SCIENCES SUP

Cours et exercices corrigés

Écoles d'ingénieurs • IUT • Licence 1re, 2e et 3e années

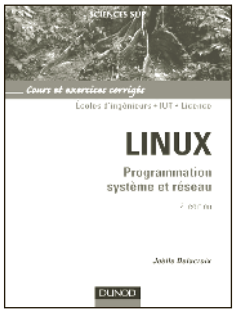
ARCHITECTURE DES MACHINES ET DES SYSTÈMES INFORMATIQUES

3e édition

Alain Cazes
Joëlle Delacroix

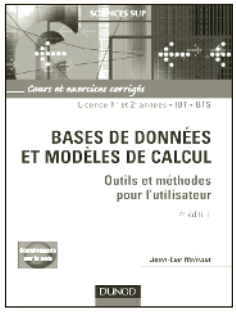
DUNOD

ARCHITECTURE DES MACHINES ET DES SYSTÈMES INFORMATIQUES



Linux
Programmation système et réseau
2e édition
Joëlle Delacroix
352 pages
Dunod, 2007

Bases de données et modèles de calcul
Outils et méthodes pour l'utilisateur
4e édition
Jean-Luc Hainaut
440 pages
Dunod, 2005



ARCHITECTURE DES MACHINES ET DES SYSTÈMES INFORMATIQUES

Cours et exercices corrigés

Alain Cazes
Maître de conférences en informatique
au Conservatoire National des Arts et Métiers

Joëlle Delacroix
Maître de conférences en informatique
au Conservatoire National des Arts et Métiers

3e édition

DUNOD

Illustration de couverture : *digitalvision*®

ISBN 978-2-10-053945-1

Table des matières

CHAPITRE 1 • **STRUCTURE GÉNÉRALE ET FONCTIONNEMENT D'UN ORDINATEUR** 1

1.1 Introduction 1

1.2 Structure et fonctionnement d'un ordinateur 3

1.2.1 Structure générale d'un ordinateur 3
1.2.2 La mémoire centrale 4
1.2.3 Le bus de communication 8
1.2.4 Le processeur central ou microprocesseur 10

1.3 Fonctionnement : relation microprocesseur / mémoire centrale 13

1.4 Un exemple 15

1.4.1 Le problème 15
1.4.2 L'ordinateur 15
1.4.3 Le langage machine 15

1.5 Les unités d'échanges 16

1.6 Conclusion 17

PARTIE **1 • PRODUCTION DE PROGRAMMES**

CHAPITRE 2 • **DU PROBLÈME AU PROGRAMME MACHINE** 23

2.1 Du problème au programme 23

2.1.1 Rappel du rôle d'un ordinateur 23
2.1.2 Problème, algorithme, programme et instructions 25

2.2 Les différents niveaux de langage de l'ordinateur 26
2.2.1 Langage machine 27
2.2.2 Langage d'assemblage 28
2.2.3 Langage de haut niveau ou évolué 29
2.3 Introduction à la chaîne de production de programmes 30
2.4 Un exemple 31
2.5 Conclusion 33

CHAPITRE 3 • **LA CHAÎNE DE PRODUCTION DE PROGRAMMES** 35
3.1 La compilation 36
3.1.1 Grammaire et structure d'un langage de haut niveau 36
3.1.2 Analyse lexicale 38
3.1.3 Analyse syntaxique 40
3.1.4 Analyse sémantique 42
3.1.5 Génération du code final 44
3.2 L'édition des liens 46
3.2.1 Rôle de l'éditeur de liens 46
3.2.2 Fonctionnement de l'éditeur de liens 47
3.3 Le chargement 59
3.3.1 Rôle du chargeur 59
3.3.2 Chargement et édition des liens dynamique 61
3.4 L'utilitaire Make 62
3.4.1 Format du fichier Makefile 62
3.4.2 Fonctionnement de l'utilitaire Make 63
3.5 Conclusion 64

CHAPITRE 4 • **LE LANGAGE MACHINE ET LA REPRÉSENTATION DES INFORMATIONS** 65
4.1 La représentation des informations 65
4.1.1 Numération binaire, octale et hexadécimale 66
4.1.2 Représentation des nombres signés 69
4.1.3 Représentation des nombres flottants 74
4.1.4 Représentation des caractères 77
4.2 Les instructions machine 79
4.2.1 Les différents types d'instructions 80
4.2.2 Les différents types d'opérandes 81
4.2.3 Un exemple 82
4.3 Les instructions du langage d'assemblage 84
4.3.1 Format d'une instruction du langage d'assemblage 85
4.3.2 Fonctionnement de l'assembleur 87
4.4 Conclusion 89

CHAPITRE 5 • **LES CIRCUITS LOGIQUES** 90

5.1 Les circuits logiques 90
5.1.1 Définition 90
5.1.2 Les circuits combinatoires 91
5.1.3 Les circuits séquentiels 99
5.1.4 Technologie des circuits logiques 101
5.2 Le futur... 106

CHAPITRE 6 • **EXERCICES CORRIGÉS** 108

Production de programmes 108
6.1 Compilation 108
6.2 Édition des liens 110
6.3 Utilitaire Make 111
6.4 Compilation 111

Représentation des informations 112
6.5 Conversions 112
6.6 Représentation des nombres signés 112
6.7 Représentation des nombres flottants 113
6.8 Synthèse 113

Langage machine 113
6.9 Manipulation des modes d'adressage 117
6.10 Programme assembleur 117
6.11 Manipulation de la pile 118
6.12 Programme assembleur 119

SOLUTIONS 120

PARTIE **2 • STRUCTURE DE L'ORDINATEUR**

CHAPITRE 7 • **LA FONCTION D'EXÉCUTION** 129

7.1 Introduction 129
7.2 Aspects externes 132
7.2.1 Le microprocesseur 132
7.2.2 Les bus 134
7.3 Aspects internes 136
7.3.1 Exécution d'une instruction machine 137
7.3.2 Microcommandes et micro-instructions 145
7.4 Les interruptions : modification du flux d'exécution d'un programme machine 154
7.4.1 Principe des interruptions 154
7.4.2 Un exemple 158

7.5 Amélioration des performances 162
7.5.1 Parallélisme des instructions 163
7.5.2 Parallélisme des processeurs 165
7.6 Conclusion 166

CHAPITRE 8 • **LA FONCTION DE MÉMORISATION** 168
8.1 Généralités 168
8.2 Mémoires de travail 171
8.2.1 Les mémoires vives 171
8.2.2 Les mémoires mortes 180
8.2.3 Les registres 180
8.3 Mémoires de stockage : le disque magnétique 181
8.3.1 Caractéristiques générales 182
8.3.2 Organisation générale 182
8.4 Amélioration des performances 184
8.4.1 Les mémoires caches 184
8.4.2 Mémoire virtuelle 195
8.5 Compléments : approches CISC/RISC 198
8.5.1 Les performances d'un processeur 199
8.5.2 La traduction des programmes 200
8.5.3 Approche CISC 200
8.5.4 Approche RISC 201
8.5.5 Pour conclure sur les RISC et les CISC 202
8.6 Conclusion 203

CHAPITRE 9 • **LA FONCTION DE COMMUNICATION** 205
9.1 Introduction 205
9.2 Les bus 210
9.2.1 Les bus ISA (ou PC-AT), MCA et EISA 211
9.2.2 Le bus PCI (Peripherical Component Interconnect) 212
9.2.3 Le bus AGP (Accelerated Graphics Port) 216
9.2.4 Deux exemples 217
9.3 Les interfaces d'accès aux périphériques 218
9.3.1 Les unités d'échanges 219
9.3.2 Les bus d'extension 232
9.4 Les différents modèles de gestion des entrées-sorties 236
9.4.1 La liaison programmée 237
9.4.2 Entrées-sorties pilotées par les interruptions 239
9.4.3 Gestion des entrées-sorties asynchrones 241
9.5 Conclusion 244

CHAPITRE 10 • **EXERCICES CORRIGÉS** 245

La fonction d'exécution 245
10.1 Révision 245
10.2 Microcommandes 245
10.3 CISC/RISC 246

La fonction de mémorisation 247
10.4 Cache à correspondance directe 247
10.5 Calcul de la taille réelle d'un cache 247
10.6 Cache associatif et remplacement de lignes 247
10.7 Cache à correspondance directe 248

La fonction de communication 248
10.8 Questions de cours 248
10.9 Entrées-sorties programmées et entrées-sorties par interruption 248
10.10 Performances des opérations d'entrées-sorties 249
10.11 Gestion des interruptions 249

Synthèse 250
10.12 Exercice de synthèse 250

SOLUTIONS 253

PARTIE **3 • LES SYSTÈMES D'EXPLOITATION**

CHAPITRE 11 • **INTRODUCTION AUX SYSTÈMES D'EXPLOITATION MULTIPROGRAMMÉS** 265

11.1 Rôle et définition d'un système d'exploitation multiprogrammé 265
11.1.1 Un premier rôle : assurer le partage de la machine physique 267
11.1.2 Un second rôle : rendre conviviale la machine physique 267
11.1.3 Définition du système d'exploitation multiprogrammé 268

11.2 Structure d'un système d'exploitation multiprogrammé 269
11.2.1 Composants d'un système d'exploitation 269
11.2.2 La norme POSIX pour les systèmes ouverts 271

11.3 Principaux types de systèmes d'exploitations multiprogrammés 271
11.3.1 Les systèmes à traitements par lots 272
11.3.2 Les systèmes interactifs 274
11.3.3 Les systèmes temps réel 275

11.4 Notions de base 276
11.4.1 Modes d'exécutions et commutations de contexte 277
11.4.2 Gestion des interruptions matérielles et logicielles 279
11.4.3 Langage de commande 282

11.5 Génération et chargement d'un système d'exploitation 285
11.5.1 Génération d'un système d'exploitation 285
11.5.2 Chargement d'un système d'exploitation 286
11.6 Conclusion 286

CHAPITRE 12 • **GESTION DE L'EXÉCUTION DES PROGRAMMES : LE PROCESSUS** 288
12.1 Notion de processus 288
12.1.1 Définitions 288
12.1.2 États d'un processus 289
12.1.3 Bloc de contrôle du processus 290
12.1.4 Opérations sur les processus 291
12.1.5 Un exemple de processus : les processus Unix 292
12.2 Ordonnancement sur l'unité centrale 295
12.2.1 Ordonnancement préemptif et non préemptif 295
12.2.2 Entités systèmes responsable de l'ordonnancement 297
12.2.3 Politiques d'ordonnancement 297
12.2.4 Exemples 302
12.3 Synchronisation et communication entre processus 304
12.3.1 L'exclusion mutuelle 305
12.3.2 Le schéma de l'allocation de ressources 309
12.3.3 Le schéma lecteurs-rédacteurs 310
12.3.4 Le schéma producteur-consommateur 312
12.4 Conclusion 314

CHAPITRE 13 • **GESTION DE LA MÉMOIRE CENTRALE** 315
13.1 Mémoire physique et mémoire logique 315
13.2 Allocation de la mémoire physique 317
13.2.1 Allocation contiguë de la mémoire physique 317
13.2.2 Allocation non contiguë de la mémoire physique 323
13.3 Mémoire virtuelle 336
13.3.1 Principe de la mémoire virtuelle 336
13.3.2 Le défaut de page 339
13.3.3 Le remplacement de pages 341
13.3.4 Performance 344
13.3.5 Exemples 345
13.3.6 Notion d'écroulement 346
13.4 Swapping des processus 347
13.5 Conclusion 347

CHAPITRE 14 • **SYSTÈME DE GESTION DE FICHIERS** 348

14.1 Le fichier logique 348
14.1.1 Définition 348
14.1.2 Les modes d'accès 349
14.1.3 Exemples 351

14.2 Le fichier physique 354
14.2.1 Structure du disque dur 354
14.2.2 Méthodes d'allocation de la mémoire secondaire 355

14.3 Correspondance fichier logique-fichier physique 366
14.3.1 Notion de répertoire 366
14.3.2 Réalisation des opérations 372

14.4 Protection 379
14.4.1 Protection contre les accès inappropriés 379
14.4.2 Protection contre les dégâts physiques 380

14.5 Conclusion 381

CHAPITRE 15 • **INTRODUCTION AUX RÉSEAUX** 383

15.1 Définition 383

15.2 Les réseaux filaires 385
15.2.1 Architecture des réseaux filaires 385
15.2.2 Circulation des informations 393
15.2.3 Exemple de réseau filaire 396

15.3 Les réseaux sans fil 399
15.3.1 Architecture des réseaux sans fil 400
15.3.2 Circulation des informations 402
15.3.3 Exemples de réseaux sans fil 407

15.4 L'interconnexion de réseaux : Internet 409
15.4.1 Architecture de l'Internet 410
15.4.2 Circulation de l'information 410

CHAPITRE 16 • **EXERCICES CORRIGÉS** 414

Ordonnancement de processus 414
16.1 Algorithmes d'ordonnancement 414
16.2 Ordonnancement par priorité préemptif et non préemptif 414
16.3 Chronogramme d'exécutions 415
16.4 Ordonnancement sous Unix 415
16.5 Ordonnancement sous Linux 416

Synchronisation de processus 417
16.6 Producteur(s)-Consommateurs(s) 417

16.7 Allocations de ressources et interblocage 417
16.8 Allocation de ressources et états des processus 418

Gestion de la mémoire centrale 419

16.9 Gestion de la mémoire par partitions variables 419
16.10 Remplacement de pages 419
16.11 Mémoire paginée et segmentée 420
16.12 Mémoire virtuelle et ordonnancement de processus 420
16.13 Pagination à la demande 421

Système de gestion de fichiers 422

16.14 Modes d'accès 422
16.15 Organisation de fichiers 422
16.16 Noms de fichiers et droits d'accès 422
16.17 Algorithmes de services des requêtes disque 423
16.18 Fichiers Unix 423
16.19 Système de gestion de fichiers FAT 423
16.20 Système de gestion Unix 424
16.21 Synthèse 424

SOLUTIONS 427

INDEX 445

Chapitre **1**

Structure générale et fonctionnement d'un ordinateur

Dans cette partie introductive nous rappelons quelques éléments fondamentaux concernant la programmation et l'algorithmique afin de présenter le vocabulaire utilisé. Il ne s'agit en aucune manière de se substituer à un cours d'algorithmique mais uniquement de replacer du vocabulaire du point de vue de la structure générale d'un ordinateur, l'objectif étant de mettre en évidence les différentes phases qui interviennent dans la résolution d'un problème avec un ordinateur. Après cette partie introductive nous présentons les principaux modules constituant l'architecture d'un ordinateur type. Nous faisons un tour d'horizon des fonctionnalités de chacun de ces modules et de leurs relations fonctionnelles. Il s'agit ici uniquement de présenter de manière globale le fonctionnement de l'ordinateur.

1.1 INTRODUCTION

Le rôle de l'informatique est de résoudre des problèmes à l'aide d'un ordinateur. Un problème s'exprime sous la forme d'un énoncé qui spécifie les fonctions que l'on souhaite réaliser. Par exemple définir toutes les fonctions d'un traitement de texte. Pour résoudre un problème les informaticiens utilisent la notion d'*algorithme*.

Pour illustrer cette notion, prenons l'exemple du problème suivant : c*onfectionner* une omelette avec 6 *œufs*.

Trouver une solution à ce problème repose sur l'existence d'un *processeur* sachant exécuter une *instruction* (*confectionner*). En général un adulte saura exécuter l'instruction confectionner, c'est-à-dire connaîtra le sens du mot, et sera capable de faire toutes les actions nécessaires permettant de résoudre le problème. On dira alors que l'adulte est un *bon processeur* au sens où il saura *exécuter* l'instruction *confectionner* portant sur la *donnée* œufs. Par contre un enfant pourra ne pas connaître le mot confectionner : il ne saura pas faire les opérations nécessaires et ne pourra donc pas résoudre le problème posé (faire une omelette). L'enfant connaît d'autres instructions, sait exécuter d'autres actions que confectionner, et pour qu'il puisse résoudre le problème il faudra l'exprimer autrement, sur la base des actions, *instructions*, qu'il est *capable* d'exécuter. Pour que l'enfant puisse résoudre le problème on pourra l'exprimer sous la forme d'une séquence d'instructions appartenant au *langage* de l'enfant. Par exemple on pourra exprimer le problème, la solution, sous la forme de la séquence des instructions suivantes :

1. *casser* 6 *œufs* dans un bol ;
2. *battre* les *œufs* avec un fouet ;
3. *saler*, *poivrer* ;
4. *placer* la *poêle* sur le *gaz* ;
5. *allumer* le *gaz* ;
6. *cuisiner* les *œufs* ;
7. *éteindre* le gaz.

Dans cet exemple, le processeur enfant sait exécuter des instructions (casser, battre, saler, poivrer, allumer, cuisiner…). De plus il connaît les objets à manipuler (œufs, gaz, poêle…). On dit alors que le processeur enfant est un bon processeur pour exécuter l'algorithme représenté par la séquence précédente puisque l'enfant est capable d'exécuter cette séquence d'instructions. Cette séquence d'instructions exécutables par le processeur enfant est une solution du problème posé pour ce processeur.

Un algorithme peut donc se définir comme une séquence d'instructions exécutables par un processeur déterminé. Cette séquence d'instructions, cet algorithme, est une solution au problème posé.

De ce petit exemple nous pouvons tirer quelques conclusions :

– un algorithme est une solution à un problème posé ;
– un algorithme est une séquence d'instructions exécutables par un processeur ;
– un algorithme est un programme exécutable par un processeur déterminé ;
– un algorithme n'a de sens que pour un processeur déterminé.

Les instructions expriment les actions que peut exécuter un processeur, elles sont codées à partir d'un alphabet (dans notre cas l'alphabet habituel). Les instructions manipulent des données (œufs, sel, poivre, gaz, poêle…). Un processeur (ou machine virtuelle) est une entité capable d'exécuter des instructions portant sur des données. L'ensemble des instructions que le processeur (la machine virtuelle) peut manipuler, constitue son langage de programmation. Ainsi le langage de programmation du processeur définit complètement ce processeur. Il y a équivalence totale entre la

machine virtuelle et son langage de programmation. Aussi, *connaître le langage de programmation d'une machine virtuelle équivaut à connaître les capacités d'exécution de cette machine*.

En résumé résoudre un problème *avec une machine virtuelle* consiste à construire *une séquence d'instructions pour cette machine* (à partir de son langage de programmation) telle que l'exécution de *cette séquence* soit une solution à *ce* problème. En informatique la machine cible, celle avec laquelle nous devons résoudre les problèmes, est l'ordinateur. Nous devons donc connaître les caractéristiques de cette machine, tout particulièrement son langage de programmation (les instructions qu'elle est capable d'exécuter), l'alphabet permettant de coder les instructions ainsi que les données et les outils permettant d'exécuter ces instructions.

Les instructions d'un ordinateur sont *les instructions machines,* elles constituent le langage de programmation de l'ordinateur : *le langage machine.* Résoudre un problème avec un ordinateur consiste donc à exprimer ce problème sous la forme d'une séquence d'instructions machines que nous devrons soumettre aux outils permettant l'exécution de cette séquence. Cette séquence d'instructions machine exécutables par l'ordinateur s'appelle *le programme machine.*

1.2 STRUCTURE ET FONCTIONNEMENT D'UN ORDINATEUR

Après ce bref rappel sur la manière algorithmique de résoudre un problème nous allons nous intéresser à la résolution d'un problème avec comme machine cible un ordinateur. Pour cela nous donnons tout d'abord une présentation de la structure matérielle d'un ordinateur, de son fonctionnement, pour ainsi en déduire comment on peut à l'aide d'un ordinateur, résoudre un problème. L'ordinateur cible nous servant de support descriptif est un ordinateur de type Von Neumann qui caractérise bien la quasi-totalité des ordinateurs actuels. Il est composé des éléments suivants :

- une *mémoire centrale* pour le stockage des informations (programme et données) ;
- un *microprocesseur* ou processeur central pour le traitement des informations logées dans la mémoire centrale ;
- des *unités de contrôle* des périphériques et des périphériques ;
- un *bus de communication* entre ces différents modules.

1.2.1 Structure générale d'un ordinateur

La figure 1.1 présente l'organisation générale d'un ordinateur. On y trouve deux parties principales :

- le processeur comprenant les modules mémoire centrale, processeur central (microprocesseur), les unités d'échange et le bus de communication entre ces différents modules ;

– les périphériques avec lesquels dialogue le processeur au travers des unités d'échange (ou contrôleurs). On distingue en général :
  – les périphériques d'entrée tels que le clavier ou la souris ;
  – les périphériques de sortie tels que les imprimantes et les écrans de visualisation ;
  – les périphériques d'entrée et de sortie tels que les disques magnétiques ou les modems pour accéder aux réseaux de communication.

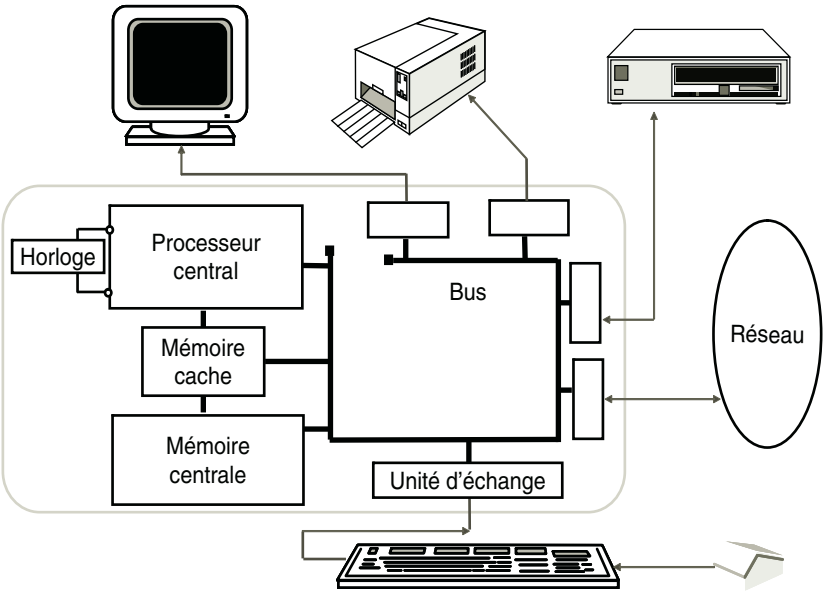


Figure 1.1 Structure matérielle générale.

Globalement le processeur permet l'exécution d'un programme. Chaque processeur dispose d'un langage de programmation (les instructions machine) spécifique. Ainsi résoudre un problème avec un processeur consiste à exprimer ce problème comme une suite de ses instructions machine. La solution à un problème est donc spécifique de chaque processeur. Le programme machine et les données qui sont manipulées par les instructions machine sont placés dans la mémoire centrale.

Examinons à présent plus en détail la composition et les fonctions de chacun des modules composant le processeur.

1.2.2 La mémoire centrale

La mémoire centrale assure la fonction de stockage de l'information qui peut être manipulée par le microprocesseur (processeur central), c'est-à-dire le programme machine accompagné de ses données. En effet, le microprocesseur n'est capable d'exécuter une instruction que si elle est placée dans la mémoire centrale.

Cette mémoire est constituée de circuits élémentaires nommés bits (*binary digit*). Il s'agit de circuits électroniques qui présentent deux états stables codés sous la forme d'un 0 ou d'un 1. De par sa structure la mémoire centrale permet donc de coder les informations sur la base d'un alphabet binaire et toute information stockée en mémoire centrale est représentée sous la forme d'une suite de digits binaires. La figure 1.2 présente l'organisation générale d'une mémoire centrale.

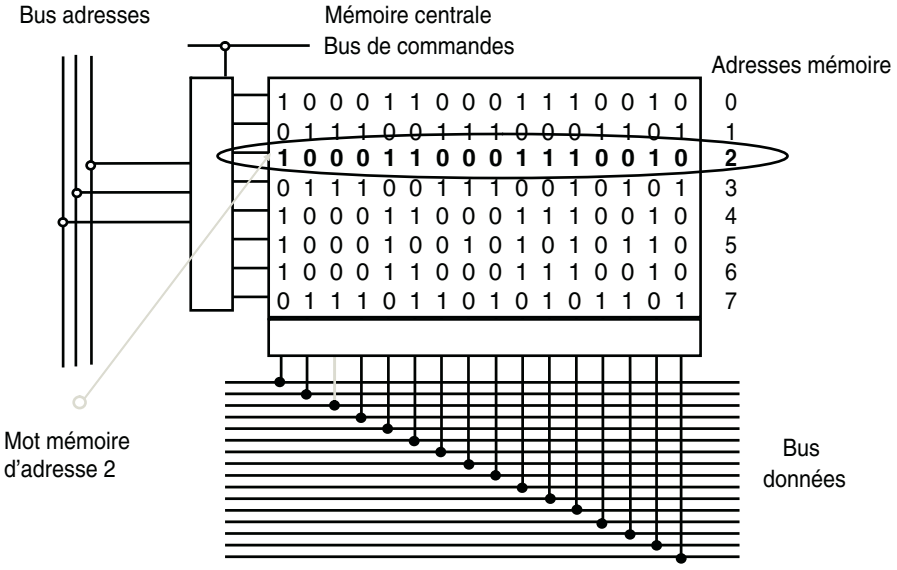


Figure 1.2 La mémoire centrale.

Pour stocker l'information la mémoire est découpée en cellules mémoires : *les mots mémoires*. Chaque mot est constitué par un certain nombre de bits qui définissent sa taille. On peut ainsi trouver des mots de 1 bit, 4 bits (*quartet*) ou encore 8 bits (*octet* ou *byte*), 16 bits voire 32 ou 64 bits. Chaque mot est repéré dans la mémoire par une *adresse*, un numéro qui identifie le mot mémoire. Ainsi un mot est un contenant accessible par son adresse et la suite de digits binaires composant le mot représente le contenu ou valeur de l'information.

La mémoire centrale est un module de stockage de l'information dont la valeur est codée sur des mots. L'information est accessible par mot. La capacité de stockage de la mémoire est définie comme étant le nombre de mots constituant celle-ci. Dans l'exemple de la figure 1.2, notre mémoire a une capacité de 8 mots de 16 bits chacun. On exprime également cette capacité en nombre d'octets ou de bits. Notre mémoire a donc une capacité de 16 octets ou de 128 bits. L'information que l'on trouve en mémoire centrale est donc codée sur un alphabet binaire. La figure 1.3 rappelle le nombre de combinaisons que l'on peut réaliser à partir d'une suite d'éléments binaires. Coder l'information en mémoire centrale c'est donc associer à chaque suite de bits un sens particulier.

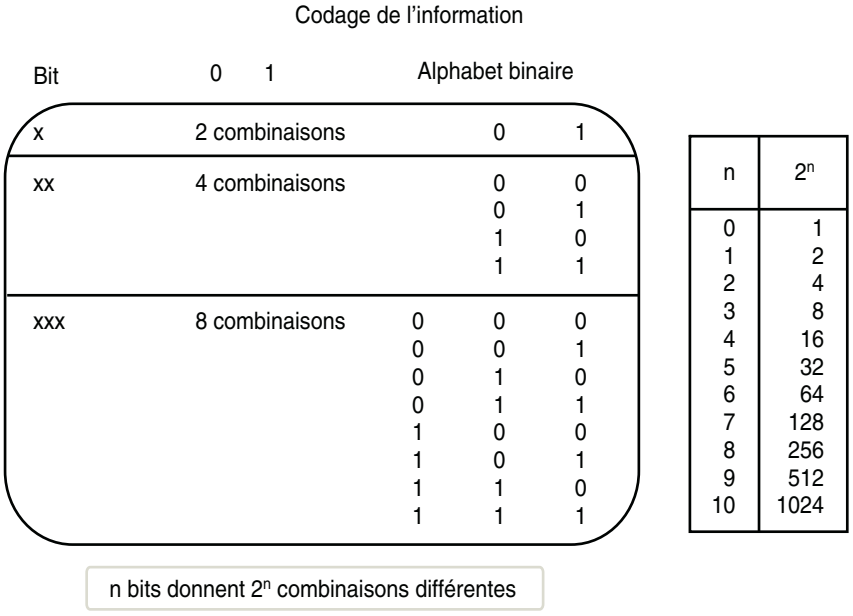


Figure 1.3 Codage binaire.

La figure 1.4 présente succinctement les différentes informations que l'on trouve dans la mémoire centrale : instructions machines et données manipulées par les instructions.

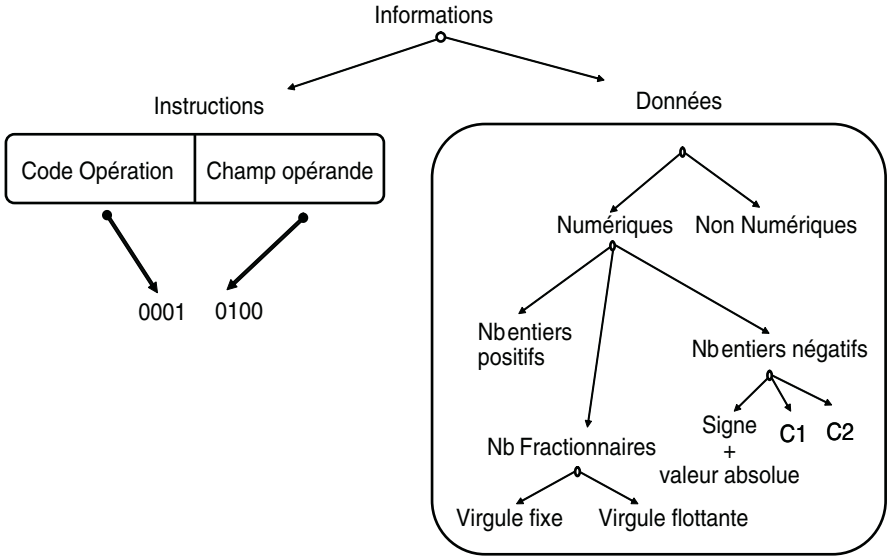


Figure 1.4 Les différentes informations présentes en mémoire centrale.

Les instructions et les données sont codées sur des mots mémoires : elles peuvent occuper un ou plusieurs mots mémoires selon la nature de l'ordinateur. Les instructions machines sont propres à chaque microprocesseur mais sont toujours construites de la même manière : un code opération qui définit l'opération à exécuter, le champ opérande qui définit la ou les données sur lesquelles portent l'opération :

– le *code opération* est codé sur un nombre de digits binaires qui caractérise un microprocesseur. Ce nombre de bits définit en fait le nombre d'opérations possibles avec cet ordinateur : un code opération sur 3 bits admet 8 combinaisons permettant la définition de 8 opérations différentes (instructions machine) possibles, sur 4 bits 16 instructions possibles etc. La taille du code opération est donc un facteur déterminant qui caractérise complètement le nombre d'instructions qu'est capable d'exécuter un processeur ;
– le *champ opérande* est une suite de bits qui permet de caractériser l'adresse de la ou des donnée(s) que manipule(nt) l'instruction machine définie par le code opération. Il existe plusieurs types d'instructions machines qui peuvent manipuler une ou plusieurs données selon la « puissance » du langage machine du microprocesseur utilisé. Il existe également plusieurs manières de définir, à partir du champ opérande, l'adresse d'une donnée : cela repose sur le mécanisme d'adressage d'un microprocesseur qui définit les différentes manières de calculer une adresse de données. On parle également de *modes d'adressages* du microprocesseur.

Les données sont les objets que manipulent les instructions, elles sont codées sur un ou plusieurs mots machines et sont donc adressables (repérables) dans la mémoire centrale. L'adresse de la donnée est déterminée par le type d'adressage utilisé par l'instruction machine. Le codage d'une donnée en mémoire dépend de son type : la figure 1.4 donne les différents types de données que manipulent les instructions machines. Pour chaque type il existe des règles de codage. Par exemple pour coder les caractères alphanumériques on utilise un dictionnaire (table ASCII, table EBCDIC, codage Unicode) tandis que pour coder un nombre entier non signé on utilise une règle traditionnelle de codage d'un nombre sur un alphabet binaire.

Dans l'exemple de la figure 1.5, on suppose un nombre codé sur un octet (8 bits) dont la position de chaque bit est numérotée de 0 à 7, en partant du bit de poids

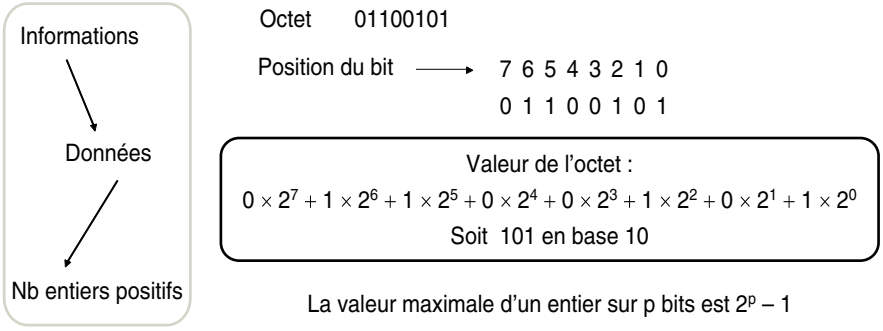


Figure 1.5 Un exemple de codage de l'information.

faible. La valeur de l'entier est alors la somme des produits de chaque bit par le nombre de symboles possibles dans la base d'expression du nombre (ici 2) élevé à la puissance du rang du bit. Cette règle est générale et permet de déterminer la valeur d'un entier codé sur n'importe quel alphabet. Dans le cas d'un alphabet binaire la valeur maximale que peut prendre un entier codé sur p bits est 2^p. Les principales normes de codages existantes à l'heure actuelle ainsi que la structure des instructions machine et les modes d'adressages courants sont détaillés au chapitre 4.

La mémoire centrale a pour objet le stockage des instructions et des données que peut manipuler le microprocesseur. Les opérations possibles sur la mémoire sont la lecture (acquisition par le microprocesseur) d'un mot et l'écriture (le microprocesseur place un nouveau contenu) dans un mot mémoire. Une opération de lecture d'un mot consiste à définir l'adresse du mot et à déclencher une commande de lecture qui amène le contenu du mot de la mémoire vers le microprocesseur. Une opération d'écriture consiste à définir l'adresse du mot dont on veut changer le contenu puis à déclencher une opération d'écriture qui transfère l'information du processeur vers le mot mémoire dont l'adresse est spécifiée.

Enfin d'autres éléments importants complètent la caractérisation d'une mémoire centrale :

– le temps d'accès à la mémoire qui mesure le temps nécessaire pour obtenir une information logée en mémoire ;
– les technologies qui président à la construction de ces mémoires ;
– le coût de réalisation de ces mémoires.

Nous reviendrons en détail sur l'ensemble de ces points dans le chapitre sur la mémorisation (chapitre 8).

1.2.3 Le bus de communication

Le *bus de communication* peut se représenter comme une nappe de fils transportant des signaux et permettant l'échange des informations entre les différents modules du processeur. Chaque fil transporte ou non un signal : il est présent ou absent. On représente par 1 un signal présent et par 0 un signal absent. Le nombre de fils du bus détermine sa largeur et définit ainsi le nombre d'informations différentes que peut véhiculer le bus. Ainsi un bus de 3 fils permet une combinaison de 8 signaux différents et donc représente 8 informations possibles différentes.

Le bus est construit comme un ensemble de trois bus :

– *le bus d'adresses* transporte des combinaisons de signaux qui sont interprétées comme des nombres entiers représentant l'adresse d'un mot mémoire. Par exemple, figure 1.6, le bus d'adresses a une largeur de 3 fils et est donc capable de coder des adresses allant de 0 à 7. Pour adresser un mot mémoire on fait appel à un circuit de sélection (décodeur) qui en entrée reçoit n signaux (3 dans notre exemple) et fournit 2^n signaux de sortie (8 dans notre exemple). Parmi les signaux de sortie un seul est positionné à 1 tous les autres valant 0. Dans notre exemple, les sorties sont numérotées de 0 à 7 et de haut en bas. Ainsi si la valeur d'entrée est 000, seule la

sortie 0 vaut 1 et toutes les autres valent 0. Pour adresser le mot mémoire 2 le microprocesseur place sur le bus d'adresses la chaîne 010 (qui a pour valeur 2 en base 10), la sortie 1 du décodeur vaut alors 1 et les autres valent 0. Ce circuit permet donc de sélectionner un mot mémoire dans la mémoire centrale. La largeur du bus d'adresses définit la *capacité d'adressage du microprocesseur* et il ne faut pas confondre capacité d'adressage et taille physique de la mémoire ;

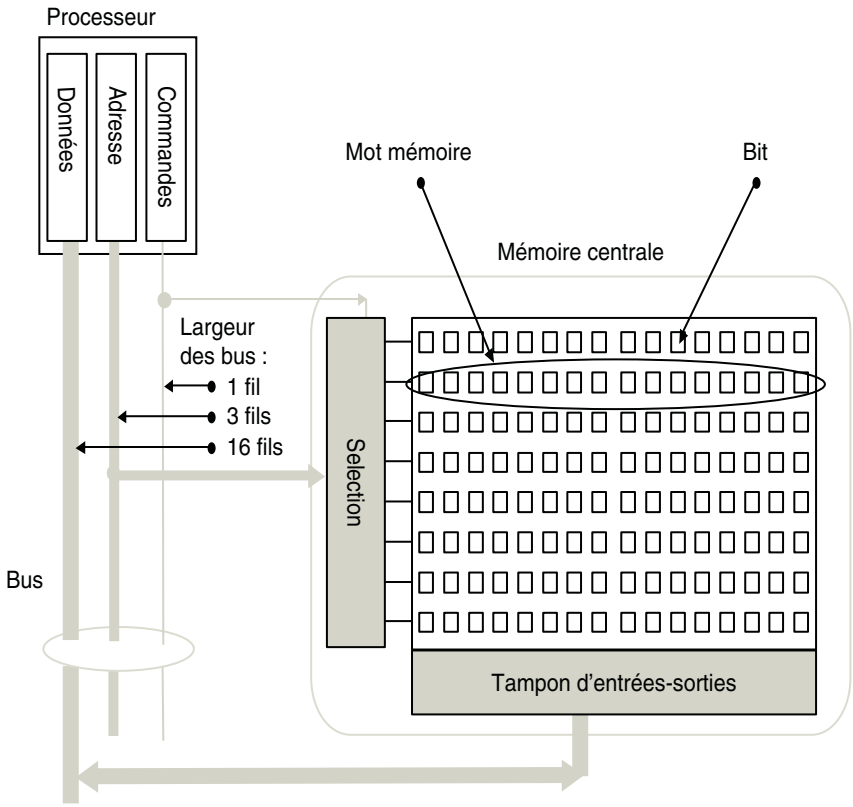


Figure 1.6 Bus de communication.

– le *bus de données* permet l'échange des informations (les contenus) entre les différents modules. Dans notre exemple le bus de données a une largeur de 16 fils et donc la taille des mots mémoires auxquels on peut accéder ou dont on peut modifier le contenu est de 16 bits ;
– le *bus de commandes* : c'est par ce bus que le microprocesseur indique la nature des opérations qu'il veut effectuer. Dans notre exemple il a une largeur d'un fil et donc le microprocesseur ne peut passer que deux commandes (la lecture et l'écriture).

La figure 1.7 résume les différents points abordés concernant la mémoire et les bus permettant la communication avec la mémoire. Sur cette figure 1.7, le bus

d'adresses a une largeur de m bits, le bus de données une largeur de p bits ce qui détermine la capacité de stockage en bits de cette mémoire, soit 2^m mots de p bits. Le bus de commandes permet de déterminer le type d'opération (lecture ou écriture) que l'on souhaite réaliser.

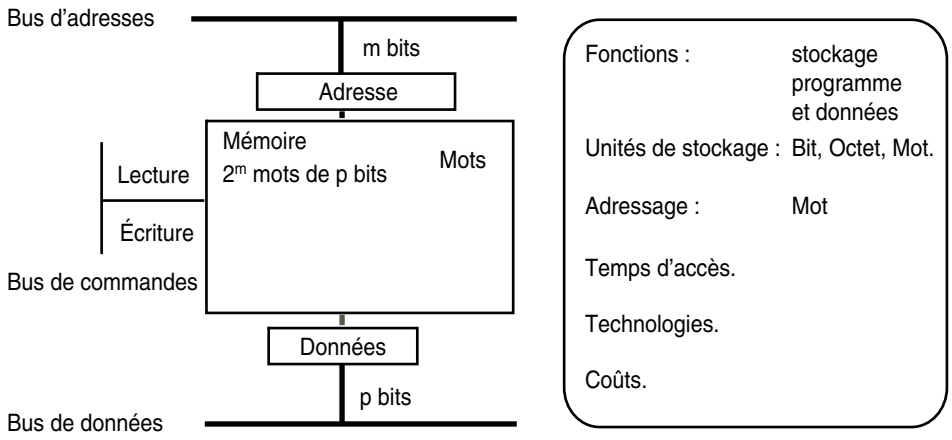


Figure 1.7 Bus de communication et mémoire centrale.

1.2.4 Le processeur central ou microprocesseur

Le *microprocesseur* (unité centrale) a pour objet d'exécuter les instructions machines placées en mémoire centrale. La figure 1.8 présente son architecture générale.

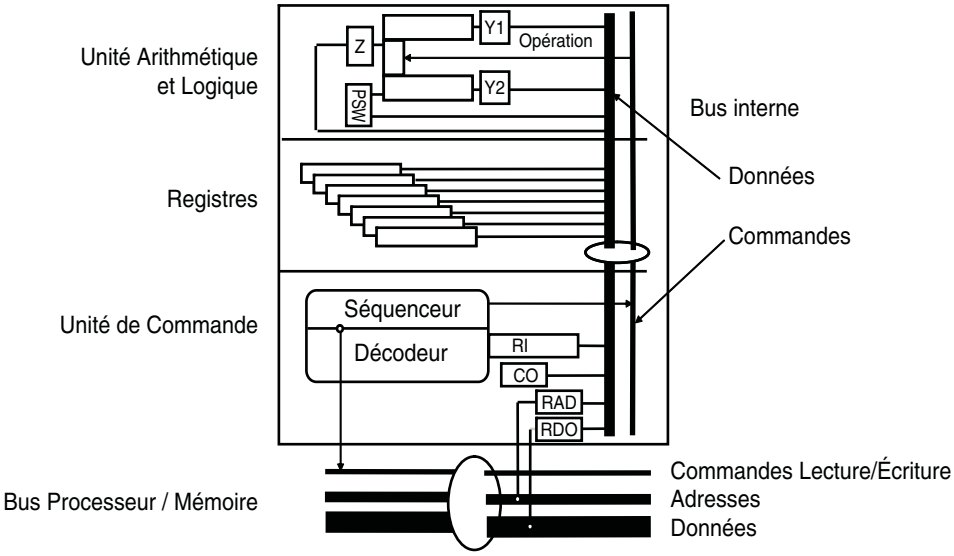


Figure 1.8 Le microprocesseur.

Il est constitué de quatre parties : l'unité arithmétique et logique (UAL), les registres, l'unité commande et le bus de communication interne permettant l'échange des données et des commandes entre les différentes parties du microprocesseur.

Les registres

Ce sont des zones de mémorisation de l'information internes au microprocesseur. Ils sont de faible capacité et de temps d'accès très faible. Leur nombre et leur taille sont variables en fonction du type de microprocesseur. Ils peuvent être de type adresse (ils contiennent alors une adresse de mot mémoire) ou données (ils contiennent alors le contenu d'un mot mémoire). Ils peuvent être spécifiques et avoir une fonction très précise (par exemple le registre pointeur de pile) ou généraux et servir essentiellement aux calculs intermédiaires, par exemple, de l'unité arithmétique et logique.

L'unité arithmétique et logique (UAL)

Ce module est chargé de l'exécution de tous les calculs que peut réaliser le microprocesseur. Cette unité est constituée de l'ensemble des circuits arithmétiques et logiques permettant au processeur d'effectuer les opérations élémentaires nécessaires à l'exécution des instructions machine. Elle inclut donc les circuits d'addition, de soustraction, de multiplication, de comparaison, etc. Dans ce module se trouvent également des registres dont l'objet est de contenir les données sur lesquelles vont porter les opérations à effectuer. Dans notre exemple, l'UAL possède deux registres d'entrée (E1 et E2) et un registre de sortie (S).

Pour faire une addition :

- la première donnée est placée dans E1 via le bus interne de données ;
- la seconde donnée est placée dans E2 via le bus interne de données ;
- la commande d'addition est délivrée au circuit d'addition via le bus interne de commandes ;
- le résultat est placé dans le registre S.

Sur notre machine on note également un registre particulier, le PSW (*Program Status Word*), qui joue un rôle fondamental de contrôle de l'exécution d'un programme et qui à tout instant donne des informations importantes sur l'état de notre microprocesseur. Par exemple puisque nous travaillons sur des mots de longueur finie la valeur d'un entier codé sur un mot ne peut dépasser la valeur maximale représentable sur ce mot. Lorsque nous faisons l'addition de deux entiers le résultat peut avoir une valeur qui n'est pas représentable sur un mot mémoire : il y a alors *dépassement de capacité*. Ce dépassement de capacité doit être signalé et noté pour ne pas perturber le fonctionnement de l'ordinateur. Ce type d'information est stocké dans le PSW.

L'unité de commande

Elle exécute les instructions machines et pour cela utilise les registres et l'UAL du microprocesseur. On y trouve deux registres pour la manipulation des instructions (le compteur ordinal CO, le registre d'instruction RI), le décodeur, le séquenceur et deux registres (le registre d'adresses RAD et le registre de données RDO) permettant

la communication avec les autres modules via le bus. Enfin, via le bus de commandes, elle commande la lecture et/ou l'écriture dans la mémoire centrale.

➤ Le compteur ordinal CO

C'est un registre d'adresses. À chaque instant il contient l'adresse de la prochaine instruction à exécuter. Lors de l'exécution d'une instruction il est prévu, au cours de cette exécution, la modification du contenu du CO. Ainsi en fin d'exécution de l'instruction courante le compteur ordinal pointe sur la prochaine instruction à exécuter et le programme machine peut continuer à se dérouler.

➤ Le registre d'instruction RI

C'est un registre de données. Il contient l'instruction à exécuter.

➤ Le décodeur

Il s'agit d'un ensemble de circuits dont la fonction est d'identifier l'instruction à exécuter qui se trouve dans le registre RI, puis d'indiquer au séquenceur la nature de cette instruction afin que ce dernier puisse déterminer la séquence des actions à réaliser.

➤ Le séquenceur

Il s'agit d'un ensemble de circuits permettant l'exécution effective de l'instruction placée dans le registre RI. Le séquenceur exécute, rythmé par l'horloge du microprocesseur, une séquence de *microcommandes* (*micro-instructions*) réalisant le travail associé à cette instruction machine. Pour son fonctionnement le séquenceur utilise les registres et l'UAL. Ainsi l'exécution effective d'une instruction machine se traduit par l'exécution d'une séquence de micro-instructions exécutables par les circuits de base du microprocesseur. Nous reviendrons plus en détail sur cet aspect des choses dans le chapitre 7, consacré à l'exécution des instructions machines.

➤ Le registre RAD

C'est un registre d'adresses. Il est connecté au bus d'adresses et permet la sélection d'un mot mémoire via le circuit de sélection. L'adresse contenue dans le registre RAD est placée sur le bus d'adresses et devient la valeur d'entrée du circuit de sélection de la mémoire centrale qui va à partir de cette entrée sélectionner le mot mémoire correspondant.

➤ Le registre RDO

C'est un registre de données. Il permet l'échange d'informations (contenu d'un mot mémoire) entre la mémoire centrale et le processeur (registre).

Ainsi lorsque le processeur doit exécuter une instruction il :

– place le contenu du registre CO dans le registre RAD via le bus d'adresses et le circuit de sélection ;
– déclenche une commande de lecture mémoire via le bus de commandes ;
– reçoit dans le registre de données RDO, via le bus de données, l'instruction ;

– place le contenu du registre de données RDO dans le registre instruction RI via le bus interne du microprocesseur.

Pour lire une donnée le processeur :

– place l'adresse de la donnée dans le registre d'adresses RAD ;
– déclenche une commande de lecture mémoire ;
– reçoit la donnée dans le registre de données RDO ;
– place le contenu de RDO dans un des registres du microprocesseur (registres généraux ou registres d'entrée de l'UAL).

On dit que pour transférer une information d'un module à l'autre le microprocesseur établit un *chemin de données* permettant l'échange d'informations. Par exemple pour acquérir une instruction depuis la mémoire centrale, le chemin de données est du type : CO, RAD, commande de lecture puis RDO, RI.

1.3 FONCTIONNEMENT : RELATION MICROPROCESSEUR / MÉMOIRE CENTRALE

L'objet de cette partie est de présenter succinctement comment s'exécute un programme machine sur le matériel que nous venons de définir. Pour être exécutable une instruction doit nécessairement être présente en mémoire centrale. La mémoire centrale contient donc des instructions et des données. De plus toutes les informations en mémoire sont codées sur un alphabet binaire. Les informations sont alors, quelle que soit leur nature, des suites de 0 et de 1. Il faut donc pouvoir différencier instructions et données afin que le registre instruction RI contienne bien des instructions et non des données (et réciproquement). Dans le cas contraire le décodeur ne pourrait interpréter la nature du travail à faire (RI ne contenant pas une instruction mais une

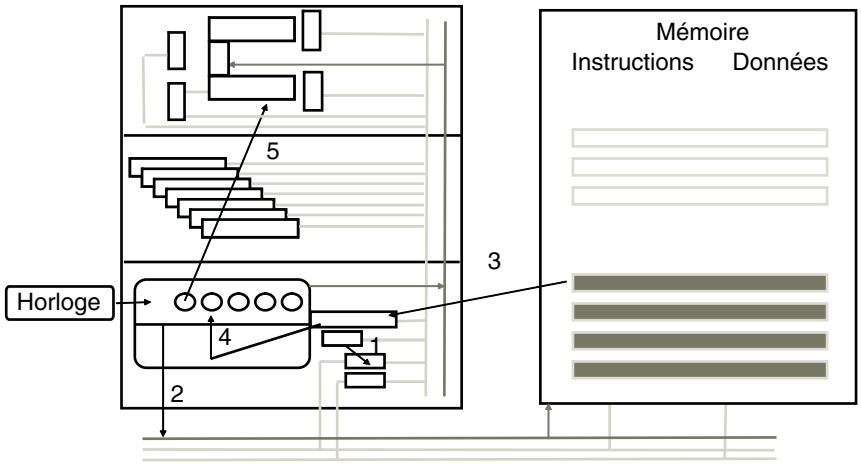


Figure 1.9 Exécution d'une instruction.

donnée). En général lors du placement du programme machine et des données dans la mémoire centrale les instructions et les données sont séparées et occupent des espaces mémoires différents (voir figure 1.9). À la fin du chargement du programme machine et des données en mémoire le compteur ordinal CO reçoit l'adresse de la première instruction du programme à exécuter.

L'exécution peut alors commencer. Le principe général d'exécution est illustré dans la figure 1.9.

Les différentes phases de l'exécution d'une instruction sont les suivantes :

1. le contenu du compteur ordinal CO est placé dans le registre d'adresses RAD : il y a sélection de l'instruction à exécuter ;
2. une commande de lecture de la mémoire centrale est déclenchée via le bus de commandes ;
3. l'instruction est transférée de la mémoire centrale vers le registre instruction RI via le bus de données et le registre de données RDO ;
4. le décodeur analyse l'instruction placée dans le registre instruction RI, reconnaît cette instruction et indique au séquenceur la nature de l'instruction ;
5. le séquenceur déclenche au rythme de l'horloge la séquence de micro-instructions nécessaires à la réalisation de l'instruction.

On peut résumer les étapes de l'exécution d'une instruction (chargement/décodage/exécution) par l'algorithme suivant :

```
début
charger l'instruction à exécuter depuis la mémoire dans le registre
➥ instruction;
modifier le compteur ordinal pour qu'il pointe sur la prochaine
➥ instruction à exécuter;
décoder l'instruction qui vient d'être chargée dans le registre
➥ d'instruction;
charger les données éventuelles dans les registres;
exécuter la séquence des micro-instructions permettant la réalisation
➥ de l'instruction;
fin
```

Les opérations charger/modifier réalisent le chargement de l'instruction dans le registre d'instruction RI. Cette phase est la *phase dite de FETCH*.

Enfin, l'exécution d'un programme machine peut être décrite par l'algorithme suivant :

```
début
exécuter la première instruction du programme
tant que ce n'est pas la dernière instruction
faire
    exécuter l'instruction;
fin faire
fin
```

1.4 UN EXEMPLE

Pour résumer ce que nous venons d'étudier, nous allons prendre un exemple de problème à résoudre avec un ordinateur. Nous définissons tout d'abord le problème et l'ordinateur cible c'est-à-dire son langage de programmation (langage machine). Puis nous construisons le programme machine exécutable par cet ordinateur. Enfin nous plaçons ce programme en mémoire centrale. Il peut alors être exécuté par le microprocesseur.

1.4.1 Le problème

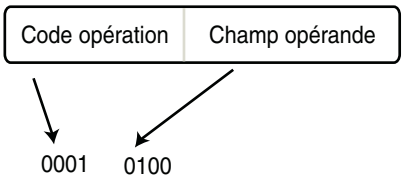
Notre problème consiste à réaliser l'addition de X qui vaut 4 avec Y qui vaut 1 et à placer le résultat dans Z. Nous souhaitons donc, à l'aide de notre ordinateur, réaliser l'opération : Z = X + Y avec X = 4 et Y = 1.

1.4.2 L'ordinateur

Cet ordinateur a la structure générale définie dans la figure 1.11. Les mots de la mémoire sont des octets (8 bits), tous les registres du microprocesseur ont une largeur de 8 bits, les instructions et les données entières sont codées sur un mot mémoire. Données : X est codé : 00000100_2 ; Y est codé : 00000001_2.

1.4.3 Le langage machine

Le code opération est codé sur 4 bits, le champ opérande sur 4 bits. Le champ opérande ne référence qu'une donnée. Ainsi pour faire l'addition de deux nombres un tel langage suppose que la première donnée est spécifiée dans l'instruction et la seconde occupe une adresse implicite. Dans de nombreuses machines cette adresse implicite est un registre appelé *registre Accumulateur* (noté A). L'addition porte alors sur la donnée spécifiée dans l'instruction et le contenu de A, le résultat étant placé dans A.



0001	0100	Charger le contenu du mot mémoire d'adresse 0100 dans le registre A
0010	0101	Additionner le contenu du registre A avec le contenu du mot mémoire d'adresse 0101 et placer le résultat dans le registre A
0011	0110	Placer le contenu du registre A dans le mot mémoire d'adresse 0110
0000		Stopper le programme

Figure 1.10 Le langage de la machine.

Les instructions du langage sont définies dans la figure 1.10. La figure 1.11 résume les différentes phases amenant à l'exécution du programme machine solution de notre problème sur cette machine :

– les données ont été chargées aux adresses 0100 (pour X), 0101 (pour Y), le résultat à l'adresse 0110 (pour Z) ;
– le programme machine est chargé à l'adresse 0111 ;
– le compteur ordinal est chargé avec l'adresse 0111.

À titre d'exercice vérifiez que l'exécution du programme machine résout bien notre problème.

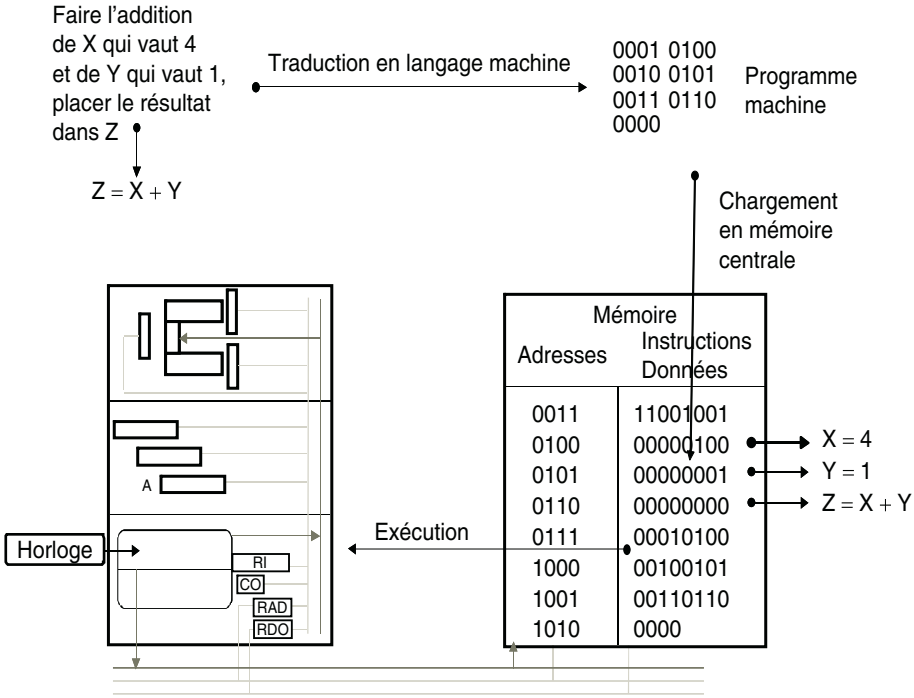


Figure 1.11 Le programme solution.

1.5 LES UNITÉS D'ÉCHANGES

Les *unités d'échanges* permettent la communication entre les modules du processeur et les périphériques. Cette fonction de communication est complexe et nous la détaillons dans le chapitre 9.

Une unité d'échange a une double nature :

– elle est en communication, via le bus interne du processeur, avec la mémoire centrale et le microprocesseur. Des instructions machines spécifiques permettent

les échanges entre mémoire centrale et unité d'échange. On trouve des instructions d'écriture permettant au microprocesseur de placer des informations dans l'unité d'échange et des instructions de lecture permettant au microprocesseur d'acquérir des informations à partir des unités d'échanges. De plus des d'outils de synchronisation permettant d'harmoniser les activités du processeur avec celles des périphériques qu'il pilote sont nécessaires : il ne faut, par exemple, envoyer des caractères vers une imprimante que si celle-ci est prête à imprimer et attendre dans le cas contraire. Dans nos ordinateurs les unités d'échanges ne communiquent pas directement avec le bus interne du processeur mais au travers de bus, dits *bus d'extension*. Il existe une assez grande variété de ces bus d'extension (ISA, USB, *FireWire*, *Fiberchanel*, PCI…) qui satisfont des fonctionnalités diverses et plus ou moins générales. Nous reviendrons plus en détail sur cette question dans le chapitre traitant des questions de communication ;

– elle est en communication avec les périphériques et à ce titre doit être capable de les piloter.

1.6 CONCLUSION

Dans ce chapitre nous avons fait le tour de toutes les étapes importantes nécessaires à la résolution d'un problème avec un ordinateur. La figure 1.12 résume ces différentes étapes.

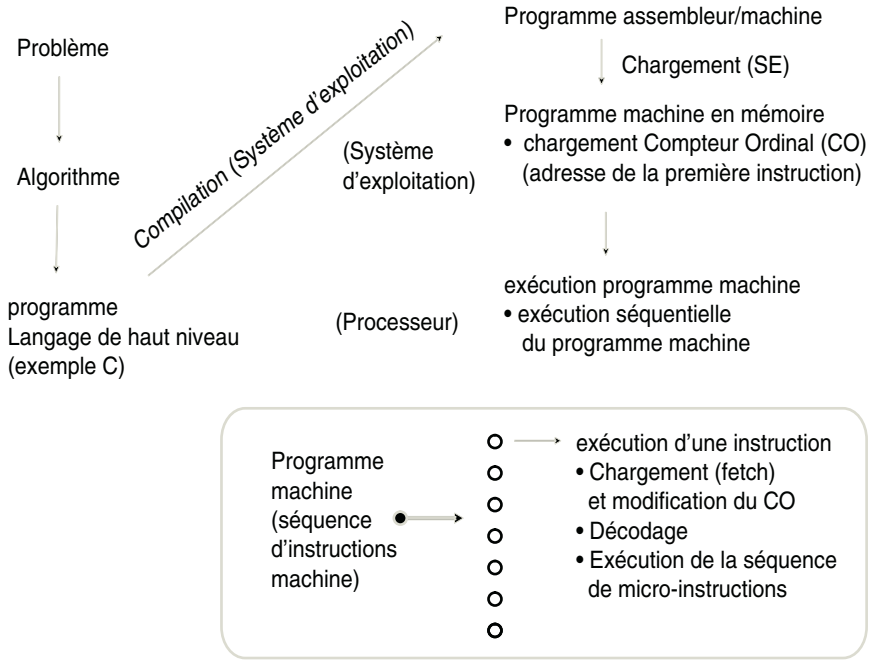


Figure 1.12 Principales étapes de la résolution d'un problème avec l'ordinateur.

Nous voyons qu'une des étapes consiste à exprimer l'algorithme avec un langage de haut niveau. En fait il existe plusieurs niveaux de langage de programmation :

– les *langages de haut niveau* (C, C++ , ADA, JAVA, COBOL, PASCAL, PYTHON…) ne sont pas directement exécutables par le processeur cible que nous utilisons : l'ordinateur. Il faut donc « traduire » les programmes exprimés dans ces langages afin d'obtenir le programme machine exécutable par le processeur cible. C'est l'objet de la *compilation* et/ou de l'*interprétation*. Les compilateurs (interpréteurs) sont des programmes (du Système d'Exploitation : SE) qui ont la connaissance de la syntaxe des langages de haut niveau et qui connaissent le langage machine de la machine cible. Ces langages sont nécessaires car on ne sait pas traduire directement le langage naturel en langage machine alors que l'on sait construire des traducteurs pour ces langages de haut niveau ;
– les *langages de bas niveaux* sont des langages directement exécutables par un microprocesseur. Ce sont les langages machines. Ils se présentent sous deux formes :
  – les langages machine codés binaires. Ce sont les seuls réellement exécutables par le processeur. Codés sous forme binaire ils sont difficiles à manipuler ;
  – les langages d'assemblage. Ce sont des langages machines symboliques au sens où les codes opération et les champs opérande d'une instruction sont exprimés à l'aide de symboles.

Par exemple dans la petite machine qui nous a servis d'exemple, 00010010 est une instruction machine demandant de charger le contenu de l'adresse mémoire 0010 dans le registre accumulateur A du microprocesseur. Il existe dans le langage d'assemblage de cette machine une instruction qui a la forme « `Load X` » où `Load` exprime à l'aide de symboles alphanumériques le code opération codé binaire 0001 et X exprime symboliquement l'adresse mémoire de la donnée X. Ce type de langage est beaucoup plus simple à utiliser que le langage machine codé binaire. Bien sûr un programme écrit en langage d'assemblage n'est pas directement exécutable par le microprocesseur. Cependant la traduction, langage d'assemblage, langage machine pur, est simple car à chaque instruction du langage machine correspond une instruction du langage d'assemblage. Le traducteur de tels langages s'appelle l'assembleur. Une programmation dans ce type de langage est possible mais de moins en moins fréquente. Elle se justifie encore lorsque l'on recherche une grande efficacité, par exemple, dans la programmation des jeux vidéos ou des interfaces graphiques.

Le traducteur (compilateur/interpréteur) place le résultat de sa traduction (programme machine) sur le disque magnétique. Le programme machine doit être chargé en mémoire centrale pour être exécuté par le microprocesseur. C'est un programme du système d'exploitation, *le chargeur*, qui réalise ce chargement en mémoire. Le chargeur connaît l'adresse de la première instruction du programme machine, il peut ainsi initialiser le compteur ordinal pour lancer l'exécution. À partir de ce moment les instructions, une à une, sont prises en charge par le microprocesseur qui les exécute.

La première partie de cet ouvrage est consacrée au passage du problème au programme machine ; elle présente en détail le fonctionnement du compilateur et des autres outils permettant de construire un programme machine. Elle approfondit

ensuite les notions rencontrées ici de langage machine, de modes d'adressages et de codages pour la représentation des informations.

Dans la seconde partie nous détaillons :

– la fonction d'exécution où nous apportons des précisions sur ce qu'est réellement l'exécution d'une instruction machine. Nous précisons la notion de micro-instruction. Dans cette partie nous abordons également la notion d'*interruption* qui est un mécanisme fondamental dans nos ordinateurs ;
– la fonction de mémorisation en précisant les différents types de mémoires constituant la mémoire centrale. Nous abordons les problèmes de synchronisation entre processeur et mémoire centrale et voyons comment la mémoire cache apporte des solutions aux différences de vitesse entre microprocesseur et mémoire centrale ;
– la fonction de communication dans laquelle nous revenons sur la manière d'échanger des informations entre processeur et périphériques.

Enfin, nous commençons à le voir, le système d'exploitation joue un rôle central dans la gestion et le fonctionnement des ordinateurs. Nous y consacrons la troisième partie de cet ouvrage.

Pour terminer cette introduction, quelques mots pour préciser le processus de démarrage d'un ordinateur. En plus de la mémoire centrale (communément appelée mémoire RAM) il existe une mémoire ROM, dite mémoire morte, uniquement accessible en lecture donc non modifiable par programme et qui de plus n'est pas volatile : quand le courant est coupé, le contenu de cette mémoire ROM n'est pas altéré contrairement aux mémoires RAM pour lesquelles le contenu est perdu lorsque le courant est coupé. Cette mémoire ROM est chargée, une fois pour toutes, avec un programme : le *bootstrap*.

Par ailleurs sur le disque magnétique est placé le système d'exploitation. Le système d'exploitation est un ensemble de programmes exécutables sur le microprocesseur. Un de ces programmes, l'interpréteur de langage de commandes (le Shell sous Unix), comprend des commandes telles que demander l'exécution d'un éditeur de texte, d'un compilateur, d'un lanceur de programmes.

Le bootstrap connaît l'adresse sur le disque du système d'exploitation, en particulier de son noyau composé entre autre de l'interpréteur de langage de commandes.

Lors de la mise sous tension de l'ordinateur, automatiquement le bootstrap s'exécute :

– le bootstrap charge en mémoire centrale le noyau du système d'exploitation (en particulier l'interpréteur du langage de commande) et lance son exécution ;
– ce dernier attend, au travers d'une interface de communication, les commandes de l'utilisateur. Celui-ci demande :
  – un éditeur de texte pour saisir le code de son programme ;
  – un compilateur pour le traduire en langage machine ;
  – le lancement de l'exécution de son programme machine résultat de la traduction. Pour cela, l'utilisateur, doit connaître le langage de commande qu'est capable de comprendre l'interpréteur de langage de commande du système d'exploitation utilisé. Il existe plusieurs types d'interface de communication qui vont de l'écriture textuelle de la ligne de commande, encore courante sous Unix, aux

interfaces graphiques sophistiquées que l'on trouve sous Windows, Mac OS X ou Linux. Avec ces interfaces graphiques on n'écrit plus de ligne de commandes, dont il faut connaître la syntaxe, mais on clique sur des icônes faisant référence au programme que l'on souhaite exécuter ;

– le lanceur du programme machine le charge (via le chargeur) en mémoire centrale, place l'adresse de la première instruction à exécuter dans le compteur ordinal : le programme machine s'exécute.

PARTIE 1

PRODUCTION DE PROGRAMMES

L'ensemble des chapitres de cette partie est centré autour de la notion de *chaîne de production de programmes*. À partir de la description du rôle premier d'un ordinateur, nous présentons succinctement les notions fondamentales d'*algorithmes*, de programmation ainsi que les différents niveaux de langages disponibles sur un ordinateur. Nous abordons ainsi les concepts de *langage machine* et *langage haut niveau*, puis nous nous intéressons aux différentes étapes permettant la traduction d'un programme écrit en langage haut niveau vers son équivalent écrit en langage machine et exécutable par l'ordinateur.

Le chapitre 2 introduit la notion d'algorithme et les différents niveaux de langages disponibles sur un ordinateur. Le chapitre 3 décrit la chaîne de production de programmes et présente les différentes étapes qui la composent, c'est-à-dire la *compilation*, *l'édition de liens* et *le chargement*. Enfin, le chapitre 4 présente la notion de langage machine, de langage d'assemblage et décrit les principales conventions de représentation des informations (nombres, caractères) sur la machine. Le chapitre 5 est consacré aux différents circuits logiques qui constituent la partie matérielle de l'ordinateur et à la technologie mise en œuvre pour leur fabrication. Cette partie s'achève avec un ensemble d'exercices corrigés.

Mots-clés : algorithme, compilation, édition des liens, langage machine, représentation binaire, circuits logiques, transistors.

Chapitre 2

Du problème au programme machine

À partir de la description du rôle premier d'un ordinateur, nous exposons le processus qui permet à un être humain de soumettre la résolution d'un problème à un ordinateur, ceci sous forme d'un programme écrit dans un langage de programmation dit langage de haut niveau, qui code une solution appelée algorithme. L'ordinateur ne pouvant exécuter que des instructions codées en langage machine, il est nécessaire de traduire ce langage de haut niveau vers le langage machine de l'ordinateur : c'est le rôle de la chaîne de production de programmes.

2.1 DU PROBLÈME AU PROGRAMME

2.1.1 Rappel du rôle d'un ordinateur

L'ordinateur est présent de nos jours dans de multiples domaines et lieux. En une cinquantaine d'années, il a envahi notre quotidien et il est bien des domaines à présent où son utilisation et sa puissance se révèlent indispensables. En effet, l'ordinateur a investi de multiples espaces et remplit des missions très diverses qui vont du calcul scientifique pour par exemple séquencer l'ADN ou analyser les modèles météorologiques, au pilotage de procédé tel que la surveillance d'une centrale nucléaire et encore les jeux ou la communication via l'Internet.

De fait, la multiplicité des missions remplies par l'ordinateur cache en quelque sorte son unique rôle : archiver des données et exécuter un programme de traitement

sur ces données en vue de résoudre un problème. L'ordinateur offre à l'être humain une puissance de stockage, de calcul et de traitement bien supérieure à ce à quoi il peut prétendre par lui-même. Prenons trois exemples :

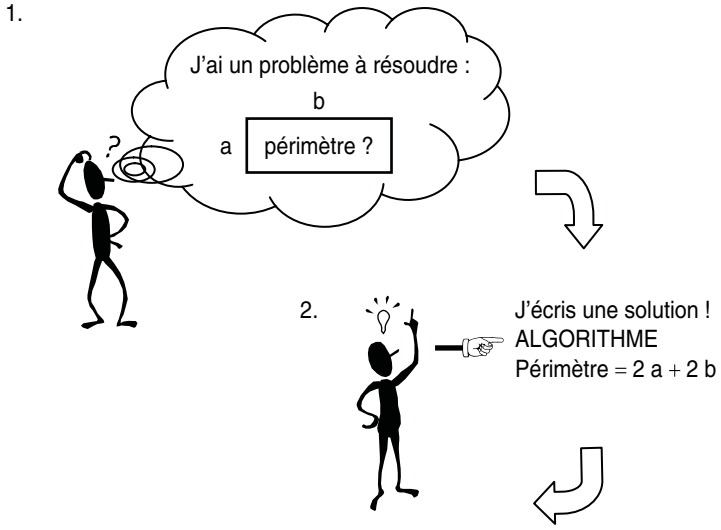
– en paléontologie, l'analyse cladistique a pour but de mettre en évidence les liens de parenté existants entre différents organismes afin d'identifier des clades[1] et de construire les arbres évolutifs. Pour parvenir à ce but, le paléontologue doit identifier entre plusieurs organismes la possession en commun de caractères dérivés due à une ascendance commune. Une telle recherche ne peut se faire que par une comparaison simultanée d'une grande quantité de données, c'est-à-dire de caractères et d'espèces, qu'il est inenvisageable de réaliser sans le concours de l'ordinateur. En effet la quantité de données à comparer est telle que pour un être humain établir un seul arbre demanderait un temps considérable. Grâce à l'ordinateur, il est possible d'établir et de calculer des millions d'arbres possibles en un temps raisonnable[2] ;
– en informatique de gestion, les systèmes de bases de données permettent de stocker, classer et gérer des millions de données dans un espace bien plus réduit qu'au temps où seul le support papier existait. Par ailleurs, les performances des méthodes d'interrogations liées à ces systèmes rendent possibles des extractions de données suivant des objectifs multicritères dans des temps réduits, là où il faudrait un traitement manuel de plusieurs heures. Songez au catalogue informatique de la bibliothèque municipale de votre ville, comme il est aisé de demander à l'ordinateur de rechercher dans l'ensemble des livres traitant de la musique baroque, les seuls ouvrages consacrés à la fois à Haendel et Vivaldi. Vous n'avez plus alors qu'à vous diriger vers le rayon désigné, prendre le livre correspondant à la cote fournie ; là où il vous aurait fallu soit compulser le classeur papier trié par ordre alphabétique ou chacun des livres du rayon musique ;
– dans le domaine du contrôle de procédé, l'ordinateur supplée à l'humain en offrant une plus grande vitesse de réaction aux événements et une plus grande fiabilité dans le contrôle réalisé. La complexité de pilotage d'un avion de chasse, par exemple, est telle, la quantité d'informations remontées au cockpit si importante (informations de pilotage, informations radios, informations liées au système d'armes, informations liées au système de défense antimissiles), que le pilote ne peut pas piloter son appareil sans le concours de l'informatique embarquée. Celle-ci lui permet entre autre une vitesse de réaction supérieure en prenant elle-même en main des réponses automatisées : ainsi le système de défense antimissiles est capable d'analyser de lui-même le type de missiles à intercepter (thermique ou radar), de lancer les leurres adaptés (fusées éclairantes ou feuilles métalliques) avant de remonter l'information au pilote pour que celui-ci dévie son avion afin d'éviter l'impact.

1. Une clade (du grec *klados* « rameau ») désigne une branche d'un arbre évolutif comprenant un ancêtre et ses descendants.
2. Tassy Pascal, *Le paléontologue et l'évolution*, Collection Quatre à Quatre, Éditions Le Pommier-Fayard, septembre 2000, 158 pages.

2.1.2 Problème, algorithme, programme et instructions

L'ordinateur est puissant et rapide mais il n'a aucune intelligence propre et aucune imagination. Tout ce dont l'ordinateur est capable, c'est d'exécuter ce que l'être humain lui commande de faire ; en aucun cas, il ne peut fournir de lui-même une solution à un problème donné. L'être humain utilise donc l'ordinateur pour résoudre un problème, encore faut-il que ce problème puisse être exprimé à l'ordinateur : c'est l'activité de programmation. Ainsi, vouloir résoudre un problème à l'aide de l'ordinateur demandera de mettre en œuvre le processus suivant (figure 2.1) :

– à charge de l'être humain de trouver une ou plusieurs solutions au problème ;
– au moins une des solutions trouvées est ensuite exprimée sous forme de règles opératoires telles que des itérations, des conditionnelles, des tests booléens, des opérations arithmétiques : c'est la construction de l'*algorithme* ;
– l'algorithme est codé sous forme d'*instruction*s dans un langage exécutable et compréhensible par l'ordinateur : c'est l'activité de *programmation* qui construit un *programme*. Une instruction équivaut à un ordre qui entraîne l'exécution par l'ordinateur d'une tâche élémentaire.



3. En utilisant un langage de programmation, je code la solution pour la faire exécuter par l'ordinateur

PROGRAMME constitué d'instructions

```
function perimetre (a, b : in integer) return integer is
begin
   perimetre := (2 * a) + (2 * b);
end;
```

Figure 2.1 Résoudre un problème à l'aide de l'ordinateur.

On notera qu'il découle de ce processus que l'ordinateur a besoin de trois éléments essentiels :

– un moyen de communication avec l'être humain : clavier, souris, écran, etc. ;
– un moyen d'exécuter les instructions du langage : le processeur ;
– un moyen de conserver des données : les périphériques de stockage tels que le disque dur, la disquette, le CD-ROM.

2.2 LES DIFFÉRENTS NIVEAUX DE LANGAGE DE L'ORDINATEUR

La programmation est donc l'activité qui consiste à traduire par un programme un algorithme dans un langage assimilable par l'ordinateur. Cette activité de program-

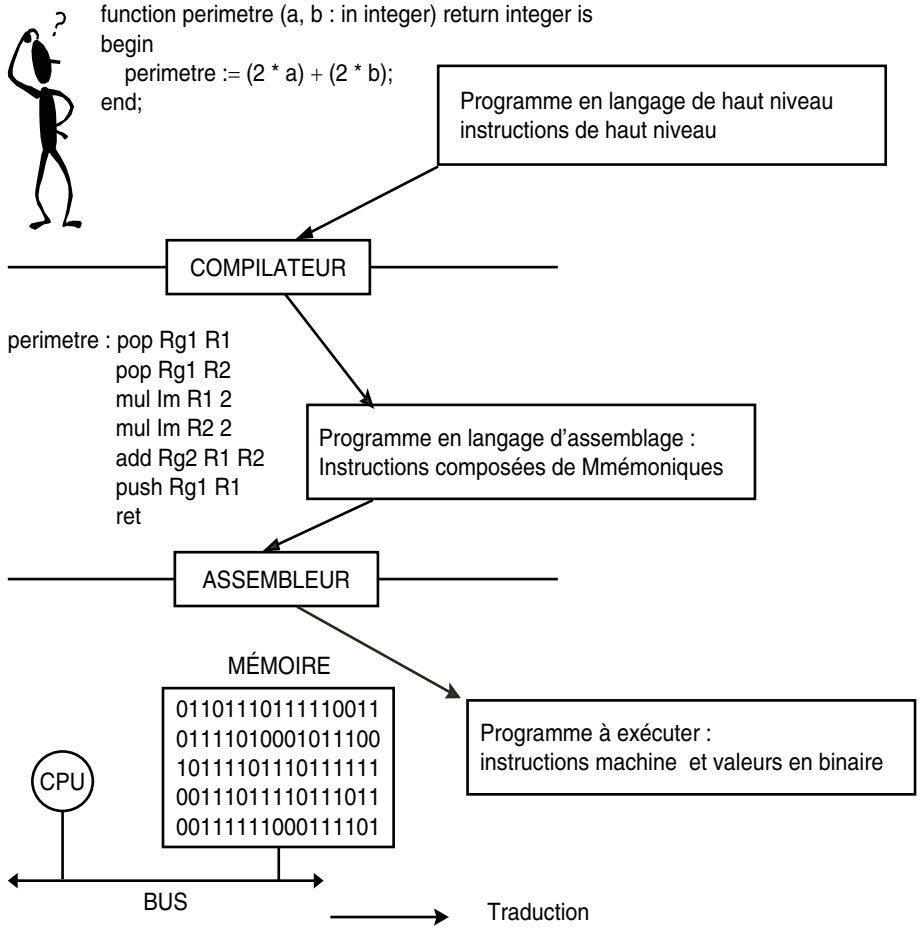


Figure 2.2 Les différents niveaux de programmation.

mation peut s'effectuer à différents niveaux, plus ou moins proches et dépendants de l'architecture physique de la machine. Essentiellement on distinguera trois niveaux (figure 2.2) : la programmation de bas niveau en langage machine, la programmation de bas niveau en langage d'assemblage, la programmation de haut niveau à l'aide d'un langage de haut niveau ou langage évolué.

2.2.1 Langage machine

La donnée de base manipulée par la machine physique est le bit (*Binary Digit*) qui ne peut prendre que deux valeurs : 0 et 1. Ce 0 et 1 correspondent aux deux niveaux de voltage (0-1 et 2-5 volts) admis pour les signaux électriques issus des composants électroniques (transistors) qui constituent les circuits physiques de la machine (voir chapitre 5, *Les circuits logiques*). Au niveau physique, toutes les informations (nombres, caractères et instructions) ne peuvent donc être représentées que par une combinaison de 0 et 1, c'est-à-dire sous forme d'une chaîne binaire. C'est le niveau de programmation le plus bas et le plus proche du matériel, celui du langage machine.

