle de sauvegarde incrémentale permanente.

### 10.2.3. Les disques à tolérance de panne

La capacité des disques est devenue telle que la sauvegarde des informations peut prendre un temps considérable. D'un autre côté, leur prix n'est plus un critère économique important. On a donc imaginé des solutions qui permettent de gérer une certaine redondance des données pour diminuer les risques de perte entre deux sauvegardes espacées. C'est ce que l'on appelle les disques RAID

pour *Redundant Array Inexpensive Disks*. 5 types de RAID (1..5) ont été décrits initialement, et 2 sont effectivement utilisés (1 et 5).

Avant de les présenter, rappelons que nous avons vu en 9.2.3 qu'un volume pouvait être constitué de plusieurs partitions situées sur des disques différents. Lorsque ces partitions sont de même taille, il est possible de les découper en bandes de tailles égales, chaque partition contenant le même nombre de bandes. On peut alors entrelacer les bandes sur les différentes partitions comme indiqué sur la figure 10.1: le volume commence sur la bande 1, se poursuit sur la bande 2, puis la bande 3, etc. Notons que, en dehors de tout aspect de tolérance aux pannes, cette organisation répartit les accès sur l'ensemble des partitions et améliore les performances. Ceci est parfois appelé RAID-0.

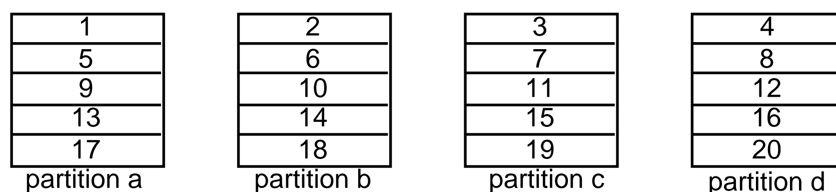



***Fig. 10.1***. *Entrelacement des bandes sur quatre partitions.*

Les deux niveaux de RAID intéressants sont les suivants:

- RAID-1, ou encore *disques miroirs*. La redondance est obtenue en écrivant les données sur 2 disques ou plus. En fait deux partitions sont nécessaires, par exemple les partitions a et b, qui sont des copies l'une de l'autre. Lorsque la bande 1 est écrite, elle est dupliquée sur la bande 2, et ainsi de suite. Les lectures, par contre, peuvent être faites sur n'importe laquelle des partitions. En cas de panne sur l'un des deux disques, les données sont encore accessibles sur l'autre, sans aucune perte. On peut constater que la moitié de l'espace disque est occupé par la redondance.
- RAID-5, ou encore *disques avec parité*. En reprenant la figure 10.1, pour chaque ligne horizontale, l'une des bandes est la bande de parité des trois autres ("ou exclusif", noté XOR); ainsi b4 = b1 XOR b2 XOR b3. Lors d'une écriture de b1, b2 ou b3, le système calcule la bande de parité b4 et l'écrit dans la partition d. Notons qu'il n'est pas nécessaire de connaître toutes les bandes pour calculer b4: lorsqu'on change b1 en b1', la nouvelle valeur de la bande b4, est obtenue par b4' = b4 XOR b1 XOR b1'. Pour que le disque de la partition d ne soit pas surchargé, on répartit les bandes de parités sur l'ensemble des partitions, par exemple la suivante sera b5, puis b10, puis b15, etc. En cas de panne sur le disque de la partition a, on peut reconstruire la bande b1 = b2 XOR b3 XOR b4. Notons que, si on utilise n disque, 1/n de l'espace est occupé par la redondance.

Evidemment, au moment d'une panne, on passe à un fonctionnement sans redondance. Cela peut être une panne locale, par exemple dans le cas d'un bloc défectueux sur le disque, et il suffit d'allouer un bloc de remplacement et de reconstruire son contenu. Cela peut être une panne du disque complet qu'il faut alors remplacer, et reconstruire ensuite les données sur ce nouveau disque.

On peut voir que dans chaque cas, on augmente le nombre d'écriture à effectuer sur disque. Cependant, ces écritures étant sur des disques distincts, peuvent être exécutées en parallèle, si le système le permet. Par ailleurs, les systèmes pratiquent de plus en plus l'*écriture paresseuse*: une demande d'écriture par un programme d'applications entraîne la modification de tampons en mémoire centrale et la reprise de l'exécution du programme. L'écriture effective sur le disque est différée à un moment ultérieur.

Notons que ces techniques peuvent être réalisées dans le contrôleur des disques, l'ensemble étant vu du système comme un seul disque. De plus, il est parfois possible de remplacer un disque de l'ensemble sans arrêter le système.

# 10.3. Conclusion

☞ La protection a pour but de se prémunir contre les altérations des données dûes à une mauvaise utilisation des opérations.

☞ La protection par droit d'accès permet de définir quels sont les utilisateurs qui sont autorisés à effectuer une opération donnée sur l'objet.

☞ La protection par mot de passe consiste à autoriser les accès si l'utilisateur fournit le bon mot de passe.

☞ La sécurité a pour but de se prémunir contre les altérations des données par suite d'une défaillance du matériel ou du logiciel.

☞ La sécurité par redondance interne consiste à disposer des informations de structure de plusieurs façons, et de contrôler la cohérence entre elles.

☞ La sécurité par sauvegarde périodique consiste à recopier régulièrement tout ou partie des objets externes sur un autre support, pour permettre de les restituer en cas d'incident.

☞ Un système RAID est un ensemble de disques où l'un d'eux contient des informations redondantes de celles contenues dans les autres, et qui permet de ne perdre aucune information en cas de panne de l'un des disques de l'ensemble.

# 11

# Quelques exemples de SGF

Pour terminer cette partie, nous allons présenter brièvement quelques systèmes de gestion de fichiers (SGF), dont nous avons déjà abordés certains aspects à titre d'exemple particuliers. Nous décrirons surtout la structure, sans entrer dans le détail des algorithmes utilisés par les systèmes, et qui peuvent évoluer d'une version à l'autre.

## 11.1. Les systèmes FAT et VFAT

VFAT est une extension de FAT dont nous avons déjà parlé. Ces SGF sont capables de gérer des partitions dont la taille est limitée à 2 Go. Cet espace est constitué de 3 parties:

- la FAT proprement dite, éventuellement dupliquée,
- le répertoire racine du volume,
- les données des objets externes du volume.

### 11.1.1. Représentation de l'espace

L'allocation d'espace se fait par bloc de taille fixe, appelés *cluster*, qui est constitué d'un nombre entier de secteurs (au plus 255). Les clusters ont un numéro sur 16 bits (au plus 65 535). La FAT a un double rôle:

- déterminer les clusters occupés, ceux qui sont libres et ceux qui sont invalides et pour lesquels il y a eu des erreurs de lecture auparavant,
- déterminer pour chaque cluster alloué à un objet externe quel est le cluster suivant de cet objet.

L'allocation d'espace se fait bloc par bloc, par un parcours séquentiel de la FAT. Lors d'un parcours séquentiel d'un fichier logique, le système suivra la liste chaînée des clusters. Lors d'un accès aléatoire, il devra également effectuer ce parcours depuis le début de l'objet externe jusqu'au cluster recherché.

### 11.1.2. Les répertoires

Ce système utilise une arborescence de répertoires, la racine de cette arborescence étant située derrière la FAT. Notons que cette racine peut contenir 512 entrées, alors que les autres répertoires peuvent en contenir un nombre quelconque. Une entrée d'un répertoire contient 32 octets et est constituée comme suit:

- Le nom du fichier sous la forme dite "8.3", c'est-à-dire, 8 caractères avant le point et 3 après. Ces noms doivent commencer par une lettre et sont insensibles à la casse (pas de différenciation entre les majuscules et les minuscules).

- Un ensemble d'attributs précisant, entre autre, la nature de l'objet externe (fichier ou répertoire) et sa protection (écriture autorisée).
- L'heure et la date de dernière modification de l'objet externe.
- Le numéro du premier cluster de l'objet externe.
- La longueur utile en octet de l'objet externe sur 32 bits.
- De plus,10 octets sont inutilisés.

Un répertoire contient toujours deux entrées particulières: "." qui désigne le répertoire lui-même et ".." qui désigne le répertoire parent.

Notons ici une extension apportée par le système VFAT, pour permettre les noms longs. Lors de la création d'un nom de fichier, qui peut être sur 255 caractères, le système crée en fait 2 noms: l'un est le nom origine, l'autre est un alias sous la forme DOS 8.3 construit à partir du nom long. Le nom DOS occupe une entrée du répertoire, et les entrées suivantes contiennent les tranches successives de 13 caractères du nom long. Un tel nom long peut donc occuper jusqu'à 21 entrées! Comme le répertoire racine est limité à 512 entrées, ceci a pour conséquence de limiter à 24 objets externes ce répertoire si on utilise des noms de 255 caractères à ce niveau.

Notons que si le système FAT (et VFAT) a été créé par Microsoft pour le DOS, il est maintenant compatible avec un grand nombre de systèmes. Par ailleurs, on peut noter qu'il permet une assez bonne occupation de l'espace et devrait donc être réservé à de petites partitions.

# 11.2. Les systèmes HFS et HFS Plus de MacOS

Le système HFS Plus est une nouvelle version de SGF, introduite par Apple pour pallier certaines limites de HFS, tout  en reprenant les mêmes concepts. Nous allons décrire HFS, et montrerons les évolutions. Une partition HFS (*Hierarchical File System*) est constituée de 6 parties:

- Les informations de démarrage du système permettent, par exemple, de localiser le système dans la partition, ou de connaître la taille des tampons internes, etc. Cette partie contient des zéros si la partition n'est pas une partition de démarrage.
- Le descripteur du volume. On y trouve, en particulier, le nom du volume (limité à 27 caractères), ainsi que des informations diverses sur la structure du volume.
- La table bitmap décrivant l'état d'allocation du volume.
- Le catalogue des fichiers et répertoires.
- Le fichier de description des extensions.
- Les espace des objets externes du volume.

Les objets externes (fichiers et répertoires) créés sur le volume reçoivent un numéro interne unique d'identification que nous appèlerons Num-ID. Ce numéro est défini à la création du fichier ou du répertoire.

## 11.2.1. La représentation de l'espace.

HFS utilise le principe de l'allocation par zone, un objet externe pouvant avoir un nombre quelconque de zones chacune étant de taille quelconque (cf. 8.2.3). La taille du quantum est un nombre entier de secteurs. Nous utiliserons le terme de bloc pour désigner les unités d'allocation. Les blocs sont numérotés sur 16 bits, limitant à 65536 le nombre de ces blocs. Ceci a deux conséquences:

- La table bitmap évoquée ci-dessus est de taille fixe d'au plus 8 Ko.
- La taille d'un quantum est égal au moins à la taille de la partition divisée par 65536, soit 64 Ko pour une partition de 4 Go.

Nous verrons ci-dessous que les répertoires n'ont pas d'espace qui leur soit alloué en propre. Par contre, tout fichier contient deux parties distinctes: la partie ressource et la partie données, chacune d'elles étant limitée en taille à 2 Go. Cette séparation est souvent utilisée par les applications pour isoler la partie présentation de la partie contenu. Ce qui nous importe ici est que chacune possède

son propre espace alloué. Notons que, en général, la partie ressource est assez petite, mais, si elle n'est pas vide, elle occupe au moins un bloc.

Le descripteur de fichier contient la description des trois premières zones allouées à chacune des parties du fichier, chaque zone étant décrite par le numéro du premier bloc et le nombre de blocs. Si une partie a besoin de plus de trois zones, les descriptions des zones manquantes sont regroupées, par groupe de 3, dans le fichier de description des extensions vu plus haut.

Chaque entrée de ce fichier est constituée des champs suivants, les trois premiers composant la clé d'identification de l'entrée:

- Num-ID du fichier.
- Indicateur précisant si on a affaire à la partie ressource ou à la partie données.
- Numéro logique à l'intérieur du fichier correspondant à la première zone. Ce numéro est en fait égal à la somme des longueurs des zones allouées qui précèdent la première du groupe.
- Description des trois zones allouées.

Pour faciliter les accès, les entrées sont organisées en arbre B*. Nous n'entrerons pas ici dans le détail d'une telle structure, mais nous suggérons le lecteur de voir un cours de structures de données. Brièvement, la figure 11.1 donne un exemple simple d'arbre B*, où les clés sont simplement des entiers. Dans notre cas, chaque nœud occupe un secteur.

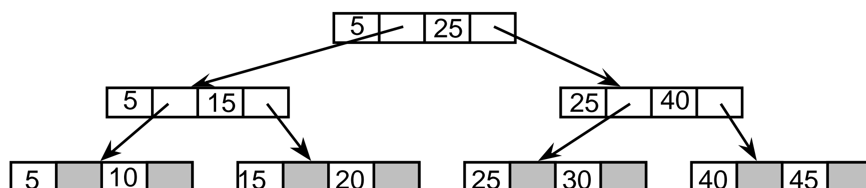



***Fig. 11.1.*** *Exemple d'arbre B* utilisé dans HFS.*

Notons que si un fichier est fragmenté, et occupe plus de 3 zones, le système devra rechercher dans l'arbre le ou les groupes de 3 zones nécessaires. Cependant, les entrées correspondantes sont en fait ordonnées dans le fichier de description des extensions, puisqu'il s'agit d'un arbre B*.

Le fichier de description des extensions est également utilisé pour conserver les blocs invalides, c'est-à-dire ceux pour lesquels on a détecté une erreur permanente. Ces blocs sont affectés, comme zone d'extension, à un objet externe de Num-ID particulier qui n'est repéré par aucun répertoire. Ainsi, ces blocs sont considérés comme alloués, mais l'objet auquel ils sont alloués n'est pas accessible.

## 11.2.2. Les répertoires

Ce système utilise une arborescence de répertoires, mais avec une représentation originale de cette arborescence, puisque l'ensemble est également représenté dans un arbre B* unique, que nous avons appelé ci-dessus le catalogue des fichiers et répertoires. À chaque objet externe, on associe une entrée de ce catalogue qui est constituée essentiellement des champs suivants, les deux premiers composant la clé d'identification de l'entrée:

- Num-ID de son répertoire parent.
- Nom de l'objet externe, limité à 31 caractères quelconque à l'exception du ':', la casse n'intervenant pas dans les comparaisons.
- Le type de l'entrée, et donc de l'objet externe, fichier ou catalogue.
- Dates de création, modification et sauvegarde.
- Num-ID de l'objet lui-même.
- Descripteur de fichier dans le cas d'un fichier. En particulier, pour chacune des parties du fichier, la taille physique (allouée), sa taille logique (utile) et les descripteurs des trois premières zones allouées à cette partie.

Par ailleurs, le catalogue contient des entrées spéciales qui permettent de remonter dans l'arborescence depuis un répertoire à son parent, et que nous appellerons un lien. Une telle entrée est constituée comme suit, les deux premiers champs étant toujours la clé de l'entrée:

- Num-ID du répertoire
- "", ou chaîne vide.
- Le type de l'entrée en tant que lien.
- Num-ID de son parent.
- Nom du répertoire lui-même.

Par ailleurs, le catalogue, en tant que arbre B*, a toutes ses entrées ordonnées par clés croissantes. Par conséquent, le lien d'un répertoire vers son parent est immédiatement suivi des entrées correspondant aux objets externes qu'il contient.

## 11.2.3. Cohérence du SGF

Comme dans tout système de gestion de fichier, les informations sur la structure sont très importantes et doivent être cohérentes entre elles. Ainsi, la table bitmap ne doit pas indiquer comme libre un bloc qui serait alloué à un fichier. Conformément à ce que nous avons dit en 10.2.1, lorsqu'il y a redondance d'informations, on peut distinguer les informations primaires et les informations secondaires, celles-ci ayant pour but de donner des accès efficaces. Dans le cas des informations sur l'état d'allocation des blocs, les informations primaires sont les descriptions des zones allouées, et la table bitmap constitue les informations secondaires permettant d'accélérer les recherches de blocs libres.

De même, pour pouvoir attribuer un Num-ID aux objets lors de la création, il faut connaître le dernier attribué. Cette information secondaire est mémorisée dans le descripteur du volume. L'information primaire est représentée en fait par la plus grande valeur des Num-ID existant sur le volume.

Pour garantir cette cohérence, HFS maintient un indicateur dans le descripteur de volume, qui indique que le volume n'a pas été démonté proprement. Cet indicateur est écrit sur disque lors du montage en écriture du volume, et remis à zéro lors du démontage du volume. Il permet donc de savoir lors d'un montage si une vérification de cohérence est nécessaire.

## 11.2.4. Le système HFS Plus

Ce système a pour but de lever les contraintes rencontrées dans HFS. Nous ne mentionnerons ici que les modifications les plus importantes pour l'usager:

- La plus importante que nous ayons vu est la limitation du nombre de quanta d'une partition, impliquant d'avoir une taille de quantum importante pour les grosses partitions. Cette contrainte est levée en prenant 32 bits pour les numéros de blocs d'allocation. Évidemment cela a pour conséquence d'augmenter notablement la table bitmap, qui devient un fichier, permettant d'homogénéiser les accès.
- La taille maximale de chacune des parties d'un fichier est elle aussi augmentée, puisqu'elle est portée à $2^{63}$ octets.
- Comme les blocs deviennent plus petits et peuvent conduire à une fragmentation plus importante, et donc un nombre plus grand de zones, les groupes ont été portés à 8, tant dans le descripteur de fichier que dans les entrées du fichier de description des extensions.
- Les noms des objets ne sont plus limités à 31 caractères MacRoman (codage sur 8 bits), puisqu'un nom peut avoir maintenant 255 caractères Unicode (16 bits). Ceci a conduit à augmenter la taille des nœuds de l'arbre B* du catalogue (4 Ko par défaut), car plus ces nœuds contiennent de valeurs, plus la hauteur de l'arbre est faible.
- Un troisième fichier de volume organisé en arbre B* a été introduit pour permettre à terme de gérer plusieurs attributs d'un fichier, mais il s'agit plus d'ouverture vers l'avenir et d'éviter ainsi d'avoir à redéfinir un nouveau SGF, car les spécifications complètes ne sont pas encore complètement définies en mars 1999. Ce fichier devrait permettre, par exemple, d'avoir plus de deux parties pour un même fichier, et de pouvoir nommer ces parties.

# 11.3. Le système NTFS

Ce système a été créé par Microsoft pour être le système de gestion de fichier de Windows NT. Plusieurs systèmes existaient initialement, comme VFAT ou HPFS de OS/2, mais aucun ne satisfaisait aux critères de fiabilité et de taille d'un système moderne.

En dehors du premier secteur de la partition, qui contient un ensemble minimal d'informations comme la localisation du fichier MFT dont il est question ci-dessous, ce SGF considère que "tout est fichier". Le formatage d'une partition en volume NTFS consiste donc à créer un certain nombre de fichiers qui donnent la structure du volume. Nous ne décrirons que certains de ces fichiers.

- Le fichier des descripteurs de fichiers, appelé la *Master File Table* (MFT), qui commence par son propre descripteur. Il peut commencer n'importe où dans la partition, il est nécessaire de connaître cet endroit a priori, pour pouvoir accéder à son descripteur, ce qui est indiqué dans le premier secteur de la partition. Notons que ce fichier est doublé pour des raisons de sécurité, la copie étant aussi repérée par le secteur 0 de la partition.
- Le fichier du volume, contenant en particulier le nom du volume.
- Le fichier bitmap décrivant l'état d'allocation du volume.
- Le répertoire racine du volume.
- Le fichier journal qui a pour but de garantir la fiabilité de la structure.