À ce niveau, la programmation et les instructions du langage sont totalement dépendantes de l'architecture de la machine et du processeur et manipulent directement les registres du processeur ou encore les adresses en mémoire physique, tout cela sous forme de chaînes binaires. Ainsi, une instruction machine (figure 2.3) est une chaîne binaire composée essentiellement de deux parties :

– le code opération désigne le type d'opération à effectuer (addition, ou logique, lecture mémoire…) ;
– le reste de l'instruction sert à désigner les opérandes, c'est-à-dire les données sur lesquelles l'opération définie par le code opération doit être réalisée. Ces opérandes sont soit des mots mémoires, soit des registres du processeur ou encore des valeurs immédiates.

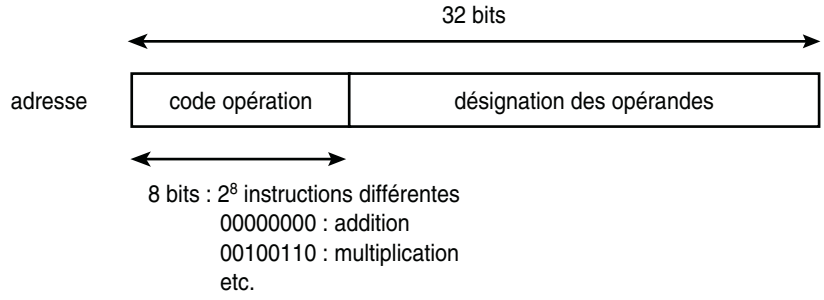


Figure 2.3 Instruction machine.

Chaque instruction est par ailleurs repérée par une adresse qui mémorise la position de l'instruction dans le programme.

Ce niveau de programmation est évidemment très fastidieux, voire impraticable par l'être humain.

2.2.2 Langage d'assemblage

Une première amélioration a consisté à substituer aux chaînes binaires représentant des codes opérations ou des adresses mémoires, des chaînes de caractères plus aisément manipulables par l'être humain que l'on appelle des mnémoniques : c'est le *langage d'assemblage* qui constitue une variante symbolique du langage machine, permettant au programmeur de manipuler les instructions de la machine en s'affranchissant notamment des codes binaires et des calculs d'adresse. Le langage d'assemblage comporte le même jeu d'instructions que le langage machine et est également spécifique de la machine.

Une instruction du langage d'assemblage (figure 2.4) est composée de champs, séparés par un ou plusieurs espaces. On identifie :

– un champ *étiquette*, non obligatoire, qui correspond à l'adresse de l'instruction machine ;
– un champ *code opération*, qui correspond à la chaîne binaire code opération de l'instruction machine ;
– un champ *opérandes* pouvant effectivement comporter plusieurs opérandes séparés par des virgules qui correspondent aux registres, mots mémoires ou valeurs immédiates apparaissant dans les instructions machine.

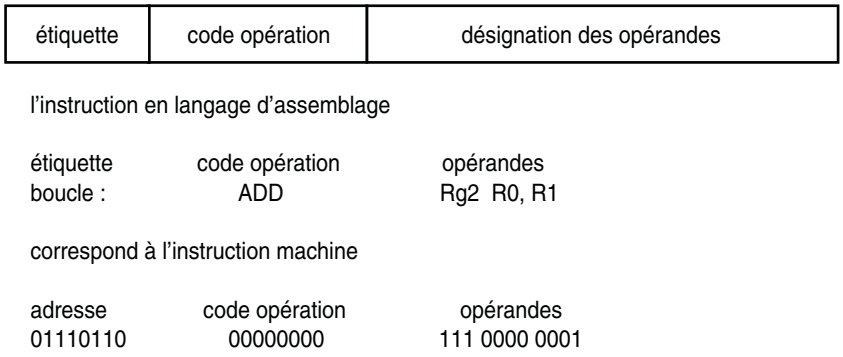


Figure 2.4 Instruction en langage d'assemblage.

La programmation au niveau du langage d'assemblage, quoique bien plus aisée qu'au niveau machine, est encore fastidieuse pour l'être humain et requiert surtout de connaître l'architecture du processeur et de la machine. On parle ainsi du langage d'assemblage du processeur Intel Pentium ou du langage d'assemblage du processeur Athlon AMD.

Lorsque la programmation en langage d'assemblage est mise en œuvre, elle nécessite une étape de traduction, car seules les instructions en langage machine sont compréhensibles et exécutables par la machine. Pour pouvoir exécuter un programme écrit en langage d'assemblage, il faut donc traduire les instructions de celui-ci vers

les instructions machine correspondantes. Cette phase de traduction est réalisée par un outil appelé l'*assembleur*[1].

2.2.3 Langage de haut niveau ou évolué

L'étape suivante est venue de la genèse des langages évolués ou langages de haut niveau. Ces langages se caractérisent principalement par le fait qu'ils sont, contrairement aux deux autres types de langage que nous venons d'aborder, totalement indépendants de l'architecture de la machine et du processeur. Par ailleurs, ils offrent un pouvoir d'expression plus riche et plus proche de la pensée humaine, rendant ainsi plus aisée la traduction des algorithmes établis pour résoudre un problème. Ces langages de fait sont davantage définis par rapport aux besoins d'expression du programmeur que par rapport aux mécanismes sous-jacents de la machine physique. Ils intègrent ainsi des structures opératoires semblables à celles des algorithmes telles que les itérations, les boucles, les conditionnelles.

Ainsi les langages de haut niveau sont plus ou moins spécialisés par rapport à une classe de problèmes à résoudre : COBOL est destiné aux applications de gestion tandis que FORTRAN est plutôt orienté vers le domaine du calcul scientifique. D'autres langages sont plus universels tels que C, C++ , Ada, Java ou encore Pascal.

De nos jours, les langages haut niveau sont classés selon plusieurs grandes familles. Deux familles de langages importants et courants sont la famille des langages dits procéduraux et la famille des langages dits objets :

– le langage procédural : l'écriture d'un programme est basée sur les notions de procédures et de fonctions, qui représentent les *traitements* à appliquer aux données du problème, de manière à aboutir à la solution du problème initial. Les langages C et Pascal sont deux exemples de langages procéduraux ;
– le langage objet : l'écriture d'un programme est basée sur la notion d'objets, qui représentent les différentes entités entrant en jeu dans la résolution du problème. À chacun de ces objets sont attachées des méthodes, qui lorsqu'elles sont activées, modifient l'état des objets. Les langages Java et Eiffel sont deux exemples de langages objets.

Les langages évolués étant indépendants de la machine, ils ne peuvent être directement exécutés par la machine. Un programme écrit en langage haut niveau doit donc être converti vers son équivalent en langage machine. C'est le rôle du traducteur de langage, qui est spécifique à chaque langage évolué utilisé.

Les traducteurs sont divisés en deux catégories : les *compilateur*s et les *interpréteurs*.

– un compilateur traduit une fois pour toutes le langage évolué en langage machine et construit ainsi un programme qualifié de programme objet qui est stocké sur un support de masse tel qu'un disque ;

1. Par abus de langage, le terme assembleur désigne tout à la fois le langage d'assemblage lui-même et l'outil de traduction.

– un interpréteur lit une à une les instructions du langage évolué, puis il les convertit immédiatement en langage machine avant qu'elles ne soient exécutées au fur et à mesure. Il n'y a pas de génération d'un fichier objet conservant la traduction des instructions en langage évolué vers le langage machine et la traduction doit donc être refaite à chaque nouvelle demande d'exécution du programme.

Plus généralement, le passage d'un programme dit programme source écrit en langage de haut niveau vers un programme exécutable en langage machine est assuré par un processus comportant plusieurs étapes dont l'une est la compilation, que l'on qualifie de *chaîne de production de programmes*.

2.3 INTRODUCTION À LA CHAÎNE DE PRODUCTION DE PROGRAMMES

La chaîne de production de programmes est présentée sur la figure 2.5.

L'éditeur de texte est un logiciel interactif permettant de saisir du texte à partir d'un clavier et de le stocker dans un fichier – le programme source –, sur un support

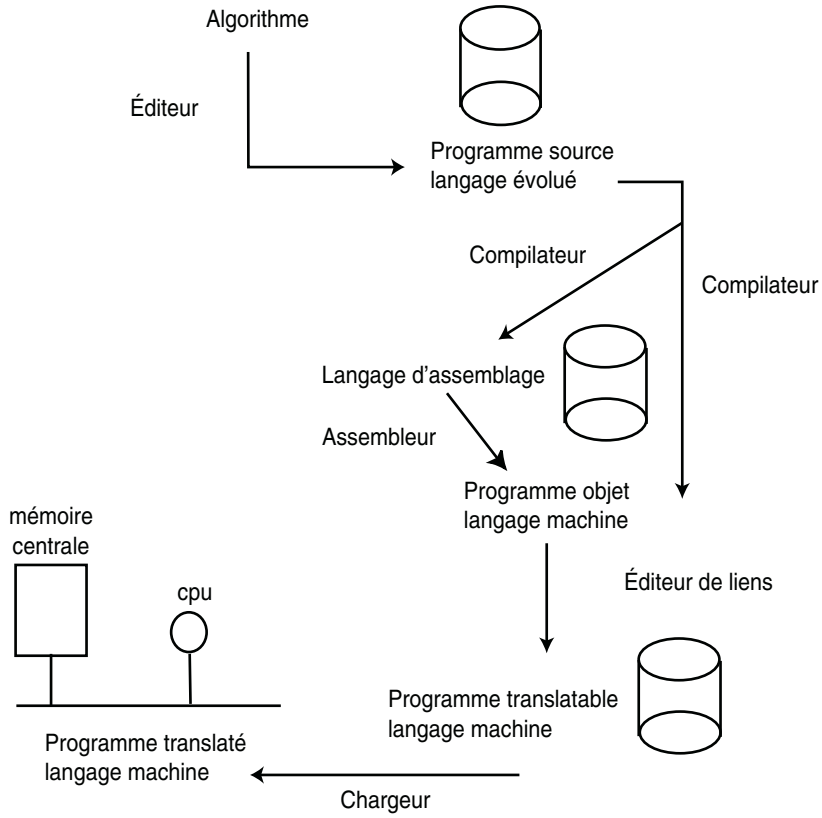


Figure 2.5 Chaîne de production de programmes.

de masse tel qu'un disque. Le *compilateur* permet la traduction du programme source en un programme objet qui est soit directement le langage machine, soit le langage d'assemblage. Le passage du langage d'assemblage au langage machine se fait par l'intermédiaire d'un autre traducteur, l'*assembleur.* Le programme objet est stocké sur le disque. L'*éditeur de liens* est un logiciel qui permet de combiner plusieurs programmes objet en un seul et de résoudre des appels à des modules de librairie. L'éditeur de liens résout les références externes.

Le programme construit à l'issue de l'édition des liens comporte une allocation des instructions commençant à 0. Pour être exécuté, le programme doit être transféré depuis le disque vers la mémoire centrale : c'est le rôle du *chargeur* de placer le programme en mémoire centrale à partir d'une adresse d'implantation et de translater toutes les adresses du programme de la valeur de l'adresse d'implantation.

2.4 UN EXEMPLE

Considérons que nous souhaitions écrire un programme qui permet le calcul du périmètre ou de la surface soit d'un cercle de rayon r, soit d'un carré de côté a, soit d'un rectangle de côtés a et b, soit d'un triangle équilatéral de côté a et de hauteur h. La première étape consiste donc à écrire un algorithme correspondant à ce programme, c'est-à-dire à donner une solution possible au problème. Dans une analyse basée sur l'utilisation finale d'un langage procédural tel que C par exemple, la solution va s'attacher à identifier les fonctions à réaliser : ici, par exemple, nous pouvons identifier deux fonctions, la première permettant le calcul d'un périmètre, la seconde permettant le calcul d'une surface, sachant que :

– le périmètre du carré est égal à $4 \times a$, celui du rectangle est égal à $2 \times a + 2 \times b$, celui du triangle est égal à $3 \times a$ et celui du cercle est égal à $2 \times \pi \times r$;
– la surface du carré est égale à a^2, celle du rectangle est égale à $a \times b$, celle du triangle est égale à $(h \times a)/2$ et celle du cercle est égale à $\pi \times r^2$.

L'algorithme suivant pourra être écrit par exemple pour la fonction périmètre :

```
fonction périmètre :
paramètres entrants : type de l'objet (triangle, carré, rectangle ou cercle)
                      caractéristiques de l'objet : valeur de a, r, h, b
                      ➥ selon l'objet
retourne un entier qui est la valeur du périmètre.
début
    cas objet :
    triangle : retourner (3 × a);
    cercle : retourner (2 ×π× r);
    carré : retourner (4 × a);
    rectangle : retourner (2 × a + 2 × b);
    fin cas;
fin
```

Qui se traduit par exemple en langage C par le programme suivant :

```
#define triangle 1
#define cercle 2
#define carre 3
#define rectangle 4

int perimetre (objet, a, r, h, b)
int objet, a, b, h, r;
{
switch (objet) {
case 1 : return (3 * a);
case 2 : return (2 * π * r);
case 3 : return (4 * a);
case 4 : return (2 * a + 2 * b);}
}
```

Dans une analyse basée sur l'utilisation finale d'un langage objet tel que C++ par exemple, la solution va s'attacher à identifier les objets concernés : ici, par exemple, nous identifierons 4 objets, le cercle, le triangle, le carré et le rectangle, chacun étant associé à deux méthodes, la méthode `perimetre` et la méthode `surface` qui lorsqu'elles sont appelées rendent respectivement le périmètre ou la surface de l'objet concerné. Le carré et le rectangle sont quant à eux tous les deux des quadrilatères pour lesquels le calcul de la surface et du périmètre s'effectue de manière identique (le carré est un rectangle de côtés a et b pour lequel a = b).

Dans cette analyse, nous allons définir une classe `quadrilatere`, une classe `cercle` et une classe `triangle`. Une classe définit une collection d'objets qui partagent les mêmes propriétés et sur lesquels les mêmes traitements peuvent être exécutés. Nous créons ensuite deux objets `carre` et `rectangle`, instances de la classe `quadrilatere`, un objet `rond` instance de la classe `cercle` et un objet `triangle_iso`, instance de la classe `triangle`.

L'algorithme qui en découle peut être de la forme suivante :

```
définition classe quadrilatère (
entier a, b; — les côtés du quadrilatère;
fonction périmètre :: retourner (2 × a + 2 × b);
fonction surface :: retourner (a × b);)

définition classe triangle (
entier a, h; — le côté et la hauteur du triangle;
fonction périmètre :: retourner (3 × a);
fonction surface :: retourner ((a × h)/2);)

définition classe cercle (
entier r;    — le rayon du cercle;
fonction périmètre :: retourner (2 × π × r);
fonction surface :: retourner (π × r²);)
```

Viennent ensuite les définitions les objets `rectangle`, `carre`, `rond` et `triangle_iso`. Cet algorithme peut se traduire par exemple en langage C++ par le programme suivant où l'on ne tient compte que de la classe `quadrilatere` et pour lequel par souci de simplification, les paramètres du rectangle et du carré sont fixés lors de la déclaration des objets associés :

```
class quadrilatere
{
int a, b;
public :
    int perimetre (void);
    int surface (void);
};
int quadrilatere :: perimetre (void)
{
    return (2 * a + 2 * b);
}
int quadrilatere :: surface (void)
{
    return (a * b);
}
int main()
{
    quadrilatere carre (5, 5);
    quadrilatere rectangle (7, 9);
    int surfacec, perimetrec, surfacer, perimetrer;
    surfacec = carre.surface();
    perimetrec = carre.perimetre();
    surfacer = rectangle.surface();
    perimetrer = rectangle.perimetre();
}
```

2.5 CONCLUSION

Ce chapitre nous a permis de comprendre le rôle essentiel d'un ordinateur à savoir exécuter un programme qui est le codage dans un langage compréhensible par la machine d'une solution à un problème posé par un être humain. La solution à ce problème est appelée algorithme.

Le programmeur dispose de plusieurs niveaux de langage pour coder son algorithme :

– le langage de haut niveau est le niveau de programmation le plus utilisé aujourd'hui. C'est un niveau de programmation indépendant de la structure physique de la machine et de l'architecture du processeur de celle-ci ;

– le langage d'assemblage est un langage au contraire dépendant de l'architecture de la machine physique. Il correspond à une forme symbolique du langage machine associé au processeur ;
– le langage machine est un langage composé sur un alphabet binaire. C'est le seul langage exécutable directement par le processeur.

Chapitre **3**

La chaîne de production de programmes

La chaîne de production de programmes désigne le processus permettant la création d'un programme exécutable placé en mémoire centrale à partir d'un programme dit source écrit en langage de haut niveau. Ce processus se décompose en plusieurs étapes (figure 3.1) que nous allons successivement étudier dans ce chapitre : la compilation, l'édition de liens et enfin le chargement.

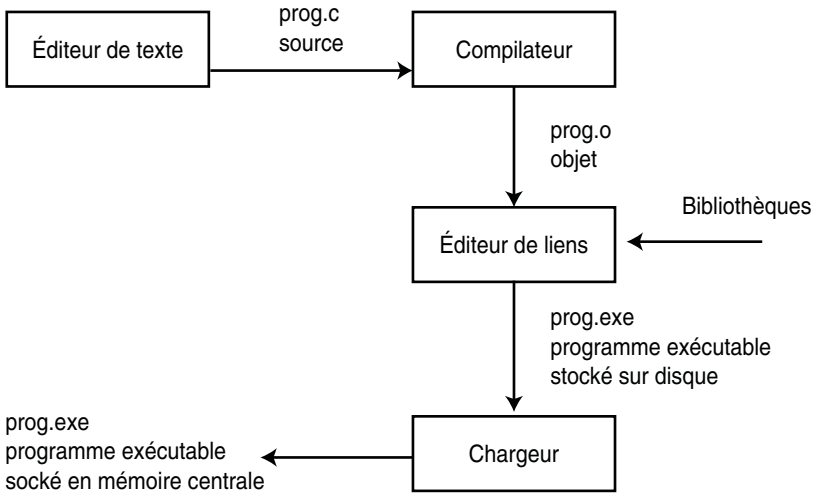


Figure 3.1 Chaîne de production de programmes.

Nous abordons ensuite le fonctionnement d'un outil très courant pour simplifier la construction d'un programme exécutable à partir de plusieurs modules sources : l'utilitaire Make.

3.1 LA COMPILATION

La *compilation* constitue la première étape de la chaîne de production de programmes. Elle permet la traduction d'un programme dit *programme source* écrit le plus souvent en langage de haut niveau vers un programme dit *programme objet* qui est soit directement le langage machine, soit le langage d'assemblage. Le programme objet est stocké sur le disque.

Le compilateur est une application. C'est un logiciel dépendant de la machine physique vers laquelle il doit produire le langage. Ainsi un programme compilé sur une machine A ne s'exécutera pas forcément sur une machine B, notamment si B est différente physiquement de A.

Exemple

La compilation d'un programme écrit en langage C s'obtient en tapant la commande `cc – o prog.c` et produit un fichier objet de nom `prog.o`. De même la compilation d'un programme écrit en Fortran s'obtient en tapant la commande `xlf – c fichier.f` et produit un fichier objet `fichier.o`.

Le travail du compilateur se divise en plusieurs phases :

– l'analyse lexicale (reconnaissance des mots du langage, c'est-à-dire appréhension du vocabulaire) ;
– l'analyse syntaxique (vérification de la syntaxe, c'est-à-dire appréhension de la grammaire) ;
– l'analyse sémantique (vérification de la sémantique, c'est-à-dire appréhension du sens) ;
– l'optimisation et la génération du code objet.

3.1.1 Grammaire et structure d'un langage de haut niveau

Structure d'un langage de haut niveau

Avant de débuter l'étude du fonctionnement du compilateur, nous nous attachons à définir la structure d'un langage haut niveau. La définition d'un langage de haut niveau s'appuie sur :

– un *alphabet* : c'est l'ensemble des symboles élémentaires disponibles dans le langage (caractères, chiffres, signes de ponctuation) ;
– un ensemble de mots encore appelés *lexèmes* : un mot est un groupe de symboles élémentaires admis par le langage dont la structure est donnée par une *grammaire* ;

– des phrases ou *instructions* : une phrase est un groupe de lexèmes du langage dont la structure est elle aussi donnée par une grammaire. Cette structure peut être décrite par un arbre qualifié *d'arbre syntaxique*.

Par exemple, A1 est un mot du langage composé des symboles élémentaires A et 1. A1 = 3 ; constitue une phrase du langage.

Finalement, un programme est constitué comme étant une suite de phrases du langage, chacune des phrases respectant une syntaxe donnée par la grammaire associée au langage. Formellement, une grammaire est définie par :

– un ensemble de *symboles terminaux* qui sont les symboles élémentaires admis dans le langage (exemples : DEBUT, FIN, *, + , = , …);
– un ensemble de *symboles non terminaux* (exemples : <nombre>, <terme>, …);
– un ensemble de règles syntaxiques encore appelées productions (exemple : <nombre> :: = <chiffre> | <nombre> <chiffre>).

Pour décrire la syntaxe d'un langage et spécifier les règles de production de la grammaire associée, on utilise couramment la notation dite *notation de Backus-Naur* ou BNF (*Backus-Naur Form*), développée à l'origine pour décrire la syntaxe du langage Algol 60. Avec ce formalisme, une règle de production s'écrit :

<objet_1 du langage> :: = <objet_2 du langage> | <objet_3 du langage>

et décrit la syntaxe de l'objet_1 du langage comme étant soit l'objet_2 du langage, soit l'objet_3 du langage. « | » code l'alternative et « :: = » sépare un objet de sa description.

Exemple

Soient les deux règles suivantes :

– <nombre> :: = <chiffre> | <nombre> <chiffre>
– <chiffre> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Le nombre 125 peut être décomposé comme suit :

– <nombre> 125 :: = <nombre> 12 <chiffre> 5
– <nombre> 12 :: = <nombre> 1 <chiffre> 2
– <nombre> 1 :: = <chiffre> 1

Un exemple

Les règles de Backus-Naur suivantes permettent la description d'un langage de programmation que nous appellerons L_exemple contenant deux types d'instructions : d'une part des instructions permettant d'opérer des déclarations de variables, d'autre part des instructions de type affectation.

```
<programme> :: = PROGRAM <identificateur> <corps de programme>
<corps de programme> :: = <suite de déclarations> DEBUT <suite
➥ d'affectations> FIN
<suite de déclarations> :: = <déclaration> | <déclaration> <suite
➥ de déclarations>
```

```
<déclaration> :: = INT <identificateur>;
<suite d'affectations> :: = <affectation> | <affectation> <suite
➥ d'affectations>
<affectation> :: = <identificateur> = <terme>; |
➥ <identificateur> = <terme> <opérateur> <terme>;
<terme> :: = <entier> | <identificateur>
<opérateur> :: = + | – | * | /
<identificateur> :: = <lettre> | <lettre> <chiffre>
<entier> :: = <chiffre> | <entier> <chiffre>
<lettre> :: = A | B | C | D | E. | … | X | Y | Z
<chiffre> :: = 0 | 1 | 2 | 3 | 4. | .. | 9
```

Ainsi, la première règle spécifie que l'objet `programme` est construit à partir du symbole terminal `PROGRAM` suivi d'un objet `<identificateur>` et d'un objet `<corps de programme>`. Un objet `<identificateur>` est lui-même décrit comme étant composé soit d'une lettre seule, soit d'une lettre suivie d'un chiffre. Les lettres admises sont les 26 lettres majuscules de l'alphabet et les chiffres admis sont les chiffres allant de 0 à 9. L'objet `<corps de programme>` est par ailleurs défini comme étant un objet `<suite de déclarations>` suivi du symbole terminal `DEBUT`, suivi d'un objet `<suite d'affectations>` suivi du symbole `FIN`. Et ainsi de suite…

Le programme Z suivant a été écrit en respectant la syntaxe dictée par ces règles.

```
PROGRAM Z
INT A;
INT B;
INT C2;
DEBUT
A = 4;
B = A / 2;
C2 = B + A;
FIN
```

3.1.2 Analyse lexicale

La première étape de l'opération de compilation s'appelle *l'analyse lexicale*. L'analyseur lexical (figure 3.2) lit le programme source caractère par caractère, et reconnaît dans cette suite de caractères les lexèmes du langage en s'appuyant sur les règles de Backus Naur les définissant. Par ailleurs, l'analyseur lexical élimine les espaces non significatifs et les commentaires qui n'ont pas de signification propre. Pour rendre plus aisées les étapes ultérieures, il traduit les lexèmes reconnus qui constituent des chaînes de caractères en symboles plus aisément manipulables, par exemple en entiers.

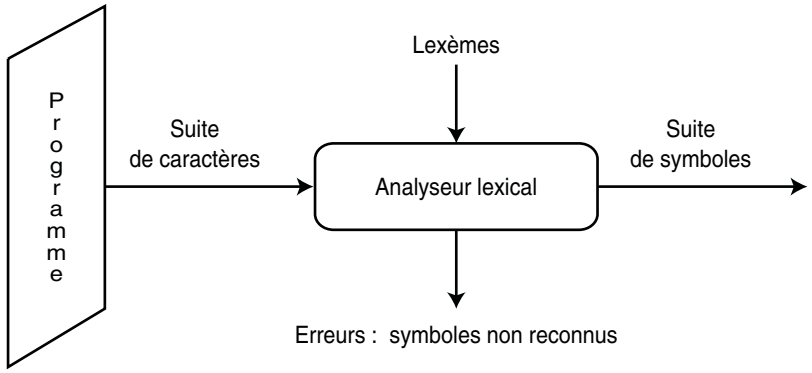


Figure 3.2 Analyseur lexical.

Ainsi, les lexèmes du langage L_exemple peuvent être codés sur l'ensemble Z des nombres entiers positifs, négatifs ou nuls en appliquant les règles suivantes :

```
cas (type_lexème reconnu) :
entier : codage_lexème = valeur de l'entier;
symbole + : codage_lexème = – 1;
symbole – : codage_lexème = – 2;
symbole * : codage_lexème = – 3;
symbole / : codage_lexème = – 4;
symbole = : codage_lexème = – 5;
symbole ; : codage_lexème = – 6;
symbole PROGRAM : codage_lexème = – 7;
symbole DEBUT : codage_lexème = – 8;
symbole FIN : codage_lexème = – 9;
symbole INT : codage_lexème = – 10;
identificateur lettre seule : – (position_lettre_alphabet + 10);
identificateur lettre chiffre : – ((position_lettre_alphabet + 10) +
➥ 26 (chiffre + 1));
fin cas;
```

Avec ce codage qui est totalement bijectif :

– un lexème entier reçoit une valeur positive ou nulle ;
– un lexème symbole du langage reçoit une valeur dans l'ensemble [– 10, – 1] ;
– un identificateur reçoit une valeur dans l'ensemble [– 296, – 11] : en effet, l'identificateur A est codé par la valeur – 11 tandis que l'identificateur Z9 est codé par la valeur – (36 + 260).

Le programme Z correspondra à l'issue de l'analyse lexicale à la suite d'entiers : – 7 (PROGRAM), – 36 (Z), – 10 (INT), – 11 (A), – 6 (;), – 10 (INT), – 12 (B), – 6 (;), – 10 (INT), – 91 (C2), – 6 (;), – 8 (DEBUT), – 11 (A), – 5 (=), 4 (4), – 6 (;), – 12 (B), – 5 (=), – 11 (A), – 4 (/), 2 (2), – 6 (;), – 91 (C2), – 5 (=), – 12 (B), – 1 (+), – 11 (A), – 6 (;), – 9 (FIN).

Des erreurs peuvent survenir et correspondent à l'utilisation de symboles non conformes au langage. Ainsi une phrase telle que `INT g5` provoquera une erreur lexicale dans le cadre du langage L_exemple car le symbole g ne fait pas partie de l'alphabet du langage (seules les lettres majuscules sont admises). Par contre la phrase `G 5 = 2;` sera reconnue par l'analyseur lexical comme étant équivalente à la suite : – 17 (G) 5 (5) – 5 (=) 2 (2) – 6 (;).

La phase d'analyse lexicale s'apparente à notre propre processus de lecture qui nous permet de reconnaître et former les mots dans la suite de caractères d'un texte.

3.1.3 Analyse syntaxique

L'étape suivante dans le processus de compilation est celle de l'analyse syntaxique. L'analyseur syntaxique (figure 3.3) analyse la suite de symboles issus de l'analyseur lexical et vérifie si cette suite de symboles est conforme à la syntaxe du langage telle qu'elle est définie par les règles de Backus-Naur. Pour cela, l'analyseur syntaxique essaye de construire *l'arbre syntaxique* correspondant au programme. Dans cet arbre, les feuilles correspondent aux symboles issus de l'analyse lexicale et les nœuds intermédiaires correspondent aux objets grammaticaux. Si l'analyseur syntaxique ne parvient pas à construire l'arbre syntaxique du programme compilé, alors cela traduit le fait que la syntaxe du programme est erronée.

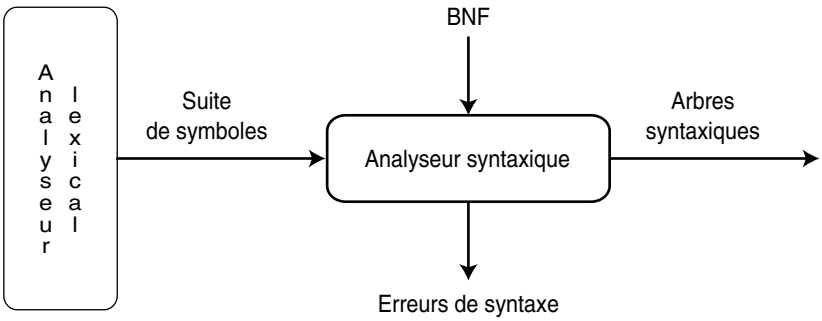


Figure 3.3 Analyseur syntaxique.

L'arbre syntaxique est construit par application successive des règles de Backus-Naur définissant la syntaxe du langage, à l'ensemble des symboles issus de l'analyse lexicale. Cette construction se poursuit soit jusqu'à ce que le dernier symbole ait été pris en compte avec succès (la syntaxe est correcte), soit jusqu'à ce qu'il ne soit plus possible de faire correspondre aucune des règles de Backus-Naur existantes avec la suite de symboles restants (la syntaxe est erronée).

Le programme Z donné en exemple au paragraphe 3.1.1 conduit à l'arbre syntaxique donné par la figure 3.4.

Reprenons à présent la phrase `G 5 = 2;` reconnue par l'analyseur lexical comme étant équivalente à la suite : – 17 (G) 5 (5) – 5 (=) 2 (2) – 6 (;). L'arbre syntaxique

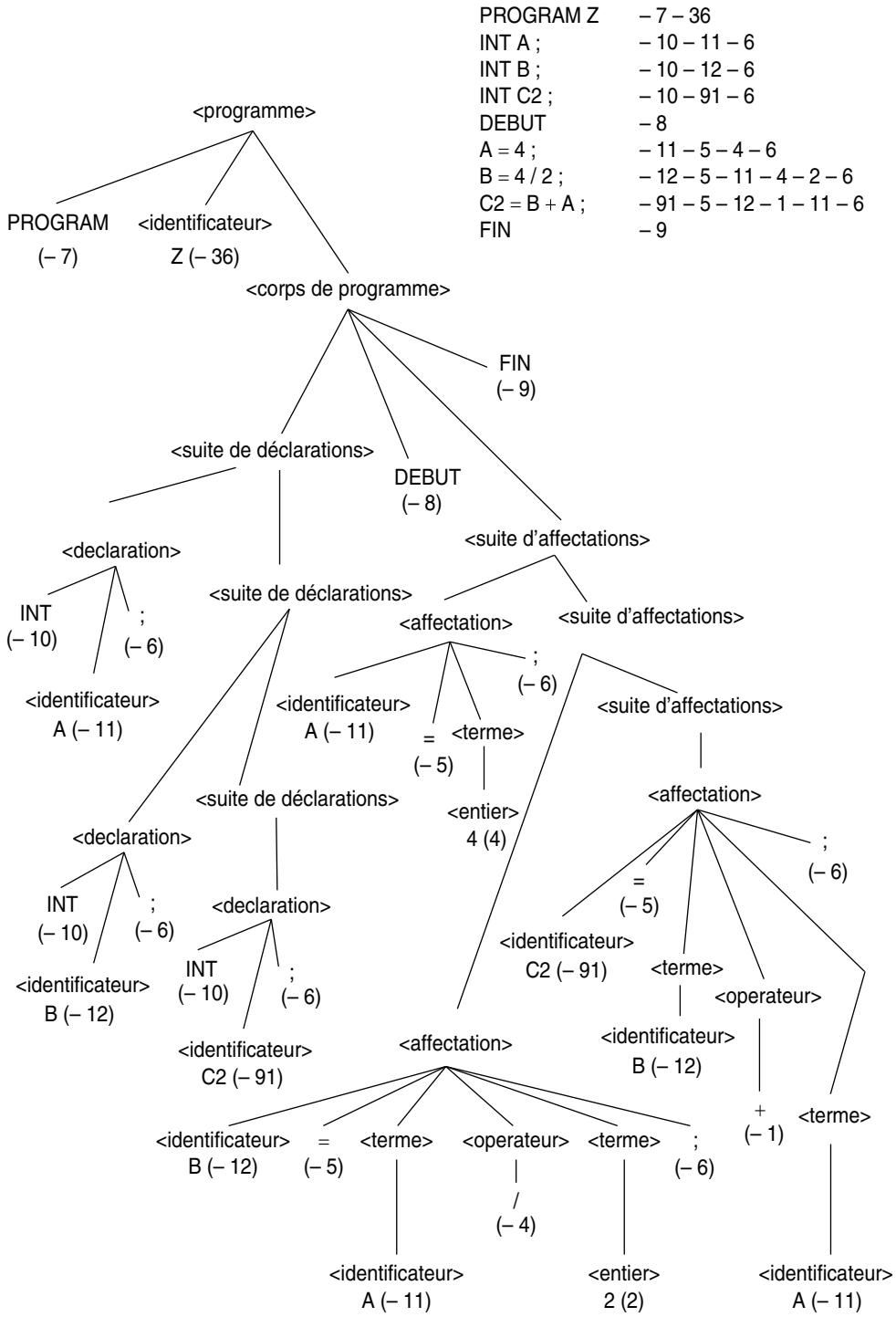


Figure 3.4 Arbre syntaxique.

correspondant à une telle phrase ne peut pas être construit. En effet, la règle de production relative à un objet de type `affectation` échoue sur le deuxième symbole. En effet, le symbole « 5 » est rencontré à la place du symbole « = ». Une erreur syntaxique est donc levée (figure 3.5).

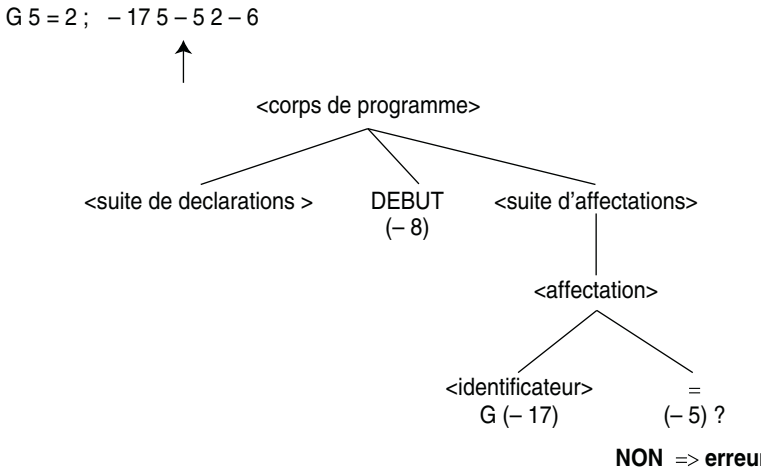


Figure 3.5 Arbre syntaxique impossible à construire sur l'analyse de G 5 = 2.

La construction des arbres syntaxiques fait apparaître la nécessité d'avoir défini la grammaire du langage sans ambiguïté afin qu'à aucun moment, une suite de symboles issue de l'analyse lexicale puisse correspondre à plusieurs règles de production différentes, ce qui conduirait à une interprétation du sens du programme peut-être différente de celle voulue par le programmeur.

3.1.4 Analyse sémantique

L'analyse sémantique constitue la troisième étape d'analyse du compilateur. Elle a pour but d'associer un sens aux différentes phrases du programme source. Ce travail recouvre principalement deux points différents :

– reconnaître les objets manipulés et analyser leurs propriétés : quel est le type de l'objet, sa durée de vie, sa taille et son adresse ?
– contrôler que l'utilisation de ces objets se fait de manière cohérente : à ce niveau, l'analyse sémantique recherche les erreurs de typage, les déclarations multiples, absentes ou inutiles, les expressions incohérentes.

Soit, par exemple, le programme Ada suivant :

```
procedure prog_ada is
float i;                    — variable i de type réel
integer A, B;               — variables A et B de type entier
tab : array(1.5) of float;  — tableau de 5 réels
```

```
begin
A : = 5;
B : = A/2;
for (i = 1 to 5)
loop
    tab(i) : = B + i;
end loop;
end;
```

L'analyseur sémantique signale dans ce programme deux erreurs sémantiques dues à des incohérences de type :

– le résultat de l'opération A/2 donne un résultat qui est de type réel alors que la variable B est déclarée de type entier ;
– la variable i qui permet de parcourir l'ensemble des cases du tableau tab dans la boucle est de type réel.

```
procedure prog_ada is
float B;                    — variable B de type réel
integer A, i;               — variables A et i de type entier
tab : array(1.5) of float;  — tableau de 5 réels
begin
A : = 5;
B : = A/2;
for (i = 1 to 5)
loop
    tab(i) : = B + i;
end loop;
end;
```

Dans le programme corrigé comme ci-dessus, la phrase `tab(i) : = B + i;` pose encore problème car elle amène à additionner entre elles une variable de type entier (i) avec une variable de type réel (B). Dans ce dernier cas, le compilateur peut insérer de lui-même un ordre de conversion de la variable i du type entier vers réel, comme l'explicite l'exemple suivant.

```
procedure prog_ada is
float B;                    — variable i de type réel
integer A, i;               — variables A et B de type entier
tab : array(1.5) of float;  — tableau de 5 réels
begin
A : = 5;
B : = A / 2;
for (i = 1 to 5)
loop
    tab(i) : = B + (entierversreel)i;
end loop;
end;
```

3.1.5 Génération du code final

La table des symboles

Tout au long des trois phases d'analyse lexicale, d'analyse syntaxique et d'analyse sémantique, le compilateur construit une table regroupant toutes les informations utiles sur les objets apparaissant dans le programme compilé. Cette table qui comprend une entrée par objet reconnu est appelée *la table des symboles*. Chaque entrée spécifie les propriétés de l'objet associé, c'est-à-dire notamment son type, sa taille et son adresse. Pour pouvoir affecter une adresse à chacun des objets, le compilateur manipule un compteur appelé *compteur d'emplacements*, initialisé à 0 pour le premier objet et incrémenté ensuite de la taille de chaque objet rencontré. L'adresse d'un objet est égale à la valeur du compteur d'emplacements.

Ainsi le tableau 3.1 suivant peut correspondre à la table des symboles construite lors de la compilation du programme Z.

Tableau 3.1 TABLE DES SYMBOLES DU PROGRAMME Z.

Nom de l'objet	Type	Taille (octets)	Adresse
A	entier	4	$(0)_{16}$
B	entier	4	$(4)_{16}$
C2	entier	4	$(8)_{16}$

Génération du code et optimisation

La génération du code constitue l'étape ultime de la compilation. Elle consiste à produire dans un fichier objet le code machine équivalent au code du langage de haut niveau. Elle se décompose généralement en trois étapes : la génération d'un code intermédiaire, l'optimisation de ce code intermédiaire, enfin, la génération du code final.

Le code obtenu à ce niveau est appelé *code relogeable*, c'est-à-dire que toutes les adresses des objets figurant dans ce code sont calculées en considérant que l'adresse du premier octet du code est égale à 0.

➤ Génération d'un code intermédiaire

La génération du code intermédiaire consiste à remplacer les phrases reconnues par des macros plus aisément manipulables. Les objets sont eux-mêmes remplacés par l'adresse qui leur a été affectée dans la table des symboles. Les macros ne sont pas des instructions machine; elles s'en différencient notamment par le fait qu'elles ne font aucune référence aux registres de la machine et restent donc indépendantes de l'architecture de celle-ci. Ainsi le programme Z suivant pourra être transformé de façon à ce que `INIT a b` soit une macro d'initialisation qui initialise l'objet `a` avec la valeur `b`, la macro `DIV a b c` correspond à la division de l'objet `b` par l'objet `c` avec stockage du résultat dans l'objet `a` et enfin la macro `ADD a b c` représente l'addition de l'objet `b` par l'objet `c` avec stockage du résultat dans l'objet `a`.

`PROGRAM Z`	`Z :`
`INT A;`	$(0)_{16}$
`INT B;`	$(4)_{16}$
`INT C2;`	$(8)_{16}$
`DEBUT`	
`A = 4;`	INIT $(0)_{16}$ 4
`B = A / 2;`	DIV $(4)_{16}$ $(0)_{16}$ 2
`C2 = B + A;`	ADD $(8)_{16}$ $(4)_{16}$ $(0)_{16}$
`FIN`	STOP

➤ Optimisation du code intermédiaire

L'optimisation de code vise à produire un code machine plus performant, c'est-à-dire d'une part un code dont l'exécution est plus rapide et d'autre part un code plus compact dont l'encombrement mémoire est moindre. Diverses améliorations peuvent être mises en œuvre ; elles diffèrent grandement d'un compilateur à l'autre. Les plus courantes sont :

– la réduction des expressions constantes : elle consiste à effectuer les calculs arithmétiques qui ne comportent que des opérandes constants et à remplacer l'expression arithmétique par son résultat. Par exemple, dans le programme Z, l'expression `B = A/2;` peut être calculée puisque la valeur de A est connue comme étant égale à 4. `B = A/2;` est donc remplacée par `B = 2;`

– simplification des boucles et pré-évaluation des expressions constantes : elle consiste à sortir les expressions invariantes des corps de boucles et à les placer juste devant ceux-ci. Ainsi, la boucle suivante :

```
for i = 1 to n
loop
    j : = 3;
    i : = i + 1;
end loop;
```

est transformée de la manière suivante :

```
j : = 3;
for i = 1 to n
loop
    i : = i + 1;
end loop;
```

– simplification des boucles et réduction de puissance des opérateurs : elle consiste à remplacer au sein d'une boucle une opération puissante et jugée plus coûteuse telle que la multiplication par une opération plus simple et moins coûteuse telle que l'addition. Ainsi le code suivant :

```
for i = 1 to n
loop
    a = i * 5;
end loop;
```

est transformé de la manière suivante :

```
for i = 1 to n
loop
    a = a + 5;
end loop;
```

On remarquera que l'ensemble des opérations présentes dans le programme Z peuvent être effectuées par la phase d'optimisation et que le programme Z peut ainsi se résumer à trois macros d'initialisation INIT.

```
PROGRAM Z                  Z :
INT A;                     (0)16
INT B;                     (4)16
INT C2;                    (8)16
DEBUT
A = 4;                     INIT (0)16 4
B = A / 2;    → B = 2;     DIV (4)16 (0)16 2      → INIT (4)16 2
C2 = B + A;   → C2 = 6;    ADD (8)16 (4)16 (0)16  → INIT (8)16 6
FIN                        STOP
```

➤ Génération du code final

L'étape finale est celle de génération du code final. Les macros sont remplacées par les instructions machine correspondantes avec utilisation des registres et les objets sont remplacés par leur adresse. Les adresses générées (données ici en base 16) sont calculées à partir de la valeur 0.

Z :	adresse	instruction	commentaire
$(0)_{16}$	$(0)_{16}$		
$(4)_{16}$	$(4)_{16}$		
$(8)_{16}$	$(8)_{16}$		
INIT $(0)_{16}$ 4	$(C)_{16}$	00000 000 0001 0000000000000100	R1 ← 4
	$(10)_{16}$	00001 001 0001 0000000000000000	R1 → $(0)_{16}$
INIT $(4)_{16}$ 2	$(14)_{16}$	00000 000 0001 0000000000000010	R1 ← 2
	$(18)_{16}$	00001 001 0001 0000000000000100	R1 → $(4)_{16}$
INIT $(8)_{16}$ 6	$(1C)_{16}$	00000 000 0001 0000000000000110	R1 ← 6
	$(20)_{16}$	00001 001 0001 0000000000001000	R1 → $(8)_{16}$
STOP			

3.2 L'ÉDITION DES LIENS

3.2.1 Rôle de l'éditeur de liens

L'édition des liens constitue la deuxième étape du processus de production de programmes. Elle permet la construction du programme exécutable final en résolvant les liens vers les bibliothèques ou entre différents modules objets construits à l'aide de *compilations séparées*.

Nous avons vu que le compilateur traduit un programme écrit en langage de haut niveau vers un programme objet, écrit en langage machine. Toute modification intervenant au niveau du programme source en langage de haut niveau ne peut être prise en compte que par le biais d'une opération de compilation, qui traduit la modification en langage machine. Or, la durée d'une opération de compilation est fortement liée à la taille du programme source à compiler : plus celui-ci est gros, plus la compilation est longue. C'est pourquoi on préfère souvent, pour l'écriture d'un gros programme, découper celui-ci en *modules*, compilables séparément. Un module correspond souvent à une unité logique du programme : une procédure avec les données qu'elle manipule, le programme principal, etc. La compilation séparée de chacun des modules source donne autant de modules objet. La construction du programme exécutable final est alors à la charge de l'éditeur de liens, qui doit relier les modules objets entre eux pour construire un seul programme exécutable.

Par ailleurs, les modules utilisateurs vont souvent faire appel à des fonctions prédéfinies disponibles dans des *bibliothèques* du langage de haut niveau utilisé : par exemple, les fonctions mathématiques, les fonctions graphiques, l'interface des fonctions du système. C'est également le rôle de l'éditeur de liens que de faire la liaison entre les appels à ces fonctions dans le code source utilisateur et le code de ces fonctions, stocké dans les bibliothèques. Pour effectuer ces opérations de liaison entre modules utilisateurs et fonctions de bibliothèques, l'éditeur de liens utilise des informations de liaison, écrites en entête de chaque module par le compilateur : ce sont *les liens à satisfaire* et *les liens utilisables* des modules.

3.2.2 Fonctionnement de l'éditeur de liens

Nous allons à présent décrire le fonctionnement de l'éditeur de liens. À partir d'un exemple, nous commençons par définir les notions de liens utilisables et de liens à satisfaire.

Notion de lien utilisable et de lien à satisfaire

Exemple

Considérons une application de recherche et de réservation de livres dans le catalogue informatique d'une bibliothèque. Le logiciel a été structuré sous la forme de trois modules sources distincts :

- le module `interface` contient la procédure chargée du dialogue avec le client ;
- le module `recherche` contient les procédures de recherche dans le catalogue de la bibliothèque et la procédure de réservation d'un livre ;
- le module `affichage` contient les procédures permettant l'affichage des résultats des recherches menées dans le catalogue.

Voici un code source pour chacun des modules, volontairement incomplet et simplifié, qui porte l'accent sur les dépendances entre les modules.

```
module interface
programme principal;
début
chaîne de caractères réponse, cote_livre, mot_clé;
liste liste_livre;
tant que (réponse < > 3)
faire
    poser_question ("  que voulez-vous faire ?
                       chercher un livre ? 1
                       réserver un livre ? 2
                       quitter ? 3");
    lire_réponse(réponse);
cas (réponse) :
    1 :  poser_question ("donnez un mot-clé :");
         lire_réponse (mot_clé);
         chercher_livre(liste_livre, mot_clé);
         afficher(liste_livre);
    2 :  poser_question ("donnez la cote du livre :");
         lire_réponse (cote_livre);
         réserver_livre(cote_livre);
    3 :  rien;
fin cas;
fin tant que;
fin
procédure lire_réponse (out chaîne : chaîne de caractères);
début
code qui lit une chaîne de caractères au clavier
appelle une fonction lit_clavier de la bibliothèque du langage
fin
procédure poser_question (in chaîne : chaîne de caractères);
début
code qui écrit une chaîne de caractères sur l'écran
appelle une fonction affiche_écran de la bibliothèque du langage
fin
fin module;
module recherche;
export chercher_livre, réserver_livre;
procédure chercher_livre(out liste liste_livre, in chaîne de caractères
mot_clé);
début
code qui effectue une recherche dans le catalogue de la bibliothèque
appelle une fonction lire_fichier de la bibliothèque du langage
fin;
```

```
procédure réserver_livre(in chaîne de caractères cote_livre);
début
code qui effectue une réservation de livre dans le catalogue
➥ de la bibliothèque
appelle une fonction écrire_fichier de la bibliothèque du langage
fin;
fin module

module affichage;
export afficher;
procédure afficher(in liste liste_livre);
début
code qui effectue un affichage des livres de la liste liste_livre;
appelle une fonction affiche_écran de la bibliothèque du langage
fin
fin module;
```

Le module `interface` fait appel :

– aux procédures `poser_question` et `lire_réponse` qui sont définies dans le module même ;
– aux procédures `chercher_livre`, `réserver_livre` et `afficher` qui ne sont pas définies localement mais dans les deux autres modules (respectivement `recherche` et `affichage`) ;
– aux procédures `lit_clavier` et `affiche_écran` qui elles aussi ne sont pas définies localement mais appartiennent à une bibliothèque du langage.

Pour sa part, le module `recherche` fait appel aux procédures `lire_fichier` et `écrire_fichier` qui ne sont pas définies localement mais appartiennent à une bibliothèque du langage.

Enfin, le module `affichage` fait appel à la procédure `affiche_écran` qui n'est pas définie localement mais appartient à une bibliothèque du langage.

Pour chacun de ces modules, ces appels à des objets qui ne sont pas définis localement correspondent à des *importations d'objets*. Ainsi, le module `interface` importe les objets `chercher_livre`, `réserver_livre` et `afficher` ainsi que les objets `lit_clavier` et `affiche_écran`. Le module `rechercher` importe les objets `lire_fichier` et `écrire_fichier` et le module `affichage` importe l'objet `affiche_écran`.

Cependant pour que des objets puissent être importés dans un module, il faut que le module dans lequel ces objets sont définis exporte ces mêmes objets, c'est-à-dire qu'il rende ces objets accessibles depuis l'extérieur. C'est le rôle des lignes de code `export chercher_livre, réserver_livre;` et `export afficher;` placées en en-tête des modules `rechercher` et `affichage`. On parlera ici *d'exportation d'objets*. La figure 3.6 schématise les relations entre les trois modules.

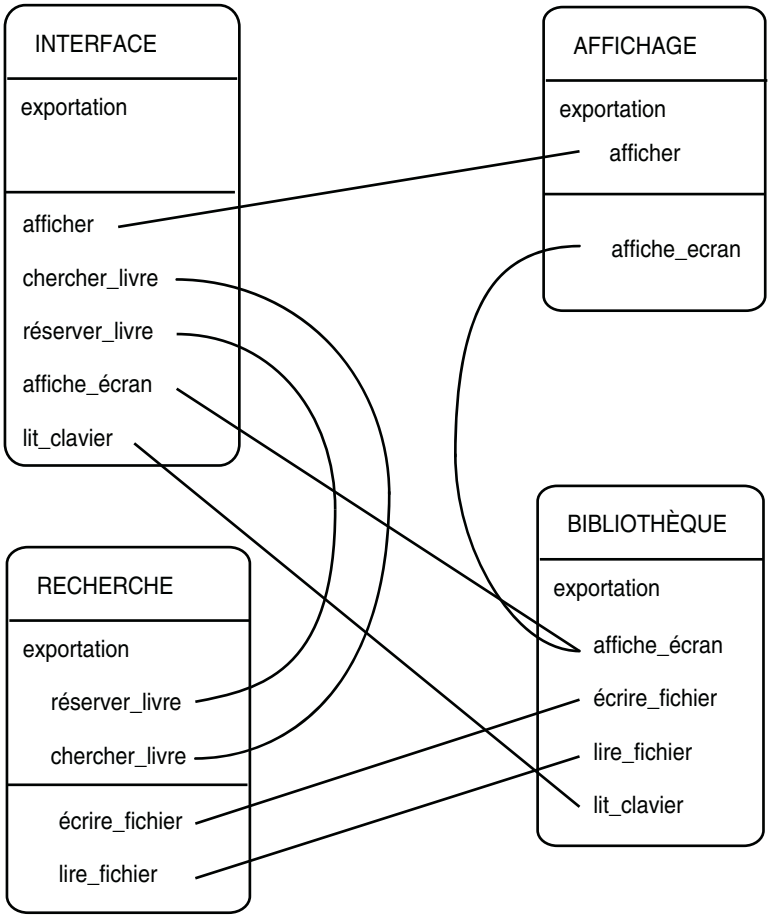


Figure 3.6 Relation d'importation et d'exportation d'objets entre les modules `affichage`, `interface` et `rechercher`.

Lors de la compilation séparée de chacun de ces modules, le code objet final ne va pas pouvoir être complètement généré puisque les modules contiennent des objets qui ne sont pas déclarés localement. Le compilateur notamment ne connaît pas l'adresse d'implantation de ces objets. Aussi, à chaque fois que le compilateur va trouver dans un module, un objet importé, c'est-à-dire un objet utilisé dans le module mais non défini dans le module, il va générer un *lien à satisfaire* concernant cet objet. Le lien à satisfaire correspond à une demande du compilateur à l'intention de l'éditeur de liens pour que celui-ci trouve dans un autre module, la déclaration de l'objet cherché et ainsi son adresse. L'éditeur de liens va rechercher l'objet en question parmi les objets exportés par d'autres modules. Aussi, lorsque le compilateur dans un module rencontre des déclarations d'objets exportés, il crée un *lien utilisable*. Un lien utilisable signale à l'éditeur de liens la présence dans un module d'un objet exporté.

➤ Exemples d'importation et d'exportation d'objets dans les langages de programmation de haut niveau

Dans un langage tel que Ada, le concept de module se traduit par celui de *paquetage*. Un paquetage est divisé en deux parties : une partie spécification qui contient la déclaration des objets du paquetage accessible depuis d'autres modules et une partie corps qui contient la définition des objets eux-mêmes, qu'ils soient exportés ou restent privés. Dans ces paquetages, l'importation d'objets externes provenant de bibliothèques du langage s'effectue grâce à la clause d'importation `with`. Ainsi, le module `recherche` s'écrit :

```
package recherche is
    procedure chercher_livre (liste_livre : out liste, mot_cle :
    ➥ in string(1.20));
    procedure reserver_livre (cote_livre : in string(1.20));
end;
with la_bibliotheque; — importation des objets de la bibliotheque
package body recherche is
procedure chercher_livre (liste_livre : out liste, mot_cle :
➥ in string(1.20)) is
begin
corps de la procedure qui utilise la fonction lire_fichier
➥ de la bibliotheque la_bibliotheque :
➥ appel sous forme la_bibliotheque.lire_fichier
end;
procedure reserver_livre (cote_livre : in string(1.20)) is
begin
corps de la procedure qui utilise la fonction ecrire_fichier
➥ de la bibliotheque la_bibliotheque :
➥ appel sous forme la_bibliotheque.ecrire_fichier
end;
end;
```

Dans un langage tel que C, l'exportation des objets s'effectue par le biais de la clause `extern`. L'objet exporté ainsi est rendu accessible à tout autre module.

➤ Définition des notions de liens utilisables et de liens à satisfaire

Trois catégories d'objets peuvent donc être répertoriées au sein d'un module :

– les objets *internes* au module, inaccessibles de l'extérieur car ils ne sont pas exportés (*objet privé*). À ces objets, le compilateur ne fait correspondre aucun lien (exemple l'objet `lire_reponse` dans le module `interface`);
– les objets internes au module mais accessibles de l'extérieur (*objet exporté ou public*). À ces objets, le compilateur fait correspondre un *lien utilisable (LU)* (exemple l'objet `chercher_livre` dans le module `recherche`);

– les objets n'appartenant pas au module, mais utilisés par le module (*objet importé ou externe*). À ces objets, le compilateur fait correspondre un *lien à satisfaire (LAS)* (exemple l'objet `chercher_livre` dans le module `interface`).

C'est donc le compilateur qui génère les liens à satisfaire et les liens utilisables en fonction des catégories d'objets qu'il rencontre lors de la compilation d'un module. Les liens à satisfaire et les liens utilisables sont placés en entête du module. Leur forme est la suivante :

– pour un lien utilisable, c'est un couple <nom_de_l'objet, valeur> où valeur est l'adresse de l'objet dans le module qui contient sa déclaration ;
– pour un lien à satisfaire, c'est un couple <nom_de_l'objet, adr_1, adr_2, .., adr_last > où adr_1, adr_2, .., adr_last sont les adresses dans le module où l'objet est utilisé.

Outre ces informations, le compilateur indique également en en-tête de chaque module sa taille en octets.

Les modules `interface`, `recherche` et `affichage` à l'issue de la compilation sont précédés des en-têtes suivants :

```
module interface.o;
taille (module) = 512 Ko
LAS <chercher_livre, adr_m_interface_1>
LAS <afficher, adr_m_interface_2>
LAS <réserver_livre, adr_m_interface_3>
LAS <lit_clavier, adr_m_interface_4>
LAS <affiche_écran, adr_m_interface_5>
code objet translatable correspondant au module dans lequel les adresses
➥ adr_m_interface_1, adr_m_interface_2, adr_m_interface_3, adr_m_interface_4
➥ et adr_m_interface_5 correspondent aux appels vers les 5 procédures
➥ chercher_livre, afficher, réserver_livre, lit_clavier, et affiche_écran.
fin module;
```

```
module recherche.o;
taille (module) = 140 Ko
LU <chercher_livre, adr_m_recherche_1>
LU <réserver_livre, adr_m_recherche_2>
LAS <lire_fichier, adr_m_recherche_3>
LAS <écrire_fichier, adr_m_recherche_4>
code objet translatable correspondant au module dans lequel les adresses
➥ adr_m_recherche_1, adr_m_recherche_2 correspondent à l'adresse
➥ des procédures chercher_livre et réserver_livre dans le module
➥ et où adr_m_recherche_3 et adr_m_recherche_4 correspondent aux appels
➥ vers les procédures lire_fichier et écrire_fichier.
fin module;
```

```
module affichage.o;
taille (module) = 128 Ko
LU <afficher, adr_m_affichage_1>
```

```
LAS <affiche_écran, adr_m_affichage_2>
code objet translatable correspondant au module dans lequel l'adresse
➥ adr_m_affichage_1 correspond à l'adresse de la procédure afficher
➥ dans le module et où adr_m_affichage_2 correspond à l'appel
➥ vers la procédure affiche_écran.
fin module;
```

Fonctionnement de l'éditeur de liens

Le rôle de l'éditeur de liens consiste donc à associer chaque lien à satisfaire avec le lien utilisable correspondant de manière à pouvoir assigner une adresse à chacun des objets présents dans les modules. L'éditeur de liens est appelé en lui fournissant en paramètre les noms de chacun des modules objets entrant en compte dans la construction du programme exécutable final.

Exemple

La commande `ld – o fich_exe module1.o module2.o` permet la construction du programme exécutable `fich_exe` à partir des modules objets `module1.o module2.o` sous un système de type Unix.

La commande `ald module_principal` permet la construction d'un programme exécutable à partir de modules objets issus d'une compilation Ada. Tous les modules ont été au moment de la compilation rangés dans une bibliothèque adalib créée au moyen de la commande `amklib adalib` et `module_principal` est le nom du module objet dans la bibliothèque contenant le programme principal.

L'édition des liens pour un ensemble de modules objets aboutissant à la construction d'un programme exécutable se déroule en trois étapes : la construction de la *carte d'implantation* du programme final, la construction de la table des liens et la construction du programme exécutable final.

➤ Construction de la carte d'implantation

La première étape de l'édition des liens est donc la construction de la *carte d'implantation* du programme final. Cette étape consiste à placer les uns derrière les autres les différents modules relogeables générés par le compilateur et à calculer les *adresses d'implantation* de ces modules, c'est-à-dire l'adresse du premier octet de chacun des modules. Le premier module garde une adresse d'implantation égale à 0 car il est le premier dans la carte. Les modules suivants par contre ont une adresse d'implantation qui est translatée de la taille en octets des modules qui les précèdent. La taille des modules objets est une information délivrée par le compilateur. Ainsi sur la figure 3.7, le module B garde une adresse d'implantation égale à 0. Le module C a une adresse d'implantation translatée de taille (B) et le module A a une adresse d'implantation qui est translatée de taille (B) + taille (C).

Si nous considérons la construction du programme exécutable final permettant d'accéder au catalogue de notre bibliothèque à partir des trois modules `interface.o`,

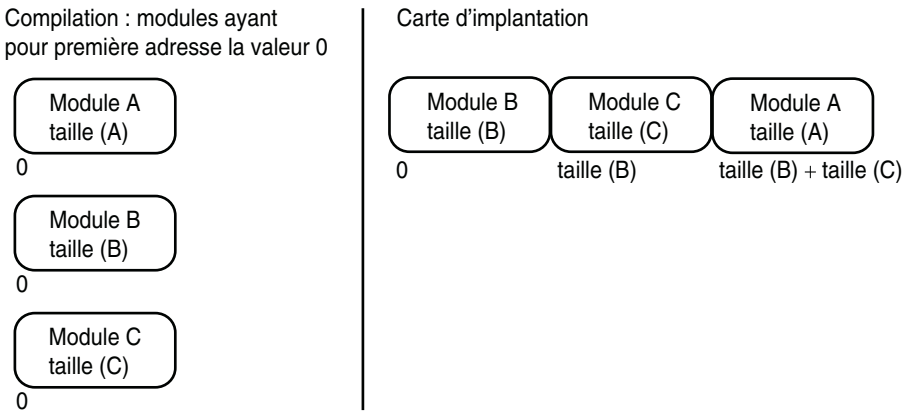


Figure 3.7 Carte d'implantation.

`recherche.o` et `affichage.o`, en les supposant placés dans cet ordre dans la carte d'implantation du programme, les adresses d'implantation suivantes sont obtenues :

– le module `interface.o` a une adresse d'implantation égale à 0;
– le module `recherche.o` a une adresse d'implantation égale à 512 Ko;
– le module `affichage.o` a une adresse d'implantation égale à 652 Ko;
– la taille totale du programme exécutable final est 780 Ko.

➤ Construction de la table des liens

La deuxième étape s'occupe de construire la table des liens : cette table a pour rôle de permettre la résolution des liens c'est-à-dire la mise en relation des liens à satisfaire avec les liens utilisables correspondants. Chaque entrée de la table des liens est constituée de deux champs : le nom du lien c'est-à-dire le nom de l'objet référencé par ce lien et son adresse dans la carte d'implantation construite à l'étape précédente. La table est construite par la prise en compte dans chacun des modules entrant dans la composition finale du programme exécutable des liens à satisfaire et des liens utilisables.

L'algorithme suivant explicite le déroulement de cette étape pour un ensemble de modules entrant dans la construction d'un programme exécutable final.

```
structure entrée_table_des_liens – une entrée de la table des liens
    nom_objet : chaîne de caractères;
    adresse_objet : type_adresse;
fin structure;
structure lien              — un lien de type LAS ou LU
    genre : type_genre;     — LAS ou LU
    nom_objet : chaîne de caractères;
    adresse_objet : type adresse;
fin structure;
```

```
table des liens : tableau (1.1000) de structure entrée_table_des_liens;
➥ — la table des liens
le_lien : structure lien;
pour tous les modules de la liste
faire
se placer sur le premier lien de l'en-tête;
tant que (il y a un lien dans l'en-tête du module)
faire
si  (table_des_liens.nom_objet < > lien.nom_objet pour toutes
➥ les entrées de la table)
alors
    — l'objet n'existe pas dans la table
    créer une nouvelle entrée nv_entree dans table_des_liens
    ➥ pour laquelle nom_objet = lien.nom_objet;
    si   lien.genre = LU
    alors
         nv_entree.adresse_objet : = lien.adresse_objet +
         ➥ adr_implantation_module;
    fsi
    si   lien.genre = LAS
    alors
         nv_entree.adresse_objet : = indéfinie;
    fsi
sinon
    il existe une entrée dans la table pour laquelle
    ➥ nom_objet = lien.nom_objet : c'est entree_trouvee
    si   (lien.genre = LU et entree_trouvee.adresse_objet = indefinie)
    alors
         — resolution LAS/LU
         entree_trouvee.adresse_objet : = lien.adresse_objet +
         ➥ adr_implantation_module;
    fsi
    si   (lien.genre = LU et entree_trouvee.adresse_objet < > indefinie)
    alors
              — deux LU pour un seul LAS : ERREUR
    fsi
    si   (lien.genre = LAS)
    alors
         ne rien faire
    fsi
fsi
passer au lien suivant;
fait
passer au module suivant;
fait
```

L'application de cet algorithme aux trois modules `interface.o`, `recherche.o` et `affichage.o` mène à construire la table des liens suivante :

– prise en compte du module `interface.o` (tableau 3.2) :

Tableau 3.2 TABLE DES LIENS APRÈS LA PRISE EN COMPTE DU MODULE INTERFACE.O.

Nom de l'objet	Adresse dans la carte d'implantation	Commentaire
chercher_livre	indéfinie	LAS
afficher	indéfinie	LAS
réserver_livre	indéfinie	LAS
lit_clavier	indéfinie	LAS
affiche_écran	indéfinie	LAS

– prise en compte du module `recherche.o` (tableau 3.3) :

Tableau 3.3 TABLE DES LIENS APRÈS LA PRISE EN COMPTE DU MODULE RECHERCHE.O.

Nom de l'objet	Adresse dans la carte d'implantation	Commentaire
chercher_livre	adr_m_recherche_1 + 512 Ko	Résolution LAS/LU
afficher	indéfinie	LAS
réserver_livre	adr_m_recherche_2 + 512 Ko	Résolution LAS/LU
lit_clavier	indéfinie	LAS
affiche_écran	indéfinie	LAS
lire_fichier	indéfinie	LAS
écrire_fichier	indéfinie	LAS

– prise en compte du module `affichage.o` (tableau 3.4) :

Tableau 3.4 TABLE DES LIENS APRÈS LA PRISE EN COMPTE DU MODULE AFFICHAGE.O.

Nom de l'objet	Adresse dans la carte d'implantation	Commentaire
chercher_livre	adr_m_recherche_1 + 512 Ko	Résolution LAS/LU
afficher	adr_m_affichage_1 + 652 Ko	Résolution LAS/LU
réserver_livre	adr_m_recherche_2 + 512 Ko	Résolution LAS/LU
lit_clavier	indéfinie	LAS
affiche_écran	indéfinie	LAS
lire_fichier	indéfinie	LAS
écrire_fichier	indéfinie	LAS

➤ Prise en compte des bibliothèques du langage

À l'issue de la prise en compte des différents modules utilisateurs entrant dans la construction du programme exécutable final, il peut rester des entrées de la table des liens pour lesquelles le champ adresse est toujours égal à la valeur indéfinie. À ce niveau, l'éditeur de liens peut suivre deux comportements différents selon si l'édition des liens demandée est une *édition des liens statique* ou une *édition des liens dynamique*. Ces deux types d'éditions des liens définissent le comportement de l'éditeur de liens vis-à-vis des bibliothèques du langage.

Une *bibliothèque* est une collection de modules objets prédéfinis et fournie avec la chaîne de production de programme relative au langage. Elle permet au programmeur de faire appel à ces fonctions sans avoir évidemment à les écrire. Les bibliothèques principales sont notamment :

– la bibliothèque mathématique qui fournit des fonctions mathématiques standard telles que la fonction racine carrée, la fonction puissance ou encore les fonctions trigonométriques…
– la bibliothèque graphique qui fournit des fonctions d'interface graphique permettant de dessiner des traits, des ronds, de se positionner sur l'écran…
– la bibliothèque d'interface avec le système d'exploitation qui contient les procédures permettant l'appel aux services du système d'exploitation, telles que `ouvrir_fichier`, `fermer_fichier`, `lire_données`, etc.

Édition des liens statique

Lors d'une édition des liens statique, l'éditeur de liens va rechercher les objets encore manquants à l'issue de la prise en compte des modules utilisateurs dans les *bibliothèques du langage*. Les bibliothèques à prendre en compte par l'éditeur de liens lui sont indiquées en même temps que la liste des modules objets utilisateur entrant dans la composition du programme exécutable final.

Exemple

La commande `ld – l nom_bib – o fich_exe module1.o module2.o` permet la construction du programme exécutable `fich_exe` à partir des modules objets utilisateurs `module1.o module2.o` ainsi que de la bibliothèque `nom_bib` sous un système de type Unix.

Les objets trouvés sont extraits des bibliothèques et placés dans la carte d'implantation du programme en cours de construction à la suite des modules utilisateurs. La table des liens est complétée. À l'issue de la prise en compte des bibliothèques, aucune entrée de la table des liens ne doit rester avec une entrée pour laquelle le champ adresse est indéfini. En effet, un tel cas de figure traduit le fait qu'un objet n'a pas été trouvé et donc la construction du programme final n'est pas possible.

Reprenons à présent notre exemple. L'éditeur de liens va rechercher à présent les 4 objets non résolus dans la bibliothèque spécifiée dans la commande d'édition des liens. Appelons `la_bibliothèque`, cette bibliothèque ; elle a le format suivant :

```
module la_bibliothèque;
taille (module) = 2 048 Ko
LU <écrire_fichier, adr_m_bibliotheque_1>
LU <lire_fichier, adr_m_bibliotheque_2>
LU <affiche_écran, adr_m_bibliotheque_3>
LU <lit_clavier, adr_m_bibliotheque_4>
Et d'autres LU correspondant aux autres fonctions présentes
➥ dans la bibliothèque
code objet translatable des fonctions incluses dans la bibliothèque
➥ la taille en octets des fonctions est connue :

taille(écrire_fichier) = 30 Ko
taille (lire_fichier) = 15 Ko
taille(affiche_écran) = 20 Ko
taille (lit_clavier) = 5 Ko
fin module;
```

À l'issue de la prise en compte de la bibliothèque `la_bibliothèque`, la carte d'implantation du programme est devenue :

– le module `interface.o` a une adresse d'implantation égale à 0;
– le module `recherche.o` a une adresse d'implantation égale à 512 Ko;
– le module `affichage.o` a une adresse d'implantation égale à 652 Ko;
– le module `lit_clavier.o` a une adresse d'implantation égale à 780 Ko;
– le module `affiche_ecran.o` a une adresse d'implantation égale à 785 Ko;
– le module `lire_fichier.o` a une adresse d'implantation égale à 805 Ko;
– le module `écrire_fichier.o` a une adresse d'implantation égale à 820 Ko;
– la taille totale du programme exécutable final est 850 Ko.

La table des liens devient (tableau 3.5) :

Tableau 3.5 TABLE DES LIENS APRÈS LA PRISE EN COMPTE DE LA BIBLIOTHÈQUE `LA_BIBLIOTHÈQUE`.

Nom de l'objet	Adresse dans la carte d'implantation	Commentaire
chercher_livre	adr_m_recherche_1 + 512 Ko	Résolution LAS/LU
afficher	adr_m_affichage_1 + 652 Ko	Résolution LAS/LU
réserver_livre	adr_m_recherche_2 + 512 Ko	Résolution LAS/LU
lit_clavier	780 Ko	Résolution LAS/LU
affiche_écran	785 Ko	Résolution LAS/LU
lire_fichier	805 Ko	Résolution LAS/LU
écrire_fichier	820 Ko	Résolution LAS/LU

On constate ici que toutes les entrées de la table des liens sont résolues.

Édition des liens dynamique

Lors d'une édition des liens dynamique, l'éditeur de liens ne recherche pas dans les bibliothèques du langage les objets qui n'ont pas été trouvés dans les modules utilisateurs. La table des liens reste donc avec des entrées pour lesquelles le champ adresse est à une valeur indéfinie.

La résolution des liens vis-à-vis des modules objet appartenant à des bibliothèques est repoussée à une étape ultérieure, soit au moment du chargement du programme en mémoire centrale, soit au moment de son exécution et de l'appel à l'objet non résolu.

➤ Construction du programme exécutable final

La dernière étape est la construction du programme exécutable final proprement dite. Dans un premier temps toutes les adresses dans les modules sont translatées de la valeur de l'adresse d'implantation du module. Dans un second temps, les références aux liens à satisfaire dans les modules sont remplacées par l'adresse de l'objet telle qu'elle est définie dans la table des liens (figure 3.8). Le programme exécutable obtenu est stocké sur un support de masse, tel qu'un disque dur.

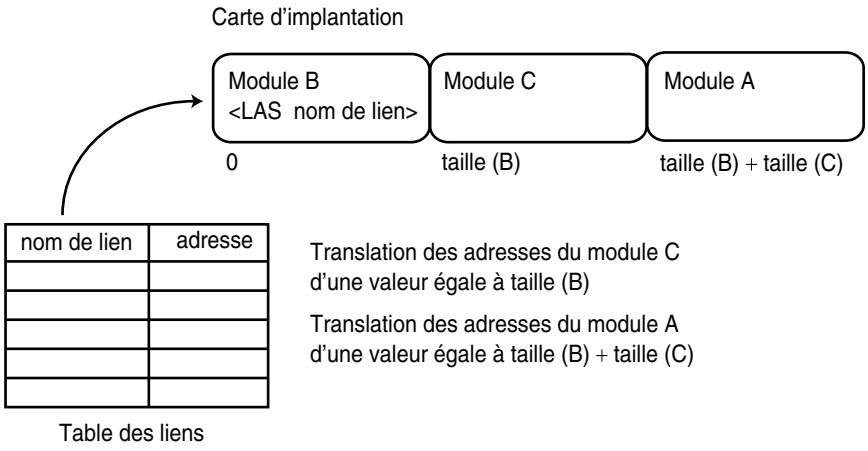


Figure 3.8 Construction du programme exécutable final.

3.3 LE CHARGEMENT

3.3.1 Rôle du chargeur

Le *chargement* constitue la dernière étape de la chaîne de production de programmes et met en œuvre l'outil *chargeur.* Le chargeur est appelé lorsque l'utilisateur souhaite exécuter son programme. Le chargeur copie alors le programme exécutable depuis le disque vers la mémoire centrale.

Le fichier exécutable stocké sur le disque par l'éditeur de liens est qualifié de fichier relogeable : toutes les adresses des instructions et des données dans ce code

exécutable sont calculées à partir de 0, c'est-à-dire que le premier octet du code exécutable à une adresse égale à 0. Lorsque le chargeur copie le code exécutable depuis le disque vers la mémoire centrale, il implante le code dans un espace libre de la mémoire centrale, dont le premier octet n'a pas forcément l'adresse 0 (et généralement jamais car la mémoire haute est réservée au système), mais une adresse quelconque appelée *adresse d'implantation mémoire*. Toutes les adresses `adr` calculées dans le programme exécutable stocké sur le disque doivent donc être modifiées pour tenir compte de cette adresse d'implantation mémoire : c'est l'opération de *translation des adresses* qui consiste à ajouter à chaque adresse `adr` apparaissant dans le programme exécutable du disque, la valeur de l'adresse d'implantation mémoire (figure 3.9).

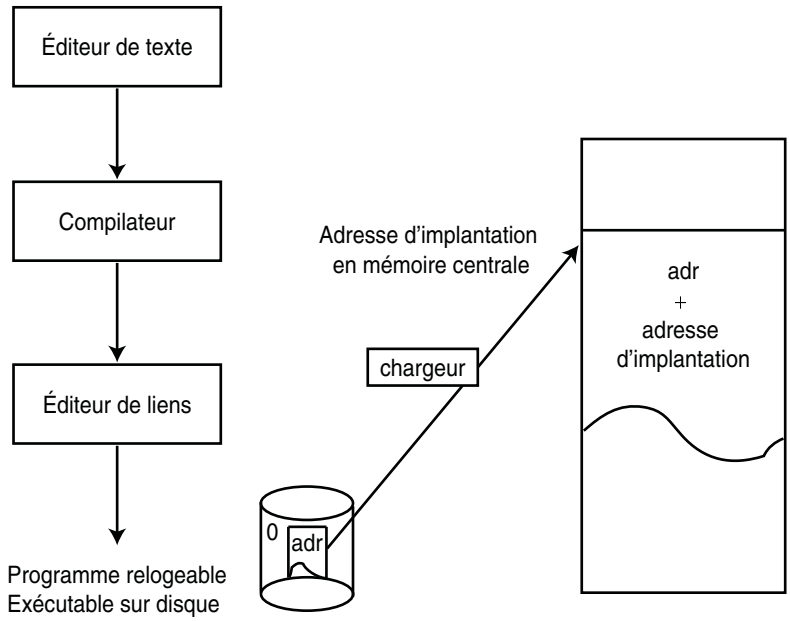


Figure 3.9 Chargement du programme exécutable relogeable.

Deux types de chargement peuvent être distingués :

– *le chargement statique* : dans ce cas, l'opération de translation des adresses `adr` est effectuée une fois pour toutes au moment du chargement et les adresses sont modifiées dans le programme exécutable ;
– *le chargement dynamique* : dans ce cas, l'opération de translation n'est pas effectuée au moment du chargement, mais seulement au moment où le processeur utilise une adresse relogeable au cours de l'exécution. La valeur de l'adresse d'implantation du programme en mémoire centrale est conservée dans un registre du processeur appelé le registre de translation et elle est ajoutée à l'adresse `adr` seulement au moment où le processeur utilise `adr`.

3.3.2 Chargement et édition des liens dynamique

Nous avons vu précédemment que lors d'une édition des liens dynamique, l'éditeur de liens ne recherche pas dans les bibliothèques du langage les objets pour lesquels des liens n'ont pas été résolus une fois les modules objets utilisateurs entrant dans la composition du programme exécutable pris en compte. Le programme exécutable construit alors est incomplet puisque des objets n'ont pas d'adresse. En effet, dans le cas d'une édition des liens dynamique, la prise en compte des bibliothèques est repoussée à l'étape de chargement. Dans ce cas les bibliothèques du langage sont chargées en mémoire centrale. Lors du chargement du programme exécutable de l'utilisateur, le chargeur résout les liens restants en fonction des adresses en mémoire centrale des objets appartenant aux bibliothèques.

Ce mode opératoire présente un avantage et un inconvénient vis-à-vis de l'édition des liens statique. Lors d'une édition des liens statique, l'éditeur des liens extrait des bibliothèques les objets utilisés par les modules utilisateurs et les ajoute à la carte d'implantation du programme exécutable en cours de construction. Imaginons que l'éditeur de liens ait construit deux programmes utilisateurs faisant référence à la même fonction mathématique `SQRT`[1]. Si ces deux programmes utilisateurs sont chargés en même temps en mémoire centrale, alors le code de la fonction `SQRT` se trouvera également deux fois en mémoire centrale, une fois dans chacun des programmes utilisateurs. Cette duplication engendre ici une perte de place inutile en mémoire centrale. L'édition des liens dynamique résout ce problème puisque la fonction `SQRT` se trouvera une seule fois en mémoire centrale, au sein de la bibliothèque (figure 3.10).

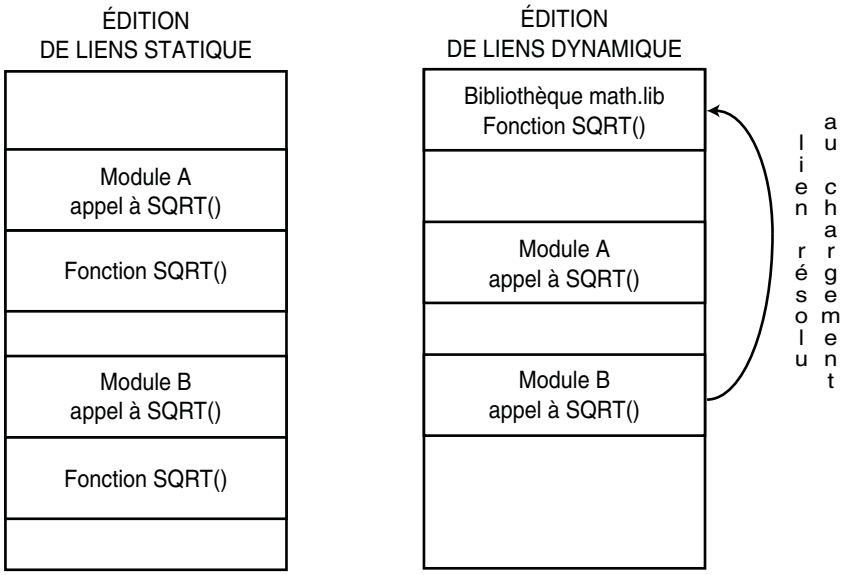


Figure 3.10 Chargement du programme exécutable relogeable.

1. Fonction racine carrée.

A contrario, résoudre les liens vers les objets des bibliothèques au moment du chargement ralentit cette phase de la chaîne de production de programmes. On notera par ailleurs que la résolution des liens vers les bibliothèques peut même être retardée jusqu'à l'exécution du programme et jusqu'au moment de l'utilisation de l'objet non résolu. Dans ce cas, c'est l'exécution même du programme qui est ralentie.

3.4 L'UTILITAIRE MAKE

Le Make est un outil qui exploite les dépendances existantes entre les modules entrant en jeu dans la construction d'un programme exécutable pour ne lancer que les opérations de compilations et éditions de liens nécessaires, lorsque ce programme exécutable doit être reconstruit suite à une modification intervenue dans les modules sources.

Exemple

Dans le programme suivant `x.c`, l'inclusion du fichier `defs` par l'ordre `#include "defs"` crée une dépendance entre ces deux modules.

```
/* fichier x.c */
#include "defs"
main()
{
    …
}
```

Cet outil utilise deux sources d'informations : un fichier de description appelé le Makefile qui contient la description des dépendances entre les modules et les noms et les dates de dernières modifications des modules.

3.4.1 Format du fichier Makefile

Le fichier Makefile décrit les dépendances existantes entre les modules intervenant dans la construction d'un exécutable : il traduit sous forme de règles le graphe de dépendance du programme exécutable à construire et indique pour chacune de ces dépendances, l'action qui lui est associée.

Une règle dans le fichier Makefile est de la forme :

```
module cible : dépendances
               commande pour construire le module cible
```

Prenons comme exemple le cas suivant : le programme exécutable `prog` est construit à partir d'une étape d'édition des liens prenant en compte les trois modules objets `x.o`, `y.o` et `z.o`. Le module objet `z.o` est issu de la compilation d'un programme source `z.c`. Les modules `x.o` et `y.o` sont à leur tour issus de la compilation respective des modules source `x.c` et `y.c`. Ces deux derniers modules utilisent un module `defs`

par le biais d'un ordre d'inclusion `#include`. Le graphe de dépendance du programme `prog` est donné sur la figure 3.11.

Le fichier Makefile résultant est :

```
prog : x.o y.o z.o
ld x.o y.o z.o – o prog
x.o : defs x.c
cc – o x.c
y.o : defs y.c
cc – c y.c
z.o : z.c
cc – c z.c
```

La première règle stipule la dépendance associée au programme exécutable `prog` qui est donc construit à partir des modules `x.o`, `y.o` et `z.o`. La commande permettant la construction du programme exécutable `prog` à partir de ces trois modules objets est la commande d'édition des liens `ld x.o y.o z.o – o prog`.

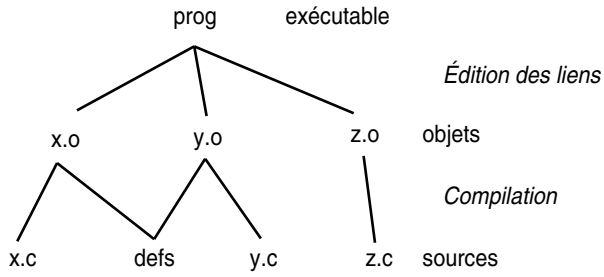


Figure 3.11 Graphe de dépendance du fichier `prog`.

La seconde et la troisième règle stipulent que le fichier objet `x.o` (respectivement `y.o`) dépend à la fois du fichier `x.c` (respectivement `y.c`) et du fichier `defs`. Les fichiers `y.o` ou `x.o` sont construits par compilation des fichiers.c correspondants.

Enfin, la dernière règle spécifie que le fichier `z.o` dépend uniquement de la compilation du fichier `z.c`.

3.4.2 Fonctionnement de l'utilitaire Make

L'outil Make utilise le fichier Makefile et les dates de dernières modifications des modules pour déterminer si un module est à jour. Un module est à jour si : le module existe et si sa date de dernière modification est plus récente que les dates de dernière modification de tous les modules dont il dépend ou bien elle est égale.

Si un module n'est pas à jour, la commande associée à ses dépendances est exécutée pour reconstruire le module.

Considérons par exemple que le module `z.c` soit modifié. L'utilitaire Make va détecter que ce module est devenu plus récent que le module objet `z.o`. Il va donc

lancer la commande associée à la règle de dépendance du module `z.o`, soit la commande de compilation `cc – c z.c`. L'exécution de cette commande va à son tour générer un module `z.o` plus récent que le programme exécutable `prog`. En conséquence, l'utilitaire Make va reconstruire le programme exécutable `prog` en lançant la commande d'édition des liens `ld x.o y.o z.o – o prog.` D'une façon similaire, toute modification au sein du module `defs` entraînera la reconstruction des modules `y.o` et `x.o`, par le biais de deux opérations de compilation, puis la reconstruction du programme exécutable `prog`. Dans ces deux cas, seules les opérations de compilation ou d'édition des liens nécessaires sont exécutées.

L'utilitaire Make est appelé au moyen de la commande `make prog` qui suppose qu'un fichier Makefile est présent dans le répertoire où la commande est lancée. La commande `make – f nom_fichier prog` fait également appel à l'utilitaire Make mais avec un fichier de dépendance appelé `nom_fichier` et non Makefile.

3.5 CONCLUSION

Ce chapitre nous a permis d'étudier la chaîne de production de programmes composée des étapes de compilation, édition des liens et chargement.

Un compilateur est un logiciel qui traduit un programme source écrit en langage de haut niveau en un programme objet en langage de bas niveau. L'analyse lexicale reconnaît dans la suite de caractères qui constitue le programme les symboles du langage. L'analyse syntaxique vérifie que la syntaxe du programme est conforme aux règles du langage. L'analyse sémantique trouve le sens et la signification des différentes phrases du langage. Enfin, la génération de code final consiste à générer un code machine relogeable. Ce code final est généré à partir d'un code intermédiaire optimisé.

Un éditeur de liens est un logiciel qui permet de combiner plusieurs modules objet obtenus par compilation séparée pour construire un seul programme exécutable. Un lien utilisable correspond à un objet exporté par un module. Un lien à satisfaire correspond à un objet importé par un module. Le rôle de l'éditeur de liens est de mettre en correspondance chaque lien à satisfaire avec un lien utilisable. L'édition des liens s'effectue en trois étapes qui sont respectivement la construction de la carte d'implantation, la construction de la table de liens et enfin la construction du programme exécutable final.

Le chargeur est un logiciel qui installe un programme exécutable en mémoire centrale en translatant toutes les adresses de celui-ci de la valeur de l'adresse d'implantation du programme. Éventuellement, il achève la résolution des liens.

Chapitre 4

Le langage machine et la représentation des informations

La donnée de base manipulée par la machine physique est le bit (*Binary Digit*) qui ne peut prendre que deux valeurs : 0 et 1. Au niveau physique, toutes les informations (nombres, caractères et instructions) ne peuvent donc être représentées que par une combinaison de 0 et 1, c'est-à-dire sous forme d'une chaîne binaire. Nous allons étudier les principales conventions de représentations des informations concernant les nombres signés, les nombres flottants et les caractères. Puis nous nous intéresserons plus particulièrement au format des instructions composant le langage machine, et sa forme symbolique, le langage d'assemblage.

4.1 LA REPRÉSENTATION DES INFORMATIONS

Les circuits physiques de la machine sont conçus à partir de transistors rassemblés sur une puce de silicium. Un transistor fonctionne selon une logique à deux états : soit le transistor est passant, auquel cas ce circuit délivre une tension comprise entre 2 et 5 volts, soit le transistor est bloquant auquel cas le circuit délivre une tension comprise entre 0 et 1 volt. Ces deux états logiques, conventionnellement notés 1 et 0, correspondent aux deux seules valeurs élémentaires disponibles pour représenter l'information au niveau physique.

Le codage de l'information se fait donc dans une base de représentation qui est la base 2. Toute information est représentée comme étant équivalente à une chaîne binaire

d'une longueur de n bits, *le bit ou Binary Digit*, étant le plus petit élément manipulable par la machine ne pouvant prendre pour valeur que 0 ou 1. Une chaîne de 4 bits est appelée un *quartet* tandis qu'une chaîne de 8 bits est appelée un *octet*. Le bit le plus à droite de la chaîne binaire est qualifié de *bit de poids faible* tandis que le bit le plus à gauche de la chaîne binaire est qualifié quant à lui de *bit de poids fort*.

Il existe plusieurs normes ou conventions pour la représentation des informations au niveau physique :

- la représentation des nombres entiers signés peut se faire selon la norme de la *valeur signée* ou selon la norme du *complément à 2*. Un autre format de représentation existant est également le *codage DCB (Décimal Codé Binaire)*;
- la représentation des nombres flottants admet de nombreuses variantes. Nous présentons uniquement la convention normalisée par l'organisme IEEE connue sous le nom de *forme IEEE 754*;
- la représentation des caractères admet elle aussi de nombreuses variantes que sont les codes *ASCII*, *EBCDIC* et *UNICODE*.

4.1.1 Numération binaire, octale et hexadécimale

Nous commençons par faire quelques rappels concernant la représentation des nombres dans une base X. Dans le cadre de la représentation des informations au niveau de la machine physique, la base utilisée est, comme nous l'avons déjà évoqué, la base 2 (système binaire). Cependant, comme les chaînes binaires ne sont pas aisément manipulables par l'esprit humain, deux autres bases sont très souvent utilisées : la base 8 (système octal) et la base 16 (système hexadécimal).

Représentation d'un nombre N en base X

Soit le nombre $N_X = d_n \ldots d_i \ldots d_2 d_1 d_0, d_{-1} \ldots d_{-i} \ldots d_{-m}$, exprimé dans la base X, avec d_n le digit de poids fort et d_{-m} le digit de poids faible, alors N s'écrit :

$$N_X = \sum_i d_i X_i \text{ avec } n \leq i \leq m$$

Ainsi $1975{,}57_{10} = 1 \times 10^3 + 9 \times 10^2 + 7 \times 10^1 + 5 \times 10^0 + 5 \times 10^{-1} + 7 \times 10^{-2}$.

Dans le reste de ce chapitre, nous allons manipuler des nombres exprimés dans la base 2, la base 10, la base 8 ou la base 16. Pour éviter toute confusion d'interprétation, nous prendrons l'habitude de préciser à côté du nombre, en indice, la valeur X de sa base d'expression tel que cela est fait ici pour le nombre $1975{,}57_{10}$ exprimé selon la

Chiffres autorisés selon la base X de représentation

Base 2 : chiffre 0 et 1.

Base 8 : chiffre 0, 1, 2, 3, 4, 5, 6, 7.

Base 16 : chiffre 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
avec $A = 10_{10}$, $B = 11_{10}$, $C = 12_{10}$, $D = 13_{10}$, $E = 14_{10}$, $F = 15_{10}$.

base 10. En effet, ce même nombre, de par les chiffres qui le composent, pourrait tout à fait être un nombre exprimé par exemple en base 16; sa valeur serait alors tout à fait différente et égale à $(1 \times 16^3 + 9 \times 16^2 + 7 \times 16^1 + 5 \times 16^0 + 5 \times 16^{-1} + 7 \times 16^{-2})$.

Conversion du nombre N exprimé en base 10 vers une base X (2, 8, 16)

➤ Conversion d'un nombre entier

Il existe deux méthodes pour convertir un nombre entier N exprimé en base 10 vers une base X, avec X = 2, 8 ou 16 : la première méthode est appelée *méthode des divisions successives* et la seconde méthode est appelée *méthode des soustractions successives*.

Méthode des divisions successives

N est itérativement divisé par X jusqu'à obtenir un quotient égal à 0. La conversion du nombre N dans la base X est obtenue en notant les restes de chacune des divisions effectuées depuis la dernière division jusqu'à la première.

Nota : les restes sont obligatoirement inférieurs à X.

Exemple

La figure 4.1 donne la conversion de 235_{10} en base 2 : $235_{10} = 11101011_2$.

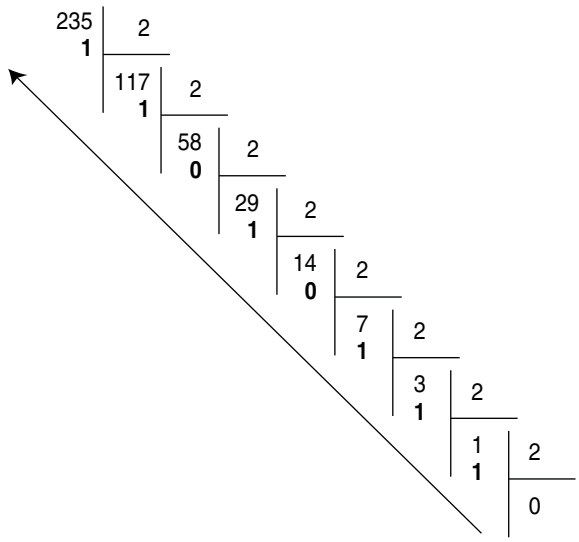


Figure 4.1 Méthode des divisions.

Méthode des soustractions

La plus grande puissance de X qui est inférieure ou égale à N est soustraite à N. Le processus de soustraction est répété sur le reste de la différence, jusqu'à obtenir un

résultat égal à 0. Le nombre N exprimé en base X est alors obtenu en notant le nombre de fois où une même puissance de X a été retirée et ce pour chaque puissance depuis la plus grande apparaissant, dans l'ordre décroissant des puissances.

Exemple

Convertir 235_{10} en base 8 :

On sait que $8^0 = 1$; $8^1 = 1$; $8^2 = 64$; $8^3 = 512$.

$235 - 64 = 171$; $171 - 64 = 107$; $107 - 64 = 43$; $43 - 8 = 35$; $35 - 8 = 27$; $27 - 8 = 19$; $19 - 8 = 11$; $11 - 8 = 3$; $3 - 1 = 2$; $2 - 1 = 1$; $1 - 1 = 0$

d'où $235_{10} = 3 \times 64 + 5 \times 8 + 3 \times 1 = 353_8$.

➤ Conversion d'un nombre fractionnaire

Lorsque le nombre N est fractionnaire, la conversion de sa partie entière vers une base X s'effectue avec l'une des deux méthodes que nous venons de voir. La conversion de la partie fractionnaire, par contre, s'effectue en multipliant cette partie fractionnaire par X. La multiplication est itérée sur la partie fractionnaire du résultat obtenu. La conversion de la partie fractionnaire du nombre N est obtenue par la suite des parties entières de chacun des résultats des multiplications effectuées.

Exemples

Convertir $0{,}45_{10}$ en base 2 :

$$0{,}45 \times 2 = 0{,}9 = 0 + 0{,}9$$
$$0{,}90 \times 2 = 1{,}8 = 1 + 0{,}8$$
$$0{,}8 \times 2 = 1{,}6 = 1 + 0{,}6$$
$$0{,}6 \times 2 = 1{,}2 = 1 + 0{,}2$$
$$0{,}2 \times 2 = 0{,}4 = 0 + 0{,}4$$
$$0{,}4 \times 2 = 0{,}8 = 0 + 0{,}8$$
$$0{,}8 \times 2 = 1{,}6 = 1 + 0{,}6$$
$$0{,}6 \times 2 = 1{,}2 = 1 + 0{,}2$$

d'où $0{,}45_{10} = 0{,}01110011_2$

Le développement s'arrête lorsque la précision voulue est obtenue.

Convertir $0{,}45_{10}$ en base 16 :

$$0{,}45 \times 16 = 7{,}20 = 7 + 0{,}2$$
$$0{,}20 \times 16 = 3{,}2 = 3 + 0{,}2$$

d'où $0{,}45_{10} = 0{,}73_{16}$

Conversion du nombre N exprimé en base X (2, 8, 16) vers la base 10

Le nombre $N_X = d_n \ldots d_i \ldots d_2 d_1 d_0, d_{-1} \ldots d_{-i} \ldots d_{-m}$, exprimé dans la base X, avec d_n le digit de poids fort et d_{-m} le digit de poids faible est converti vers la base 10 en appliquant la formule suivante :

$$N_X = d_n \times X^n + \ldots + d_i \times X^i + \ldots + d_2 \times X^2 + d_1 \times X^1 + d_0 \times X^0 + d_{-1} \times X^{-1} + \ldots + d_{-i} \times X^{-i} + \ldots + d_{-m} \times X^{-m} = M_{10}$$

Exemple

$$01010110_2 = 0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 0 \times 2^5 + 1 \times 2^6 + 0 \times 2^7$$
$$= 2 + 4 + 16 + 64 = 86_{10}$$

Conversion du nombre N exprimé dans la base 8 ou 16 vers la base 2 (et vice versa)

➤ Conversion de la base 8 vers la base 2 (et vice versa)

Les chiffres de la base 8 sont les chiffres allant de 0 à 7, soit un total de 8 chiffres. En binaire, un ensemble de n bits permet de représenter 2^n chiffres compris entre les valeurs 0 et 2^{n-1}. Aussi, pour représenter les 8 chiffres de la base 8 en binaire, il suffit d'un ensemble de 3 bits tels que : $000_2 = 0_8$; $001_2 = 1_8$; $010_2 = 2_8$; $011_2 = 3_8$; $100_2 = 4_8$; $101_2 = 5_8$; $110_2 = 6_8$; $111_2 = 7_8$.

Convertir un nombre N exprimé en base 8 vers la base 2 s'effectue en remplaçant simplement chacun des chiffres du nombre X en base 8 par leur équivalent binaire sur 3 bits.

Convertir un nombre N exprimé en base 2 vers la base 8 s'effectue en découpant la chaîne binaire N en paquet de 3 bits, depuis le bit de poids faible jusqu'au bit de poids fort pour la partie entière et inversement pour la partie fractionnaire, et en remplaçant chaque paquet de 3 bits par leur équivalent dans la base 8.

Exemple

$$452_8 = 100\ 101\ 010_2 \text{ et } 11\ 001\ 111{,}111\ 01_2 = 317{,}72_8$$

➤ Conversion de la base 16 vers la base 2 (et vice versa)

Le même principe s'applique strictement à la conversion entre la base 16 et la base 2. Les chiffres admis pour la base 16 sont maintenant les chiffres allant de 0 à 15, soit un total de 16 chiffres. 4 bits sont nécessaires pour représenter ces 16 valeurs.

Convertir un nombre N exprimé en base 16 vers la base 2 s'effectue en remplaçant simplement chacun des chiffres du nombre X en base 16 par leur équivalent binaire sur 4 bits.

Convertir un nombre N exprimé en base 2 vers la base 16 s'effectue en découpant la chaîne binaire N en paquet de 4 bits, depuis le bit de poids faible jusqu'au bit de poids fort pour la partie entière et inversement pour la partie fractionnaire, et en remplaçant chaque paquet de 4 bits par leur équivalent dans la base 16.

Exemple

$$45\ A_{16} = 0100\ 0101\ 1010_2 \text{ et } 1100\ 1111_2 = CF_{16}$$

4.1.2 Représentation des nombres signés

La représentation d'un nombre signé s'effectue selon une chaîne binaire d'une longueur fixée à n bits. On parlera ainsi d'une représentation des nombres signés sur 8 bits, 16 bits ou 32 bits.

Plusieurs conventions de représentation existent. Le choix entre l'une ou l'autre des conventions est effectué par le constructeur de la machine et éventuellement par le programmeur en fonction du type affecté aux variables déclarées. Par exemple, dans le langage C, une déclaration avec un type `int`, détermine une représentation sur 2 octets selon la convention du complément à 2. Une déclaration avec un type `unsigned short` détermine au contraire une représentation d'un nombre sur 8 bits, non signé.

Convention de la valeur signée

Dans la convention de la valeur signée, la chaîne de n bits $b_{n-1} \ldots b_0$ représentant le nombre signé est interprétée comme suit :

– le bit de poids fort b_{n-1} est le bit de signe. Si sa valeur est 0, alors le nombre codé est positif. Si sa valeur est 1, alors le nombre codé est négatif ;
– les autres bits $b_{n-2} \ldots b_0$ codent la valeur absolue du nombre.

Exemple

– Représentation de $+ 77_{10}$ sur 8 bits : 01001101_2.
– Représentation de $- 77_{10}$ sur 8 bits : 11001101_2.

Les remarques suivantes peuvent être faites à propos de cette représentation :

– deux représentations de la valeur 0 sont possibles, correspondant d'une part à un zéro positif ($00000000_2 = (+ 0)_{10}$ sur 8 bits), d'autre part à un zéro négatif ($10000000_2 = (- 0)_{10}$ sur 8 bits) ;
– l'intervalle des nombres signés représentables (tableau 4.1) est borné en fonction de la longueur de la chaîne binaire utilisée pour la représentation. Ainsi, sur 8 bits, l'intervalle des nombres représentables est $[11111111_2, 011111111_2]$, soit l'intervalle $[127_{10}, + 127_{10}]$. On notera donc ici que l'arithmétique des machines est différente de celle de l'être humain puisque l'intervalle des nombres n'est plus infini, mais dépend de la longueur des chaînes de bits manipulées par la machine ;
– la réalisation d'une opération de type soustraction nécessite un circuit particulier différent de celui permettant la réalisation des additions.

Tableau 4.1 INTERVALLES DES NOMBRES REPRÉSENTABLES EN VALEUR SIGNÉE.

Longueur de la chaîne de bits	Intervalle en base 10
8 bits	[– 127, + 127]
16 bits	[– 32 767, + 32 767]
32 bits	[– 2 147 483 647, + 2 147 483 647]
p bits	$[- 2^{p-1} - 1, + 2^{p-1} - 1]$

Convention du complément à 2

➤ Complément à 2 d'un nombre binaire N

Le *complément à 2* ou *complément vrai* d'un nombre binaire $N = b_{n-1} \ldots b_0$ s'obtient en ajoutant la valeur + 1 au complément restreint ou complément à 1 de ce nombre.

Le *complément à 1* ou *complément restreint* d'un nombre binaire $N = b_{n-1} \ldots b_0$ s'obtient en inversant la valeur de chacun des bits de ce nombre.

Exemple

$$\begin{array}{lr} & 10001001_2 \\ \text{complément restreint} & 01110110_2 \\ + & 1_2 \\ \text{complément vrai} & 01110111_2 \end{array}$$

➤ Convention du complément à 2

Dans la convention du complément à 2, un nombre négatif – N exprimé sur n bits est représenté en prenant le complément à 2 de son équivalent positif + N. Un nombre positif + N est quant à lui représenté par sa valeur binaire sur n bits.

Exemple

$$\begin{array}{lll} \text{Représentation de } +77_{10} \text{ sur 8 bits : } 01001101_2 & & \\ \text{Représentation de } -77_{10} \text{ sur 8 bits : } +77_{10} & & 01001101_2 \\ \quad \text{complément restreint} & & 10110010_2 \\ & + & 1_2 \\ -77_{10} \quad \text{complément vrai} & & 10110011_2 \end{array}$$

Le bit de poids fort b_{n-1} de la chaîne binaire $b_{n-1} \ldots b_0$ peut être également interprété comme bit de signe. Ainsi :

– si $b_{n-1} = 0$, alors la chaîne binaire $b_{n-1} \ldots b_0$ représente un nombre positif + N dont la valeur décimale est donnée directement par la conversion de la chaîne depuis la base 2 vers la base 10 ;

– si $b_{n-1} = 1$, alors la chaîne binaire $b_{n-1} \ldots b_0$ représente un nombre négatif – N dont la valeur décimale est celle du nombre positif associé + N obtenu en complémentant à 2 la chaîne $b_{n-1} \ldots b_0$.

Exemple

La chaîne 00110011_2 code un nombre positif qui a la valeur :

$$+ (2^0 + 2^1 + 2^4 + 2^5)_{10} = +51_{10}$$

La chaîne 10110011_2 code un nombre négatif dont la valeur est obtenue en prenant son complément à 2, soit :

$$\begin{array}{lr} & 10110011_2 \\ & 01001100_2 \\ + & 1_2 \\ & 01001101_2 = + (2^0 + 2^2 + 2^3 + 2^6)_{10} = +77_{10} \end{array}$$

d'où $10110011_2 = -77_{10}$.

Les remarques suivantes peuvent être faites à propos de cette représentation :

– une seule représentation du zéro est admise : $00000000_2 = (+\,0)_{10}$ sur 8 bits.
– l'intervalle des nombres signés représentables (tableau 4.2) est borné en fonction de la longueur de la chaîne binaire utilisée pour la représentation. Ainsi, sur 8 bits, l'intervalle des nombres représentables est $[10000000_2, 011111111_2]$, soit l'intervalle $[-\,128_{10}, +\,127_{10}]$. La chaîne 10000000_2 complémentée à 2 donne de nouveau la chaîne 10000000_2. Par convention, elle représente la valeur $-\,128_{10}$;
– la réalisation d'une soustraction ne nécessite pas de circuit particulier. Soustraire un nombre A à un autre nombre B équivaut à additionner au nombre B le complément à 2 du nombre A.

Tableau 4.2 Intervalles des nombres représentables en complément à 2.

Longueur de la chaîne de bits	Intervalle en base 10
8 bits	[– 128, + 127]
16 bits	[– 32 768, + 32 767]
32 bits	[– 2 147 483 648, + 2 147 483 647]
p bits	$[-\,2^{2p-1}, +\,2^{p-1} - 1]$

Convention du codage DCB (Décimal Codé Binaire)

Chaque chiffre du nombre N_{10} est codé par son équivalent binaire. Comme les chiffres de la base 10 admettent 10 valeurs différentes, le codage doit se réaliser sur 4 bits. Ainsi : $0000_2 = 0_{10}$; $0001_2 = 1_{10}$; $0010_2 = 2_{10}$; $0011_2 = 3_{10}$; $0100_2 = 4_{10}$; $0101_2 = 5_{10}$; $0110_2 = 6_{10}$; $0111_2 = 7_{10}$; $1000_2 = 8_{10}$; $1001_2 = 9_{10}$.

Le codage du signe peut suivre différentes conventions. La plus courante consiste à coder le signe + par la valeur 1011_2 et le signe – par la valeur 1101_2.

Exemple

– Représentation de $+\,77_{10}$: $1011\ 0111\ 0111_2$.
– Représentation de $-\,77_{10}$: $1101\ 0111\ 0111_2$.

L'un des inconvénients de ce codage est qu'il ne se prête pas directement aux opérations arithmétiques : en effet, l'addition de deux valeurs dont la somme est comprise entre 10_{10} et 15_{10} donne un code binaire sans signification. Aussi, lorsque la somme de deux chiffres décimaux est supérieure à 9_{10}, la machine doit effectuer un *ajustement décimal*, consistant à ajouter la valeur $+\,6_{10}$ à la somme des deux chiffres.

Exemple

Réalisons l'opération $81_{10} + 22_{10}$:

$$\begin{array}{rr} & 1000\ 0001_2 \\ + & 0010\ 0010_2 \\ \hline & 1010\ 0011_2 \end{array}$$

Le deuxième quartet 1010_2 a une valeur supérieure à 9_{10} qui est sans signification ; la valeur 0110_2 lui est ajoutée.

$$\begin{array}{rr} & 1010\ 0011_2 \\ + & 0110_2 \\ \hline & 1\ 0000\ 0011_2 \end{array}$$

Ce qui équivaut effectivement au nombre 103_{10}.

Notion de carry et d'overflow

➤ Notion de carry

Lors d'une opération arithmétique effectuée sur des nombres de p bits, un p + 1er bit peut être généré. Ce bit supplémentaire de poids fort n'est pas perdu et est mémorisé comme étant le *bit de carry*. Cette mémorisation s'effectue dans un registre du processeur appelé *registre d'état* (*PSW*) qui comporte plusieurs indicateurs de 1 bit, dont l'un noté C, est justement positionné par l'occurrence d'un carry lors d'une opération arithmétique.

Exemple

Sur 8 bits, nous effectuons l'addition des nombres + 72_{10} et + 3_{10} représentés selon la convention du complément à 2 :

$$\begin{array}{rr} & 0100\ 1000_2 \\ + & 0000\ 0011_2 \\ \hline & 0100\ 1011_2 \end{array}$$

Il n'y a pas de carry.

Sur 8 bits, nous effectuons maintenant l'addition des nombres + 127_{10} et – 2_{10} représentés selon la convention du complément à 2 :

$$\begin{array}{rr} & 0111\ 1111_2 \\ + & 1111\ 1110_2 \\ \hline & 1\ 0111\ 1101_2 \end{array}$$

Le 9e bit qui apparaît est le bit de carry.

➤ Notion d'overflow

Lors d'une opération arithmétique mettant en jeu des nombres de p bits et de même signe, le résultat peut se révéler être trop grand ou trop petit pour être représentable par la machine, c'est-à-dire que ce résultat est en dehors de l'intervalle des nombres représentables sur p bits par la convention choisie pour la représentation de ces nombres signés. Le résultat obtenu est alors erroné au regard de son interprétation. On parle alors d'*overflow* ou de *dépassement de capacité*. À l'instar du carry, l'occurrence d'un overflow est mémorisée dans le *registre d'état* (*PSW*) du processeur par l'intermédiaire d'un indicateur de 1 bit noté O.

Exemple

Sur 8 bits, nous effectuons l'addition des nombres $+ 127_{10}$ et $+ 2_{10}$ représentés selon la convention du complément à 2 :

$$\begin{array}{rr} & 0111\ 1111_2 \\ + & 0000\ 0010_2 \\ & 1000\ 0001_2 \end{array}$$

Le résultat obtenu est un nombre négatif qui a pour valeur $- 127_{10}$ et non pas la valeur attendue $+ 129_{10}$. Il y a dépassement de capacité ; en effet, l'intervalle des nombres représentables sur 8 bits selon la convention du complément à 2 est $[- 127_{10}, + 127_{10}]$.

4.1.3 Représentation des nombres flottants

Principe général

Un nombre est représenté en virgule flottante dans la base X s'il est mis sous la forme : $\pm$ M1, M2 $\cdot X^{\pm c}$ où M1, M2 est appelé la *mantisse* du nombre, c est la *caractéristique* ou *exposant.*

Un nombre représenté en virgule flottante est normalisé s'il est sous la forme : $\pm$ 0,M $\cdot X^{\pm c}$ où M est un nombre dont le premier chiffre est non nul.

Exemple

$+ 59{,}4151 \times 10^{-5}$ est normalisé sous la forme $+ 0{,}594151 \times 10^{-3}$.

Il s'agit de représenter la mantisse et son signe, ainsi que l'exposant et son signe.

➤ Représentation de la mantisse et de son signe : $\pm$ 0,M

Seul le nombre M est représenté, soit selon la convention de la valeur signée, soit selon la convention du complément à 2, soit en base 2 non signée.

➤ Représentation de l'exposant et de son signe : $\pm$ c

La caractéristique c est translatée de manière à toujours coder en interne une valeur positive. Ainsi, seule la valeur de c a besoin d'être représentée.

Supposons que 5 bits soient réservés au codage de la caractéristique. Les valeurs positives allant de $+ 0_{10}$ à $+ 31_{10}$ sont représentables pour la caractéristique c, ce qui permet en appliquant une translation k égale à 16_{10} de représenter les exposants allant de $- 16_{10}$ à $+ 15_{10}$. La constante k est appelée *constante d'excentrement*.

➤ Un exemple : le format IBM

La représentation des nombres flottants pour les architectures IBM 370 (figure 4.2) admet trois formats : un format court sur 32 bits, un format long sur 64 bits et un format étendu sur 128 bits. Le nombre flottant est normalisé sous la forme $\pm 0{,}M_{16} \cdot 16^{\pm c}$. Chacun des trois formats adopte la codification suivante :

– le signe de la mantisse est codé sur le premier bit de l'octet de poids fort. Ce bit vaut 0 si la mantisse est positive, 1 si elle est négative ;
– les 7 bits restants de l'octet de poids fort codent la caractéristique c. C'est une puissance de 16 codée en interne avec un excentrement égal à 64_{10} ;
– les bits restants codent la mantisse, c'est-à-dire le nombre M exprimé en base 16.

L'intervalle des nombres pouvant être représentés dans ce format est l'intervalle $[16^{-64}, 16^{+63}]$.

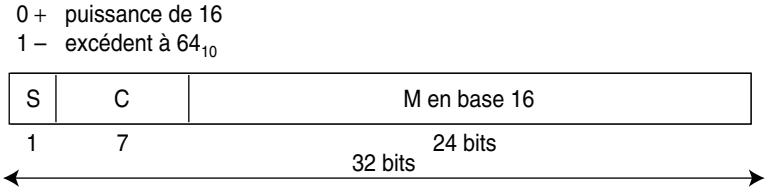


Figure 4.2 Le format IBM 32 bits.

Exemple

Représentons le nombre $-10{,}125_{10}$ selon le format court sur 32 bits.

$$10{,}125_{10} = 1010{,}001_2 = A{,}2_{16} = 0{,}A2 \times 16^1$$

L'exposant c = 1 est translaté de la valeur 64. c' = $65_{10} = 01000001_2$.

Le signe de la mantisse est négatif et vaut donc 1.

Le codage donne donc la chaîne binaire :

$$1\ 01000001\ 10100010000000000000000_2 = A0D10000_{16}$$

La norme IEEE 754

➤ Format normalisé d'un nombre

La norme IEEE 754 définit un format standardisé qui vise à unifier la représentation des nombres flottants, qui est très diverse selon les constructeurs.

Cette norme propose deux formats de représentation : un format simple précision sur 32 bits et un format double précision sur 64 bits (figure 4.3).

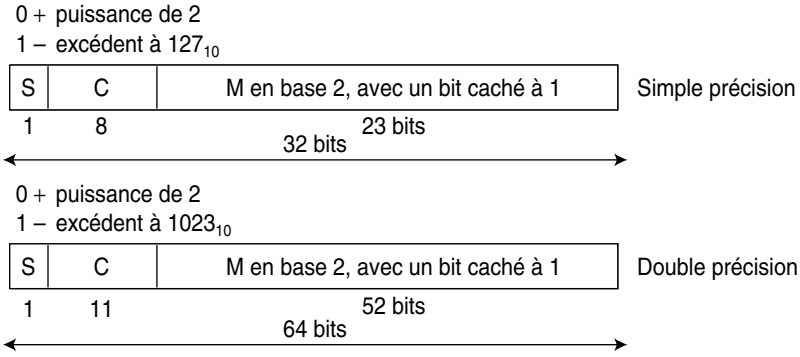


Figure 4.3 Les formats IEEE 754.

Dans le format simple précision, la chaîne de 32 bits représentant le nombre est décomposée en 1 bit pour le signe de la mantisse, 8 bits pour l'exposant et 23 bits pour le codage de la mantisse.

Dans le format double précision, la chaîne de 64 bits représentant le nombre est décomposée en 1 bit pour le signe de la mantisse, 11 bits pour l'exposant et 52 bits pour le codage de la mantisse.

La mantisse est normalisée sous la forme $\pm 1{,}M \cdot 2^{\pm c}$ où M est un nombre quelconque. On parle alors de *pseudo-mantisse*. Le 1 précédant la virgule n'est pas codé en machine et est appelé *bit caché*. Le signe de la mantisse est codé sur un bit (bit de poids fort de l'entier de 32 bits ou de 64 bits) valant 0 si la mantisse est positive, et 1 si elle est négative. La valeur du nombre M est codée selon la base 2.

Pour le format simple précision, la constante k d'excentrement appliquée à l'exposant est égale à $+ 127_{10}$. Elle est égale à $+ 1\,023_{10}$ pour le format double précision. L'exposant c´ codé en interne est alors égal à $\pm c + 127_{10}$ ou $\pm c + 1\,023_{10}$.

Exemple

Représentons le nombre $- 10{,}125_{10}$ selon le format IEEE 754 simple précision.

$$10{,}125_{10} = 1010{,}001_2 = 1{,}010001_2 \times 2^3$$

L'exposant c = 3 est translaté de la valeur 127.

$$c' = 130_{10} = 10000010_2$$

Le signe de la mantisse est négatif et vaut donc 1.

Le codage donne donc la chaîne binaire :

$$1\ 10000010\ 01000100000000000000000_2 = C1220000_{16}$$

➤ Représentation du zéro, des infinis, représentations dénormalisées

En dehors du format normalisé, le norme IEEE admet des codages spéciaux pour la représentation de la valeur 0, des valeurs $+ \infty$ et $- \infty$, ainsi que des représentations dénormalisées (tableau 4.3) :

- représentation du zéro : la valeur 0 est représentée avec un exposant $c = - 127_{10}$ et une mantisse M = 0. Le signe S est quelconque; il y a donc existence d'un zéro positif et d'un zéro négatif;
- représentation de $+ \infty$ et $- \infty$: les valeurs $+ \infty$ et $- \infty$ sont représentées avec un exposant $c = + 128_{10}$ et une mantisse M = 0. Le signe S est fonction de l'infini représenté;
- représentation dénormalisée : la représentation dénormalisée permet de représenter des nombres plus petits que ceux admis par le format IEEE normalisé. Dans ce cas, les nombres dénormalisés sont représentés avec un exposant $c = - 127_{10}$ et une mantisse sous la forme $\pm 0{,}M \cdot 2^{\pm c}$. Par ailleurs, la norme admet également la représentation de codes permettant la signalisation d'erreurs (par exemple, résultat d'une division par 0). Ce sont les NaNs (*Not a Number*) codés avec un exposant $c = + 128_{10}$ et une mantisse M non nulle.

Tableau 4.3 RÉSUMÉ DES FORMATS DE LA NORME IEEE 754.

Format	Valeurs représentées
Normalisé - – 126 ≤ c ≤ + 127	2^{-126} à 2^{128}
c = – 127, M = 0, S = + ou –	+ 0 ou – 0
c = + 128, M = 0, S = + ou –	+ ∞ ou – ∞
c = – 127, M ≠ 0, S = + ou –	dénormalisé
c = + 128, M ≠ 0, S = + ou –	NaNs

4.1.4 Représentation des caractères

La représentation des caractères s'effectue par l'association d'une chaîne binaire à chaque caractère. Selon la longueur de la chaîne binaire choisie par le code, le nombre de caractères pouvant être codés est plus ou moins important. Le nombre de caractères représentables par un code est appelé *puissance lexicographique du code*. Les caractères à représenter sont :

– d'une part les caractères éditables, c'est-à-dire les lettres majuscules et minuscules, les signes de ponctuation, les signes mathématiques, etc. ce qui représente environ 90 caractères ;

– d'autre part les caractères non éditables ou commandes qui sont réservés à la réalisation de fonctions particulières par l'ordinateur. Par exemple le caractère BEL provoque le retentissement d'une sonnerie au niveau du terminal.

Le code ASCII

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	'	P
1	SOH	DC1	!	1	A	Q	a	Q
2	STX	DC2	«	2	B	R	b	R
3	ETX	DC3	#	3	C	S	c	S
4	EOT	DC4	$	4	D	T	d	T
5	ENQ	NAK	%	5	E	U	e	U
6	ACK	SYN	&	6	F	V	f	V
7	BEL	ETB	'	7	G	W	g	W
8	BS	CAN	(	8	H	X	h	X
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Figure 4.4 Table ASCII standard.

Le codage ASCII (*American Standard Code for Information Interchange*) encore connu sous le nom d'alphabet international n°5 est un code à 7 bits qui permet donc de représenter 128 caractères. Chacun des codes associés à un caractère est donné dans une table à deux entrées, la première entrée codant la valeur du quartet de poids faible et la seconde entrée codant la valeur des 3 bits de poids fort du code associé au caractère (figure 4.4). Ainsi le caractère A est codé par la chaîne $100\ 0001_2$ soit le code hexadécimal 41_{16}.

Le code ASCII est très utilisé sur les processeurs de la famille Intel.

Le code EBCDIC

Le codage EBCDIC (*Extended Binary Coded Decimal Interchange Code*) est un code à 8 bits permettant donc de coder 256 caractères. Chacun des codes associés à un caractère est donné dans une table à deux entrées, la première entrée codant la valeur du quartet de poids faible et la seconde entrée codant la valeur du quartet de poids fort du code associé au caractère (figure 4.5). Ainsi le caractère A est codé par la chaîne $1100\ 0001_2$ soit le code hexadécimal $C1_{16}$.

Le code EBCDIC est très utilisé sur les gros systèmes, notamment les systèmes de la famille IBM tels que les architectures 370 et 390.

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX		PT			GE				FF	CR		
1	DLE	SBA	EUA	1C		NL				EM			DUP	SF	FM	ITB
2							ETB	ESC						ENQ		
3			SYN					EOT					RA	NAK		
4	SP										¢	.	<	(	+	
5	&										!	$	*	)	;	¬
6	-	/									¦	,	%	_	>	?
7											:	#	@	'	=	"
8		a	b	c	d	E	f	g	h	I						
9		j	k	l	m	N	o	p	q	r						
A		~	s	t	u	v	w	x	y	z						
B																
C	{	A	B	C	D	E	F	G	H	I						
D	}	J	K	L	M	N	O	P	Q	R						
E	\		S	T	U	V	W	X	Y	Z						
F	0	1	2	3	4	5	6	7	8	9						
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure 4.5 Table EBCDIC standard.

Le code UNICODE

Le codage UNICODE (*Universal Code*) est un code à 16 bits en cours de définition qui a pour but de coder le plus grand nombre possible de symboles en usage dans le monde. Les 16 bits de code permettent de coder 65 536 caractères différents. Le codage

UNICODE reprend le codage ASCII en ce qui concerne les principaux caractères, en étendant le code à 16 bits (figure 4.6). Ainsi, le caractère A est codé par la chaîne hexadécimale 0041_{16}. Ce code est utilisé notamment sous les processeurs de type Pentium.

Figure 4.6 Table UNICODE standard.

4.2 LES INSTRUCTIONS MACHINE

Une *instruction* (figure 4.7) désigne un ordre donné au processeur et qui permet à celui-ci de réaliser un traitement élémentaire. L'instruction machine est une chaîne binaire de p bits composée principalement de deux parties :

– le champ *code opération* composé de m bits : il indique au processeur le type de traitement à réaliser (addition, lecture d'une case mémoire, etc.). Un code opéra-

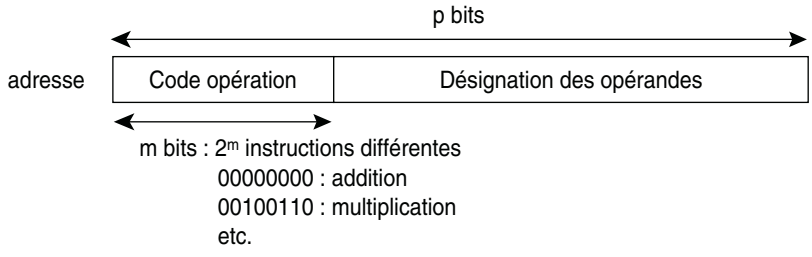


Figure 4.7 Format général d'une instruction machine.

tion de m bits permet de définir 2^m opérations différentes pour la machine. Le nombre d'opérations différentes autorisées pour une machine définit le *jeu d'instructions* de la machine ;

– le champ *opérandes* composé de p – m bits : il permet d'indiquer la nature des données sur lesquelles l'opération désignée par le code opération doit être effectuée. La façon de désigner un opérande dans une instruction peut prendre différentes formes : on parle alors de *mode d'adressage* des opérandes.

4.2.1 Les différents types d'instructions

Les instructions du langage machine peuvent être rangées selon six catégories :

– les instructions arithmétiques et logiques : ce sont les instructions qui permettent de réaliser les calculs entre nombres (addition, soustraction, multiplication…) et les opérations logiques (ou, et, ou exclusif…). Elles mettent en jeu les circuits de l'UAL. Ainsi l'instruction `ADD Im R1 3` effectue l'addition du contenu du registre R1 avec la valeur immédiate 3 et stocke le résultat dans le registre R1 ;

– les instructions de transfert de données : ce sont les instructions qui permettent de transférer une donnée depuis les registres du processeur vers la mémoire centrale et vice versa ainsi qu'entre registres du processeur. Ainsi l'instruction `LOAD D R1 3` range la valeur contenue à l'adresse 3 en mémoire centrale dans le registre R1 tandis que l'instruction `STORE D R1 3` écrit le contenu du registre R1 à l'adresse mémoire 3 ;

– les instructions d'entrées-sorties : ce sont les instructions qui permettent au processeur de lire une donnée depuis un périphérique (par exemple le clavier) ou d'écrire une donnée vers un périphérique (par exemple l'imprimante) ;

– les instructions de rupture de séquence d'exécution encore appelées instructions de saut ou de branchement : ce sont des instructions qui permettent de rompre l'exécution séquentielle des instructions d'un programme. L'instruction exécutée à la suite d'un saut n'est pas celle qui suit immédiatement l'instruction de saut, mais celle dont l'adresse a été spécifiée dans l'instruction de saut. On distingue ici deux types d'instructions de sauts. Les instructions de sauts inconditionnels effectuent toujours le débranchement de l'exécution à l'adresse spécifiée. Les instructions de sauts conditionnels effectuent ce débranchement si et seulement si une condition liée aux indicateurs du registre d'état de l'UAL est vérifiée. Ainsi l'instruction `JMP D 128` effectue toujours un branchement dans le code du programme à l'adresse 128 tandis que l'instruction `JMPO D 128` effectue ce même branchement si et seulement si un dépassement de capacité est positionné dans le registre d'état de l'UAL (indicateur O) ;

– les instructions d'appels de sous-programmes (`CALL`, `RET`) qui permettent de modifier la valeur courante du compteur ordinal CO pour aller exécuter une suite d'instructions machine constituant une fonction (c'est l'objet de l'instruction `CALL` qui effectue l'appel de sous-programme) puis de revenir exécuter les instructions machine situées juste après l'appel (instruction `RET`) ;

– les instructions particulières permettant par exemple d'arrêter le processeur (`HALT`) ou encore de masquer/démasquer les interruptions (`DI`/`EI`).

4.2.2 Les différents types d'opérandes

Les opérandes désignent les données sur lesquelles le code opération de l'instruction doit être réalisé. Selon le type d'instruction, le code opération peut admettre 1, 2 ou 3 opérandes.

Un opérande peut être de trois natures différentes :

– l'opérande est une valeur immédiate, par exemple la valeur 3 ;
– l'opérande est un registre du processeur, par exemple le registre R1 ;
– l'opérande est un mot mémoire, par exemple le mot mémoire d'adresse 63_{16}.

La nature de l'opérande et la façon de l'atteindre sont indiquées par l'intermédiaire du *mode d'adressage*. Le format du champ opérande est donc schématiquement composé de deux parties : le mode d'adressage lié à l'opérande et une information complémentaire qui permet conjointement avec le mode d'adressage de trouver l'opérande.

Exemple

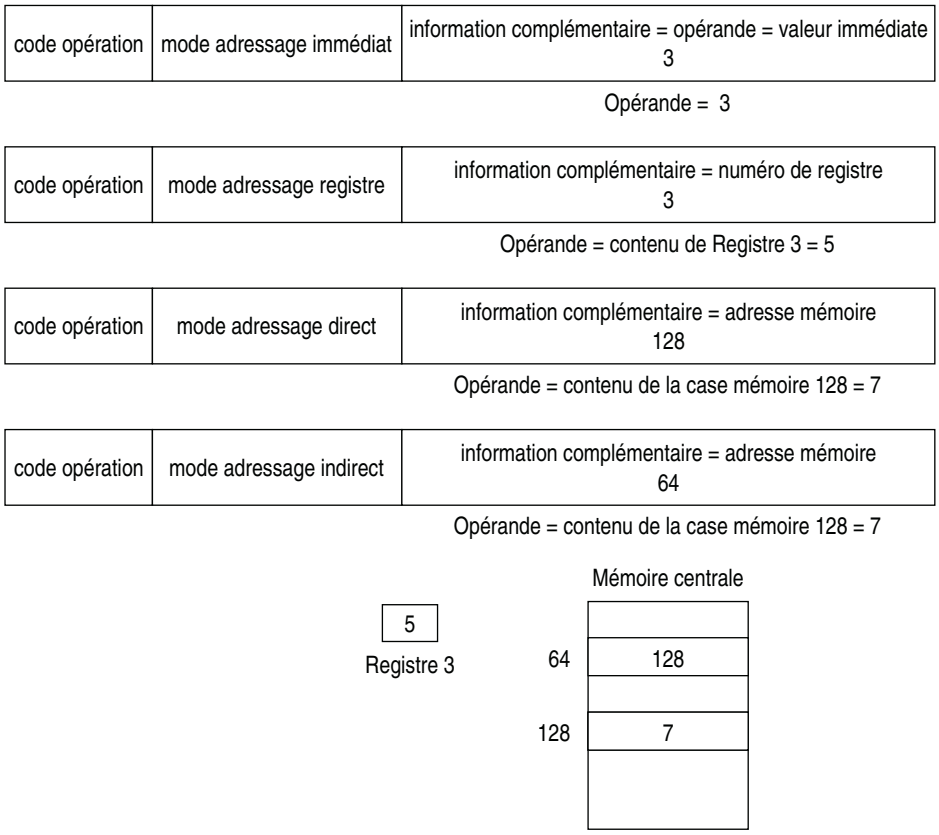


Figure 4.8 Modes d'adressages.

Lorsque l'opérande est une valeur immédiate, le mode d'adressage associé à l'opérande est le mode d'adressage *immédiat*. L'information complémentaire est l'opérande lui-même (figure 4.8).

Lorsque l'opérande est contenu dans un registre, alors le mode d'adressage associé est un mode d'adressage registre ou implicite. L'information complémentaire est le numéro du registre qui contient l'opérande.

Lorsque l'opérande est un mot mémoire, l'un des modes d'adressage associés peut-être le mode d'adressage *direct*. Dans ce cas, l'information complémentaire désigne l'adresse du mot mémoire contenant l'opérande concerné par l'opération. Un autre mode d'adressage associé peut être le mode d'adressage *indirect*. Dans ce cas, l'information complémentaire désigne également l'adresse d'un mot mémoire, mais ce mot mémoire contient lui-même une adresse qui est l'adresse du mot mémoire contenant l'opérande concerné par l'opération.

4.2.3 Un exemple

Imaginons une machine qui admet des instructions sur 32 bits de type registre/mémoire. Le format d'une instruction est donné par la figure 4.9.

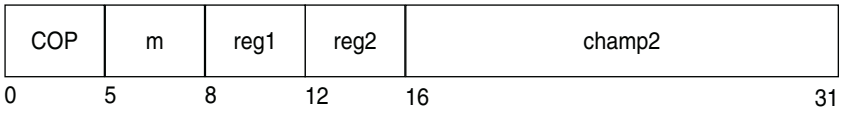


Figure 4.9 Format d'une instruction machine.

Ce format comprend :

- COP est le Code OPération codé sur 5 bits ;
- m est le mode d'adressage codé sur 3 bits ;
- reg1 et reg2 codent un numéro de registre sur 4 bits (de 0000_2 à 1111_2) ;
- champ2 est une valeur immédiate, une adresse mémoire ou un déplacement codé sur 16 bits.

reg1, reg2 et champ2 codent l'information complémentaire du mode d'adressage m permettant de trouver les opérandes de l'opération.

L'instruction admet soit :

- un seul opérande qui est alors un registre du processeur codé dans le champ reg1 ;
- deux opérandes, qui sont soit deux registres du processeur codés dans reg1 et reg2, soit un registre codé dans reg1 et une valeur immédiate codée dans champ2, soit un registre codé dans reg1 et un mot mémoire codé via le mode d'adressage m et la valeur précisée dans champ2.

Les codes opérations

Le code opération est une chaîne de 5 bits ce qui permet de coder 32 opérations différentes. Le tableau 4.4 donne quelques codes opérations :

Tableau 4.4 QUELQUES CODES OPÉRATIONS.

Nature de l'opération	Code binaire
chargement d'un registre	00000
chargement d'un mot mémoire	00001
…	…
addition	00101
complément à 2	00110
et logique	01000
ou logique	01001
débranchement dans le code	01100

Les modes d'adressage : le champ m

Le champ m code le mode d'adressage utilisé vis-à-vis des opérandes. m étant codé sur 3 bits, il autorise 8 modes d'adressage différents. Le tableau 4.5 donne l'exemple de quelques modes d'adressage.

Tableau 4.5 QUELQUES MODES D'ADRESSAGE.

Mode d'adressage	Signification pour l'opérande	Valeur
immédiat	champ2 = valeur immédiate	m = 000
direct	champ2 = adresse de l'opérande	m = 001
…	…	…
indirect	champ2 = adresse d'un mot mémoire qui contient l'adresse du mot mémoire contenant l'opérande	m = 011

Pour les valeurs de m comprises entre 0 et 5, le code opération travaille sur deux opérandes ; le premier est un registre dont le numéro est codé par le champ reg1 ; le deuxième est soit une valeur immédiate, soit une adresse déduite de champ2 et m.

Les valeurs m = 110 et m = 111 sont utilisées pour les opérations sur des registres :

– 110 : la valeur champ2 ainsi que celle du champ reg2 ne sont pas significatives ; le code opération travaille sur un seul opérande, le registre reg1 ;
– 111 : le code opération travaille sur deux registres reg1 et reg2. La valeur champ2 n'est pas significative.

Les valeurs des champs reg

Les valeurs reg(1ou2) = 0000 à 1111 codent les numéros de registres du processeur.

Exemples d'instructions machines

Prenons deux exemples d'instructions machine dans le contexte que nous venons de définir :

```
 COP    m    reg1  reg2       champ2
00101  000  0000  xxxx  00000000000001000
```

Le code opération correspond à une addition avec deux opérandes. Le premier opérande est un registre codé dans le champ reg1 qui est donc le registre R0 du processeur. Le champ reg2 est sans signification (xxxx). Le deuxième opérande est désigné par un mode d'adressage m = 000. C'est un mode d'adressage immédiat, ce qui signifie que le deuxième opérande est la valeur 8 codée dans champ2. In fine, cette instruction effectue l'addition entre le contenu du registre R0 et la valeur 8 et range le résultat de l'opération dans R0.

```
 COP    m    reg1  reg2       champ2
00000  001  0000  xxxx  00000000000100000
```

Le code opération correspond à un chargement de registre. Le premier opérande est un registre codé dans le champ reg1 qui est donc le registre R0 du processeur. Le champ reg2 est sans signification (xxxx). Le deuxième opérande est désigné par un mode d'adressage m = 001. C'est un mode d'adressage direct, ce qui signifie que le deuxième opérande est le contenu du mot mémoire pour lequel l'adresse est codée dans champ2. In fine, cette instruction effectue le chargement du registre R0 avec le contenu de la case mémoire d'adresse 32.

Pour terminer, le programme relogeable suivant réalise l'incrémentation infinie du contenu du registre R1 en écrivant à chaque tour de boucle le contenu du registre R1 à l'adresse mémoire 0.

```
adresse        COP    m   reg1 reg2       champ2          signification
(00000000)16 cet emplacement est réservé pour stocker le contenu de R1
(00000004)16   00000 000  0001 0000  0000000000000000     R1 ← 0
(00000008)16   00101 000  0001 xxxx  0000000000000001     R1 ← R1 + 1
(0000000C)16   00001 001  0001 xxxx  0000000000000000     (0)16 ← R1
(00000010)16   01100 001  xxxx xxxx  0000000000001000
                     retourner à l'instruction d'adresse 816
```

4.3 LES INSTRUCTIONS DU LANGAGE D'ASSEMBLAGE

Le langage machine se compose d'instructions exprimées en binaire, telles qu'on les trouve dans la mémoire au moment de l'exécution du programme.

Le langage d'assemblage est une variante symbolique du langage machine, permettant au programmeur de manipuler les instructions de la machine en s'affranchissant notamment des codes binaires et des calculs d'adresse. Le langage d'assemblage comporte le même jeu d'instructions que le langage machine et est également spécifique de la machine.

Pour pouvoir être exécuté par la machine, le programme écrit en langage d'assemblage doit être traduit en langage machine. Cette traduction est effectuée par un outil appelé l'assembleur.

4.3.1 Format d'une instruction du langage d'assemblage

Une instruction du langage d'assemblage est composée de champs, séparés par un ou plusieurs espaces. On identifie un champ étiquette, un champ code opération, un champ opérandes pouvant effectivement comporter plusieurs opérandes séparés par des virgules et un champ commentaires (figure 4.10).

Nous allons étudier plus en détail le format des instructions du langage d'assemblage et la composition du langage d'assemblage en considérant que ce langage d'assemblage est celui associé au langage machine de l'exemple précédent.

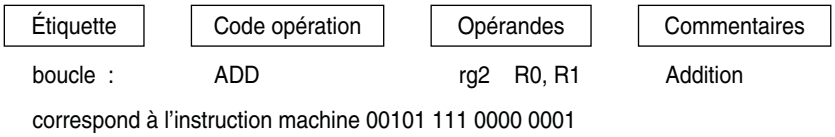


Figure 4.10 Format d'une instruction en langage d'assemblage.

Codes opérations

Le *code opération* est une chaîne de caractères mnémonique du code opération binaire. Le tableau 4.6 donne l'exemple de quelques mnémoniques.

Tableau 4.6 QUELQUES CODES OPÉRATIONS MNÉMONIQUES.

Nature de l'opération	Code assembleur mnémonique	Code binaire
chargement d'un registre	LOAD	00000
chargement d'un mot mémoire	STORE	00001
…		…
addition	ADD	00101
complément à 2	CP2	00110
et logique	ET	01000
ou logique	OU	01001
débranchement dans le code	JMP	01100

Les étiquettes

Une *étiquette* est une chaîne de caractères permettant de nommer une instruction ou une variable. Une étiquette correspond à une adresse dans le programme, soit celle de l'instruction, soit celle de la variable.

Exemple

L'étiquette `boucle` remplace l'adresse binaire de l'instruction `ADD Rg2 R1, R0` :

```
boucle : ADD Rg2 R1, R0
         ADD Rg2 R1, R3
         JMP Im boucle
```

Une étiquette peut également permettre de nommer une constante.

Les opérandes

Chaque opérande dans une instruction du langage d'assemblage possède un nom permettant de le référencer. Les opérandes d'une instruction sont séparés par une virgule.

Pour les opérandes variables ou constantes, le nom est l'étiquette associée par le programmeur au moment de la déclaration des variables et constantes (cf. paragraphe 4.3.1).

Pour les opérandes adresse d'instruction, le nom est l'étiquette associée à l'instruction par le programmeur. Chaque registre de la machine est référencé par un nom : R0, R1, R2, …, R11.

Le mode d'adressage d'un opérande mémoire est spécifié par une chaîne placée après le code opération et qui remplace la chaîne binaire m. Le tableau 4.7 donne quelques exemples de mnémoniques associés aux modes d'adressage.

Tableau 4.7 QUELQUES MODES D'ADRESSAGE.

Mode d'adressage	Signification pour l'opérande	Mnémonique	Valeur binaire
immédiat	champ2 = valeur immédiate	Im	m = 000
direct	champ2 = adresse de l'opérande	D	m = 001
…	…		…
indirect	champ2 = adresse d'un mot mémoire qui contient l'adresse du mot mémoire contenant l'opérande	I	m = 011
registre seul	reg1 code un numéro de registre	Rg1	m = 110
deux registres	reg1 et reg2 codent un numéro de registre	Rg2	m = 111

Les directives

Les directives sont des pseudo-instructions : elles ne correspondent à aucune instruction machine ; ce sont des ordres destinés au traducteur assembleur.

Les directives servent notamment à la définition des variables : ainsi la directive `DS n` permet de réserver n mots mémoire pour une variable. Associer une étiquette à une déclaration de variable permet ensuite d'accéder à cette donnée par le biais d'un nom symbolique (en l'occurrence l'étiquette). De même, la directive `DC n` permet de déclarer une constante prenant la valeur n. La directive `STOP` indique la fin d'un programme.

Exemple

```
vecteur : DS 50 définition de la variable vecteur et réservation de 50 mots
un : DC 1  définition d'une constante nommée un ayant pour valeur 1
```

4.3.2 Fonctionnement de l'assembleur

L'assembleur est un programme qui traduit le langage d'assemblage en langage machine. Son fonctionnement est très proche de celui du compilateur en phase de génération de code.

La traduction effectuée par l'assembleur ne peut pas se faire en une seule passe, c'est-à-dire que l'assembleur ne peut pas travailler en traduisant directement une à une les instructions du langage d'assemblage en instructions du langage machine, ceci à cause du problème *des références en avant*, c'est-à-dire des références à des étiquettes non encore connues de l'assembleur.

Exemple

```
          JMP boucle
          …
          …
boucle : …
```

Lorsque l'assembleur traite l'instruction `JMP boucle`, il ne peut pas traduire le symbole `boucle` par une adresse, puisque l'assembleur n'a pas encore rencontré la définition de l'étiquette `boucle`. L'assembleur travaille donc en deux passes. Lors de la première passe, l'assembleur rassemble l'ensemble des symboles et étiquettes dans une table et leur associe une adresse dans le code. Lors de la deuxième passe, l'assembleur résout les références en avant en utilisant la table construite lors de la première passe.

Première passe de l'assembleur

La principale fonction de la première passe est de construire la *table des symboles*. Cette table contient une entrée pour chaque nouveau symbole du code en langage d'assemblage et met en correspondance le nom du symbole avec sa valeur dans le code. Cette valeur est une adresse si l'étiquette est associée à une instruction ou à une variable ; c'est une valeur constante si l'étiquette est associée à une constante.

Pour pouvoir affecter une adresse à chacun des symboles du code en langage d'assemblage, l'assembleur doit assigner une adresse à chacune des instructions et des déclarations de variables. Pour ce faire, l'assembleur manipule un compteur appelé *compteur d'emplacement*, mis à 0 au début de la première passe et incrémenté de la longueur de l'instruction ou de la longueur de la variable à chaque instruction ou déclaration traitée. L'assembleur réalise alors une allocation du programme machine relativement à l'adresse 0. La première passe élimine également les commentaires.

Exemple

On considère le programme en langage d'assemblage suivant qui correspond au programme machine vu précédemment :

```
var : DS 1  — définition de la variable var et réservation d'un mot mémoire
un : DC 1   — définition de la constante 1 nommée un
            LOAD Im R1, 0
boucle :    ADD Im R1, un
            STORE D R1, var
            JMP boucle
            STOP
```

La première passe de l'assembleur construit la table des symboles suivante (tableau 4.8) :

Tableau 4.8 TABLE DES SYMBOLES.

Nom du symbole	Valeur
var	0
un	1
boucle	8

Deuxième passe de l'assembleur

Lors de la deuxième passe, l'assembleur génère les instructions en code machine. Pour cela, l'assembleur remplace chaque mnémonique par son code binaire, chaque symbole par la valeur qui lui a été attribuée dans la table des symboles. S'il y a des expressions arithmétiques, celles-ci sont évaluées.

Si cela a été demandé, l'assembleur génère également un listing d'assemblage du programme. Ce listing comporte généralement pour chaque ligne : un numéro de ligne, le texte source, la représentation interne en hexadécimal ou en octal de l'instruction machine, la signalisation d'éventuelles erreurs de syntaxe.

Exemple

Le programme en langage d'assemblage suivant :

```
var : DS 1  — définition de la variable var et réservation d'un mot mémoire
un : DC 1   — définition de la constante 1 nommée un
            LOAD Im R1, 0
boucle :    ADD Im R1, un
            STORE D R1, var
            JMP D boucle
            STOP
```

équivaut au programme machine suivant :

```
adresse      codeop  m   reg1 reg2       champ2          signification
(00000000)16 cet emplacement correspond à la déclaration de la variable var
(00000004)16  00000  000  0001 0000  0000000000000000     R1 ← 0
(00000008)16  00101  000  0001 xxxx  0000000000000001     R1 ← R1 + 1
(0000000C)16  00001  001  0001 xxxx  0000000000000000     (0)16 ← R1
(00000010)16  01100  001  xxxx xxxx  0000000000001000
                     retourner à l'instruction d'adresse 816
```

4.4 CONCLUSION

Ce chapitre nous a permis d'étudier les différentes normes de codage existantes pour représenter au sein de l'ordinateur les différentes informations manipulées par celui-ci. Ce sont :

– les données qui peuvent être des nombres signés, des nombres flottants et des caractères ;
– les instructions machine composées d'un code opération qui spécifie la nature de l'opération à réaliser et d'opérandes qui spécifient les données sur lesquelles l'opération doit être effectuée. Les opérandes sont décrits selon un mode d'adressage qui permet d'indiquer la nature de l'opérande et la façon de le trouver.

Le langage d'assemblage constitue une variante symbolique du langage machine, qui lui est strictement équivalente. L'outil assembleur permet de traduire un code écrit en langage d'assemblage vers son équivalent langage machine, seul exécutable par la machine.

Chapitre **5**

Les circuits logiques

La donnée de base manipulée par la machine physique est le bit (*binary digit*) qui ne peut prendre que deux valeurs : 0 et 1. Ces 0 et 1 correspondent aux deux niveaux de voltage (0-1 et 2-5 volts) admis pour les signaux électriques issus des composants électroniques (transistors) qui constituent les circuits physiques de la machine. Ces circuits physiques sont construits à partir de circuits logiques, c'est-à-dire de circuits intégrés spécialisés, destinés à réaliser une opération booléenne.

5.1 LES CIRCUITS LOGIQUES

5.1.1 Définition

Un *circuit logique* est un circuit intégré spécialisé destiné à réaliser une opération booléenne. Pour un tel circuit, on fait correspondre aux signaux électriques existants en entrée et en sortie, un état logique valant 0 ou 1. L'état 0, correspondant à la présence d'une tension égale ou inférieure au tiers environ de la tension d'alimentation, correspond à *l'état bas* tandis que l'état 1, correspondant à la présence d'une tension égale ou supérieure aux deux tiers environ de la tension d'alimentation[1], est appelé *l'état haut*[2].

1. Les plages des tensions correspondant aux deux états dépendent de la technologie utilisée pour la réalisation des circuits (TTL, CMOS, etc.).
2. Dans le cas de la logique dite positive. Dans le cas de la logique négative, la valeur 1 est attribuée à l'état bas et la valeur 0 est attribuée à l'état haut.

Deux catégories de circuits peuvent être différenciées :

– les *circuits combinatoires* : pour de tels circuits, les valeurs en sortie ne dépendent que des valeurs en entrée ;
– les *circuits séquentiels* : pour de tels circuits, les valeurs en sortie dépendent des valeurs en entrée ainsi que du temps et des états antérieurs.

5.1.2 Les circuits combinatoires

Un circuit combinatoire est un circuit logique pour lequel les sorties ne dépendent que des entrées, et non du temps et des états antérieurs.

Le circuit combinatoire est défini lorsque son nombre d'entrées, son nombre de sorties ainsi que l'état de chaque sortie en fonction des entrées ont été précisés. Ces informations sont généralement fournies grâce à une *table de vérité* dans laquelle les entrées et les sorties sont exprimées par des variables booléennes.

La table de vérité

Une variable booléenne ou variable logique est une variable dont la valeur appartient à l'ensemble {0, 1}.

Une fonction logique de n variables définies dans le référentiel E, notée f (a_n, a_{n-1}, …, a_0), est une fonction définissant toute partie du référentiel E par la combinaison des variables (a_n, a_{n-1}, …, a_0), au moyen des opérations somme logique, produit logique et complémentation.

➤ Somme logique

Notée « + », la *somme logique* de deux ensembles A et B définis dans le référentiel E est constituée des éléments appartenant soit à A, soit à B, soit aux deux ensembles à la fois. La somme logique correspond à un opérateur OU.

➤ Produit logique

Notée « · », le *produit logique* de deux ensembles A et B définis dans le référentiel E est constitué des éléments communs aux deux ensembles A et B. Le produit logique correspond à un opérateur ET.

➤ Complémentation

Le *complément* $\overline{A}$ de l'ensemble A défini sur E, est l'ensemble $\overline{A}$ tel que : $A + \overline{A} = 1$ et $A \cdot \overline{A} = 0$.

Exemple

La fonction $f(a, b) = \overline{a} \cdot b + a \cdot b$ est une fonction logique.

La table de vérité d'une fonction booléenne f à n variables est une table à n + 1 colonnes, n colonnes correspondant aux n variables de la fonction et la $n + 1^{ème}$ au résultat de la fonction. Cette table présente autant de lignes qu'il y a de cas possibles ; plus précisément pour une fonction f à n variables, elle comporte donc 2^n lignes. L'expres-

sion de la fonction f à partir de sa table de vérité s'obtient en effectuant la somme logique des produits logiques des n variables pour lesquels le résultat de f est à 1.

Exemple

La table de vérité de la fonction f(a, b) = $\overline{a} \cdot b + a \cdot b$ est une table à 3 colonnes et 4 lignes (tableau 5.1).

Tableau 5.1 TABLE DE VÉRITÉ DE LA FONCTION F(A, B) = $\overline{A} \cdot B + A \cdot B$.

a	b	f
0	0	0
0	1	1
1	0	0
1	1	1

L'expression de la fonction booléenne f est schématisée à partir de plusieurs opérateurs de base qui représentent des fonctions logiques élémentaires par l'assemblage desquelles il est possible de réaliser des fonctions plus complexes. À chacune de ces fonctions élémentaires, correspond un circuit que l'on appelle une *porte*.

Opérateurs de base

➤ L'opérateur NON (NOT)

La fonction NON est une fonction à une variable a qui fait correspondre à celle-ci, son complément $\overline{a}$. Elle est également appelée inverseur. Sa table de vérité est (tableau 5.2) :

Tableau 5.2 TABLE DE VÉRITÉ DE LA FONCTION NON(A) = $\overline{A}$.

a	NON
0	1
1	0

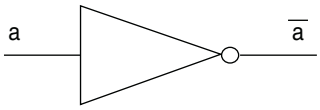


Figure 5.1 Circuit NON.

Et le circuit associé est schématisé sur la figure 5.1.

➤ L'opérateur OU (OR)

La fonction OU est une fonction à deux variables a et b qui fait correspondre à celles-ci, la somme logique de a et b, soit a + b. Sa table de vérité est (tableau 5.3) :

$$\begin{aligned} f(a,\ b) &= \overline{a} \cdot b + a \cdot \overline{b} + a \cdot b \\ &= a\,(b + \overline{b}) + a \cdot \overline{b} \\ &= a + a \cdot \overline{b} \\ &= a + b\ (\textit{loi d'absorption}) \end{aligned}$$

Tableau 5.3 TABLE DE VÉRITÉ DE LA FONCTION OU(A, B) = A + B.

a	b	OU
0	0	0
0	1	1
1	0	1
1	1	1

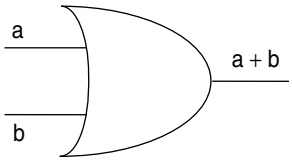


Figure 5.2 Circuit OU.

Et le circuit associé est donné par figure 5.2.

➤ L'opérateur ET (AND)

La fonction ET est une fonction à deux variables a et b qui fait correspondre à celles-ci, le produit logique de a et b, soit $a \cdot b$. Sa table de vérité est (tableau 5.4) :

Tableau 5.4 TABLE DE VÉRITÉ DE LA FONCTION ET(A, B) = A · B.

a	b	ET
0	0	0
0	1	0
1	0	0
1	1	1

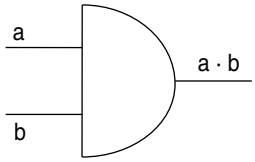


Figure 5.3 Circuit ET.

Et la figure 5.3 donne le circuit associé.

➤ L'opérateur OU exclusif (XOR)

La fonction OU exclusif est une fonction à deux variables a et b qui fait correspondre à celles-ci, une sortie à 1 si les valeurs des deux variables sont différentes. Elle est notée $a \oplus b$. Sa table de vérité est (tableau 5.5) :

Tableau 5.5 TABLE DE VÉRITÉ DE LA FONCTION XOR(A, B) = A ⊕ B.

a	b	XOR
0	0	0
0	1	1
1	0	1
1	1	0

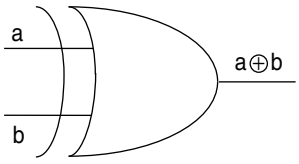


Figure 5.4 Circuit XOR.

$$f(a, b) = \bar{a} \cdot b + a \cdot \bar{b} = a \oplus b$$

Et la figure 5.4 donne le circuit associé.

Un exemple de circuit combinatoire : l'additionneur n bits

Nous allons étudier la composition du circuit combinatoire permettant de réaliser l'addition binaire de deux nombres A et B de n bits. Ce circuit est décomposé en deux parties :

– un circuit correspondant à l'addition des bits de poids faible a_0 et b_0 pour lesquels il n'y a pas de retenue propagée à prendre en compte ;
– un circuit correspondant à l'addition des bits de poids supérieur a_i et b_i, $0 < i < n$ pour lesquels il faut prendre en compte la retenue r_{i-1} propagée depuis le rang $i - 1$ inférieur.

➤ Additionneur 1 bit pour les bits de poids faible a_0 et b_0

L'addition des bits a_0 et b_0 délivre deux résultats : le bit résultat c_0 et la retenue propagée vers le rang supérieur r_0 : $a_0 + b_0$ donne la retenue r_0 et le résultat c_0 La table de vérité correspondante est (tableau 5.6) :

Tableau 5.6 TABLE DE VÉRITÉ POUR L'ADDITION DES BITS DE POIDS FAIBLE A_0 ET B_0.

a_0	b_0	c_0	r_0
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

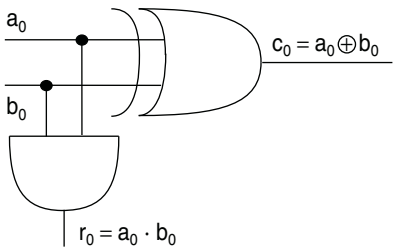


Figure 5.5 Circuit additionneur de poids faible.

Il en découle : $c_0 = \overline{a_0} \cdot b_0 + a_0 \cdot \overline{b_0} = a_0 \oplus b_0$ et $r_0 = a_0 \cdot b_0$.

Le circuit logique associé est donc formé d'une porte XOR et d'une porte ET (figure 5.5). Comme ce circuit ne tient pas compte d'une retenue propagée depuis le rang inférieur, il est qualifié de demi-additionneur.

➤ Additionneur 1 bit pour les bits de poids fort a_i et b_i

L'addition des bits de poids fort a_i et b_i doit être faite en tenant compte de la retenue r_{i-1} propagée depuis le rang $i - 1$ inférieur. Cette addition délivre deux résultats : le bit résultat c_i et la retenue propagée vers le rang supérieur r_i.

$$\begin{array}{rrr} r_i & r_{i-1} & \\ a_{i+1} & a_i & a_{i-1} \\ + \; b_{i+1} & b_i & b_{i-1} \\ \hline \multicolumn{3}{c}{\text{résultat } c_i} \end{array}$$

La table de vérité correspondante est (tableau 5.7) :

Tableau 5.7 TABLE DE VÉRITÉ POUR L'ADDITION DES BITS DE POIDS FORT A_i ET B_i.

a_i	b_i	r_{i-1}	c_i	r_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

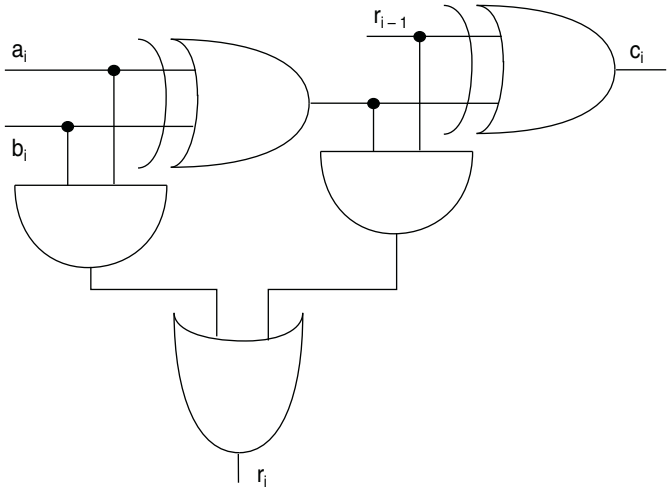


Figure 5.6 Circuit additionneur de poids fort.

Il en découle :

$$
\begin{aligned}
c_i &= \overline{a_i} \cdot \overline{b_i} \cdot r_{i-1} + \overline{a_i} \cdot b_i \cdot \overline{r_{i-1}} + a_i \cdot \overline{b_i} \cdot \overline{r_{i-1}} + a_i \cdot b_i \cdot r_{i-1} \\
&= r_{i-1}\,(\overline{a_i} \cdot \overline{b_i} + a_i \cdot b_i) + \overline{r_{i-1}}(a_i \cdot \overline{b_i} + \overline{a_i} \cdot b_i) \\
&= r_{i-1}\,\overline{(a_i \oplus b_i)} + \overline{r_{i-1}}(a_i \oplus b_i) \\
&= a_i \oplus b_i \oplus r_{i-1} \\
r_i &= \overline{a_i} \cdot b_i \cdot r_{i-1} + a_i \cdot \overline{b_i} \cdot r_{i-1} + a_i \cdot b_i \cdot \overline{r_{i-1}} + a_i \cdot b_i \cdot r_{i-1} \\
&= r_{i-1}\,(a_i \cdot \overline{b_i} + \overline{a_i} \cdot b_i) + a_i \cdot b_i(\overline{r_{i-1}} + r_{i-1}) \\
&= r_{i-1}\,(a_i \oplus b_i) + a_i \cdot b_i
\end{aligned}
$$

Le circuit logique associé, appelé additionneur complet, est donné par la figure 5.6.

➤ Additionneur complet n bits

L'additionneur n bits est obtenu en chaînant entre eux un demi-additionneur et n – 1 additionneurs 1 bit complets. Le chaînage s'effectue par le biais des retenues propagées comme le montre la figure 5.7 pour un additionneur 4 bits.

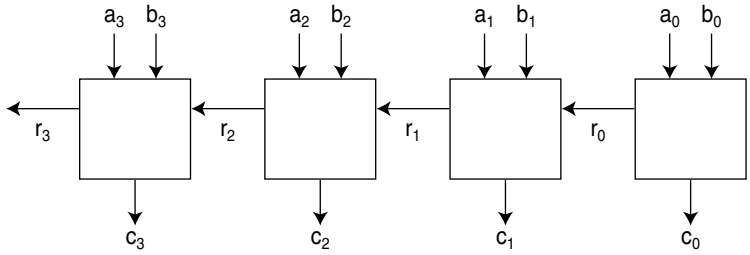


Figure 5.7 Additionneur 4 bits.

Indicateur de carry

Nous avons vu au chapitre précédent que lors d'une opération arithmétique effectuée sur des nombres de n bits, un n + 1^{e} bit, appelé bit de carry, peut être généré. Ce bit de carry, mémorisé par l'indicateur C du registre d'état du processeur, le PSW, correspond tout simplement au niveau de l'additionneur n bits, à une retenue r_{n-1} égale à 1 pour l'additionneur complet 1 bit de plus haut niveau (figure 5.8).

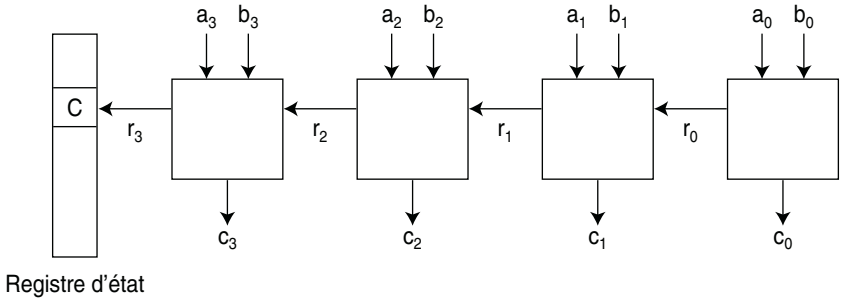


Figure 5.8 Indicateur de carry pour l'additionneur 4 bits.