## 11.3.1. Les descripteurs de fichiers

Chaque objet externe reçoit, à sa création, un numéro qui est l'indice dans la MFT où est situé son descripteur. La taille de ces descripteurs est fixée à la création du volume et est comprise entre 1 Ko et 4 Ko, ce qui est donc relativement important. Un objet externe, fichier ou répertoire, est constitué d'un certain nombre d'attributs qui peuvent être résidents, c'est-à-dire, rangés dans le descripteur de l'objet externe lui-même, ou non résident et donc à l'extérieur de la MFT dans un espace qui lui est alloué en propre. Voici quelques exemples d'attributs :

- Le nom de l'objet externe sous forme d'une suite de au plus 255 caractères Unicode.
- Les informations de base habituelles, comme les dates de création, modification ou d'accès.
- Les informations de protections de l'objet.
- Le ou les contenus des fichiers ou répertoires.

Un même objet externe peut avoir plusieurs attributs "nom", correspondant au principe des liens physiques énoncés au §9.4.3.

Si un fichier est suffisamment petit, tous ses attributs, et donc son contenu se trouvera dans le descripteur. Dans les autres cas, certains attributs (en particulier les données) seront non résidents.

NTFS utilise le principe de l'allocation par zone pour les attributs non résidents d'un objet externe , et la valeur de l'attribut est remplacée dans le descripteur par la suite des descriptions des zones allouées. Le nombre de zones est quelconque, chacune étant de taille quelconque (cf. 8.2.3). La taille du quantum est un nombre entier de secteurs, ce nombre étant une puissance de 2. Le terme de cluster désigne les unités d'allocation. Les clusters sont numérotés sur 64 bits, ce qui implique qu'il n'y a pratiquement pas de limite aux nombres de clusters d'un volume. Comme un descripteur de zone doit localiser le premier cluster de la zone et son nombre de clusters, soit 16 octets, une technique de compression est utilisée, d'une part en prenant son déplacement par rapport au premier cluster de la zone précédente, et en supprimant les octets nuls en tête de ces deux valeurs.

La longueur d'un attribut non résident est mémorisée sur 64 bits, ce qui permet donc des fichiers dont la taille n'est pratiquement limitée que par l'espace disponible sur le volume.

Notons que le contenu d'un fichier est un attribut sans nom, mais l'utilisateur peut créer des attributs nommés qui peuvent avoir des contenus séparés dans des espaces distincts. L'implantation de serveur de fichier MacIntosh sur Windows NT utilise ce principe pour séparer la partie ressource et la partie donnée (voir HFS).

## 11.3.2. Les Répertoires

NTFS utilise une arborescence de répertoire. Le contenu d'un répertoire est organisé en arbre B+, qui se rapprochent des arbres B* vus plus haut, si ce n'est que les données associées aux clés sont réparties dans tous les nœuds au lieu de n'être que dans les feuilles comme sur la figure 11.1. Les entrées d'un répertoire contiennent les informations suivantes, la première étant la clé:

- Le nom de l'objet,
- Le numéro de l'objet dans la MFT, permettant de localiser son descripteur.
- Les dates de création, modification ou d'accès de l'objet,
- La taille de l'objet
- Le numéro du répertoire parent qui le contient dans la MFT.

En fait seuls les deux premiers sont effectivement nécessaires, puisque les trois derniers se trouvent dans le descripteur de l'objet. Cette duplication a un avantage et un inconvénient. L'avantage est que l'on a accès aux informations essentielles de l'objet sans devoir accéder au descripteur. L'inconvénient est inhérent à la duplication: toute modification de ces informations doit être portée à plusieurs endroits sur le disque, sous peine d'avoir des données incohérentes. Notons qu'il peut paraître surprenant de trouver dans cette entrée le numéro du répertoire qui contient l'entrée! En fait, Toutes ces informations, avec la clé, font partie l'attribut "nom de fichier", présent dans le descripteur du fichier, et en sont une copie. Or, le numéro du répertoire parent d'un fichier permet de remonter l'arborescence des fichiers et répertoires, en retrouvant ainsi le parent de chaque répertoire.

Le principe des arbres B implique que des nœuds soient créés ou supprimés au fur et à mesure des adjonctions ou suppressions. La suppression d'un nœud peut ne pas être le dernier nœud physique du répertoire, et il faut prévoir sa réutilisation ultérieure. Ceci se fait en définissant un attribut "bitmap" pour chaque répertoire qui indique quels sont les nœuds libres. S'il n'y en a plus, une allocation d'une nouvelle zone est effectuée, et la table bitmap est mise à jour pour tenir compte des nœuds libres ajoutés.

## 11.3.3. Compression de données

NTFS propose une technique de compression/décompression des fichiers au fur et à mesure des accès. L'idée est de découper le contenu du fichier par tranche dont la taille correspond à 16 clusters, et de tenter de compresser chaque tranche indépendamment les unes des autres. La compression n'est effective que si elle fait gagner au moins 1 cluster. Dans ce cas, les clusters gagnés ainsi sont mémorisé dans la suite des descripteurs de zones par une zone marquée non allouée dont la taille correspond au nombre de clusters gagnés. A la lecture, l'opération inverse est effectuée. Notons que la mise en œuvre de la compression/décompression sur un fichier est mémorisé dans le descripteur du fichier, l'utilisateur n'ayant pas à s'en préoccuper ensuite. Évidemment cette technique est coûteuse en temps processeur, et doit être utilisée à bon escient.

## 11.3.4. Sécurité par fichier journal

Les opérations mises en œuvre sur les structures de fichiers, en général, concernent plusieurs secteurs distincts répartis sur le volume. Les modifications ne peuvent toutes être portées sur le disque en une seule fois, mais doivent être effectuées en séquence. Évidemment, les systèmes tiennent compte de l'éventualité d'une panne qui empêche le déroulement complet de la séquence et entraîne une incohérence. Ceci est obtenu en définissant pour chaque opération l'ordre qui donnera le minimum de dégât en cas de panne. En particulier, il est préférable de perdre de l'espace disque plutôt que de risquer d'allouer deux fois le même bloc à deux objets différents. En fait, il est presque toujours possible de rétablir la cohérence et retrouver les blocs non alloués, mais cela peut être coûteux en temps.

Dans le cas des SGF modernes, le risque est aggravé par le fait que bien souvent les opérations sont effectuées dans des tampons en mémoire, et que l'écriture de ces tampons sur disque est effectuée plus tard, pour gagner en performance et en efficacité (principe de l'écriture paresseuse). Par exemple, une création de fichier donnera souvent lieu dans un avenir proche à une ou plusieurs

allocations d'espace disque. Le report des écritures de la table bitmap peut avoir pour conséquence qu'elle ne sera écrite qu'une seule fois lorsque tout sera terminé.

La structuration d'un volume en NTFS contient un fichier particulier, dit fichier journal, qui va avoir un rôle analogue au fichier journal des systèmes de gestion de bases de données classiques, dans lequel sera mémorisée la séquence des actions effectuées sur le disque dans le but de pouvoir soit les refaire soit les défaire après une panne. Cette utilisation fait appel à la notion de transaction, qui signifie, entre autre, que l'on garantit que toutes les opérations élémentaires de la transaction sont effectuées, même en cas de panne, ou que aucune n'est faite. Expliquons brièvement le déroulement des opérations lors de la création d'un fichier avec allocation d'espace. Les opérations sont les suivantes:

- Initialisation d'une transaction.
- Création d'un descripteur du fichier dans la MFT.
- Création de l'entrée dans le répertoire parent,
- Mise à l'état non libre des bits correspondants aux clusters alloués, dans la bitmap,
- Clôture de la transaction.

Les opérations 2, 3 et 4 sont faites dans des tampons en mémoire, et lors de la clôture de la transaction, il est probable que ces tampons n'ont pas encore été écrits sur disque. Chacune de ces opérations donne lieu à un enregistrement dans le fichier journal qui décrit comment "refaire" et comment "défaire" l'opération correspondante. En fait ces enregistrements sont mis dans des tampons en mémoire, en vue d'une écriture ultérieure, comme pour tout fichier. Cependant le gestionnaire de ces tampons fera en sorte que les tampons du fichier journal soient écrits sur disque avant l'écriture des autres tampons. De plus, périodiquement, l'état des transactions ainsi que la liste des tampons en mémoire non encore écrits sur disque sont enregistrés dans le fichier journal. En cas de panne une analyse du fichier journal permet de savoir les transactions clôturées dont les actions n'ont pas été portées physiquement sur disque et qui doivent donc être refaites, ou les transactions non clôturées dont les actions portées physiquement sur disque doivent être défaites.

# 11.4. Le système ext2fs de Linux

Linux est un système d'exploitation qui s'apparente fort à Unix, tout en étant un logiciel libre. Le système de fichier initial était dérivé de celui du système Minix, lui aussi dérivé d'Unix pour l'enseignement. Le SGF de Minix étant cependant trop contraint, un nouveau système de gestion de fichier a été écrit, ext2fs.

## 11.4.1. La représentation de l'espace

Comme d'habitude, l'ensemble du disque est découpé en blocs dont la taille est un multiple (puissance de 2) de la taille d'un secteur. Ext2fs pratique l'allocation par bloc de taille fixe, à plusieurs niveaux (cf. 8.3.2). Les descripteurs d'objets externes sont représentés dans ce qui est appelé un inœud, qui sont regroupés dans une table, l'indice dans la table, appelé le numéro du inœud (i est là pour integer), identifiant de façon unique cet objet. Une table bitmap décrit l'état d'allocation des inœuds, et une autre décrit l'état d'allocation des blocs. Pour des raisons de performances, en particulier sur les gros disques, ces tables sont morcelées et réparties dans la partition.

Une partition est découpée en groupes de même taille, chaque groupe comportant 6 parties:

- Le *super bloc*, qui contient les informations de structure du volume.
- La liste des descripteurs de groupe, qui localise sur le disque les informations essentielles de chaque groupe (localisation des tables).
- La table bitmap d'état d'allocation des blocs du groupe.
- La table bitmap d'état d'allocation des inœuds du groupe.
- La table des inœuds du groupe.
- Les blocs de données.

Le super bloc et la liste des descripteurs de groupe sont donc répétés au début de chaque groupe, pour des raisons de fiabilité. Lors du montage, on ne lit en fait que ceux du premier groupe. Cette séparation en groupes doit plus être vue comme une répartition des données sur le disque, dans le but d'améliorer les performances. La taille d'un groupe est d'ailleurs limitée, puisque chaque table bitmap d'un groupe doit tenir dans un seul bloc. Si la taille de la partition est importante, il y aura beaucoup de groupes.

Les groupes ont pour but de rapprocher dans un espace physique proche les informations qui sont reliées entre elles. Ainsi, lors d'une allocation d'un inœud, on cherchera de préférence dans le groupe du répertoire où il est référencé; de même, lors de l'allocation d'un bloc, on cherchera d'abord le bloc qui suit le dernier alloué à l'objet, puis dans son voisinage immédiat (±32), puis dans le même groupe et enfin dans les autres groupes.

Un descripteur d'objet externe contient évidemment le type de l'objet, des informations pour sa protection, sa longueur, les dates habituelles de création, modification, accès et destruction, ainsi que les informations de localisation du contenu, qui sont semblables à celles de la figure 8.4 (la seule différence est qu'il y a en général 12 blocs directs au lieu de 10). La taille d'un objet externe est limité à 2 Go, mais la taille d'une partition peut atteindre 4 To.

### 11.4.2. Les répertoires

Ext2fs gère une arborescence de fichiers. Le contenu d'un répertoire étant une liste d'entrées constituées de couples <inœud, nom>, le nom étant limité à 255 caractères. Les deux premières entrées d'un répertoire sont les entrées "." et ".." habituelles. Il est possible d'avoir plusieurs noms ou chemins d'accès pour le même inœud. Ceci a pour conséquence que la suppression d'une entrée dans un répertoire n'entraîne pas obligatoirement la suppression du inœud correspondant. Celle-ci ne sera effective que s'il n'existe plus d'entrée associée à ce inœud dans aucun répertoire. En dehors de la suppression d'une entrée dans un répertoire, qui n'affecte donc que cette entrée, toutes les autres opérations effectuées par l'un des chemins aura les mêmes répercussions par l'autre.

Ce système permet aussi la création de liens symboliques (cf. 9.4.3). Dans ce cas, il s'agit de créer un objet externe (avec inœud et contenu), du type lien symbolique, et dont le contenu sera le chemin d'accès à l'objet relié. Rappelons que dans ce cas la recherche de l'objet externe relié implique la réévaluation du chemin d'accès.

### 11.4.3. Sécurité

Comme beaucoup de systèmes, ext2fs est basé sur l'utilisation de caches mémoire des blocs disques, l'écriture des blocs modifiés étant reportée (écriture paresseuse). Cela peut avoir pour conséquence des incohérences dans la structure du volume. L'écriture d'un bloc est effective lorsque le tampon qu'il occupait est récupéré pour un autre bloc. De plus, cette écriture est forcée automatiquement à une période régulière. Comme tous les systèmes, il est cependant recommandé de toujours l'arrêter par la commande associée. Dans ext2fs, à chaque objet externe, est attaché un indicateur qui, lorsqu'il est positionné, demande que le inœud et les blocs indirects sur disque soient synchrones avec leur contenu en mémoire, c'est-à-dire, que toute modification en mémoire soit portée immédiatement sur le disque, avant de retourner au programme d'application. Pour l'utilisateur, la fin d'exécution de l'opération signifie pour lui un état cohérent de la structure disque, en regard de son fichier.

Notons deux attributs définis par ext2fs et attachés à chaque fichier :

- L'attribut de "secret" implique que, lors de la destruction du fichier, le contenu doive être effacé en le remplissant de données aléatoires. Cela évite que des données sensibles puissent être vues lors de la réallocation des blocs qui les contenaient.
- L'attribut de "récupération" permet la récupération d'un fichier détruit, jusqu'à un certain point.

## 11.5. Structuration en couche du système

Certains systèmes permettent de manipuler, au même moment, des volumes structurés différemment. C'est assez naturel lorsque l'un des SGF est une extension de l'autre par le même concepteur, comme par exemple HFS et HFS Plus. La caractéristique de standard de fait de FAT

fait que pratiquement tous les systèmes implantent ce SGF en plus de leur propre SGF propriétaire. Windows NT reconnaît et gère FAT, VFAT, HPFS (de OS/2) et NTFS. Il ne reconnaît pas HFS, mais est capable de simuler un serveur de fichier de ce type sur un réseau, en s'appuyant sur NTFS. Linux permet la manipulation de tous les SGF évoqués dans ce chapitre, au moins en lecture, si ce n'est en écriture.

La reconnaissance simultanée de plusieurs SGF est en général basé sur un découpage du logiciel en couches, en définissant les spécifications entre les couches:

- Le système de fichier virtuel (SFV) est la vue homogène de tous les SGF réels. Il est défini en interne en terme de structures de représentations et d'opérations de manipulations de ces structures.
- Le système de tampons mémoire qui contiennent les blocs disque (le cache).
- Le disque abstrait est une vue homogène des tous les disques supportés par l'installation. Il définit un ensemble d'opérations d'accès disques élémentaires indépendantes des périphériques physiques eux-mêmes .

Les couches du logiciel sont les suivantes:

- La couche haute, proche de l'application, implante les opérations d'accès nécessaires aux programmes d'applications en voyant le système de fichier avec la structure de SFV.
- La couche intermédiaire, liée à un SGF, assure l'interface entre le SFV et les tampons mémoire. Elle assure la transformation entre les structures du SFV et du SGF particulier et implante les opérations du SFV, selon les caractéristiques du SGF, en se servant de ces tampons.
- Le gestionnaire de cache gère les tampons mémoire et lance les opérations nécessaires vers le disque abstrait.
- Les pilotes des disques, liés aux périphériques eux-mêmes, implantent les opérations du disque abstrait en terme d'opérations concrètes du disque.

# QUATRIÈME PARTIE

---

# ENVIRONNEMENT PHYSIQUE

# 12

# La gestion des processus

Nous avons vu dans les premiers chapitres que l'introduction des activités parallèles avait permis d'améliorer la rentabilité des machines. Le parallélisme peut être lié à des activités spécifiques, et donc obtenu par des processeurs spécialisés. Par exemple, un transfert d'entrées-sorties peut être exécuté en même temps qu'un calcul interne au processeur. Il peut aussi être le moyen d'obtenir plus de puissance de calcul dans le même temps imparti, en utilisant plusieurs processeurs de calcul.

Plus généralement, un processeur peut exécuter une séquence d'instructions d'un programme, puis ensuite une séquence d'instructions d'un autre programme, avant de revenir au premier, et ainsi de suite. Si la durée d'exécution des séquences est assez petite, les utilisateurs auront l'impression que les deux programmes s'exécutent en parallèle. On parle alors de *pseudo-parallélisme*. Évidemment, vu du processeur lui-même, il s'agit d'une activité unique et séquentielle, un programme de supervision décidant de la séquence à exécuter. C'est ce programme de supervision, inclus dans le système, qui donne en fait l'illusion du parallélisme. Si la recherche de la meilleure efficacité du matériel a été à l'origine du parallélisme et reste un aspect important, le pseudo-parallélisme permet à plusieurs utilisateurs d'utiliser une même machine au même instant, et à chacun d'eux de lancer plusieurs activités en même temps, comme par exemple compiler un module source pendant que l'on modifie un autre module source. La gestion de ces activités parallèles est en général assez complexe, et difficile à appréhender. Le concept de processus est fondamental pour la compréhension de cette gestion, tant pour le concepteur de système que pour l'utilisateur.

## 12.1. La notion de processus

Le terme de *programme* est souvent utilisé à des sens différents. Lorsqu'il est utilisé pour désigner l'ensemble des modules sources, l'ensemble des modules objets ou le résultat de l'édition de liens, il représente toujours la même abstraction, écrite dans différents langages de représentation, de la *description* des actions à entreprendre pour aboutir au résultat recherché. Cet aspect description des actions est en fait fondamental, et le terme de programme ne devrait être utilisé que dans ce sens.

Le terme de *processeur* est utilisé pour désigner l'entité (matérielle ou logicielle) qui est capable d'exécuter des instructions. Jusqu'à maintenant, nous l'avons utilisé essentiellement pour désigner une entité matérielle, comme le processeur de calcul, ou le processeur d'entrées-sorties, et nous continuerons à l'utiliser dans ce sens, mais il peut parfois être intéressant de considérer qu'un interpréteur, par exemple, est un processeur logiciel capable d'exécuter des instructions du langage qu'il interprète.

La notion de *processus* a été introduite pour désigner l'entité dynamique qui correspond à l'exécution d'une suite d'instructions. C'est un concept abstrait, en ce sens qu'il désigne l'évolution d'une activité dans le temps. C'est le processeur qui fait évoluer cette activité en exécutant les instructions définies par un programme. Le système doit permettre de distinguer les différents

processus qui existent à un instant donné, et doit représenter leur état. Celui-ci permet, d'abord, de localiser le programme lui-même, c'est-à-dire la suite d'instructions, ensuite, de déterminer les données propres et les variables, enfin de représenter l'état du processeur après l'exécution d'une instruction quelconque du programme (registres, compteur ordinal, mot d'état programme,...).

Pour bien montrer la différence entre programme et processus, nous reprendrons une image de Tanenbaum. Supposez qu'un informaticien décide de faire un gâteau pour l'anniversaire de sa fille. Pour être certain de le réussir, il ouvre devant lui son livre de recettes de cuisine à la page voulue. La recette lui précise les denrées dont il a besoin, ainsi que les instruments nécessaires. À ce stade, nous pouvons comparer la recette sur le livre de cuisine au programme, et l'informaticien au processeur. Les denrées sont les données d'entrées, le gâteau est la donnée de sortie, et les instruments sont les ressources nécessaires. Le processus est l'activité dynamique qui transforme les denrées en gâteau.

Supposez que, sur ces entrefaites, le fils de l'informaticien vient se plaindre qu'il a été piqué par une abeille. L'informaticien interrompt son travail, enregistre l'endroit où il en est, extrait de sa bibliothèque le livre de première urgence, et localise la description des soins à apporter dans ce cas. Lorsque ces soins sont donnés et que son fils est calmé, l'informaticien retourne à sa cuisine pour poursuivre l'exécution de sa recette.

On constate qu'il y a en fait deux processus distincts, avec deux programmes différents. Le processeur a partagé son temps entre les deux processus, mais ces deux processus sont indépendants.

Supposez maintenant que la fille de l'informaticien vienne lui annoncer de nouveaux invités. Devant le nombre, l'informaticien décide de faire un deuxième gâteau d'anniversaire identique au premier. Il va devoir partager son temps entre deux processus distincts qui correspondent néanmoins à la même recette, c'est-à-dire au même programme. Il est évidemment très important que le processeur considère qu'il y a deux processus indépendants, qui ont leur propre état, même si le programme est le même.

De même que l'informaticien aura un seul exemplaire de la recette pour les deux processus, de même dans une situation analogue en machine, le programme lui-même pourra se trouver en un seul exemplaire en mémoire, pourvu que d'une part le code ne se modifie pas lui-même, d'autre part qu'il permette d'accéder à des zones de données différentes suivant que les instructions sont exécutées pour le compte de l'un ou l'autre des processus. On dit alors que le programme est *réentrant*.

L'indépendance entre les processus conduit parfois à introduire la notion de *processeur virtuel*. Chaque processus se voit attribuer un tel processeur virtuel qu'il possède en propre, et qui exécute les instructions du programme pour le compte du processus. Le système implante ces processeurs virtuels en attribuant le (ou les) processeur réel alternativement aux différents processeurs virtuels. Chaque processus avance au rythme de son processeur virtuel, induisant un *temps virtuel*, qui correspond au temps d'utilisation du processeur virtuel. Il correspond aussi à la fraction du temps réel pendant laquelle un processeur réel a été attribué au processeur virtuel. En conséquence un temps virtuel de 1 seconde correspond à l'exécution d'un certain nombre d'instructions par le processeur réel, que ces instructions aient été réalisées en une seule fois, ou en plusieurs tranches de temps.

# 12.2. La hiérarchie de processus

Nous avons dit que la représentation d'un processus comportait trois composantes principales, le programme, les variables et l'état du processeur.

## 12.2.1. Un nombre fixe de processus banalisés

Certains systèmes créent, à l'initialisation, un nombre fixe de processus banalisés.

Ces processus peuvent être attachés chacun à un terminal particulier. Le programme du processus consiste alors en l'identification de l'usager (login) suivi de l'exécution de ses commandes et se termine par la remise dans l'état initial lors de la déconnexion (logout).

Ces processus peuvent être attachés à un terminal au moment de la demande de connexion depuis de terminal. Le programme est alors comparable au précédent. Le nombre de processus peut être inférieur au nombre de terminaux potentiels, entraînant un refus de connexion si aucun processus n'est disponible. Il est également possible d'allouer le processus à un couple de fichiers <entrée, sortie>, qui simulent le comportement d'un usager à son terminal, ce qui fournit une certaine forme de traitement par lot.

Ces processus peuvent être alloués à la demande pour une exécution d'une commande unique. Le programme, qui décrit leur comportement, est une boucle infinie qui consiste en l'attente d'une commande, suivie de son exécution. Suivant le cas, plusieurs processus pourront être alloués à un même usager en fonction de ses besoins. Cependant, il est nécessaire de disposer de plus de processus que dans les cas précédents, et la représentation de la partie propre de leur état occupe de la place en mémoire.

## 12.2.2. La création dynamique de processus

La création statique de processus banalisés présente l'inconvénient d'occuper de la place en mémoire, même lorsqu'ils ne sont pas utilisés. La deuxième méthode consiste à fournir une opération de création dynamique de processus ainsi qu'une opération de destruction de ces processus. Un telle opération de création doit permettre d'initialiser l'état du nouveau processus. Elle doit donc définir d'une part le programme (ou la suite d'instructions) décrivant l'activité du processus, ainsi que l'état initial de ses données, de ses variables et des registres du processeur. Elle doit donc se présenter sous la forme suivante:

```
id := créer_processus (programme, contexte)
```

où les paramètres définissent cet état initial. La valeur retournée éventuellement par cette opération permet d'identifier le processus qui a été ainsi créé. Cette dernière valeur n'a d'intérêt que si le processus créateur peut agir ultérieurement sur le processus créé au moyen d'autres opérations.

La plupart des systèmes qui permettent la création dynamique de processus considèrent que la relation entre le processus créateur et le processus créé est importante, ce qui conduit à structurer l'ensemble des processus sous la forme d'un arbre, et à maintenir cette structure. Lors de la création de processus, le processus créé est relié automatiquement comme fils du processus créateur. Si un processus est détruit, on peut parcourir sa descendance pour la détruire au préalable. Lors de la fin d'exécution normale d'un processus *P*, deux solutions sont possibles:

- La destruction de *P* n'est effective que lorsque tous ses fils sont eux-mêmes achevés. Ceci est nécessaire lorsque le contexte initial du fils est inclus dans le contexte de *P*, puisque la destruction de celui-ci entraînerait la perte d'une partie du contexte du fils, et donc un déroulement anormal du processus fils s'il pouvait exister après la destruction de son père.
- La destruction de *P* entraîne le rattachement de ses fils à l'un de ses ancêtres. Cependant comme l'ascendance du processus *P* n'a pas de connaissance de ce qu'a fait *P*, on rattache souvent les fils soit au processus qui a initialisé le travail (login), soit à la racine qui est un processus standard et éternel.

## 12.2.3. L'exemple de Unix

Dans le système Unix, la création dynamique de processus est simplifiée à l'extrême, puisqu'il s'agit de créer un processus qui est une exacte copie de celui qui demande la création. Aucun paramètre n'est donc nécessaire, puisque le programme est le même, et que le système réalise une copie des données, variables et registres du processus demandeur pour créer le nouveau processus. La seule distinction entre le processus créateur et le processus créé, réside dans la valeur retournée par la fonction de création. Le processus créateur reçoit l'identité du processus créé, alors que ce dernier reçoit la valeur 0. Par ailleurs, le système structure les processus en arbre. Considérons le petit morceau de programme suivant:

```
id_fils := fork ();  { création du fils }
si id_fils = 0 alors { il s'agit du processus fils }
                sinon { il s'agit du processus père }
```

La fonction `fork ()` crée le processus fils comme copie conforme du processus père. Les deux processus repartent donc après l'appel de la fonction, avec le même programme, et leur propres

zones de données et de variables identiques à ce moment. La mémorisation du résultat dans la variable `id_fils` signifie la mémorisation dans un emplacement mémoire propre à chacun d'eux. Notons que, puisque le fils obtient une copie des données du père, celui-ci peut lui transmettre tout ce qu'il désire à ce moment. Par la suite, les deux processus sont effectivement indépendants et n'ont pas de données communes à l'exception des objets externes.

Si le processus père se termine avant le processus fils, le fils est rattaché au processus racine de façon à conserver la structure d'arbre. Le processus père peut attendre la terminaison d'un de ses fils par la fonction (en langage C):

```
id_fils := wait (&status);
```

qui retourne le numéro d'un processus fils qui est terminé, en mettant dans la variable `status` un code indiquant la façon dont ce processsus s'est terminé. Si aucun fils n'est terminé alors qu'il y en a encore d'actifs, le processus père est mis en attente d'une terminaison de l'un de ses fils. S'il n'y a plus de processus fils, la fonction retourne la valeur `-1`.

Par ailleurs le système Unix fournit une fonction `exec` qui permet à un processus de changer le programme en cours d'exécution. Cette fonction n'affecte évidemment que le processus demandeur. Elle change l'état du processus en remplaçant le code des instructions, en supprimant les données, variables et registres qui étaient relatifs à l'ancien programme et en les initialisant pour le nouveau. La combinaison de la fonction `fork` et de cette fonction `exec` par le processus fils permet de réaliser la fonction `créer_processus` mentionnée ci-dessus.

# 12.3. La notion de ressources

Si nous reprenons notre exemple du processus de réalisation du gâteau d'anniversaire, ce processus ne peut s'exécuter que si les denrées nécessaires pour la recette sont disponibles, ainsi que les ustensiles de cuisines correspondants. Ce sont des ressources dont a besoin ce processus. Si on désire faire un deuxième gâteau identique, un deuxième processus doit être créé pour le réaliser. Cet autre processus aura besoin également de ressources, et peut alors être en conflit avec le premier processus s'il n'y a pas assez de certaines ressources. Par exemple, si la cuisson doit être faite au four, il est possible que le four soit assez grand pour admettre les deux gâteaux en même temps. Le batteur ne pourra pas par contre être utilisé en même temps par les deux processus. Le processeur étant unique (l'informaticien) ne pourra pas commencer de battre la pâte pour le compte de l'un des processus, si l'autre possède la ressource batteur et ne l'a pas restituée (lavé et essuyé les embouts). Par contre, l'horloge peut être partagée par tous les processus éventuels gérés par l'informaticien.

On appelle *ressource* toute entité dont a besoin un processus pour s'exécuter. Il en est ainsi du processeur physique, de la mémoire, des périphériques. Il en est également ainsi des données dont a besoin le processus et qui seraient momentanément indisponibles. Il en est enfin ainsi de l'événement de fin d'exécution d'un processus Unix pour le processus père qui l'attend. Pour chaque ressource, une caractéristique importante est le nombre de processus qui peuvent utiliser la ressource au même moment.

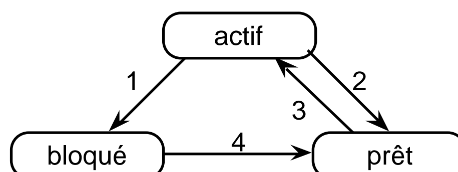
- Il peut y en avoir un nombre quelconque. Il n'y a alors pas de contrôle à mettre en œuvre. Dans l'exemple culinaire, l'horloge a été considérée comme une ressource qui peut être accédée par un nombre quelconque de processus.
- Il peut y en avoir plusieurs, mais en nombre limité. Il faut alors contrôler lors des allocations que ce nombre n'est pas dépassé. Dans l'exemple culinaire, le four ne peut être utilisé simultanément que par deux processus.
- Il peut y avoir au plus un processus qui utilise la ressource. Dans l'exemple culinaire, le batteur ne peut être utilisé que par au plus un processus. On dit alors que la ressource est une *ressource critique*. On dit aussi que les processus sont en *exclusion mutuelle* pour l'accès à cette ressource. Il en est ainsi du processeur physique, d'une imprimante, d'un dérouleur de bande, etc...

Pour les ressources physiques critiques, nous avons déjà proposé une première solution pour le processeur, qui peut être généralisée dans certains cas, en remplaçant une telle ressource physique par une *ressource virtuelle*. Dans ce cas, le système met en œuvre des mécanismes de coopération entre les processus, qui sont cachés dans le système et donc invisibles des processus eux-mêmes, pour leur permettre de “faire comme si” ils avaient la ressource pour eux seuls. On peut ainsi

simuler un nombre quelconque d'imprimantes virtuelles sous forme de fichiers sur disque dont les impressions sont réalisées successivement par le système sur une même imprimante physique.

# 12.4. Les états d'un processus

Lorsqu'un processus n'a pas toutes les ressources dont il a besoin pour s'exécuter, il est nécessaire de le *bloquer* en attendant que ces ressources soient disponibles. La figure 12.1 montre les différents états que peut prendre un processus, lorsqu'il existe. Lors de la création il est mis, en général, dans l'état bloqué, en attendant qu'il ait toutes les ressources dont il a besoin initialement. Sa destruction peut subvenir dans n'importe quel état à la suite d'une décision interne s'il est actif, ou externe s'il est dans un autre état. Dans ce cas, il faut récupérer toutes les ressources qu'il possédait.




1 Le processus a besoin d'une ressource dont il ne dispose pas
2/3 Décision de l'allocateur du processeur
4 Sur intervention extérieure au processus, lors de l'allocation de la ressource

***Fig. 12.1.*** *Les principaux états d'un processus.*

Il est assez naturel de considérer de façon particulière la ressource processeur physique. Pour le moment c'est encore une ressource chère, qu'il vaut mieux ne pas gaspiller. Par ailleurs il est inutile de la donner à un processus à qui il manque une autre ressource, puisque ce processus ne peut alors évoluer: le processeur ne pourrait que constater ce manque et attendre la disponibilité de cette ressource (c'est ce que l'on appelle l'*attente active*). C'est pourquoi on distingue trois états:

- l'*état actif* où le processus dispose de toutes les ressources dont il a besoin,
- l'*état bloqué* où le processus a besoin d'au moins une ressource autre que le processeur physique,
- l'*état prêt* où le processus dispose de toutes les ressources à l'exception du processeur physique.

La transition 1 survient lorsque, le processus étant actif, il exprime le besoin de disposer d'une nouvelle ressource. Cette expression peut être explicite, sous forme d'une demande au système, ou implicite, sous forme d'un accès à cette ressource non allouée. Le processus doit évidemment disposer du processeur pour pouvoir exprimer ce besoin. Il peut se faire que la transition vers l'état bloqué par manque de ressources, entraîne que le système lui retire d'autres ressources qu'il possédait, en plus de la ressource processeur physique, comme par exemple de la mémoire centrale. En général, on ne lui retire que des ressources chères, qu'il est relativement facile et peu coûteux de lui restituer ultérieurement, dans l'état où elles étaient au moment où on les lui a enlevées. Il est assez naturel de considérer de façon particulière la ressource processeur physique. L'état bloqué correspond alors au manque d'une ressource autre que le processeur physique, et l'état prêt correspond alors au manque de la seule ressource processeur.

La transition 4 est la conséquence d'un événement extérieur au processus. Celui-ci était bloqué en attente d'une ou plusieurs ressources. Lorsque le système constate qu'il a pu lui allouer toutes les ressources (autres que le processeur) dont il a besoin, il le fait passer dans l'état prêt. La disponibilité de la ressource attendue se présente, en général, sous l'une des deux formes suivantes:

- C'est le résultat d'une action par un autre processus. Il peut s'agir de la libération de la ressource, ou de la création de la ressource elle-même. Ainsi la fin d'exécution d'un processus crée la ressource événement attendue par son père.
- C'est le résultat d'une action extérieure. Il peut s'agir d'une interruption en provenance d'un périphérique, par exemple, ou d'un événement déclenché par l'opérateur ou un utilisateur.

Les transitions 2 et 3 sont sous la responsabilité de la partie du système qui alloue le processeur physique. C'est à ce niveau que s'implante la notion de processeur virtuel dont nous avons déjà

parlé. Certains systèmes n'implantent pas la transition 2. Il s'ensuit que lorsqu'un processus est actif, il le reste jusqu'à ce qu'il soit terminé ou qu'il ait besoin d'une ressource qu'il n'a pas. L'allocation du processeur consiste alors à choisir un processus parmi les processus prêts, et à le faire passer dans l'état actif. Cela peut entraîner que les processus prêts peuvent attendre très longtemps pour devenir actif, si le choix a conduit à rendre actif un processus qui calcule sans exprimer le besoin de nouvelles ressources. Ceci peut être évité par le biais de la transition 2 qui doit être la conséquence d'une action extérieure au processus actif. C'est en général un des rôles attribués à l'interruption d'horloge de déclencher cette transition.

Il est important de noter que ces transitions ne sont pas instantanées, puisqu'il s'agit de ranger en mémoire l'état du processeur relatif à l'ancien processus actif, et de restituer l'état du processeur avec celui relatif au nouveau processus actif, une fois le choix de celui-ci effectué. L'exécution de ces transitions consomme du temps de processeur physique, qui n'est pas utilisé par les processus eux-mêmes. C'est ce que l'on appelle la *déperdition* (*overhead*) résultant de la gestion des processus. Tous les systèmes visent à réduire cette déperdition dans des limites raisonnables. C'est le prix à payer pour une meilleure fonctionnalité.

# 12.5. Conclusion

☞ Un processus est l'entité dynamique qui correspond à l'exécution d'une suite d'instructions définie par le programme. Le processeur est l'entité qui exécute les instructions pour le compte du processus.

☞ Lorsqu'un processus n'a pas le processeur, il ne peut évoluer, et son état doit être conservé en mémoire. Lorsqu'il a le processeur, son état évolue, et est représenté en partie par les registres du processeur.

☞ Il peut y avoir un nombre fixe de processus dans un système, ou bien les processus peuvent être créés dynamiquement. Dans ce cas, l'ensemble des processus est structuré en arborescence suivant la relation créateur-créé.

☞ La création d'un processus doit définir le programme qu'il doit exécuter, et le contexte initial de cette exécution. Sur Unix, le processus créé est une copie conforme du processus créateur.

☞ Une ressource est une entité dont a besoin un processus à un instant donné pour s'exécuter. Elle est caractérisée, en particulier, par le nombre de processus qui peuvent l'utiliser au même moment. Ce nombre peut être illimité, et aucun contrôle n'est nécessaire lors de l'allocation.

☞ Si le nombre de processus pouvant utiliser une ressource est borné, il faut contrôler lors de l'allocation que la borne n'est pas atteinte. Lorsque cette borne est égale à 1, on dit que la ressource est critique, et que les processus sont en exclusion mutuelle pour cette ressource.

☞ Les ressources induisent trois états sur les processus: l'état actif lorsque le processus a toutes ses ressources, l'état prêt lorsque le processus a toutes ses ressources sauf le processeur et l'état bloqué lorsqu'il lui manque des ressources autres que le processeur.

☞ Le passage de l'état actif à l'état bloqué est en général volontaire, ou à la suite d'une réquisition. Le passage de l'état bloqué à l'état prêt est dû à une action externe au processus.

# 13

# Synchronisation et communication entre processus

Nous avons déjà dit que, si les processus étaient des entités autonomes et indépendantes, ils pouvaient se trouver en conflit pour l'accès à certaines ressources communes. Ceci nécessite la mise en œuvre de mécanismes dits de synchronisation pour gérer ces conflits. Par ailleurs, si les processus sont en général indépendants, cela ne doit pas interdire la communication. La conséquence de ces mécanismes est le blocage des processus en attente d'une ressource momentanément indisponible. Deux problèmes en découlent: l'interblocage et la famine.

## 13.1. Les mécanismes de synchronisation

Considérons un compte bancaire, dont le montant est mémorisé dans un emplacement donné `A` sur disque. Le programme qui consiste à ajouter `100` à ce compte pourrait être le suivant, où `N` est une variable locale du programme:

```
lire (N, A);
N := N + 100;
écrire (N, A);
```

Si maintenant deux processus différents consistent en l'exécution de ce même programme, le processeur sera alloué à chacun d'eux dans un ordre quelconque. En particulier, on peut imaginer que l'ordre soit le suivant:

| processus $P_1$ | processus $P_2$ |
|---|---|
| `lire (N, A);` | |
| | `lire (N, A);` |
| | `N := N + 100;` |
| | `écrire (N, A);` |
| `N := N + 100;` | |
| `écrire (N, A);` | |

`N` étant une variable locale du programme, implique qu'il en existe un exemplaire pour chacun des deux processus. Il s'ensuit que, si la valeur initiale du compte est `1000`, on constate qu'après ces exécutions, il est de `1100` au lieu de `1200`. Les deux processus ne sont pas en fait totalement indépendants. Ils partagent la ressource commune qu'est la valeur du compte bancaire à l'adresse `A` sur disque. Cette ressource doit être à un seul point d'accès, c'est donc une ressource critique, et les processus sont en exclusion mutuelle sur cette ressource critique.