Indicateur d'overflow

De même, nous avons vu au chapitre précédent que lors d'une opération arithmétique mettant en jeu des nombres de n bits et de même signe, le résultat peut être en dehors de l'intervalle des nombres représentables sur n bits par la convention choisie

pour la représentation de ces nombres signés. Ce dépassement de capacité est mémorisé dans le registre d'état du processeur, le PSW, par l'intermédiaire d'un indicateur de 1 bit noté O.

Un dépassement de capacité ne peut se produire que lors de l'addition de deux nombres de même signe, c'est-à-dire soit deux nombres positifs, soit deux nombres négatifs. Examinons ce que cela signifie en considérant que l'opération A + B = C est effectuée :

– *Cas 1* : deux nombres positifs, alors on a $a_{n-1} = b_{n-1} = 0$.

  Il se produit un overflow si le résultat de l'addition est négatif, c'est-à-dire si on a $c_{n-1} = 1$.

  On a : $a_{n-1} = b_{n-1} = 0 => a_{n-1} + b_{n-1} = 0 => r_{n-1} = 0$; d'où c_{n-1} ne peut être égal à 1 que si $r_{n-2} = 1$; on peut noter que $r_{n-2} \neq r_{n-1}$.

– *Cas 2* : deux nombres négatifs, alors on a $a_{n-1} = b_{n-1} = 1$.

  Il se produit un overflow si le résultat de l'addition est positif, c'est-à-dire si on a $c_{n-1} = 0$.

  On a : $a_{n-1} = b_{n-1} = 1 => r_{n-1} = 1$; d'où c_{n-1} ne peut être égal à 0 que si $r_{n-2} = 0$; on peut noter que $r_{n-2} \neq r_{n-1}$.

– À présent, observons quelle est la relation existante entre r_{n-2} et r_{n-1} si les deux nombres à additionner sont de signes différents, c'est-à-dire si on a $a_{n-1} \neq b_{n-1}$.
  – Si $a_{n-1} \neq b_{n-1}$, alors $a_{n-1} + b_{n-1} = 1$.
  – Si $r_{n-2} = 1$, $c_{n-1} = 0$ et $r_{n-1} = 1$.
  – Si $r_{n-2} = 0$, $c_{n-1} = 1$ et $r_{n-1} = 0$.

  On peut noter que $r_{n-2} = r_{n-1}$.

– *Conclusion* : un overflow peut être détecté en effectuant un test de comparaison entre r_{n-2} et r_{n-1}. Il y a overflow si $r_{n-2} \neq r_{n-1}$.

Regardons à présent quel circuit peut traduire cette condition. La table de vérité associée à $r_{n-2} \neq r_{n-1}$ est (tableau 5.8) :

Tableau 5.8 Table de vérité pour l'indicateur d'overflow O.

r_{n-1}	r_{n-2}	O
0	0	0
0	1	1
1	0	1
1	1	0

Ce qui correspond à une porte XOR. Donc $O = r_{n-2} \oplus r_{n-1}$ (figure 5.9).

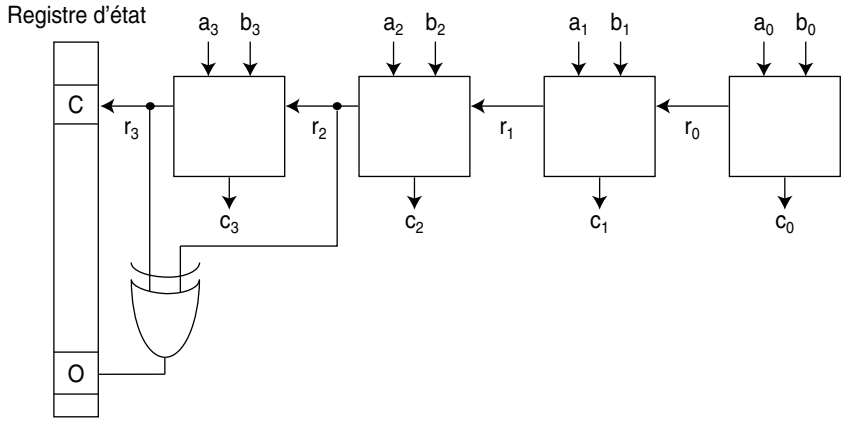


Figure 5.9 Indicateur d'overflow.

Autres circuits combinatoires

Citons quelques autres circuits combinatoires :

– le multiplexeur-démultiplexeur (figure 5.10) : un multiplexeur est un circuit qui permet d'aiguiller plusieurs entrées sur une seule sortie. Le démultiplexeur réalise la fonction inverse, c'est-à-dire qu'il permet d'aiguiller une entrée sur une sortie parmi plusieurs. Dans les deux cas, une commande de sélection C permet de régler les conflits ;

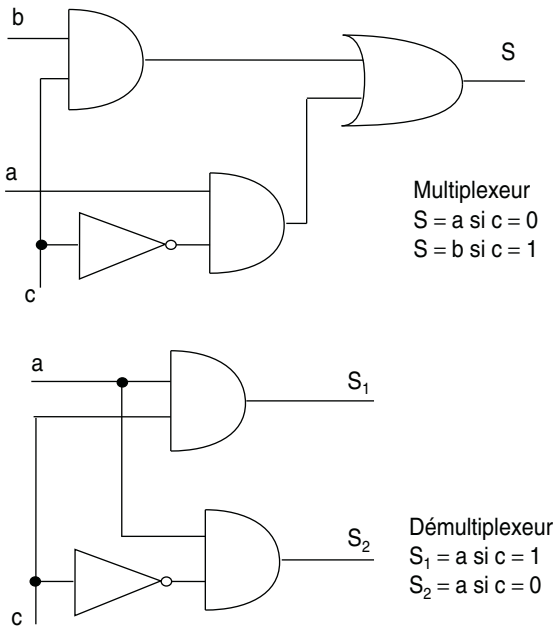


Figure 5.10 Multiplexeur et démultiplexeur à deux entrées.

– le décodeur d'adresses (figure 5.11) : ce circuit présente n entrées appelées adresse et 2^n sorties. À chaque combinaison des n entrées, correspond une seule sortie parmi les 2^n.

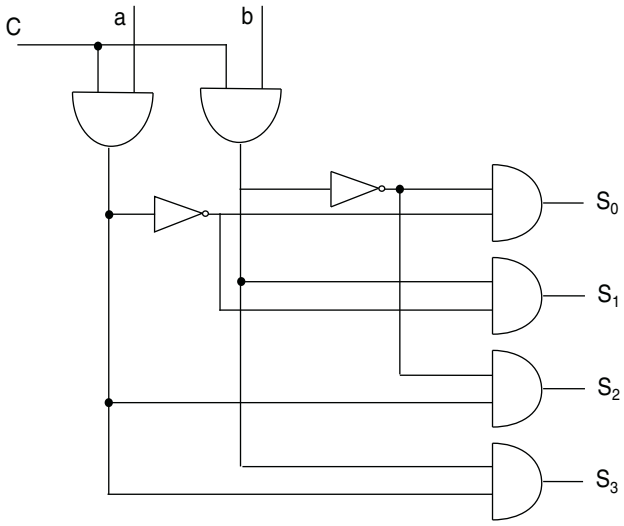


Figure 5.11 Décodeur d'adresses à deux entrées.

5.1.3 Les circuits séquentiels

Un circuit séquentiel est un circuit logique pour lequel l'état des sorties dépend de l'état des entrées, mais aussi des sorties antérieures. Ce type de circuit tient donc compte dans le temps des étapes passées, qu'il est capable de mémoriser. Tous ces circuits, que l'on qualifie de *bascules*, comprennent donc un *état de mémorisation* qui permet de mémoriser la valeur de 1 bit.

Principe de fonctionnement d'un circuit séquentiel

Considérons le circuit de la figure 5.12. Une particularité de ce circuit qui le différencie des circuits combinatoires précédents est que la sortie S du circuit est réinjectée à l'entrée du circuit. On parle de *rétroaction*. L'état de sortie du circuit va donc être influencé par l'état antérieur.

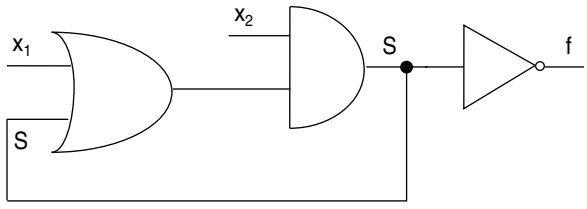


Figure 5.12 Un circuit séquentiel.

La table de vérité correspondant à ce circuit est :

Tableau 5.9 Table de vérité pour le circuit séquentiel de la figure 5.1.

x_1	x_2	S	f
0	0	0	1
0	1	S	$\overline{S}$
1	0	0	1
1	1	1	0

L'état pour lequel $x_1 = 0$ et $x_2 = 1$ correspondant à l'état de mémorisation du circuit séquentiel.

Il existe plusieurs types de bascules élémentaires qui combinées entre elles, permettent de réaliser des circuits de mémorisation complexe, telle que les registres ou encore les cellules mémoire. Nous évoquons succinctement les différentes bascules existantes puis le principe de réalisation d'un registre.

Panorama succinct des bascules élémentaires

La figure 5.13 dresse un panorama succinct des différentes bascules.

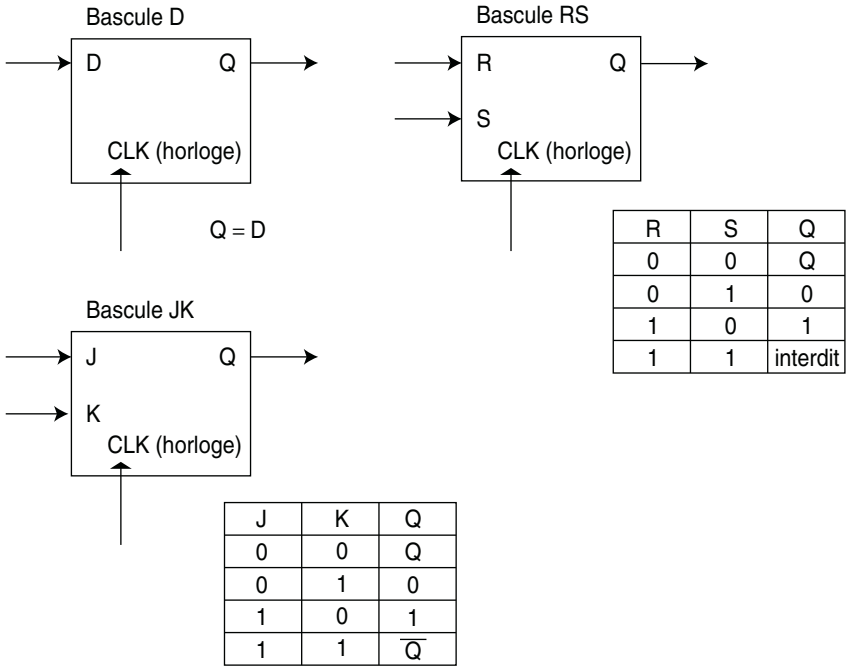


Figure 5.13 Bascules élémentaires.

➤ Bascule D

Cette bascule enregistre sur sa sortie Q la donnée présente sur son entrée D, à chaque top horloge.

➤ Bascule RS

Cette bascule possède deux entrées de données R et S. L'état pour lequel R = S = 0 est l'état de mémorisation. Les autres états (R = 1, S = 0) et (R = 0, S = 1) permettent de réinitialiser la bascule soit à 0, soit à 1. L'état (R = 1, S = 1) est interdit.

➤ Bascule JK

Cette bascule possède également deux entrées de données J et K. Elle a un fonctionnement identique à la bascule RS à la différence près que l'état (J = 1, K = 1) est autorisé et conduit à l'inversion de l'état de la bascule.

Principe de réalisation d'un registre

Un registre de stockage est un élément de mémorisation permettant de stocker une information d'une longueur de n bits. Une bascule ne permettant de mémoriser qu'un seul bit, un registre de n bits sera donc composé d'un ensemble de n bascules. Les bascules employées sont soit des bascules D, soit des bascules JK (figure 5.14).

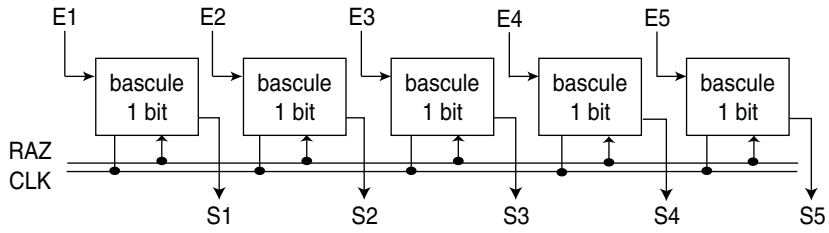


Figure 5.14 Un registre de 5 bits.

La commande RAZ permet de mettre à zéro toutes les cellules du registre en même temps. Le mot à charger est présenté sur les entrées E1 à E5 et les cinq entrées sont chargées en parallèle, sur un signal d'horloge délivré sur la ligne CLK. La lecture du mot s'effectue sur les sorties S1 à S5, également en parallèle.

5.1.4 Technologie des circuits logiques

Les circuits logiques sont construits à partir de *transistors*. Un transistor est un circuit électronique fonctionnant comme un interrupteur et pouvant prendre deux états auxquels sont associés les niveaux logiques 0 et 1 : l'état saturé et l'état bloqué. Dans l'état saturé, le transistor délivre en sortie une tension positive à laquelle on fait correspondre le niveau logique 1. Dans l'état bloqué, le transistor

au contraire délivre une tension voisine de 0, à laquelle on fait correspondre le niveau logique 0[1].

Il existe différentes technologies de transistors : les technologies TTL (*Transistor, Transistor Logic*) à base de transistors bipolaires, les technologies FET (*Field Effect Transistor*) à base de transistors à effet de champ et encore les technologies MOS (*Metal Oxyde Semi-conducteur*) à base de transistors à effet de champ MOS. Ces différentes technologies diffèrent au niveau performance, consommation électrique et niveau d'intégration, mais dans tous les cas elles mettent en jeu un même matériau : le *semi-conducteur silicium*.

Qu'est-ce qu'un semi-conducteur ?

Un semi-conducteur est un matériau à structure cristalline qui se comporte soit comme un conducteur, soit comme un isolant. Les liaisons de covalence que les atomes de ces cristaux entretiennent entre eux sont fragiles et lorsque la température augmente, les électrons engagés dans ces liaisons vont pouvoir s'agiter suffisamment pour briser ces liaisons et devenir des électrons libres. Les électrons qui quittent les liaisons entre atomes créent des « trous » chargés positivement, qui vont à leur tour attirer les électrons des liaisons voisines. Il s'ensuit la création d'un courant électrique qui rend le cristal, initialement isolant, conducteur.

La conductivité d'un semi-conducteur peut être également provoquée par l'insertion dans le cristal d'atomes étrangers qui vont soit enrichir le cristal en charges positives, soit en charges négatives. C'est l'opération de *dopage* du cristal. Dans le cas du dopage de type P, le cristal est enrichi en charges positives en introduisant dans le cristal des atomes possédant moins d'électrons que les atomes du cristal : il en découle la formation de liaisons de covalence incomplètes entre les atomes et donc une augmentation du nombre de « trous » positifs. Dans le cas du dopage de type N, le cristal est enrichi en charges négatives, en introduisant dans le cristal des atomes possédant au contraire plus d'électrons que les atomes du cristal : des électrons en surnombre ne sont pas impliqués dans les liaisons entre atomes et forment donc des électrons libres supplémentaires.

Le cristal le plus couramment utilisé pour la fabrication des circuits intégrés et des transistors est le silicium (*Si*, N = 14). Il est couramment dopé négativement avec des atomes de phosphore (*P,* N = 15) et positivement avec des atomes de bore (*B*, N = 5).

Les différentes technologies de transistors

Les transistors, quelle que soit la technologie mise en œuvre, sont composés par un assemblage de zones de silicium de type N avec des zones de silicium de type P. On distingue principalement deux familles de transistors :

– la famille TTL (*Transistor, Transistor Logic*) conçue à base de transistors bipolaires ;
– la famille MOS (*Metal Oxyde Semi-conductor*) conçue à base de transistors unipolaires MOS.

1. Dans le cas de l logique positive. L'association inverse est réalisée dans le cadre de la logique négative.

➤ Les transistors bipolaires

Un transistor bipolaire (figure 5.15) est constitué d'un empilement de trois couches de semi-conducteur, la première appelée *l'émetteur* dopée N, la seconde appelée la *base* dopée P et la troisième dénommée le *collecteur* dopée N[1].

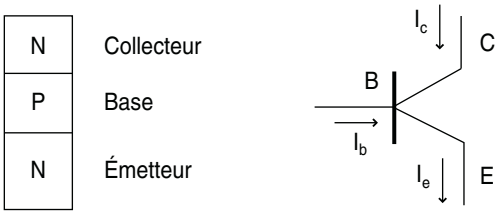


Figure 5.15 Transistor bipolaire.

Sans tension électrique appliquée à la base, les électrons de l'émetteur sont bloqués par la barrière de potentiel induite entre les régions N et P : le transistor est bloqué. Au contraire, si une tension positive est appliquée à la base, alors la barrière de potentiel s'efface et les électrons excédentaires de l'émetteur transitent vers la zone collecteur : le transistor est devenu passant (saturé).

La technologie TTL basée sur ce type de transistors permet des vitesses de fonctionnement élevées des portes logiques, au prix cependant d'une consommation de courant relativement importante.

➤ Les transistors unipolaires

Le transistor unipolaire (figure 5.16) est constitué d'une zone de silicium dopée P (*le substrat*), dans laquelle sont implantées deux zones de dopage inverse N, qui constituent la *source* et le *drain*[2]. La région intermédiaire à la source et au drain est recouverte d'une couche isolante de bioxyde de silicium (silice) dans laquelle est insérée une électrode appelée la *grille*.

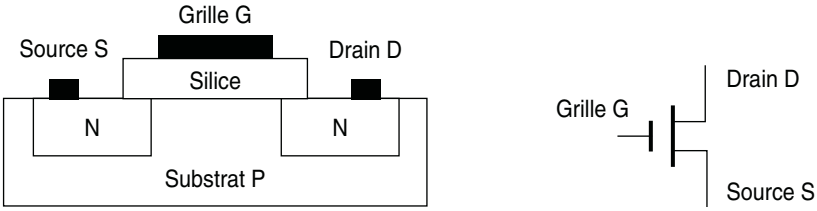


Figure 5.16 Transistor unipolaire.

1. Ceci dans le cas d'un transistor NPN. On trouve également des transistors PNP, pour lesquels l'émetteur et le collecteur sont des zones dopées P et la base est une zone dopée N.
2. Ceci dans le cas d'un transistor à canal N. Il existe également des transistors unipolaires à canal P pour lesquels la source et le drain sont des zones dopées P.

Lorsqu'une tension nulle ou négative est appliquée à la grille, les électrons sont chassés de la surface du substrat ce qui isole électriquement la source et le drain : le transistor est bloqué. Au contraire, lorsqu'une tension positive est appliquée à la grille, les électrons du substrat viennent s'accumuler à la surface, réalisant un canal d'électrons entre la source et le drain : le transistor est devenu passant.

La technologie MOS consomme moins de courant que la technologie TTL, mais elle est plus lente que cette dernière. Cette technologie est elle-même divisée en trois familles :

– la technologie PMOS qui ne met en jeu que des transistors MOS à canal P ;
– la technologie NMOS qui ne met en jeu que des transistors MOS à canal N, plus rapide que la précédente ;
– la technologie CMOS qui ne met en jeu à la fois des transistors MOS à canal P et des transistors MOS à canal N. Les circuits résultants sont peu gourmands en électricité. C'est la technologie la plus utilisée à l'heure actuelle.

➤ Un exemple : réalisation d'une porte inverseur

Considérons le circuit de la figure 5.17. Dans ce montage, le transistor MOS est chargé, par le drain, avec une impédance de + 5 volts[1]. La source est par ailleurs reliée à la masse (0 volt).

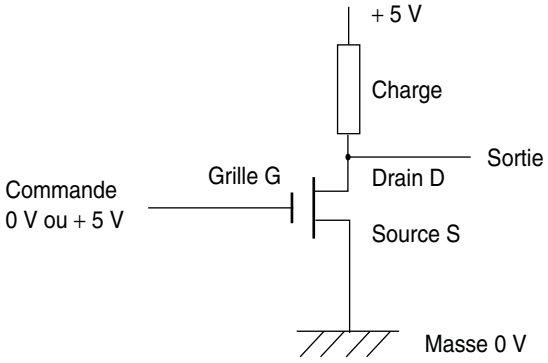


Figure 5.17 Transistor unipolaire et porte inverseur.

Une tension nulle est appliquée à la grille : le transistor est donc bloqué. Il se comporte comme un circuit ouvert. En conséquence, la tension sur la porte de sortie est celle de l'impédance de charge, soit + 5 volts.

Une tension de + 5 volts est maintenant appliquée à la grille : le transistor devient passant. Il se comporte comme un court-circuit. En conséquence, la tension sur la porte de sortie est celle de la masse, soit + 0 volt.

On voit donc ici que ce montage équivaut à une porte inverseur.

1. Sur les processeurs pentium, la tension d'alimentation des transistors est ramenée à 3,3 volts.

Fabrication d'un circuit intégré et des fonctions logiques

Les transistors, quelle que soit la technologie mise en œuvre, sont donc composés par un assemblage de zones de silicium de type N avec des zones de silicium de type P. Leur fabrication consiste donc à implanter de façon plus ou moins complexe ces différentes zones à des endroits très précis.

Cette fabrication met en œuvre un procédé particulier appelé *microlithographie optique* ou encore *photolithographie* qui permet de transférer l'image d'une plaque permanente appelée le *masque* sur une rondelle de semi-conducteur appelée *wafer*. Le masque est usuellement une plaquette de quartz sur laquelle ont été dessinés grâce à un logiciel de dessin assisté par ordinateur, les circuits du microprocesseur. Le transfert du dessin s'effectue à travers le masque en illuminant par des rayons ultraviolets une résine photosensible qui recouvre le wafer. La résine exposée est soluble par un solvant, ce qui permet après l'illumination de décaper les parties exposées pour ne laisser sur la wafer que l'empreinte des circuits à base de silice.

Plus précisément, les étapes suivantes permettent la création en parallèle de plusieurs transistors :

- la rondelle de silicium qui est dopée de type P ou N est enduite d'une couche de bioxyde de silicium (silice), puis de résine photosensible ;
- la rondelle de silicium est soumise à un rayonnement ultraviolet à travers un masque. La partie de résine exposée à la lumière est ensuite éliminée grâce à un solvant, ce qui permet de découvrir le dessin des circuits gravé sur la silice. La silice exposée est à son tour détruite chimiquement, puis la couche restante de résine qui n'avait pas été exposée est détruite. Il reste donc sur la rondelle de silicium, l'empreinte des circuits à base de silice ;
- une nouvelle couche de silice appelée oxyde de grille est à nouveau déposée sur la rondelle, puis recouverte d'une couche conductrice de polysilicium et d'une nouvelle couche de résine. Un nouveau masque est appliqué, qui correspond à la création de la grille du transistor. Le métal et l'oxyde sont attaqués comme précédemment, ce qui dénude des zones de silicium de part et d'autre de la grille créée ;
- la plaquette de silicium est maintenant soumise à un processus de dopage en vue de la création du drain et de la source ;
- enfin, la rondelle est enduite d'une couche métallique visant à mettre en contact les différentes parties du transistor et les différents transistors entre eux. Un troisième masque est appliqué, puis les parties métalliques inutiles sont détruites ;
- chaque circuit ou puce ainsi réalisé est ensuite testé puis encapsulé dans un boîtier rectangulaire en plastique ou en céramique, sur lequel des broches réparties de part et d'autre, permettent d'assurer les connexions électriques de la puce interne. Cette encapsulation est connue sous le nom de boîtier DIL ou DIP (*Dual Inline Package*). Les boîtiers PGA (*Pin Grid Array*) sont quant à eux des boîtiers carrés disposant de connexions réparties sur les quatre côtés (figure 5.18).

Toutes les étapes de cette fabrication sont réalisées dans des usines spéciales qualifiées de *salles blanches* dont les particularités sont d'une part d'être appauvries en lumière dans le spectre des bleus et des violets afin de ne pas interférer avec le

bombardement ultraviolet réalisé pour le transfert des masques, d'autre part de ne contenir que trente-cinq particules de poussières par mètres cubes d'air alors que l'air usuel en contient près de 500 millions. Les ouvriers procédant à cette fabrication sont eux-mêmes vêtus de combinaisons, de bottes et de gants hermétiques alors que l'air qu'ils exhalent est absorbé par un dispositif particulier porté à la ceinture qui protège les tranches de silicium de toute impureté susceptible d'endommager les circuits gravés.

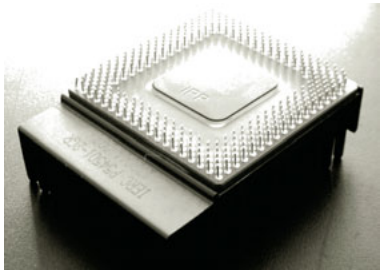


Figure 5.18 Boîtier PGA.

Niveaux d'intégration

La densité d'intégration pour un circuit logique et sa famille détermine le nombre de portes ou de transistors par circuit ou par mm^2. Ainsi, les familles suivantes peuvent être identifiées :

- SSI (*Small Scale Integration*) : ce sont des circuits à faible intégration groupant de 1 à 10 portes par circuit ;
- MSI (*Medium Scale Integration*) : ce sont des circuits à moyenne intégration groupant de 10 à 100 portes par circuit ;
- LSI (*Large Scale Integration*) : ce sont des circuits à haute intégration groupant de 100 à 100 000 portes par circuit ;
- VLSI (*Very Large Scale Integration*) : ce sont des circuits à très haute intégration groupant plus 100 000 portes par circuit. À ce niveau, se trouvent les circuits tels que les mémoires et les microprocesseurs. À titre d'exemple, la puce du processeur Intel Pentium intègre plus de trois millions de transistors, sur un boîtier comportant 273 broches.

5.2 LE FUTUR…

La densité des transistors présents sur une puce actuelle est à peu près 1 500 fois supérieures à ce qu'elle était au moment de leur invention, dans les années 1970. Cette densité, suivant la loi de Moore, augmente de 50 % tous les deux ans.

Loi de Moore

La loi de Moore est une loi proposée en 1965 par Gordon Moore, cofondateur d'Intel, et décrivant le taux de croissance du nombre de transistors par unité de surface, compte tenu des progrès technologiques. Cette loi prévoit un doublement du nombre de transistors sur une même surface tous les deux ans.

Cette augmentation incessante de la densité des transistors lève à l'heure actuelle un certain nombre de défis technologiques :

– la taille des transistors dont la largeur de traits se rapproche de plus en plus du dixième de micron ne cesse de diminuer et tend de plus en plus vers celle de la longueur d'onde de la lumière visible. L'utilisation de celle-ci ou des rayons ultraviolets dans le processus de photolithographie devient problématique pour la conservation de la netteté des images tracées. Un recours à des rayons ultraviolets de plus courte longueur d'ondes ou même aux rayons X sera sans doute nécessaire ;
– le nombre de défauts par centimètre carré de silicium qui est à l'heure actuelle de 200 pour un million de puces devra être réduit afin de tendre vers zéro ;
– la chaleur dégagée par une puce du fait de la densité de transistors présents sur celle-ci et de l'augmentation de leur vitesse de commutation tend à devenir trop importante. La tension d'alimentation des transistors a déjà été ramenée de 5 volts à 3,3 volts sur le microprocesseur Intel Pentium, mais elle devra être encore abaissée pour tendre vers 1,8 volts.

Chapitre **6**

Exercices corrigés

PRODUCTION DE PROGRAMMES

6.1 Compilation

Soit le langage défini par les règles de Backus-Naur suivantes :

```
<programme> :: = PROGRAM <identificateur> <corps de programme>
<corps de programme> :: = <suite de declarations> DEBUT <suite
➥ d'instructions> FIN
<suite de declarations> :: = <declaration> | <declaration> <suite
➥ de declarations>
<declaration> :: = INT <identificateur>; | BOOLEAN <identificateur>;
<suite d'instructions> :: = <instruction> | <instruction> <suite
➥ d'instructions>
<instruction> :: = <affectation> | <conditionnelle> | <iteration>
<affectation> :: = <identificateur> = <terme>; | <identificateur> = <terme>
<operateur> <terme>; | <identificateur> = VRAI; |
➥ <identificateur> = FAUX;
<conditionnelle> :: = SI <expression> ALORS <suite d'instructions> FSI
<iteration> :: = LOOP <expression> FAIRE <suite d'instructions> FAIT
<expression> :: = (<terme> <operexpr> <terme>) | (<identificateur> == VRAI) |
➥ (<identificateur> == FAUX)
<operexpr> :: = == | <|> | ?
<terme> :: = <entier>| <identificateur>
<operateur> :: = + | – | * | /
<identificateur> :: = <lettre> | <lettre> <chiffre>
```

```
<entier> :: = <chiffre> | <entier> <chiffre>
<lettre> :: = A | B | C | D | E … | X | Y | Z
<chiffre> :: = 0 | 1 | 2 | 3 | 4 … | 9
```

Soit à présent le programme suivant :

```
PROGRAM Y3
INT A;
INT B;
BOOLEAN C;
DEBUT
A = 6;
B = A * 3;
C = VRAI;
LOOP (C = VRAI)
FAIRE
SI (A > B) ALORS C = FAUX FSI
B = B – 1;
FAIT
FIN
```

1. Donnez la suite de codes obtenus à l'issue de l'analyse lexicale du programme sachant que :

```
cas :
entier : codage_lexème = valeur de l'entier;
symbole + : codage_lexème = – 1;
symbole – : codage_lexème = – 2;
symbole * : codage_lexème = – 3;
symbole / : codage_lexème = – 4;
symbole = : codage_lexème = – 5;
symbole; : codage_lexème = – 6;
symbole PROGRAM : codage_lexème = – 7;
symbole DEBUT : codage_lexème = – 8;
symbole FIN : codage_lexème = – 9;
symbole INT : codage_lexème = – 10;
symbole SI : codage_lexème = – 11;
symbole ALORS : codage_lexème = – 12;
symbole FSI : codage_lexème = – 13;
symbole VRAI : codage_lexème = – 14;
symbole FAUX : codage_lexème = – 15;
symbole LOOP : codage_lexème = – 16;
symbole FAIRE : codage_lexème = – 17;
symbole FAIT : codage_lexème = – 18;
symbole ( : codage_lexème = – 19;
symbole ) : codage_lexème = – 20;
```

```
symbole < : codage_lexème = – 21;
symbole > : codage_lexème = – 22;
symbole == : codage_lexème = – 23;
symbole ? : codage_lexème = – 24;
symbole BOOLEAN : codage_lexème = – 25;
identificateur lettre seule : – (position_lettre_alphabet + 25);
identificateur lettre chiffre : – ((position_lettre_alphabet + 25)
➥ + 26 (chiffre + 1));
fin cas;
```

2. Peut-on construire l'arbre syntaxique de ce programme ? Pourquoi ?

6.2 Édition des liens

On considère un ensemble de quatre modules objets participant à la construction d'un même programme exécutable prog.exe. À l'issue de la compilation, les liens utilisables et les liens à satisfaire suivants ont été référencés dans chacun des modules.

```
module module_1;
taille (module) = 1 024 Ko
LU <afficher, 128>
LU <imprimer, 260>
LU <enregistrer, 512>
fin module

module module_2;
taille (module) = 512 Ko
LAS <afficher, 1>
LAS <perimetre, 2>
LAS <surface, 3>
LU <rectangle, 64>
LU <carre, 128>
fin module

module module_3;
taille (module) = 64 Ko
LU <perimetre, 16>
LU <surface, 32>
fin module

module module_4;
taille (module) = 100 Ko
LAS <rectangle, 1>
LAS <carre, 2>
LAS <imprimer, 3>
LAS <enregistrer, 4>
fin module
```

1. Construisez la carte d'implantation du programme exécutable en supposant que les modules sont mis dans l'ordre 2, 3, 4, 1.
2. Construisez la table des liens. Les modules sont pris en compte dans l'ordre de la carte d'implantation. L'édition des liens est-elle correcte ?

6.3 Utilitaire Make

Soit le graphe de dépendance du programme exécutable `prog.exe` donné sur la figure 6.1.

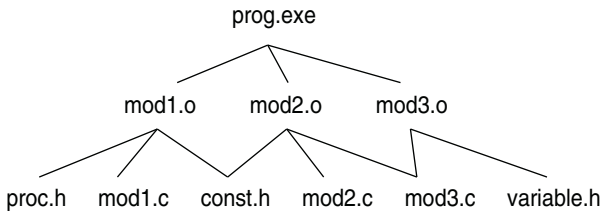


Figure 6.1 Graphe de dépendance du programme prog.exe.

1. Donnez la structure du Makefile correspondant.
2. Que se passe-t-il si le fichier `const.h` est modifié ?

6.4 Compilation

Soit le langage défini par les règles de Backus-Naur suivantes :

```
<programme> ::= PROGRAM <nom de programme> <corps de programme>
<corps de programme> ::= <suite de declarations><suite d'instructions> FIN
<suite de declarations> ::= <declaration> | <declaration> <suite de decla-
rations>
<declaration> ::= <identificateur> : REEL := <valeur réelle>; |
<identificateur> : ENTIER := <valeur entière>;
<suite d'instructions> ::= <instruction> | <instruction> <suite d'instruc-
tions>
<instruction> ::= <addition> | <multiplication>
<identificateur> ::= <lettre>
<valeur entière> ::= <chiffre>
<valeur réelle> ::= <chiffre>, <chiffre>
<nom de programme> ::= <lettre> <chiffre>
<lettre> ::= A |B | C | D | E....| X | Y | Z
<chiffre> ::= 0 |1 | 2 | 3 | 4...| 9
```

1. Une addition est de la forme `A := B + 3;` ou `A := B + 3,2;`
   A et B sont des identificateurs, 3 est un entier et 3,2 est un réel.

Une multiplication est de la forme suivante : `A := B * C;` A, B et C sont des identificateurs.

Écrivez la règle BNF donnant la syntaxe d'une addition et celle donnant la syntaxe d'une multiplication.

2. Soit à présent le programme suivant :

```
PROGRAM A3 /
    A : REEL := 3,2;
    B : ENTIER := 3,2;
    A := A + 5,1;
    B := B * 61,
FIN
```

Entourez les unités lexicales reconnues. Signalez les éventuelles erreurs lexicales rencontrées à ce niveau.

3. Pour chaque phrase du programme donnée ci-dessous, donnez l'arbre syntaxique correspondant. Signalez les éventuelles erreurs syntaxiques rencontrées à ce niveau.

```
Phrase 1      A : REEL := 3,2;
Phrase 2      B : ENTIER := 3,2;
Phrase 3      A := A + 5,1;
Phrase 4      B := B * 61,
```

REPRÉSENTATION DES INFORMATIONS

6.5 Conversions

Effectuez les conversions suivantes :

- $+ 1\,432{,}45_{10}$ vers la base 2.
- $+ 1\,432{,}45_{10}$ vers la base 16.
- $+ 1\,432{,}45_{16}$ vers la base 10.
- 111010100101_2 vers la base 10, puis vers la base 8.

6.6 Représentation des nombres signés

1. On considère une représentation des nombres signés sur 16 bits. Donnez la représentation interne de :
   - $+ 1\,032_{10}$ en valeur signée, en complément à 2, puis en DCB.
   - $- 721_{10}$ en valeur signée, en complément à 2, puis en DCB.
2. On considère une représentation des nombres signés sur 16 bits. Donnez la valeur décimale des chaînes binaires suivantes, en considérant successivement qu'il s'agit d'une représentation en valeur signée, en complément à 2, puis en DCB :
   - $0101\ 0000\ 1100\ 1101_2$
   - $1010\ 0001\ 0011\ 0011_2$

6.7 Représentation des nombres flottants

1. On considère une représentation des nombres flottants selon la norme IEEE 754 simple précision. Quelle est la représentation interne des nombres suivants ? Exprimez le résultat final en base 16.
   - $+ 1\,432{,}45_{10}$
   - $- 721{,}25_{10}$
2. On considère une représentation des nombres flottants selon la norme IEEE 754 simple précision. Quelle est la valeur décimale des représentations internes suivantes ?
   - $C6570000_{16}$
   - $42EF9100_{16}$

6.8 Synthèse

Codez l'information $- 78_{10}$ selon les formats suivants :
– valeur signée sur 8 bits ;
– complément à 2 sur 8 bits ;
– IEEE 754 simple précision.

LANGAGE MACHINE

Dans la série d'exercices qui suit, nous utilisons le langage machine d'un processeur virtuel que nous spécifions maintenant.

Les registres de l'unité centrale sont des registres de 32 bits.

On distingue :

– un compteur ordinal CO, qui contient l'adresse de la prochaine instruction à exécuter ;
– un registre instruction RI, qui contient l'instruction couramment exécutée ;
– 4 registres généraux : de R0 à R3 ;
– 1 registre spécialisé pour l'adressage : RB, le registre de base ;
– 1 registre Pointeur de pile : RSP ;
– les registres RAD et RDO ;
– 1 registre d'état, PSW, qui contient notamment les indicateurs suivants :
  – O : positionné à 1 si overflow, 0 sinon ;
  – Z : positionné à 1 si résultat opération nul, 0 sinon ;
  – C : positionné à 1 si carry, 0 sinon ;
  – S : positionné à 0 si résultat opération positif, 1 sinon ;
  – I : masquage des interruptions : positionné à 1 si interruption masquée, 0 sinon.

L'unité centrale contient également une Unité Arithmétique et Logique.

La machine admet des instructions sur 32 bits, selon le format suivant :

– un code opération codé sur 8 bits ;
– un mode d'adressage m codé sur 4 bits ;

– un champ reg sur 4 bits code un numéro de registre (de 0000 à 1111);
– un champ X sur 16 bits code une valeur immédiate, une adresse mémoire, un déplacement ou un numéro de registre.

La machine supporte les modes d'adressage mémoire suivants :

Mode	Signification	Mnémonique	Valeur binaire
Immédiat	Opérande = valeur immédiate	Im	m = 0000
Direct	Opérande = [adresse]	D	m = 0001
Indirect	Opérande = [[adresse]]	I	m = 0010
Basé	Opérande = [[RB] + déplacement]	B	m = 0011

Pour les valeurs de m comprises entre 0 et 3, le code opération travaille sur deux opérandes : le premier est un registre dont le numéro est codé par le champ reg; le second est soit une valeur immédiate, soit une adresse déduite de X et m.

Les valeurs m = 0100 et m = 0101 sont utilisées pour les opérations sur des registres :

– 0100 (Rg1) : le code opération travaille sur un seul opérande registre reg;
– 0101 (Rg2) : le code opération travaille sur deux registres, l'un codé dans le champ reg, l'autre dans le champ X.

La valeur reg allant de 0000 à 0011 code les numéros de registres généraux R0 à R3. Les autres valeurs sont réservées pour coder les autres registres du processeur. Ainsi :

– reg = 1110 désigne le registre RB;
– reg = 1111 désigne la registre PSW.

Ces valeurs de registres sont également utilisées dans le champ X avec un mode d'adressage Rg2.
Le jeu d'instructions du processeur comporte les instructions suivantes :

Les instructions de transfert de données

➤ Transfert d'un mot mémoire vers un registre banalisé

LOAD m reg X	m = B, D, I, Im	X est une adresse ou un déplacement ou une valeur immédiate.

Exemples

`LOAD D R1 (000A)`$_{16}$: chargement du registre R1 avec la case mémoire d'adresse $(000A)_{16}$ adressée en mode direct.
`LOAD Im R1 (000A)`$_{16}$: chargement du registre R1 avec la valeur immédiate $(000A)_{16}$.

➤ Transfert d'un registre vers un mot mémoire

STORE m reg X	m = B, D, I	X est une adresse ou un déplacement

Exemple

`STORE D R1 (000A)`$_{16}$: écriture du contenu du registre R1 dans la case mémoire d'adresse $(000A)_{16}$ adressée en mode direct.

Les instructions de traitement des données

Ces instructions regroupent les fonctions mathématiques et les fonctions booléennes. Dans ces instructions, le registre d'état PSW est modifié en fonction du résultat de l'opération.

➤ Fonctions arithmétiques

ADD m reg X	m = B, D, I, Rg2, Im X est soit une valeur immédiate, soit une adresse mémoire, soit un déplacement, soit un numéro de registre.	Addition entre le contenu de reg et l'opérande déduit de m et X, puis stockage du résultat dans reg.
MUL m reg X	m = B, D, I, Rg2, Im X est soit une valeur immédiate, soit une adresse mémoire, soit un déplacement, soit un numéro de registre.	Multiplication entre le contenu de reg et l'opérande déduit de m et X, puis stockage du résultat dans reg.
NEG Rg1 reg		Complément à 2 de reg puis stockage du résultat dans reg.

Exemple

`ADD Im R0 (000A)`$_{16}$: addition de la valeur immédiate $(000A)_{16}$ avec le contenu du registre R0 et stockage du résultat dans R0.

➤ Fonctions booléennes

AND m reg X OR m reg X XOR m reg X	m = B, D, I, Rg2, Im X est soit une valeur immédiate, soit une adresse mémoire, soit un déplacement, soit un numéro de registre.	ET logique (OU, OU exclusif) entre le contenu de reg et l'opérande déduit de m et X, puis stockage du résultat dans reg.
NOT Rg1 reg		Complément à 1 de reg puis stockage du résultat dans reg.

Exemple

`AND Im R0 (000A)`$_{16}$: ET logique entre la valeur immédiate $(000A)_{16}$ et le contenu du registre R0 et stockage du résultat dans R0.

Les instructions de rupture de séquence

Ces instructions permettent d'effectuer des sauts dans le code d'un programme, vers une instruction donnée. Il existe deux types de sauts : les sauts inconditionnels qui sont toujours réalisés et les sauts conditionnels qui ne sont effectués que si une condition est vraie au moment où l'instruction est exécutée par le processeur.

➤ Sauts inconditionnels

Cette instruction permet un branchement à une adresse donnée et ceci inconditionnellement. Le saut est donc toujours effectué.

JMP X	X est une adresse. Les champs reg et m sont sans signification.	Saut inconditionnel à l'adresse X.

Exemple

JMP $(12CF)_{16}$: saut à l'adresse $(12CF)_{16}$.

➤ Sauts conditionnels

Ces instructions permettent d'effectuer un branchement à une adresse donnée si une condition est réalisée. Ces conditions sont relatives aux indicateurs du registre d'état PSW.

Si la condition n'est pas réalisée, l'exécution du code se poursuit en séquence.

JMPP X	Saut si positif. X est une adresse. Les champs reg et m sont sans signification.	Saut à l'adresse X conditionné au positionnement à 0 du bit S du registre PSW.
JMPN X	Saut si négatif. X est une adresse. Les champs reg et m sont sans signification.	Saut à l'adresse X conditionné au positionnement à 1 du bit S du registre PSW.
JMPO X	Saut si overflow. X est une adresse. Les champs reg et m sont sans signification.	Saut à l'adresse X conditionné au positionnement à 1 du bit O du registre PSW.
JMPC X	Saut si carry. X est une adresse. Les champs reg et m sont sans signification.	Saut à l'adresse X conditionné au positionnement à 1 du bit C du registre PSW.
JMPZ X	Saut si zéro. X est une adresse. Les champs reg et m sont sans signification.	Saut à l'adresse X conditionné au positionnement à 1 du bit Z du registre PSW.

Les instructions de manipulation de la pile

Ces instructions permettent d'enregistrer un élément dans la pile ou d'ôter un élément de la pile. La pile est une zone mémoire gérée selon un ordre LIFO (*Last In First Out*).

➤ Enregistrer un élément dans la pile

PUSH Rg1 reg	Le contenu du registre reg est écrit dans le mot au sommet de la pile.

➤ Ôter un élément de la pile

POP Rg1 reg	Le mot au sommet de la pile est copié dans le registre reg.

6.9 Manipulation des modes d'adressage

À l'issue de l'exécution du code assembleur suivant et compte tenu de l'état initial de la mémoire et des registres du processeur, la case mémoire d'adresse 1000 a pour contenu la valeur 100, a ou 1998 ?
La représentation des nombres signés utilise la convention du complément à 2.

Adresse mémoire	Contenu
400	2000
404	412
408	d
412	3000
416	305

Registre	Contenu
RB	100

Code assembleur :

```
LOAD D R0 400
LOAD Im R1 1002
ADD Rg2 R0 R1
NEG Rg1 R1
ADD I R1 404
STORE B R1 900
```

6.10 Programme assembleur

Écrivez un programme en langage d'assemblage qui réalise le calcul suivant :
$B = (A \times 5) + (6 + B)$.
A et B sont deux variables correspondant chacune à un mot mémoire.

6.11 Manipulation de la pile

La pile constitue une zone particulière de la mémoire centrale gérée selon une politique LIFO (*Last In, First Out*). Cette structure de données est caractérisée par :

– une base qui constitue le fond de la pile ;
– un sommet dont l'adresse est contenue dans un registre particulier du processeur, le registre RSP (*Register Stack Pointer*).

Seul le sommet de la pile repéré par le registre RSP est accessible en lecture ou en écriture par deux opérations spécifiques : PUSH et POP. Le registre RSP contient l'adresse du prochain mot où écrire dans la pile.

Plus précisément, l'opération `PUSH opérande` écrit l'opérande à l'adresse mémoire contenue dans le registre RSP puis incrémente le contenu du registre RSP pour que celui-ci contienne l'adresse du prochain mot où l'écriture se fera dans la pile. L'opération `POP destination`, où destination est un registre, décrémente le registre RSP pour que celui-ci contienne l'adresse de la donnée présente au sommet de la pile, puis retire cette donnée de la pile pour la placer dans destination.

La politique LIFO appliquée ici signifie que la dernière donnée enregistrée dans la pile est la première à en sortir.

Ainsi la figure 6.2 illustre le principe de ces deux opérations avec une pile dont la base est constituée par le mot d'adresse $(0020)_{16}$. En reprenant le contexte mémoire de l'exercice précédent et en considérant une pile dans l'état du dernier dessin de la figure 6.2, représentez l'évolution de la pile et du registre RSP au fur et à mesure de l'exécution des opérations suivantes :

```
POP Rg1 R1
POP Rg1 R3
STORE D R3 100
LOAD Im R2 7
ADD D R2 100
PUSH Rg1 R2
```

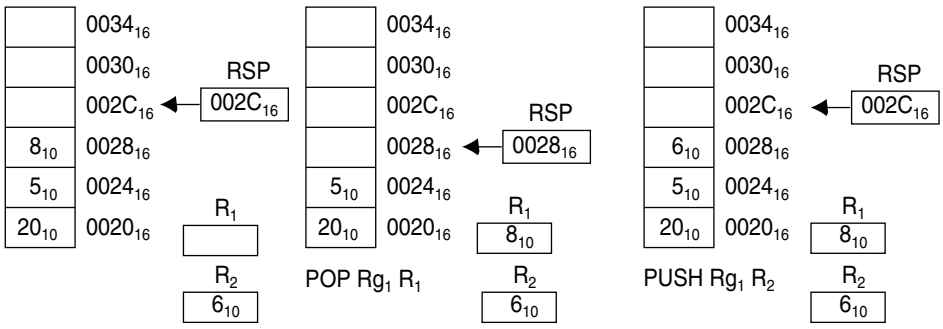


Figure 6.2 Pile, opérations `PUSH` et `POP`.

6.12 Programme assembleur

Écrivez un programme assembleur qui additionne l'entier contenu dans une case de mémoire centrale d'adresse A avec les trois premiers nombres retirés de la pile.

Si l'opération d'addition produit un carry, alors le calcul est arrêté et le résultat est stocké dans la pile.

Si l'opération d'addition produit un overflow, alors le calcul est arrêté, et le résultat de l'opération est stocké à l'adresse en mémoire centrale (RB) + 100.

Si l'opération d'addition ne provoque ni carry, ni overflow le résultat de l'opération est stocké à l'adresse A.

SOLUTIONS

6.1 Compilation

1. La suite des codes générés lors de l'analyse lexicale est :
   – 7 – 154 – 10 – 26 – 6 – 10 – 27 – 6 – 25 – 28 – 6 – 8 – 26 – 5 6 – 6 – 27 – 5 – 26 – 33 – 6 – 28 – 5 – 14 – 6 – 16 – 19 – 28 – 5 – 14 – 20 – 17 – 11 – 19 – 26 – 22 – 27 – 20 – 12 – 28 – 5 – 15 – 13 – 27 – 5 – 27 – 2 1 – 6 – 18 – 9
2. La solution est donnée par la figure 6.3. L'arbre syntaxique ne peut pas être construit jusqu'au bout car il existe une erreur de syntaxe au niveau de l'expression de la boucle `LOOP`. Un signe = a été mis à la place d'un signe = =.

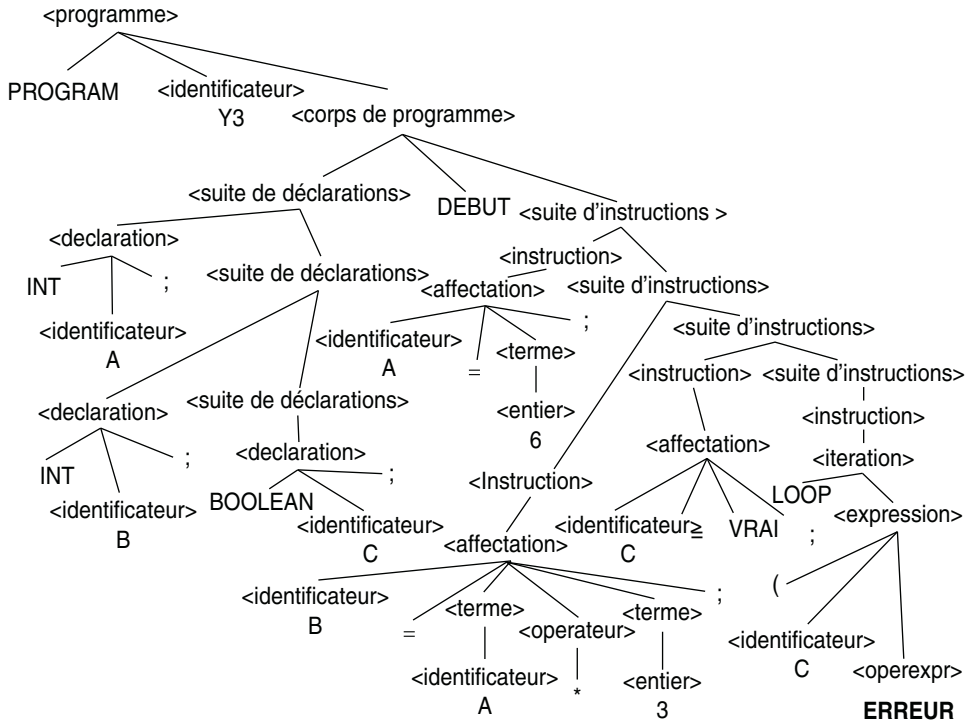


Figure 6.3 Arbre syntaxique.

6.2 Édition des liens

1. La carte d'implantation des modules est donnée par la figure 6.4 :

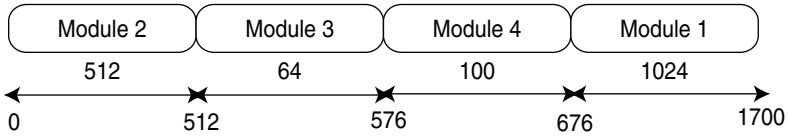


Figure 6.4 Carte d'implantation.

2. La table des symboles évolue comme le décrivent les 4 tableaux qui suivent (tableaux 6.2 à 6.5).

Tableau 6.2 Prise en compte du module 2.

Nom de l'objet	Adresse dans la carte d'implantation	Commentaire
afficher	indéfinie	LAS
perimetre	indéfinie	LAS
surface	indéfinie	LAS
rectangle	64	LU
carre	128	LU

Tableau 6.3 Prise en compte du module 3.

Nom de l'objet	Adresse dans la carte d'implantation	Commentaire
afficher	indéfinie	LAS
perimetre	16 + 512	LAS/LU
surface	32 + 512	LAS/LU
rectangle	64	LU
carre	128	LU

Tableau 6.4 Prise en compte du module 4.

Nom de l'objet	Adresse dans la carte d'implantation	Commentaire
afficher	indéfinie	LAS
perimetre	16 + 512	LAS/LU
surface	32 + 512	LAS/LU
rectangle	64	LU
carre	128	LU
imprimer	indéfinie	LAS
enregistrer	indéfinie	LAS

Tableau 6.5 PRISE EN COMPTE DU MODULE 1.

Nom de l'objet	Adresse dans la carte d'implantation	Commentaire
afficher	128 + 676	LAS/LU
perimetre	16 + 512	LAS/LU
surface	32 + 512	LAS/LU
rectangle	64	LU
carre	128	LU
imprimer	260 + 676	LAS/LU
enregistrer	512 + 676	LAS/LU

À l'issue de la prise en compte de tous les modules, aucune entrée de table des liens ne demeure indéfinie. L'édition des liens est donc correctement réalisée.

6.3 Utilitaire Make

1. Le fichier Makefile a la structure suivante :

```
prog.exe : mod1.o mod2.o mod3.o
ld mod1.o mod2.o mod3.o – o prog.exe
mod1.o : proc.h mod1.c const.h
cc – c mod1.c
mod2.o : mod2.c mod3.c const.h
cc – c mod2.c mod3.c
mod3.o : mod3.c variable.h
cc – c mod3.c
```

2. La modification du fichier `const.h` entraîne les actions suivantes :
   – Reconstruction du fichier `mod2.o` et du fichier `mod1.o` par compilation ;

```
cc – c mod2.c mod3.c
cc – c mod1.c
```

   – Reconstruction du fichier prog.exe par édition des liens.

```
ld mod1.o mod2.o mod3.o – o prog.exe
```

6.4 Compilation

1. `<addition> ::= <identificateur> := <identificateur> + <valeur entière>; | <identificateur> := <identificateur> + <valeur réelle>;`
   `<multiplication> ::= <identificateur> := <identificateur> * <identificateur>;`

2. PROGRAM A3 /
A : REEL := 3,2 ;

```
B : ENTIER := 3,2 ;
A := A + 5,1 ;
B := B * 6 1 ,
FIN
```

Une erreur lexicale est levée à la lecture du symbole / qui ne fait pas partie des symboles admis dans le langage.

3. La figure 6.5 donne les arbres syntaxiques correspondant aux quatre phrases.

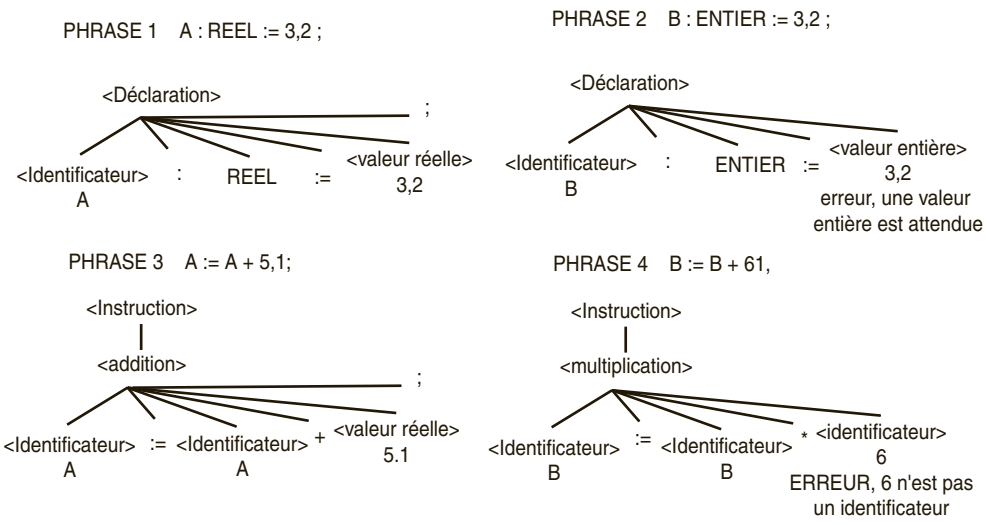


Figure 6.5 Arbre syntaxique.

6.5 Conversions

- $+ 1\,432{,}45_{10}$ vers la base 2 :
  $+ 1\,432_{10} = 1\,024 + 256 + 128 + 16 + 8 = 2^{10} + 2^8 + 2^7 + 2^4 + 2^3 = 10110011000_2$
  $0{,}45 \times 2 = 0{,}90$; $0{,}90 \times 2 = 1{,}8$; $0{,}8 \times 2 = 1{,}6$; $0{,}6 \times 2 = 1{,}2$; $0{,}2 \times 2 = 0{,}4$; $0{,}4 \times 2 = 0{,}8$
  d'où $0{,}45_{10} = 011100110011_2$
  d'où $+ 1\,432{,}45_{10} = 10110011000, 011100110011_2$
- $+ 1\,432{,}45_{10}$ vers la base 16 :
  $+ 1432{,}45_{10} = 101\ 1001\ 1000, 0111\ 0011\ 0011_2 = 598{,}733_{16}$
- $+ 1\,432{,}45_{16}$ vers la base 10 :
  $1 \times 16^3 + 4 \times 16^2 + 3 \times 16^1 + 2 \times 16^0 + 4 \times 16^{-1} + 5 \times 16^{-2} = + 5\,170{,}2695_{10}$
- 111010100101_2 vers la base 10 :
  $2^{11} + 2^{10} + 2^9 + 2^7 + 2^5 + 2^2 + 2^0 = 2\,048 + 1\,024 + 512 + 128 + 32 + 4 + 1 = + 3\,749_{10}$
- $111\ 010\ 100\ 101_2$ vers la base 8 : 7245_8

6.6 Représentation des nombres signés

1. Représentation interne :
   - $+ 1\,032_{10} = 1\,024 + 8 = 0000010000001000_2$ en valeur signée et en complément à 2.
     $+ 1\,032_{10} = 1011\ 0001\ 0000\ 0011\ 0010_2$ en DCB.
   - $+ 721_{10} = 512 + 128 + 64 + 16 + 1 = 0000001011010001_2$
     d'où $- 721_{10} = 1000001011010001_2$ en valeur signée ;
     d'où $- 721_{10} = 1111110100101111_2$ en complément à 2.
   - $721_{10} = 1101\ 0111\ 0010\ 0001_2$ en codage DCB.
2. Valeur décimale :
   - $0101\ 0000\ 1100\ 1101_2 = + 20\,685_{10}$ en valeur signée et en complément à 2.
     $0101\ 0000\ 1100\ 1101_2$ ne peut pas être un nombre codé selon le format DCB car les deux derniers quartets codent des valeurs supérieures à 9.
   - $1010\ 0001\ 0011\ 0011_2 = - 8\,499_{10}$ en valeur signée.
     $1010\ 0001\ 0011\ 0011_2 = - 0101\ 1110\ 1100\ 1101_2 = - 24\,269_{10}$ en complément à 2.
     $1010\ 0001\ 0011\ 0011_2$ ne peut pas être un nombre codé selon le format DCB car le quartet de poids fort correspond à un code supérieur à 9.

6.7 Représentation des nombres flottants

1. Représentation interne :
   - $+ 1\,432{,}45_{10} = 10110011000{,}0111001100_2 = 1{,}01100110000111001100_2 \times 2^{10}$
     L'exposant excentré est égal à $10 + 127 = 137 = 10001001_2$ d'où la représentation interne simple précision :
     $0\ 10001001\ 01100110000111001100000_2 = 44B30E60_{16}$
   - $721{,}25_{10} = 1011010001{,}01_2 = 1{,}01101000101_2 \times 2^9$
     L'exposant excentré est égal à $9 + 127 = 136 = 10001000_2$ d'où la représentation interne simple précision :
     $1\ 10001000\ 01101000101000000000000_2 = C4345000_{16}$
2. Valeur décimale :
   - $C6570000_{16} = 1\ 10001100\ 10101110000000000000000_2$
     Le signe de la mantisse est négatif. L'exposant est égal à $140 - 127 = 13$
     La mantisse est : $1{,}10101110000000000000000_2$ d'où le nombre représenté est :
     $- 1{,}1010111_2 \times 2^{13} = - 14\,080_{10}$
   - $42EF9100_{16} = 0\ 10000101\ 11011111001000100000000_2$
     Le signe de la mantisse est positif. L'exposant est égal à $133 - 127 = 6$.
     La mantisse est $1,\ 11011111001000100000000_2$ d'où le nombre représenté est :
     $1,\ 110111110010001_2 \times 2^6 = + 119{,}783203125_{10}$.

6.8 Synthèse

Valeur signée sur 8 bits : 11001110
Complément à 2 sur 8 bits : 10110010

IEEE 754 simple précision : 1 10000101 00111000000000000000000 soit $(C29C0000)_{16}$

6.9 Manipulation des modes d'adressage

LOAD D R0 400	R0 est chargé avec la valeur contenue dans le mot mémoire d'adresse 400 soit 2000
LOAD Im R1 1002	R1 est chargé avec la valeur immédiate 1002
ADD Rg2 R0 R1	R0 ⟵ R0 + R1 = 3002
NEG Rg1 R1	R1 ⟵ – 1002 (le complément à 2 représente l'équivalent négatif du nombre)
ADD I R1 404	R1 ⟵ R1 + ((404)) = – 1002 + 3000 = 1998. Le mot mémoire d'adresse 404 contient l'adresse de l'opérande. L'opérande est donc le contenu du mot d'adresse 412.
STORE B R1 900	(1000) ⟵ 1998. Le contenu du registre R1 est écrit à l'adresse obtenue selon un mode basé (RB) + 900, soit 100 + 900 = 1000

6.10 Programme assembleur

```
LOAD D R1 A
LOAD D R2 B
ADD Im R2 6
MUL Im R1 5
ADD Rg2 R1 R2
STORE D R1 B
```

6.11 Manipulation de la pile

La solution est donnée par la figure 6.6.

6.12 Programme assembleur

```
          LOAD D R1 A -- R1 est chargé avec le contenu du mot d'adresse A.
          LOAD IM R2 3 -- R2 est chargé avec la valeur immédiate 3.
Loop :    POP Rg1 R3 -- Le sommet de pile est placé dans R3.
          ADD Rg2 R1 R3 -- R1 = R1 + R3
          JMPC Carry -- si l'addition précédente produit un carry aller
          ➥ à l'instruction désignée par l'étiquette Carry.
          JMPO Overflow -- si l'addition précédente produit un overflow aller
          ➥ à l'instruction désignée par l'étiquette Overflow.
          ADD Im R2 – 1 -- R2 = R2 – 1
          JMPZ Finok -- si R2 est égal à 0, sortie de boucle. Les trois premiers
          ➥ éléments de la pile ont été pris en compte.
          JMP Loop -- sinon aller à l'instruction appelée Loop et continuer.
```

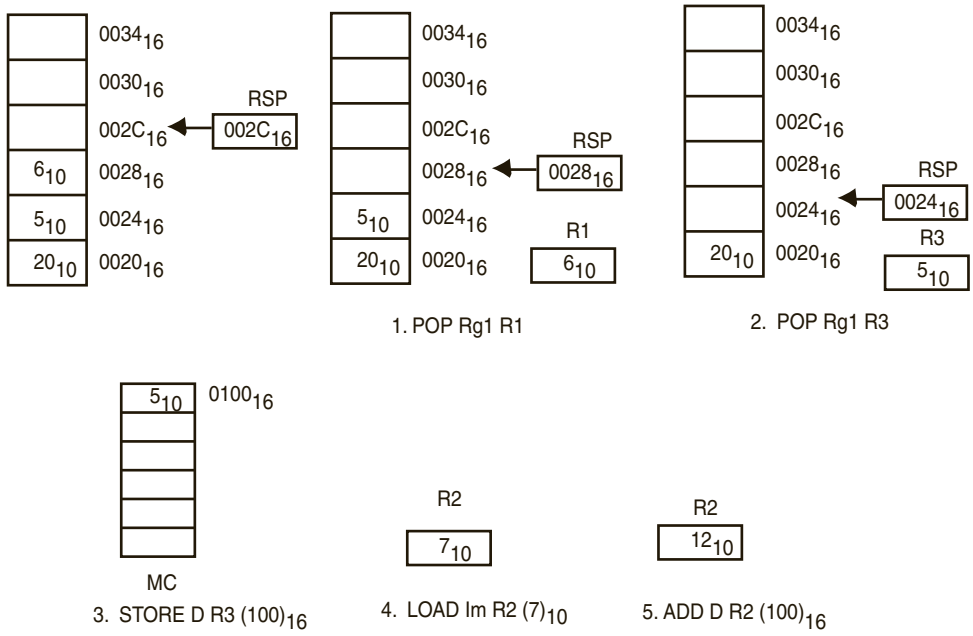


Figure 6.6 Solution de l'exercice 6.11.

```
Carry :  PUSH Rg1 R1 -- il y a carry. R1 qui contient la somme en cours
         ➥ est placé dans la pile.
         JMP Fin
Overflow :    STORE B R1 100 -- il y a overflow. R1 qui contient la somme
              ➥ en cours est écrit à l'adresse basée (RB) + 100.
              JMP fin
Finok :  STORE D R1 A -- il y a ni carry, ni overflow. R1 qui contient la somme
         ➥ des trois premiers éléments de la pile est écrit à l'adresse A.
Fin :    STOP -- arrêter le programme.
```

PARTIE 2

STRUCTURE DE L'ORDINATEUR

Cette deuxième partie s'intéresse au fonctionnement et à la structure d'un ordinateur. Elle présente les caractéristiques de chacune des trois fonctions principales d'un ordinateur à savoir :

– la fonction d'exécution qui permet l'exécution par un microprocesseur d'un programme écrit en langage machine ;
– la fonction de mémorisation qui recouvre d'une part la gestion de la mémoire centrale et d'autre part le mécanisme de hiérarchie de mémoires ;
– la fonction de communication qui permet au processeur de dialoguer avec l'extérieur via des périphériques tels que le clavier, l'imprimante, l'écran, la souris, etc.

Ainsi le chapitre 7 est consacré à la fonction d'exécution et explicite la manière dont le microprocesseur exécute une instruction machine et plus généralement un programme machine. Le chapitre 8 s'intéresse aux mémoires caches ainsi qu'à la mémoire virtuelle et il définit également les différentes technologies de mémoires existantes. Le chapitre 9 détaille la fonction de communication. Le chapitre 10 propose un ensemble d'exercices corrigés.

Mots-clés : programme machine, instruction machine, microprocesseur, mémoire RAM et ROM, mémoire cache, unité d'échange, bus de communication.

Chapitre 7

La fonction d'exécution

Les machines que nous étudions sont des machines à programme enregistré ce qui signifie que pour être exécuté le programme machine (suite d'instructions machines) doit être placé (chargé) en mémoire principale (mémoire centrale, mémoire RAM). Dans cette section nous nous proposons d'examiner comment est exécuté un programme machine. Dans un premier temps, nous décrivons les caractéristiques générales du microprocesseur et du bus lui permettant de communiquer avec la mémoire centrale. Nous étudions ensuite le principe de l'exécution d'une instruction par le microprocesseur, ainsi que les notions liées aux microcommandes et aux séquenceurs. Nous terminons en introduisant le concept très important des interruptions, puis par un aperçu des méthodes permettant une amélioration des performances des microprocesseurs.

7.1 INTRODUCTION

D'un point de vue « macroscopique », un programme machine s'exécute instruction après instruction selon l'algorithme suivant :

```
début (exécution d'une instruction)
    lecture de l'instruction;
    mise à jour du compteur ordinal;
    décodage de l'instruction;
    lecture éventuelle des opérandes;
    exécution;
    rangement des résultats;
fin (exécution de l'instruction).
```

Le langage machine définit les instructions machines exécutables par un microprocesseur et caractérise complètement une machine dans le sens où il permet de décrire un problème comme une suite d'instructions exécutables par le matériel (*hardware*). Le langage machine est donc caractéristique d'un matériel spécifique et l'expression d'un problème dans un langage machine n'est valide que pour un matériel (une machine spécifique : un programme machine n'est exécutable que sur une machine spécifique reconnaissant ce langage). Le matériel qui exécute les instructions machine est le *microprocesseur*. Il existe donc autant de langages machines qu'il y a de type de microprocesseurs. Un programme qui doit être exécuté par un microprocesseur Intel devra être écrit dans le langage machine de ce microprocesseur. Il ne sera exécutable que par lui et donc ne sera pas exécuté par un microprocesseur d'un autre constructeur (par exemple Motorola ou PowerPC).

Bien que tous les langages machines soient différents, les instructions machines répondent à la même structure :

– un champ *code opération*. Il définit la nature de l'opération à exécuter;
– un champ *opérande* qui référence le ou les opérandes sur lesquels l'opération doit s'effectuer. Ce champ peut être considéré comme étant composé de deux parties, une partie *mode d'adressage* qui définit la manière dont on va accéder à l'opérande et une information complémentaire, dont le sens est défini en fonction du mode d'adressage. Dans le cas d'un adressage dit « immédiat », l'information complémentaire représente la valeur de l'opérande. Dans le cas d'un adressage dit « direct », l'information complémentaire représente l'adresse mémoire de l'opérande.

Il existe plusieurs types d'architectures de microprocesseurs et à chacun de ces types correspond un langage machine de nature différente. On trouve ainsi :

– les langages de type registre/mémoire : les instructions arithmétiques ont un opérande situé en mémoire centrale, l'autre opérande est dans un registre de l'unité centrale (microprocesseur), enfin le résultat est placé dans un registre de l'unité centrale. Ce sont des instructions à 1,5 opérandes, le registre intervenant pour 0,5 opérande. Ce type d'instructions est celui le plus couramment rencontré dans les machines CISC (*Complex instruction set computer*). Dans un tel contexte résoudre l'équation Z = X + Y qui signifie additionner X et Y puis placer le résultat dans Z, demande 3 instructions machines :

```
load D, R, @X   (placer le contenu de l'adresse
                ➥ mémoire X dans le registre R)
add D, R, @Y    (additionner le contenu de l'adresse
                ➥ mémoire Y au contenu du registre R et
                ➥ placer le résultat dans le registre R)
store D, R, @Z  (placer le contenu du registre R
                ➥ à l'adresse mémoire Z)
```

– les langages mémoire/mémoire : l'instruction porte sur des opérandes placés en mémoire centrale, le résultat étant lui aussi placé en mémoire. Ce type d'instructions se trouve dans les machines CISC. Dans un tel contexte l'équation Z = X + Y donne lieu à une seule instruction.

```
add D, @Z, @X, @Y  (prendre le contenu de l'adresse Y,
                   ➥ lui ajouter le contenu de l'adresse X,
                   ➥ placer le résultat à l'adresse Z).
```

– les langages registre/registre : une instruction porte sur des opérandes préalablement placés dans des registres de l'unité centrale, le résultat étant lui aussi placé dans un registre. Ce type d'instruction se trouve surtout dans les machines RISC (*Reduced instruction set computer*). Dans ce contexte l'équation de notre exemple donne lieu à la séquence d'instructions machine donnée suivante :

```
load D, R1, @X     (placer le contenu de l'adresse X
                   ➥ dans le registre R1)
load D, R2, @Y     (placer le contenu de l'adresse Y
                   ➥ dans le registre R2)
add D, R3, R2, R1  (additionner le contenu de R1 au contenu de R2
                   ➥ puis placer le résultat dans R3)
store D, R3, @Z    (placer le contenu du registre R3 à l'adresse Z)
```

Dans la section comparant les architectures CISC et RISC, nous fournirons des précisions sur les raisons qui président à l'existence de ces différents types de langages et donc de microprocesseurs. Ainsi, dans les processeurs CISC (Intel par exemple) les langages sont plutôt orientés registre/mémoire alors que dans les processeurs RISC (PowerPC par exemple) les langages sont orientés registre/registre.

L'exécution d'une instruction implique :

– *le microprocesseur* (unité centrale). Il est organisé autour des registres, de l'unité arithmétique et logique (UAL), de l'unité de commande et d'un ou plusieurs bus internes permettant la communication entre ces différents modules. Le nombre et l'organisation du (des) bus de l'unité centrale sont variables et les constructeurs ont une grande liberté de conception ;
– *le bus de communication mémoire/unité centrale*. Il supporte les échanges de données et de commandes entre l'unité centrale et la mémoire centrale. Dans ce domaine les constructeurs disposent de peu de liberté quant à leurs choix architecturaux car ils doivent prendre en compte une très grande variété des matériels (prise en compte de nombreux contrôleurs et périphériques) ;
– *la mémoire centrale*. Elle contient les instructions et les données. Pour qu'une instruction soit exécutable par le microprocesseur, il faut (comme les données manipulées par l'instruction) que celle-ci soit présente en mémoire centrale.

Exécuter une instruction équivaut à permettre des interactions efficaces entre ces trois composants. Pour examiner ces interactions nous adoptons tout d'abord un point de vue externe et descriptif permettant de présenter les différents modules, leurs caractéristiques et fonctionnalités générales. Ensuite nous regardons d'un point de vue interne l'exécution des instructions machine. Pour cela, afin de mettre en évidence les principes fins de fonctionnement, nous définissons une machine arbitraire qui respecte les caractéristiques des machines de Von Neumann.

7.2 ASPECTS EXTERNES

Nous présentons ici les caractéristiques générales des microprocesseurs et des bus de communication entre mémoire et microprocesseur. Les aspects relatifs à la mémoire sont abordés dans le chapitre 8, concernant la fonction de mémorisation.

7.2.1 Le microprocesseur

Le microprocesseur communique avec les autres modules de la machine par le biais de signaux électriques. Il est placé dans un boîtier qui dispose de plusieurs broches pour le transport des signaux (figure 7.1).

Bien que chaque microprocesseur possède un jeu spécifique de signaux, on peut, sans être cependant exhaustif, classer ceux-ci en grandes catégories qui se retrouvent sur tout type de microprocesseur :

– alimentation électrique ;
– horloge ;
– bus d'adresses (A0-A15), de données (D0-D7), de commandes (READ, WRITE) ;
– interruptions (INTR, INTA, NMI) ;
– sélection et synchronisation mémoire ou périphériques (READY, mémoire, périphérique, PRÊT) ;
– gestion du bus de communication (demande de bus, ACK bus).

Les signaux sont soit des signaux d'entrées, soit des signaux de sorties, ou encore des signaux d'entrées-sorties. Les flèches sur la figure 7.1 indiquent le sens des différents signaux.

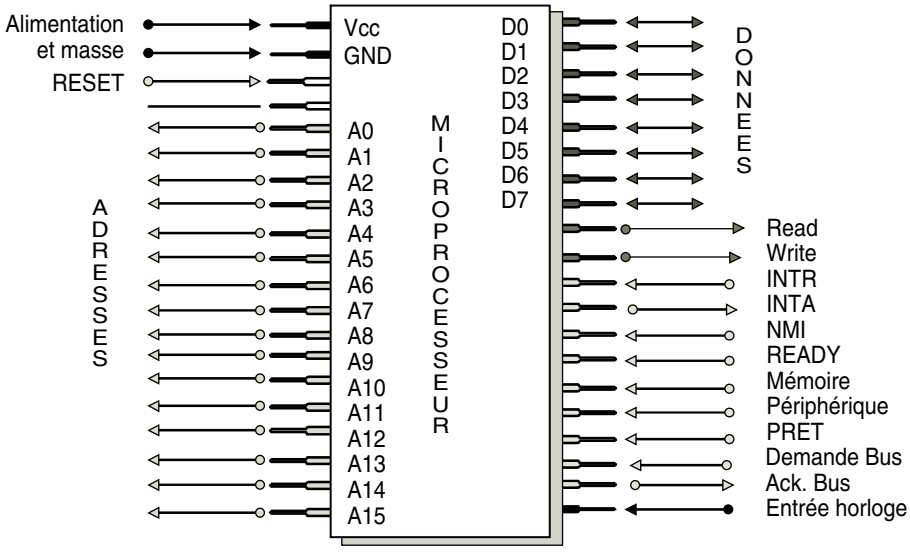


Figure 7.1 Le microprocesseur.

Les signaux d'adresses sont des signaux de sorties pour le microprocesseur et des signaux d'entrées pour les boîtiers mémoire. Les signaux de données sont bidirectionnels, ils transportent les informations de la mémoire vers le microprocesseur ou au contraire du microprocesseur vers la mémoire. Les signaux de commande sont des signaux de sorties pour piloter les boîtiers mémoire. Les interruptions sont des signaux d'entrées provenant en général des périphériques. Ils permettent d'interrompre le fonctionnement du microprocesseur pour prendre en charge l'interruption. Il existe d'autres types d'interruptions que celles produites par les périphériques, par exemple des interruptions logicielles, permettant de prendre en compte des dysfonctionnements de l'ordinateur (une division par zéro, un dépassement de capacité dans une opération arithmétique…). Les signaux concernant le bus permettent, en particulier, au processeur de synchroniser le partage du bus entre les différents modules matériels qui peuvent en avoir besoin simultanément.

L'horloge est un signal très important, caractérisé par sa fréquence mesurée en mégahertz (1 MHz = 10^6 hertz), qui rythme le fonctionnement du microprocesseur et définit ainsi le cycle du microprocesseur. En effet, le temps de cycle du microprocesseur s'exprime comme l'inverse de la fréquence de l'horloge et est donné en nanoseconde (ns). Ainsi pour une fréquence d'horloge égale à 1 gigahertz, le temps de cycle processeur équivaut à 1/1 GHz = $1/10^3 \times 10^6 = 10^{-9}$ seconde = 1 nanoseconde, sachant que 1 GHz est égal à 10^3 MHz.

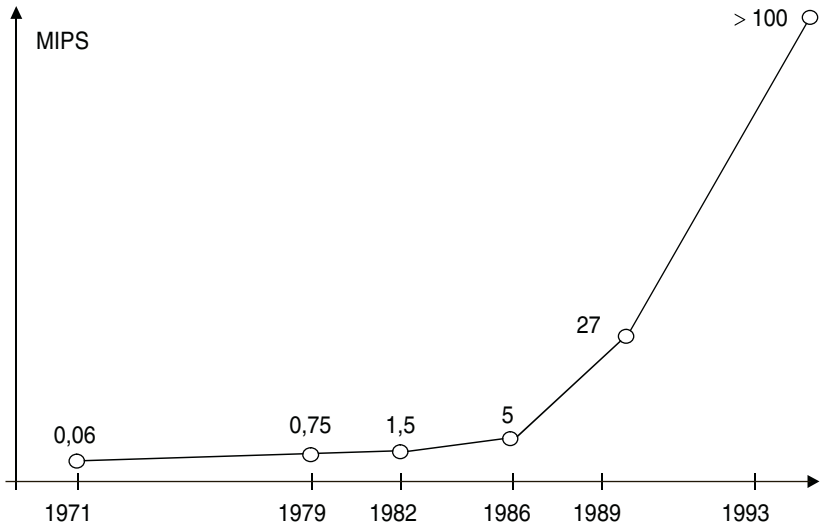


Figure 7.2 Évolution des performances des microprocesseurs.

La fréquence d'horloge des processeurs est en pleine évolution (figure 7.2) et de nos jours, on trouve fréquemment des processeurs munis d'une horloge dont la fréquence est égale à 1 gigahertz. Un tel processeur a donc un temps de cycle égal à 1 nanoseconde.

7.2.2 Les bus

Comme le schéma général de la figure 7.3 le montre notre ordinateur est d'une part organisé autour d'un bus interne au microprocesseur, d'autre part autour d'un bus externe au microprocesseur servant pour la communication entre la mémoire et le microprocesseur. Le bus interne est unique dans notre cas.