Si la variable `N` est commune aux deux processus, le problème n'est pas résolu pour autant. En effet l'instruction `N := N + 100` n'est pas une instruction “indivisible”. En fait elle est décomposée en trois instructions de la machine qui sont par exemple:

```
LOAD      N
ADD       100
STORE     N
```

L'allocateur du processeur prend en compte les instructions de la machine et non les instructions du langage évolué pour déterminer les moments où il peut remplacer le processus actif par un autre. En particulier, dans cet exemple, il est possible que le changement ait lieu juste après la première instruction, donnant la séquence suivante:

| processus $P_1$ | processus $P_2$ |
|---|---|
| `LOAD      N` | |
| | `LOAD      N` |
| | `ADD       100` |
| | `STORE     N` |
| `ADD       100` | |
| `STORE     N` | |

## 13.1.1. Les verrous

Un mécanisme proposé pour permettre de résoudre l'exclusion mutuelle d'accès à une ressource est le mécanisme de *verrou* (en anglais *lock*). Un verrou est un objet système sur lequel deux opérations sont définies. Un tel objet peut s'assimiler à une ressource logiciel, les opérations permettant d'acquérir ou de libérer cette ressource.

- *verrouiller (v)* permet au processus d'acquérir le verrou *v* s'il est disponible. S'il n'est pas disponible, le processus est bloqué en attente de la ressource.
- *déverrouiller (v)* permet au processus de libérer le verrou *v* qu'il possédait. Si un ou plusieurs processus étaient en attente de ce verrou, un seul de ces processus est réactivé et reçoit le verrou.

En tant qu'opérations systèmes, ces opérations sont indivisibles, c'est-à-dire que le système garantit l'absence d'interférence entre leurs exécutions par plusieurs processus. On dit encore que ce sont des *primitives*.

## 13.1.2. Les sémaphores

Un *sémaphore* est un mécanisme proposé par E.W.Dijkstra en 1965, et qui est un peu plus général que le verrou. Il se présente comme un distributeur de jetons, mais le nombre de jetons est fixe et non renouvelable: les processus doivent restituer leur jeton après utilisation. S'il y a un seul jeton en circulation, on retrouve le verrou. Les deux opérations sur un sémaphore sont traditionnellement dénotées *P(s)* et *V(s)*, mais cette notation est de peu de signification pour ceux qui parlent le Hollandais, et aucune pour les autres[6]!

- *P(s)* ou *down(s)* permet à un processus d'obtenir un jeton, s'il y en a de disponibles. Si aucun n'est disponible, le processus est bloqué.
- *V(s)* ou *up(s)* permet à un processus de restituer un jeton. Si des processus étaient en attente de jeton, l'un d'entre eux est réactivé et le reçoit.

Notons qu'un sémaphore peut être vu comme un couple constitué d'un entier, encore appelé le *niveau du sémaphore* et d'une file d'attente de processus. Le niveau est le nombre de jetons encore disponibles. Il est évident que si le niveau est positif, la file d'attente est vide. Parfois on utilise les valeurs négatives du niveau pour représenter le “déficit” en jetons, c'est à dire le nombre de processus en attente de jeton.

À la création d'un sémaphore, il faut décider du nombre de jetons dont il dispose. On voit que si une ressource est à un seul point d'accès (critique), le sémaphore doit avoir initialement 1 jeton, qui sera attribué successivement à chacun des processus demandeurs. Si une ressource est à `n` points d'accès, c'est-à-dire, peut être utilisée par au plus `n` processus à la fois, il suffit d'un sémaphore initialisé avec `n` jetons. Dans les deux cas, un processus qui cherche à utiliser la ressource, demande d'abord un jeton, puis utilise la ressource lorsqu'il a obtenu le jeton et enfin rend le jeton lorsqu'il n'a plus besoin de la ressource.

---

[6] En Français, on peut leur donner éventuellement la signification mnémonique *Puis-je* pour *P*, et *Vas-y* pour *V*.

### 13.1.3. Les mécanismes plus élaborés

Les mécanismes proposés ci-dessus sont assez rudimentaires. Ils posent deux problèmes:

- Que faire lorsqu'on désire mettre en œuvre une politique d'allocation plus élaborée?
- Que faire pour obliger les processus à respecter une certaine règle du jeu?

Le premier problème peut être résolu en construisant un algorithme plus ou moins complexe, qui utilise une zone de mémoire commune aux processus et des sémaphores pour assurer la synchronisation. La figure 13.1 donne un exemple où l'on distingue deux sortes de processus qui accèdent à un même fichier, les “lecteurs” qui exécutent une suite de lectures sur le fichier, et les “rédacteurs” qui exécutent une suite de lectures et d'écritures. Les morceaux de programme correspondants montrent une façon de garantir qu'un rédacteur accède seul au fichier, alors que les lecteurs peuvent y accéder ensembles.

```
                         initialisation
                         mutex.niveau := 1;
                         lect_redac.niveau := 1;
                         n_l := 0;
processus lecteurs                 processus rédacteurs
down(mutex);
n_l := n_l + 1;
si n_l = 1 alors
     down(lect_redac);             down(lect_redac);
finsi;
up(mutex);
...
lectures                           lectures et écritures
...
down(mutex);
n_l := n_l - 1;
si n_l = 0 alors
     up(lect_redac);               up(lect_redac);
finsi;
up(mutex);
```

***Fig. 13.1.*** *Exemple de synchronisation entre des lecteurs et des rédacteurs.*

Deux sémaphores dotés chacun d'un seul jeton sont utilisés, ainsi qu'un compteur commun qui note le nombre de lecteurs en cours. Le jeton `lect_redac` est possédé soit par un rédacteur, soit par l'ensemble des lecteurs. C'est pourquoi, seul le premier lecteur demande ce jeton, qui est restitué par le dernier lecteur qui sort. Le deuxième sémaphore contrôle l'accès à la ressource critique `n_l` par les lecteurs.

Le deuxième problème évoqué ci-dessus est bien visible dans cet exemple: comment obliger les processus lecteurs ou rédacteurs à respecter la règle du jeu qui vient d'être décrite? La solution adoptée par certains systèmes est de définir plusieurs politiques de gestion de certaines ressources par le biais d'opérations spécifiques, et de contrôler le respect de la règle du jeu par les processus dans ces opérations.

Ainsi certains systèmes de gestion de fichiers ont un paramètre particulier de l'opération d'ouverture qui précise si le processus désire l'accès exclusif ou partagé au fichier.

- Lorsque le processus demande l'accès exclusif, il sera mis en attente si un autre processus accède déjà au fichier; de plus, lorsque le processus obtient l'accès, le système empêche tout autre processus d'y accéder.
- Lorsque le processus demande l'accès partagé, il ne sera mis en attente que si un processus a obtenu l'accès en exclusif sur le fichier.

Notons que ceci implique que le système exécute pour le compte du processus un algorithme voisin de celui d'un lecteur ou d'un rédacteur proposé plus haut.

De même, les SGBD prennent en compte les conflits d'accès aux données de la base, et obligent explicitement ou implicitement les processus à respecter une règle du jeu fixée par le SGBD ou par l'administrateur de la base de données. Du point de vue des processus, une règle du jeu implicite

transforme la base de données en une ressource virtuelle au sens où nous l'avons défini dans le chapitre précédent, les processus pouvant y accéder comme s'ils étaient seuls.

# 13.2. La communication entre processus

Lorsque des processus ont besoin d'échanger des informations, ils peuvent le faire par l'intermédiaire d'une zone de mémoire commune. Nous en avons déjà vu un exemple lors de la présentation des lecteurs et des rédacteurs ci-dessus: les lecteurs accèdent au même compteur `n_l`. Cette communication présente l'inconvénient d'être peu structurée, et de nécessiter l'utilisation de l'exclusion mutuelle d'accès à cette zone commune. Par ailleurs elle pose des problèmes de désignation des données de cette zone qui doit être dans l'espace de chaque processus qui désire communiquer avec les autres.

## 13.2.1. Le schéma producteur-consommateur

On peut définir un mécanisme général de communication entre processus, où l'un est l'émetteur de l'information (on lui donne le nom de *producteur*), et l'autre est le récepteur de l'information (on lui donne le nom de *consommateur*). Il n'est souvent pas nécessaire de faire attendre le producteur jusqu'à ce que le consommateur ait reçu l'information. Pour cela il suffit de disposer d'un tampon entre les deux processus pour assurer la mémorisation des informations produites non encore consommées. Nous appellerons *message* le contenu d'une telle information. Le tampon peut recevoir un nombre limité de ces messages, noté `N`. Les deux processus doivent se synchroniser entre eux de façon à respecter certaines contraintes de bon fonctionnement. La figure 13.2 donne un schéma de cette synchronisation.

```
                    initialisation
                    n_plein.niveau := 0;
                    n_vide.niveau := N;
producteur                              consommateur
down(n_vide);                           down(n_plein);
dépôt dans le tampon;                   retrait du tampon;
up(n_plein);                            up(n_vide);
```

***Fig. 13.2.*** *Le schéma producteur-consommateur.*

- Le producteur ne peut déposer un message dans le tampon s'il n'y a plus de place libre. Le nombre de places libres dans le tampon peut être symbolisé par un nombre correspondant de jetons disponibles; ces jetons peuvent être conservés par un sémaphore `n_vide`.
- Le consommateur ne peut retirer un message depuis le tampon s'il n'y en a pas. Le nombre de messages déposés peut être également symbolisé par un nombre correspondant de jetons disponibles; ces jetons peuvent être conservés par un sémaphore `n_plein`.
- Le consommateur ne doit pas retirer un message que le producteur est en train de déposer. Cette dernière contrainte est implicitement obtenue, dans le schéma de la figure, si les messages sont retirés dans l'ordre où ils ont été mis.

Informellement, on peut dire que le producteur prend d'abord un jeton de place libre, dépose son message et rend un jeton de message disponible. De son côté, le consommateur prend un jeton de message disponible, retire le message et rend un jeton de place libre.

Pour obliger les processus à respecter la règle du jeu, les systèmes proposent souvent des opérations de dépôt et de retrait qui assurent elles-mêmes la synchronisation entre les processus.

## 13.2.2. Les tubes Unix

Les *tubes* Unix (en anglais *pipes*) sont une implantation de la communication suivant le schéma producteur-consommateur, avec un tampon de taille fixe, interne au système. Pour un processus, un tube se présente comme deux flots, l'un ouvert en écriture (pour le producteur), l'autre ouvert en lecture (pour le consommateur). Lorsqu'un processus crée un tube, le système lui retourne les deux flots. Si ce processus crée ensuite un processus fils (opération `fork` vue précédemment), celui-ci obtient une copie des flots ouverts, lui permettant ainsi de lire ou d'écrire dans le tube, à moins que

l'un de ces flots n'ait été fermé. Ne peuvent donc communiquer par un tube que les processus situés dans la sous-arborescence dont la racine est le processus qui a créé ce tube.

Lors d'une écriture de `n` octets par le producteur, celui-ci sera mis en attente s'il n'y a pas assez de place dans le tube pour y mettre les `n` octets. Il sera réactivé lorsque les `n` octets auront pu être tous écrits dans le tube. Par ailleurs, le système interrompra l'exécution normale du processus s'il n'y a plus de consommateur pour ce tube.

Lors d'une lecture de `p` octets par le consommateur, celui-ci sera mis en attente s'il n'y a pas assez d'octets dans le tube pour satisfaire la demande. Il sera réactivé lorsque le nombre d'octets demandé aura pu lui être délivré. Par ailleurs, s'il n'y a plus de producteur pour ce tube, le système lui délivrera les octets restants, à charge pour le consommateur, lorsqu'il obtient un nombre d'octets nul, d'en déduire que la production est terminée.

La figure 13.3 donne un exemple de cette communication. Le processus père crée le tube et le processus fils, puis envoie dans le tube la suite des entiers de 0 à 1000, les entiers étant supposés sur 2 octets. Le processus fils lit les octets 2 par 2 depuis le tube dans un entier et imprime celui-ci. Il s'arrête lorsque le tube est vide et que le père a fermé le tube en écriture, puisque alors il n'y a plus de producteur.

```
var tube: tableau [0..1] de flot;
   i : entier;
début pipe(tube);                                { création du tube }
     si fork()   0 alors                      { création du processus fils }
          fermer (tube[0]);              { le père ne lit pas dans le tube }
          pour i := 0 jusqu'à 1000 faire
               écrire (tube[1], i, 2);          { écriture de deux octets }
          fait;
          fermer (tube[1]);                               { le père a fini }
          wait(0);                         { attente de la fin du fils }
     sinon
          fermer (tube[1]);              { le fils n'écrit pas dans le tube }
          tant que lire (tube[0], i, 2) = 2 faire
               imprimer (i); { on suppose qu'un entier occupe 2 octets }
          fait;
          fermer (tube[0]);                                { le fils a fini }
     finsi;
fin;
```

***Fig. 13.3.*** *Communication par tubes Unix.*

Les tubes Unix ne conservent pas la structuration des messages qui sont envoyés par le producteur. Dans l'exemple ci-dessus, producteur et consommateur participent tous les deux à cette structuration: le consommateur lit depuis le tube les octets 2 par 2 pour retrouver la structure des messages émis par le producteur. Notons que ceci est conforme à la philosophie déjà énoncée de ce système qui présente tout objet externe comme une suite linéaire d'octets. Cette uniformité présente l'avantage de cacher au programme la nature de l'objet externe manipulé, même lorsqu'il s'agit d'un tube.

## 13.2.3. La communication par boîte aux lettres

Nous appellerons *boîte aux lettres* un objet système qui implante le schéma producteur-consommateur défini plus haut, en conservant le découpage en messages tels que les a déposés le producteur. Les messages sont parfois typés, comme dans iRMX86, où les messages comportent un en-tête qui précise, en particulier, sa longueur et son type sous la forme d'un code défini par le programmeur.

Le problème de la désignation d'une boîte aux lettres par les processus est résolu de différentes façons. Il s'agit, en effet, de permettre à des processus dont les espaces mémoire sont disjoints de pouvoir désigner une même boîte aux lettres. Une solution comme celle proposée par Unix pour les tubes, peut être retenue, qui consiste à mettre cette désignation dans le contexte des processus au moment de leur création. On préfère, en général, une solution semblable à une édition de liens, qui consiste à donner un nom symbolique aux boîtes aux lettres, sous forme d'une chaîne de caractères. Le système maintient une table de ces noms symboliques associés aux boîtes aux lettres qui ont déjà été créées et non encore détruites. Lorsqu'un processus demande l'accès à une telle boîte aux lettres,

le système consulte cette table, et retourne au demandeur le descripteur de la boîte aux lettres (c'est, par exemple, son indice dans le table). Les opérations suivantes se feront en utilisant ce descripteur. Dans certains cas, cette demande d'accès entraîne sa création implicite si elle n'existe pas, alors que d'autres systèmes séparent l'opération de création explicite de la demande d'accès.

Les opérations sur une boîte aux lettres sont l'envoi et la réception de message. L'envoi peut être avec ou sans blocage du demandeur lorsque la boîte aux lettres est pleine. La réception peut être également avec ou sans blocage lorsque la boîte aux lettres est vide. De plus, certains systèmes permettent de filtrer les messages reçus, c'est-à-dire, qu'il est possible de préciser le type des messages que l'on accepte de recevoir.

Les boîtes aux lettres ont souvent une capacité limitée de messages, comme nous l'avons indiqué dans le schéma général. Parfois cette capacité est nulle, l'envoi de message ne pouvant avoir lieu que lorsque le receveur est prêt à recevoir le message. Constatons que le schéma précédent doit être modifié; nous en laissons le soin au lecteur. On dit alors qu'il y a communication synchrone entre le producteur et le consommateur. L'intérêt essentiel est que cette fois, après le retour de la primitive d'envoi, le producteur a la garantie que le consommateur a effectivement reçu le message, alors que dans le cas général, le producteur ne peut préjuger du moment où cette réception aura lieu. En contrepartie, les processus supportent une contrainte de synchronisation plus forte.

## 13.3. Les conséquences sur les temps de réponse

Dans les premiers chapitres, nous avons défini le temps de réponse comme le délai qui sépare l'envoi d'une commande par un utilisateur, de la fin d'exécution de la commande. Nous pouvons ici décomposer ce délai en trois mesures différentes, suivant les états du processus correspondant.

Dans l'état actif, on mesure le temps de processeur (le temps virtuel) nécessaire à l'exécution proprement dite de la commande. Ce temps est évidemment incompressible, pour un programme donné. L'algorithme de la commande elle-même doit être efficace pour minimiser ce temps.

Dans l'état prêt, on mesure le temps pendant lequel le processus attend le processeur, ressource possédée par un autre processus. L'idéal, pour l'utilisateur, est de réduire le plus possible ce temps. C'est en quelque sorte le prix qu'il doit payer pour ne pas avoir à payer plus cher la machine. Sur un système normalement chargé, ce temps ne devrait pas être important. Si ce temps est important pour tous les processus, il est probable que ce soit l'indication d'un système surchargé, qui nécessite une augmentation de puissance.

Dans l'état bloqué, on mesure le temps pendant lequel le processus attend une ressource autre que le processeur. Cette attente peut avoir deux causes:

- La première est due à la lenteur de l'accès demandé lors des entrées-sorties. Comme pour le temps de processeur, le programmeur peut améliorer l'algorithme pour réduire le nombre de ces entrées-sorties. Un choix plus judicieux du support peut également avoir une certaine influence.
- L'autre cause est due au fait que la ressource est possédée par un autre processus. Ceci implique un conflit entre les utilisateurs. Évidemment, si l'utilisateur était seul, ce conflit n'existerait pas. C'est souvent un paramètre très important, mais parfois négligé et difficile à contrôler. La réduction de ces conflits nécessite de multiplier les ressources (beaucoup de petites au lieu d'une grosse), pour diminuer la probabilité de conflit, et de minimiser la portion de programme pendant laquelle les processus réservent l'accès à ces ressources (réservation le plus tard possible et libération le plus tôt possible).

## 13.4. La notion d'interblocage

La synchronisation et la communication des processus peut conduire à un autre problème. Supposons qu'un premier processus $P_1$ demande et obtienne la ressource imprimante, seule ressource de ce type. Un second processus $P_2$ demande ensuite et obtient la ressource dérouleur de bande, également seule de ce type. Si $P_1$ demande ultérieurement la ressource dérouleur de bande sans avoir libéré la ressource imprimante, il est bloqué en attendant la libération par $P_2$ . Si maintenant, $P_2$ demande la ressource imprimante avant de libérer la ressource dérouleur de bande, il est bloqué en attendant la libération par $P_1$ de cette ressource. On voit que chacun des processus est

bloqué, en attendant que l'autre libère la ressource qu'il possède. On dit qu'il y a un *interblocage* entre les processus (en anglais *deadlock*).

On peut donner une représentation graphique à l'interblocage. Représentons les ressources par des rectangles, et les processus par des cercles. On met une flèche d'une ressource vers un processus lorsque la ressource est allouée au processus, et une flèche d'un processus vers une ressource lorsque le processus attend cette ressource. La figure 13.4 représente l'évolution du graphe de notre exemple précédent conduisant à l'interblocage.

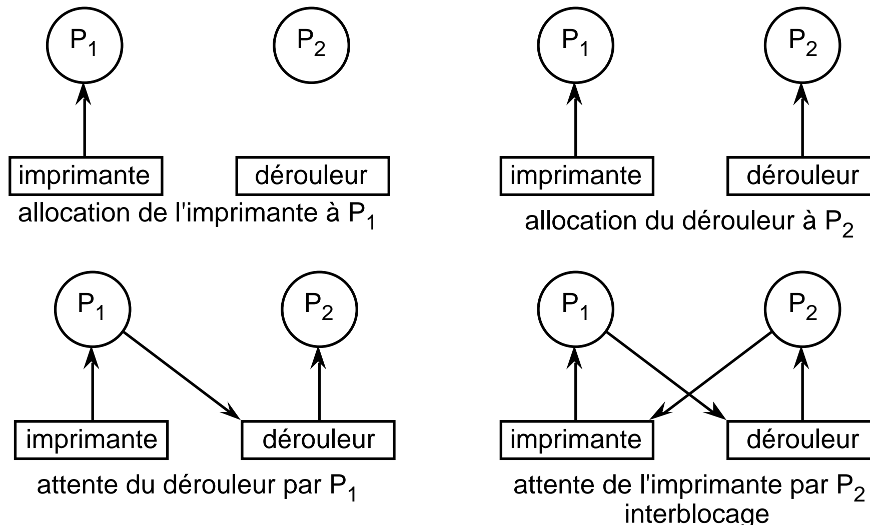
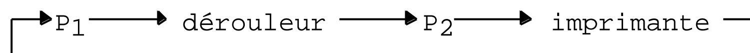



***Fig. 13.4.*** *Allocation conduisant à l'interblocage.*

On peut constater que dans le graphe final, il y a un cycle représenté par:




En fait, tout interblocage est signalé par la présence d'un tel cycle dans ce graphe, et inversement, ce qui explique l'intérêt de cette représentation.

Le problème de l'interblocage est résolu de diverses façons par les systèmes.

- La première méthode est la *politique de l'autruche*: elle consiste à ignorer le problème! Considérant que le coût des autres méthodes est assez élevé, il peut s'avérer rentable de ne pas résoudre le problème lorsque la probabilité d'avoir un interblocage est faible. S'il survient, il faut relancer le système.
- La deuxième méthode est une méthode de *détection-guérison*. Elle consiste à vérifier de temps en temps la présence de tels cycles, et à tuer un ou plusieurs processus jusqu'à la disparition de ces cycles. Chaque destruction d'un tel processus entraîne évidemment la restitution des ressources qu'il possédait.
- La troisième méthode est une méthode de *prévention*. Elle consiste à imposer des contraintes sur les demandes des processus de telle sorte qu'un tel cycle ne puisse jamais se produire. Ainsi, si les deux processus ci-dessus demandaient leurs ressources dans le même ordre (par exemple imprimante puis dérouleur), le deuxième processus serait bloqué sur la demande de la première ressource, sans avoir encore obtenu aucune ressource, et il ne pourrait y avoir de cycle.
- La quatrième méthode est une méthode d'*évitement*. Elle consiste à prévoir à l'avance les différents cas où un cycle pourrait se former, et à éviter l'interblocage en retardant une allocation qui serait possible, jusqu'à être certain qu'il n'y a pas de risque d'interblocage. La figure 13.5 montre sur notre exemple qu'en retardant l'allocation du dérouleur à $P_2$ jusqu'à ce que $P_1$ ait libéré l'imprimante, on permet à celui-ci d'acquérir le dérouleur avant $P_2$ et on évite ainsi la création du cycle. Il faut évidemment savoir à l'avance quels seront les besoins des processus dans ce cas.

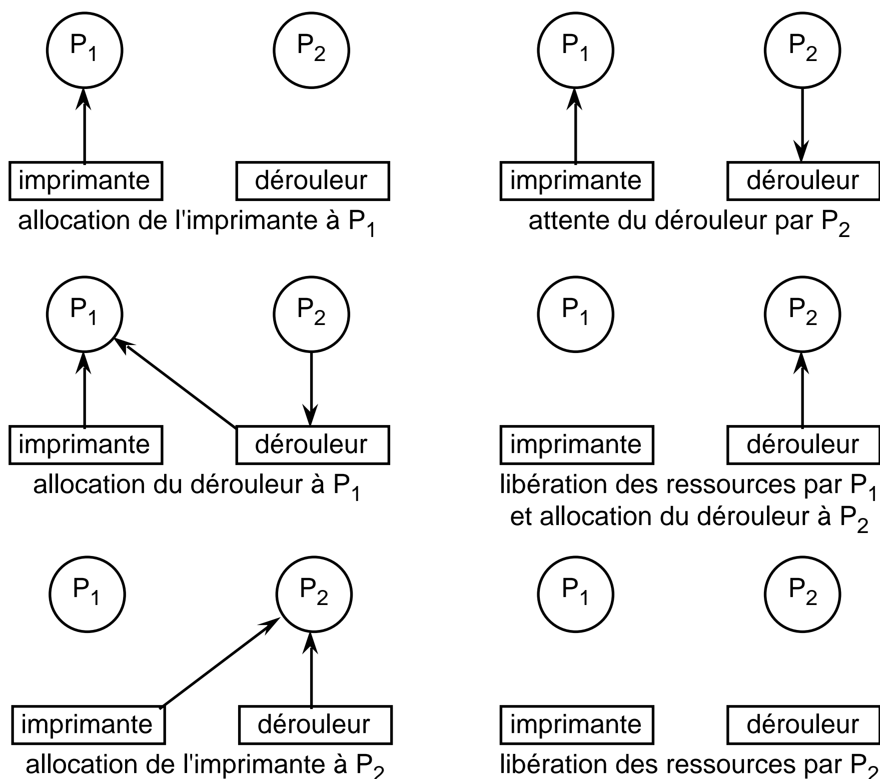



***Fig. 13.5.*** *Évitement de l'interblocage.*

La méthode utilisée dépend des applications qui sont les plus courantes sur le système. Par exemple, les systèmes d'IBM, mais aussi ceux de bien d'autres constructeurs, ont utilisé successivement plusieurs de ces méthodes. Après avoir appliqué la politique de l'autruche justifiée par le fait que les "plantages" du système étaient bien plus fréquents que les interblocages, la politique de prévention a été utilisée pour le traitement par lot. La politique d'évitement est également souvent utilisée pour le traitement par lot. Par contre, la plupart des systèmes transactionnels préfèrent la politique de détection-guérison, en détruisant une transaction qui demande une ressource si son blocage entraîne un cycle. Le noyau d'Unix utilise la politique de l'autruche, considérant que l'utilisateur qui constate un interblocage entre ses processus, peut toujours les tuer lui-même depuis sa console.

# 13.5. La notion de famine

Un processus qui est en attente d'une ressource peut rester bloqué indéfiniment sans être pour autant en situation d'interblocage. On dit que l'on a une situation de *famine* (en anglais *starvation*), lorsque un ou plusieurs processus n'obtiennent pas les ressources dont ils ont besoin par suite du comportement de l'ensemble des processus, sans être en situation d'interblocage. La différence essentielle réside dans le fait qu'il est possible de sortir de la situation de famine par suite du changement du comportement des processus, alors qu'il n'est pas possible de sortir d'un interblocage sans retirer autoritairement une ressource à un des processus en interblocage.

Reprenons l'exemple des lecteurs-rédacteurs que nous avons présenté plus haut. Supposons qu'aucun processus n'accède initialement au fichier. Lorsque le premier lecteur obtient le jeton `lect_redac`, il interdit à tous les écrivains de l'obtenir tant que le dernier lecteur ne l'a pas été libéré. Par contre les lecteurs peuvent toujours obtenir l'accès en lecture qu'ils désirent. Il peut s'en suivre une coalition des lecteurs de telle sorte qu'il y ait toujours un lecteur en permanence sur le fichier. Les rédacteurs n'obtiendront alors jamais satisfaction. Cependant, il ne s'agit pas d'un interblocage, puisque si les lecteurs s'arrêtent de travailler, le dernier libèrera le jeton `lect_redac` permettant ainsi aux rédacteurs d'obtenir l'un après l'autre satisfaction.

```
                    initialisation
                    équité.niveau := 1;
                    mutex.niveau := 1;
                    lect_redac.niveau := 1;
                    n_l := 0;
processus lecteurs              processus rédacteurs
down(équité);                   down(équité);
down(mutex);
n_l := n_l + 1;
si n_l = 1 alors
     down(lect_redac);          down(lect_redac);
finsi;
up(mutex);
up(équité);                     up(équité);
...
lectures                        lectures et écritures
...
down(mutex);
n_l := n_l - 1;
si n_l = 0 alors
     up(lect_redac);            up(lect_redac);
finsi;
up(mutex);
```

***Fig. 13.6.*** *Synchronisation équitable des lecteurs et des rédacteurs.*

La famine est souvent une conséquence de la politique d'allocation qui est suivie. Si elle est équitable, elle sera évitée. Sur notre exemple (figure 13.6), l'introduction d'un sémaphore supplémentaire `équité` évite la famine, pourvu que le déblocage des processus en attente sur un sémaphore soit effectué dans l'ordre où ils sont arrivés. En effet, tous les processus doivent maintenant acquérir un jeton `équité` pour savoir s'ils peuvent entrer, et le libèrent lorsqu'ils ont obtenu l'autorisation. Comme l'attribution du jeton `équité` est faite dans l'ordre où les processus sont demandeurs, un processus lecteur n'obtiendra le sien que lorsque tous les rédacteurs arrivés avant lui auront obtenu satisfaction. Par contre, si plusieurs lecteurs se suivent, le premier obtiendra le jeton `équité` puis le jeton `lect_redac`, et libèrera le jeton `équité` permettant aux lecteurs qui le suivent immédiatement d'entrer dans leur phase de lecture.

# 13.6. Conclusion

☞ Le mécanisme de verrouillage permet de faire en sorte qu'au plus un seul processus possède le verrou à un instant donné. Les opérations sur les verrous sont des primitives systèmes.

☞ Un sémaphore est une généralisation du verrou. Il s'apparente à un distributeur de jetons non renouvelables qui sont rendus après utilisation. Lorsqu'un processus désire un jeton et qu'aucun n'est disponible, il est mis en attente jusqu'à ce qu'un autre processus rende son jeton.

☞ Des mécanismes plus élaborés peuvent être construits à l'aide des sémaphores et de variables communes aux processus. Le respect d'une règle du jeu est souvent imposé par le système en fournissant directement ces mécanismes, comme par exemple l'accès exclusif ou partagé à un fichier.

☞ Le schéma producteur-consommateur est un mécanisme général de communication entre processus, dont les tubes Unix sont une implantation fournie par le système. Les boîtes aux lettres sont aussi une implantation de ce schéma qui tient compte de la structure des messages échangés.

☞ Le blocage des processus en attente de ressources a des conséquences évidentes sur les temps de réponse, et ceci ne doit pas être négligé.

☞ La conséquence du blocage des processus est le risque d'interblocage. Ceci se présente lorsqu'on a un ensemble de processus qui attendent des ressources qui sont toutes possédées par des processus de l'ensemble. Lorsqu'il survient, il est nécessaire de tuer des processus de l'ensemble.

☞ Un processus peut également être en famine, lorsqu'il attend une ressource qu'il n'obtient pas parce que les autres processus se coalisent pour l'en empêcher. On l'évite par une allocation équitable.

☞ La différence entre la famine et l'interblocage est qu'un changement de comportement des processus permettrait de satisfaire le processus en famine, alors que l'on ne peut sortir de l'interblocage qu'en retirant autoritairement une ressource à l'un des processus.

# 14

# La gestion de la mémoire centrale

A l'origine, la mémoire centrale était une ressource chère et de taille limitée. Elle devait être gérée avec soin. Sa taille a considérablement augmenté depuis, puisqu'un compatible PC a souvent aujourd'hui la même taille de mémoire que les plus gros ordinateurs de la fin des années 60. Néanmoins, le problème de la gestion reste important du fait des besoins croissants des utilisateurs.

## 14.1. La notion de multiprogrammation

La multiprogrammation a été présentée dans le premier chapitre comme un moyen pour améliorer l'efficacité de l'utilisation du processeur. Un processus est alternativement dans des phases actives, pendant lesquelles il évolue, et dans des phases d'attente d'une ressource, pendant lesquelles il n'a pas besoin du processeur. Dans un système *monoprogrammé*, la mémoire centrale est découpée en deux parties: une partie est réservée au système et le reste est attribué au processus. Lorsque le processus est en attente de ressource, le processeur n'a rien à faire. La *multiprogrammation* consiste à découper la mémoire centrale en plusieurs parties, de façon à mettre plusieurs processus en mémoire au même moment, de telle sorte à avoir plusieurs processus candidats au processeur.

### 14.1.1. L'intérêt de la multiprogrammation

Supposons que deux processus $P_1$ et $P_2$ soient présents en mémoire et que leur comportement corresponde à l'alternance de 100 ms de processeur et 100 ms d'attente d'entrées-sorties. On voit que le processeur va pouvoir faire évoluer $P_1$ pendant 100 ms jusqu'au lancement de son entrée-sortie, puis faire évoluer $P_2$ pendant 100 ms jusqu'au lancement de la sienne. A ce moment, l'entrée-sortie de $P_1$ étant terminée, le processeur pourra le faire évoluer pendant une nouvelle période de 100 ms, et ainsi de suite. De la sorte, d'une part, les deux processus travaillent à leur vitesse maximum, d'autre part, le processeur est utilisé à 100% au lieu de 50% en monoprogrammation.

La situation ci-dessus est en fait idéale. Les durées d'activité ou d'attente d'entrées-sorties des processus ne sont pas en fait constantes. Si la situation est améliorée, c'est-à-dire, si le processeur est mieux utilisé, il faut mettre, dans cet exemple, plus de deux processus en mémoire pour approcher un taux d'activité du processeur de 100%. La figure 14.1 donne le taux d'activité du processeur que l'on obtient en fonction du nombre de processus présents en mémoire, pour différents taux d'attente d'entrées-sorties de ces processus. Le nombre de processus présents en mémoire est encore appelé *degré de multiprogrammation*. En particulier, ces courbes montrent qu'il faut 5 processus ayant un taux d'entrées-sorties de 50% pour approcher 97% du taux d'activité du processeur.

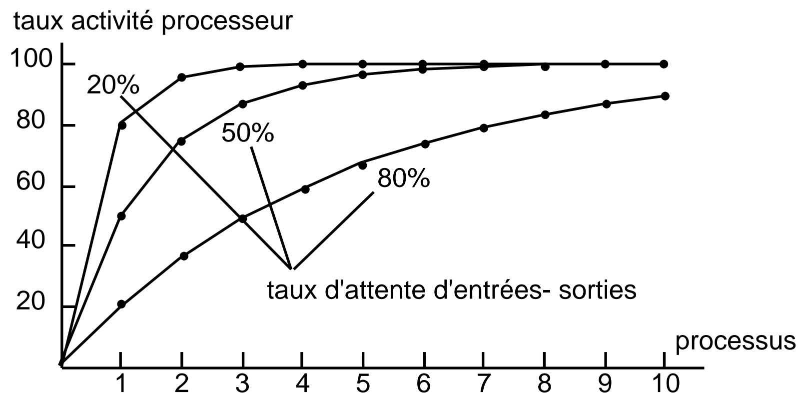



***Fig. 14.1.*** *Taux d'activité du CPU en fonction du nombre de processus.*

La justification de ces courbes est probabiliste. Le taux d'attente d'entrées-sorties peut s'interpréter comme la probabilité que le processus soit en attente d'entrées-sorties à un instant donné. Si le degré de multiprogrammation est n, la probabilité pour que tous les processus soient en attente d'entrées-sorties est alors $p^n$. Il s'ensuit que la probabilité pour qu'un processus au moins soit prêt, et donc que le processeur soit actif est: `taux_processeur = 1 - p`$^n$. Ces courbes sont le reflet de cette relation.

Notons que lorsque le taux d'activité est de 20%, par exemple dans le cas d'une alternance de 10 ms de processeur et de 40 ms d'entrées-sorties, il faut 10 processus pour atteindre un taux d'activité de 90%. Or les applications de gestion ont souvent un taux d'activité inférieur à une telle valeur, ce qui justifie dans ce cas un degré de multiprogrammation important pour avoir une bonne utilisation du processeur.

## 14.1.2. Les conséquences de la multiprogrammation

La multiprogrammation a des conséquences sur la gestion de la mémoire centrale. On peut distinguer trois problèmes que doit résoudre le système.

- Définir un espace d'adresses par processus. Chaque processus doit conserver son indépendance. Il doit pouvoir créer des objets dans son espace d'adresses et les détruire, sans qu'il y ait interférence avec les autres processus. Les caractéristiques du matériel sont importantes, car elles offrent plus ou moins de souplesse pour cette définition.
- Protéger les processus entre eux. Il est nécessaire que “chacun reste chez soi”, du moins pour les objets qui leur sont propres. Il faut donc pouvoir limiter les actions des processus sur certains objets, en particulier interdire les accès à ceux qui ne leur appartiennent pas.
- Attribuer un espace de mémoire physique. Comme un objet n'est accessible au processeur que s'il est en mémoire physique, il faut “allouer” de la mémoire physique aux processus, et assurer la traduction d'une adresse d'un objet dans son espace propre en l'adresse en mémoire physique où il est situé.

Il faut bien distinguer l'espace des adresses d'un processus de l'espace de mémoire physique. Si dans certains cas, il y a une certaine identité, comme dans le partitionnement, nous verrons avec la segmentation que les structures de ces deux espaces peuvent être très différentes.

## 14.1.3. Les difficultés du partitionnement

Considérons une mémoire centrale linéaire de N emplacements. L'espace des adresses physiques est dans ce cas l'ensemble des entiers compris entre 0 et N-1. La première méthode pour résoudre les problèmes ci-dessus a été de découper cet espace des adresses physiques en zones disjointes, que l'on appelé *partitions*, et de construire ainsi des espaces d'adresses disjoints pour plusieurs processus.

- Lors de sa création, un processus dispose de l'un de ces espaces, par exemple la partition [A..B]. Il est libre de placer les objets comme il l'entend à l'intérieur de cet espace. Si un autre processus est présent en mémoire en même temps que lui, celui-ci disposera d'un autre espace disjoint du précédent, par exemple la partition [C..D], telle que l'on ait $B<C$ ou $D<A$.

- La protection peut être obtenue simplement en interdisant au processus de construire une adresse en dehors de sa partition. Par exemple, le processeur se déroutera vers le système si, lorsqu'il exécute une instruction pour un processus dont l'espace des adresses est la partition [A..B], il génère une adresse en dehors de cette partition.
- L'allocation de mémoire physique est en fait assez simple, puisqu'un processus dont l'espace des adresses est la partition [A..B] reçoit la portion de mémoire physique comprise entre les adresses A et B.

On peut définir un partitionnement fixe, par exemple au lancement du système, les partitions n'étant pas forcément toutes de la même taille, comme le montre la figure 14.2. Il est possible de définir a priori la partition dont dispose un processus donné. L'allocation de mémoire physique est alors obtenue en associant une file d'attente par partition, les processus d'une même partition étant amenés les uns après les autres dans la portion correspondante de mémoire physique. Cette méthode présente l'inconvénient qu'une partition peut être libre (pas de processus en mémoire ni dans la file d'attente), alors que d'autres sont surchargées, entraînant une mauvaise utilisation de la mémoire et du processeur.