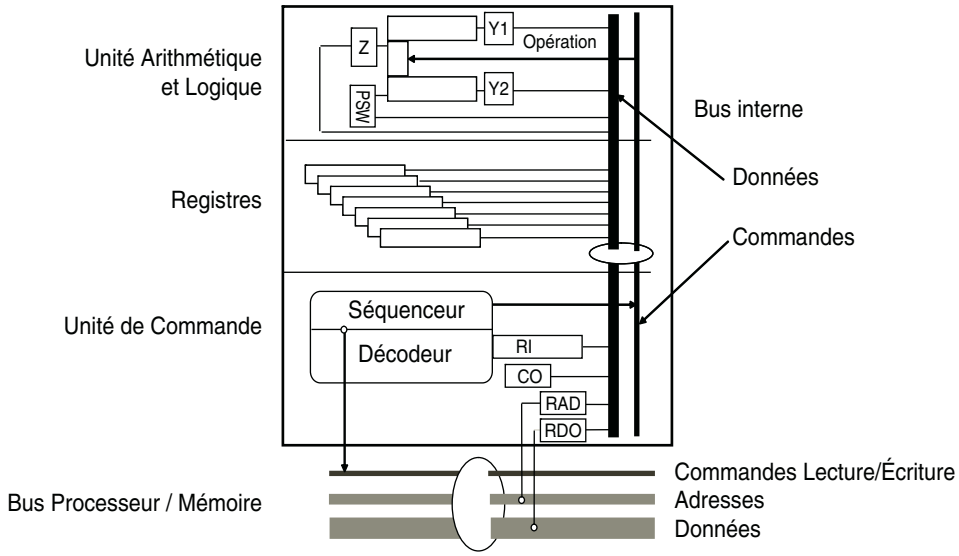


Figure 7.3 Schéma général du processeur.

La conception et l'organisation du microprocesseur, en particulier du, ou des bus internes, laisse une grande liberté au concepteur. Par contre le bus externe est beaucoup plus contraint du fait de la grande variété des modules matériels à interconnecter.

Ce type de bus s'appuie sur des caractéristiques importantes telles que : la largeur du bus, le cadencement du bus, l'arbitrage pour l'accès au bus, les modes de fonctionnement du bus.

Dans cette section nous étudions les modalités d'exécution des instructions machine. Ces modalités concernent le fonctionnement et l'organisation du bus interne et les aspects largeur et cadencement du bus externe. Les aspects arbitrage et modes de fonctionnement seront abordés respectivement, dans le chapitre 9 traitant des entrées-sorties et dans le chapitre 8 traitant de la gestion des caches mémoire.

Largeur du bus

La *largeur* du bus définit le nombre de lignes du bus c'est-à-dire le nombre de bits que le bus peut véhiculer en parallèle. Cette largeur de bus constitue une donnée technique très importante et plus particulièrement la largeur du bus d'adresses. En effet, plus le bus d'adresses est large plus l'espace d'adressage du processeur est

grand. Ainsi n lignes d'adresses permettent d'adresser 2^n mots mémoires. Augmenter la largeur du bus d'adresses équivaut à augmenter le nombre de broches du boîtier et donc la surface de microprocesseur.

La taille du bus de données croît également afin de satisfaire l'augmentation des performances des microprocesseurs. En effet, si la mémoire dispose de mots de 32 bits et que le bus de données a une largeur de 8 bits, il faut 4 accès mémoires pour que le microprocesseur dispose d'un mot mémoire.

L'augmentation de la largeur du bus de données n'est pas le seul moyen d'augmenter les performances du bus, car on peut également augmenter la fréquence d'horloge du bus.

Les microprocesseurs évoluent très vite et il est difficile de faire varier la largeur des bus pour « suivre » les processeurs. Une manière d'opérer consiste à décider d'une largeur de bus (par exemple 32 bits) et d'utiliser ce bus, selon le cas, comme un bus de données ou un bus d'adresses : c'est *le multiplexage temporel* du bus. Cette technique diminue le nombre de broches donc le coût mais aussi les performances.

Cadencement du bus

On trouve deux types de bus : les *bus synchrones* et les *bus asynchrones*.

➤ Les bus synchrones

Ils disposent d'une horloge propre fonctionnant à une fréquence qui caractérise le *cycle du bus*. La fréquence d'horloge des bus varie entre 8 MHz à 500 MHz (pour les plus récents), ce qui correspond à un temps de cycle allant de 125 ns à 2 ns.

Avec ce type de bus, l'opération de lecture d'un mot mémoire par le microprocesseur se déroule de la manière suivante :

– le microprocesseur dépose l'adresse du mot sur le bus d'adresses ;

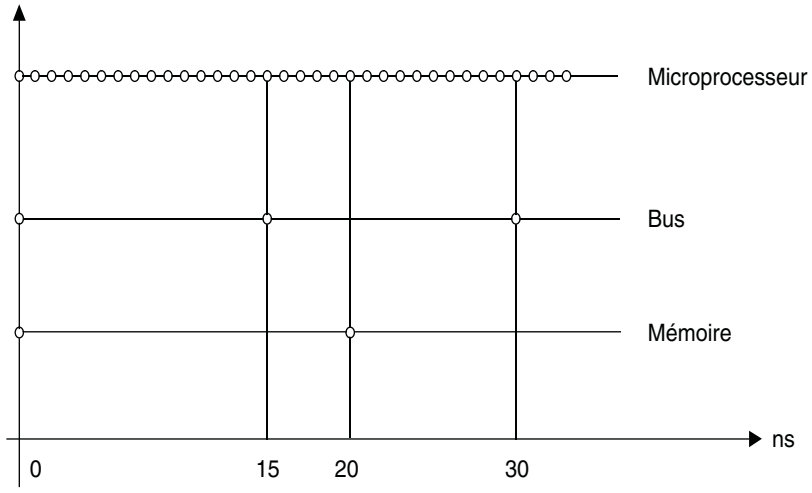


Figure 7.4 Bus synchrone.

– le microprocesseur dépose un signal de lecture sur le bus de commandes ;
– le microprocesseur attend que la mémoire dépose un signal de donnée prête ;
– le microprocesseur récupère la donnée sur le bus de données.

Prenons l'exemple de la figure 7.4 avec les données suivantes :

– la fréquence du processeur est égale à 1 GHz soit un temps de cycle égal à 1 ns ;
– la fréquence du bus est égale à 66 MHz soit un temps de cycle égal à 15 ns ;
– le temps d'accès à la mémoire centrale (RAM) est égal à 20 ns.

Dans cet exemple le microprocesseur lance une commande de lecture au top d'horloge 0. On suppose que cette demande est instantanément délivrée au bus et à la mémoire de manière à aligner les horloges. La mémoire ne répond que 20 ns plus tard ; il faut donc 2 cycles de bus pour que le microprocesseur puisse obtenir la donnée sur le bus de données.

Compte tenu des différences entre les cycles de mémoire, bus et microprocesseur on considère que ce dernier est toujours à même de se synchroniser avec le bus. Avec ce type de bus le processeur sollicite le bus et la mémoire qui répondent à leurs rythmes sans concertation ni synchronisation réelles. Le nombre de cycles du bus est toujours un nombre entier ce qui n'est pas forcément optimum. Ce type de bus ne facilite pas les évolutions en fonction des modifications technologiques. Une amélioration du cycle de la mémoire (diminution du temps d'accès) ne se traduira pas forcément par une amélioration des échanges puisqu'elle n'entraînera pas nécessairement la diminution du nombre de cycle du bus. Pour être synchrone avec la mémoire, le bus est généralement obligé d'ajouter des cycles d'attente (*Wait state*).

➤ Les bus asynchrones

Ce type de bus ne s'appuie pas sur une horloge associée à ce bus. Le fonctionnement du bus utilise des signaux permettant aux différents modules de se coordonner lors d'un échange et de se mettre ainsi d'accord pour dire que l'échange a eu lieu. L'enchaînement des opérations n'est pas lié à un signal d'horloge mais à un enchaînement de signaux propres à l'échange. Ce type de bus est bien supérieur mais beaucoup plus difficile à construire.

En général les bus utilisés sont de type synchrone à cause de leur facilité de construction et donc de leur coût moindre. Pour obtenir un fonctionnement correct il faut choisir avec soin les modules afin que leurs caractéristiques soient compatibles.

7.3 ASPECTS INTERNES

Dans le chapitre 1 concernant la structure générale et le fonctionnement d'un ordinateur, nous avons vu que le rôle du microprocesseur consiste à exécuter le programme machine placé dans la mémoire principale. Cette exécution s'effectue instruction après instruction de la première jusqu'à l'instruction STOP, dernière instruction du programme. Ce séquencement de l'exécution des instructions constitue ce que l'on

appelle le *flux d'exécution du programme machine*. Ce flux d'exécution du programme machine peut être décrit par l'algorithme suivant :

```
début
charger la première instruction (FETCH)
tant que ce n'est pas STOP
faire
    modifier le compteur ordinal (incrémentation);
    décoder l'instruction;
    charger les données éventuelles dans les registres;
    exécuter les micro-instructions;
    charger l'instruction pointée par le compteur ordinal :
    ➥ c'est l'instruction suivante (FETCH);
finfaire
fin
```

Le programme machine est placé dans la mémoire centrale par le chargeur qui à la fin du chargement initialise le compteur ordinal CO avec l'adresse de la première instruction du programme. Le programme démarre donc par le chargement dans le registre instruction RI de cette première instruction. Ensuite le microprocesseur va réaliser le flux d'exécution du programme machine. L'algorithme ci-dessus rend compte de ce flux en utilisant un bloc « `tant que` » qui exprime que tant que le code opération de l'instruction actuellement dans le registre instruction n'est pas STOP, les instructions du bloc « `faire/finfaire` » sont exécutées. Quand le code opération est STOP le bloc « `faire/finfaire` » est sauté et l'instruction `fin` exécutée : le programme est alors terminé.

Sans trop entrer dans les détails, nous avons indiqué au chapitre 1 que l'unité de commande est chargée de l'exécution des différentes phases de réalisation des instructions. Le séquenceur, rythmé par l'horloge, exécute une séquence de microcommandes qui rend compte de l'instruction machine traitée.

Nous allons maintenant examiner plus en détail comment, pour un processeur déterminé, s'exécute une instruction machine sur le matériel (*hardware*) de cette machine.

7.3.1 Exécution d'une instruction machine

Pour étudier comment est exécutée une instruction nous définissons une machine arbitraire qui respecte les fonctionnalités principales d'une machine réelle (machine de Von Neumann) mais qui ne correspond à aucune machine du marché. Elle sert de guide pour illustrer les fonctions importantes d'un ordinateur. La figure 7.5 définit l'organisation matérielle et le programme machine que nous voulons exécuter.

Ce processeur est très proche de celui décrit dans le chapitre 1 de cette partie de l'ouvrage, *Structure générale et fonctionnement*. Il s'agit là encore d'un microprocesseur à bus interne unique (certains microprocesseurs disposent de plusieurs bus internes afin d'améliorer les performances en parallélisant des opérations). L'unité arithmétique et logique est plus simple : un seul registre d'entrée (Y) reçoit une donnée, la seconde entrée de l'UAL étant directement reliée au bus interne de données.

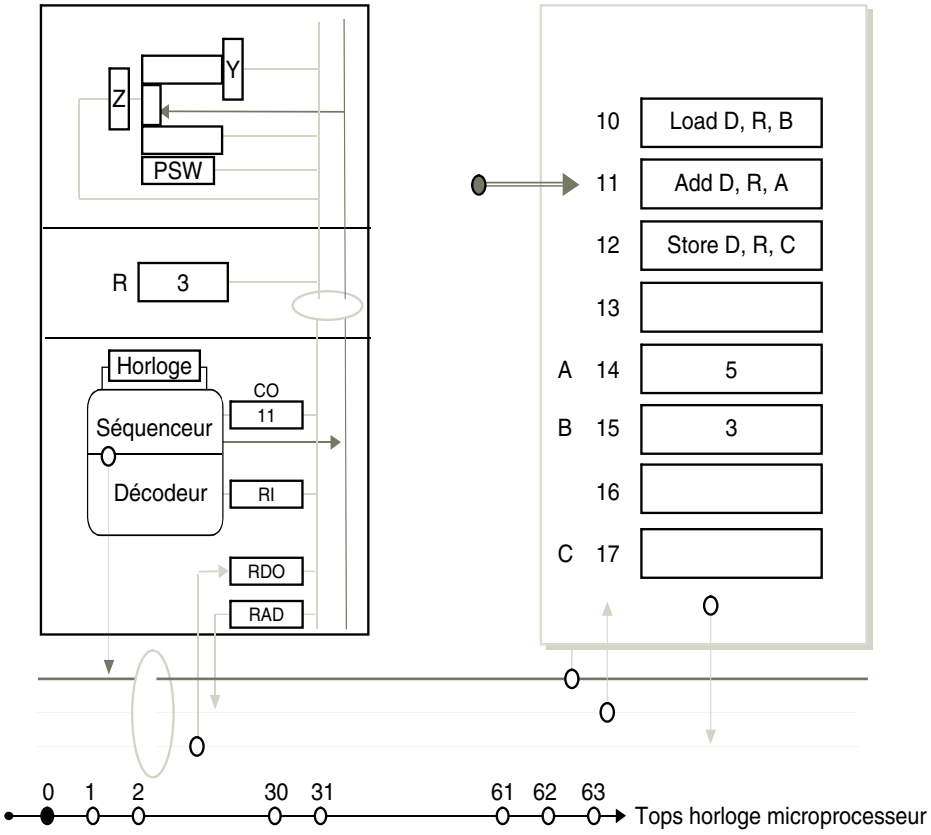


Figure 7.5 Machine et programme de l'exemple.

L'ensemble des registres est (pour des raisons de simplicité graphique) réduit à un seul registre (R).

Notre machine a les caractéristiques suivantes :

– la fréquence du processeur est égale à 1 GHz, soit un cycle égal à 1 ns ;
– la fréquence du bus est égale à 66 MHz, soit un cycle égal à 15 ns ;
– le temps d'accès à la mémoire centrale est égal à 20 ns.

Le bus de communication est de type synchrone. Ainsi lorsque le microprocesseur fait une demande de lecture mémoire il obtiendra sa donnée 30 nanosecondes plus tard (après deux cycles du bus).

Les instructions de notre machine sont de type registre/mémoire : l'un des opérandes est en mémoire, l'autre est dans un registre. Avec ce type d'instructions notre problème, C = A + B s'exprime sous la forme de la séquence d'instructions :

```
load D, R, @B                          Charger le contenu du mot d'adresse B
                                       ➥ dans le registre R. (R) = (B)
add D, R, @A        (R) = (R) + (A)    Contenu de R + contenu de A dans R
store D, R, @C      (C) = (R)          Placer le contenu de R à l'adresse C
```

Le programme machine placé en mémoire centrale est décrit sous la forme d'un programme écrit en langage d'assemblage. Ici, par souci de simplicité d'écriture, les adresses mémoire sont exprimées en base 10. La première instruction est à l'adresse 10, la seconde à l'adresse 11, la troisième à l'adresse 12. La première instruction indique que l'on place le contenu de l'adresse B (adresse 15) dans le registre R. La seconde instruction réalise l'addition du contenu de R avec le contenu de l'adresse A (adresse 14), le résultat étant placé dans R. Enfin, la troisième instruction place le contenu de R à l'adresse mémoire C (adresse 17). Ce programme réalise donc l'addition des valeurs 3 et 5, le résultat étant placé dans C (adresse 17).

Notre programme est en cours d'exécution, la première instruction ayant été exécutée, le registre R contient 3 et le compteur ordinal CO contient 11 (adresse de l'instruction à exécuter). Nous allons, dans ce contexte, détailler le processus d'exécution de l'instruction machine : `add D, R, A` et examiner les différentes phases de son exécution.

FETCH

C'est la phase de recherche et de chargement de l'instruction, pointée par le compteur ordinal CO, dans le registre instruction RI. Cette phase comporte trois étapes.

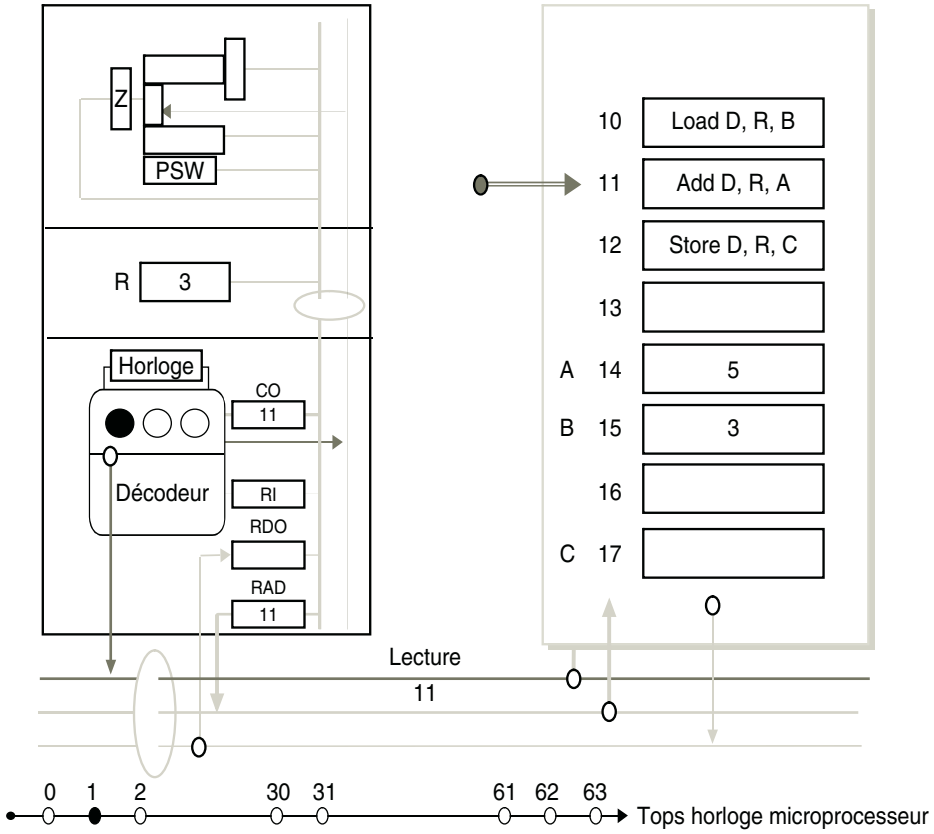


Figure 7.6 Étape 1.

➤ Étape 1

Le contenu du compteur ordinal CO est placé dans le registre d'adresses RAD, donc à l'entrée de la mémoire centrale via le bus d'adresses. Une commande de lecture est placée sur le bus de commandes et le signal AttenteMémoire est positionné. Lorsque la donnée est disponible la mémoire place un signal PRÊT qui indique au microprocesseur que la donnée demandée est disponible sur le bus de données donc dans le registre de données RDO. Compte tenu des cycles respectifs de la mémoire et du bus ce dernier place un état d'attente (WaitState) et la donnée est donc disponible 30 nanosecondes après la demande (figure 7.6).

➤ Étape 2

À cette étape le compteur ordinal CO est incrémenté. Pour cela on suppose que notre machine est dotée d'un compteur ordinal piloté par un signal, « InCo », qui déclenche l'incrémentation (ajoute 1 à CO). À la fin de cette étape, le compteur ordinal contient l'adresse de la prochaine instruction à exécuter (figure 7.7).

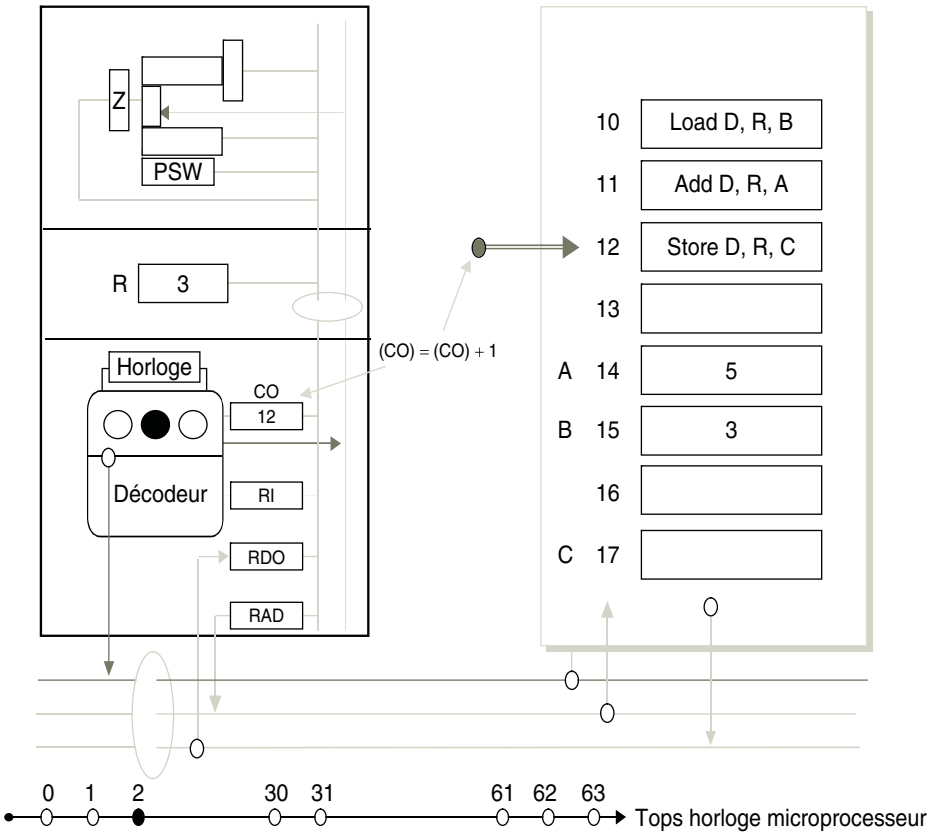


Figure 7.7 Étape 2.

➤ Étape 3

Cette étape se produit 30 nanosecondes après la demande de lecture faite par le microprocesseur. Elle consiste à placer le contenu du registre RDO dans le registre instruction RI. À la fin de cette étape le registre d'instruction RI contient l'instruction `Add D, R, A`. La phase de recherche et de chargement (FETCH) de l'instruction dans le registre d'instruction RI est alors terminée. La phase de décodage commence (figure 7.8).

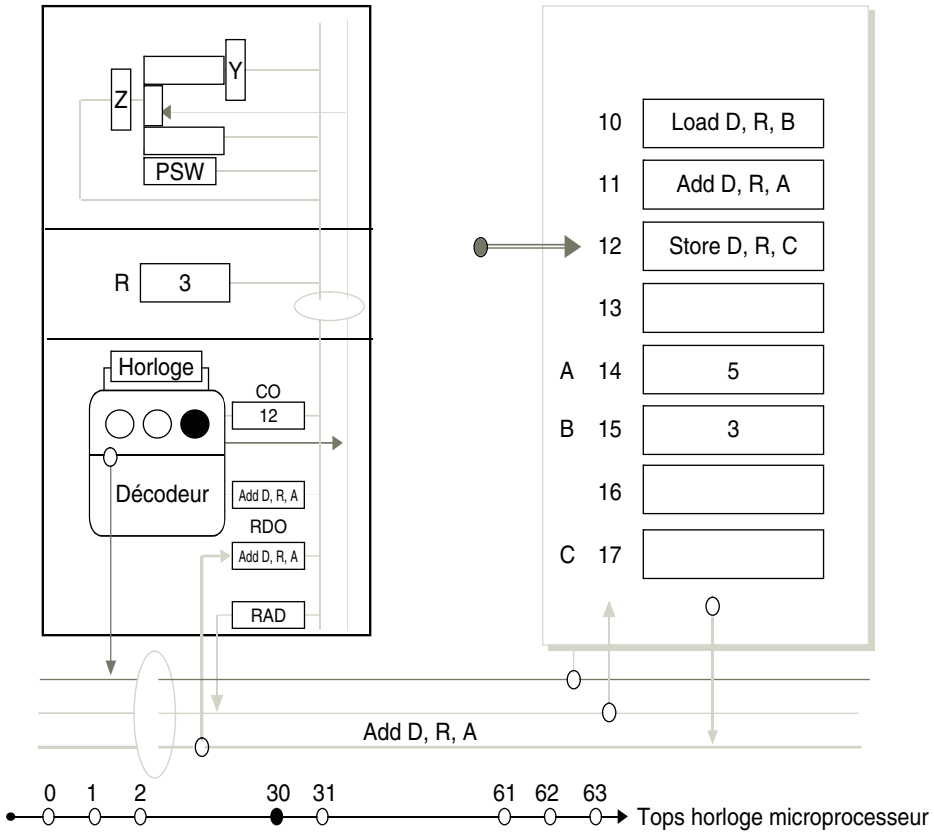


Figure 7.8 Étape 3.

Décodage

La phase de décodage permet l'interprétation du code opération et du mode d'adressage de l'opérande. Ici il s'agit d'une addition du contenu de l'adresse mémoire A avec le registre R, le résultat devant être placé dans R. Cette phase va permettre d'acquérir le contenu de A qui sera placé dans le registre Y. La phase d'exécution pourra alors être réalisée.

➤ Étape 4

Le champ opérande, A, est placé dans le registre d'adresses RAD puis une commande de lecture est positionnée sur le bus de commandes par l'unité de commande. Le signal AttenteMémoire est également placé et le microprocesseur est prévenu que la donnée est prête lorsque la mémoire positionne le signal PRÊT (figure 7.9).

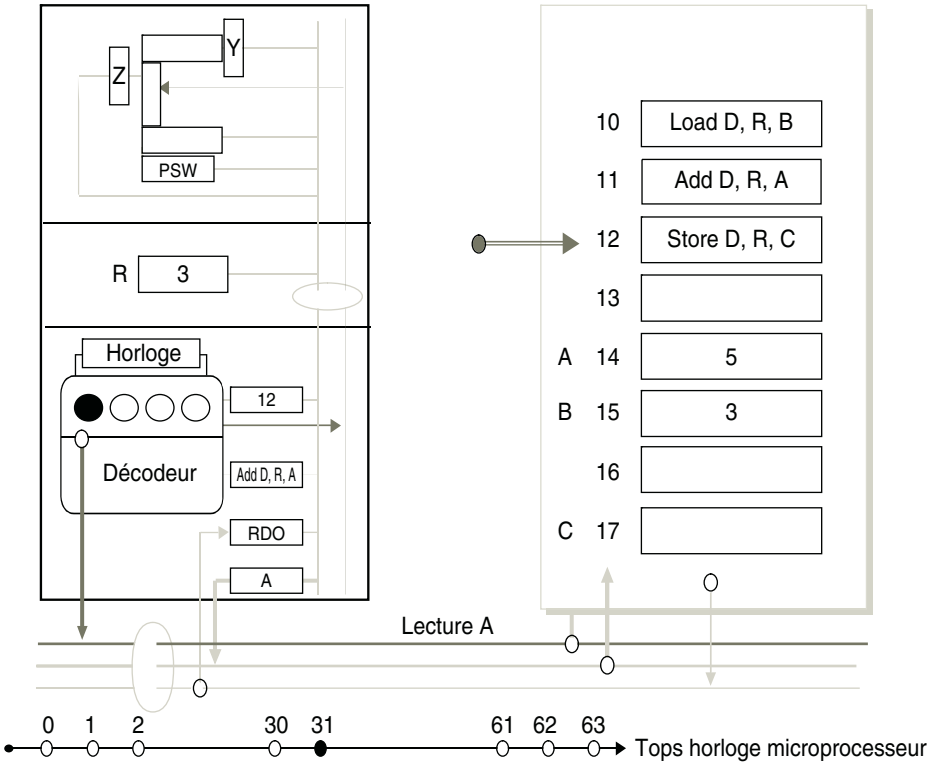


Figure 7.9 Étape 4.

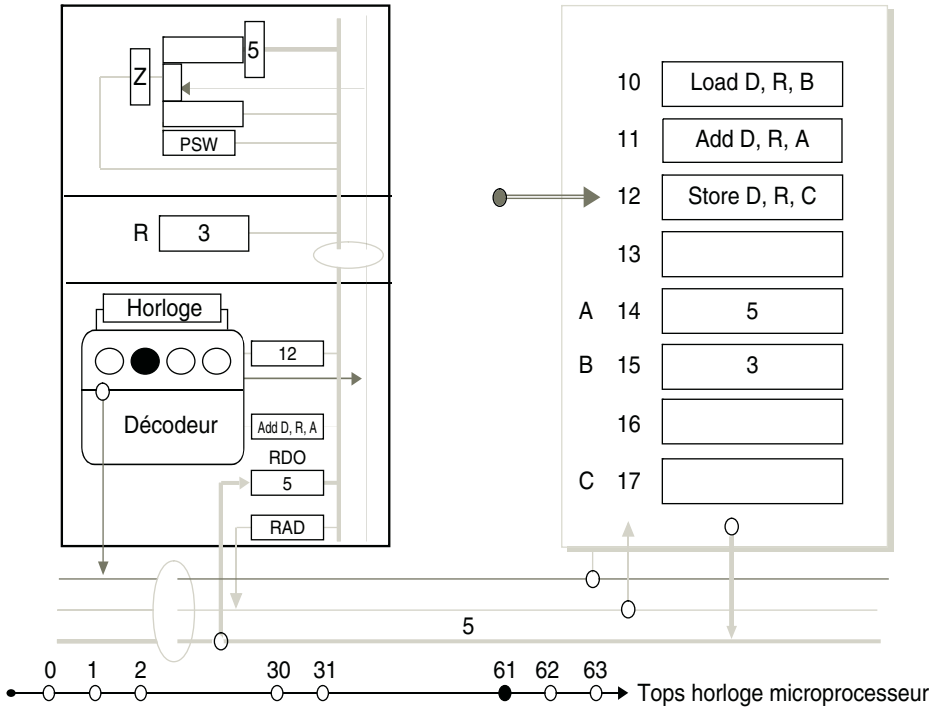
➤ Étape 5

La donnée demandée est disponible dans le registre RDO. Elle est placée dans le registre d'entrée Y de l'unité arithmétique et logique. La phase de décodage est terminée, l'exécution peut commencer (figure 7.10).

Exécution

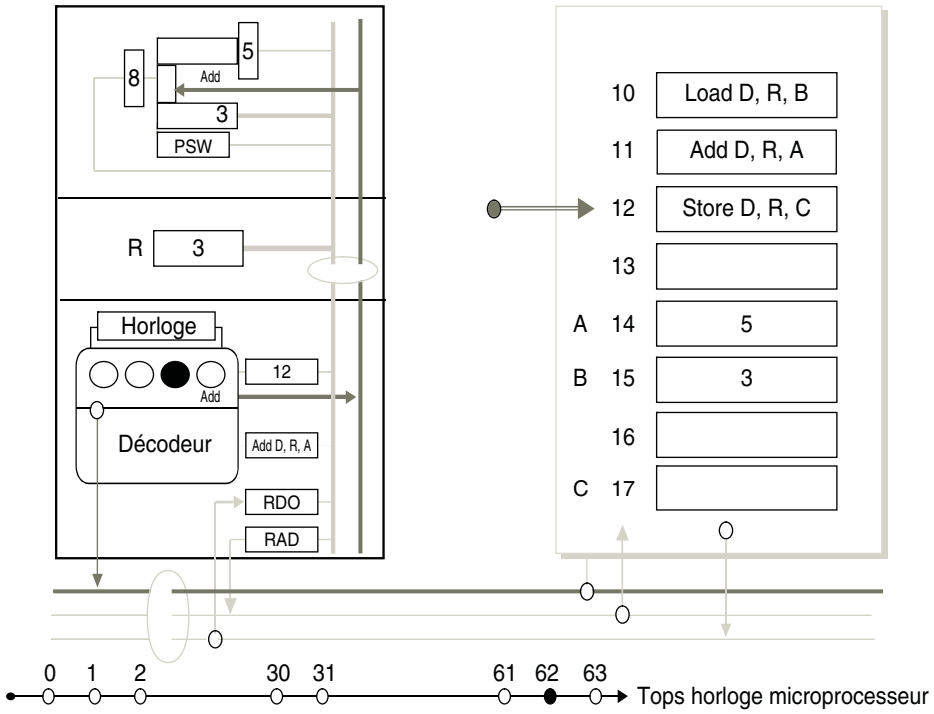
➤ Étape 6

L'exécution de l'addition s'effectue en plaçant le contenu du registre R sur le bus interne de données, donc en entrée de l'UAL qui reçoit ainsi sa deuxième donnée. Une commande d'addition est déclenchée pour l'UAL. Le résultat se trouve dans le registre Z (figure 7.11).



↑ **Figure 7.10** Étape 5.

Figure 7.11 Étape 6. ↓



➤ Étape 7

Pour conclure, il suffit maintenant de placer le contenu du registre Z dans le registre R. On a alors bien réalisé l'opération souhaitée (figure 7.12).

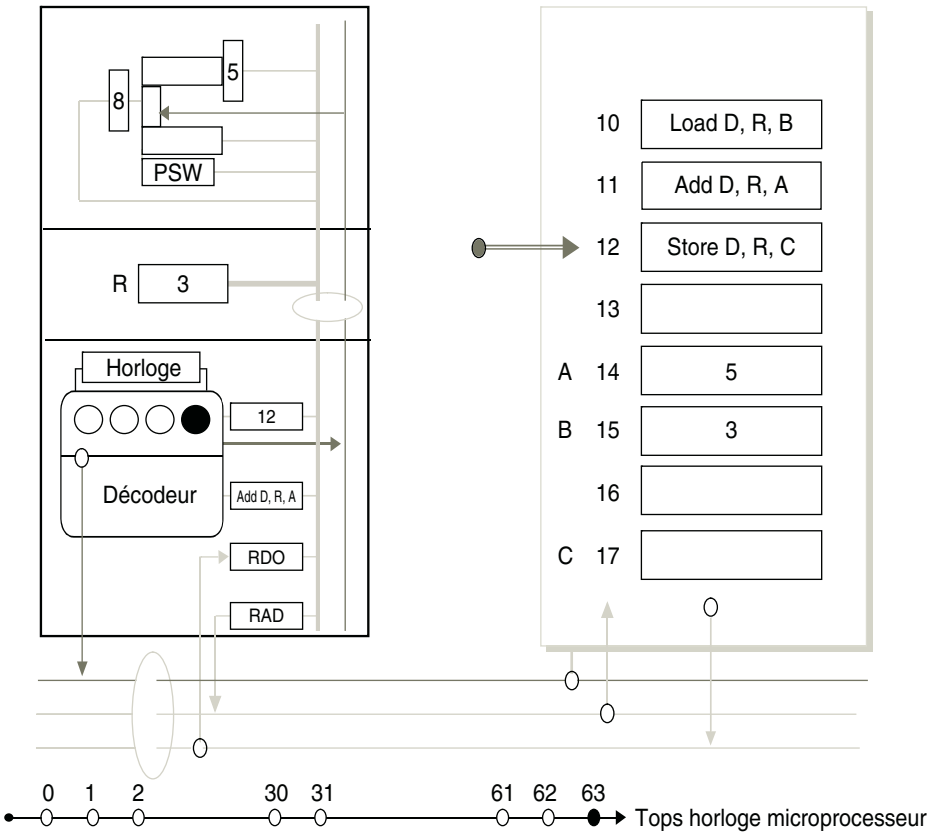


Figure 7.12 Étape 7.

Conclusion

Cet exemple met en évidence les différentes étapes de l'exécution d'une instruction ainsi que le sens qu'il faut donner au mot exécution. L'ensemble registres/UAL définit le « chemin de données ». Ce chemin de données est caractéristique du microprocesseur, nous verrons que les processeurs CISC et RISC n'ont pas le même chemin de données. Enfin le cycle du chemin de données caractérise la vitesse d'exécution d'une instruction. Pour réaliser les différentes étapes (fetch, décodage, exécution), l'unité de commande constituée du séquenceur et du décodeur, place des signaux qui permettent le séquencement des différentes étapes. Le décodeur reconnaît l'instruction et permet au séquenceur de positionner, pour chacune des étapes, les signaux nécessaires à l'exécution de l'instruction décodée.

7.3.2 Microcommandes et micro-instructions

Les différents modules de l'unité centrale sont pilotés par les signaux que le séquenceur positionne au bon moment.

Dans cette partie nous examinons les signaux ou *microcommandes* du séquenceur pour piloter les registres et l'UAL, puis la production des *micro-instructions* (ensemble de microcommandes) permettant de réaliser les différentes étapes de l'exécution d'une instruction de notre programme sur notre processeur.

Les registres

Les registres sont des zones de mémoires permettant le stockage d'informations que peut manipuler directement le microprocesseur. La figure 7.13 nous donne l'organisation matérielle d'un registre.

Le registre considéré ici contient 8 digits binaires. Il est relié au bus interne de données du microprocesseur par le biais de barrières d'entrée et de sortie. Ces barrières sont pilotées par des signaux d'entrée et de sortie positionnés par le séquenceur. Ces signaux ouvrent ou ferment les barrières permettant à l'information de circuler entre le bus et le registre. Le mode de fonctionnement du registre est indiqué dans la figure 7.14.

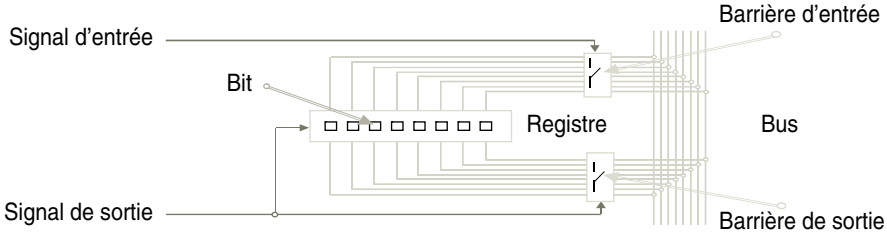


Figure 7.13 Registre.

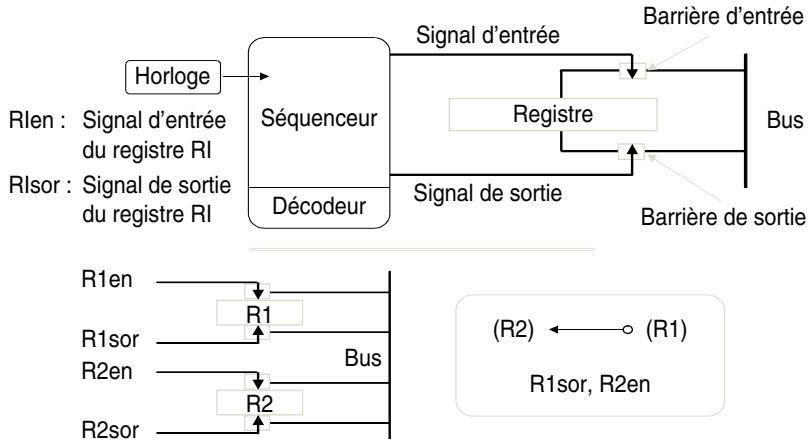


Figure 7.14 Principe de fonctionnement d'un registre.

Au niveau de la notation, si un registre porte le nom R, alors le signal d'entrée pilotant la barrière d'entrée de ce registre est nommé Ren et le signal de sortie Rsor. Le contenu du registre R sera noté (R). Ainsi pour placer le contenu du registre R1 dans le registre R2, les signaux R1sor et R2en sont positionnés. Nous réalisons alors (R2) = (R1).

L'Unité arithmétique et logique (UAL)

Cette unité comprend l'ensemble des circuits réalisant les opérations arithmétiques et logiques d'un microprocesseur. La figure 7.15 nous en rappelle l'architecture fonctionnelle en précisant quelques signaux permettant son pilotage.

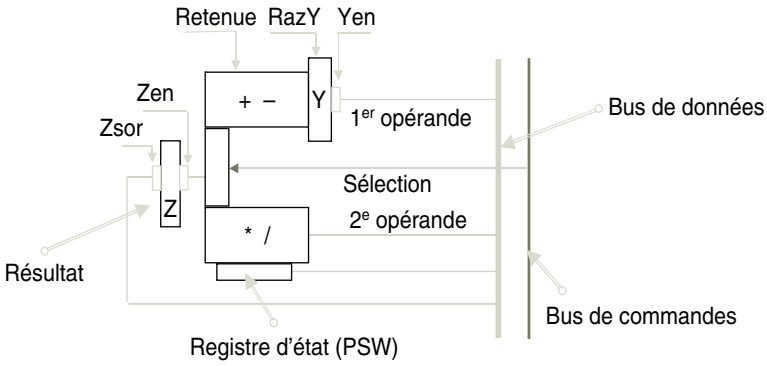


Figure 7.15 Unité arithmétique et logique.

Le registre Y est piloté par le signal d'entrée Yen et l'on suppose qu'il est directement relié aux circuits de calcul de l'UAL. Le registre de sortie Z est piloté par deux signaux d'entrée et de sortie, Zen et Zsor. Par ailleurs le signal RazY permet de réinitialiser à la valeur 0 le contenu de Y. Enfin, le signal de retenue Retenue permet d'introduire une retenue dans l'addition des bits de poids faible.

À titre indicatif la figure 7.16 donne un schéma de réalisation d'une UAL 1 bit, c'est-à-dire pour laquelle les opérandes représentent des valeurs codées sur 1 bit. On y trouve des circuits ET, des circuits OU, des inverseurs, des OU exclusif, etc.

Les opérandes sont notés A et B. Cette UAL permet de réaliser les quatre opérations ET, OU, Inverse, Addition. Pour sélectionner une opération on dispose de deux bits d'entrées (F0, F1) permettant 2^2 combinaisons. La valeur du couple (F0, F1) détermine le circuit arithmétique qui va opérer sur les entrées (A, B). Par exemple si (F0, F1) vaut (0, 0) un ET est réalisé sur (A, B), le résultat étant placé sur S. Par contre si (F0, F1) vaut (1, 1), alors une addition avec retenue de sortie est réalisée, le résultat étant placé sur S. Ainsi si (F0, F1) vaut (1, 1) et (A, B) vaut également (1, 1) alors S vaut 0 et la retenue de sortie vaut 1 également.

La figure 7.17 présente le schéma d'une UAL 8 bits. Celle-ci est constituée par 8 UAL 1 bit, chaînées entre elles par la propagation de la retenue du circuit additionneur.

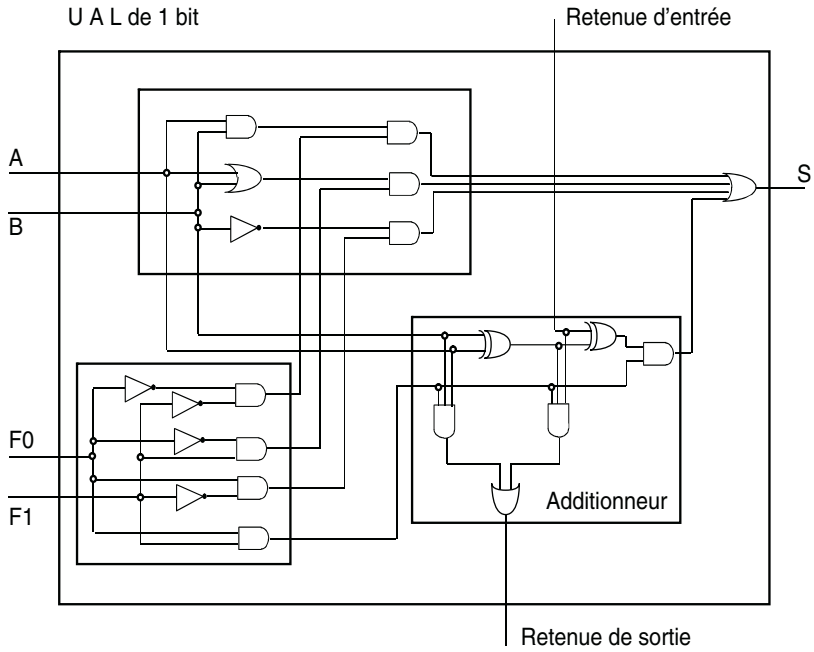


Figure 7.16 Unité arithmétique et logique de 1 bit.

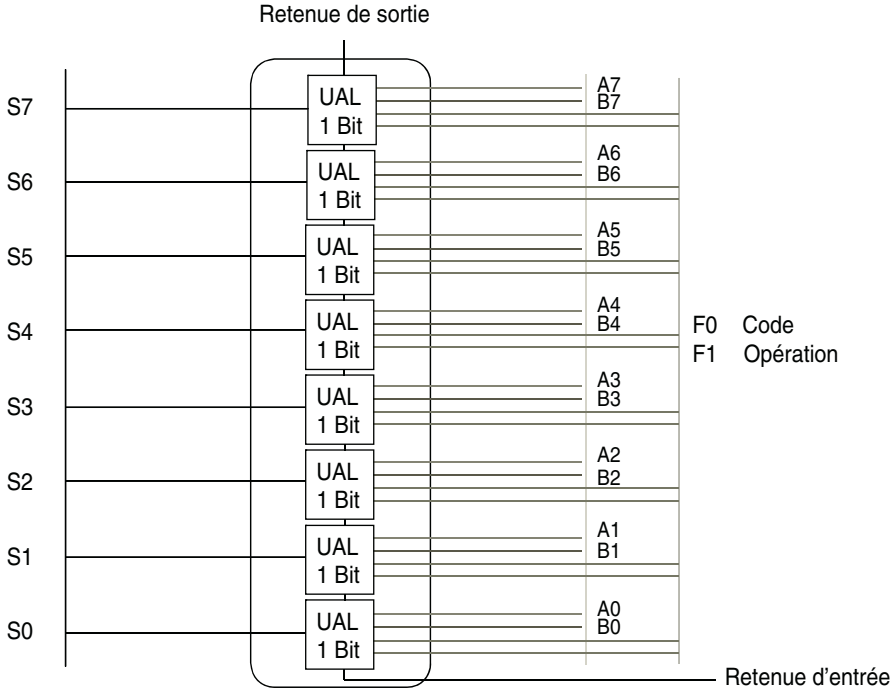


Figure 7.17 Unité arithmétique et logique de 8 bits.

Les micro-instructions

À chaque étape de l'exécution d'une instruction l'unité de commande positionne des signaux. L'ensemble des microcommandes positionnées à une étape constitue la *micro-instruction* qui sera exécutée à cette étape. Nous avons vu que l'exécution de l'instruction `Add D, R, A` demandait 7 étapes. La figure 7.18 résume les différentes étapes et les signaux positionnés à chacune des étapes.

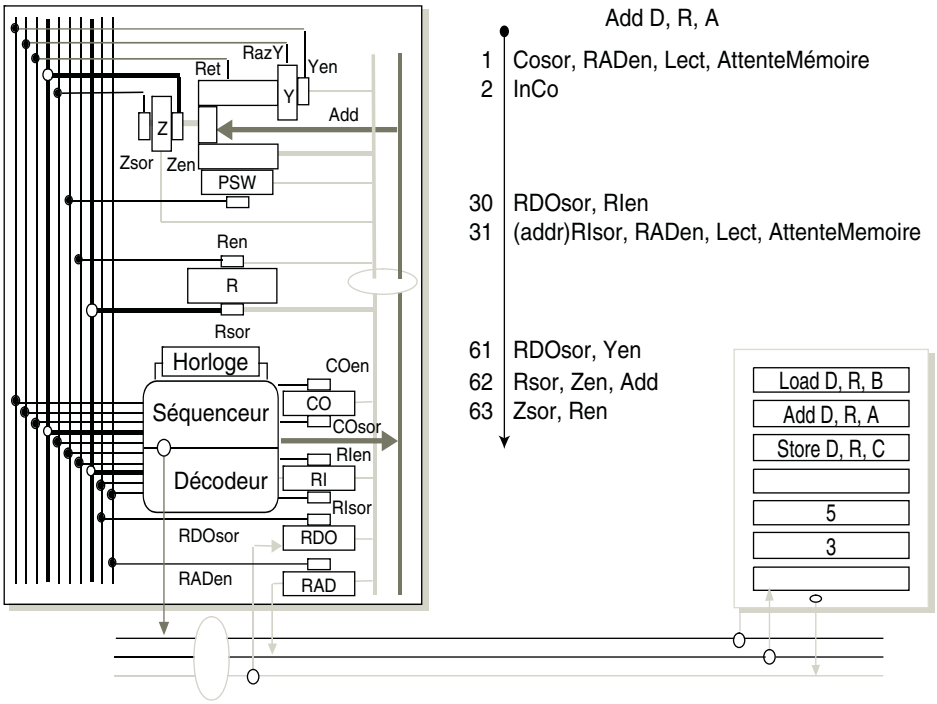


Figure 7.18 Microcommandes pour l'exécution de `Add D, R, A`.

Au top d'horloge 1 l'étape 1 est réalisée par le positionnement des signaux indiqués. Au deuxième top le compteur ordinal est incrémenté, au 62e top (étape 6) on exécute :

$$(Z) = (Y) + (R)$$

La figure 7.18 présente également une esquisse du « câblage » interne du microprocesseur destinée uniquement à sensibiliser à la complexité de réalisation d'un tel microprocesseur. Les différentes étapes de l'exécution sont cette fois exprimées sous la forme d'une séquence de microcommandes. Chaque micro-instruction est définie comme l'ensemble des signaux que le séquenceur positionne. Par exemple, la micro-instruction `COsor`, `RADen`, `Lect`, `AttenteMémoire`, déclenche les opérations :

– (RAD) = (CO) ;
– Lect : le séquenceur demande la lecture du mot mémoire dont l'adresse est dans RAD ;
– AttenteMémoire est actif.

De cet exemple particulier nous pouvons tirer quelques remarques générales :

– l'architecture matérielle détermine la nature et le nombre des microcommandes nécessaires à l'exécution d'une instruction. Nous avons ici un microprocesseur avec un seul bus interne ; nous verrons dans la section traitant des machines CISC et RISC que l'on peut avoir plusieurs bus internes ce qui change les modalités d'exécution d'une instruction ;
– quelle que soit la nature de l'instruction à exécuter, la phase de FETCH est toujours identique pour ce microprocesseur ;
– le décodage et l'exécution de l'instruction dépendent de la nature de cette instruction.

Prenons comme autre exemple l'instruction de chargement du registre R : `Load D, R, B`. À titre d'exercice on peut vérifier que la séquence de micro-instructions est :

```
1. COsor, RADen, Lect, AttenteMémoire
2. InCo
3. RDOsor, RIen
4. (Adresse)RIsor, RADen, Lect, AttenteMémoire
5. RDOsor, Ren
```

Cette séquence met en évidence que le nombre et la nature des micro-instructions ne sont pas les mêmes que dans le cas de l'addition. Nous avons toujours les trois mêmes micro-instructions pour la phase de FETCH et seulement deux micro-instructions pour le décodage/exécution.

Enfin dans notre premier exemple, « `Add D, R, A` », indique que l'on veut additionner le contenu de R avec le contenu de l'adresse mémoire A, le résultat étant placé dans R. En fait l'adresse mémoire d'un opérande est définie par le mode d'adressage. Dans notre cas « D » définit un mode d'adressage direct indiquant que A représente l'adresse de la donnée. L'instruction notée « `Add I, R, A` » indique, cette fois, que le mode d'adressage est indirect : la donnée n'est plus le contenu de A mais le contenu du contenu de A. Ainsi, dans ce cas, A n'est plus l'adresse de la donnée mais contient l'adresse de la donnée.

On peut alors vérifier que dans le cas de l'instruction `Add I, R, A` la séquence de micro-instructions est :

```
1. COsor, RADen, Lect, AttenteMémoire
2. InCo
3. RDOsor, RIen
4. (Adresse)RIsor, RADen, Lect, AttenteMémoire
5. RDOsor, RADen, AttenteMémoire
6. RDOsor, Yen
7. Rsor, Zen, Add
8. Zsor, Ren
```

Ainsi la phase FETCH de cette instruction conduit à la génération des mêmes microcommandes, par contre les phases de décodage et d'exécution donnent lieu à des séquences différentes. La phase de décodage ne tient donc pas seulement compte

du code opération pour déterminer la séquence des micro-instructions, le mode d'adressage intervient également.

Séquencement des microcommandes

Nous avons vu que l'exécution d'une instruction machine se traduisait par une séquence de microcommandes. Le séquenceur produit les signaux et assure l'exécution matérielle des microcommandes. Il existe deux types de séquenceurs : les *séquenceurs câblés* (on parle alors de contrôle câblé) et les *séquenceurs microprogrammés* (on parle alors de contrôle microprogrammé). On conçoit aisément que la réalisation des séquenceurs est très complexe. Il ne s'agit donc pas ici de donner la réalisation de tel ou tel constructeur mais plutôt de mettre en évidence les grandes fonctionnalités qui président à de telles réalisations. Séquenceur câblé et séquenceur microprogrammé représentent deux approches différentes pour le contrôle du chemin de données.

➤ Séquenceur câblé

Le contrôle câblé utilise un automate à états finis implémenté en logique aléatoire ou encore des réseaux logiques programmables (*PLA, Programmed Logic Array*). À titre d'exemple la figure 7.19 donne le schéma de principe d'un PLA. Il s'agit d'un

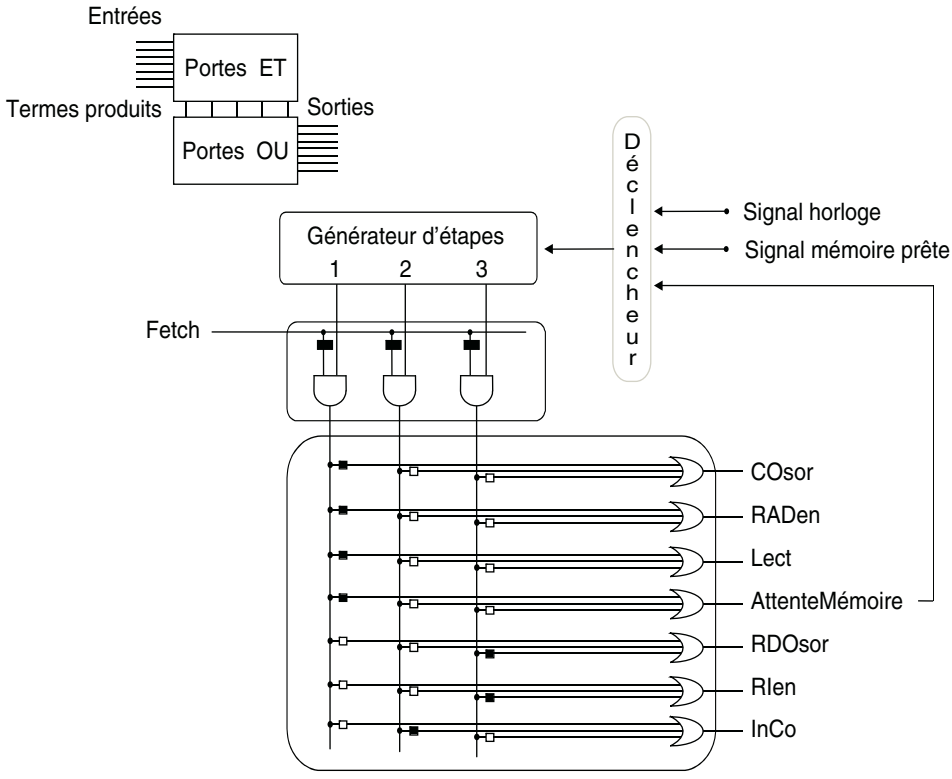


Figure 7.19 Principe d'un circuit PLA.

ensemble de portes ET formant les termes produits et de portes OU formant les termes sommes. À chaque porte ET est associé un fusible qui est intact à l'origine. La programmation de la matrice des portes ET consiste à détruire des fusibles afin d'obtenir les termes produits de la fonction à réaliser. De même pour les portes OU auxquelles sont associés des fusibles que l'on détruit sélectivement pour obtenir les termes sommes souhaités. Dans notre exemple nous avons :

– une entrée nommée Fetch à laquelle trois portes ET sont associées ;
– 7 sorties avec 7 portes OU. Chaque porte OU reçoit les 3 sorties des termes produits.

Les fusibles marqués en noir sont ceux gardés intacts et ce circuit PLA, ainsi programmé, permet la réalisation câblée de la fonction de chargement d'une instruction de la mémoire centrale vers le registre d'instruction.

Le circuit PLA piloté par un générateur d'étapes est lui-même dirigé par un circuit déclencheur. Le circuit déclencheur reçoit en entrée le signal d'horloge, le signal mémoire prête et le signal AttenteMémoire. Si le signal mémoire prête et le signal AttenteMémoire sont positionnés, alors le générateur d'étape est incrémenté d'une unité au prochain top d'horloge. Dans le cas où le signal AttenteMémoire est positionné seul, alors le générateur d'étapes n'est pas incrémenté lors d'un top d'horloge.

Cet ensemble de circuits permet donc la réalisation câblée de la phase Fetch. À l'étape 1, la première porte ET a sa sortie à 1 et donc les signaux Cosor, RADen, Lect et AttenteMémoire sont positionnés à 1. Tant que le signal mémoire prête n'est pas positionné, le générateur d'étapes ne progresse pas. Sitôt le signal positionné, la sortie de la deuxième porte ET devient égale à 1 et ainsi que le signal InCo.

La figure 7.20 synthétise le fonctionnement d'un tel séquenceur. Chaque « pastille » symbolise des portes ET et des portes OU ainsi que les fusibles gardés intacts lors de la réalisation du circuit. Le registre instruction RI contient l'instruction à exécuter soit « `Add D, R, A` », un circuit de décodage interprète le code opération et sélectionne une des entrées de la matrice de termes produits. Dans notre cas il y a 7 entrées et 8 portes ET produisant une matrice de 7×8 portes ET. Il y a de même 16 signaux de sorties nécessitant 16×8 portes OU. Chaque porte OU reçoit 8 entrées.

À l'étape 4 de l'exécution de cette instruction les microcommandes Lect, RIsor, RADen, AttenteMémoire sont positionnés par la logique câblée.

Le circuit déclencheur fonctionne selon l'algorithme :

```
si « horloge » = 1
alors
    si « AttenteMémoire » = 1
    alors
        si « Mémoire Prête » = 1
        alors « Sortie » = 1;
        sinon « Sortie » = 0;
        fsi
    sinon « Sortie » = 1;
    fsi
sinon « Sortie » = 0;
fsi
```

La notation « horloge » = 1 signifie que le signal d'horloge est présent, dans le cas contraire le signal est égal à 0.

Ainsi le circuit déclencheur a une sortie à 1 à chaque top horloge si le signal AttenteMémoire est faux ou si le signal AttenteMémoire est vrai et que la mémoire est prête.

Le générateur d'étapes est un compteur d'étapes qui admet comme signal d'entrée le signal de sortie du déclencheur. Il incrémente le numéro de l'étape à chaque fois que la sortie du déclencheur est à 1.

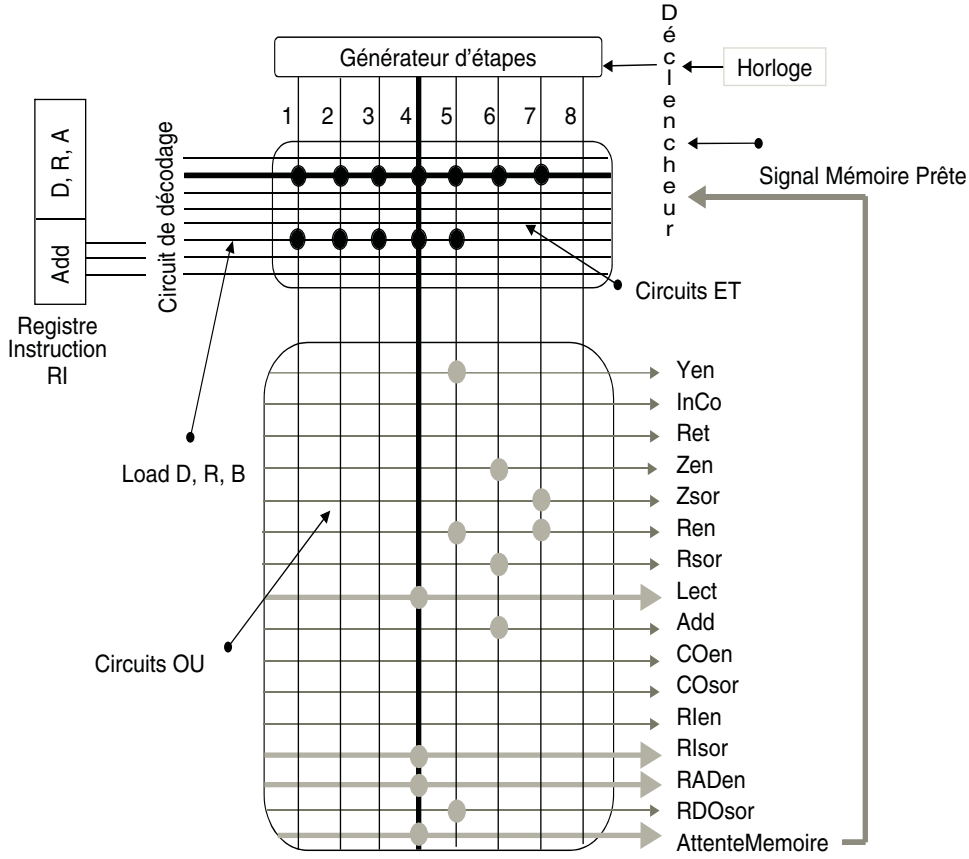


Figure 7.20 Séquenceur câblé.

La figure 7.20 montre la complexité du « câblage » d'un tel séquenceur et illustre la terminologie « séquenceur câblé ». Ces séquenceurs très complexes dans leur réalisation sont très performants car entièrement constitués de circuits électroniques. Par contre aucune erreur n'est permise lors de leur réalisation car il est impossible de modifier le câblage. Enfin il est impossible d'ajouter une instruction qui pourrait être utile après la réalisation d'un tel circuit.

➤ Séquenceur microprogrammé

L'objectif est exactement le même que pour le séquenceur câblé mais la manière d'aborder la question, et de la résoudre est radicalement différente. La figure 7.21 donne le schéma de principe d'un tel séquenceur.

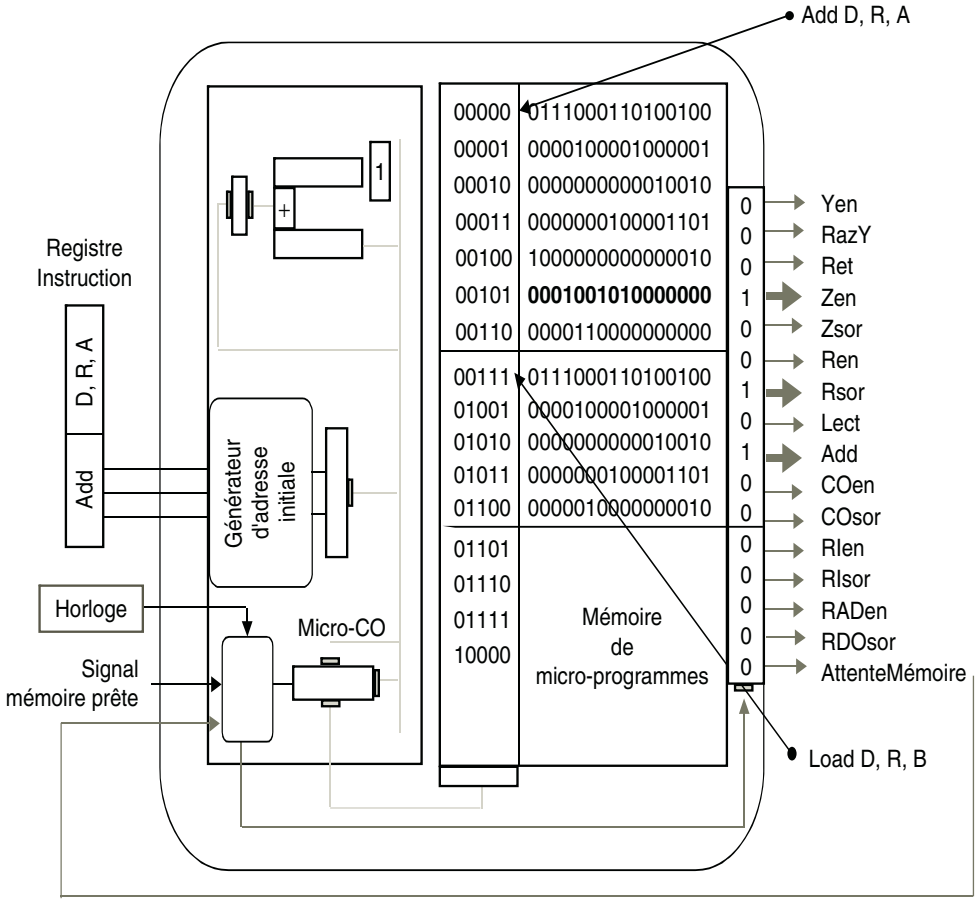


Figure 7.21 Séquenceur microprogrammé.

Ce séquenceur est organisé autour d'une mémoire, dite mémoire de microprogrammes, d'une unité arithmétique, d'un compteur ordinal, dit *microcompteur ordinal,* d'un circuit générateur de l'adresse initiale de l'instruction et enfin d'un circuit déclencheur de l'exécution d'une micro-instruction.

La mémoire de microprogramme contient les micro-instructions associées à chaque instruction machine disponible pour notre microprocesseur. Chaque mot (une ligne sur la figure 7.20) de cette mémoire est adressable et contient une suite de bits associés aux signaux que peut produire le séquenceur. Un mot mémoire contient autant de bits qu'il y a de signaux possibles pour ce séquenceur. Un bit positionné à 1 indique

que le signal associé doit être actif, il est positionné à 0 dans le cas contraire. Par exemple à l'adresse 00101 de la mémoire se trouve une suite de digits binaires permettant de rendre actifs les signaux Zen, Rsor, Add. De l'adresse 00000 à l'adresse 00110 se trouve une suite de mots mémoire correspondant chacun à une micro-instruction. On peut vérifier que les suites de 0 et de 1 de chaque mot équivalent bien à l'activation des signaux permettant l'exécution de l'instruction machine « `Add D, R, A` ». Une telle suite s'appelle un *microprogramme*. Il y a autant de microprogrammes dans la mémoire que d'instructions machines disponibles pour le processeur.

Le registre instruction RI contient l'instruction machine à exécuter. Le code opération est présenté au circuit d'initialisation qui au vu du code opération détermine l'adresse de la première instruction du microprogramme correspondant. Le contenu de la première micro-instruction est placé dans le tampon de sélection des signaux de sortie. Les bits positionnés à 1 rendent actifs les signaux correspondants. Le microcompteur ordinal est incrémenté de 1 et la micro-instruction suivante est exécutée, ainsi de suite jusqu'à la dernière micro-instruction de l'instruction machine. On peut remarquer que l'on a une gestion analogue de l'attente mémoire que dans le cas du séquenceur câblé.

Ce type de séquenceur présente le très gros avantage de permettre des modifications dans le cas d'erreurs de conception des microcommandes. Il permet également d'ajouter des instructions. Les microprogrammes peuvent être stockés dans des mémoires de type ROM.

Lorsque l'on doit réaliser des séquenceurs peu complexes le contrôle câblé en logique aléatoire est plus efficace que le contrôle microprogrammé. Par contre dans le cas de contrôle très complexe la comparaison des différentes approches est beaucoup plus difficile et le choix n'est pas toujours évident.

7.4 LES INTERRUPTIONS : MODIFICATION DU FLUX D'EXÉCUTION D'UN PROGRAMME MACHINE

7.4.1 Principe des interruptions

Nous avons vu que l'exécution d'un programme machine se faisait instruction après instruction de la première jusqu'à la dernière. Une interruption permet d'arrêter l'exécution du programme en cours afin d'exécuter une tâche jugée plus urgente. De cette manière, un périphérique, par exemple, peut signaler un événement important (fin de papier dans l'imprimante, dépassement de température signalée par une sonde placée dans un four…) au processeur qui, s'il accepte cette interruption, exécute un programme de service (dit *programme d'interruption*) traitant l'événement. À la fin du programme d'interruption le programme interrompu reprend son exécution normale.

On distingue plusieurs types d'événements provoquant des interruptions, ils proviennent :

– d'un périphérique. On parle alors d'interruptions externes qui permettent à un périphérique de se manifester auprès du processeur;

– d'un programme en cours d'exécution. Il s'agit d'interruptions logicielles internes souvent nommées *appels systèmes*. Il s'agit de permettre à un programme en cours d'exécution de se dérouter vers un programme du système d'exploitation qui doit gérer une tâche particulière. Par exemple les instructions d'entrées-sorties, permettant les échanges d'informations entre le processeur et les périphériques, sont traitées de cette manière. Un ordre d'entrées-sorties est un appel au système (une interruption logicielle) qui interrompt le programme en cours au profit du programme spécifique (driver ou pilote) de gestion d'un périphérique ;
– du processeur lui-même pour traiter des événements exceptionnels de type division par zéro, dépassement de capacité lors d'une opération arithmétique.

La prise en compte et le traitement d'une interruption s'appuient sur un mécanisme relativement complexe. Afin d'illustrer ce mécanisme nous prenons l'exemple du traitement d'une interruption externe, donc produite par un périphérique. Pour prendre en compte une interruption et la traiter il faut en déterminer son origine car il y a plusieurs sources possibles d'interruptions, puis exécuter le programme adapté. Ce mécanisme s'appuie pour partie sur le matériel (un périphérique positionne un signal indiquant qu'il veut alerter le processeur) et pour partie sur du logiciel de traitement de l'interruption. La figure 7.22 représente la prise en compte d'un événement externe provoquant une interruption.

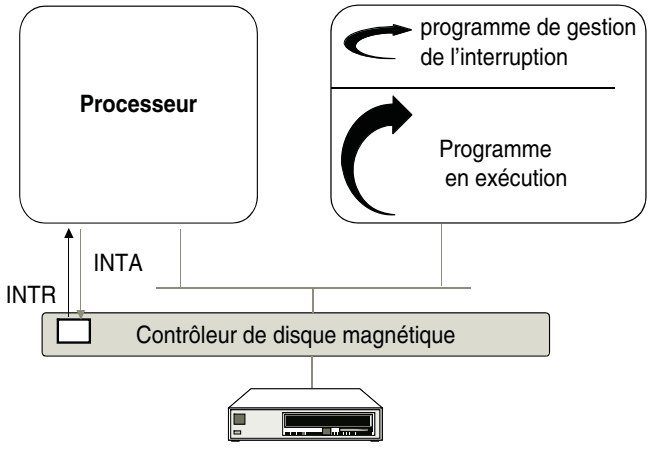


Figure 7.22 Prise en compte d'une interruption externe.

Dans ce schéma tous les périphériques signalent un événement au processeur par le biais d'une ligne d'interruption unique en positionnant le signal INTR. Le processeur, par le signal INTA, indique au périphérique que l'événement a été reçu et qu'il va être pris en compte (il faut éviter que le périphérique continue d'émettre des signaux). En général il n'existe qu'une seule ligne d'interruption et il faut donc déterminer l'origine de l'interruption émise. Cela peut se faire par scrutation, en interrogeant tous les périphériques. Une autre méthode consiste à ce que le périphérique,

après avoir déposé un signal d'interruption, place sur le bus de communication l'identification de l'interruption. Le processeur prévenu d'un signal d'interruption (INTR) sait qu'il doit lire le bus de communication pour connaître la nature de l'interruption. Connaissant la nature de l'interruption, le processeur peut exécuter le programme spécifique de traitement. Ce mécanisme implique des aspects matériels (mise en place du signal) et logiciels (programmes de reconnaissance et de traitement). On voit donc que la prise en compte de ce mécanisme implique la coexistence simultanée en mémoire de plusieurs programmes machine. Les uns sont de type système d'exploitation (les programmes de traitement des interruptions), les autres de type utilisateur.

La figure 7.23 résume ce mécanisme. On trouve les programmes de gestion des interruptions (reconnaissance et traitement) et une zone mémoire, appelée *vecteur d'interruptions*, contenant les adresses mémoires des programmes de traitement d'une interruption. À chaque entrée de ce vecteur correspond une interruption particulière. Lorsque le périphérique signale une interruption il dépose sur le bus le numéro de l'interruption, ce numéro identifie un point d'entrée dans le vecteur d'interruptions donc l'adresse du programme de traitement.

La prise en compte d'une interruption se fait selon la séquence :

1. le programme utilisateur dispose du processeur. C'est lui qui est en cours d'exécution. Il dispose des registres, de l'unité arithmétique et logique, de l'unité de

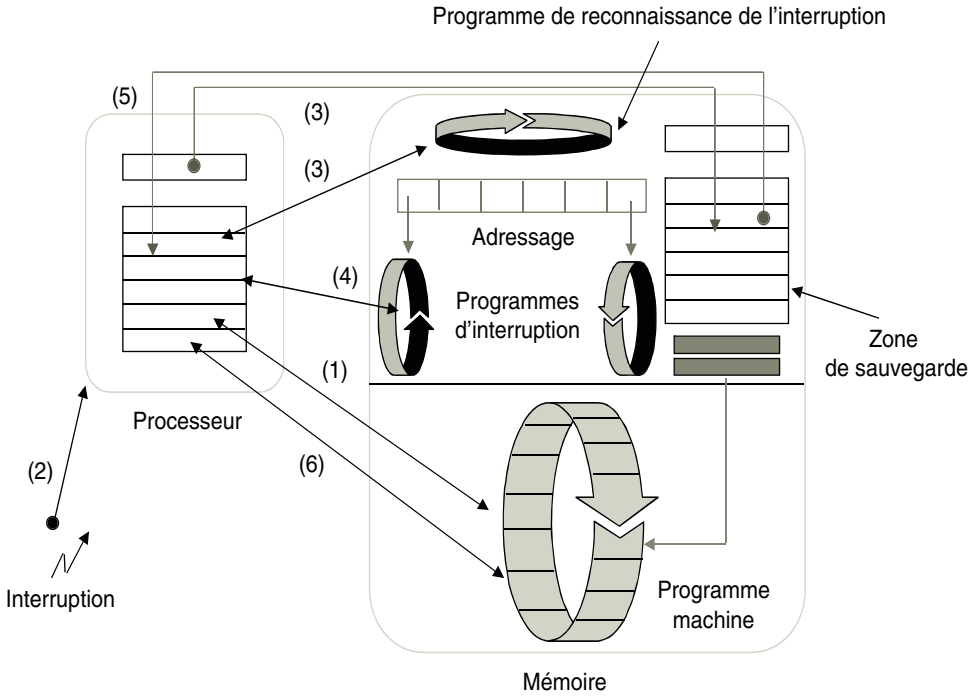


Figure 7.23 Prise en compte d'une interruption.

commande. Le compteur ordinal CO contient l'adresse de la prochaine instruction à exécuter ;

2. une interruption est postée par un périphérique ;
3. il y a sauvegarde du contexte matériel d'exécution du programme utilisateur et en particulier du compteur ordinal CO. Ceci est nécessaire pour la reprise ultérieure de ce programme. Le programme de reconnaissance s'exécute et lit le numéro de l'interruption. Le numéro de l'interruption permet l'identification de l'adresse du programme de traitement ;
4. le compteur ordinal CO est chargé avec l'adresse du programme de traitement et celui-ci s'exécute ;
5. le contexte d'exécution du programme utilisateur est rechargé dans le processeur à la fin de l'exécution du programme d'interruption ;
6. le programme utilisateur reprend son exécution.

En résumé le mécanisme d'interruption modifie le flux standard d'exécution d'un programme machine. Le flux d'exécution, tenant compte de possibles interruptions, d'un programme est résumé dans la figure 7.24.

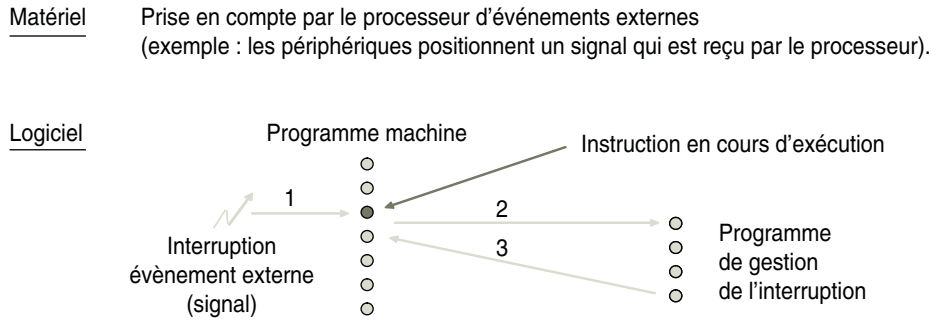


Figure 7.24 Flux d'un programme avec prise en compte des interruptions.

Avant d'exécuter une instruction (nouvelle phase de Fetch), le processeur vérifie la présence ou l'absence d'un signal d'interruption. Si le signal d'interruption est posté, le mécanisme de traitement de l'interruption est mis en œuvre sinon l'exécution du programme utilisateur continue en séquence. Une instruction commencée se termine toujours même si une interruption est arrivée pendant son exécution. L'interruption sera prise en compte après la fin de l'instruction et avant le début de l'instruction suivante.

Ce mécanisme est fondamental, nous n'avons fait que l'aborder superficiellement car il est d'une grande complexité dès que l'on entre dans les détails de sa réalisation. Il est, en particulier, utilisé par le système d'exploitation pour le traitement efficace des entrées-sorties, nous y reviendrons donc dans la partie concernant le traitement matériel des entrées-sorties mais aussi dans les chapitres concernant le système d'exploitation.

7.4.2 Un exemple

Nous allons examiner, sous forme d'un exemple, les différents composants matériels, et la manière dont certaines interruptions sont prises en compte dans le cas d'un PC piloté par un processeur de type 80xx (jusqu'à 80486). Avec les nouveaux processeurs les traitements et matériels évoqués dans cet exemple ont évolué mais les mécanismes décrits et leur prise en compte gardent leur généralité.

D'une manière générale lorsqu'une interruption provenant d'un périphérique se produit il faut répondre à plusieurs questions :

– Comment l'unité centrale peut-elle reconnaître le périphérique ayant émis le signal d'interruption ?
– Comment l'unité centrale détermine l'adresse du programme de traitement de l'interruption ?
– Comment prendre en compte la situation ou deux, voir plus, interruptions se produisent simultanément ?
– Comment prendre en compte une nouvelle interruption qui se produit pendant l'exécution du programme de traitement d'une autre interruption (imbrication des interruptions) ?

Les signaux

Lorsqu'un périphérique émet une interruption, l'unité centrale est prévenue par le signal INTR. Celle-ci émet alors le signal d'acquittement INTA, puis elle réserve le bus de communication et attend que le périphérique envoie le numéro de l'interruption en utilisant le bus de communication.

Vecteur d'interruptions

Pour déterminer l'adresse du programme de traitement de l'interruption l'unité centrale dispose d'un tableau appelé *vecteur d'interruptions* ayant autant d'entrées que d'interruptions possibles. Le tableau 7.1 donne une image partielle du vecteur d'interruptions.

La colonne de gauche indique les adresses d'implantation du vecteur d'interruptions. Ces adresses sont exprimées en notation hexadécimale. À chaque ligne du tableau correspond une entrée dans le vecteur d'interruptions, chaque entrée contient 4 octets. Chaque entrée du vecteur d'interruptions est associée à une interruption et délivre l'adresse du programme de traitement correspondant.

La deuxième colonne contient le numéro de l'interruption et la troisième colonne son identification. Par exemple l'adresse 2C-2F du vecteur d'interruptions contient l'adresse du programme de traitement de l'interruption associée au port de communication COM1.

Toutes les entrées du vecteur d'interruptions ne sont pas notées dans le tableau, le vecteur d'interruptions occupant toutes les adresses mémoire de 0 à 7F. La dernière colonne n'est pas référencée car le nommage des points d'entrée dans les logiciels de traitement dépend du système d'exploitation. Ainsi les signaux INTR, INTA et le

vecteur d'interruptions permettent de répondre aux deux premières questions que pose la prise en compte d'une interruption.

Pour les deux dernières questions, notre ordinateur est muni de matériel permettant d'une part d'arbitrer les conflits d'accès au bus (boîtier 8288), d'autre part d'accepter plusieurs lignes d'interruption logique, de positionner le signal INTR et de respecter une logique de gestion des priorités et de l'imbrication des différentes interruptions (boîtier 8259). Implicitement les interruptions sont classées par ordre de priorité. Quand deux interruptions se présentent simultanément, la plus prioritaire des deux est traitée, l'autre étant mise en attente. Une interruption survenant alors qu'une autre interruption est déjà en cours de traitement ne sera prise en compte que si sa priorité est plus élevée que celle de l'interruption actuellement en cours de traitement.

Tableau 7.1 VECTEUR D'INTERRUPTIONS.

Adresse (Héxadécimal)	Numéro de l'interruption (Héxadécimal)	Nom de l'interruption	Point d'entrée dans le logiciel de traitement
0-3	0	Division par 0	
10-13	4	Débordement	
20-23	8	Horloge	
24-27	9	Clavier	
28-2B	A	Réservé	
2C-2F	B	Communication1	
30-33	C	Communication2	
34-37	D	Disque	
38-3B	E	Disquette	
3C-3F	F	Imprimante	
40-43	10	Écran	
70-73	1C	Top horloge	
7C-7F	1F	Caractères graphiques d'écran	

Architecture du matériel permettant la prise en compte des interruptions

Le matériel qui intervient comprend un boîtier (boîtier 8259) recevant les interruptions. Pour ce boîtier les interruptions sont notées IRQx. Ainsi, IRQ1 caractérise une interruption clavier, IRQ0 l'interruption horloge, IRQ5 une interruption disque etc. À la réception d'un ou plusieurs signaux le boîtier émet le signal INTR vers le processeur. Il s'adresse alors à l'arbitre de bus afin de réserver le bus au travers duquel le périphérique ayant émis le signal transmet le numéro de l'interruption. L'arbitre de bus est un module fondamental qui permet le partage d'une ressource matérielle unique (le bus) entre plusieurs utilisateurs potentiels (figure 7.25).

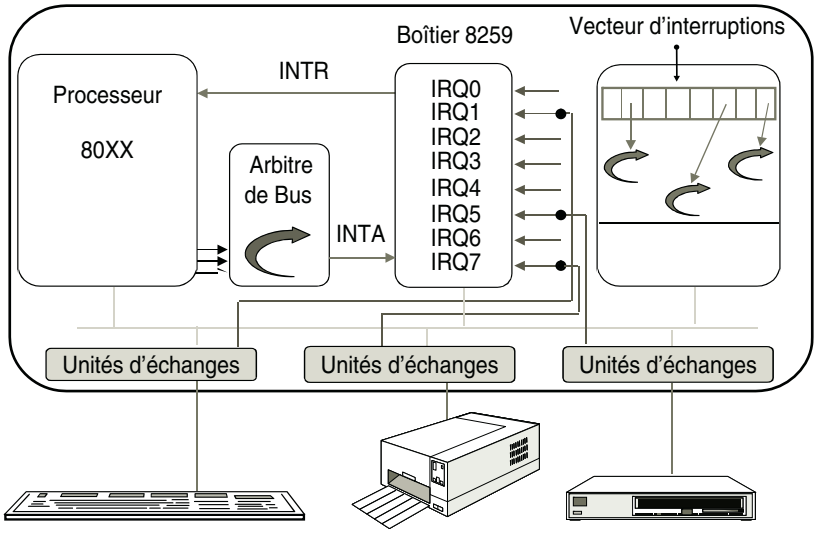


Figure 7.25 Matériel pour la prise en compte des interruptions.

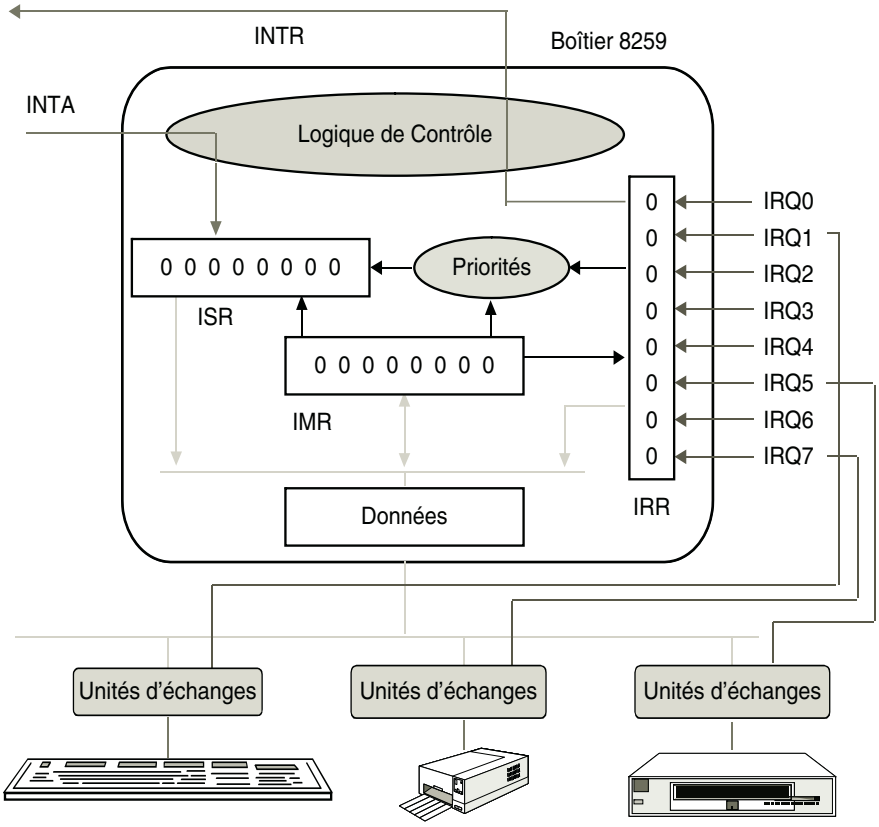


Figure 7.26 Boîtier 8259.

L'arbitre de bus envoie le signal d'acquittement INTA indiquant que le signal a été pris en compte. Le numéro de l'interruption est alors envoyé au processeur. Ce numéro permet de trouver le point d'entrée dans le vecteur d'interruptions et donc de connaître l'adresse du programme de traitement du signal.

La figure 7.26 détaille le contenu du boîtier 8259 afin de mieux comprendre les mécanismes de gestion des priorités et d'imbrication des interruptions. Les interruptions sont classées selon un ordre de priorité, 0 est la plus forte, 7 la plus faible. Le registre IRR (registre des requêtes d'interruptions) stocke les demandes d'interruptions en positionnant à 1 le bit correspondant dans le registre. Le registre ISR (registre des interruptions en cours) contient un enregistrement des interruptions en cours de traitement. Le registre IMR (masque des interruptions) contient un enregistrement des interruptions actives.

Le tableau 7.2 donne un exemple de comportement du boîtier qui opère en mode totalement imbriqué, c'est-à-dire qui gère les interruptions simultanées et les éventuelles imbrications.

Tableau 7.2 EXEMPLE DE COMPORTEMENT DU BOÎTIER 8259.

Événement	Statut	IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	IRQ0
	ISR Priorités Int. acceptées	0 7 x	0 6 x	0 5 x	0 4 x	0 3 x	0 2 x	0 1 x	0 0 x
Requête Int.				*					
	ISR Priorités Int acceptées	0 7	0 6	1 5	0 4 x	0 3 x	0 2 x	0 1 x	0 0 x
Requête Int.		*			*	*			
	ISR Priorités Int acceptées	0 7	0 6	1 5	0 4	1 3	0 2 x	0 1 x	0 0 x
Fin Int.									
	ISR Priorités Int acceptées	0 7	0 6	1 5	0 4 x	0 3 x	0 2 x	0 1 x	0 0 x
Requête Int.		*			*				
	ISR Priorités Int acceptées	0 7	0 6	1 5	1 4	0 3 x	0 2 x	0 1 x	0 0 x

Dans notre exemple, au début tous les niveaux d'interruption peuvent être acceptés : il n'y a aucun bit positionné dans l'ISR. La première interruption est de niveau 5 et donc seuls les niveaux de 4 à 0 peuvent êtres honorés. Dans le second état trois interruptions interviennent simultanément : IRQ3, IRQ4, IRQ7. L'arbitre

élimine IRQ7 puisque IRQ5 est en cours de traitement et qu'elle est plus prioritaire. Il élimine également IRQ4 puisque IRQ3 est plus prioritaire. IRQ3 provoque donc l'émission d'un signal INTR vers le processeur. Le processeur en accuse réception en positionnant le bit correspondant dans l'ISR. La procédure de traitement de IRQ5 est interrompue au profit de IRQ3. L'événement Fin Int. marque la fin du traitement de IRQ3, le bit 3 de ISR est donc remis à 0, ce qui rend éligible les interruptions de niveaux inférieurs à 5. IRQ5 est terminée, puis IRQ4 en attente est ensuite traitée. IRQ7 reste en attente.

Ce petit exemple met en évidence la manière de régler les deux dernières questions posées par la prise en compte d'interruptions (priorité et imbrication). C'est un mécanisme qui fonctionne sur le mode d'une pile, permettant la suspension d'un service de plus faible priorité et sa reprise dès que le service de plus haute priorité est terminé.

7.5 AMÉLIORATION DES PERFORMANCES

Pour améliorer les performances globales d'un ordinateur, différentes approches sont possibles :

– augmenter la vitesse des microprocesseurs. C'est en général la première approche qui se traduit par l'augmentation de la fréquence d'horloge. Actuellement on trouve fréquemment des processeurs à 1,5 GHz voire 3 ou 4 GHz. Cette évolution a pourtant des limites et dépasser certaines vitesses implique des changements technologiques importants qui ne sont pas nécessairement simples à mettre en place surtout dans le cas de production d'ordinateurs pour le grand public ;
– travailler sur l'architecture interne du microprocesseur. Dans nos exemples nous avons choisi une architecture simple à bus interne unique. En fait on peut considérablement améliorer les performances en multipliant le nombre de bus internes. On trouve fréquemment des microprocesseurs avec deux voire trois bus internes permettant la parallélisation de certaines parties du chemin de données. La multiplication des registres internes peut permettre aussi une amélioration des performances. Ce type de considération est à l'origine de la conception des processeurs RISC en comparaison aux processeurs CISC. Nous examinerons plus en détail ces questions dans la partie consacrée à la comparaison RISC/CISC ;
– améliorer les performances du bus de communication entre processeur et mémoire centrale. Ces améliorations portent généralement sur la fréquence du bus et sa largeur. Cependant augmenter la largeur du bus pour transporter en un seul cycle une plus grande quantité d'information nécessite une plus grande surface ce qui est contradictoire avec la miniaturisation nécessaire sur d'autres plans ;
– en fait d'autres approches sont possibles. Elles portent principalement sur la parallélisation de certaines opérations qui consiste à faire plusieurs choses en même temps. On trouve deux formes de parallélisation : le parallélisme au niveau des instructions et le parallélisme au niveau des processeurs. Dans le premier cas on permet l'exécution simultanée de plusieurs instructions machine et l'on exécute

ainsi plus d'instructions par seconde. Dans le deuxième cas plusieurs processeurs travaillent simultanément sur un programme machine ce qui améliore globalement les performances de l'ordinateur.

7.5.1 Parallélisme des instructions

L'exécution d'une instruction peut se décrire par :

```
début
fetch :     lecture de l'instruction et modification du compteur ordinal;
décodage :  reconnaissance de l'instruction, lectures de opérandes
            ➥ ou calcul des adresses mémoire, ou calcul d'une adresse
            ➥ de branchement;
exécution : exécution des micro-instructions;
rangement : stockage des résultats;
fin
```

Technique de pipeline

La figure 7.27 montre que l'exécution d'une instruction peut être décomposée en plusieurs phases qui s'exécutent indépendamment les unes des autres si l'on dispose d'unités fonctionnelles (du matériel) le permettant. Dans notre exemple l'exécution est ainsi décomposée en 4 phases, chaque phase étant prise en charge par une unité fonctionnelle différente, créant ainsi un *pipeline à 4 étages*.

Dans notre exemple la phase k de l'instruction i s'exécute en même temps que la phase k – 1 de l'instruction i + 1, que la phase k – 2 de l'instruction i + 2, etc.

Dans le cas idéal chaque phase est réalisée en un cycle d'horloge et si une instruction peut être décomposée en n phases alors n instructions peuvent être exécutées en parallèle. Ainsi pour notre pipeline à 4 étages si le cycle horloge est de 2 nanosecondes, alors il faut 8 nanosecondes pour exécuter une instruction et cette machine fonctionne en apparence à 80 MIPS[1]. En réalité la technique du pipeline lui permet

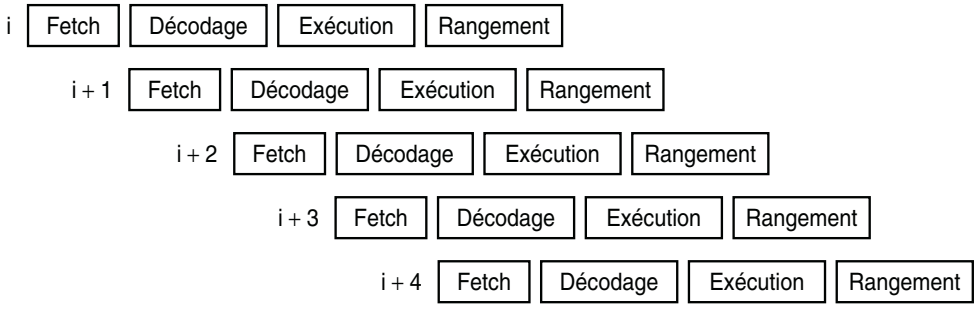


Figure 7.27 Pipeline à 4 étages.

1. MIPS (*Million of Instructions Per Second*) est une mesure de performances d'un microprocesseur qui mesure le nombre d'instructions exécutables par seconde.

de fonctionner à 4 × 80 MIPS puisque le pipeline est à 4 étages et permet donc à 4 instructions de s'exécuter en parallèle. En fait ce n'est que lorsque le pipeline est plein que le processeur fonctionne à cette vitesse.

En réalité ce schéma est idéal mais plusieurs circonstances mettent en défaut ce schéma et ralentissent donc le fonctionnement du pipeline :

– les conflits de ressources. Il faut interdire l'accès simultané à une ressource par plusieurs instructions ;
– les dépendances des données. C'est le cas où une instruction i a besoin comme opérande du résultat de l'instruction i – 1. Les deux phases ne peuvent pas être menées simultanément ;
– les conflits liés au contrôle. Les instructions de sauts inconditionnels et de branchements conditionnels modifient le compteur ordinal et donc perturbent la simultanéité d'exécution qu'impose le pipeline.

Malgré les circonstances qui empêchent le fonctionnement idéal d'un pipeline cette technique reste très efficace pour l'amélioration des performances d'un ordinateur. Multiplier les pipelines permet encore des améliorations de performances. La figure 7.28 donne le schéma d'un pipeline double.

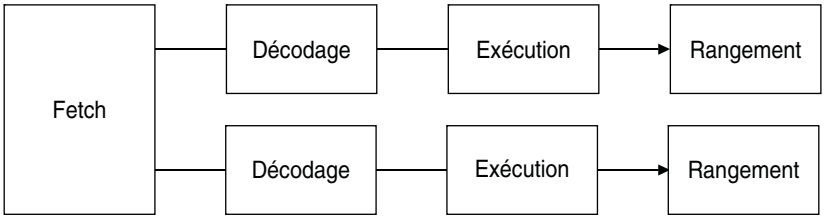


Figure 7.28 Pipeline double.

Pour ce pipeline double, une unité unique de chargement charge des paires d'instructions qui s'exécutent simultanément dans leur propre pipeline. Cette technique implique impérativement que les instructions qui s'exécutent n'aient aucune interrelations et dépendances, autrement dit elles ne doivent pas utiliser de ressources communes. C'est le compilateur qui a en charge la garantie de l'indépendance fonctionnelle des instructions (c'était déjà le cas pour le pipeline unique). Actuellement les machines disposent toutes de pipelines simples ou doubles dans les machines RISC. Le processeur Intel 486 (machine CISC) dispose d'un pipeline tandis que le Pentium (CISC) dispose de deux pipelines à 5 étages.

En fait les règles qui permettent de déterminer si deux instructions d'une paire peuvent effectivement s'exécuter simultanément sont extrêmement complexes. Une instruction d'une paire peut voir son exécution retardée parce qu'elle n'est pas compatible avec l'autre instruction, ce qui oblige à réorganiser l'ordonnancement de l'exécution des instructions pour profiter de l'efficacité des pipelines doubles. Il s'agit du concept *d'ordonnancement dynamique des instructions*. C'est le compilateur qui est chargé de cette réorganisation très complexe. Plusieurs études montrent

que l'on peut avoir des gains de performances allant du simple au double entre le processeur 486 et le Pentium fonctionnant à la même fréquence d'horloge par utilisation du pipeline double du Pentium.

Architecture superscalaire

Une autre approche consiste à disposer dans le pipeline de plusieurs unités d'exécutions (figure 7.29).

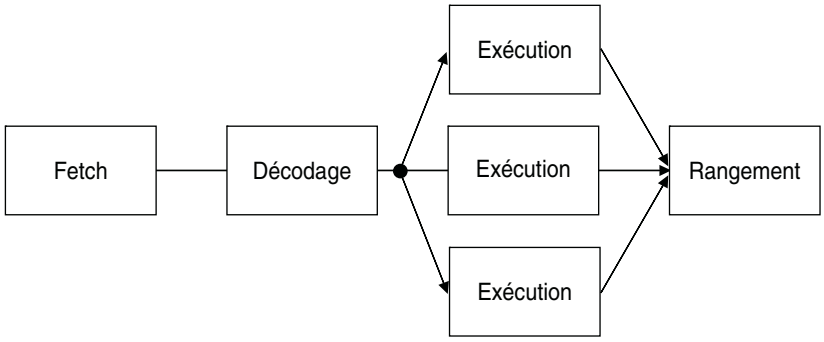


Figure 7.29 Architecture superscalaire.

Le Pentium 2 dispose de ce type de pipeline. Une telle architecture correspond à ce que l'on appelle une *architecture superscalaire*. Ce type d'architecture est en fait assez ancien et de tels choix architecturaux existaient déjà dans les ordinateurs Control Data tel que le CDC 6600. Le CDC 6600 disposait de 10 unités d'exécutions qui parallélisaient donc très fortement l'exécution effective des instructions.

7.5.2 Parallélisme des processeurs

Cet aspect de l'amélioration des performances par l'utilisation d'un parallélisme entre processeurs est complexe et dépasse probablement le cadre d'un tel ouvrage. Nous donnons donc ici uniquement les grandes classes d'organisations que l'on trouve dans ce domaine.

Processeurs matriciels et vectoriels

Il s'agit d'une organisation où l'on dispose d'une seule unité de commande et de plusieurs unités de calcul (UAL). L'unité de commande pilote et organise la parallélisation de l'exécution des instructions.

Un processeur matriciel exécute la même séquence d'instructions sur des données différentes et un *processeur vectoriel* traite efficacement des séquences d'instructions portant sur des paires de données (les machines Cray sont à base de processeurs vectoriels).

Les processeurs matriciels sont plus performants mais aussi beaucoup plus chers et beaucoup plus complexes à programmer que les processeurs vectoriels. Un proces-

seur vectoriel peut être associé comme coprocesseur arithmétique à un processeur conventionnel.

Multiprocesseur

Dans un tel système tous les processeurs sont autonomes et possèdent chacun leur unité de commande et leur UAL mais ils partagent la mémoire centrale. Chaque processeur utilise une partie de cette mémoire. A priori chaque processeur peut lire et/ou écrire dans la mémoire partagée et c'est au système d'exploitation de gérer les conflits d'accès à cette ressource commune partagée. Plusieurs implantations d'une telle architecture sont possibles :

– les processeurs se partagent l'accès à la mémoire commune au travers d'un bus unique, ce qui génère un grand nombre de conflits d'accès à la mémoire ;
– les processeurs disposent de mémoires privées pour les instructions et les données qui n'ont pas à être partagées.

Les multiprocesseurs sont relativement simples à programmer ce qui en fait leur intérêt.

Les multi-ordinateurs

En fait il s'agit ici de remédier aux difficultés liées à l'architecture multiprocesseurs et d'éviter les conflits d'accès à une mémoire commune. Dans cette architecture, dite architecture distribuée, la communication entre ordinateurs se résume à des échanges de messages au travers d'un réseau de communication. Dans ce contexte on trouve des applications telles que les échanges de fichiers (ftp) ou la mise en place de terminaux distants faciles à utiliser (telnet). En fait l'efficacité du système d'ensemble repose essentiellement sur la qualité du réseau de communication.

7.6 CONCLUSION

Dans cette partie nous avons vu que l'objet de la fonction d'exécution d'un ordinateur est d'exécuter une suite d'instructions sur un ensemble de données. Pour cela un problème est traduit en une suite d'instructions machines caractéristiques d'un processeur capable de les exécuter en utilisant un ensemble de circuits électroniques. Nous avons été amenés à indiquer que le programme machine exécutable doit être placé dans la mémoire principale (mémoire centrale, mémoire RAM) et que le microprocesseur exécute ce programme instruction après instruction selon un algorithme précis. En fait en présentant ce chapitre nous avons présenté les fonctions essentielles de l'exécution d'un programme reposant sur une architecture de machine dite architecture de Von Neumann qui correspond encore à l'architecture la plus répandue. Dans la partie amélioration des performances nous avons évoqué différentes possibilités d'augmentation des performances par la parallélisation des instructions et des processeurs. Cet aspect des choses nous amène en fait à considérer plusieurs architectures possibles de machines. En 1972 Flynn propose une classification des machines

qui s'appuie sur le fait que l'exécution d'un programme correspond à l'exécution d'un flux d'instruction portant sur un flux de données. Il y a alors quatre manières d'associer flux d'instructions et flux de données, ces quatre manières caractérisant quatre architectures possibles de machines. Ces différentes possibilités sont résumées dans le tableau 7.3.

Tableau 7.3 CLASSIFICATION DES ARCHITECTURES D'ORDINATEURS.

Flux d'instructions	Flux de données	Désignation	Type de machine
1	1	SISD : Single Instruction Single Data	Von Neumann
1	Plusieurs	SIMD : Single Instruction Multiple Data	Calculateur vectoriel
Plusieurs	1	MISD : Multiple Instruction Single Data	
Plusieurs	Plusieurs	MIMD : Mutiple Instruction Multiple Data	Multiprocesseur, Multicalculateur

Chapitre **8**

La fonction de mémorisation

Ce chapitre est consacré à l'étude de la fonction de mémorisation. Celle-ci est organisée autour de plusieurs niveaux de mémoire qui réalisent ainsi une hiérarchie de mémoires, dont le but principal est de simuler l'existence d'une mémoire de grande capacité et de grande vitesse. Dans cette hiérarchie, les mémoires les plus proches du processeur sont les mémoires les plus rapides mais aussi les plus petites en capacité. Au contraire, les mémoires les plus éloignées du processeur sont les plus lentes mais sont de grandes capacités. Nous commençons par étudier physiquement les différents types de mémoires existantes sur un ordinateur, telles que les mémoires RAM et ROM. Puis dans une seconde partie consacrée à l'amélioration des performances, nous abordons les mécanismes de mémoire caches et de mémoire virtuelle. Nous terminons par un complément sur les architectures RISC et CISC.

8.1 GÉNÉRALITÉS

Nous avons vu que le programme et les données sont enregistrés dans la mémoire centrale (la mémoire RAM) pour être exécutés par le processeur central (microprocesseur). Lors de l'exécution d'une instruction, le processeur utilise des registres, qui sont également des zones de stockage de l'information. Au moment du démarrage de l'ordinateur s'exécute un programme préenregistré dans une mémoire spécialisée (la mémoire ROM) qui charge dans la mémoire centrale le noyau du système d'exploitation à partir du disque magnétique. Lors de l'exécution d'une instruction par le

processeur nous avons vu que le processeur peut exécuter des cycles d'horloge sans avoir d'activité : il attend la réponse de la mémoire à une demande de lecture. Pour harmoniser les vitesses du processeur et la mémoire RAM, et ainsi améliorer les performances, une mémoire plus rapide est placée entre le processeur et la mémoire RAM : c'est la *mémoire cache*. Ainsi la mémorisation de l'information dans un ordinateur ne se fait pas en un lieu unique mais, comme le montre la figure 8.1, est organisée au travers d'une *hiérarchie de mémoires*.

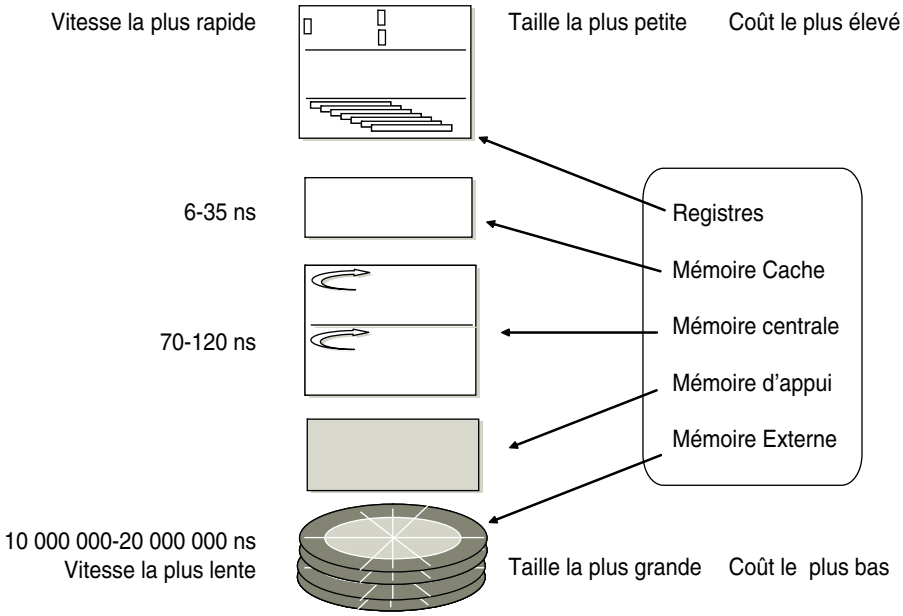


Figure 8.1 Hiérarchie de mémoires.

Toutes les mémoires, même si elles participent à la même fonction, ne jouent pas le même rôle et on peut arbitrairement les classer en deux grandes catégories :

– *les mémoires de travail* désignent les mémoires qui sont actives dans l'exécution d'un programme. On y trouve les registres du processeur, la mémoire centrale, la mémoire cache, la mémoire d'appui, la mémoire morte. Ce sont des mémoires électroniques ;
– *les mémoires de stockage* telles que le disque magnétique ont pour objet de conserver de manière permanente de grandes quantités d'informations. Les informations qui y sont stockées ne participent pas directement à l'exécution d'un programme mais doivent êtres chargées en mémoire centrale pour être exploitées par le processeur. Ce sont les mémoires de masses, elles sont de type magnétique ou optique.

La fonction de mémorisation peut être considérée selon deux angles :

– l'organisation générale et le fonctionnement des différentes mémoires. C'est cet aspect que nous traiterons dans ce chapitre ;

– la gestion de la mémoire. Il s'agit là de définir comment et par qui est utilisée cette mémoire. À titre d'exemple pour qu'un programme s'exécute il faut qu'il soit chargé dans la mémoire centrale. Il existe plusieurs manières de réaliser cette fonction de chargement. De même nous avons vu que pour prendre en charge le mécanisme d'interruptions, plusieurs programmes, les programmes du système d'exploitation et le(s) programme(s) utilisateur(s), doivent simultanément être présents dans la mémoire centrale. Dans ce cas la gestion de la mémoire consiste en un partage harmonieux de cette mémoire entre les différents programmes. Dans le cas où plusieurs utilisateurs utilisent en même temps le même ordinateur il faut une gestion de la mémoire leur permettant de partager cette ressource commune. Il ne s'agit plus là de savoir comment fonctionnent les différentes mémoires mais comment, compte tenu de ces fonctionnements, elles peuvent être utilisées par les programmes utilisateurs. Comment par exemple permettre à un programme dont la taille est plus grande que la mémoire centrale de s'exécuter dans cette mémoire centrale. C'est une des tâches importantes du système d'exploitation que d'assurer la gestion de la mémoire. Cette question est abordée dans la troisième partie de l'ouvrage qui concerne les systèmes d'exploitation (chapitre 13).

Dans ce qui suit nous nous intéressons à l'organisation générale des mémoires et à leur fonctionnement. L'accent est surtout mis sur les mémoires de travail, quelques indications sont cependant données concernant les mémoires de masse.

Plusieurs critères importants caractérisent les mémoires :

– la *capacité* de la mémoire indique la quantité d'information qu'une mémoire peut stocker. En général cette capacité peut s'exprimer en bits, en octets, plus rarement en mots. Le tableau 8.1 résume les principales expressions de la capacité mémoire;
– le *temps d'accès de la mémoire* caractérise le temps nécessaire pour obtenir une information en mémoire. Pour une mémoire électronique qui est une mémoire très rapide (RAM, ROM, registre…) ce temps se mesure en nanosecondes (milliardième de seconde : 10^{-9} s). Cette vitesse peut également s'exprimer comme une fréquence d'horloge caractéristique de la mémoire, égale à l'inverse du temps d'accès et mesurée en hertz (Hz). Ainsi un temps d'accès égal à 10 nanosecondes correspond à une fréquence de 100 Mhz (1 Mhz = 10^6 Hz). Pour des mémoires magnétiques ou optiques (mémoire de masse) ce temps se mesure en millisecondes (millième de seconde : 10^{-3} s). Cet écart de temps est fondamental à noter pour comprendre le rôle respectif de ces différentes mémoires dans un ordinateur;

Tableau 8.1 EXPRESSIONS DE LA CAPACITÉ MÉMOIRE.

Terminologie	Abréviation	Capacité en octets	
Kilo-octet	Ko	1 024 octets	2^{10} octets
Méga-octet	Mo	1 024 Ko	2^{20} octets
Giga-octet	Go	1 024 Mo	2^{30} octets
Téra-octet	To	1 024 Go	2^{40} octets

– la *bande passante* de la mémoire. Ce critère s'exprime sous la forme du produit de la largeur du bus de données et de la fréquence de la mémoire. Par exemple pour une mémoire de 64 bits de largeur sur un bus à 100 MHz la bande passante est de 800 Mo/s. Ce critère est très utilisé et tend à remplacer le critère de temps d'accès, il permet, en effet, de mieux évaluer les différents débits à synchroniser entre les modules de l'ordinateur ;
– le *temps de latence* mesure le temps nécessaire à la réalisation d'une opération. C'est un critère important qui intervient pour tous les types de mémoire. Il exprime que tous les cycles horloge ne pourront être utilisés. Une latence de 3 indique que l'obtention de l'information « coûte » 3 cycles horloge, il y a donc perte de cycles horloge. Pour un disque magnétique, le temps de latence mesure le temps qu'il faut pour que, une fois la piste atteinte, la donnée se trouve sous la tête de lecture. Il varie de 0 à une révolution complète du disque ;
– la *volatilité* représente le temps pendant lequel une information est disponible en mémoire. Les mémoires magnétiques sont non volatiles : l'information est conservée même après l'arrêt de l'alimentation électrique. Les mémoires électroniques sont généralement volatiles et perdent l'information dès que l'alimentation électrique est coupée : c'est le cas des mémoires vives telles que les mémoires RAM ;
– l'*encombrement*. Les mémoires physiques occupent une place de plus en plus petite ce qui permet une plus grande intégration. C'est un facteur important de développement de l'informatique ;
– le *coût* est un critère très important dans les développements de l'informatique. Les mémoires électroniques ont un coût de stockage relativement élevé et donc leurs capacités sont d'autant plus faibles. Les mémoires magnétiques sont beaucoup moins onéreuses et présentent donc de beaucoup plus grandes capacités de stockage.

8.2 MÉMOIRES DE TRAVAIL

Dans cette partie nous examinons selon divers points de vue les différentes mémoires de travail : RAM, ROM, registres. Nous consacrerons un paragraphe particulier aux mémoires caches qui bien que faisant parties des mémoires de travail jouent un rôle si important que nous avons souhaité les traiter à part. Dans une première partie nous abordons une approche externe, puis un point de vue interne où pour quelques cas nous présentons des exemples de réalisation possible.

8.2.1 Les mémoires vives

Ce sont des mémoires sur lesquelles les opérations de lecture et d'écriture sont possibles. Ce sont les mémoires dites RAM (*Random Access Memory*), où le temps d'accès est indépendant de la place de l'information dans la mémoire. Elles sont volatiles et le risque de perte d'informations est non négligeable (microcoupures de l'alimentation). Le temps d'accès est très faible (mémoire rapide) et la consommation électrique est faible. Elles sont essentiellement utilisées en tant que mémoire centrale et mémoires caches.

Les différents types de mémoire vive

La figure 8.2 présente schématiquement les différentes mémoires vives que l'on trouve actuellement :

– Les mémoires SRAM (*Static RAM*) sont construites à partir de bascules de transistors qui permettent, si l'alimentation est maintenue, de conserver l'information très longtemps. Ce sont des mémoires très rapides mais chères et qui induisent des difficultés d'intégration. Elles sont de faibles capacités et sont plutôt réservées aux mémoires caches.

– Les mémoires DRAM (*Dynamic RAM*) sont la base des mémoires centrales. Le bit est associé à un seul transistor (contrairement aux SRAM) ce qui offre donc une très grande économie de place et favorise une grande densité d'intégration. Le transistor est associé à la charge d'un condensateur qui diminue avec le temps. La DRAM est donc volatile et nécessite un rafraîchissement régulier. Ces mémoires sont en très forte évolution tant au plan des coûts que des performances. Les mémoires EDO sont de type DRAM traditionnelle mais leur temps de latence est beaucoup plus faible et permet donc des temps d'accès plus petits. La mémoire SDRAM permet des échanges synchronisés avec le processeur. On peut ainsi éviter les temps d'attente (*Wait State*). Une évolution des SDRAM est la DDR SDRAM (ou SDRAM II) qui double le taux de transfert actuel des SDRAM. La RDRAM, produit de la société Rambus, est de conception très nouvelle et originale. Elle permet des taux de transfert 10 fois plus élevés que ceux des DRAM traditionnelles, mais nécessite des contrôleurs spécifiques ce qui en limite l'usage. Intel s'oriente vers le développement de DRDRAM qui permettrait d'atteindre des débits de 1,6 Go/s.

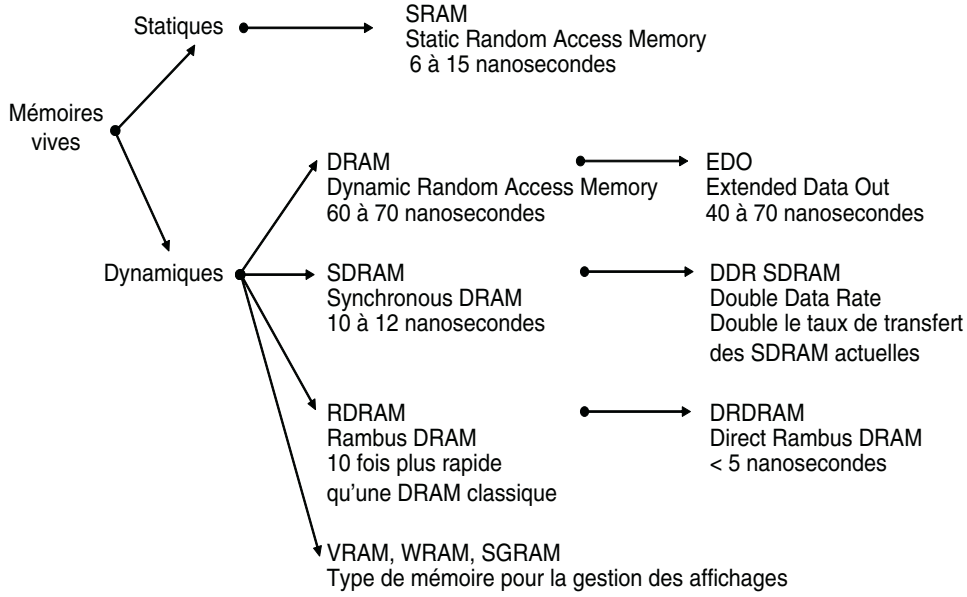


Figure 8.2 Les différents types de mémoires vives.

Le tableau 8.2 résume quelques caractéristiques de ces mémoires.

Tableau 8.2 EXPRESSIONS DE LA CAPACITÉ MÉMOIRE.

Type	Broches	Largeur (bits)	Fréquence (Mhz)	Bande passante	Commentaires
SDRAM	168	64	100	800 Mo/s	Type un peu ancien et lent. On trouve également des SDRAM à 133 Mhz pour une largeur de bande de 1 064 Mo/s
DDR	184	64	200	1 600 Mo/s	Des évolutions vers des fréquences de 266 et 333 Mhz pour des largeurs pouvant atteindre 128 bits et débits allant de 2 128 à 2 664 Mo/s.
RDRAM	184	16	800	1 600 Mo/s	Des évolutions sur la largeur du bus (de 16 à 32 bits) et des fréquences plus fortes permettraient des largeurs de bandes allant jusqu'à 4 264 Mo/s

Les capacités des mémoires sont en fortes évolutions et l'on peut prévoir dans les années à venir des capacités de plusieurs giga-octets équivalentes aux capacités actuelles de disques magnétiques.

Brochage et fonctionnement

Il existe plusieurs manières de réaliser une mémoire d'une capacité déterminée. Soit par exemple une mémoire d'une capacité de 8 Ko soit 64 kbits. Elle peut être réalisée selon le brochage de la figure 8.3.

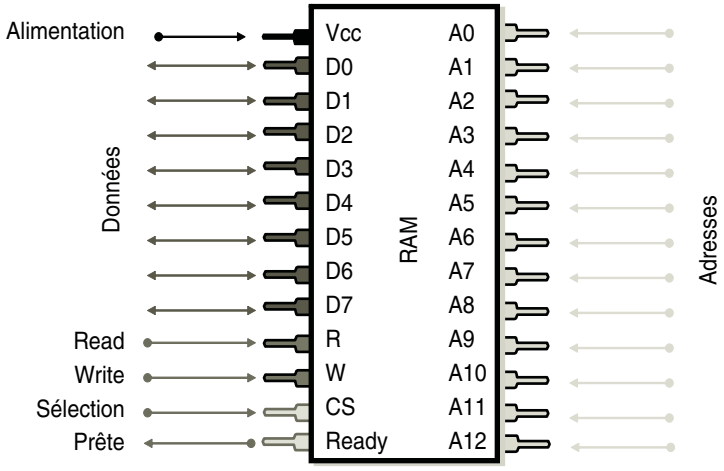


Figure 8.3 Brochage mémoire.

Les broches :

– A0 à A12 correspondent aux broches d'adressage (bus d'adresses) ;
– D0 à D7 correspondent aux données (bus de données) ;
– R, W correspondent au bus de commandes pour déclencher une lecture ou une écriture ;
– CS permet la sélection d'un boîtier de mémoire (puce mémoire) ;
– ready est un signal sortant vers le processeur permettant de synchroniser la fréquence d'horloge de la mémoire avec celle du processeur.

Cette puce mémoire a donc une capacité de 2^{13} mots de 8 bits (soit 2^{13} octets, ou 2^{16} bits).

Les opérations possibles sont la lecture ou l'écriture mémoire. C'est le processeur qui réalise ces opérations :

– Pour réaliser une écriture, le processeur :
  1. sélectionne une adresse mémoire et dépose l'adresse sur le bus d'adresses ;
  2. dépose la donnée sur le bus de données ;
  3. active la commande d'écriture.
– Pour réaliser une lecture, le processeur :
  1. sélectionne une adresse mémoire et dépose l'adresse sur le bus d'adresses ;
  2. active la commande de lecture ;
  3. lit la donnée sur le bus de données.

On voit donc que cette organisation permet l'adressage d'un octet et le déclenchement d'une opération de lecture ou d'écriture en un seul cycle mémoire. On peut maintenant organiser cette mémoire sous la forme d'une matrice de lignes et colonnes de bits (ici 8 lignes et 8 colonnes, soit $2^8 \times 2^8 = 2^{16}$ bits). Cela correspond au brochage de la figure 8.4.

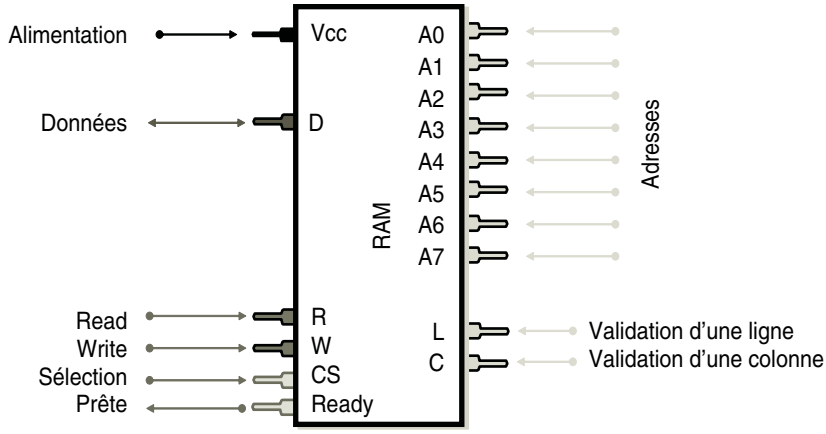


Figure 8.4 Organisation de la mémoire sous forme d'une matrice.

Les broches A0 à A7 permettent d'adresser soit une ligne soit une colonne, D correspond à la donnée sélectionnée, L et C permettent de valider la ligne et la colonne sélectionnée.

Le fonctionnement ne sera donc pas le même que dans le cas précédent :

– Pour réaliser une lecture, le processeur :
  1. sélectionne une ligne en déposant une adresse sur le bus d'adresses ;
  2. valide la ligne en activant L ;
  3. sélectionne une colonne en déposant une adresse sur le bus d'adresses ;
  4. valide la colonne en activant C ;
  5. active la commande de lecture ;
  6. lit la donnée sur le bus de données c'est-à-dire un bit.
– La réalisation d'une opération d'écriture s'entend facilement à partir de l'exemple de la lecture.

La deuxième organisation minimise le nombre de broches pour la réalisation de ce type de boîtier. Par contre les opérations de lecture et d'écriture sont plus longues à réaliser qu'avec la première organisation. Les mémoires de grandes capacités sont souvent construites sous la forme d'une matrice de lignes et de colonnes afin de diminuer les coûts de réalisation.

La figure 8.5 fournit un schéma synthétique reliant le boîtier processeur aux boîtiers mémoire centrale.

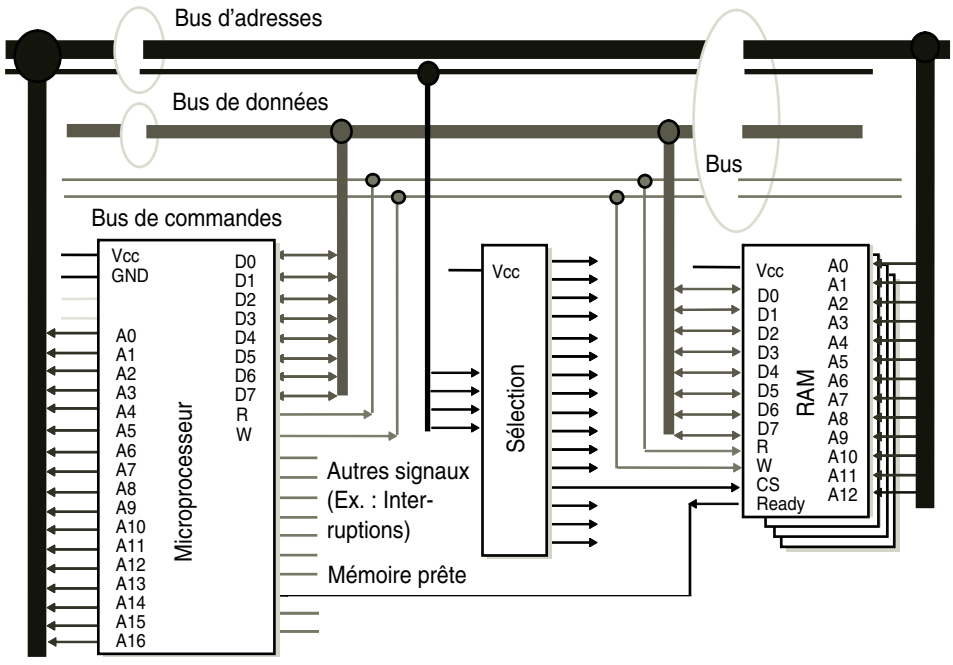


Figure 8.5 Liaison processeur – mémoire.

Le processeur de cette figure a une capacité d'adressage de 2^{17} mots de 8 bits. Chaque boîtier mémoire a une capacité de 2^{13} mots de 8 bits (2^{13} octets). Le processeur peut donc adresser 16 boîtiers de mémoire. Une adresse mémoire correspond donc à un numéro de boîtier et un numéro de mot dans le boîtier. Pour cela 4 bits du bus d'adresses sont connectés à un circuit spécial permettant la sélection d'un boîtier, les 13 autres bits du bus permettent l'adressage d'un mot du boîtier sélectionné. Le signal « Mémoire prête » permet la synchronisation entre la mémoire et le processeur.

Présentation physique

En général la mémoire se présente sous la forme de « puces » mémoire (boîtiers) dont la capacité moyenne est de 32 Mo et qui sont généralement regroupées sur des modules. Ces modules sont vendus selon 4 formats standards de 64, 128, 256 et 512 Mo. Ils se présentent sous la forme de barrettes SIMM (*Single Inline Memory Module*) ou DIMM (*Double Inline Memory Module*) qui offrent deux fois plus de connecteurs que la SIMM. Un nouveau format voit actuellement le jour, les barrettes RIMM (*Rambus Inline Memory Module*) qui correspondent aux RDRAM. Ce format permet la lecture des données en série et non plus en parallèle et est plus économique à fabriquer.

Les aspects internes : circuits et fonctionnement

La mémoire centrale est construite autour d'un ensemble de bits réalisés à partir de circuits permettant de coder une information sous forme binaire, que l'on note 0 ou 1. C'est la plus petite unité de stockage.

Elle est organisée comme un ensemble de cellules contenant l'information. Chaque cellule contient le même nombre de bits et est adressable : elle est identifiée par un numéro qui permet de la référencer dans les opérations de lecture/écriture mémoire. La taille des cellules est très variable en fonction des mémoires mais les adresses des cellules sont toujours consécutives. La cellule mémoire correspond à la plus petite quantité de mémoire adressable. Depuis quelques années les fabricants semblent se mettre d'accord sur des cellules d'une taille de 8 bits (octet ou *byte*). Un *mot mémoire* est constitué par un ensemble d'octets et correspond à l'information manipulée par les instructions machines. Les mots sont de tailles variables, par exemple une machine 32 bits dispose d'instructions manipulant des mots de 32 bits et est donc construite avec des registres de 32 bits.

La mémoire centrale dialogue avec le processeur pour échanger (lecture/écriture) des informations. La figure 8.6 illustre l'organisation d'une mémoire RAM. Il s'agit ici d'une mémoire conventionnelle rapide où l'on sélectionne une ligne (une cellule) et l'on accède en une seule opération au contenu de cette cellule.

Une opération d'écriture est, dans ce cas, réalisée en trois phases : (1) dépôt de l'adresse de la cellule sur le bus d'adresses, (2) dépôt du contenu à écrire sur le bus de données et (3) dépôt d'une commande d'écriture sur le bus de commandes.

Le contenu du bus d'adresses est placé en entrée d'un circuit de sélection (décodeur d'adresses) permettant de sélectionner la cellule mémoire correspondant à la

valeur déposée sur le bus d'adresses. Le contenu du bus de données est placé en entrée de circuits d'échanges qui servent de tampons entre le bus de données et la cellule mémoire sélectionnée.

Le cas d'une lecture se déduit facilement de celui d'une écriture (déposer une adresse, commander une lecture, lire le contenu du bus de données).

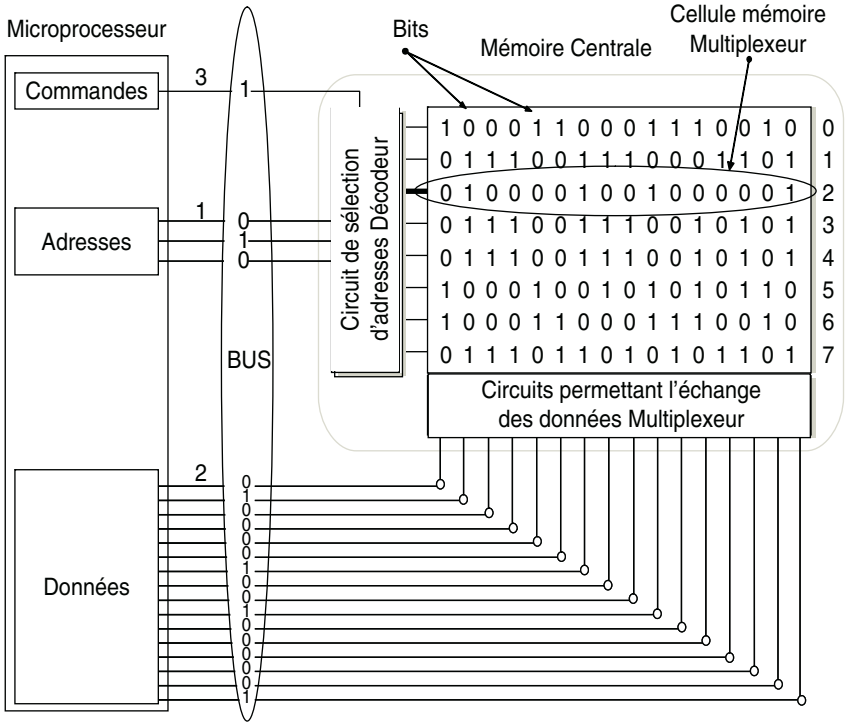


Figure 8.6 Organisation d'une mémoire RAM.

Les circuits fondamentaux qui interviennent dans la réalisation d'une mémoire sont donc le circuit « bit », le décodeur d'adresses, les circuits d'échanges.

Le bit (figure 8.7) est un circuit présentant deux états stables, codés 0 et 1, dont on peut décrire la logique de fonctionnement par :

```
début
si C est actif
alors S = E;
sinon S = S;
fsi
fin
```

La valeur du signal de sortie S (0 ou 1) dépend donc du signal de commande C. Tant que C n'est pas actif la valeur du signal de sortie est inchangée quelle que soit

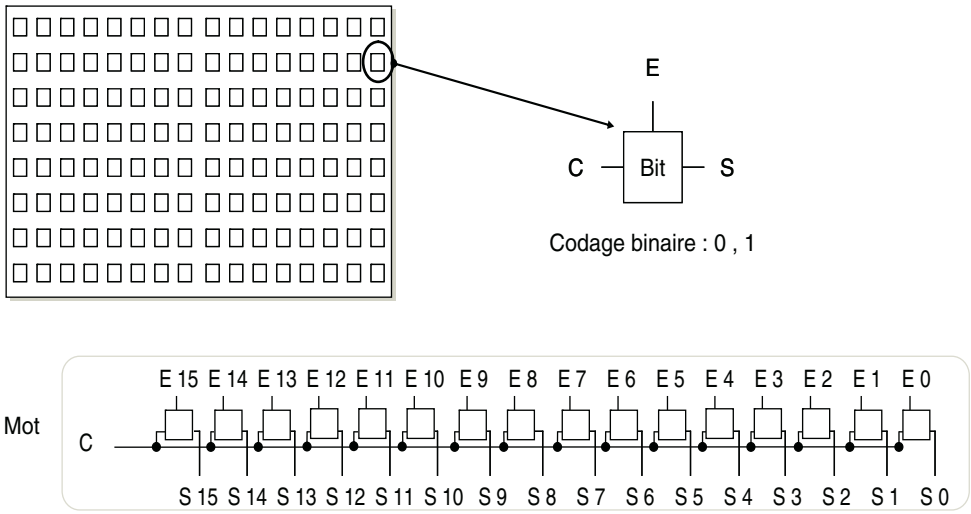


Figure 8.7 Un bit de mémorisation.

la valeur du signal d'entrée E. La valeur du signal de sortie est modifiée et prend la valeur du signal d'entrée dès que C est actif.

La réalisation matérielle d'un tel circuit peut se faire à l'aide :

– d'un circuit de type bascule (voir chapitre 4 sur les circuits logiques). Ce type de circuit (figure 8.8) répond bien à l'algorithme décrivant le fonctionnement d'un bit. On vérifie que si S est à l'état 0 et que C est également à l'état 0 la valeur de S est maintenue à 0 quelle que soit la valeur de E. Si S et C valent 0, tandis que E vaut 1, la sortie du circuit ET piloté par E a alors sa sortie à 0, les deux entrées du circuit OR exclusif du haut valent 0 et donc la sortie vaut 1. Cette sortie devient une entrée du OR exclusif du bas dont les deux entrées sont donc 0 et 1 ce qui

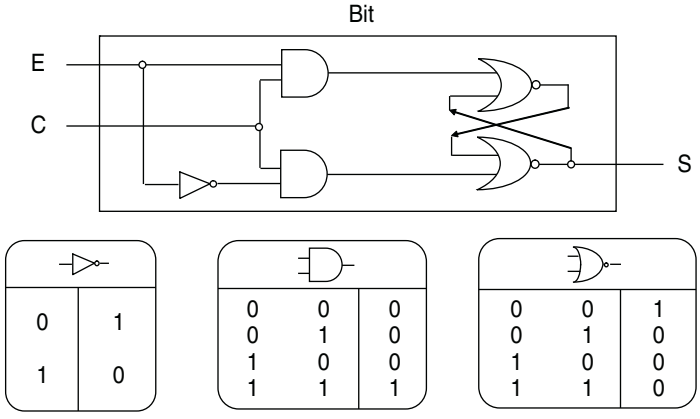


Figure 8.8 Utilisation d'une bascule pour réaliser un bit de mémorisation.

donne une sortie à 0. L'état du bit est donc conservé. La caractéristique de tels circuits est de garder la mémorisation aussi longtemps que l'alimentation électrique est maintenue. Les mémoires SRAM sont conçues à partir de circuits de cette nature ;

– d'une association d'un transistor et d'un petit condensateur. Si le condensateur présente une charge électrique, alors la valeur codée du bit vaut 1 sinon elle vaut 0. Malheureusement les condensateurs ne gardent pas une charge constante, elle diminue avec le temps, il est donc nécessaire de rafraîchir ce circuit pour maintenir le bit à une valeur stable. Cette caractéristique complexifie la réalisation de telles mémoires. Cependant un bit ne nécessite ici qu'un transistor pour sa réalisation alors qu'il en faut plusieurs pour réaliser une bascule, d'où une beaucoup plus grande densité d'intégration. Les mémoires DRAM sont réalisées à partir de tels circuits. Leur défaut essentiel est le temps d'accès qui est important.

Dans nos ordinateurs nous trouvons donc les deux types de mémoires, SRAM plutôt pour les mémoires caches, et DRAM plutôt pour les mémoires centrales.

La figure 8.9 est une illustration de la réalisation d'une mémoire à partir de bascules D. On y trouve les bascules supportant les bits, les circuits de décodage d'adresses, de sélection d'un mot mémoire, d'échanges et les barrières d'entrée et de

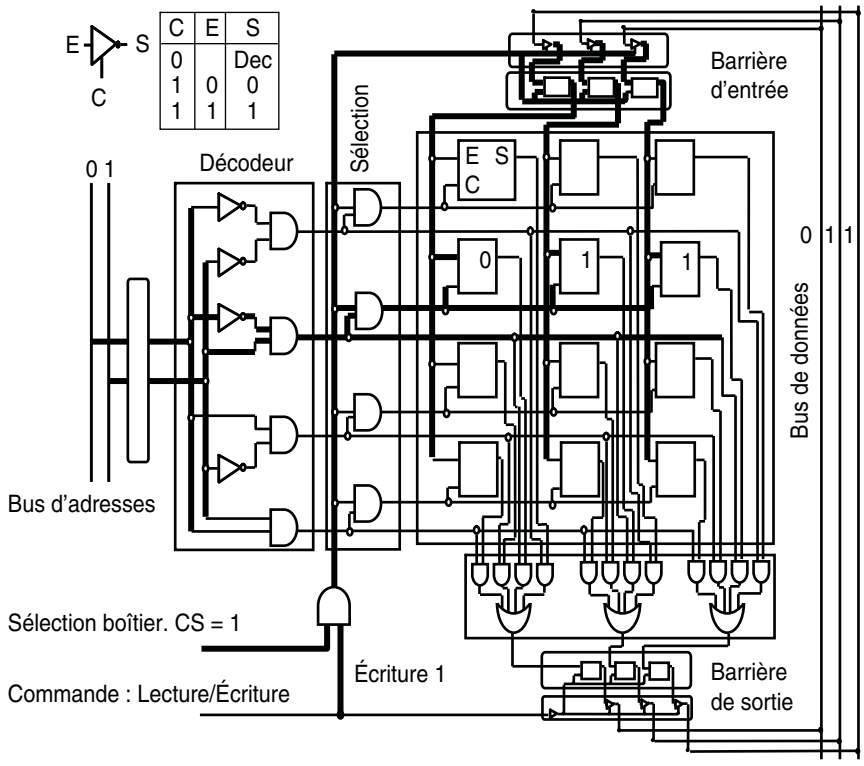


Figure 8.9 Réalisation d'une mémoire à partir de bascules D.

sortie. Il s'agit d'une mémoire de 12 bits organisée en mots de 3 bits, respectivement d'adresse 0, 1, 2, 3 (à partir du haut du schéma). Le bus de commandes transporte le signal de commande (1 pour une écriture, 0 pour une lecture). Le bus d'adresses peut adresser les 4 mots et le bus de données transporte 3 bits de données.

Pour écrire dans le mot 1 de ce boîtier mémoire, le processeur :

– active CS à 1 ;
– dépose la valeur d'adresse binaire « 01 » sur le bus d'adresses. La ligne 1 du décodeur est activée et prend la valeur 1 ;
– place 1 sur le bus de commande. Le mot d'adresse 1 est alors sélectionné. Le signal C de chaque bit du mot 1 est activé, donc les signaux S de ces bits prennent la valeur des signaux E. Les signaux C des autres mots sont positionnés à 0. La barrière de sortie est fermée (le contenu actuel du mot d'adresse 1 n'affecte pas le bus de données). La barrière d'entrée est ouverte : les signaux E du mot d'adresse 1 prennent la valeur des signaux du bus de données. L'écriture mémoire est terminée.

L'opération de lecture est très semblable. On notera l'importance des barrières d'entrée et de sortie qui dans le cas d'une écriture (barrière de sortie) empêche que les valeurs actuelles des bits ne viennent perturber les valeurs transportées par le bus de données, et dans le cas d'une lecture (barrière d'entrée) empêche que les données déposées sur le bus ne viennent perturber les signaux d'entrée de la mémoire.

8.2.2 Les mémoires mortes

Les mémoires vives sont accessibles en lecture et écriture mais sont volatiles. Bien des applications (programme et données) impliquent d'êtres stockées de manière permanente, même en l'absence d'alimentation électrique, comme par exemple le programme de « boot » d'un ordinateur. Ces mémoires, accessibles uniquement en lecture, sont connues sous le nom de ROM (*Read Only Memory*).

Les différentes évolutions technologiques de ce type de mémoire sont résumées dans la figure 8.10.

La caractéristique essentielle de ce type de mémoire est un temps d'accès très faible, sauf peut-être pour les mémoires flash (environ 100 nanosecondes). Comparativement aux disques magnétiques (temps d'accès en millisecondes) les mémoires flash sont très performantes mais ont, pour le moment, des durées de vie trop faibles pour remplacer les disques magnétiques.

8.2.3 Les registres

On peut réaliser des registres à partir de circuits flip-flops ou des bascules comme le montre la figure 8.11. La broche d'horloge CK alimente toutes les entrées CK des bascules composant le registre. La broche CLR (Clear) est également commune à toutes les bascules du registre et permet de remettre à 0 le contenu de chaque bascule.

Le brochage d'un registre de 16 bits nécessite donc, dans cette architecture, 36 broches. Comme nous l'avons vu pour les mémoires de grande capacité ce n'est pas ce type de montage qui est réalisé.

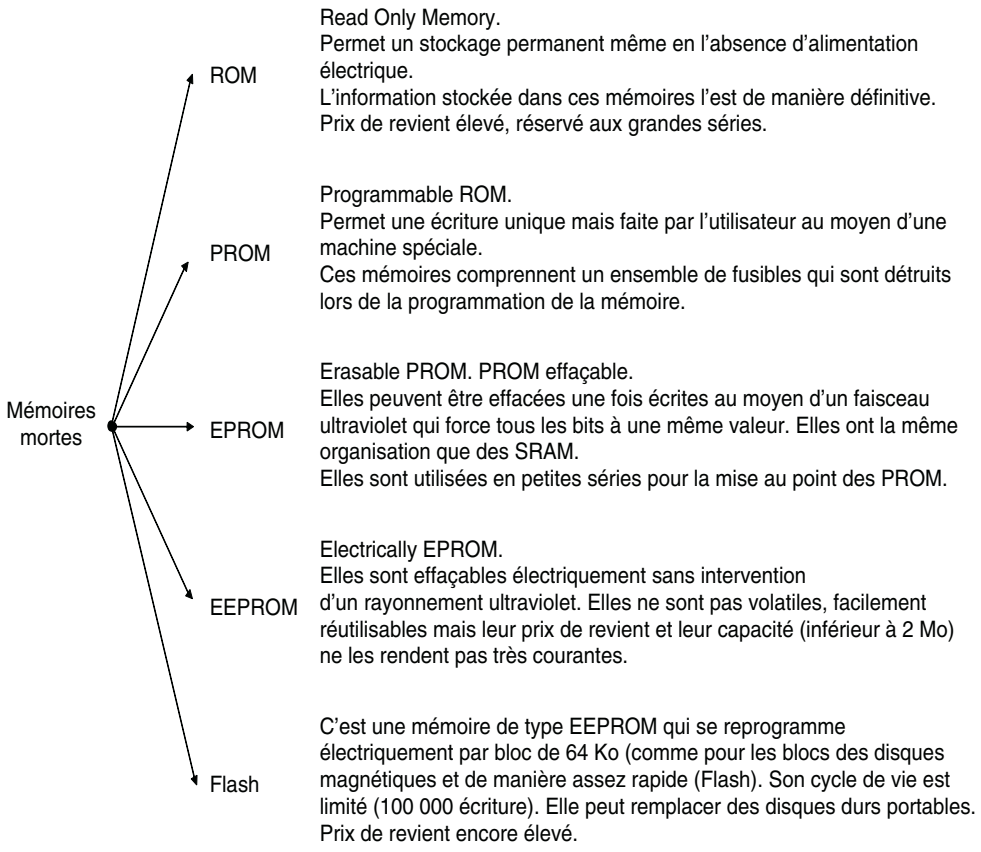


Figure 8.10 Les différents types de mémoire ROM.

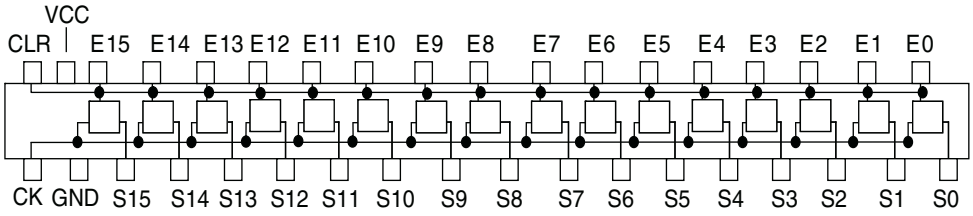


Figure 8.11 Réalisation d'un registre.

8.3 MÉMOIRES DE STOCKAGE : LE DISQUE MAGNÉTIQUE

Le disque magnétique est présenté ici comme le « représentant » des mémoires de masse dans la hiérarchie de mémoires. Il existe plusieurs autres supports de stockage de masse et nous examinerons leurs caractéristiques essentielles dans le chapitre consacré à la fonction de communication.

8.3.1 Caractéristiques générales

Un disque magnétique est constitué d'un ou plusieurs *plateaux* métalliques recouverts sur chaque face d'une matière magnétisable. La tête de lecture/écriture est constituée d'un bobinage fer/nickel (figure 8.12) qui flotte à une faible distance du matériau magnétisable. Lorsqu'un courant électrique traverse le bobinage, une magnétisation de la surface se produit.

Cette magnétisation oriente les particules du matériau. L'orientation des particules dépend du sens du courant électrique. C'est ainsi que l'on réalise une écriture, l'orientation des particules définissant la valeur à attribuer à la partie magnétisée (0 ou 1). Lorsque la tête passe au-dessus d'une zone magnétisée un champ électrique est induit dans la bobine. Le sens du champ électrique induit dépend du type de magnétisation. On peut ainsi en fonction du sens du champ électrique savoir la valeur de l'information que l'on vient de lire.

8.3.2 Organisation générale

Lors de son fonctionnement le disque tourne à une vitesse constante sous la tête de lecture.

Sur chaque plateau se trouvent les *pistes*, zones circulaires sur lesquelles sont enregistrées les séquences de bits codant les informations stockées. Chaque piste est divisée en secteurs. Un *secteur* débute par une zone de préambule qui permet de se synchroniser avant de faire une opération de lecture ou d'écriture. De même le secteur se termine par un ensemble de bits permettant d'effectuer des contrôles d'erreurs sur les informations stockées dans le secteur. Entre deux secteurs se trouve une zone ne

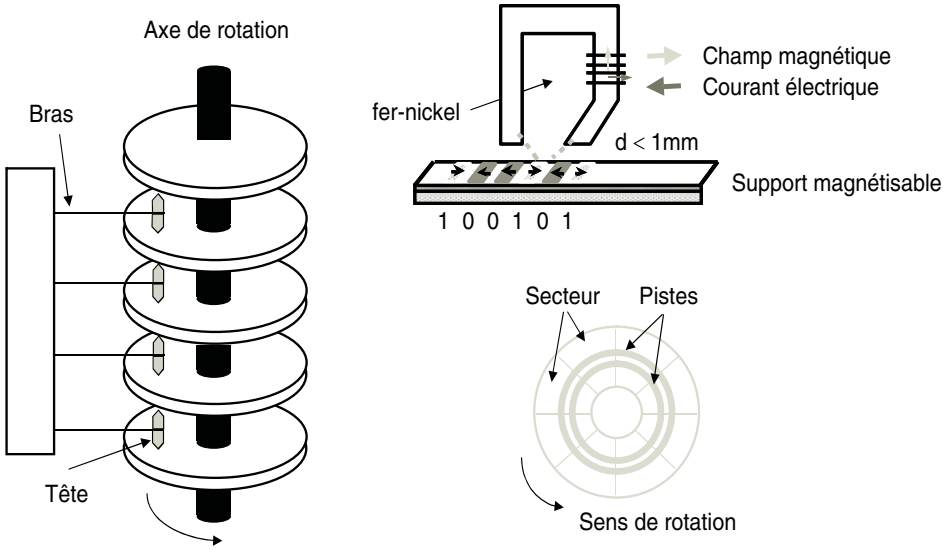


Figure 8.12 Structure d'un disque magnétique.

contenant pas de bits, constituant le *gap inter secteur*. L'opération de *formatage* consiste à organiser le disque en pistes et secteurs. Ainsi la capacité de stockage d'un disque formaté est inférieure à la capacité d'un disque non formaté (figure 8.12).

Chaque piste a une largeur de quelques microns (un micron = un millième de mm). Un bit sur une piste a une largeur de quelques dizaines de microns. Cette largeur dépend essentiellement de la largeur de la tête de lecture. Sur les disques courants on dénombre de 800 à 2 000 pistes par centimètre.

On appelle *densité d'enregistrement* le nombre de bits par millimètre. Cette densité est variable selon la qualité des disques et peut varier entre 50 000 et 100 000 bits par centimètre.

Les pistes sont des cercles concentriques et donc la longueur des pistes n'est pas constante, elle dépend de l'éloignement de la piste par rapport au centre du plateau. Ainsi les secteurs ne sont pas de longueur constante ce qui pose un problème si l'on veut que chaque secteur ait une capacité constante (chaque secteur contient le même nombre de bits). Pour remédier à cela on peut avoir une densité d'enregistrement qui dépend de la piste : elle est plus faible pour les pistes les plus proches du bord, plus forte pour les pistes proches du centre. Ainsi lorsqu'on parle de densité d'enregistrement on parle de densité moyenne correspondant à la densité d'enregistrement des pistes médianes. Une autre solution consiste à avoir un nombre de secteurs par piste variable en fonction de la piste : les pistes externes contiennent plus de secteurs que les pistes proches du centre. Cette technique plus complexe à implanter donne une plus grande capacité de stockage.

Les secteurs ont le plus fréquemment des capacités de stockage de 512 ou 1 024 octets. Les têtes de lecture/écriture sont portées par des bras qui se déplacent radialement de l'intérieur vers l'extérieur et inversement. Entre chaque piste existe un espace inter piste. Un disque magnétique est généralement constitué de plusieurs plateaux ayant tous le même nombre de pistes. On appelle *cylindre* l'ensemble des pistes situées à la même distance de l'axe de rotation du disque.

Adresse d'une donnée

En mémoire centrale une information est repérée par l'adresse du mot la contenant. Sur un disque magnétique l'adresse d'une information est un triplet : numéro de cylindre, numéro de bras, numéro de secteur. Lire une donnée consiste donc à trouver l'adresse de début d'un secteur et à déclencher la lecture du secteur. On trouve ensuite la donnée précise en analysant le contenu du secteur. C'est le système d'exploitation au travers du système de gestion des fichiers (SGF) qui gèrent les adresses physiques des informations stockées.

Temps d'accès à une information

Trouver une information consiste d'abord à positionner le bon bras sur le bon cylindre : c'est l'opération de déplacement du bras caractérisée par le temps de déplacement du bras. La piste adéquate est alors sélectionnée. Il faut ensuite trouver le bon secteur. Le temps de positionnement de la tête sur le bon secteur est variable : soit le début

du secteur cherché est juste sous la tête et le temps de positionnement est nul, soit le début du secteur vient de passer sous la tête et il faut faire un tour de disque complet pour retrouver le début du secteur cherché. On parle donc généralement de temps moyen de positionnement de la tête, ce temps moyen correspond à un demi-tour de disque. Le *temps de latence rotationnel* mesure le délai pour que la tête de lecture soit placée sur le début du bon secteur (figure 8.13).

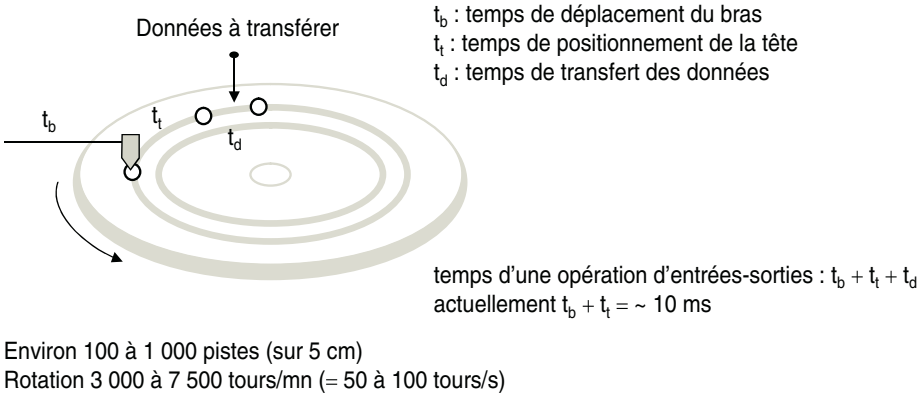


Figure 8.13 Temps d'accès à une information.

Enfin, il faut lire (écrire) le contenu du secteur. Le temps de lecture (écriture) est caractérisé par la vitesse de transfert de l'information ou *débit*. Cette vitesse dépend de la densité d'enregistrement et de la vitesse de rotation du disque. Les vitesses classiques de rotation sont de 5 400 et 7 200 tours/minute. Les débits types sont de l'ordre de 5 à 20 Mo/s. Ce sont donc les temps de déplacement et de positionnement de la tête qui pénalisent le plus les performances d'un disque magnétique.

Pour avoir de bonnes performances il apparaît clairement qu'il ne faut pas que les informations soient stockées dans des secteurs aléatoires du disque. Il faut essayer de regrouper les données dans des secteurs contigus. L'usage du disque induit nécessairement une fragmentation qui correspond au fait que l'information est distribuée aléatoirement sur le disque. L'opération de défragmentation consiste à utiliser un logiciel permettant de réorganiser le disque de manière à ce que les données soient placées à des adresses contiguës.

8.4 AMÉLIORATION DES PERFORMANCES

8.4.1 Les mémoires caches

Les performances (vitesse) des processeurs augmentent sans cesse. Les performances des mémoires augmentent également mais surtout en matière de capacité de stockage et beaucoup moins en ce qui concerne les temps d'accès. La fréquence d'un processeur est aujourd'hui facilement de l'ordre de 1 GHz (1 000 MHz) ce qui signifie qu'il

peut engendrer des actions 10^9 fois par seconde. Une mémoire DRAM dont le temps d'accès est de 20 nanosecondes pourra fournir 5×10^7 informations par seconde. L'écart est donc considérable et le processeur ne fonctionne pas au meilleur rythme. Les mémoires SRAM ont une densité d'intégration trop faible et un coût trop élevé pour constituer des mémoires centrales de grandes capacités.

Dans une architecture de machine de type Von Neumann (qui correspond à nos machines) les échanges, entre le processeur et la mémoire, sont très nombreux puisqu'un programme et ses données doivent êtres placés en mémoire centrale afin d'être exécutés par le processeur. On sait techniquement réaliser des mémoires très rapides, qui ne pénaliseraient pas le processeur, mais à condition qu'elles soient placées très près du processeur. En effet c'est souvent le temps de transfert par le bus qui est un facteur de ralentissement.

On peut envisager plusieurs solutions pour améliorer les performances :

– augmenter la fréquence des mémoires, ce qui améliorerait la bande passante. On constate cependant que cette évolution est lente ;
– augmenter la largeur du bus, ce qui augmenterait aussi la bande passante mais qui n'est pas facile à réaliser, à cause en particulier de l'encombrement ;
– réduire les besoins en bande passante pour le processeur. C'est ce qui est fait avec l'utilisation des mémoires cache (*antémémoire*).

Principe de fonctionnement

Le principe est de faire coopérer des mémoires de faible capacité, très rapides et à proximité du processeur avec des mémoires plus lentes et de grandes capacités (figure 8.14). Les mots de mémoire centrale les plus fréquemment utilisés sont placés dans le cache. Le processeur cherche d'abord le mot dans le cache, s'il est présent il l'obtient rapidement. Si le mot n'est pas présent, le processeur fait un accès à la mémoire centrale (plus lente) et place ce mot dans le cache. Ultérieurement si ce mot est demandé il sera obtenu plus rapidement.

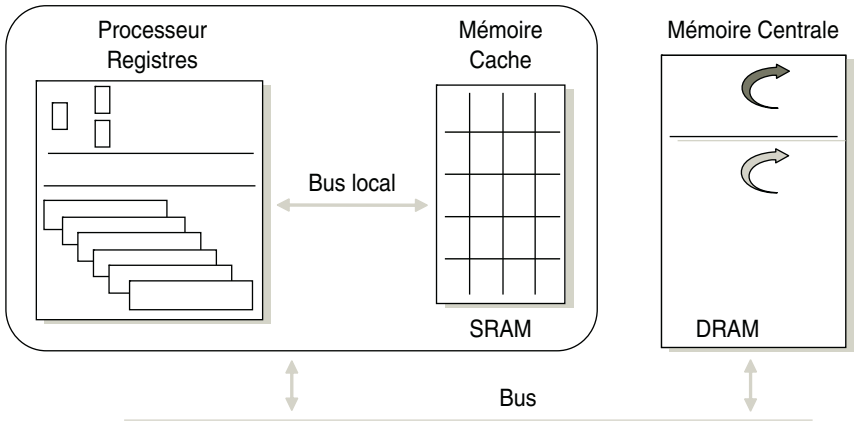


Figure 8.14 Hiérarchie de mémoires.

Il faut distinguer la lecture et l'écriture d'un mot. Le principe de fonctionnement pour la lecture d'un mot est décrit par l'algorithme :

```
si mot présent
alors charger processeur avec le mot;
sinon
    si cache plein
    alors charger cache(remplacer);
          charger processeur;
    sinon
          charger cache;
          charger processeur;
finsi
```

Le processeur cherche d'abord si le mot mémoire adressé est dans le cache (`mot présent`). Si l'information est présente on parle de succès (*cache hit*). Le cache est de petite taille et ne contient pas toute l'information de la mémoire centrale : l'information peut donc ne pas être dans le cache, on parle alors d'échec (*cache miss*). Dans ce dernier cas, il faut aller chercher l'information dans la mémoire centrale et placer celle-ci dans le cache (`charger cache`) avec éventuellement un remplacement d'informations actuellement présentes dans le cache puisque le cache est de taille finie (`charger cache (remplacer)`). Enfin, le processeur est chargé avec l'information maintenant disponible dans le cache.

L'efficacité du cache dépend donc de son taux de succès et donc de la probabilité de présence dans le cache du mot cherché. Après un échec le cache est chargé avec le mot manquant, ainsi si ce mot est redemandé ultérieurement, il sera obtenu conformément au temps d'accès du cache (donc plus rapidement que s'il était en mémoire centrale).

Un programme ne s'exécute pas de manière aléatoire : de nombreuses études statistiques montrent que lorsqu'une instruction référence une adresse il est très probable que la référence mémoire qui suit soit dans le voisinage de cette adresse. En effet les programmes travaillent sur des informations organisées par des structures de données (exemple les tableaux) qui les placent dans des adresses mémoires très proches les unes des autres (ces adresses sont contiguës dans le cas des tableaux). Par ailleurs les instructions d'un programme sont placées dans des mots mémoires contigus et le flux d'exécution se fait séquentiellement (instruction après instruction) sauf dans le cas de branchements ou de déroutements.

Ainsi quand une instruction s'exécute, il est très probable que la prochaine instruction à exécuter soit dans le mot suivant de la mémoire : c'est le *principe de localité spatiale*. Ce principe nous conduit à penser que lorsqu'une donnée ou une instruction doit être chargée dans un cache il serait intéressant de charger également les données (et/ou instructions) qui sont proches en mémoire. On augmente ainsi la probabilité que le processeur trouve la prochaine donnée (ou instruction) dans le cache.

Les programmes font beaucoup d'itérations et les données qui y sont traitées sont très souvent utilisées dans de courts laps de temps (une variable qui pilote une boucle

sera très souvent utilisée et il est intéressant qu'elle soit placée dans le cache). Aussi, lorsque le processeur référence un mot donné, il existe une très forte probabilité que le processeur référence de nouveau ce mot dans les instants qui suivent. C'est le *principe de localité temporelle.*

Ces deux principes de localité sont à la base des systèmes utilisant les caches.

Si Teff est le temps effectif pour accéder à une information, h la probabilité de présence de l'information dans le cache, Tc le temps d'accès au cache et Tm le temps d'accès à la mémoire centrale, le temps d'accès effectif s'exprime par :

$$\text{Teff} = h \times Tc + (1 - h) \times Tm$$

Si par exemple Tm/Tc = 10 (le temps d'accès du cache est dix fois plus petit que celui de la mémoire centrale) on montre facilement qu'une variation de 1 % de h entraîne une variation de 10 % de Teff.

Ce résultat montre qu'il faut soigneusement organiser les caches afin d'augmenter la probabilité de présence d'une information dans le cache et ainsi diminuer le temps effectif d'accès à l'information.