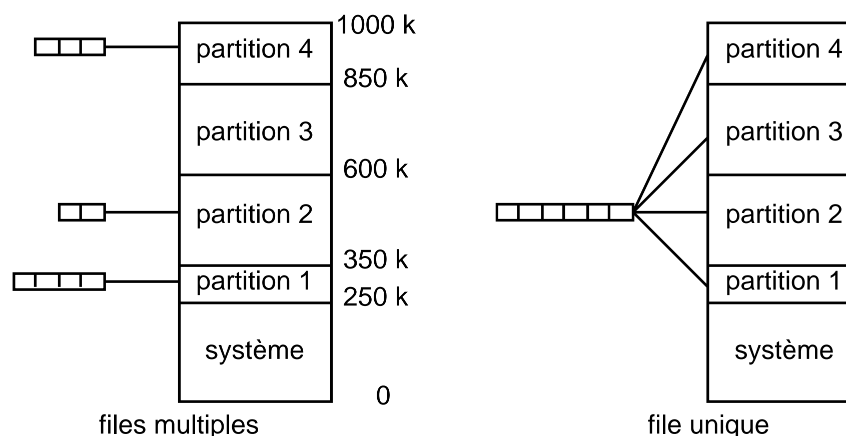



***Fig. 14.2.*** *Gestion mémoire par partitionnement.*

En utilisant un chargeur translatable, il est possible de retarder l'attribution de la partition au processus. On dispose alors d'une seule file d'attente de processus. Lorsqu'une partition est libre, le système recherche un processus de la file qui peut se satisfaire de la taille de cette partition, et le charge en mémoire dans cette partition. La recherche peut s'arrêter sur le premier trouvé, ou rechercher celui qui a le plus gros besoin d'espace, tout en étant inférieur à la taille de la partition; notons que ceci peut entraîner la famine d'un processus qui ne demanderait qu'un tout petit espace.

Un inconvénient du partitionnement fixe réside dans une mauvaise utilisation de la mémoire physique, dans la mesure où un processus reçoit une portion de mémoire plus grande que nécessaire. Il est possible de rendre variables les tailles des partitions, un processus ne recevant que ses besoins stricts. L'espace alloué à un processus devant être contigu, on retrouve alors les mêmes inconvénients que ceux rencontrés pour l'allocation par zone des objets externes sur disque. Cette méthode est souvent utilisée dans les micro-ordinateurs, pour la mise en mémoire de plusieurs “programmes”, tout en n'ayant toujours qu'un seul de ces programmes qui s'exécute.

# 14.2. La notion de mémoire segmentée

Un espace d'adresse linéaire pour un processus est parfois une gène, car cela complique la gestion des objets dans cet espace par le processus. Cette difficulté est accrue si le processus désire partager des données ou du code instruction avec un autre processus. Dans ce cas, comment chacun de ces processus va-t-il désigner ces objets partagés dans son propre espace? Si l'espace est assez grand, les processus peuvent, par exemple, décider a priori de réserver des parties de leur espace à de tels objets.

Pour éviter ces inconvénients, on a introduit la notion de *mémoire segmentée*, qui est une sorte d'espace à deux dimensions (figure 14.3). Chaque processus dispose de son propre espace mémoire constitué d'un ensemble de *segments*, chaque segment étant un espace linéaire de taille limitée. Un

emplacement de cet espace est désigné par un couple `<s, d>` où `s` désigne le segment, et `d` un déplacement à l'intérieur du segment. Un segment regroupe en général des objets de même nature (instructions, données ou constantes, partagées ou propres, etc...). On retrouve ici une notion déjà rencontrée lors de l'édition de liens: un module est constitué de sections, et l'éditeur de liens regroupe ensemble les sections de même nature.

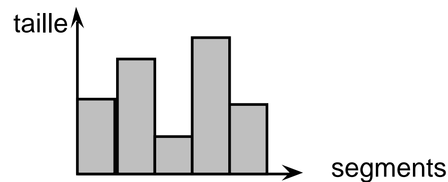



***Fig. 14.3.*** *Représentation d'une mémoire segmentée.*

Cette notion de mémoire segmentée peut ou non être prise en compte par le matériel. Lorsqu'elle ne l'est pas, l'éditeur de liens ou le chargeur place les segments dans un espace linéaire, avant de lancer l'exécution du processus. Ce placement est alors statique, et la taille des segments est souvent invariable.

Certains matériels prennent en compte un tel espace, en assurant dynamiquement la traduction d'une adresse segmentée en une adresse dans un espace linéaire, au moyen de tables spécifiques qui contiennent les informations de localisation de chaque segment, encore appelés *descripteurs de segment*. Un tel descripteur contient les informations suivantes:

- L'indicateur de *validité* indique si le segment existe ou non.
- L'*adresse de début* indique où commence le segment dans l'espace linéaire.
- La *taille du segment* indique le nombre d'emplacements du segment. Le processus doit donner des déplacements dans le segment qui soient inférieurs à la taille.
- Les *droits* indiquent les opérations du processus sur le segment qui sont autorisées.

La figure 14.4 montre le fonctionnement de l'adressage segmenté. Lorsque le processeur interprète une adresse segmentée `<s, d>` pour le compte d'un processus, il compare d'abord la valeur de `s` avec le champs `LT` de son registre spécialisé qui définit la table des descripteurs de segments du processus. Si la valeur est acceptable, il lit en mémoire ce descripteur, dont il vérifie l'indicateur de validité puis contrôle que les droits sont suffisants. Il compare ensuite `d` avec le champs `taille` de ce descripteur. Si la valeur est acceptable, il ajoute `d` à l'adresse de début du segment et obtient ainsi l'adresse de l'emplacement dans la mémoire linéaire. En général, pour éviter l'accès mémoire supplémentaire pour obtenir le descripteur de segment, le processeur dispose de registres spécialisés qui contiennent les descripteurs des derniers segments accédés par le processus.

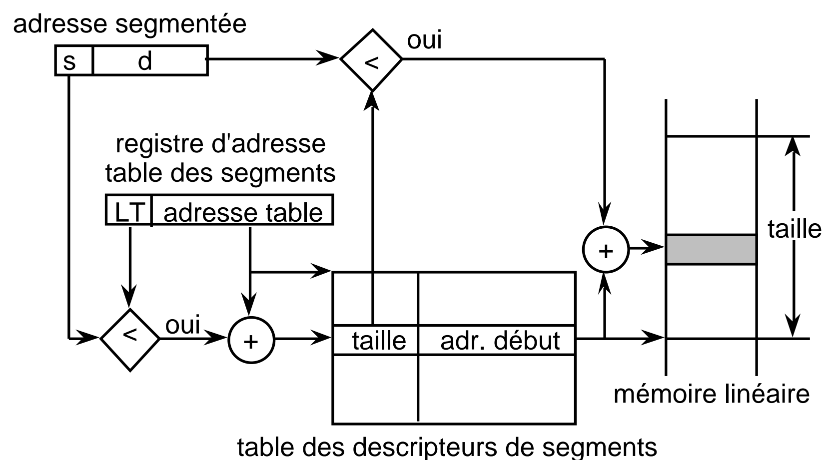



***Fig. 14.4.*** *Fonctionnement de l'adressage segmenté.*

La définition du contenu de la table des descripteurs de segments est à la charge du système, qui doit allouer l'espace dans la mémoire linéaire lors de la création des segments. L'allocation est une allocation par zone contiguë. Mais, notons que le passage systématique par la table des descripteurs

de segments permet de déplacer un segment dans la mémoire linéaire, puisqu'il suffit ensuite de modifier le champs `adresse de début` du descripteur pour permettre au processus de continuer son fonctionnement après ce déplacement.

Voici quelques exemples de processeurs disposant de l'adressage segmenté:

- Dans Multics, un processus peut avoir 32768 segments, chacun des segments pouvant atteindre 256 Kmots de 36 bits.
- Dans l'intel iAPX286 (le processeur des PC-AT), la table des descripteurs de segments d'un processus est divisée en deux parties, une table des segments communs à tous les processus, et une table propre à chaque processus. Chacune de ces tables peut avoir 8192 segments, et chaque segment peut avoir jusqu'à 64 Koctets. La mémoire linéaire est la mémoire physique, et peut atteindre 16 Moctets.
- L'intel iAPX386 (et suivants) est assez voisin de l'iAPX286, du point de vue adressage segmenté, si ce n'est que les segments peuvent atteindre 4 Goctets. La mémoire linéaire peut atteindre 4 Goctets et peut, de plus, être paginée (voir ci-dessous).

# 14.3. Le mécanisme de pagination

Le mécanisme de pagination a été imaginé pour lever la contrainte de contiguïté de l'espace de mémoire physique allouée aux processus. Pour cela, la mémoire physique est découpée en blocs de taille fixe, que nous appellerons *case* (en anglais *frame*). Par ailleurs, la mémoire linéaire des processus, encore appelée *mémoire virtuelle*, est elle-même découpée en blocs de taille fixe, que nous appellerons *page*. La taille d'une page correspond à la taille d'une case. Chaque page peut alors être placée dans une case quelconque. Le *mécanisme de pagination* du matériel assure la traduction des adresses de mémoire virtuelle en une adresse de mémoire physique, de façon à retrouver la contiguïté de cette mémoire virtuelle.

## 14.3.1. La pagination à un niveau

Le mécanisme de traduction des adresses virtuelles en adresses réelles doit associer à chaque numéro de page virtuelle le numéro de case réelle qui contient cette page, si elle existe. La figure 14.5 montre le fonctionnement de ce mécanisme.

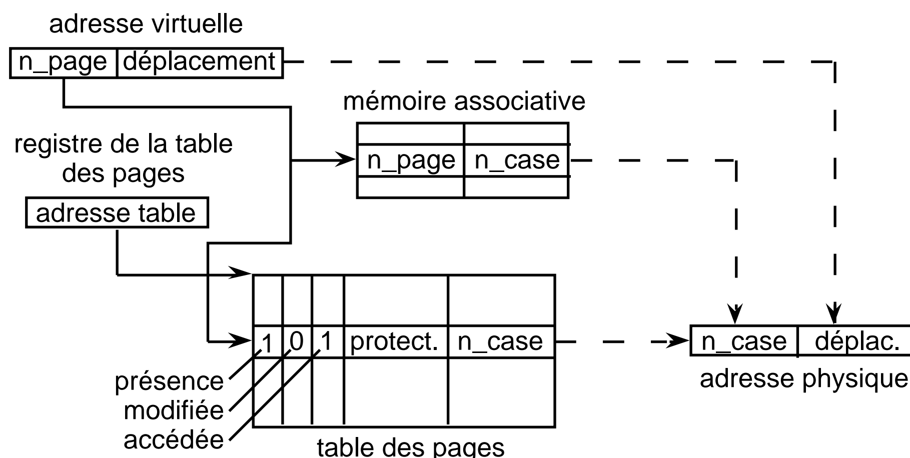



***Fig. 14.5.*** *Fonctionnement de la pagination à un niveau.*

Un registre spécialisé du processeur contient l'adresse de la table qui mémorise cette association; cette table est appelée la *table des pages* du processus. En général, l'adresse de cette table est une adresse physique. Chaque entrée de cette table contient les informations suivantes:

- L'indicateur de *présence* indique s'il y a une case allouée à cette page.
- Le *numéro de case* alloué à cette page.
- Les indicateurs de *protection* de la page indique les opérations autorisées sur cette page par le processus.
- L'indicateur de *page accédée* est positionné lors d'un accès quelconque.

- L'indicateur de *page modifiée* est positionné lors d'un accès en écriture.

Ces deux derniers indicateurs servent à la gestion de la mémoire physique que nous verrons par la suite.

Lorsque le processeur traduit une adresse virtuelle, il en isole le numéro de page virtuelle qu'il utilise comme déplacement dans cette table des pages pour en trouver l'entrée correspondante. Si l'indicateur de présence est positionné, il contrôle la validité de l'accès vis à vis des indicateurs de protection. Si l'accès est accepté, il concatène le numéro de case au déplacement virtuel dans la page pour obtenir l'adresse en mémoire physique de l'emplacement recherché. Pour éviter l'accès mémoire supplémentaire à la table des pages, une mémoire associative contient les entrées de la table correspondant aux derniers accès.

VAX-VMS utilise un tel mécanisme de pagination à un niveau, avec des pages de 512 octets. Mais pour éviter d'avoir des tailles trop importantes pour la table des pages, la mémoire virtuelle est découpée en trois régions, pouvant comporter chacune jusqu'à 1 Goctets. Chacune de ces régions dispose de sa propre table, et les registres correspondants contiennent leur taille effective.

- La région système est commune à tous les processus. L'adresse de sa table de pages est une adresse de mémoire physique.
- Les régions P0 et P1 sont propres au processus, et leur table des pages est dans la région virtuelle système. L'adresse de leur table de pages respective est donc une adresse virtuelle. Les pages de la région P0 sont dans les adresses virtuelles basses, alors que celles de la région P1 sont dans les adresses virtuelles hautes. L'augmentation de la taille de la région P0 se fait donc vers les adresses virtuelles croissantes, alors que celle de la région P1 se fait vers les adresses décroissantes (figure 14.6).

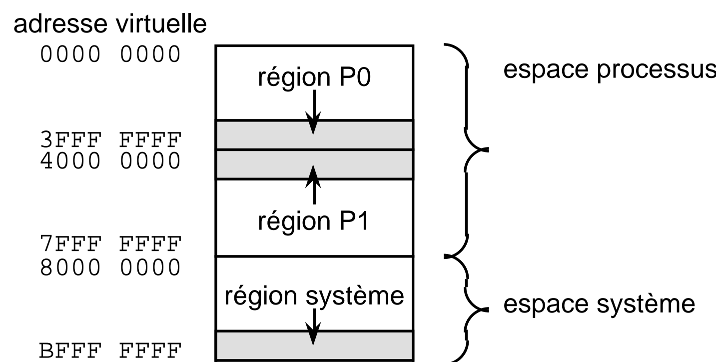



***Fig. 14.6.*** *L'espace virtuel de VAX-VMS.*

## 14.3.2. La pagination à deux niveaux

La pagination à un niveau d'un espace virtuel de grande taille peut conduire à des tailles importantes de la table des pages. L'exemple précédent de VAX-VMS montre une approche pour réduire cet encombrement, mais impose des contraintes sur la gestion des objets à l'intérieur même de cette mémoire virtuelle, puisqu'il faut qu'ils se trouvent au début ou à la fin de l'espace du processus. La pagination à deux niveaux a pour but de réduire la représentation de la table des pages d'un processus, sans avoir des contraintes aussi fortes.

La figure 14.7 montre le fonctionnement de ce mécanisme. La mémoire virtuelle est divisée en *hyperpages* de taille fixe, chaque hyperpage étant découpée en pages de même taille qu'une case de mémoire physique. Une hyperpage contient donc un nombre limité de pages. À chaque hyperpage, on associe une table de pages analogue à ce qui précède. Par ailleurs, la mémoire virtuelle d'un processus est représentée par une table des hyperpages, qui permet de localiser la table des pages de chaque hyperpage. Le gain de la représentation est obtenu par l'indicateur de présence d'hyperpage qui précise l'existence ou non de la table des pages de cette hyperpage.

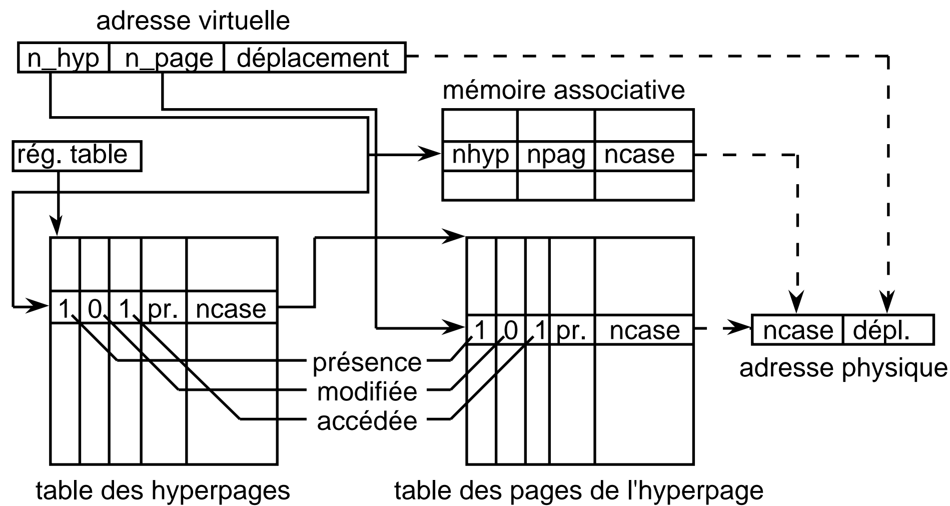



***Fig. 14.7***. *Fonctionnement de la pagination à deux niveaux.*

Lorsque le processeur traduit une adresse virtuelle, il isole d'abord le numéro d'hyperpage qu'il utilise comme index dans la table des hyperpages (dont l'adresse est dans un registre spécialisé) pour obtenir l'entrée associée à cette hyperpage. Si l'indicateur de présence est positionné, il vérifie que les indicateurs de protection de l'ensemble de l'hyperpage autorisent l'accès désiré. Puis le processeur isole dans l'adresse virtuelle le numéro de page qu'il utilise comme index dans la table des pages de l'hyperpage pour obtenir l'entrée associée à cette page. Il poursuit avec cette entrée comme pour la pagination à un niveau. Comme précédemment, les accès mémoires supplémentaires peuvent être évités par l'utilisation d'une mémoire associative qui conserve les entrées de pages (repérées par leur numéro d'hyperpage et de page) correspondant aux derniers accès.

Voici quelques exemples de machines ayant la pagination à deux niveaux:

- L'IBM 370 dispose de plusieurs configurations possibles. Sur les machines à mémoire virtuelle sur 16 Moctets, on peut avoir, par exemple, 256 hypergages de 16 pages de 4096 octets.
- L'iAPX386 structure la mémoire virtuelle (appelée aussi mémoire linéaire lors de l'étude de la segmentation) en 1024 hyperpages de 1024 pages de 4096 octets.

## 14.3.3. La segmentation vis à vis de la pagination

La segmentation et la pagination sont deux notions différentes qui sont utilisées conjointement sur certaines machines (par exemple, Multics, iAPX386). La segmentation doit être vue comme une structuration de l'espace des adresses d'un processus, alors que la pagination doit être vue comme un moyen d'adaptation de la mémoire virtuelle à la mémoire réelle.

- Avec la segmentation, le processus dispose d'un espace des adresses à deux dimensions que le processeur transforme en une adresse dans une mémoire linéaire. Sans la segmentation, le processus dispose directement de cette mémoire linéaire.
- Avec la pagination, on dispose d'une fonction de transformation dynamique des adresses de la mémoire linéaire en adresses de la mémoire physique, qui permet de placer les pages de la mémoire linéaire dans des cases quelconques de mémoire physique. Sans la pagination, les pages de la mémoire linéaire doivent être placées dans les cases de la mémoire physique de même numéro.

Lorsqu'on dispose de la segmentation, les segments doivent être alloués dans la mémoire linéaire en utilisant les techniques d'allocation par zone (portion de mémoire contiguë de taille donnée). Lorsqu'on dispose de la pagination, les pages de la mémoire linéaire doivent être allouées dans la mémoire physique. Il peut sembler complexe de devoir mettre en œuvre les deux notions. Cependant, elles sont complémentaires, et la pagination simplifie l'allocation par zone de la segmentation. D'une part, il peut alors y avoir des pages libres au milieu de pages occupées dans la mémoire linéaire, sans perte de place en mémoire physique. D'autre part, le déplacement d'une page de la mémoire linéaire dans une autre page, peut être obtenue sans déplacement dans la mémoire physique, puisqu'il suffit de modifier les entrées correspondantes de la table des pages.

Notons que le partage de données entre deux processus peut être obtenu de différentes façons. Par exemple, il peut être obtenu au niveau des pages de leur mémoire linéaire, en allouant la même case de mémoire physique aux deux processus, au même instant. Il peut être obtenu au niveau des hyperpages de leur mémoire linéaire, en allouant la même table des pages à deux hyperpages de ces processus. Sans pagination, il peut être obtenu au niveau segment en allouant la même zone de mémoire physique à deux segments de chacun des deux processus.

## 14.3.4. Le principe de la pagination à la demande

Notons tout d'abord que la mémoire virtuelle telle qu'elle a été présentée ci-dessus, peut être de taille très importante, et beaucoup plus grande que ce que peut être une taille raisonnable de mémoire physique. Dans un contexte de multiprogrammation, il peut être nécessaire de disposer de plusieurs mémoires virtuelles. Pour résoudre cette difficulté, on a imaginé de ne mettre en mémoire physique que les pages de mémoire virtuelle dont les processus ont besoin pour leur exécution courante, les autres étant conservées sur mémoire secondaire (disque). Lorsqu'un processus accède à une page qui n'est pas en mémoire physique (indicateur de présence non positionné), l'instruction est interrompue, et un déroutement au système est exécuté. On dit qu'il y a *défaut de page*. Le système doit alors allouer une case à cette page, lire la page depuis la mémoire secondaire dans cette case et relancer l'exécution de l'instruction qui pourra alors se dérouler jusqu'à son terme.

Lorsqu'un défaut de page se produit, s'il y a une case libre, le système peut allouer cette case. Si aucune case n'est libre, il est nécessaire de réquisitionner une case occupée par une autre page. On dit qu'il y a *remplacement* de page. Plusieurs algorithmes de remplacement de pages ont été proposés qui diffèrent par le choix de la page remplacée. Si celle-ci a été modifiée, il faut auparavant la réécrire en mémoire secondaire.

- L'*algorithme optimal* consiste à choisir la page qui sera accédée dans un avenir le plus lointain possible. Évidemment un tel algorithme n'est pas réalisable pratiquement, puisqu'il n'est pas possible de prévoir les accès futurs des processus. Il a surtout l'avantage de la comparaison: c'est en fait celui qui donnerait la meilleure efficacité.
- L'*algorithme de fréquence d'utilisation* (encore appelé *LFU* pour *least frequently used*) consiste à remplacer la page qui a été la moins fréquemment utilisée. Il faut maintenir une liste des numéros de page ordonnée par leur fréquence d'utilisation. Cette liste change lors de chaque accès mémoire, et doit donc être maintenue par le matériel.
- L'*algorithme chronologique d'utilisation* (encore appelé *LRU* pour *least recently used*) consiste à remplacer la page qui n'a pas été utilisée depuis le plus longtemps. Il faut maintenir une liste des numéros de pages ordonnée par leur dernier moment d'utilisation. Cette liste change lors de chaque accès mémoire, et doit donc être maintenue par le matériel.
- L'*algorithme chronologique de chargement* (encore appelé *FIFO* pour *first in first out*) consiste à remplacer la page qui est en mémoire depuis le plus longtemps. Sa mise en œuvre est simple, puisqu'il suffit d'avoir une file des numéros de pages dans l'ordre où elles ont été amenées en mémoire.
- L'*algorithme aléatoire* (encore appelé *random*) consiste à tirer au sort la page à remplacer. Sa mise en œuvre est simple, puisqu'il suffit d'utiliser une fonction comme `random`, qui est une fonction standard de presque tous les systèmes.

D'autres algorithmes ont été construits qui sont des compromis entre le matériel et le logiciel.

- L'*algorithme de la seconde chance* est dérivé de l'algorithme FIFO et utilise l'indicateur de page `accédée`. Lors d'un remplacement, si la dernière page de la file a son indicateur positionné, elle est mise en tête de file, et son indicateur est remis à zéro (on lui donne une seconde chance), et on recommence avec la nouvelle dernière, jusqu'à en trouver une qui ait un indicateur nul et qui est alors remplacée. Cela évite de remplacer des pages qui sont utilisées pendant de très longues périodes.
- L'*algorithme de non récente utilisation* (encore appelé *NRU* pour *not recently used*), est dérivé de LRU, et utilise les indicateurs de page `accédée` et de page `modifiée` de la table des pages, pour définir six catégories de pages (figure 14.8). Lorsque les pages sont dans les catégories `C1` à `C4`, elles sont présentes dans la table des pages, et le matériel gérant les indicateurs peut les faire changer de catégories (transitions marquées `*` sur la figure). Lorsqu'elles sont dans les catégories `C5` et `C6`, elles ne sont pas présentes dans les tables; un accès à une telle page provoquera donc un

défaut de page. Périodiquement, le système transfère les pages des catégories C1 à C4 dans la catégorie qui est située immédiatement à droite sur la figure. De plus, il lance de temps en temps la réécriture sur mémoire secondaire de celles qui sont dans C5. Lorsqu'une telle réécriture est terminée, si la page est toujours dans C5, elle est transférée dans C6. Lorsqu'un défaut de page survient, le système recherche d'abord si la page demandée n'est pas dans C5 ou C6, pour la remettre alors dans la table, et ainsi la transférer dans C3 ou C4 suivant le cas (le matériel la mettra ensuite dans C1 ou C2). Si elle n'y est pas, il y a remplacement de la première page trouvée dans une catégorie, par numéro décroissant, c'est-à-dire, en commençant par C6.

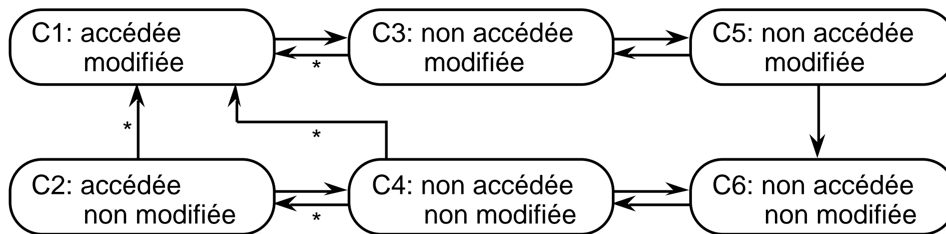



***Fig. 14.8.*** *Les catégories de l'algorithme NRU. Les transitions marquées * sont provoquées par le matériel.*

On peut mesurer l'efficacité d'un algorithme de remplacement par la chance que la page soit présente en mémoire lors d'un accès, ou encore par la probabilité d'avoir un défaut de page lors d'un tel accès. On constate que les algorithmes se comportent presque tous assez bien et de la même façon. Évidemment l'algorithme optimal est le meilleur, suivi par les algorithmes LRU et LFU, ensuite par NRU et la seconde chance, et enfin par FIFO et aléatoire. Mais cette probabilité est beaucoup plus influencée par le nombre de cases attribuées à un processus que par l'algorithme lui-même.

# 14.4. La notion d'espace de travail

Si un processus a en mémoire toutes les pages dont il a besoin pour s'exécuter pendant une période suffisante, la probabilité d'avoir un défaut de page est nulle. Au contraire, si toutes les pages ne sont pas présentes, il y aura des défauts entraînant des remplacements de pages qui peuvent redevenir nécessaires par la suite. En d'autres termes, si un processus a un nombre de cases suffisant, presque toutes les pages dont il a besoin seront en mémoire, il y a peu de défaut. Si ce nombre tombe en dessous d'un seuil dépendant du processus, la probabilité de défaut augmente brutalement. Or chaque défaut de page bloque le processus en attendant que la page soit présente en mémoire. On augmente ainsi son temps global d'attente d'entrées-sorties, et nous avons vu au début du chapitre l'influence de ce paramètre sur le taux d'utilisation du processeur.

Un processus, qui fait au total N accès à la mémoire, avec une probabilité de défaut de pages p, sera bloqué pour défaut de page, pendant Td=pNT, si T est le temps pour lire une page. Comme, par ailleurs, on peut évaluer son temps de processeur à Tuc=Nt, où t est le temps pour un accès à la mémoire physique, on en déduit que Td=p(T/t)Tuc. Or T/t est environ de $10^4$ actuellement, ce qui montre que p doit être maintenu à des valeurs très faibles sous peine d'avoir un temps Td très grand par rapport à Tuc, entraînant une augmentation du taux d'attente d'entrées-sorties du processus, et par voie de conséquence une diminution du taux d'utilisation du processeur. Ce phénomène, assez brutal du fait des coefficients, est appelé l'*écroulement* du système (en anglais *thrashing*).

On appelle *espace de travail* (en anglais *working set*) les pages dont a besoin un processus, à un instant donné, pour s'exécuter. Le paragraphe précédent montre l'intérêt d'évaluer la taille de cet espace de travail, puisqu'elle correspond au nombre de cases nécessaires au processus. En dessous, il y aura des défauts de pages, au-dessus la mémoire physique sera mal utilisée. Si T(P) est la taille de cet espace pour le processus P, il faut alors déterminer l'ensemble E des processus que l'on peut mettre en mémoire de telle sorte que l'on ait:

$$\sum_{P \in E} T(P) \leq M$$

où M est la taille totale de la mémoire disponible. VAX-VMS utilise cette notion pour décider quels sont les processus qui sont résidents en mémoire centrale, et ceux dont l'état est temporairement conservé sur mémoire secondaire.

# 14.5. Conclusion

☞ La multiprogrammation a pour but d'améliorer l'efficacité du processeur par la présence de plusieurs processus en mémoire de telle sorte qu'il y en ait presque toujours un qui soit prêt.

☞ Les trois problèmes à résoudre sont la définition de l'espace d'adresses des processus, la protection de leurs objets propres et l'attribution d'une partie de la mémoire physique à chacun d'eux.

☞ Dans le partitionnement, les processus reçoivent une partie contiguë de la mémoire physique qui constitue leur espace d'adresses. Cela conduit à une mauvaise utilisation de la mémoire physique.

☞ La mémoire segmentée fournit aux processus un espace d'adresses à deux dimensions leur permettant de regrouper ensembles des objets de même nature, et de conserver cette structuration à l'exécution lorsque le matériel supporte ce mécanisme. Cela simplifie la gestion de leur mémoire virtuelle.

☞ La pagination est un mécanisme qui permet de plaquer la mémoire virtuelle sur la mémoire réelle, en les découpant respectivement en pages et en cases de même taille, et en mettant les pages dans n'importe quelles cases sans devoir respecter la contiguïté.

☞ La pagination peut être à un niveau, une table des pages assurant la correspondance entre les pages et les cases. La table des pages peut être réduite, par une pagination à deux niveaux, en découpant la mémoire virtuelle en hyperpages elles-mêmes découpées en pages, une table des hyperpages repérant la table des pages de chacune des hyperpages.

☞ La segmentation et la pagination peuvent être utilisées conjointement puisque ce sont des mécanismes qui s'adressent à deux problèmes différents.

☞ La pagination à la demande consiste à n'allouer une case à une page que lorsqu'un processus en a besoin. Cela implique de mettre en œuvre un algorithme de remplacement lorsque toutes les cases sont occupées. Ces algorithmes peuvent être uniquement logiciels ou utiliser certains dispositifs matériels. Ils se comportent presque tous assez bien.

☞ Chaque processus a besoin d'un ensemble de pages pour s'exécuter, et qui constitue son espace de travail. Le nombre de cases dont dispose un processus doit être égal à la taille de cet espace de travail. S'il est plus grand, la mémoire est mal utilisée. S'il est plus petit, le nombre de défaut de pages augmente, ce qui peut conduire à l'écroulement du système.

# 15

# Le langage de commandes

Le langage de commandes est à la base de la communication entre un utilisateur et le système. Son rôle essentiel est de permettre à un tel utilisateur de lancer l'exécution de programmes. Nous avons vu qu'un programme s'exécutait dans un certain environnement, et pouvait manipuler des objets externes. Le langage de commande doit donc aussi permettre de définir cet environnement et les liaisons avec ces objets externes. Le besoin de contrôler les différentes opérations faites par ces programmes entraîne la nécessité de pouvoir identifier l'utilisateur qui lance les commandes.

## 15.1. Le langage issu du traitement par lot

La forme syntaxique de ce langage varie suivant les constructeurs et le mode d'utilisation de la machine. À l'origine, ce mode étant uniquement le traitement par lot à partir de cartes perforées, le langage présentait une forme très rudimentaire. Même si les cartes ne sont plus utilisées de nos jours, le langage pour ce mode en est resté fortement imprégné. Les lignes contenant une phrase du langage de commandes commencent par un (ou plusieurs) caractère spécial, comme '$' ou '//'. Elles définissent la suite des *étapes* (*step*) d'un *travail* (*job*), chaque étape correspondant à l'exécution d'un programme particulier. La définition d'une étape consiste donc à définir le programme à exécuter, l'environnement d'exécution de ce programme, et les liaisons avec les objets externes. Les communications d'une étape à la suivante sont souvent inexistantes. Les utilisateurs doivent prévoir ces communications par le biais de fichiers. Ce langage comporte souvent un nombre réduit de commandes, chacune de ces commandes comportant par contre un nombre important de paramètres. De fait, trois commandes sont essentielles:

- La commande d'identification du travail a pour but de préciser l'identité du demandeur, et les ressources globales nécessaires.
- La commande de définition d'une étape définit le programme à exécuter et les paramètres de son environnement.
- La commande de définition d'une liaison permet de relier un flot particulier du programme de l'étape à un objet externe.

Le grand nombre de paramètres d'une commande condamne, en général, la définition des paramètres par position, comme dans les langages de programmation habituels. Elle se fait plutôt par le biais de mots clés, sous la forme d'une liste d'associations, où une association est un mot clé suivi de la valeur du paramètre correspondant. Chaque valeur d'un paramètre étant associée à un mot clé, l'ordre des paramètres n'a plus d'importance. Par ailleurs, l'interpréteur du langage peut donner des valeurs par défaut aux paramètres que la commande ne définit pas. En général, ces valeurs par défaut sont choisies par l'ingénieur système de l'installation de façon à correspondre aux valeurs les plus couramment demandées par les utilisateurs.

Dans certains langages, il est possible de définir des *macrocommandes*. Une macrocommande est désignée par un nom et représente une suite de commandes prédéfinies. Elle peut être paramétrée, ce qui permet de lui donner une certaine généralité. Lorsqu'elle est appelée, la suite de commande qu'elle représente est d'abord modifiée en fonction de ces paramètres, puis exécutée. Ce mécanisme permet de simplifier l'utilisation du langage de commande lorsqu'une même suite de commandes est fréquemment utilisée.

# 15.2. Le langage interactif

En mode interactif, l'utilisateur doit d'abord fournir au système son identité pour établir la connexion (*login*). Le système lance ensuite l'interpréteur de commandes qui va permettre à l'utilisateur de faire exécuter les commandes qu'il désire durant une période plus ou moins longue, que l'on appelle une *session*, et qui se terminera par une déconnexion (commande *logout*). La durée de cette session, et le fait de la présence de l'utilisateur au terminal implique une forme différente du langage de commandes.

## 15.2.1. Le message d'invite

Alors que, en traitement par lot, l'utilisateur doit définir l'ensemble des commandes et des données dans un fichier de travail avant de le soumettre au système, en mode interactif, il doit pouvoir prendre ses décisions au vu des résultats de chaque commande successive. En conséquence, il ne doit pas être contraint de fournir des informations à l'avance, mais uniquement lorsqu'elles sont effectivement nécessaires. La notion de fichier de travail a donc moins de sens. Il n'est plus nécessaire de distinguer une ligne de commande d'une ligne contenant des données par des caractères spéciaux comme précédemment, mais d'indiquer à l'utilisateur l'outil logiciel qui attend des informations de sa part. C'est ce que l'on appelle l'*invite* (en anglais le *prompt*), qui est constituée d'une chaîne de caractères qui est envoyée à l'utilisateur par l'outil lorsqu'il attend ces informations. Par exemple, l'interpréteur de commandes de VMS, comme celui d'Unix, envoie normalement un '$' en début de ligne lorsqu'il attend une nouvelle commande. Souvent l'invite est une chaîne paramétrable, permettant ainsi à l'utilisateur de personnaliser ce dialogue.

## 15.2.2. Un langage orienté verbe

La deuxième différence, avec le traitement par lot, est le besoin de plus de souplesse et de convivialité du langage lui-même. Il s'ensuit que, au lieu d'avoir peu de commandes avec un grand nombre de paramètres, l'orientation est plutôt vers un grand nombre de commandes ayant chacune peu de paramètres. On dit parfois que l'on a une syntaxe *orientée verbe*, car la commande est constituée d'un verbe, qui est le nom de la commande, suivi des paramètres éventuels de la commande. Certaines de ces commandes sont reconnues et exécutées directement par l'interpréteur. Les autres commandes sont considérées comme des demandes d'exécution d'un programme. L'interpréteur recherche alors le programme exécutable de ce nom, en demande le chargement et le lancement au système, en lui fournissant les paramètres de la commande. Ceci donne une caractéristique importante au langage de commandes qui est son extensibilité, puisque au fur et à mesure où de nouveaux programmes deviennent opérationnels, ils deviennent de nouvelles commandes du système.

## 15.2.3. La gestion de l'environnement

La troisième caractéristique du langage de commandes interactif concerne la gestion de l'environnement de l'utilisateur. Alors que, en traitement par lot, une session de travail consiste en un nombre réduit d'étapes qui s'effectuent chacune dans un environnement propre et souvent indépendant, en mode interactif, une session peut durer plusieurs heures, et donc être constituée d'un grand nombre de commandes. Il serait fastidieux pour l'utilisateur de devoir préciser complètement l'environnement de chacune de ces commandes. On peut définir l'*environnement* comme un ensemble d'informations propres à l'utilisateur dans sa session courante, qui sont gérées par l'interpréteur de commandes, qu'il utilise pour modifier les commandes et leurs paramètres préalablement à leur exécution, et qu'il peut transmettre en totalité ou en partie aux programmes correspondants. Nous avons déjà eu un aperçu de telles informations:

- Le message d'invite est en général défini par une variable de l'environnement. La personnalisation de ce message est obtenue par l'utilisateur en modifiant cette variable.
- Le répertoire de travail, dont nous avons parlé dans le chapitre sur la désignation des objets externes, fait partie de cet environnement. Il peut être changé à tout moment, et est communiqué aux programmes (commande `cd` de beaucoup de systèmes).
- Les règles de recherche, dont nous avons également parlé dans le chapitre 10, font aussi partie de l'environnement. En particulier, une variable de l'environnement définit par exemple la liste des noms absolus des répertoires dans lesquels l'interpréteur recherche les programmes exécutables correspondant aux noms des commandes (variable `PATH` d'Unix). L'utilisateur peut également définir d'autres variables de ce type, qui peuvent être transmises aux programmes pour leurs propres besoins de recherche. Par exemple un programme d'aide en ligne (*help* dans Multics ou VMS, *man* dans Unix) peut utiliser une telle variable pour localiser les fichiers de documentation des applications manipulées par l'utilisateur.
- Le type de terminal, utilisé dans la connexion, est souvent défini par une variable de l'environnement (variable `TERM` d'Unix). Elle peut être consultée par la méthode d'accès terminal virtuel ou par les programmes spécifiques (éditeurs pleine page).

### 15.2.4. Le traitement préliminaire de la commande

Pour simplifier la frappe des commandes par l'utilisateur, les interpréteurs de commandes effectuent souvent différents traitements sur la commande avant son exécution proprement dite.

Le *mécanisme de substitution* permet le remplacement dans la commande des variables de l'environnement par leur valeur courante. Par exemple, dans Unix, la variable `HOME` de l'environnement est initialisée par le nom absolu du répertoire courant de l'utilisateur lors de l'établissement de la connexion. `${HOME}` dans une commande sera remplacée par ce nom. Il peut également être utilisé pour construire la liste des noms de fichiers vérifiant un certain profil. Par exemple, `'*.c'` dans une commande sera remplacé par la liste des noms de fichiers du répertoire courant qui sont suffixés par `'.c'`.