Lorsque le processeur doit écrire un résultat il doit accéder à la mémoire cache pour vérifier si l'information est présente dans le cache et éventuellement la modifier. La mémoire cache ne contient pas toutes les informations de la mémoire centrale mais cette dernière contient toutes les informations du cache et il faut maintenir la cohérence des informations entre le cache et la mémoire principale. Une écriture dans le cache modifiant une information doit donc entraîner la modification de cette information dans la mémoire centrale.

L'algorithme suivant décrit le fonctionnement d'une écriture :

```
si mot présent
alors
    modifier cache;
    modifier mémoire principale;
sinon
    modifier mémoire principale;
finsi
```

Quand on doit modifier le cache il faut modifier la mémoire principale. Il existe plusieurs techniques pour réaliser cette modification :

– écriture immédiate (*Write through*). On écrit simultanément dans le cache et la mémoire principale. On garantit donc la cohérence ;
– écriture différée (*Write back*). Il y a là encore plusieurs techniques. On peut mettre à jour la mémoire centrale quand l'information de la mémoire cache doit être remplacée ou dès que le bus de communication est libre. Dans ces techniques on ne garantit pas en permanence la cohérence, mais le temps d'écriture est plus faible.

Organisation et fonctionnement : généralités

Le principe de localité suggère de considérer la mémoire centrale comme une suite de blocs mémoires : les lignes de mémoires. Dans notre exemple (figure 8.15) la

mémoire centrale est organisée en ligne de 4 mots. L'adresse du premier mot de la première ligne est 000, l'adresse du premier mot de la deuxième ligne est 004. L'adresse de « c » est 022 que l'on peut voir comme le numéro de ligne 020 et un déplacement (*offset*) de 2 mots.

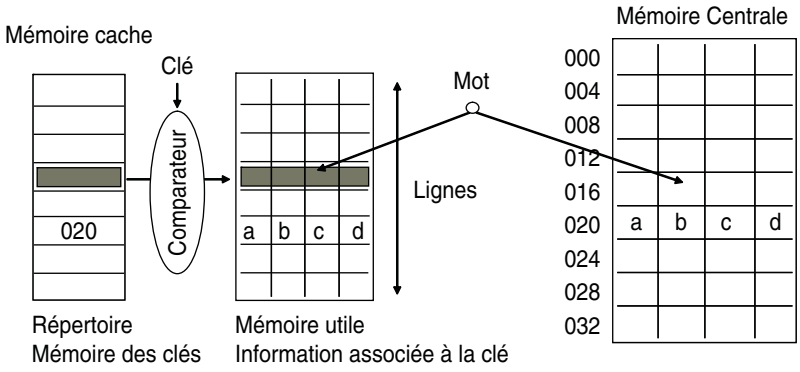


Figure 8.15 Organisation de la mémoire centrale en lignes mémoires.

Fonctionnellement la mémoire cache est organisée autour d'un répertoire de clés, de comparateurs et de la mémoire utile qui contient l'information. La mémoire utile comprend un certain nombre de lignes de même taille que les lignes de la mémoire centrale : ce sont les lignes de cache. Le répertoire des clés a autant d'entrées qu'il y a de lignes de cache, une clé est un numéro de ligne de mémoire. Lorsqu'un numéro de ligne est présent dans le répertoire alors la ligne de cache associée à la clé contient les valeurs de la ligne de mémoire correspondante. Dans l'exemple puisque 020 est dans le répertoire alors les mots mémoire de la ligne 020 sont dans le cache.

Lors d'une opération de lecture, par exemple le processeur veut acquérir « c » c'est-à-dire le contenu du mot d'adresse 022, le processeur doit vérifier si « c » est présent dans le cache. Pour cela l'adresse est présentée au(x) comparateur(s) qui vérifie si 020 est dans le répertoire. Si c'est le cas la donnée est présente dans le cache. Si ce n'est pas le cas le cache est chargé avec la ligne 020 et le répertoire est mis à jour. Charger une ligne et non pas uniquement la donnée cherchée permet de prendre en compte le principe de localité.

Il y a plusieurs manières de construire des caches qui correspondent à ce schéma fonctionnel et plusieurs facteurs sont déterminants pour une bonne conception d'un cache :

– la taille. Plus le cache est gros, plus il est efficace mais aussi plus coûteux ;
– la longueur des lignes de cache. Pour une taille de cache (mémoire utile) donnée il y a plusieurs manières de choisir le nombre et la longueur des lignes de cache ;
– le mode de gestion du cache. Les caches sont à accès très rapides mais l'information que l'on souhaite obtenir n'est pas nécessairement présente dans le cache (contrairement à ce qui se passe dans une mémoire centrale). Aussi pour obtenir une infor-

mation il faut d'abord vérifier si elle est présente. Pour que le cache soit efficace, l'organisation et la conception des caches doivent minimiser ce temps de vérification ;

– le nombre et la localisation du ou des caches.

Les différents types de cache

➤ Cache direct

Il s'agit du cache le plus simple, il est également appelé *cache à correspondance directe.* La figure 8.16, dans laquelle pour des raisons de simplicité d'écriture, nous adoptons une notation décimale, donne un exemple d'un tel cache. Ainsi la mémoire centrale a une capacité de 10 000 mots dont les adresses évoluent de 0000 à 9999. En notation décimale il suffit de 4 symboles pour représenter cet espace d'adressage. D'un point de vue fonctionnel la mémoire centrale est vue comme une suite de lignes composées chacune 10 mots.

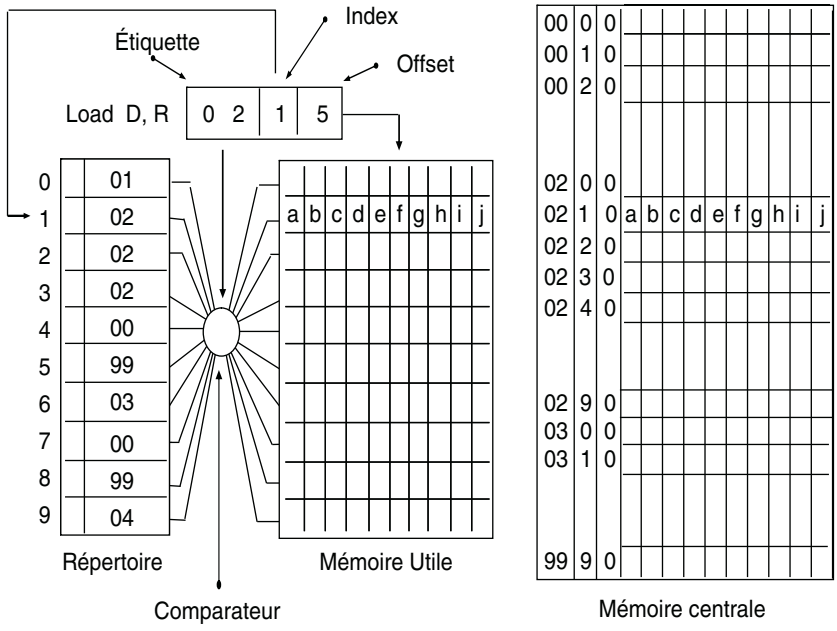


Figure 8.16 Cache à correspondance directe.

Le cache est organisé autour :

– de la mémoire utile. Elle contient les données. Chaque ligne a une longueur de 10 mots. Il y a 10 lignes ;

– un répertoire de 10 lignes. Chaque ligne comprend un bit de validité qui indique si des données valides sont disponibles dans cette ligne ainsi qu'une clé permettant d'identifier précisément une ligne ;

– un comparateur qui vérifie si la valeur d'étiquette de l'adresse est égale à la clé.

Chaque ligne du cache comprend donc un bit de validité, une clé et une ligne de données.

Lorsqu'une adresse est présentée au cache, le contrôleur de cache décompose cette adresse en trois parties : l'*index*, l'*étiquette* et l'*offset*. L'index repère une entrée du cache, tandis que l'offset repère un mot à l'intérieur d'une ligne de cache. Notre exemple comprend dix lignes de cache, il suffit donc d'un seul symbole pour référencer toutes les entrées du cache. Par ailleurs, comme il y a 10 mots par ligne, il suffit de même d'un seul symbole pour numéroter tous les mots d'une ligne de cache. Comme l'adresse est codée sur 4 symboles, il reste 2 symboles pour fixer la valeur de l'étiquette. Pour vérifier si une donnée est dans le cache le contrôleur de cache extrait la valeur d'index de l'adresse et le comparateur vérifie que l'entrée pointée par l'index contient bien l'étiquette. Si c'est vrai alors la donnée est présente dans la mémoire utile.

Le fonctionnement de ce cache est résumé par l'algorithme :

```
si répertoire[index] = étiquette
alors
    charger le processeur avec mémoire_utile[index, offset];
sinon
    sauvegarder mémoire_utile[index] dans la mémoire centrale;
    répertoire[index] = étiquette;
    charger mémoire_utile[index] à partir de la mémoire centrale;
    charger le processeur avec mémoire_utile[index, offset];
finsi
```

Supposons maintenant que le cache soit celui de l'exemple et que dans ces conditions le processeur exécute le programme suivant :

```
1. Load D, R, 0215
2. Load D, R, 0212
3. Load D, R, 0114
4. Load D, R, 0225
5. Load D, R, 0116
```

L'instruction 1 aboutit à un succès (hit) puisque `répertoire[index]` (02) vaut `etiquette` (02). Dans ce cas, l'accès à l'information est donc plus rapide que si l'information avait été dans la mémoire centrale. L'instruction 2 aboutit également à un succès. L'instruction 3 aboutit à un échec (miss). La ligne de mémoire centrale 011 remplace la ligne de cache indexée par 1 et la ligne de répertoire indexée par 1 reçoit 01 (étiquette). Dans ce cas, l'accès à l'information est plus lent que si l'on avait directement adressé la mémoire centrale. Mais comme on a chargé une ligne complète de mémoire le principe de localité nous indique qu'il y a toutes les chances que pour les prochaines instructions du programme aboutissent à des succès. De plus le temps de chargement d'une ligne en continue est plus petit que la répétition des chargements mot par mot. L'instruction 4 aboutit à un succès ainsi que l'instruction 5.

Le défaut principal de ce cache est qu'il peut exister des collisions : c'est ce qui s'est produit lors de l'exécution de l'instruction 3. Les adresses 001x, 021x, 031x, etc. produisent des collisions puisqu'elles ont toutes la même valeur d'index (1) et donc référencent la même ligne de cache. Un cas très défavorable est celui où l'on ne fait (à cause des collisions) que charger des lignes de caches, car les accès sont systématiquement en collision. Dans ce cas le cache serait plus lent que la mémoire centrale. Ce cache est simple, facile à réaliser et donne de bons résultats. C'est le cache le plus commun dans nos machines.

➤ Cache purement associatif

Ce type de cache est plus complexe et plus cher à construire. Il y a autant de comparateurs que de lignes de cache. Le chargement dans le cache d'une ligne de mémoire n'est pas fixé par une valeur d'index comme dans le cas du cache direct, c'est-à-dire qu'une ligne de données entre dans n'importe quelle entrée libre du cache. L'adresse est interprétée comme une étiquette et un offset. Le contrôleur de cache vérifie en une seule opération (c'est pourquoi il y a autant de comparateurs que de lignes) si une étiquette est présente dans une des lignes du répertoire. Si la réponse est un succès c'est que la donnée cherchée est dans la mémoire utile.

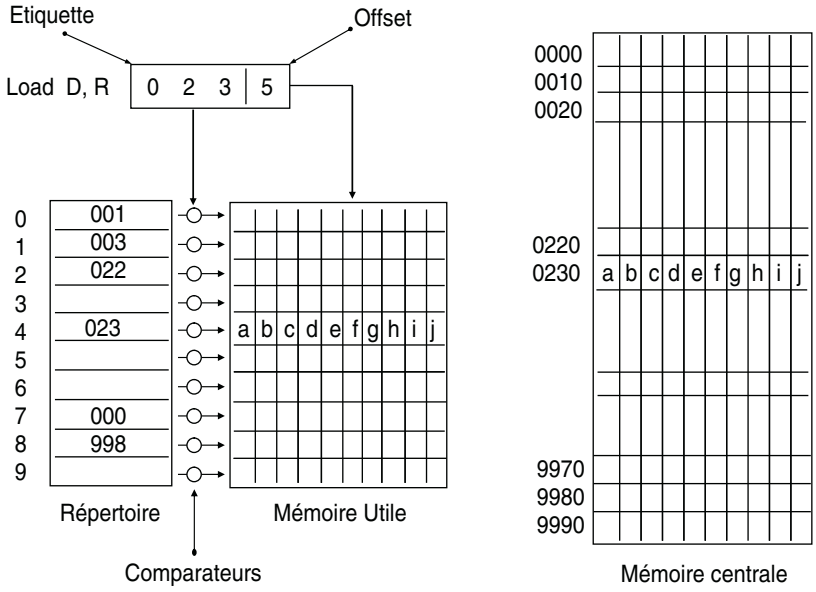


Figure 8.17 Cache purement associatif.

Ainsi sur la figure 8.17, la recherche dans le cache du mot d'adresse 0235 est un succès. L'étiquette 023 est présentée au répertoire du cache et la comparaison aboutit à délivrer la ligne 4 du cache comme étant celle contenant le mot cherché. L'offset 5 permet de récupérer effectivement le mot « f ».

Le principe de fonctionnement est décrit par l'algorithme :

```
si répertoire contient étiquette
alors
    information trouvée;
    charger le processeur;
sinon
    si répertoire plein
    alors
        algorithme de remplacement de ligne;
        remplir la ligne choisie;
        charger le processeur;
    sinon
        remplir une ligne libre;
        charger le processeur;
    finsi
finsi
```

Cette gestion est plus complexe que dans le cas du cache direct. Dans le cas d'un échec une ligne de mémoire doit être chargée dans le cache mais contrairement au cache direct on ne connaît pas le numéro de la ligne du cache dans laquelle l'information doit être placée. Il faut donc vérifier si le cache est plein, et dans ce cas il faut exécuter un algorithme permettant de choisir la ligne à remplacer. C'est le cas le plus défavorable et les performances de l'algorithme sont critiques pour que le cache soit efficace : ainsi il ne faut pas que l'on remplace la ligne qui va être utilisée immédiatement après son remplacement.

Algorithmes de remplacement de lignes

Il existe différents algorithmes de remplacement de lignes. Les principaux sont :

– FIFO (*First In, First Out*) : dans ce cas, la ligne remplacée est la ligne la plus anciennement chargée ;
– LRU (*Least Recently Used*) : dans ce cas, la ligne remplacée est la ligne la moins récemment accédée. Cette politique est meilleure que la précédente car elle tient compte des accès effectués par le processeur au cache, mais elle est coûteuse car nécessite de maintenir l'ordre des accès réalisés ;
– NMRU (*Not MostRecently Used*) : la ligne remplacée n'est pas la plus récemment utilisée. Dans cette politique, la ligne remplacée est une ligne choisie au hasard dans l'ensemble des lignes du cache, hormis la ligne la plus récemment accédée.

La dernière politique offre de bonnes performances et est couramment mise en œuvre dans les caches associatifs.

➤ Cache mixte

Le cache mixte (figure 8.18), utilisé dans certains de nos micro-ordinateurs, utilise les techniques des deux caches précédents. Le cache est divisé en blocs gérés comme

des caches directs et il existe un comparateur par bloc. Lorsqu'une adresse est présentée au cache, l'index référence simultanément une ligne par bloc et en une seule opération les comparateurs vérifient si l'étiquette est dans une des lignes. En cas d'échec, la ligne de mémoire correspondante doit être chargée dans une des lignes référencées. Il faut donc utiliser un algorithme de remplacement pour choisir laquelle des lignes remplacer, si celles-ci sont toutes occupées.

Le fonctionnement de ce type de cache est décrit par l'algorithme :

```
si répertoires[index] contient étiquette
alors
    charger memoire_utile[bloc, index, offset] dans le processeur;
sinon
    choisir bloc pour remplacer ligne;
    remplacer ligne dans bloc choisi;
    charger mémoire_utile[bloc choisi, index, offset] dans le
    processeur;
finsi
```

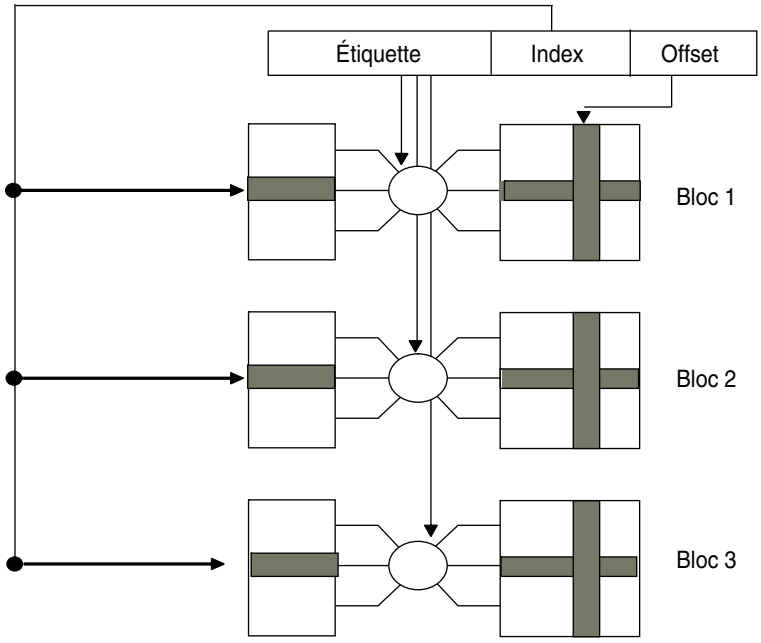


Figure 8.18 Cache mixte.

Nombre et localisation des caches

L'objet des caches est d'améliorer la bande passante de la mémoire afin d'augmenter la quantité d'informations transférée au processeur par unité de temps. Une des manières de réaliser cet objectif est d'augmenter la fréquence des mémoires (dimi-

nuer les temps d'accès) mais aussi d'améliorer la qualité des transferts entre processeur et mémoire. Dans cet esprit l'idée est de rapprocher le plus possible la mémoire du processeur afin de diminuer les délais d'acheminements, l'idéal étant de placer la mémoire dans la puce du processeur. Malheureusement les mémoires rapides ont un faible degré d'intégration et on ne peut pas loger des mémoires rapides de grande capacité dans la puce du processeur. Une hiérarchie de mémoires est donc mise en place sur trois niveaux :

– le premier niveau de mémoire cache, petite et très rapide, est placé dans le processeur ;
– le deuxième niveau, de capacité plus importante et d'accès également rapide, est mis à l'extérieur du processeur ;
– le troisième niveau est constitué par la mémoire centrale (figure 8.19).

Le premier niveau est désigné sous le vocable de *cache de niveau 1*, le second niveau constitue quant à lui le *cache de niveau 2*. Les caches de niveau 1 et 2 sont reliés par un bus local privé très rapide afin de diminuer les temps de transferts entre ces caches. Le processeur cherche d'abord la donnée ou l'instruction dans le cache de niveau 1, en cas d'échec l'information est cherchée dans le cache de niveau 2, enfin dans la mémoire centrale en cas d'échec au niveau 2. Un bon fonctionnement implique que toute l'information du cache 1 se retrouve dans le cache de niveau 2 et que l'information du cache de niveau 2 est dans la mémoire centrale.

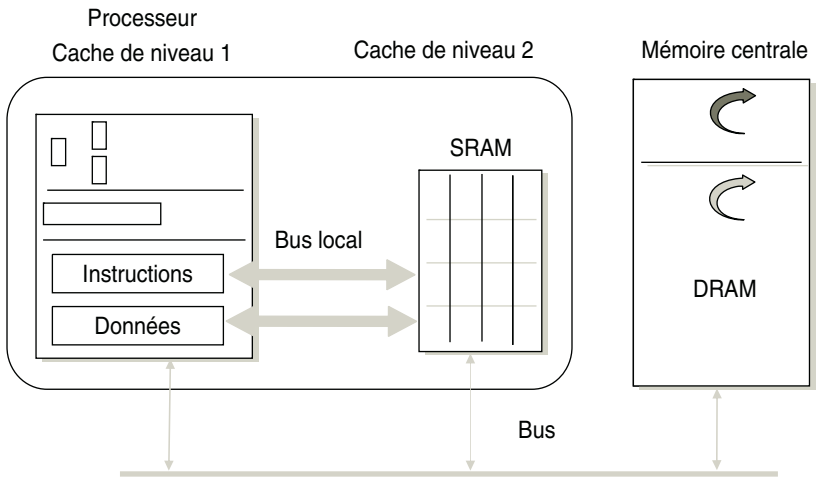


Figure 8.19 Différents niveaux de caches.

Le plus souvent, au niveau 1, il existe deux caches séparés, un pour les instructions et un pour les données. Ces deux caches fonctionnent en parallélisme total et profitent au maximum des techniques de pipeline que l'on trouve sur les ordinateurs modernes. Le cache de niveau 2 est unique et mélange données et instructions.

8.4.2 Mémoire virtuelle

Jusqu'à présent nous avons implicitement (et explicitement) fait l'hypothèse que le programme machine était intégralement présent dans la mémoire centrale. Ceci impose donc que la taille du programme soit plus petite que la taille de la mémoire physique. Cette hypothèse implique que le compilateur ou le chargeur (voir chapitre 1, *Du problème au programme machine*) attribue à chaque donnée ou instruction une adresse physique en mémoire centrale. Il n'est pas toujours possible de tenir cette hypothèse, ni même souhaitable en terme d'efficacité. Nous avons vu que le processeur est « ralenti » par les autres modules de l'ordinateur (temps d'accès à la mémoire centrale mais aussi gestion des entrées-sorties) et n'est donc pas toujours utilisé au mieux. Une manière d'améliorer les performances globales est de permettre à plusieurs utilisateurs (plusieurs programmes machines) placés simultanément en mémoire de se partager le processeur. Pour utiliser le processeur les instructions machine des programmes doivent être en mémoire centrale mais la totalité des instructions de tous les programmes peuvent ne pas être simultanément en mémoire centrale. Ainsi l'intégralité de tous les programmes peut ne pas être présente en mémoire en même temps. Permettre le partage du processeur entre plusieurs programmes c'est donc permettre le partage de la mémoire centrale entre des parties de programmes présentes simultanément en mémoire centrale. Dans cette hypothèse ni le compilateur ni le chargeur ne peuvent a priori attribuer aux données et aux instructions une adresse physique en mémoire centrale. On est amené à séparer l'espace d'adressage du programme et l'espace d'adressage physique en définissant des *adresses virtuelles* pour les instructions et les données. Cette adresse virtuelle ne dépend que de l'espace d'adressage du programme et ne tient pas compte de l'adressage physique (réel) de la mémoire centrale. C'est seulement au moment de l'exécution d'une instruction, et non pas au moment de la compilation ou du chargement, que la correspondance « adresse virtuelle », « adresse physique » est établie. Pour définir une adresse virtuelle on peut considérer le programme comme une suite de blocs, chaque bloc étant référencé par un numéro. L'adresse virtuelle d'un mot de programme (figure 8.20) est alors définie par le doublet {numéro de bloc, déplacement dans le bloc} :

```
adresse_virtuelle = numéro_de_bloc + déplacement dans le bloc.
```

Dans notre hypothèse tous les blocs d'un programme ne sont pas tous nécessairement présents en mémoire centrale. Dans l'espace d'adressage du programme nous appelons les blocs, des blocs logiques. Quand un bloc logique est présent en mémoire principale il correspond à un bloc physique. Le bloc physique a la même taille que le bloc logique, il est implanté en mémoire centrale à l'adresse, « adresse_début_bloc ». Pour déterminer l'adresse effective (physique) d'un mot à partir de son adresse virtuelle, on dispose, pour le programme, d'une table de correspondance (table_des_blocs) qui a autant d'entrées qu'il y a de blocs logiques. Pour chaque entrée on trouve un indicateur de présence ou d'absence du bloc logique en mémoire (existence du bloc physique correspondant au bloc logique) et dans le cas de la présence du bloc physique, l'adresse physique du début de ce bloc, soit adresse_début_bloc (figure 8.20). Dans ces conditions :

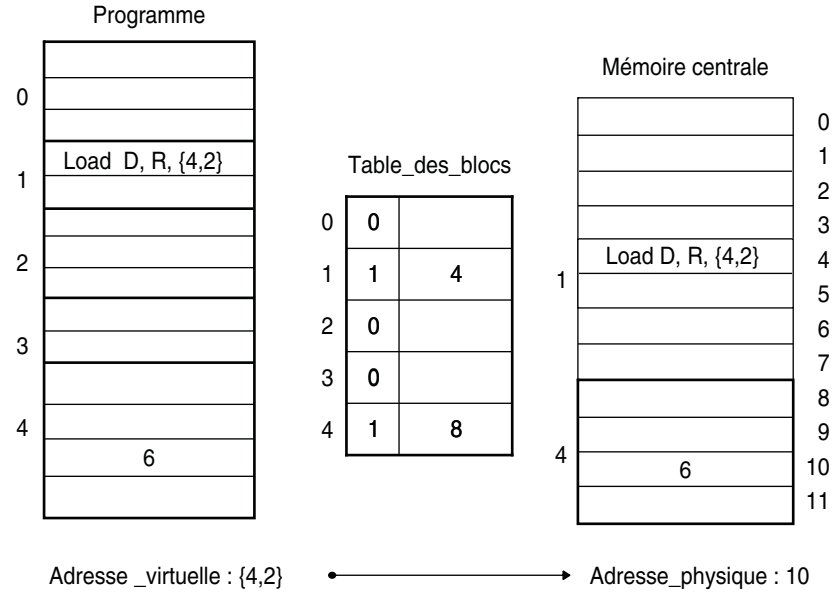


Figure 8.20 Adresse virtuelle.

```
adresse _physique = table_des_blocs[numéro_de_bloc_logique, 1] + déplacement
```

Dans notre exemple (figure 8.20) le programme a une taille de 14 mots. Il est organisé en 5 blocs numérotés de 0 à 4. La mémoire physique a une taille de 11 mots et contient deux blocs logiques (1 et 4). Le bloc logique 1 est implanté à l'adresse physique 4 et le bloc logique 4 à l'adresse physique 8. La table table_des_blocs traduit cette implantation. Dans ces conditions l'instruction `Load D, R, {4, 2}` est chargée en mémoire centrale (donc exécutable) et la donnée « 6 » placée à l'adresse 10 est accessible par cette instruction.

Pour établir la correspondance « adresse virtuelle », « adresse physique » le mécanisme de gestion de la mémoire virtuelle utilise un module matériel (figure 8.21) : le MMU (*Memory Management Unit*). Ce module reçoit en entrée une adresse virtuelle et convertit cette adresse en une adresse physique en utilisant la table table_des_blocs associée au programme en cours d'exécution.

Le fonctionnement du MMU est décrit par l'algorithme suivant : en entrée le MMU reçoit une adresse virtuelle {numéro_de_bloc_logique, déplacement} et en sortie il produit une adresse physique.

```
début
si table_des_blocs [numéro_de_bloc_logique, 0] = 0
alors
    défaut de bloc physique;
    lire bloc_logique sur disque;
    si place libre en mémoire physique
```

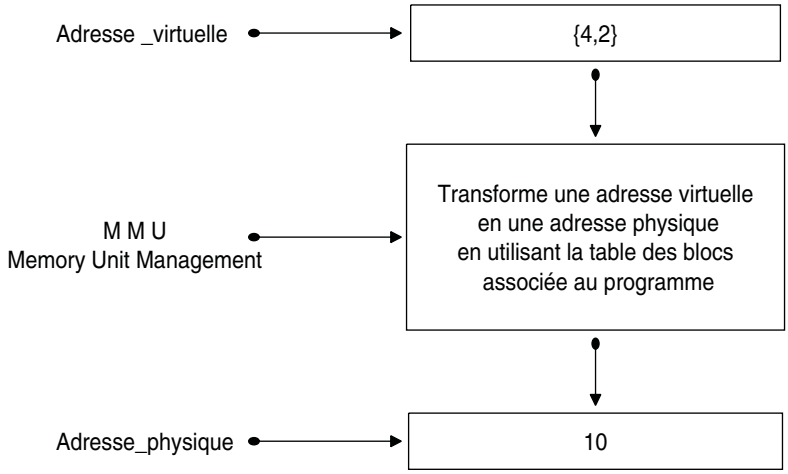


Figure 8.21 Fonctionnement de la MMU.

```
    alors
        charger bloc_logique;
    sinon
        algorithme de remplacement d'un bloc;
        modifier la table table_des_blocs;
    finsi
finsi
calculer adresse_physique;
fin
```

Lorsqu'un bloc logique n'est pas présent en mémoire (défaut de bloc physique) le MMU produit un déroutement vers le système d'exploitation (interruption logicielle) qui examine la place disponible en mémoire centrale. S'il n'y a pas de place pour charger le bloc logique nécessaire à la poursuite de l'exécution, le système d'exploitation exécute alors un algorithme de remplacement de bloc.

Il y a plusieurs choix possibles pour l'installation physique du module MMU, soit à l'intérieur de la puce processeur, soit à l'extérieur. La figure 8.22 donne un exemple d'implantation.

Dans cet exemple, une adresse virtuelle, est présentée au MMU. Il en déduit une adresse physique qui est présentée au cache de niveau 1. C'est alors le mécanisme de gestion du cache qui prend le relais pour charger l'information demandée dans le cache si elle n'est pas présente. Cette implantation n'est pas forcément la plus efficace puisqu'à chaque adresse virtuelle on fait référence au MMU alors qu'il est possible de prévoir si une information est présente ou non dans le cache. De plus le MMU, pour calculer l'adresse physique, fait référence à la table des blocs relative au programme en cours d'exécution. La localisation de cette table est très importante : si elle est en mémoire centrale le calcul de l'adresse physique implique un accès à la mémoire et

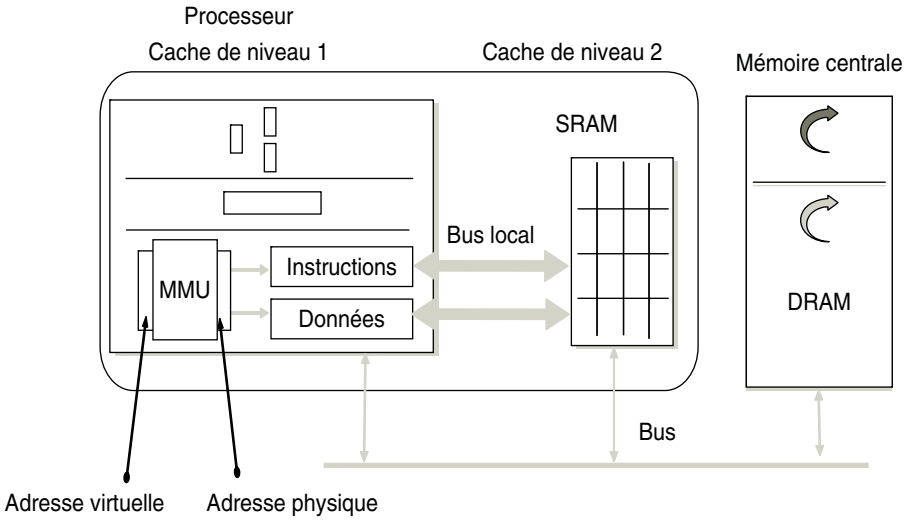


Figure 8.22 Implantation du module MMU dans le processeur.

donc ralentit le processus d'exécution. Il est préférable de placer ce type de table dans le cache afin d'accéder à son contenu le plus rapidement possible.

Le mécanisme de gestion de mémoire virtuelle n'est pas conceptuellement différent du mécanisme de gestion des caches. La différence porte essentiellement sur l'implantation de ces mécanismes. Par exemple dans le cas des caches les algorithmes de remplacement sont pris en charge par le matériel (algorithme hardware) alors que pour la mémoire virtuelle cette gestion est logicielle (prise en compte par le système d'exploitation).

Il y a plusieurs mécanismes possibles pour la gestion de la mémoire virtuelle, taille des blocs fixe ou variable, implication des programmeurs ou transparence au niveau de la programmation, implication de modules matériels ou non. Ces questions d'implantation du mécanisme de gestion de la mémoire virtuelle sont détaillées dans le chapitre relatif aux systèmes d'exploitation traitant de la gestion de la mémoire.

8.5 COMPLÉMENTS : APPROCHES CISC/RISC

L'objectif de ce complément sur les approches CISC et RISC est d'éclairer les différences les plus marquantes concernant ces deux types de processeurs et de montrer les architectures matérielles qu'ils engendrent. Il ne s'agit pas de faire une comparaison exhaustive, encore moins des mesures comparatives, de ces deux classes de processeurs mais plutôt de dégager les raisons principales expliquant cette évolution du CISC au RISC.

Dans une première période les évolutions technologiques ont permis aux architectes d'augmenter les fonctionnalités et les performances des processeurs et des ordinateurs. Ces évolutions ont conduit à une architecture typique comportant un petit nombre

de registres, des modes d'adressage variés et complexes, enfin un grand nombre d'instructions d'une grande complexité. Les gros ordinateurs tels que les IBM370 et les VAX sont représentatifs de ces évolutions et illustrent ce que l'on appelle l'approche CISC (*Complex Instruction Set Computer*).

Dans les années soixante-dix, l'architecture CISC est résumée par un certain nombre de règles de conception énoncées par Patterson et décrite par Étiemble :

– « comme la technologie mémoire utilisée pour la microprogrammation croît très vite, il ne coûte rien ou presque d'utiliser de très gros microprogrammes » ;
– « comme les micro-instructions sont beaucoup plus rapides que les instructions machines, le transfert de fonctions logicielles au niveau microcode accélère le processeur et rend les fonctions plus fiables » ;
– « puisque la vitesse d'exécution est proportionnelle à la taille du programme, les architectures qui diminuent la taille des programmes accélèrent les processeurs » ;
– « les registres sont démodés et rendent difficiles la réalisation des compilateurs. Les piles ou les architectures mémoire à mémoire sont des modèles d'exécution supérieurs ».

Dans ce type d'architecture, le contrôle (séquenceur de l'unité de commande) est microprogrammé : la complexité des instructions machines justifie ce choix en particulier pour la mise au point efficace des séquenceurs. Le microprocesseur Motorola 68000 32 bits implante une puce VLSI (*Very Large Square Integration*) caractéristique de l'approche CISC. Il se trouve que les besoins en performances croissent plus rapidement que les évolutions technologiques, phénomène qui a conduit à un réexamen des architectures des ordinateurs. En 1974, la compagnie IBM lance un projet d'architecture qui conduira à ce qui s'appellera plus tard l'architecture RISC (*Reduced Instruction Set Computer*). Cette architecture est caractérisée par l'existence de caches d'instructions et de données, l'absence d'opérandes en mémoire pour les instructions arithmétiques et logiques ; les instructions sont simples et de longueur fixe.

Les premiers processeurs de ce type ont vu le jour à Berkeley (RISC1) et Stanford (MIPS). Le terme RISC a été introduit par Patterson pour caractériser des architectures où l'on cherche à exécuter une instruction en un cycle mémoire.

Afin de mieux comprendre l'évolution du CISC vers le RISC il convient de préciser quelques points importants concernant les performances des processeurs et la traduction des programmes.

8.5.1 Les performances d'un processeur

Elles font intervenir plusieurs facteurs : le temps d'exécution d'un programme, la gestion de la mémoire (au sens des modes d'accès à la mémoire qui implique les différents modes d'adressage dont on dispose dans les instructions machines), les interruptions et les changements de contexte.

On peut caractériser le temps d'exécution d'un programme, Te, par :

$$Te = Ni * Nc * Tc$$

Ni représente le nombre d'instructions à exécuter. Ce nombre dépend de la nature et de la richesse du jeu d'instructions machine disponible. C'est le compilateur qui, à partir d'un programme écrit dans un langage de haut niveau (LHN), génère la séquence d'instructions machines correspondant à chaque instruction du LHN. *Tc* est le temps d'un cycle machine et *Nc* le nombre de cycles nécessaires à l'exécution d'une instruction machine. Ces facteurs dépendent essentiellement de la complexité des instructions machines.

Ainsi améliorer les performances d'un programme consiste à minimiser Te donc le produit des trois facteurs précédents.

L'exécution d'une instruction se traduit, comme nous l'avons vu précédemment, par l'ensemble des étapes suivantes : fetch, modification du compteur ordinal, décodage de l'instruction, recherche éventuelle du ou des opérandes, exécution (micro-instructions) et rangement des opérandes.

Nous avons également vu que plusieurs de ces étapes pouvaient êtres réalisées en parallèle ce qui s'est traduit par l'implantation de mécanismes de pipeline. Ainsi les approches CISC et RISC pour la minimisation de Te et l'utilisation du pipeline sont différentes.

8.5.2 La traduction des programmes

Pour résoudre les problèmes les programmeurs utilisent des langages de haut niveau que les compilateurs traduisent en langage machine. Au moins deux approches sont possibles pour cette traduction :

– faire correspondre à chaque structure de données exprimée dans le langage de haut niveau un mode d'adressage adapté dans le langage machine, c'est l'approche CISC;
– n'implanter en machine que les mécanismes réellement utiles (dont on a statistiquement montré l'utilité), c'est l'approche RISC.

8.5.3 Approche CISC

Cette approche se caractérise par :

– un jeu d'instructions très riche et une grande variété de modes d'adressages;
– un grand nombre d'instructions;
– des instructions complexes et de longueurs variables afin de répondre à la grande variété des instructions des langages de haut niveau.

On peut étayer les raisons de ces choix par les objectifs suivants :

– simplifier les compilateurs et améliorer leurs performances. Il s'agit ici de réduire la « distance » entre langage de haut niveau et langage machine. On passe plus facilement d'une instruction en langage de haut niveau à la séquence correspondante d'instructions machine;
– un jeu d'instructions riche et réalisant des fonctions complexes permet de réduire la taille du programme machine et donc d'économiser de la place en mémoire principale.

Quelques conséquences de ces choix :

– les instructions, pour traduire facilement la complexité du langage de haut niveau, sont de longueurs variables et opèrent souvent sur un seul opérande en mémoire. Cela implique des modalités de gestion complexe et coûteuse en accès mémoire ;
– un grand nombre de modes d'adressages est nécessaire pour caractériser la richesse des structures de données des LHN.

L'approche CISC se caractérise par une volonté de rendre indépendant le matériel (architecture interne et langage machine) et le logiciel (les langages de haut niveau). Pour obtenir de bonnes performances avec ce type d'architecture il faut donc :

– un compilateur qui permet d'utiliser efficacement le jeu d'instructions machine. Or il existe toujours plusieurs manières de passer d'une instruction du langage de haut niveau à la séquence d'instructions machine qui la réalise sur le matériel. Il faut donc essayer de réaliser le meilleur choix possible dans tous les cas ce qui est difficile voire impossible. En fait des études statistiques montrent que les compilateurs utilisent plutôt les instructions simples que les instructions complexes ;
– la complexité des instructions et des modes d'adressage implique un séquencement microprogrammé. Ce type de séquenceur occupe une surface importante qui peut atteindre 60 % de la surface totale du composant ce qui réduit d'autant la place disponible pour implanter un plus grand nombre de registres et des mémoires caches.

Ainsi c'est l'examen de la complexité des modes d'adressages, des instructions réellement utilisées par les programmes et la complexité du séquencement qui a conduit à l'évolution vers l'approche RISC.

8.5.4 Approche RISC

Si dans l'approche CISC la volonté était de séparer la machine matérielle de l'implantation, dans l'approche RISC c'est la volonté d'une optimisation globale (matérielle et logicielle) qui est le moteur : on souhaite réaliser des architectures efficaces pour l'exécution des programmes.

La définition des architectures RISC repose sur l'examen statistique de l'exécution de programmes. Ces statistiques ont porté au cours des années 70-80 essentiellement sur l'étude des instructions réellement utilisées, les impacts en temps de l'exécution, les modes d'adressages effectivement utilisés. Dans un langage machine on peut classer les instructions en plusieurs types qui sont : l'affectation, l'itération, l'appel de procédure, les branchements conditionnels et les branchements inconditionnels.

Quelques résultats sur ces statistiques et mesures effectuées par Patterson, Seguin, Tanenbaum sont :

– si l'on mesure le pourcentage relatif d'une classe d'instruction par rapport à l'ensemble des instructions utilisées dans un programme il apparaît que les affectations avec transfert simple de données sont très majoritaires (60 %) ;
– sur les impacts en temps d'exécution. Les appels de procédures représentent 20 % des instructions du langage de haut niveau, 60 % du temps d'exécution des instructions

machine et 70 % des temps d'accès à la mémoire. Ce point marque l'importance de l'optimisation des passages de paramètres et du nombre des paramètres ;
– l'essentiel des références mémoire porte sur des variables simples : 75 % sur des constantes et variables scalaires. Les structures de données complexes sont très minoritaires.

Ces résultats sont à l'origine des architectures RISC que l'on peut caractériser par :
– une diminution de la complexité de la partie unité de commande. Le séquenceur est câblé et est donc plus rapide ;
– une diminution de la surface du séquenceur, ce qui permet d'augmenter le nombre de registres et d'utiliser des mémoires caches séparées pour les instructions et les données sur le composant ;
– une simplification des modes d'adressage et des instructions d'où une simplification de la compilation ;
– une implantation d'instructions de longueurs fixes permettant l'utilisation d'un pipeline efficace.

Définition

La définition donnée par M. Slater donne bien l'idée de cette notion d'optimisation globale : « Un processeur RISC a un jeu d'instruction conçu pour une exécution efficace par un processeur pipeliné et pour la génération de code par un compilateur optimisant ».

La traduction en terme matériel

Cette définition implique une pipeline efficace afin de répondre à l'impératif d'une exécution en un cycle machine. L'exécution efficace par un processeur pipeliné implique les caractéristiques suivantes :
– instruction de longueur fixe ;
– codage simple et homogène des instructions ;
– exécution en un cycle machine de la plupart des instructions ;
– accès à la mémoire uniquement par les instructions `load` (chargement d'un mot mémoire dans un registre) et `store` (placement du contenu d'un registre dans un mot mémoire) ;
– modes d'adressages simples ;
– branchements retardés.

Au niveau de la traduction par le compilateur ces caractéristiques se traduisent par un format des instructions à trois adresses et un grand nombre de registres.

8.5.5 Pour conclure sur les RISC et les CISC

L'apparition des processeurs RISC au début des années quatre-vingts a été une remise en cause très forte de l'architecture CISC. L'importance de cette remise en cause a eu l'effet d'une révolution avec ses conséquences : les « anti » et les « pro » qui se sont

assez fortement opposés. Le coût plus faible de réalisation de noyaux RISC a permis un développement industriel dont les machines SPARC et MIPS sont le reflet. Des processeurs ayant des performances proches d'une instruction par cycle machine se sont imposés sur le marché ; notamment des stations de travail. Ensuite des processeurs superscalaires (entre autres le RS/6000) ont permis l'exécution de plus d'une instruction par cycle machine. Les processeurs RISC se sont imposés sur le marché des stations de travail, des contrôleurs, des machines graphiques, des supercalculateurs. Aujourd'hui on retrouve des architectures RISC comme base d'architectures de machines grand public comme Apple. En fait rien n'est vraiment simple en la matière et les processeurs à architecture CISC ont souvent intégré ces évolutions. D'aucuns disent que l'approche RISC a permis le développement de processeurs CISC efficaces. Les processeurs 80486 et ultérieurs d'Intel et les processeurs Motorola à partir du 68040 intègrent le cache dans le pipeline afin de pouvoir exécuter en un cycle machine les instructions les plus fréquemment utilisées : on voit aujourd'hui les progrès spectaculaires des machines CISC.

Aujourd'hui le « pur » RISC a évolué. On trouve dans ces architectures des instructions complexes. On trouve également des architectures mixtes à noyau RISC avec séquenceur câblé et extension CISC à séquenceur microprogrammé. L'apport essentiel de l'approche RISC est la volonté d'une optimisation globale matériel-logiciel-système ; en ce sens il s'agit sûrement d'une étape historique de l'évolution de l'architecture des processeurs. Il est assez vain aujourd'hui de tenter de prévoir les prochaines étapes de l'évolution des architectures RISC ou de la synthèse RISC/CISC ; l'évolution des technologies définira et imposera à coup sûr des compromis permettant d'aller vers des machines plus efficaces.

8.6 CONCLUSION

L'étude de la fonction de mémorisation nous amène à préciser l'architecture matérielle de notre ordinateur. La figure 8.23 représente cette nouvelle architecture.

On y trouve une mémoire ROM contenant des informations permanentes, typiquement le programme de bootstrap dont l'objet est de charger, à partir du disque magnétique, le noyau du système d'exploitation qui nous permet d'accéder aux ressources matérielles et logicielles de l'ordinateur : ces questions seront abordées dans les chapitres consacrés aux systèmes d'exploitation.

La mémoire centrale est au cœur de cette machine et assure la fonction de stockage du programme et des données utilisateur. Notre machine est dite à programme enregistré : pour être exécutable le programme doit être placé en mémoire centrale. Le processeur est chargé de l'exécution de ce programme, il le fait instruction par instruction. Dans une première approche nous avons considéré que le programme doit être intégralement placé en mémoire centrale. Cette approche n'est pas obligatoire et grâce au mécanisme de gestion de la mémoire virtuelle il est possible d'exécuter un programme dont seulement une partie est dans la mémoire centrale. Ce mécanisme repose sur l'existence de composants matériels nouveaux, en particu-

lier le MMU. Ce mécanisme est pris en charge par le système d'exploitation et sera étudié en détail dans les chapitres concernant l'étude du système d'exploitation. Ce dispositif s'appuie sur la prise en compte du système de gestion des interruptions à partir duquel le système d'exploitation peut gérer plusieurs programmes placés simultanément en mémoire centrale. Cela permet une amélioration des performances (plusieurs programmes se partagent le processeur) mais implique de gérer le partage des ressources uniques mémoire centrale et processeur. Cette gestion est assurée par le système d'exploitation.

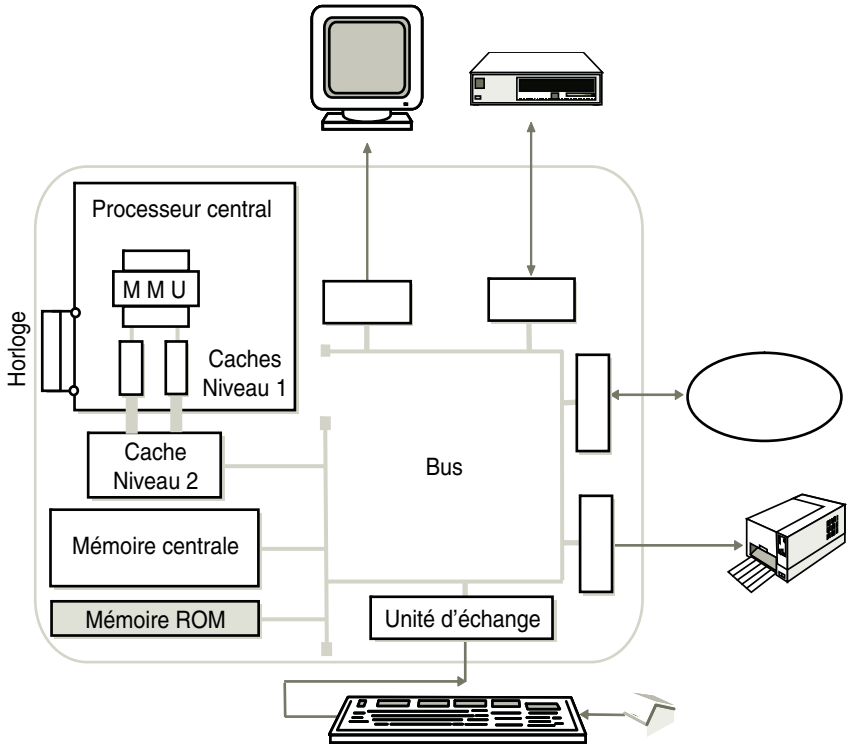


Figure 8.23 Architecture matérielle.

Afin d'améliorer les performances globales de notre ordinateur nous avons introduit les mémoires caches permettant de « rapprocher » du processeur les informations qu'il doit traiter.

Les mémoires de masse, en particulier, le disque magnétique jouent un rôle important dans cette fonction de mémorisation tout particulièrement pour la prise en compte de la gestion de la mémoire virtuelle. Elles permettent également de conserver les données et les programmes au-delà de l'arrêt de l'ordinateur Cette architecture est caractéristique des ordinateurs actuels.

Chapitre **9**

La fonction de communication

Dans ce chapitre nous étudions une fonction fondamentale de nos ordinateurs : la fonction de communication. Cette fonction recouvre toutes les activités permettant les échanges d'informations entre les périphériques, le processeur central et la mémoire centrale. Elle recouvre ce que l'on appelle généralement la gestion des entrées-sorties. Il s'agit d'une fonction complexe au regard des différents éléments intervenant dans cette gestion qui, si elle est mal réalisée est très pénalisante pour les performances globales d'un ordinateur. Pour examiner cette fonction nous utilisons notre machine de base (figure 9.1) qui va nous permettre de présenter les différents composants intervenants dans la gestion des entrées-sorties puis de montrer les évolutions de ces modules afin d'obtenir la structure moderne et efficace des ordinateurs actuels.

9.1 INTRODUCTION

La gestion des entrées-sorties est complexe et génératrice d'un fonctionnement peu efficace de notre ordinateur car elle implique beaucoup de modules qu'il faut savoir combiner harmonieusement.

Les périphériques

Ce sont tous les dispositifs matériels permettant d'assurer les échanges d'informations en entrée et en sortie entre l'ordinateur et l'extérieur ou de stocker de manière

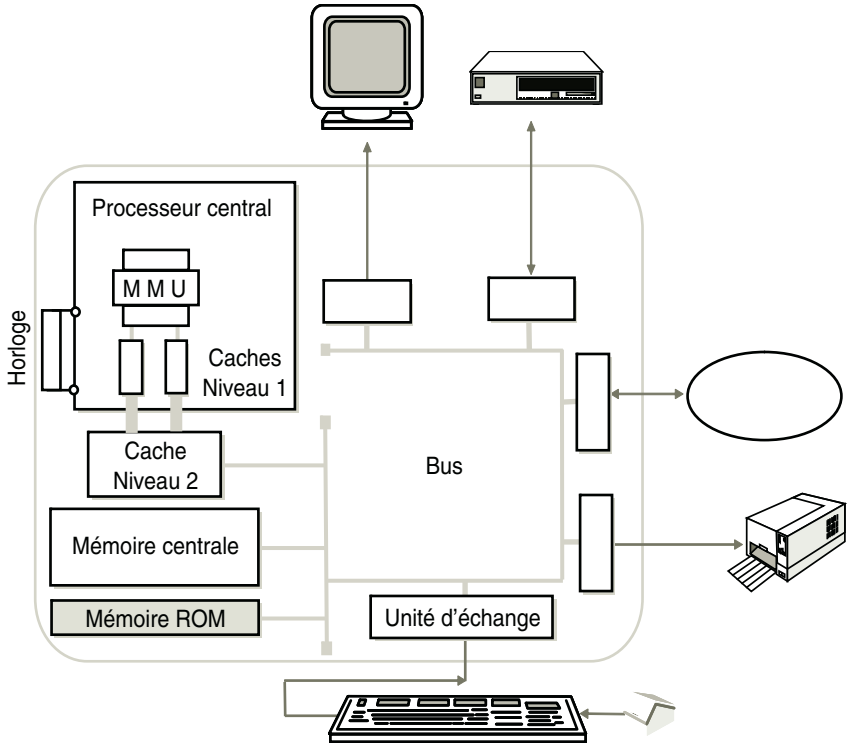


Figure 9.1 Machine de référence.

permanente des informations. On y trouve clavier, souris, imprimantes, écrans, modems et pour le stockage par exemple des disques magnétiques. Chaque périphérique est piloté par un ensemble de signaux spécifiques au périphérique, et activés par l'unité d'échange.

Les interfaces d'accès aux périphériques

Le processeur et la mémoire centrale gèrent des informations numériques, les périphériques sont pilotés par des signaux électriques. Les informations numériques transitent dans l'ordinateur par le bus de communication. Pour écrire des données, par exemple sur une imprimante il faut activer un programme de gestion d'entrées-sorties qui génère un flux de bits qu'il dépose sur le bus de communication à destination du contrôleur de l'imprimante. À la réception de ce flux de bits le contrôleur génère des signaux électriques à destination de l'imprimante qui ainsi activée gère la « mécanique » d'impression en sélectionnant, par exemple, les buses d'impression pour une imprimante à jet d'encre. Le rôle d'un contrôleur est donc de piloter un périphérique et de gérer les accès au bus afin de permettre à un périphérique de communiquer avec l'ordinateur. Dans la fonction d'exécution nous avons vu que les données manipulées par le processeur doivent êtres placées en mémoire centrale. Chaque donnée possède une adresse mémoire permettant d'y accéder. Pour imprimer

une donnée il faut accéder à celle-ci en mémoire centrale (on dispose pour cela de son adresse mémoire) et la faire parvenir via le bus au contrôleur de l'imprimante. Nous serons donc amenés à définir l'adresse et la structure générale d'un contrôleur lui permettant d'accueillir une donnée. Les signaux à produire selon que l'on doive gérer une imprimante, un clavier ou encore un disque magnétique sont différents et sont spécifiques du périphérique. Il existe donc plusieurs catégories de contrôleurs. Certains contrôleurs peuvent directement accéder à la mémoire centrale pour effectuer des échanges avec les périphériques, on dit qu'ils opèrent des accès directs à la mémoire et sont identifiés sous le nom de DMA (*Direct Memory Access*). Afin d'améliorer l'efficacité des échanges, les ordinateurs modernes permettent d'effectuer des opérations d'entrées-sorties en parallèle de l'activité du processeur central (les opérations d'entrées-sorties sont réalisées de manière autonome sans intervention du processeur et sont dites asynchrones). À la fin d'une opération d'entrées-sorties le contrôleur génère une interruption qui interrompt l'activité du processeur au profit d'un programme de gestion de l'interruption spécifique de l'opération d'entrées-sorties en cours. Nous entrevoyons déjà que les mécanismes mis en jeu (asynchronisme, interruptions…) lors d'une opération d'entrées-sorties sont nombreux et complexes.

Le processeur et la mémoire centrale

Une opération d'entrées-sorties fait intervenir un programme exécutant des instructions spécifiques à l'échange et au périphérique. Ce programme, comme tous les programmes exécutables, se trouve en mémoire centrale et est exécuté selon le schéma d'exécution étudié dans la fonction d'exécution.

Le bus de communication

Le programme d'entrées-sorties gère les échanges de données entre mémoire centrale et périphérique, il utilise le bus de communication pour ces échanges. La machine présentée est une machine dite à bus unique. Cette architecture a pendant longtemps été celle de nos ordinateurs personnels, elle pose cependant des problèmes d'efficacité à cause de la très grande hétérogénéité des performances des composants que le bus interconnecte. Par exemple lorsqu'un contrôleur et le processeur central souhaitent utiliser, au même moment, le bus de communication pour réaliser un échange il y a conflit d'accès à cette ressource unique. Il faut, pour régler ce conflit, faire appel à un dispositif particulier (l'arbitre de bus) afin d'ordonnancer les demandes d'accès au bus. Dans une telle architecture la priorité est généralement donnée au contrôleur afin que l'opération d'entrées-sorties puisse s'effectuer sans perte d'informations (si une opération d'entrées-sorties disque est en cours il n'est pas possible de stopper cet échange, pour attribuer le bus au processeur, sans risquer de perdre des informations). Ainsi lorsqu'un programme est en cours d'exécution il utilise à sa guise le bus mais si une opération d'entrées-sorties doit être réalisée, le contrôleur émet une requête à l'arbitre de bus pour signaler qu'un transfert de données est nécessaire. Pendant cette requête le processeur est privé du bus. Ce mécanisme est appelé *vol de cycle*. Ce mécanisme ralentit légèrement le processeur dans l'exécution d'un

programme. Cette architecture est convenable, du point de vue des performances, tant que les différents modules intervenants (processeur, mémoire, contrôleurs) ont été conçus globalement pour fonctionner ensemble. Ces dernières années il y a eu une très forte évolution des performances des processeurs, des mémoires et des unités d'échange. Les bus n'ont pas suivi la même évolution et sont donc devenus un des facteurs importants mettant en cause les performances globales de nos ordinateurs. Nous examinerons plus en détail les évolutions des bus dans les nouvelles architectures.

Le pilote

Le programme de gestion d'une opération d'entrées-sorties est appelé pilote (*driver*). Le dernier élément intervenant dans la prise en compte d'une opération d'entrées-sorties est le programme de gestion des entrées-sorties. Lorsque nous achetons une imprimante ou un disque magnétique le constructeur fournit une disquette (ou maintenant un CD-ROM) contenant un logiciel : le driver (ou pilote) du périphérique. Ce logiciel connaît toutes les caractéristiques techniques du périphérique et est spécifique de ce périphérique. Ainsi un utilisateur travaillant avec un logiciel de traitement de textes clique sur une icône pour déclencher l'impression d'un document (fichier). Cette action sur l'icône d'impression permet d'activer le pilote de l'imprimante, qui reçoit les données (le fichier) à imprimer. Le pilote gère alors l'impression. Ainsi ce n'est pas le logiciel de traitement de texte qui gère directement les impressions, il connaît le pilote du périphérique et lui délègue cette tâche. Le logiciel de traitement de textes connaît le pilote d'impression car ce dernier est installé dans le système d'exploitation : quand nous achetons un périphérique nous devons installer le pilote. Ce mécanisme (déléguer le traitement des entrées-sorties à un programme du système d'exploitation) est général et fondamental dans le traitement des entrées-sorties. En conséquence un programme utilisateur pour réaliser des échanges ne communique pas directement avec le périphérique mais avec le pilote de ce périphérique : les programmes ne connaissent les périphériques qu'au travers des pilotes qui leur sont associés.

Comme l'indique la figure 9.2 la mémoire centrale est partagée en deux parties, l'une contenant le noyau du système d'exploitation, l'autre le ou les programmes utilisateurs selon que la machine est multiprogrammée ou non (ces questions seront détaillées dans les chapitres consacrés au système d'exploitation).

Pour qu'un programme soit exécutable par un processeur il faut, comme nous l'avons vu, le traduire dans le langage du processeur (c'est l'opération de compilation), puis le placer en mémoire centrale (c'est l'opération de chargement en mémoire). On dispose alors en mémoire du programme machine. Pour commander une impression il faut placer dans le programme les instructions d'entrées-sorties pilotant une impression (la syntaxe de ces instructions dépend du langage utilisé pour écrire le programme). Les instructions d'entrées-sorties sont traitées de manière particulière dans cette phase de traduction : un ordre d'opération d'entrées-sorties est traduit par un « branchement » (*SVC, Supervisor Call*) qui est un appel au système d'exploitation. On parle également de trappe système qui permet d'enlever le

processeur au programme utilisateur au profit du pilote capable de gérer cette opération d'entrées-sorties. Ce mécanisme est fondamental à plus d'un titre. On remarque d'abord que les fonctions de gestion des entrées-sorties ne sont pas confiées aux programmes utilisateurs mais au système d'exploitation : l'utilisateur n'a pas à supporter la complexité de la gestion technique d'une très grande variété de périphériques. Par ailleurs ce mécanisme permet de changer de périphériques (par exemple d'imprimante) sans avoir à modifier les logiciels utilisant ce périphérique : si l'on ne disposait pas d'un tel mécanisme il serait nécessaire de modifier son traitement de texte favori à chaque fois que l'on change d'imprimante. De plus, comme nous le verrons un peu plus loin mais surtout dans les parties consacrées aux systèmes d'exploitation, ce mécanisme est déterminant dans la réalisation de systèmes multiprogrammés (Linux, Windows NT, Mac OS X) permettant de gérer simultanément plusieurs programmes utilisateurs.

Dans la suite de ce chapitre sont abordées les questions relatives aux bus, aux interfaces d'accès aux périphériques, aux périphériques ainsi que les différents modes de gestion des entrées-sorties.

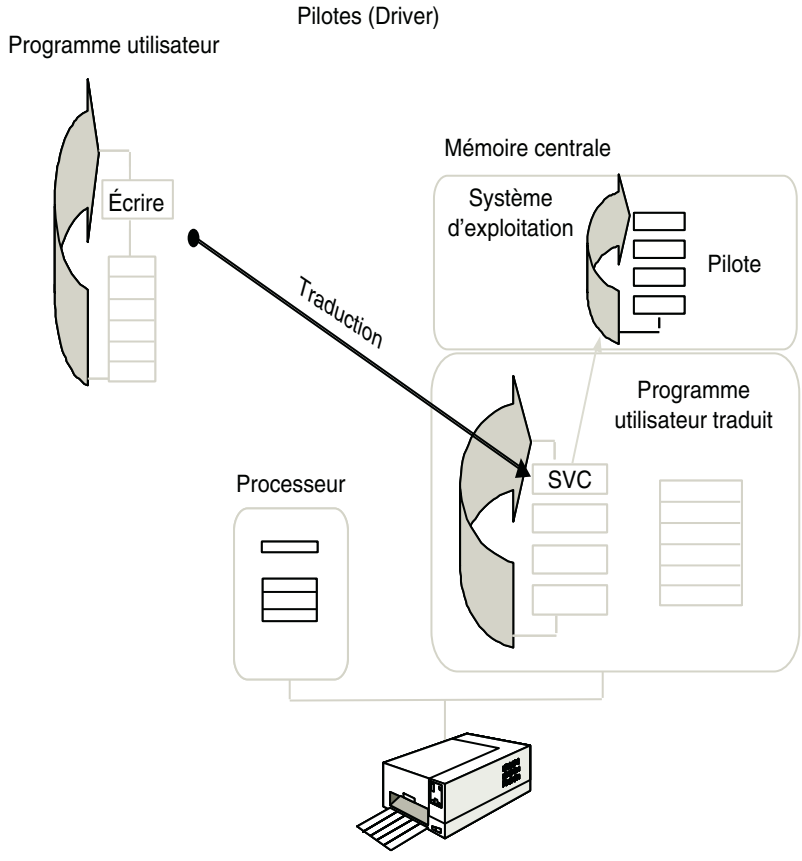


Figure 9.2 Activation du pilote pour la gestion d'une opération d'entrées-sorties.

9.2 LES BUS

Comme nous l'avons indiqué dans les chapitres précédents, un bus peut être vu comme un ensemble de « fils » qui relient les divers composants d'un ordinateur. Dans le chapitre concernant le fonctionnement et l'architecture matérielle du processeur nous avons étudié le bus interne au processeur interconnectant l'unité arithmétique et logique, les registres et l'unité de commande. Dans le chapitre concernant la fonction d'exécution nous avons mis en évidence le rôle du bus de communication entre le processeur et la mémoire centrale. Nous avons défini plusieurs caractéristiques importantes des bus notamment le fonctionnement synchrone ou asynchrone des bus. Nous avons introduit un indicateur important permettant de mesurer l'efficacité d'un bus : la bande passante qui s'exprime comme le produit de la fréquence par la largeur du bus. Une autre caractéristique importante est la manière dont les bits sont transportés. À cet égard on peut classer les bus selon deux grandes catégories :

– *Les bus parallèles*. Ce sont des bus simples constitués d'autant de « fils » qu'il y a de bits à transporter. Ces bus sont coûteux et peu fiables pour des distances importantes. Ils sont utilisés sur des distances courtes, par exemple pour relier le processeur, la mémoire et les unités d'échanges.
– *Les bus série*. Ils permettent des transmissions sur de grandes distances. Ils utilisent une seule voie de communication sur laquelle les bits sont sérialisés et envoyés les uns à la suite des autres.

L'ordinateur qui nous sert de guide et dont nous affinons l'architecture au cours des différents chapitres est organisé autour d'un bus unique (figure 9.1). En fait nous avons prolongé le bus reliant processeur et mémoire centrale afin de connecter les autres composants c'est-à-dire les unités d'échanges (contrôleurs) pilotant les périphériques. Ce type d'architecture a été pendant une longue période l'architecture générale des ordinateurs. Cette architecture convenait dans la mesure où l'ordinateur était vu globalement et que les performances des différents composants étaient compatibles. Ces dernières années ont vu une évolution très forte des performances des processeurs et des mémoires. Par ailleurs les constructeurs ont mis sur le marché des périphériques de plus en plus sophistiqués aux performances élevées. Enfin le développement des interfaces graphiques et des applications multimédia qui véhiculent de très grandes quantités d'informations ont obligé les constructeurs à faire évoluer les bus. En effet les bus n'ont pas suivi la même évolution, que l'ensemble des composants des ordinateurs, et sont devenus un facteur critique pour ce qui concerne les performances globales. De plus l'organisation autour d'un bus unique est mal adaptée pour relier des composants hétérogènes aux performances très différentes : la mémoire et le processeur ont besoin de très hauts débits alors qu'une imprimante ou une souris se satisfont de flux réduit. Une approche a consisté à mettre en place des dispositifs matériels permettant à des périphériques d'échanger des informations avec la mémoire centrale sans utiliser le processeur : les DMA (*Direct Memory Access*). Ces dispositifs sont encore très largement utilisés. Ce dispositif couplé avec le mécanisme d'interruptions permet de rendre asynchrones les entrées-sorties libérant ainsi le

processeur qui peut alors effectuer d'autres tâches pendant qu'une opération d'entrées-sorties physique se déroule.

La figure 9.1 est caractéristique de l'architecture à bus unique. Le bus ISA (*Industry Standard Architecture*) a été le standard des bus pour les ordinateurs ayant une architecture à bus unique. Dans ce type d'architecture, le bus est un goulet d'étranglement : les processeurs très performants produisent des informations qui ne sont traitées par le bus qu'avec les performances de ce dernier. Au total ce sont donc les caractéristiques du bus qui déterminent les performances globales de l'ordinateur. L'arrivée des interfaces graphiques a accentué ce problème du goulot d'étranglement. La gestion des fenêtres et le traitement des graphiques réclament des débits importants qui ne peuvent être assurés par le bus ISA (fréquence de 8 MHz pour une largeur de 16 bits soit une bande passante de 8,33 Mo/s). Une évolution importante est venue de la compagnie IBM qui a développé le bus MCA (*Micro Channel Architecture*) aux performances plus élevées que le bus ISA pour les ordinateurs PS/2. Malheureusement ce bus ne supportait pas les cartes d'accès au bus ISA. Or, l'industrie micro-informatique, pour protéger ses marchés, a défini le bus ISA comme le standard, ce qui est une probable explication au retard pris dans l'évolution des bus. EISA (*Extended Industry Standard Architecture*) est une évolution du bus ISA qui présente de meilleures performances et dont l'objet a été de concurrencer le bus MCA. Au total les bus MCA et EISA n'ont pas été acceptés à cause de leur coût et de leur incompatibilité.

Le groupement VESA (*Video Electronics Standard Association*) développa l'idée de connecter directement le contrôleur graphique sur le bus reliant mémoire centrale et processeur. Ainsi un processeur manipulant des données sur 4 octets avec une fréquence de 33 MHz pouvait offrir un débit de 132 Mo/s. Ce projet donna lieu à la création du VLBus. Malheureusement ce bus a été conçu au départ pour le processeur 80486 d'Intel ce qui en limitait trop les perspectives d'évolutions.

À partir de 1991, Intel a donc développé un bus PCI (*Peripheral Component Interconnect*) dont les caractéristiques devaient permettre des performances élevées (débits supérieurs à 100 Mo/s) et une grande souplesse d'utilisation. Ce bus, qui est devenu un standard, a permis la définition d'un bus graphique AGP (*Accelerated Graphic Port*) destiné plus spécifiquement à la gestion des contrôleurs graphiques. Enfin Intel a développé des « chipset » permettant à la fois une plus grande intégration des composants et la possibilité d'interconnecter des bus de type différents.

Afin d'élargir l'utilisation des bus PCI, Intel a mis dans le domaine public tous les brevets attachés à ce bus. Ce type de bus permet une évolution vers des architectures à plusieurs bus mieux adaptées à l'hétérogénéité des dispositifs.

9.2.1 Les bus ISA (ou PC-AT), MCA et EISA

Le bus ISA est apparu dans les années 1985 avec les ordinateurs PC-AT. Il est encore très répandu. Il s'agit d'un bus synchrone. Sa fréquence est celle du processeur 80286 soit 8,33 MHz et a une largeur de 2 octets ce qui lui donne une bande passante de 16,7 Mo/s. L'évolution des processeurs, plus rapides et manipulant des données

sur 16 et 32 bits, a obligé les constructeurs à définir de nouveaux bus. Ce sont les bus MCA et EISA.

Le bus MCA développé par IBM pour les ordinateurs PS/2 est un bus de 32 bits, asynchrone fonctionnant à 10 MHz avec une largeur de 32 bits. Ces caractéristiques lui confèrent une bande passante de 40 Mo/s bien supérieure à celle du bus ISA. Indépendant du processeur il peut s'adapter à différentes architectures matérielles. Cependant il ne reconnaît pas les cartes ISA et sa complexité de fabrication le rend trop coûteux.

Le bus EISA est une amélioration du bus ISA ayant une largeur de 32 bits. Il doit cependant rester compatible avec son aîné et fonctionne donc nécessairement à une fréquence de 8 MHz. Sa bande passante est donc double de celle du bus ISA.

Avec l'apparition des applications graphiques et multimédias utilisant, par exemple de la vidéo, les performances de ces bus sont devenues totalement insuffisantes. Les bus deviennent alors un facteur important de la réduction des performances des ordinateurs.

Examinons le cas d'un affichage vidéo sur un écran, le fichier d'images provenant d'un disque magnétique ou d'un CD-ROM. Le moniteur couleur est standard et a une définition de $1\,024 \times 768$, la couleur de chaque point affiché (*pixel*) est codée sur 16 bits (soit 32 000 couleurs différentes, ce qui aujourd'hui est très pauvre). Une image nécessite donc $1\,024 \times 768 \times 2$ octets = 1 572 864 octets. Pour qu'une image soit stable il faut qu'elle soit affichée au moins 25 fois par seconde sur l'écran. La bande passante nécessaire est donc de $1\,572\,864 \times 25$ octets/s soit environ 40 Mo/s. En réalité si l'on examine le chemin parcouru par les données (disque vers mémoire puis mémoire vers moniteur) la bande passante que doit avoir le bus est de 80 Mo/s. Ces chiffres sont totalement incompatibles avec les performances des bus que nous venons de voir (Ces résultats sont d'autant moins compatibles que nous avons choisi un codage des couleurs sur 16 bits alors qu'aujourd'hui les couleurs sont codées sur au moins 24 bits ce qui impliquerait une bande passante d'environ 120 Mo/s). Ce sont ces contraintes qui ont amené, vers 1990, Intel à définir un bus à haute performance : le bus PCI.

9.2.2 Le bus PCI (*Peripherical Component Interconnect*)

Caractéristiques

Le bus PCI s'est imposé comme composant des cartes mères grâce à ses performances et à l'acceptation de ses normes par un grand nombre de constructeurs.

On peut résumer ses caractéristiques principales par :

– C'est un bus local synchrone cadencé à une fréquence de 33 ou 66 MHz (version 2.1).
– Il a une largeur de 32 ou 64 bits ce qui lui confère une bande passante de 132, 264 ou 528 Mo/s. Grâce à cette capacité de transmission le bus PCI répond à tous les types de périphériques et à tout type d'application. Il n'est donc pas un goulot d'étranglement pour le système informatique.

– Afin de réduire l'encombrement et le nombre de broches le bus PCI est multiplexé. Dans ce mode les lignes du bus transportent alternativement des adresses et des données. On divise ainsi par deux le nombre de lignes du bus. Les opérations de lecture et d'écriture impliquent alors plusieurs cycles de bus. Les transactions sont réalisées entre un maître (qui initie la transaction) et un esclave (la cible de la transaction). Par exemple une transaction de lecture se fait en 3 cycles de bus : le maître place l'adresse de la donnée à lire sur les lignes du bus, le maître passe la main à l'esclave qui peut alors utiliser le bus, l'esclave place la donnée adressée sur le bus qui est alors disponible pour le maître.
– Les spécifications du bus PCI permettent d'installer une architecture *Plug And Play*[1].
– Comme les bus ISA le bus PCI permet de gérer d'autres unités centrales, de la mémoire et des entrées-sorties. Aussi de nombreuses fonctions du bus standard ISA migrent vers le bus PCI.

Bien que la bande passante de ce bus soit très importante, ce bus ne convient pas aux transferts entre mémoire centrale et processeur. De plus il ne permet pas la prise en compte des cartes additionnelles compatibles avec le bus ISA.

Le tableau 9.1 résume les principales performances des bus présentés jusqu'à ce point.

Tableau 9.1 CARACTÉRISTIQUES DES BUS DE COMMUNICATION.

Type	Largeur : bits	Fréquence : Mhz	Connecteurs	Bande passante : Mo/s
ISA	16	8	8	16
EISA	32	8	18	32
MCA	32	8	8	32
VESA	32	33	3	132
PCI	32	33	10	132
PCI	32	66		264
PCI	64	33		264
PCI	64	66		528

Architecture

Pour répondre à la grande variété de besoins, une architecture multibus s'est imposée. La figure 9.3 présente une telle architecture. La plupart des micro-ordinateurs conçus à partir de processeurs Pentium II repose sur une telle architecture. Cette architecture n'est pas seulement celle des PC mais on la retrouve également avec quelques variantes chez Apple. Les composants clés de cette organisation sont les composants d'interconnexion de plusieurs environnements hétérogènes.

1. Le périphérique est automatiquement reconnu et le pilote nécessaire à son fonctionnement automatiquement installé.

Dans la terminologie Intel ces composants sont appelés des *ponts*. Le pont PCI interconnecte le processeur, la mémoire centrale et le bus PCI. Le pont PCI/ISA interconnecte le bus PCI et le bus ISA et peut piloter directement un ou plusieurs disques IDE ainsi que les ports USB.

Ces composants – les ponts pour Intel – existent pour d'autres types d'ordinateurs. En fait ils correspondent à une évolution de la technologie qui permet de réduire le nombre de composants sur la carte mère. Ces composants, les *Chipset*, intègrent un grand nombre de fonctionnalités. Ils gèrent tout particulièrement des fonctions fondamentales (qui auparavant nécessitaient plusieurs composants) telles que :

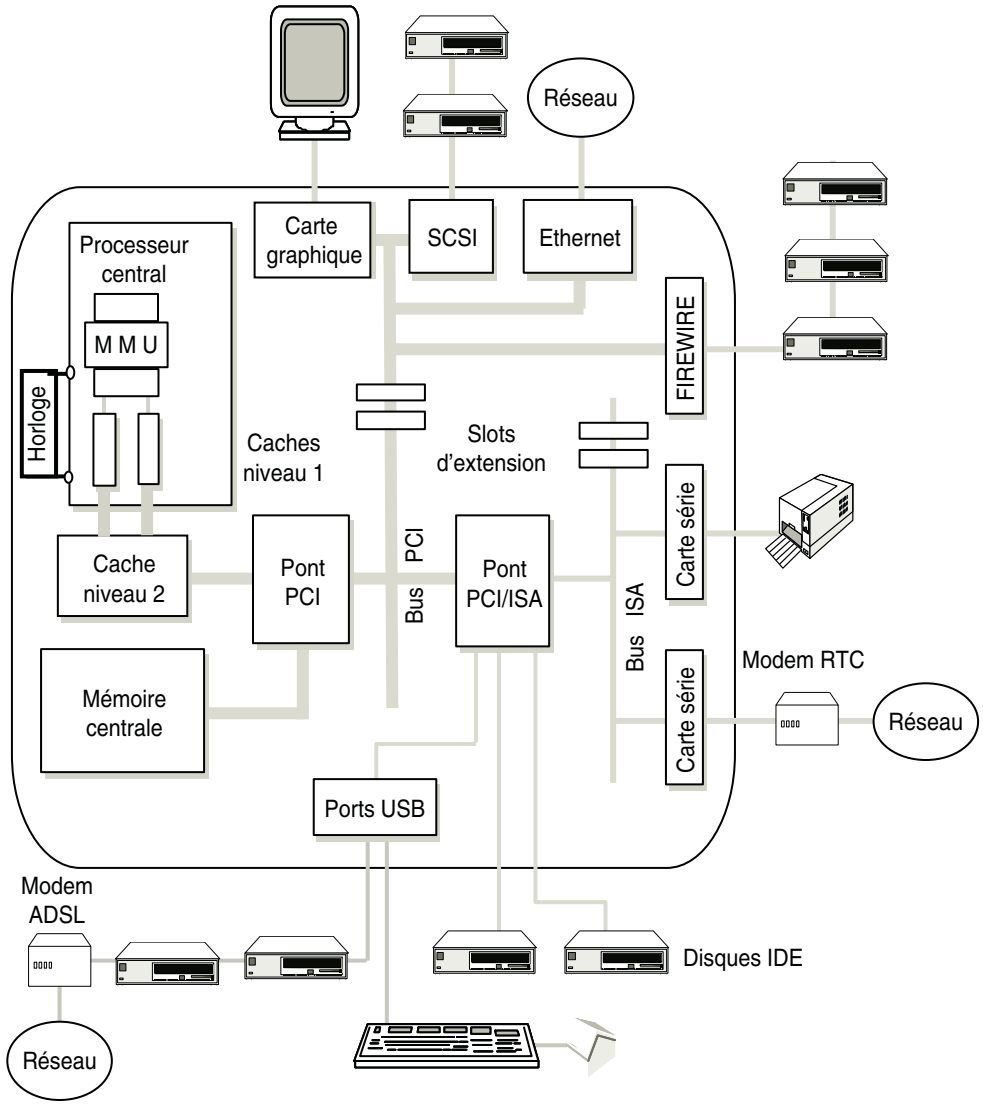


Figure 9.3 Architecture multibus (bus PCI).

– gestion et arbitrage des accès au bus ;
– gestion du temps à l'aide de timers ;
– contrôleurs d'interruptions.

À titre d'exemple Intel a développé un chipset Intel 40BX composé d'un module 82443BX (qui permet l'interfaçage entre le processeur, la mémoire centrale et les autres bus, PCI et AGP) et du contrôleur PIIX4 (interfaçage entre les bus PCI et ISA).

Les principales caractéristiques du premier composant sont :

– gestion d'un ou deux processeurs Pentium II interconnectés par un bus système à 100 MHz ;
– contrôle de la mémoire centrale. Il supporte de la mémoire de type EDO à 60 ns ou SDRAM (100 MHz) ;
– gestion de l'interface avec le bus PCI ayant une fréquence de 33 MHz ;
– gestion d'une interface pour le bus AGP dont la fréquence de 133 MHz permet un débit de 500 Mo/s.

Enfin ce composant contient des registres accessibles dans l'espace d'adressage du processeur, il peut donc contrôler les accès aux périphériques pilotés par les bus PCI et AGP (nous revenons sur ces notions d'adressage dans la partie concernant les interfaces d'accès aux périphériques).

Pour ce qui concerne le deuxième composant de ce chipset on peut noter les caractéristiques suivantes :

– interfaçage PCI/ISA ;
– contrôle des interfaces IDE. Il peut gérer jusqu'à 4 disques ;
– une gestion USB pour deux ports à 12 ou 1,5 Mbits/s ;
– gestion des fonctionnalités correspondant à deux contrôleurs DMA ;
– gestions des interruptions qui correspondent à deux contrôleurs 82C59 (dont nous avons parlé dans le chapitre concernant la fonction d'exécution).

Cet exemple montre le haut niveau d'intégration que permettent ces chipset.

L'avantage majeur de ce type d'architecture est certainement que la bande passante du bus est bien adaptée aux périphériques associés. Le couple processeur/mémoire dispose d'un bus privé dont la bande passante est très élevée, le bus PCI (dont la bande passante est élevée) permet une connexion bien adaptée des disques à hautes performances (par exemple de type SCSI ou FIREWIRE). La possibilité d'interconnexion au bus ISA permet de satisfaire les besoins des périphériques lents (imprimantes, modems RTC). Enfin ces composants intègrent la connexion aux ports USB que nous étudierons dans la partie concernant l'interfaçage avec les périphériques.