Le *mécanisme d'abréviation* permet à l'utilisateur de donner une forme plus simple à certaines commandes ou suite de commandes dont il se sert souvent, en fixant certains paramètres et en permettant éventuellement une certaine forme de substitution. Ce mécanisme s'apparente à la macrosubstitution. Par exemple, l'abréviation suivante:

```
alias rm 'rm -i'
```

est très conseillée sur Unix ou ses dérivés. Elle change l'option par défaut de la commande `rm` de destruction d'un fichier, de façon à demander systématiquement la confirmation à l'utilisateur.

Considérons, dans un deuxième exemple, la définition de l'abréviation suivante:

```
alias save 'chmod u+w \!:1.sav; cp \!:1.c \!:1.sav; chmod u-w \!:1.sav'
```

Lors de la frappe de la commande `save truc`, le système constate l'utilisation de l'abréviation de nom `save`, avec le paramètre `truc`. Il substitue partout dans l'abréviation la chaîne `\!:1`, qui désigne le premier paramètre, par `truc` et obtient la suite de commandes suivante dont il lance l'exécution. Cette suite consiste à forcer d'abord l'autorisation d'écriture par le propriétaire du fichier `truc.sav` puis à copier le fichier `truc.c` dans `truc.sav` et à enlever l'autorisation d'écriture par le propriétaire sur ce fichier.

```
chmod u+w truc.sav; cp truc.c truc.sav; chmod u-w truc.sav
```

L'utilisateur se protège ainsi d'une modification involontaire de ses fichiers `*.sav`.

### 15.2.5. L'exécution de la commande

Nous avons déjà dit que l'exécution d'une commande, non reconnue par l'interpréteur, consistait à rechercher le fichier de même nom par les règles de recherche, à demander au système de charger le programme correspondant, à lui transmettre les paramètres et l'environnement, et enfin à lancer son exécution. Dans certains systèmes il y a de plus création d'un processus pour cette exécution. La plupart du temps, les paramètres sont les chaînes de caractères présentes dans la ligne de commande, et dont l'interprétation est laissée au programme. L'environnement peut comprendre

évidemment certaines variables dont nous avons déjà parlé, mais il doit définir l'ensemble du contexte d'exécution du programme. En particulier, le programme doit être capable d'accéder aux objets externes dont il a besoin. Si la définition des liaisons correspondantes ne font pas partie du programme, elles doivent lui être transmises par l'intermédiaire de l'environnement. La plupart des systèmes considèrent que l'environnement doit définir au moins trois liaisons:

- l'entrée standard où le programme lira ses données,
- la sortie standard où le programme écrira ses résultats,
- la sortie des erreurs où le programme écrira ses messages d'erreur.

En mode interactif, il est assez naturel que les trois liaisons correspondent au terminal de l'utilisateur. Cependant, il peut arriver que l'utilisateur désire que ce soit autrement. Par exemple, il désire que le programme lise ses données dans un fichier, ou écrive ses résultats dans un fichier. Le langage de commandes doit alors fournir un mécanisme de *redirection*. Dans Unix ou MS-DOS et leurs dérivés, ceci est obtenu non pas par une définition des liaisons dans l'environnement, mais en établissant ces trois liaisons préalablement au lancement du processus. De plus, comme Unix crée un processus par commande, l'interpréteur du langage de commandes permet la redirection de la sortie standard d'un processus vers un tube, et la redirection de l'entrée standard d'un autre processus depuis ce tube. Par exemple, la ligne de commandes

```
cat *.biblio | sort | uniq > biblio.tri
```

lance trois processus sur chacune des trois commandes `cat`, `sort` et `uniq`, la sortie du premier processus, qui est la concaténation des fichiers suffixés `.biblio` du répertoire courant, étant délivrée en entrée du deuxième, qui trie en ordre alphabétique les lignes de cette concaténation pour les envoyer au troisième, qui élimine les répétitions et envoie le résultat dans le fichier `biblio.tri`. Comme les sorties d'erreurs de ces processus ne sont pas redirigées, leurs éventuels messages d'erreurs seront envoyés au terminal.

Signalons enfin que bien souvent les programmes peuvent transmettre un code de retour à l'interpréteur de commande lorsqu'ils se terminent. Il est donc possible de tenir compte de ces codes de retour pour la poursuite de l'exécution d'une séquence de commandes. Ceci conduit à disposer, dans le langage de commandes, de structures de contrôle proches de celles des langages de programmation.

# 15.3. Les structures de contrôle du langage de commandes

Certains systèmes offrent un langage de commandes qui est un véritable langage de programmation, en ce sens qu'il possède des structures de contrôle complexes. Il n'est pas question évidemment de remplacer les langages de programmation classiques, mais de fournir des structures de contrôle qui soient appropriées aux objets manipulés par le langage de commandes, c'est-à-dire, essentiellement les fichiers et les processus. Nous mentionnerons deux structures de ce types et montrerons leur utilisation sur des exemples.

• La *commande conditionnelle* permet, en fonction du code de retour d'un programme d'exécuter l'une ou l'autre parmi deux suites de commandes. Par exemple, nous utilisons la commande `test -f toto` qui permet de savoir si le fichier `toto` existe comme fichier ordinaire dans le répertoire courant. S'il existe, on trace une ligne de tirets de séparation après ce qui y est déjà, et sinon on y écrit une première ligne d'indication. On y écrit ensuite la date courante suivie d'une nouvelle ligne de tirets. Noter l'utilisation de `>` pour une redirection et écriture en début de fichier, et `>>` pour une redirection en prolongement d'un fichier déjà existant sans perte de son contenu.

```
if test -f toto
     then echo "---------------------------------------------" >>toto
     else echo "Etats des resultats successif de test" > toto
fi
date >>toto
echo "---------------------------------------------" >>toto
```

• L'*itération bornée* permet d'exécuter un ensemble de commandes, sur une suite de valeurs prises dans une liste. Par exemple:

```
for i in toto truc bidule; do cp /usr/moi/$i /tous; done
```

exécute la commande `cp` pour toutes les valeurs possibles de la liste `toto truc bidule`. Ces trois fichiers sont donc recopiés depuis le répertoire `/usr/moi` dans le répertoire `/tous`.

Évidemment la liste peut être construite automatiquement par le mécanisme de substitution vu précédemment. Ainsi, pour faire la compilation de tous les fichiers contenant des modules sources en langage C du répertoire courant, il suffit d'exécuter:

```
for i in *.c; do cc -c $i; done
```

Il n'est pas question de développer ici un langage de commandes particulier, c'est pourquoi nous ne décrirons pas plus ces structures de contrôle. Il nous paraît plus important que le lecteur comprenne leur importance, et le gain qu'il peut en obtenir. La complexité des commandes que l'on peut avoir, conduit naturellement à la notion de programme de commandes, mémorisé dans un fichier, et exécutable en tant que tel, ce qui est un autre aspect de l'extensibilité du langage de commandes dont nous avons déjà parlé. Il nous semble avoir, dans ces deux exemples ci-dessus, montré la différence entre le langage de commandes et un langage de programmation. Le premier manipule surtout des fichiers sur lesquels il exécute des commandes, le second manipule des données sur lesquels il effectue des calculs.

## 15.4. Le langage à base de menus ou d'icônes

La convivialité du poste de travail est souvent grandement amélioré par l'utilisation d'un langage de commandes basé soit sur des menus déroulants soit sur des icônes, soit sur les deux. Par exemple, sur MacIntosh ou sur Atari, la copie de fichier peut être obtenue en déplaçant l'icône du fichier d'une fenêtre dans une autre au moyen de la souris. De même sous X-window, l'utilisateur peut construire ses propres menus déroulants pour ses commandes les plus courantes. Pour le moment, ce type de langage fonctionne très bien pour un nombre réduit de commandes souvent sans paramètres. Il est agréable et doit être systématiquement développé et utilisé pour les besoins courants, mais il ne peut probablement pas satisfaire l'ensemble des besoins des utilisateurs.

## 15.5. Conclusion

☞ Le langage issu du traitement par lot comporte peu de commandes, avec beaucoup de paramètres. Les commandes permettent l'identification du travail, sa décomposition en étapes et les définitions des liaisons avec les objets externes. Les paramètres sont définis par mots clés, avec des valeurs par défaut.

☞ En interactif, les outils logiciels se présentent aux utilisateurs par leur message d'invite.

☞ Le langage de commandes interactif est plus convivial, avec une syntaxe orientée verbe, définissant de nombreuses commandes ayant chacune peu de paramètres. Chaque commande étant en fait un programme, le langage est facilement extensible, par adjonction de nouveaux programmes.

☞ L'environnement est un ensemble d'informations propres à l'utilisateur dans sa session courante, qui peuvent être utilisées par l'interpréteur de commandes et les programmes pour adapter leur comportement aux besoins spécifiques actuels de l'usager (message d'invite, répertoire de travail, règles de recherche, type de terminal).

☞ Les commandes de l'utilisateur peuvent être simplifiées, l'interpréteur pouvant les développer par les mécanismes de substitution ou d'abréviation.

☞ L'environnement d'exécution d'un programme comporte la définition des liaisons avec les objets externes. Au moins trois liaisons doivent être définies pour l'entrée, la sortie et les erreurs. Si, en général, ces liaisons sont avec le terminal de l'utilisateur, il faut pouvoir les rediriger vers des fichiers.

☞ Les programmes peuvent transmettre, en fin d'exécution, un code de retour à l'interpréteur de commandes, qui peut en tenir compte lors de l'exécution d'un ensemble de commandes.

☞ Les langages de commandes modernes disposent de toutes les structures de contrôle des langages de programmation classiques. La différence essentielle réside dans les objets manipulés par ces deux types de langage.

☞ Le langage à base de menus déroulants ou d'icônes offre une convivialité remarquable pour les besoins les plus courants. Il ne peut couvrir l'ensemble des besoins, où des structures de contrôle élaborées sont nécessaires.

# Bibliographie


AHO A.; SETHI R.; ULLMAN J., *Compilateurs - principes, techniques et outils*, InterEditions, 1989.

BEAUQUIER J.; BERARD B., *Systèmes d'exploitation, concepts et algorithmes*, McGraw-Hill, 1990.

BERSTEL J.; PERROT J.-F., *Multics. Guide de l'usager*, Masson, 1986.

CARREZ C., *Structures de données en Java, C++ et Ada 95*, InterEditions, 1997.

CORNAFION, *Systèmes informatiques répartis*, Dunod, 1981.

CROCUS, *Systèmes d'exploitation des ordinateurs*, Dunod, 1975.

CUSTER H., *Au cœur du système de fichier de Windows NT*, Microsoft Press, 1994.

FELDMAN S.I., *Make - a program for maintaining computer programs*, Bell laboratories, 1978.

KERNIGHAN B.W.; RITCHIE D.M., *Le langage C*, Masson, 1984.

KRAKOWIAK S., *Principes des systèmes d'exploitation des ordinateurs*, Dunod, 1985.

LISTER A.M., *Principes fondamentaux des systèmes d'exploitation*, Eyrolles, 1984.

MONTAGNON J.-A., *Architecture des ordinateurs, tome 2, Systèmes d'exploitation et extensions du matériel*, Masson, 1987.

RIFFLET J.-M., *La programmation sous Unix*, McGraw-Hill, 1986.

RUSLING D.A., *The Linux Kernel*, [sur les sites linux, comme ftp.lip6.fr, pub/linux/sunsite/docs/linux-doc-project/linux-kernel/ftk-0.8-3.ps.gz], 1999.

SOLOMON D.A., *Inside Windows NT*, Microsoft Press, 1998.

TANENBAUM A., *Les systèmes d'exploitation - conception et mise en œuvre*, InterEditions, 1989.

TANENBAUM A., *Systèmes d'exploitation - systèmes centralisés - systèmes distribués*, InterEditions, 1994.

ZANELLA P.; LIGIER Y., *Architecture et technologie des ordinateurs*, Dunod, 1989.

*Inside MacOS*, [sur les sites Apple, comme developper.apple.com, techpubs/Files/], Apple Computer Inc., 1996.

*Introduction to the 80386 including the 80386 data-sheet*, Intel, 1986.

*Multics: Programmers reference manual*, CII-Honeywell-Bull, 1983.

*VAX-VMS: Architecture handbook*, Digital Equipment, 1986.

*VAX-VMS: System software handbook*, Digital Equipment, 1989.

A propos de NTFS, consulter le site: http://www.informatik.hu-berlin.de/~loewis/ntfs/

# Index


***6***

68000 15,42

***A***

accès direct à la mémoire 16,17,20,23

allocation par blocs de taille fixe 72,73,77

allocation par zone 70,71,75,77,96,99,125,127,129

amorce logicielle 4,5

analyse lexicale 34,35,39,60

analyse sémantique 36,37,39

analyse syntaxique 35,39

appel système 18,47

arborescence de répertoire 83,84,86,87,95,97,100

archiveur 59

arrière plan (travaux) 8,17

assembleur 15,33,34,39,42,44,51

autonome (programme) 10

avant plan (travaux) 8,17

***B***

bande magnétique 80,82

bibliothécaire 28

bibliothèque de modules objets 46,47

bloc de contrôle de données 67

boîte aux lettres 117,118

***C***

catalogue 82,96,97,98

chaîne de production de programme 27,28,29,30,60,91

chargeur 28,30,51,53,54,125,126

CICS 27

CMS 27

communication entre processus 113,116,121

compilateur 3,4,5,9,33,34,36,37,39,51

comportement dynamique d'un programme 55,60

compteur ordinal 14,17,19,23,108

conception assistée (mode d'utilisation) 25,26,27,29

contrôle de procédés 8,26,27

***D***

débit des travaux 5,6,11

déperdition 112

descripteur d'objet externe 83,102

diff 60

différé (mode d'utilisation) 25,26,27,29

DPS-8 27

***E***

écroulement du système 131,132

éditeur de liens 28,30,34,41,42,44,45,47,48,49,50,51,52,53,54,59,60,67,126

éditeur de texte 55

entrées-sorties 5,6,7,8,9,11,15,16,17,19,20,21,22,23,26,64,67,68,107,118,123,124,131

entrées-sorties par accès direct à la mémoire 16,21,22

entrées-sorties par processeur spécialisé 16

entrées-sorties programmées 15,19,20,21,23

environnement 18,29,30,51,79,86,87,133,134,135,136,137

espace de travail 131,132

espace libre 72,75,76,77,78,80,83,87,91
étapes (steps) 5,133
exclusion mutuelle 110,112,113,114,116
ext2fs 101,102
extension (d'un objet externe) 70,71,72,76,77,96,97,98

***F***

famine 113,120,121,122,125
FAT 72,73,76,91,95,96,102
fichier à accès aléatoire 66,68
fichier logique 65,67,68,72,79,95
fichier physique 65,67,80
fichier séquentiel 65,66,68,74
fichier spécial (périphérique Unix) 84
forme normale de Backus-Naur 35

***G***

génération de code 37,39

***I***

iAPX286 127
iAPX386 127,129
iAPX432 41
IBM370 47
implantation séquentielle avec extensions 71
IMS 27
indenteur 60
instruction 4,14,15,17,18,19,34,37,42,45,46,47,51,54,55,56,60,108,113,114,125,130
interactif (mode d'utilisation) 25,26,27,29,54,134,136,137
interblocage 113,118,119,120,121,122
interface homme-machine 9,10
interpréteur 28,30,33,34,39,107,133,134,135,136,137
interpréteur de commande 28,30,134,136,137
interruption 6,8,11,16,17,18,19,20,23,112
invite (prompt) 134,135,137
iRMX86 27,117

***L***

langage d'assemblage 33,37,38,39,51
langage de commandes 81,133,134,136,137
langage évolué 33,37,38,39,54,56,60,79,114
lien 42,43,44,45,46,47,48,49,50,51,52,79,81,86,89,90,98,102
lien à satisfaire (importé, externe) 42,43,44,45,46,48,49,50,52
lien physique (fichier) 86
lien symbolique (fichier) 86,102
lien utilisable (exporté, public) 42,43,44,45,46,47,48,49,50,52
ligne série asynchrone 19,22
Linux 73,76,101,103

***M***

machine virtuelle 29,30,34,63
macrogénérateur 57,60
make 57,58,59,60
masquage des interruptions 18,19,23
mémoire segmentée 125,126,132
mémoire virtuelle 9,49,127,128,129,130,132
metteur au point 28,54,55,60
module éditable 41
module objet 34,37,38,41,42,43,44,45,51,53,54,58
module source 34,35,37,38,41,43,44,54,55,56,57,58,60,107
module translatable 41
moniteur d'enchaînement des travaux 4,5,10
montage de volume 81,85

mot d'état programme 23,108

MS-DOS 27,47,72,76,81,83,84,86,91,136

multiprogrammation7,8,9,11,123,124,130,132

multi-volumes 82

MVS 27

***N***

NTFS 72,76,99,100,101,103

numéro virtuel de secteur 69

***O***

objets externes 28,29,30,63,64,65,67,68,69,70,72,73,74,79,80,82,83,85,86,89,91,94,95,96,98,101,110,125,133,135,136,137

optimisation de code 37

***P***

pagination 127,128,129,130,132

pagination à la demande 130,132

paragrapheur 28,60

parallélisme 6,107

partage d'informations 10

pas à pas (mise au point) 54,55,60

point d'arrêt (mise au point) 54,55,60

préprocesseur 56,57,60

primitive 118

processeur 6,8,9,11,13,14,15,16,17,18,19,20,21,22,23,25,26,27,29,46,63,100,107,108,109,110,111,112,113,114,118,123,124,125,126,127,128,129,131,132

processeur virtuel 108,112

processus 107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,135,136

programme autonome 10

programme exécutable 28,37,41,49,50,51,52,54,55,58,60,67,86,134

protection 5,7,11,89,90,91,93,94,96,102,125,128,129,132

protection par droits d'accès 89

protection par mot de passe 90,94

protocole de communication 22

pseudo-parallélisme 107

***Q***

quantum d'allocation 75,77

***R***

reconnaissance automatique de volume 81

recouvrement 49,50,52

redondance interne 91,94

références croisées 28,50,52

registres 14,17,23,47,108,109,110,112,126,128

règles de production 35,36

règles de recherche 86,87,135,137

répertoire 58,80,82,83,84,85,86,87,89,95,96,97,98,99,100,101,102,135,136,137

répertoire de travail 86,87,135,137

ressource 26,96,97,99,110,111,112,113,114,115,116,118,119,120,122,123

ressource critique 110,113,115

RT-11 27

***S***

sauvegarde 10,17,18,19,90,91,92,94,97

sauvegarde du contexte 17,18

sauvegarde incrémentale 92

sauvegarde périodique 90,91,92,94

schéma producteur-consommateur 116,117,121

sécurité 89,90,91,92,94,99

sémaphore 114,115,116,121

sous-programme réentrant 18

SPART 27

SPS-7 27

structures de contrôle du langage de commandes 136

superviseur d'entrées-sorties 5,6,7,11,23

système d'exploitation 8,11,18,23,25,27,63,101

## *T*

table de bits 76,78

temps de réponse 5,6,8,10,11,26,29,118,121

temps de traitement 26,29

temps partagé 9,11,17,25,26,27,29,92

temps réel 9,26,27,29,108

temps virtuel 108,118

trace d'un programme 60

traduction croisée 37,38,39

train de travaux 6

transaction 10,101,120

travail (job) 5,133

TSO 27

tube (pipe) 116,117,121

## *U*

UNIX 27,35,47,48,57,59,73,74,76,77,83,84,85,86,87,89,90,91,101,109,110,112,116,117,120,121,134,135,136

## *V*

va-et-vient 9

verrou 114,121

VFAT 95,96,99,103

VM 27

VMS 27,51,72,86,134,135

vol de cycle 16,23

volume 80,81,82,83,84,85,87,91,92,93,95,96,98,99,100,101,102