Les chipset existent dans d'autres architectures que celles développées par Intel. Ils ne sont pas nécessairement organisés avec le même nombre de composants. Une des raisons de cette architecture « multiponts » chez Intel vient probablement de la nécessité de prendre en compte une compatibilité ascendante non obligatoire pour d'autres constructeurs (Apple par exemple).

9.2.3 Le bus AGP (*Accelerated Graphics Port*)

Caractéristiques

L'utilisation intensive des graphismes en 3D et de la vidéo impose des débits toujours plus importants. Une solution est la mise en place d'un bus spécialisé pour le traitement graphique : le bus AGP. La spécification de ce type de bus date de 1996, elle s'appuie sur la spécification du bus PCI en lui apportant des améliorations. La principale de ces améliorations est le démultiplexage des adresses et des données sur le bus et l'introduction du mode pipeline pour les opérations de lecture et d'écriture en mémoire.

Au début le bus AGP à 66 MHz autorisait un débit de 266 Mo/s. Avec le mode AGP2X puis AGP4X on obtient des débits de 533 Mo/s et 1 Go/s.

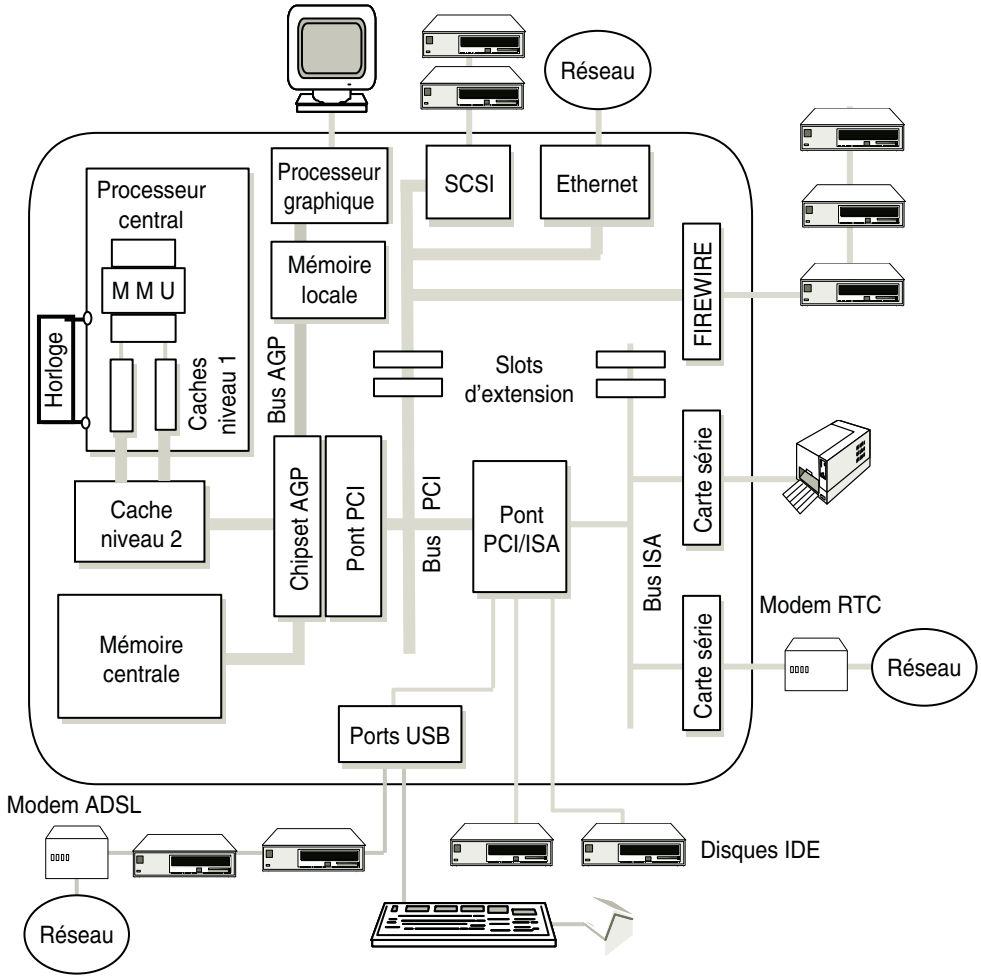


Figure 9.4 Architecture multibus (bus PCI et AGP).

Architecture

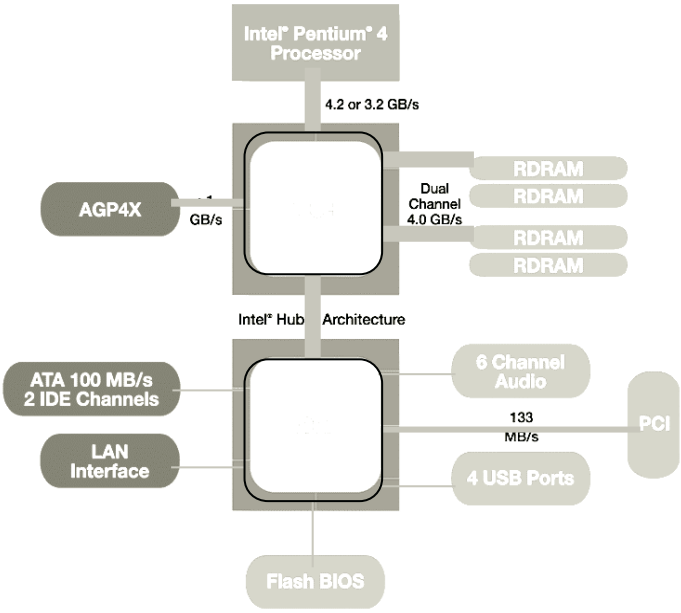
La figure 9.4 présente une architecture qui peut être considérée comme l'architecture de base actuelle pour nos micro-ordinateurs.

9.2.4 Deux exemples

Nous présentons ici deux exemples d'architectures tirées de la documentation des constructeurs Intel et Apple. Ces présentations graphiques ne sont là que pour illustrer ce qui a été dit précédemment.

Architecture Intel

Cette architecture (figure 9.5) correspond bien aux schémas génériques présentés précédemment. On voit que le bus AGP est annoncé comme ayant un débit supérieur à 1 Go/s ce qui est très favorable pour les échanges et la gestion des applications graphiques utilisant par exemple la vidéo. De la même manière les bus processeur/pont et pont/mémoire sont à très hauts débits. Enfin la mémoire de type RDRAM est comme nous l'avons vu un type de mémoire ayant un temps d'accès très faible. Cette architecture permet en particulier des échanges directs entre mémoire principale et mémoire vidéo avec des débits très élevés. Ce sont assurément des facteurs d'augmentation globale des performances.



Architecture Mono-processeur Pentium 4
Documentation Intel

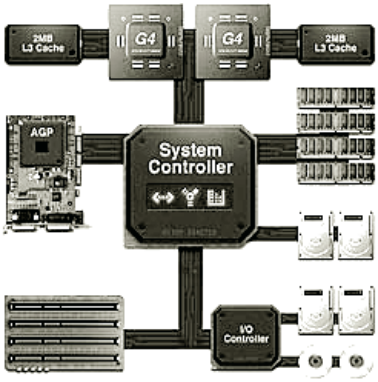
Figure 9.5 Architecture Intel.

Il faut noter qu'il existe aussi chez Intel des architectures similaires mais multi-processeurs avec un bus processeur/processeur à très haut débit.

Architecture G4 (Apple)

L'architecture présentée sur la figure 9.6 est une architecture biprocesseurs assez classique chez Apple. Là également nous sommes très proches des architectures génériques présentées. Les performances des bus et des composants sont proches de celles du monde Intel. Les différences proviennent en particulier des processeurs. Chez Intel on dispose de processeurs de type CISC alors que ce sont des processeurs de type RISC chez Apple. Dans ces conditions il est assez difficile de comparer les performances pures des processeurs. Peut-on raisonnablement et brutalement comparer le mégahertz voire le gigahertz de processeurs d'architectures si différentes ?

Au plan architectural on peut tout de même noter une différence. Il n'existe qu'un seul pont contrairement à l'architecture Intel. Pour Apple il s'agit là d'un facteur d'augmentation de l'efficacité à cause de la diminution du nombre de composants. Comme précédemment il faut faire très attention aux comparaisons hâtives.



Architecture bi-processeurs G4
Documentation Apple

Figure 9.6 Architecture Apple.

9.3 LES INTERFACES D'ACCÈS AUX PÉRIPHÉRIQUES

Les micro-ordinateurs sont organisés autour d'un ou plusieurs processeur(s), de circuits mémoires, d'unités d'échanges (contrôleurs) et du bus d'interconnexion reliant ces différents composants. Nous avons examiné les fonctionnements des processeurs, des mémoires et l'architecture des bus permettant de faire fonctionner ces composants ensemble. Une opération d'opération d'entrées-sorties implique d'échanger des informations entre les composants internes et les périphériques. Pour réaliser ces échanges on dispose de composants particuliers, les *unités d'échanges*, dont le rôle est de piloter les périphériques. Nous allons donc étudier dans cette partie les unités

d'échanges et plus généralement les interfaces d'accès aux périphériques. Nous abordons cette présentation en présentant les fonctionnalités des unités d'échanges que l'on peut considérer comme traditionnelles (cartes séries, parallèles, les cartes d'accès aux réseaux locaux.). Puis nous présentons les bus d'extension (pour la gestion des entrées-sorties) et plus particulièrement les bus séries USB, SCSI et FIREWIRE. Il s'agit là d'un choix arbitraire (nous aurions pu décrire ces bus dans la section consacrée aux bus) qui d'un point de vue fonctionnel nous est apparu cohérent.

9.3.1 Les unités d'échanges

Nous allons d'abord préciser la manière d'accéder à une unité d'échange, c'est-à-dire comme nous l'avons fait pour la mémoire, définir « l'adresse » d'une unité d'échange. Nous précisons ensuite la structure fonctionnelle d'un contrôleur. Enfin nous donnons des exemples de telles unités d'échanges.

Généralités : adressage, structure et fonctionnement d'une unité d'échange

Il y a deux manières d'atteindre un circuit de gestion d'entrées-sorties. On peut le voir du point de vue de l'adressage, comme un composant spécifique. Il faut alors disposer d'un signal particulier que l'on dépose sur le bus pour indiquer que l'on veut faire une opération d'entrées-sorties. L'autre manière est de décider que les circuits d'opération d'entrées-sorties font partie de l'espace d'adressage du processeur. Le choix de l'une ou l'autre des méthodes est arbitraire. La figure 9.7 présente l'adressage d'une unité d'échange comme faisant partie de l'espace d'adressage du processeur.

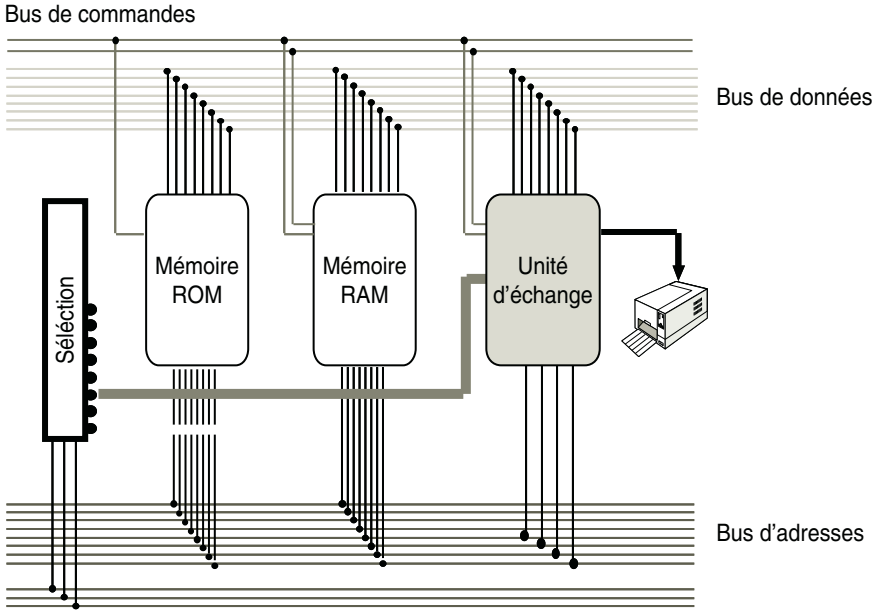


Figure 9.7 Espace d'adressage unique.

Dans cette hypothèse on adresse une unité d'échange comme l'on adresse un banc mémoire. Une adresse est déposée sur le bus d'adresses, une partie de cette adresse est prise en charge par un circuit de sélection (circuit de décodage) qui permet de sélectionner un « boîtier » qui sera soit un banc mémoire soit une unité d'échange. Dans la figure 9.7, 3 fils sont réservés à la sélection d'un boîtier. Ces 3 fils gérés par le circuit de décodage permettent un accès à 8 boîtiers différents.

La structure générale d'une unité d'échange est donnée par la figure 9.8. Il s'agit d'une structure fonctionnelle qui n'enlève rien à la généralité du fonctionnement réel des contrôleurs. Une unité d'échange est organisée autour de plusieurs registres. Leur nombre et leur gestion dépendent des spécificités particulières de l'unité d'échange.

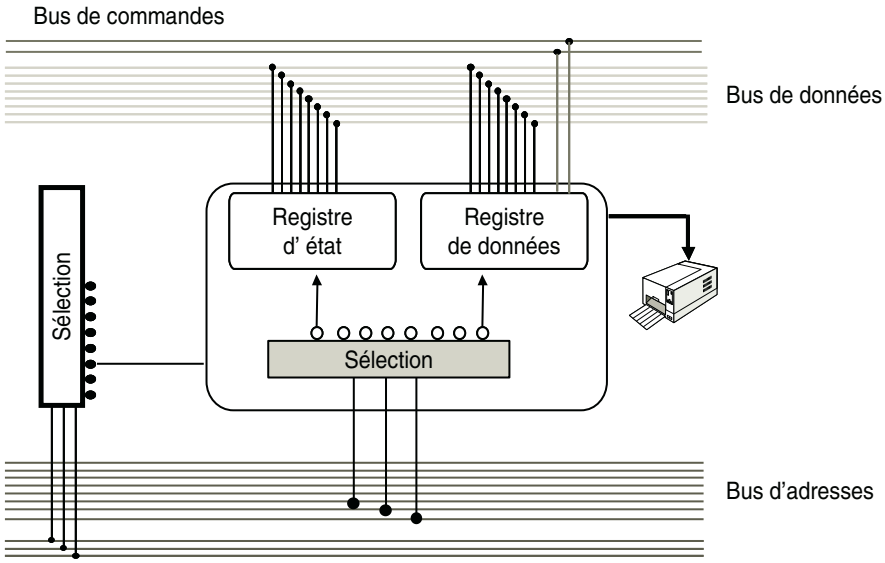


Figure 9.8 Structure générale d'une unité d'échange.

Fonctionnellement on trouve :

– un registre d'état qui permet de connaître l'état du périphérique piloté par cette unité d'échange. Chaque état est caractérisé par une valeur numérique disponible dans le registre d'état. Par exemple une imprimante qui n'a plus de papier émet un signal vers le contrôleur qui peut alors charger le registre d'état avec la valeur numérique correspondant à cet état. Cette information maintenant placée dans le registre d'état est disponible pour le processeur ;
– un registre de données. C'est au travers de ce registre que se font les échanges de données entre la mémoire et l'unité d'échange.

Les registres sont adressables : chaque registre a une adresse. C'est exactement ce qui se passe pour la mémoire centrale où l'on sélectionne un banc mémoire et un mot dans ce banc. Une partie du bus de commande (figure 9.8) adresse un circuit de

décodage interne à l'unité d'échange afin de sélectionner un des registres du contrôleur. Par exemple pour réaliser une opération d'écriture sur une imprimante on pourra exécuter la séquence :

– le processeur place l'adresse du registre d'état sur le bus d'adresses. Cette adresse est composée d'une part de l'adresse du boîtier, d'autre part de l'adresse du registre dans le boîtier ;
– le processeur lit le registre d'état ;
– si le contenu du registre indique que l'imprimante est libre, le processeur adresse le registre de données ;
– le processeur dépose la donnée à imprimer sur le bus de données. La donnée est maintenant disponible pour l'imprimante ;
– l'unité d'échange va positionner les signaux nécessaires déclenchant l'impression physique de la donnée sur l'imprimante.

Cette présentation simplifiée est en fait générale. Dans l'architecture matérielle que nous avons utilisée dans la section sur les bus, nous avons vu la complexité du schéma d'accès à une unité d'échange. Notre schéma d'adressage pourrait sembler inadapté. En fait les boîtiers de type « pont », permettant l'interconnexion des parties hétérogènes, disposent de registres de communication avec le processeur. Ces registres contiennent les adresses des unités d'échanges, le pont effectue alors des « translations » d'adresses afin de s'adapter aux différents composants transportant cette information.

Exemple 1 : la liaison série

Cet exemple a pour objet de préciser sur un cas concret l'ensemble des éléments mis en cause lors d'un échange. Comme le montre la figure 9.9 nous avons choisi d'étudier la communication entre deux ordinateurs distants au travers du réseau téléphonique commuté (RTC).

Du point de vue logiciel cette communication implique l'existence de deux programmes (un sur chaque ordinateur) qui exécutent des instructions pilotant le matériel connectant les ordinateurs. Les programmes vont échanger des messages selon un *protocole de communication*. Dans une telle communication les ordinateurs sont identifiés par un numéro de téléphone. Pour communiquer, un ordinateur doit exécuter un programme qui demande à son modem de composer le numéro de téléphone du modem distant. Le modem émetteur attend alors l'acceptation du modem distant, il transmet cette acceptation au programme qui peut alors échanger avec le programme distant au travers du réseau.

Pour cette communication le programme émetteur, au travers des différents bus, envoie des données à l'unité d'échange spécifique des communications séries (UART). Cette dernière poste les signaux électroniques adéquats pour le modem. Lorsque le modem reçoit des données il poste des signaux pour l'UART qui agit en fonction de la nature des signaux reçus. Cet exemple illustre bien les modalités complètes de fonctionnement d'une unité d'échange :

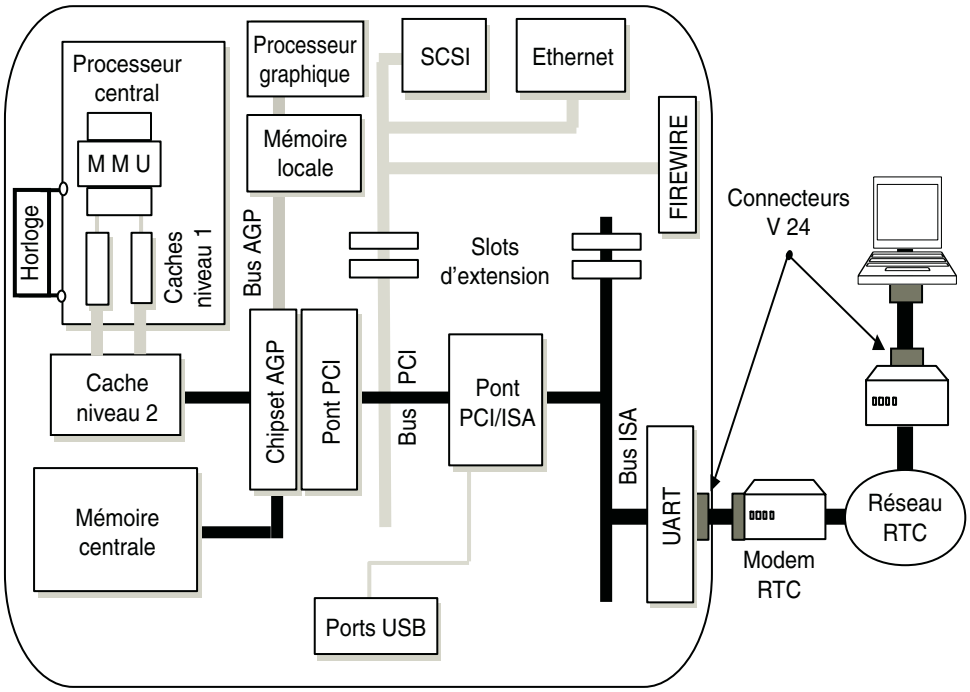


Figure 9.9 Communication entre deux ordinateurs distants.

– communication d'informations numériques avec le programme d'entrées-sorties au travers des composants internes ;
– production et réception de signaux électroniques pour les échanges avec le monde extérieur.

Nous allons examiner plus en détail l'architecture de l'unité d'échange, la manière d'échanger des données avec les composants internes, la production et la réception des signaux électroniques pour réaliser un tel échange. Du point de vue matériel cette communication met en jeu l'UART, le connecteur V 24 (RS232) et le modem.

➤ UART : Universal Asynchronous Receiver Transmitter

La figure 9.10 présente l'architecture d'une UART. Globalement on retrouve les composants qui ont fait l'objet de la description fonctionnelle d'une unité d'échange. L'UART est une unité d'échange, le système d'exploitation inclut donc un pilote (driver) spécifique de cette unité d'échange. Le pilote connaît l'adresse des différents registres composant l'UART et effectue des échanges avec ces registres.

Le bus est en relation avec les registres suivants :

– le registre de réception. Il contient le caractère en provenance de l'ordinateur distant. L'UART manipule les informations caractère par caractère ;
– le registre de validation des interruptions que l'on accepte de prendre en compte ;

– le registre indiquant le statut de la ligne de transmission (libre, occupée) ;
– le registre d'identification de l'interruption. Lorsqu'un caractère vient d'être reçu dans le registre de réception ou vient d'être émis depuis le registre d'émission, il y a levée d'une interruption qui prévient le processeur qu'un caractère est disponible dans l'UART ou que l'UART est disponible pour une nouvelle émission. On peut donc déclencher l'exécution du programme d'extraction du caractère ;
– le registre diviseur. C'est le registre dont le contenu permet de définir la vitesse d'émission et de réception de la ligne ;
– le registre de contrôle du modem. C'est à partir de ce registre que l'on positionne les signaux à destination du modem ;
– le registre statut du modem. Il contient les informations permettant de connaître l'état du modem. Ce registre est alimenté par les signaux en provenance du modem et positionnés par ce dernier ;
– le registre d'émission. Il contient le caractère à transmettre à l'ordinateur distant via le modem.

Les informations véhiculées par le bus interne sont transmises en parallèle. La liaison externe utilisée ici est une liaison série et il est donc nécessaire de sérialiser un caractère lorsque l'on émet et de paralléliser les bits à la réception des caractères. Ce travail est effectué par les registres à décalage.

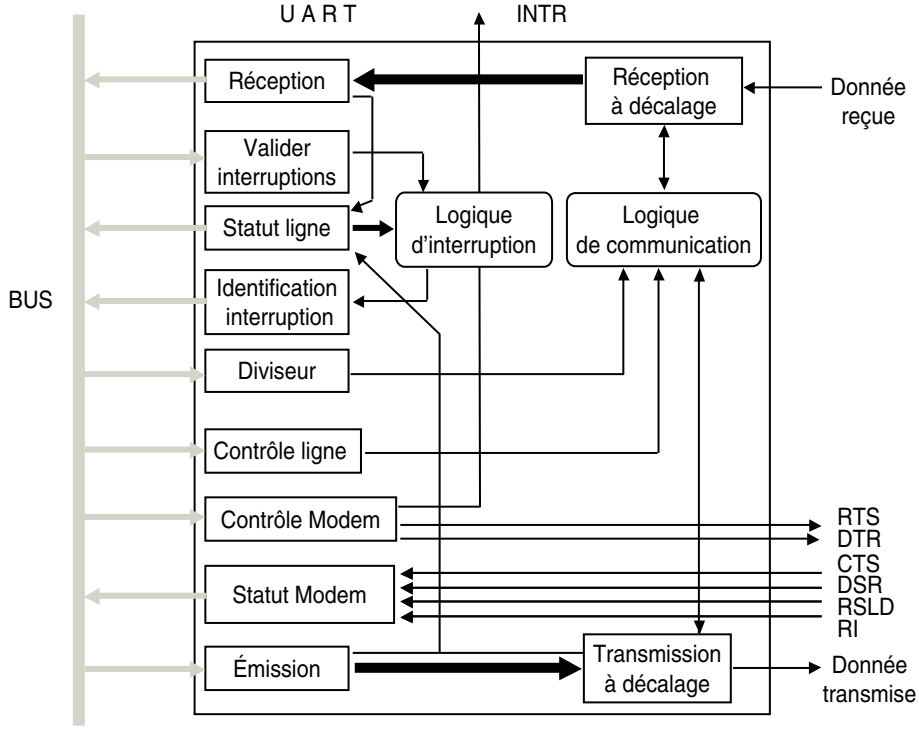


Figure 9.10 Architecture de l'UART.

Enfin l'UART communique avec le modem au moyen de signaux électroniques principalement : RTS (*Request To Send*), DTR (*Data Terminal Ready*), CTS (*Clear To Send*), DSR (*Data Set Ready*), RLSD (*Received Line Signal Detector*) et RI (*Ring Indicator*).

➤ Le connecteur V24 et l'interface de communication standard (RS232)

Ce type de communication série obéit à une norme de connexion entre un Équipement Terminal de Traitement de Données (ETTD : l'ordinateur) et un Équipement Terminal de Circuit de Données (ETCD : le modem). La connexion entre l'ordinateur et le modem se fait par un câble multifils dont chaque extrémité comporte un connecteur de 25 broches (figure 9.11).

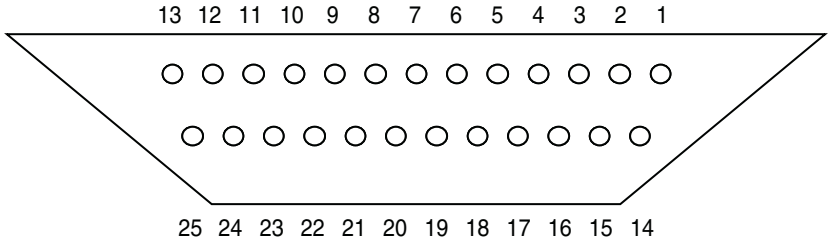


Figure 9.11 Connecteur V24.

La plupart des logiciels de communication n'utilisent pas la totalité des signaux offerts par cette norme. Dans la plupart des situations on n'utilise au maximum une dizaine de signaux. Le tableau 9.2 décrit ces signaux et le brochage des connecteurs utilisés pour une telle connexion.

Tableau 9.2 Signaux et brochage des connecteurs V24.

Signal	Broche	Terre	Contrôle	Données	De	Vers
Terre de protection	1	X				
Terre de signalisation	7	X				
Émission de données	2			X	ETTD	ETCD
Réception de données	3			X	ETCD	ETTD
Demande pour émettre	4		X		ETTD	ETCD
Prêt à émettre	5		X		ETCD	ETTD
Poste de données prêt	6		X		ETCD	ETTD
Terminal de données prêt	20		X		ETTD	ETCD
Indicateur d'appel	22		X		ETCD	ETTD
Détection de signal	8		X		ETCD	ETTD

Les signaux ont la signification suivante :

– terre de protection. C'est un circuit optionnel qui ne joue pas de rôle dans la communication ;
– terre de signalisation. C'est le zéro de référence, les autres signaux ont une valeur positive ou négative par rapport à ce signal ;
– émission de données. C'est le fil sur lequel l'ordinateur envoie des bits au modem ;
– réception de données. C'est le fil par lequel l'ordinateur reçoit les bits en provenance du modem ;
– demande pour émettre (RTS). Lorsque l'ETTD veut émettre il demande l'autorisation de le faire par le biais du signal RTS. L'ETCD répond s'il est prêt en positionnant le signal prêt à émettre (CTS) ;
– prêt à émettre (CTS). L'ETCD répond à une demande pour émettre en indiquant à l'ETTD qu'il est prêt à envoyer des données sur la ligne de données ;
– poste de données prêt (DSR). Ce signal positionné indique à l'ETTD que l'ETCD est connecté au canal de communications et est prêt à émettre ;
– terminal de données prêt (DTR). Par ce signal l'ETTD prévient l'ETCD qu'il est prêt à émettre ;
– indicateur d'appel (RI). Ce signal est utilisé dans les modems autorépondeurs. L'ETCD indique à l'ETTD qu'il reçoit un appel ;
– détection de signal (RLSD). Ce signal est le signal de détection de porteuse.

Muni de ce matériel un programme qui veut transférer des données vers un ordinateur distant doit exécuter le protocole indiqué dans la figure 9.12.

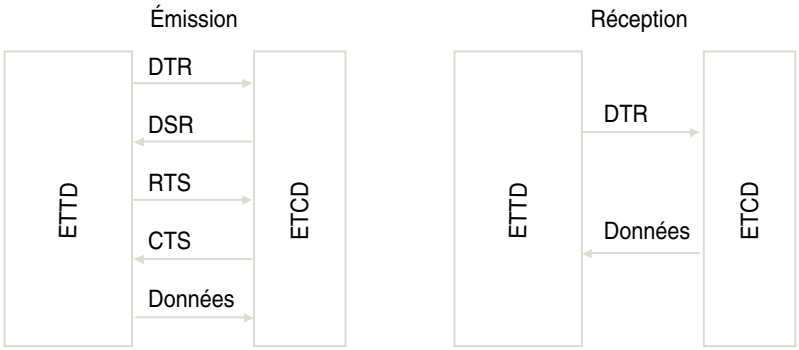


Figure 9.12 Protocole pour le transfert de données (réception).

Le déroulement temporel du protocole est le suivant :

1. l'ordinateur positionne le signal DTR et indique au modem que l'ordinateur est prêt à émettre ;
2. le modem répond par le signal DSR. Cela indique que la connexion avec l'ordinateur distant est établie ;

3. l'ordinateur positionne le signal RTS. Il demande ainsi au modem l'autorisation d'émettre ;
4. le modem positionne le signal CTS ce qui indique qu'il est prêt à émettre ;
5. l'ordinateur émet des données.

Les conditions nécessaires à une réception de données sont beaucoup moins contraignantes. Le protocole de réception est décrit dans la figure 9.12.

Le déroulement temporel de ce protocole est le suivant :

1. l'ordinateur positionne le signal DTR et indique ainsi qu'il est prêt à recevoir ;
2. le modem émet vers l'ordinateur les données lorsqu'elles existent.

Ainsi le modem se contente de retransmettre les données qu'il reçoit aussi longtemps que l'ordinateur est prêt. Cette caractéristique fait dire que le protocole RS232 est orienté émission.

➤ En résumé

Cet exemple a pour but essentiel de mettre en évidence tout ce qui est mis en action lors d'une opération d'entrées-sorties au travers d'une unité d'échange. Cette description, bien que s'appuyant sur un exemple particulier, garde un caractère général. Les éléments essentiels que l'on peut retenir sont :

– sur le plan matériel l'unité d'échange communique avec le processeur et la mémoire centrale au travers des bus par le biais de registres et avec le périphérique au travers de signaux électriques. De ce point de vue les unités d'échanges ont une double nature. Une unité d'échange est adressable au même titre qu'un mot de la mémoire centrale : chaque registre est adressable et donc on peut par programme lire et écrire dans ces registres et donc communiquer avec l'unité d'échange ;
– le système d'exploitation intègre pour chaque unité d'échange un programme spécifique de gestion de chaque unité d'échange : le pilote (driver). Ce pilote peut adresser les registres et c'est lui qui peut lire et écrire dans ces registres. Ainsi lorsqu'un programme utilisateur veut échanger avec un périphérique il s'adresse au pilote et lui indique les données qu'il veut transférer. Le pilote gère alors les échanges au travers du protocole spécifique de gestion des échanges pour cette unité d'échange.

Cet exemple de fonctionnement de cette unité d'échange a été relativement détaillé car il est exemplaire du fonctionnement général des unités d'échanges.

➤ Une application importante utilisant la liaison série

À titre indicatif nous donnons en exemple une application importante utilisant la liaison série, celle de l'accès à un serveur web au travers d'une liaison téléphonique (RTC). La figure 9.13 donne l'architecture générale de cette application.

L'utilisateur dispose d'un compte chez un fournisseur d'accès (*provider*) et il connaît ce provider par un numéro téléphonique. Pour accéder à un serveur web l'utilisateur dispose d'un logiciel, le butineur (*browser*), tel qu'Internet Explorer ou Netscape. Pour communiquer avec un serveur web, l'utilisateur lance l'exécution de son

browser. Ce dernier exécute un programme de connexion au provider, qui connaît l'adresse du provider et exécute le protocole que nous avons précédemment décrit (nous sommes à ce niveau exactement dans la situation décrite dans l'exemple précédent). Le résultat de cette connexion est qu'il existe maintenant une liaison entre le browser de l'utilisateur et le provider. L'utilisateur peut, au travers du browser, préciser l'adresse internet du serveur qu'il veut rejoindre (http ://……) qui est transmise au provider par le lien établi. Le provider au travers des protocoles réseau se connecte au serveur web. Il existe maintenant un lien de bout en bout entre le browser de l'utilisateur et le serveur web. Les échanges peuvent avoir lieu.

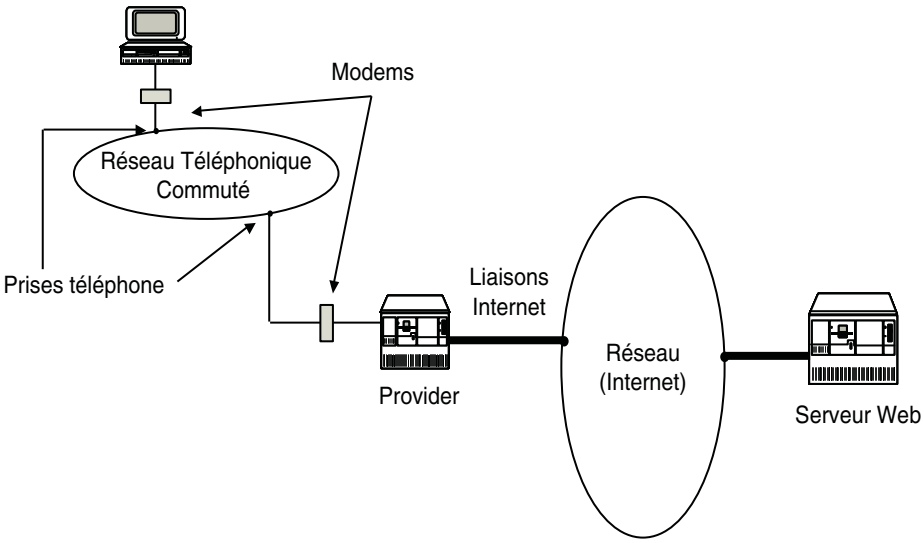


Figure 9.13 Schéma de connexion par le réseau téléphonique commuté.

Exemple 2 : liaison Ethernet

Un autre exemple important est celui de la liaison éthernet permettant le raccordement à un réseau local de type ethernet. La figure 9.14 présente cet exemple.

L'objet ici est seulement de mettre en évidence l'architecture matérielle d'interconnexion. La logique de fonctionnement est la même que celle que nous avons décrite dans l'exemple précédent. La carte éthernet dispose d'une adresse permettant au processeur d'y accéder, le système d'exploitation dispose d'un pilote pour ce type d'unité d'échange et connaît le protocole permettant les échanges de données à partir de ce type d'unité et des périphériques qui y sont connectés.

La carte contrôleur est organisée, comme indiqué sue la figure 9.14, autour de 3 couches :

– la mise en forme des données. Il s'agit ici de regrouper les données dans des structures (trames éthernet) compatibles avec les protocoles de gestion des entrées-sorties pour ce type d'unité d'échange ;

– l'exécution du protocole de communication gérant les échanges (protocole CSMA-CD par exemple) ;
– la production ou l'interprétation des signaux électriques nécessaires à la gestion du média de transmission.

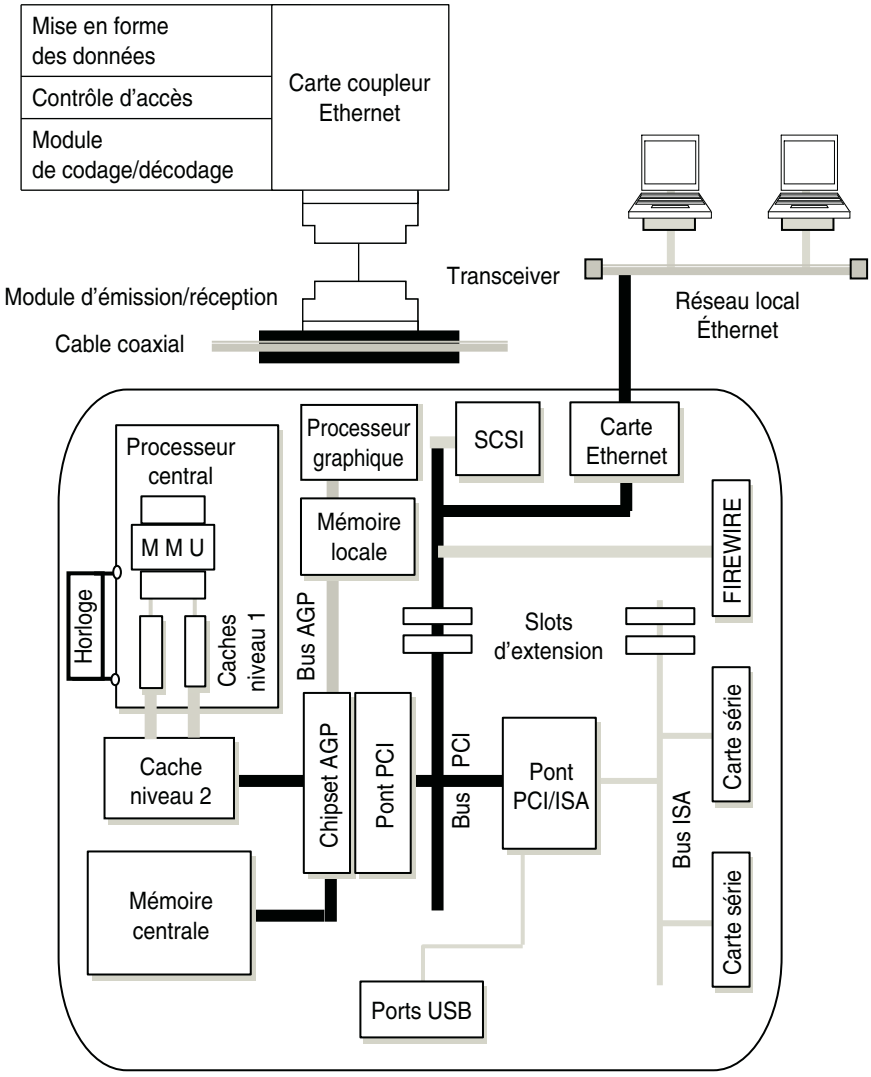


Figure 9.14 Connexion à un réseau éthernet.

La connexion via un contrôleur éthernet n'est pas restreinte à la connexion entre ordinateurs. On peut également grâce à ce type de connexion partager des périphériques tels que des imprimantes et des disques magnétiques. La figure 9.15 illustre ce point.

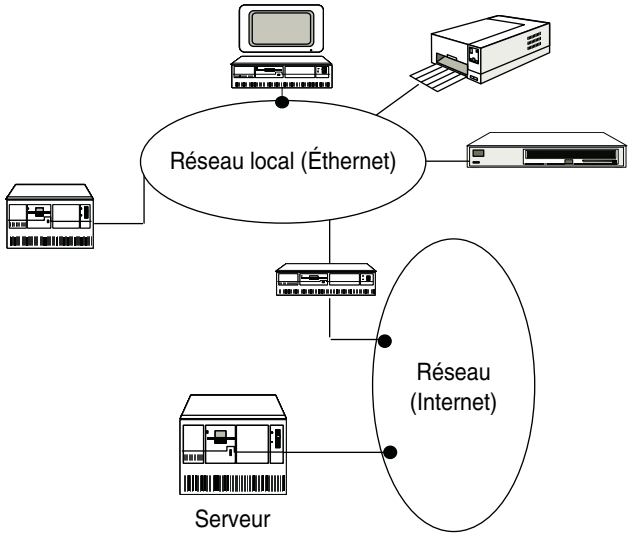


Figure 9.15 Partage de ressources au travers d'une connexion internet.

Ce type de connexion est aussi la principale source de connexion à l'internet au travers des réseaux locaux qui existent généralement dans les universités et les entreprises.

Les contrôleurs de disques magnétiques

Comme la figure 9.4 le montre les disques de type IDE (*Integrated Drive Electronic*) sont directement connectés sur le pont PCI/ISA de notre architecture de base. La structure du contrôleur de ce type de disque correspond à la structure générale des unités d'échanges. La figure 9.16 présente les principaux signaux électriques gérés par ce type de contrôleur. On y trouve les signaux spécifiant en particulier l'adresse d'une information disque (numéro de cylindre, numéro de face, numéro d'unité), des signaux de commande (lecture/écriture) et des indicateurs d'états permettant de

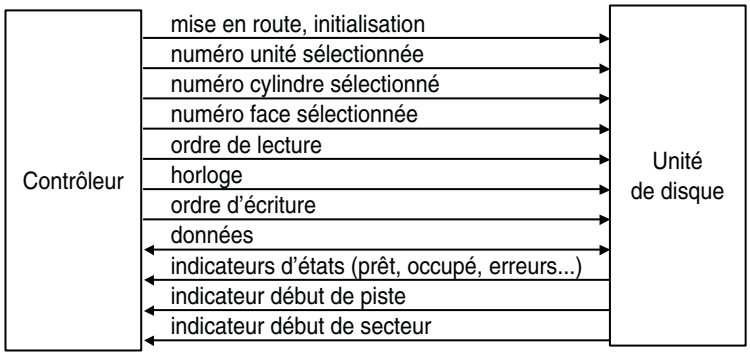


Figure 9.16 Échanges des signaux entre contrôleur et unité de disque.

connaître la position de la tête de lecture/écriture. Grâce à ces d'indicateurs, le pilote peut gérer correctement les opérations d'entrées-sorties pour ce type de périphériques.

Cette sorte de contrôleur est encore relativement utilisée et permet de gérer des disques de 20 à 500 Mo avec un débit d'environ 8 Mo/s. Cependant ce contrôleur est maintenant délaissé au profit du mode EIDE (*Enhanced IDE*) autrement nommé ATA-2, ATA/33 et ATA/66. Ces différents contrôleurs autorisent un fonctionnement en mode DMA (*Direct Memory Access*) permettant des échanges directs avec la mémoire centrale sans utilisation du processeur central. Ce mode offre des taux de transferts allant de 4 à 17 Mo/s. Le tableau 9.3 résume les principales caractéristiques de ces différentes interfaces.

Tableau 9.3 CARACTÉRISTIQUES DES INTERFACES DISQUE.

Interfaces ATA avec DMA	Taux de transferts : Mo/s
DMA	11
Fast ATA	17
ATA/33	33
ATA/66	66

Les cartes graphiques

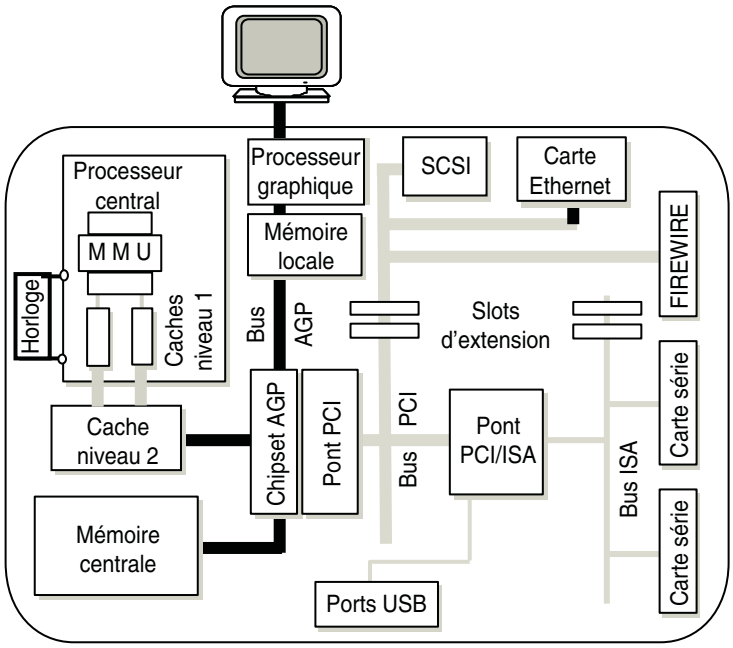


Figure 9.17 Composants principaux gérant une opération d'entrées-sorties graphique.

Les applications graphiques et multimédias utilisant de plus en plus de ressources vidéo sont pour une large part à l'origine de l'architecture actuelle des ordinateurs que nous utilisons. La figure 9.17 nous indique les composants principaux intervenants dans la gestion des opérations d'entrées-sorties graphiques. Le processeur, la mémoire centrale et le contrôleur graphique sont reliés par le biais d'un chipset : le chipset AGP.

Cette architecture permet des échanges extrêmement rapides entre ces trois composants. De plus la mémoire centrale et la mémoire locale de la carte graphique peuvent communiquer directement. La carte graphique est organisée autour d'une mémoire graphique et d'un processeur spécialisé dans l'affichage sur l'écran.

La figure 9.18 précise le contenu et le fonctionnement d'une carte graphique pour ce qui concerne plus particulièrement la gestion des couleurs.

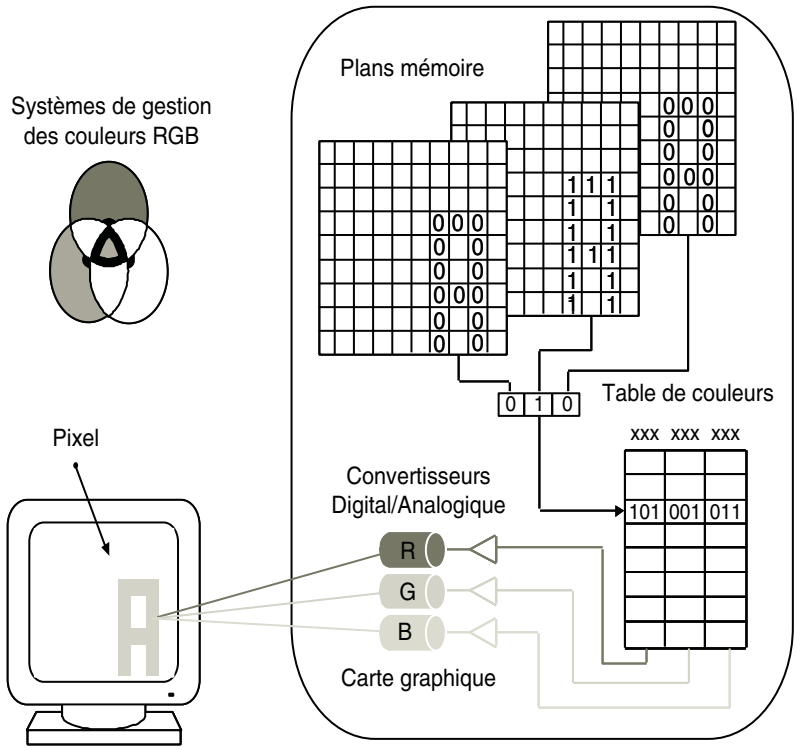


Figure 9.18 Gestion des couleurs par la carte graphique.

L'écran graphique se présente comme un ensemble de *pixels*. Chaque pixel peut s'afficher en plusieurs couleurs. La résolution de l'écran mesure le nombre de lignes multiplié par le nombre de colonnes (un écran d'une résolution de 1 024 par 768 affiche 768 432 pixels). La carte graphique comprend une mémoire d'image (bitmap) composée de plusieurs plans mémoire. Le nombre de bits de chaque plan est le même que le nombre de pixels de l'écran. Le nombre de plan permet d'associer plusieurs bits à

chaque pixel; dans le cas où nous n'aurions qu'un plan mémoire nous aurions un affichage monochrome, un pixel étant allumé ou éteint.

Notre exemple présente une méthode d'attribution des couleurs à un pixel. Cette méthode utilise une table de couleurs et 3 plans mémoire. Chaque pixel est codé sur 3 bits, ce qui donne 2^3 combinaisons différentes. La table de couleurs a 8 entrées, chaque entrée a 9 bits, 3 pour le rouge, 3 pour le vert, 3 pour le bleu. Pour chaque entrée on a donc 2^9 combinaisons différentes. Les 3 bits pour le rouge forment l'entrée d'un convertisseur digital/analogique qui produit un faisceau électrique dont l'intensité dépend de la valeur représentée par les 3 bits pour le rouge. On dispose également d'un convertisseur pour le vert et un pour le bleu. Les trois faisceaux convergent sur l'écran pour allumer le pixel correspondant. La couleur du pixel dépend de l'intensité relative des différents faisceaux rouge/vert/bleu. Notre exemple permet donc d'afficher 256 couleurs différentes (autant que d'entrées dans la table des couleurs) avec pour chaque couleur 2^9 teintes différentes.

Au plan de la gestion de l'affichage, pour qu'une image soit stable, il faut afficher l'image au moins 25 fois par seconde ce qui marque la nécessité d'un processeur spécialisé pour garantir correctement cet affichage. Pour un écran de $1\,024 \times 768$ pixels si la mémoire d'image dispose de trois plans (ce qui est peu aujourd'hui) et que chaque ligne de la table des couleurs a 24 bits (ce qui est le cas le plus fréquent) le processeur doit balayer 25 fois par seconde cette bitmap en concordance avec la table des couleurs pour afficher une image stable.

La gestion des images animées (vidéo) implique des modifications de la bitmap, aussi au-delà de la nécessité du processeur d'affichage on comprend l'intérêt d'un bus très rapide entre la mémoire principale et la mémoire locale afin de garantir la fluidité des mouvements.

9.3.2 Les bus d'extension

Au-delà des unités d'échanges il existe d'autres interfaces de communication avec les périphériques. On détaillera plus particulièrement le bus USB afin de mettre en évidence les caractéristiques principales de ce type de bus et l'on donnera quelques indications techniques sur les bus SCSI et FIREWIRE.

Le bus série USB

Le bus PCI est très performant pour les périphériques à haut débit. Par contre dans le cas de périphériques lents il n'est pas nécessaire d'avoir les débits que ce bus permet. De plus avec le bus PCI on ne dispose que d'un nombre restreint de slots d'extension ce qui en limite son utilisation. Sept compagnies (Compacq, DEC, IBM, INTEL, Microsoft, NEC, Northern Telecom) sont à la base de la création du bus série USB (*Universal Serial Bus*). L'objectif est de simplifier l'interface d'accès avec de nombreux périphériques en permettant de connecter une majorité de périphériques à un connecteur unique en lieu et place des multiples connecteurs tels que les connecteurs séries, parallèles, souris, microphones, etc. Seuls resteraient les connecteurs parallèles et ceux permettant le raccordement des périphériques à hauts débits.

Le cahier des charges devait suivre un certain nombre de points : avoir un seul type de câble, avoir une alimentation par le câble, pouvoir raccorder 127 périphériques, accepter les périphériques fonctionnant en temps réel, garantir le Plug And Play et ne pas avoir à relancer le système d'exploitation après le branchement à chaud d'un périphérique.

➤ Les caractéristiques physiques

La figure 9.19 présente l'organisation générale du bus USB :

– la topologie du bus USB est une organisation arborescente. Un contrôleur principal (*root hub*) est connecté soit au bus PCI soit, comme sur notre figure, à un pont. Ce contrôleur comprend des connecteurs permettant le raccordement soit de connecteurs secondaires soit directement de périphériques ;
– il existe deux types de connecteurs, les connecteurs des contrôleurs et les connecteurs pour les périphériques. On ne peut brancher deux connecteurs de même type entre eux ;
– le câble qui supporte le bus est constitué de 4 fils. Deux fils sont dédiés au transport des données, un autre à l'alimentation, le dernier à la masse ;

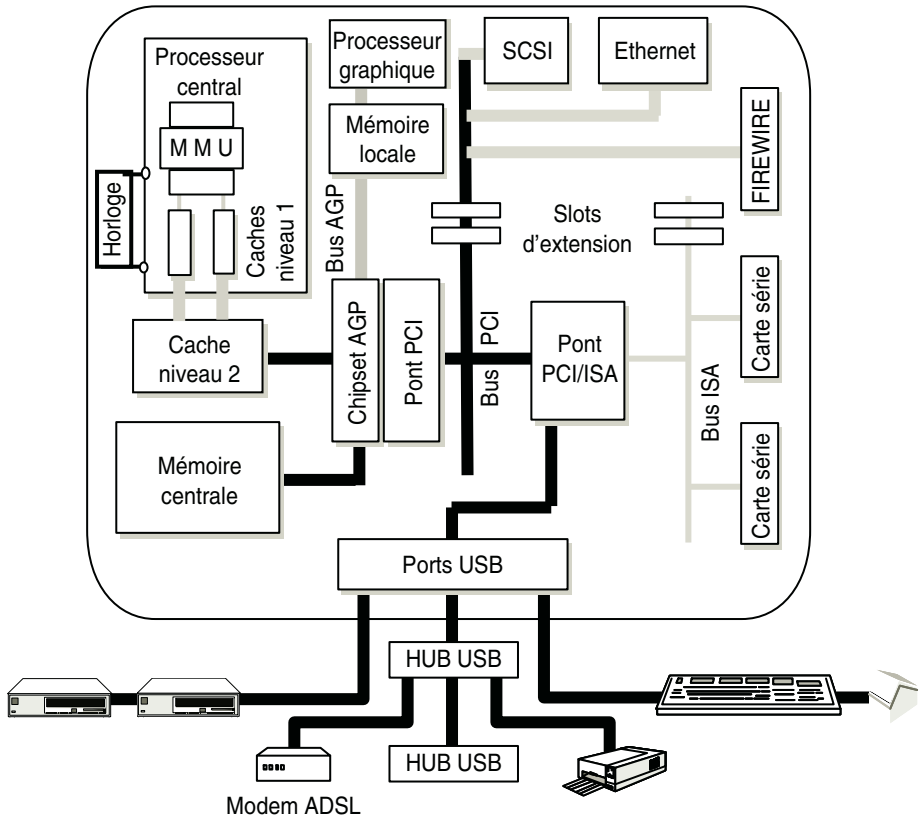


Figure 9.19 Le bus USB.

– la bande passante est de 1.5 à 12 Mbits/s. Les futures évolutions prévoient un débit 30 à 40 fois plus important ;
– on peut connecter jusqu'à 127 périphériques sur le bus USB.

➤ Fonctionnement

Fonctionnellement le bus doit être considéré comme un canal permettant d'effectuer des échanges en mode série entre le contrôleur principal et les périphériques. Chaque périphérique peut subdiviser son canal en plusieurs sous-canaux ce qui permet pour un même périphérique de distinguer des flux d'entrées et de sorties (par exemple pour séparer les flux d'entrées et de sorties d'un dispositif d'acquisition et de restitution du son).

Aucune communication n'est possible directement entre deux périphériques. Elle doit impérativement passer par le contrôleur principal. Sur chaque canal ou sous-canal, les communications sont unidirectionnelles ou bidirectionnelles. Les échanges d'informations se font au travers de trames.

Il existe quatre types de trames :

– Les *trames de supervision* – Elles servent à configurer les équipements et demander aux périphériques des informations sur leur état.
– Les *trames isochrones* – Elles concernent plus particulièrement des périphériques de type temps réels qui ne nécessitent pas de réémission de l'information en cas d'erreur de transmission (par exemple les périphériques audios).
– Les *trames de données* – Ce sont les trames transportant les données, par exemple pour une imprimante. Dans ce type de transmission on doit pouvoir réémettre des données en cas d'erreur de transmission.
– Les *trames d'interruption* – Le bus USB ne dispose pas d'un mécanisme d'interruption intégré. Traditionnellement lorsque l'on appuie sur une touche d'un clavier une interruption est générée. Lorsque le clavier est connecté au travers d'un bus USB ce n'est pas le cas. Régulièrement le système d'exploitation (le pilote du bus USB) interroge le clavier pour savoir si une touche a été enfoncée.

Les trames sont constituées de paquets d'informations. On distingue plus particulièrement :

– Les paquets de commandes. Seul le contrôleur principal peut émettre ce type de paquets. On trouve principalement les paquets :
  – SOF (*Start Of Frame*) qui indique le début de transmission d'une trame. S'il n'y a pas de données le paquet SOF est seul dans la trame ;
  – IN qui est un paquet d'interrogation d'un périphérique lui demandant d'envoyer des informations. Dans ce paquet est précisé le canal ou le sous-canal concerné ;
  – OUT est un paquet indiquant qu'une donnée à destination d'un périphérique suit.
– Les paquets de données. Ce sont les paquets qui transportent les données. Ils ont la structure donnée par la figure 9.20. SYN est un champ de synchronisation, PID spécifie le type du paquet et CRC est un champ de contrôle des erreurs.
– Les paquets d'acquittement. Il en existe trois types :
  – ACK qui indique que la transmission précédente était correcte ;

– NAK qui indique une erreur de transmission dans le paquet précédent, détectée grâce au CRC;
– STALL qui est une demande de mise en attente pour périphérique occupé.

SYN (8 bits)	PID (8 bits)	Données (1 à 1 023 octets)	CRC (16 bits)

Figure 9.20 Structure d'un paquet de données.

Le contrôleur principal émet régulièrement, toutes les millisecondes, une trame. Il s'agit soit d'une trame de synchronisation, soit d'une trame d'information. Lors d'une communication la première trame est toujours à l'initiative du contrôleur principal vers un périphérique, les trames suivantes peuvent provenir du contrôleur principal, d'un périphérique ou d'un autre contrôleur selon la logique de la communication.

Ainsi lorsque l'on branche un nouveau périphérique le contrôleur principal, par le biais des trames qu'il diffuse régulièrement, détecte celui-ci et émet une interruption vers le pilote (système d'exploitation) du bus USB. Le pilote par scrutation mesure les besoins en bande passante du nouveau périphérique et détermine s'il y a assez de bande passante. Dans l'affirmative il donne une adresse au périphérique (numéro entre 1 et 127) puis il transmet ces informations au nouvel équipement qui est maintenant identifié dans le système et pourra être reconnu. Ce mécanisme permet d'insérer dynamiquement un nouveau périphérique sans avoir à arrêter le système.

Régulièrement le contrôleur principal diffuse des trames. S'il n'y a rien à faire les trames se réduisent au paquet SOF. Dans le cas d'une demande de données à partir d'un périphérique, par exemple obtenir les images en provenance d'un appareil photo, l'enchaînement des paquets est celui donné par la figure 9.21. La partie grisée indique les paquets émis par le périphérique vers l'ordinateur. Le paquet ACK indique que l'ordinateur a bien reçu le paquet précédent. Le premier et le dernier SOF sont des paquets de synchronisation : il n'y a rien à faire. Le deuxième SOF suivi de IN indique la demande de données au périphérique.

SOF	SOF	IN	SYN	PID	Données	CRC	ACK	SOF

Figure 9.21 Enchaînement des paquets pour une demande de données à partir d'un périphérique.

La figure 9.22 illustre la succession de paquets pour une opération d'écriture par exemple sur une imprimante. En grisé apparaît le paquet d'acquittement émis par le périphérique en réponse aux paquets de données émis par l'ordinateur.

SOF	OUT	SYN	PID	Données	CRC	ACK

Figure 9.22 Enchaînement des paquets pour une écriture sur une imprimante.

Les bus parallèles SCSI

Le bus SCSI (*Small Computer System Interface*) est un bus parallèle performant permettant d'interfacer plusieurs types différents de périphériques : disques durs, lecteur de CD-ROM, scanners… Cette interface est commune à beaucoup d'ordinateurs, PC, Macintosh, SUN etc.

Sa vitesse de transfert varie de 4 Mo/s à 80 Mo/s selon la largeur du bus et le standard SCSI utilisé. Il permet les échanges directs entre deux périphériques sans intervention du processeur central. Comparativement aux technologies EIDE, le bus SCSI est très onéreux et malgré ses caractéristiques et ses performances très intéressantes, il reste plutôt réservé aux ordinateurs hauts de gamme (les serveurs plutôt que les ordinateurs domestiques). Le tableau 9.4 résume les caractéristiques des différents standards SCSI.

Tableau 9.4 CARACTÉRISTIQUES DES STANDARDS SCSI.

		Bande passante : Mo/s		
	Fréquence bus : MHz	**Largeur 8 bits**	**Largeur 16 bits**	**Nombre de périphériques supportés**
SCSI 1	5	5	–	7
SCSI 2	10	10	20	7
Ultra SCSI	20	20	40	7
Ultra 2 SCSI	40	40	80	31

Le bus série FIREWIRE (IEEE 1394)

Ce bus est aussi appelé bus SCSI série. Il est destiné aux périphériques rapides. Il est apparu d'abord sur les machines Apple et a été standardisé en 1995. Il permet la connexion de périphériques en tout numériques n'imposant plus de conversions analogique/digital. Comme dans le cas du bus USB les connexions sont de type Plug And Play et ne nécessitent pas de redémarrage du système lors de la connexion d'un nouveau périphérique. Les performances sont élevées, actuellement de l'ordre de 50 Mo/s soit 20 fois plus élevées que le bus USB (des évolutions sont prévues pour augmenter les débits et atteindre 100 voire 200 Mo/s). On peut connecter jusqu'à 63 périphériques (caméscopes, imprimantes, lecteur de DVD…). C'est un bus asynchrone qui peut supporter des transferts isochrones, par exemple pour la vidéo. Comme dans le cas du bus USB la mise en place de tels bus permet de diminuer considérablement les connecteurs d'entrées-sorties.

9.4 LES DIFFÉRENTS MODÈLES DE GESTION DES ENTRÉES-SORTIES

Dans les sections précédentes nous avons passé en revue tous les participants à la gestion d'une opération d'entrées-sorties :

– le pilote. C'est le programme du système d'exploitation qui exécute les instructions d'entrées-sorties ;
– les bus permettant le transport des informations entre les différents composants ;
– les interfaces d'entrées-sorties. Ce sont les composants qui assurent l'interfaçage entre les composants internes à l'ordinateur et les périphériques ;
– les périphériques.

Nous allons dans cette section revenir sur la notion de pilote pour préciser plusieurs points fondamentaux.

En premier lieu nous avons dit qu'une demande d'opération d'entrées-sorties effectuée par un programme utilisateur (comme par exemple un traitement de texte ou un programme spécifique écrit dans un langage de haut niveau) était traduite par le compilateur en un appel à un programme spécifique : le programme pilote gestionnaire de cette opération d'entrées-sorties. Ainsi il existe simultanément deux types de programmes en mémoire centrale : les programmes du noyau du système d'exploitation (en particulier les pilotes) et le programme utilisateur. Comme nous l'avons vu il n'existe qu'un processeur central et les programmes doivent se partager cette ressource unique. Le mécanisme de gestion d'une opération d'entrées-sorties consiste à interrompre l'exécution du programme utilisateur au profit du programme pilote de l'opération d'entrées-sorties puis après le traitement de l'opération d'entrées-sorties à reprendre l'exécution du programme utilisateur. La réalisation de ce schéma d'exécution repose sur le *mécanisme de basculement de contexte matériel*. Un programme qui s'exécute dispose du processeur matériel c'est-à-dire de l'ensemble des registres du processeur (tout particulièrement du compteur ordinal qui contient l'adresse de la prochaine instruction à exécuter). Lorsque l'on interrompt l'exécution d'un programme au profit d'un autre il faut sauvegarder en mémoire ce contexte matériel d'exécution afin de pouvoir reprendre l'exécution ultérieurement, là où on l'a abandonnée. Ce contexte étant sauvegardé on peut charger les registres avec le contexte d'exécution du nouveau programme et tout particulièrement : placer dans le compteur ordinal l'adresse de la première instruction de ce nouveau programme. Le nouveau programme s'exécute alors sur le processeur matériel. Ce mécanisme tout à fait fondamental est, comme nous le verrons plus tard, à la base des systèmes d'exploitation multiprogrammés.

En deuxième lieu nous devons examiner les différentes manières dont peut s'exécuter un programme pilote. Nous distinguons trois méthodes de gestion des entrées-sorties : la liaison programmée, les entrées-sorties pilotées par les interruptions, l'utilisation d'un dispositif permettant des accès directs à la mémoire, le DMA.

9.4.1 La liaison programmée

Dans ce mode d'échanges, le pilote utilise totalement le processeur central pour contrôler et piloter les échanges avec le périphérique. Nous illustrons ce mode en prenant l'exemple d'un pilote d'imprimante utilisant une carte série (unité d'échange). L'unité d'échange a deux registres, un registre d'état et un registre de données d'une largeur d'un octet (un caractère). Ces caractéristiques sont les caractéristiques géné-

rales d'une carte série (transmission série caractère par caractère). Une telle unité d'échange peut fonctionner en utilisant le mécanisme d'interruption ou en ne l'utilisant pas. Dans cet exemple nous utilisons cette carte série en mode sans interruption, nous examinerons dans l'exemple suivant ce qu'induit le fait de fonctionner en mode avec interruptions.

Pour réaliser une telle opération d'entrées-sorties le pilote exécute l'algorithme suivant qui donne la logique du programme machine correspondant au pilote. L'application de l'utilisateur est par exemple un programme de traitement de texte (par exemple Word) et l'utilisateur décide d'imprimer, dans ce contexte, un fichier de caractères.

```
lire registre_état;
répéter
tant que périphérique_occupé;
faire
lire registre_état;
finfaire
écrire un caractère dans registre de données;
jusqu'à ce qu'il n'y ait plus de caractères à imprimer.
```

Dans cet algorithme le pilote lit le registre d'état qui contient un code caractéristique de l'état de l'imprimante. Ce code peut par exemple indiquer qu'il n'y a plus de papier ; le pilote affichera alors un message pour l'utilisateur indiquant qu'il n'y a plus de papier. Ce code peut également indiquer que l'imprimante imprime actuellement un caractère et qu'elle n'est donc pas disponible pour imprimer un autre caractère. Dès que le caractère est imprimé l'imprimante devient libre et elle le signale en positionnant le contenu du registre d'état à une valeur qui signifie : imprimante libre. C'est cette situation que nous examinons.

Cet algorithme est donc organisé comme deux boucles itératives imbriquées. La plus extérieure (`répéter… jusqu'à`) contrôle le nombre de caractères à imprimer. La boucle intérieure (`tantque… faire… finfaire`) exprime que tant que le registre d'état indique que l'imprimante est occupée le pilote relit le registre d'état pour savoir quand l'imprimante va être disponible.

La figure 9.23 illustre cette exécution. Le programme Word est en cours d'exécution. Lorsque l'utilisateur clique sur l'icône d'impression, cette action se traduit par un appel au pilote gestionnaire de l'imprimante. Le pilote reçoit le nombre de caractères à imprimer. Pour pouvoir s'exécuter le pilote doit disposer du processeur, il faut donc sauvegarder le contexte matériel d'exécution du programme Word (2). Le pilote réalise l'opération d'entrées-sorties par exécution de l'algorithme précédent (3). À la fin de cette exécution le pilote recharge dans le processeur le contexte d'exécution de Word qui reprend donc son exécution à l'endroit où il avait été interrompu (4).

Ainsi pendant tout le temps de l'opération d'entrées-sorties, le processeur est occupé : il exécute les instructions de la boucle `tant… que`. On dit que le processeur fait une attente active : il attend que le périphérique soit libre en exécutant des instructions. Ce mode de gestion est inefficace puisque globalement les performances

de l'ordinateur sont déterminées par celles du périphérique : si le processeur est très rapide (haute fréquence) il exécutera beaucoup d'instructions machine mais ces instructions ne seront que des instructions de contrôle de l'état du périphérique et ne sont donc pas très productives. Le schéma temporel d'exécution montre que si le processeur est constamment occupé ce n'est pas l'utilisateur qui en profite : Word ne peut travailler pendant le déroulement de l'impression et ainsi on ne peut saisir des caractères pendant l'impression.

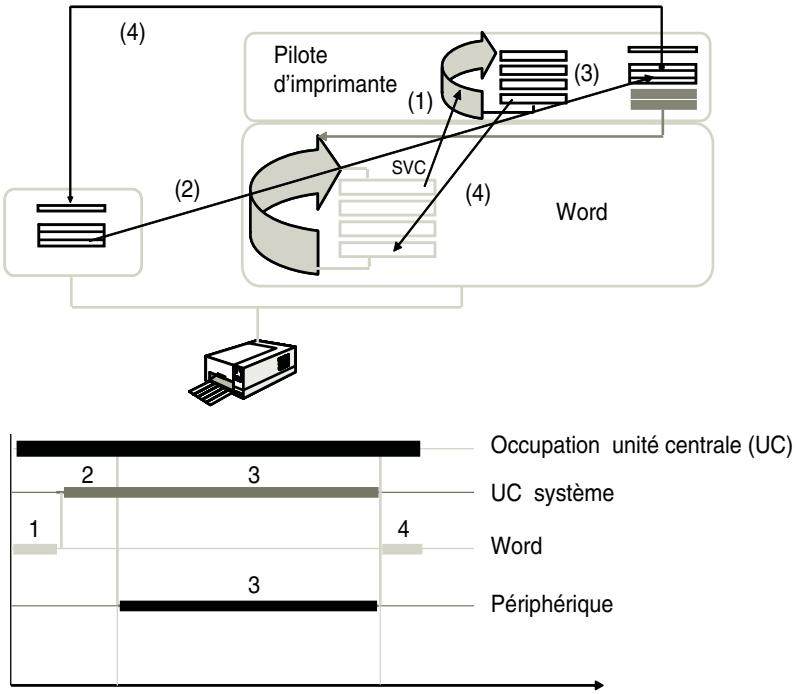


Figure 9.23 Entrées-sorties programmées.

9.4.2 Entrées-sorties pilotées par les interruptions

Afin d'améliorer les performances de l'ordinateur lors de l'exécution d'une opération d'entrées-sorties, on utilise le mécanisme d'interruptions. Dans ce mode de gestion l'unité d'échange utilise le mécanisme des interruptions pour signaler qu'elle est prête. Ainsi, à chaque fois qu'un caractère est imprimé, l'imprimante le signale à l'unité d'échange qui émet une interruption vers le processeur. Pour prendre en charge les interruptions le système d'exploitation dispose d'un ensemble de programmes de gestion des interruptions. À la réception d'une interruption le programme en cours d'exécution est arrêté au profit du programme de gestion de l'interruption. La gestion de l'opération d'entrées-sorties est donc très différente comme le montre la figure 9.24.

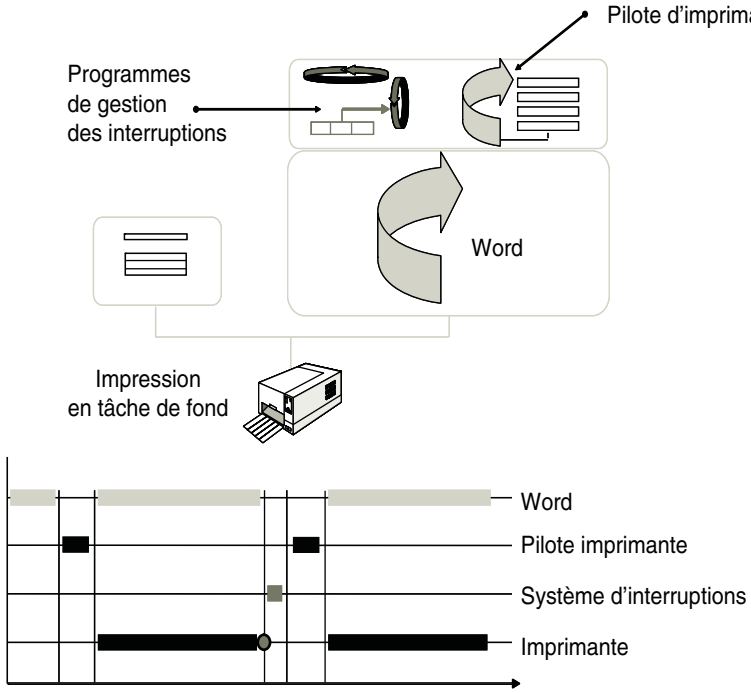


Figure 9.24 Entrées-sorties pilotées par interruption.

Le schéma temporel d'exécution est maintenant :

– Word est en cours d'exécution et l'utilisateur clique sur l'icône d'impression.
– Le pilote est alerté et vérifie que l'imprimante est libre. Si l'imprimante est libre le pilote charge le registre de données et l'impression du premier caractère est lancée.
– À partir de ce moment le processeur est libre. Il n'a pas à vérifier si l'imprimante est prête car il sera prévenu par une interruption. Le processeur peut donc être rendu au programme Word qui permettra la saisie de caractères pendant l'impression d'un caractère.
– À la fin de l'impression du caractère l'exécution de Word est interrompue au profit du programme de gestion de l'interruption. Ce traitement se termine par un appel au pilote qui vérifie s'il reste des caractères à imprimer. Si c'est le cas il place un nouveau caractère dans le registre de données et l'impression du caractère suivant commence.
– Le processeur est de nouveau attribué à Word.

Ce processus se poursuit tant qu'il y a des caractères à imprimer. On voit donc que la prise en compte de l'opération d'entrées-sorties selon ce mode est très différente du point de vue de l'utilisateur. Il possède en permanence le processeur sauf pendant les périodes où le processeur est attribué au programme de gestion de l'interruption et du pilote. Ce traitement de l'impression est dit *traitement en tâche de fond* puisque

l'utilisateur a l'impression de travailler en permanence avec Word y compris pendant la phase d'impression.

9.4.3 Gestion des entrées-sorties asynchrones

L'efficacité du mécanisme précédent est évidente. Toutefois il ne faut pas que le temps utilisé par le processeur pour le programme de gestion de l'interruption et du pilote soit trop important. Une solution est de diminuer le nombre d'interruptions, par exemple l'imprimante ne génère plus une interruption à chaque caractère mais uniquement à la fin de l'impression d'une ligne (voire plusieurs lignes). Ceci implique d'une part que l'imprimante dispose d'un tampon mémoire permettant le stockage d'une ou plusieurs lignes et d'autre part que le chargement d'une ou plusieurs lignes à partir de la mémoire principale se fasse sans utilisation du processeur central. Dans ces conditions le processeur est libre durant tout le temps de l'opération d'entrées-sorties et peut donc être utilisé à autre chose.

Accès direct à la mémoire (DMA)

Le dispositif DMA (*Direct Memory Access*) est un composant matériel permettant d'effectuer des échanges entre mémoire centrale et unité d'échange sans utilisation du processeur central (figure 9.25).

Le DMA comprend :

– un registre d'adresse qui reçoit l'adresse du premier caractère à transférer ;
– un registre de comptage qui reçoit le nombre de caractères à transférer ;

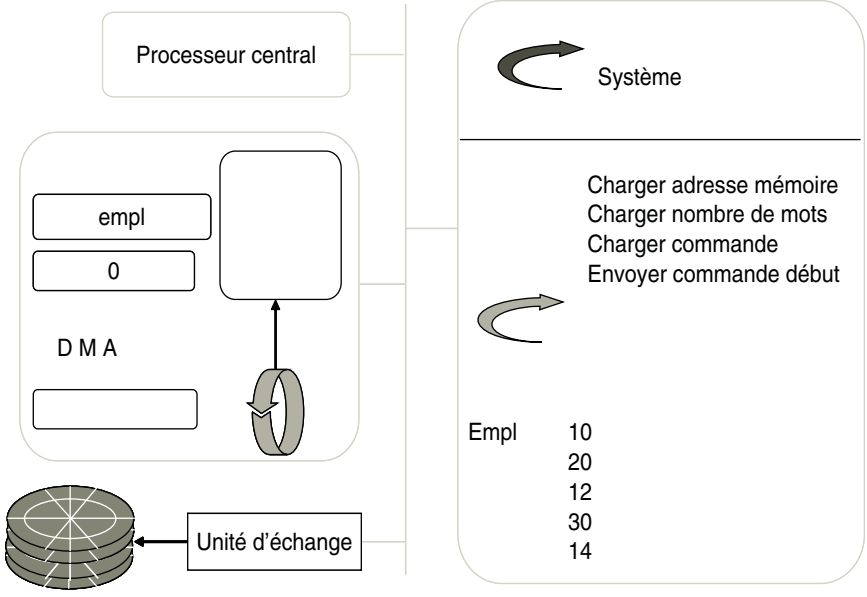


Figure 9.25 Mécanisme de DMA.

– un registre de commande qui reçoit le type d'opération à effectuer (lecture ou écriture) ;
– une zone tampon permettant le stockage de données ;
– un composant actif, de type processeur, qui exécute un transfert sans utilisation du processeur central.

Le programme pilote est alors très simple et effectue simplement les opérations d'initialisation du DMA : chargement des registres puis lancement du processeur périphérique. À partir de ce moment le transfert s'effectue sans utilisation du processeur central qui est alors libre d'effectuer un autre travail. À la fin du transfert une interruption est émise qui interrompt le programme en cours d'exécution au profit du programme de gestion de l'interruption.

La figure 9.26 résume une opération d'entrées-sorties gérée à l'aide d'un DMA associé au mécanisme d'interruption :

– le programme d'écriture sur disque est compilé et chargé en mémoire. Les ordres d'entrées-sorties sont traduits en langage machine sous la forme d'un appel au système d'exploitation (le pilote) (phases 0 et 1) ;
– le programme s'exécute et déclenche l'appel au système pour réaliser l'opération d'entrées-sorties. L'exécution du SVC est un appel au pilote et produit la sauvegarde du contexte matériel d'exécution du programme utilisateur (phases 2 et 3) ;
– le pilote s'exécute en plaçant tout ou partie des informations à transférer dans une zone mémoire réservée au DMA (phase 4). Il faut en effet penser que le proces-

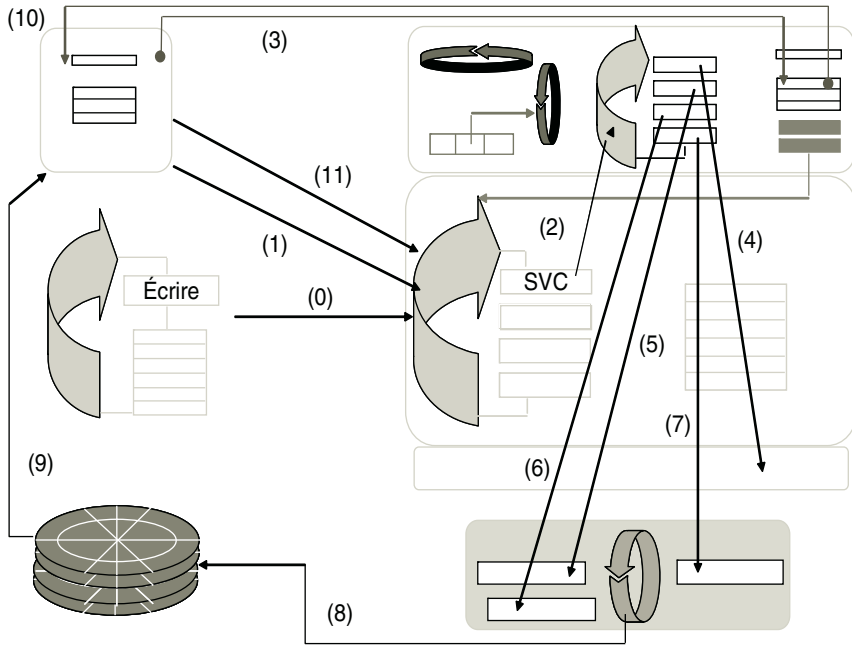


Figure 9.26 Opération d'entrées-sorties gérée par DMA.

seur du DMA va exécuter des instructions lisant des données en mémoire sans utilisation du processeur central. Il va donc y avoir deux processeurs qui peuvent simultanément utiliser la ressource mémoire ; dans ce contexte, réserver une zone mémoire au DMA c'est éviter les conflits d'accès à la mémoire ;

– le DMA initialise le DMA (phase 5, 6, 7) ;
– le DMA réalise l'échange (phase 8) ;
– en fin de transfert, une interruption est émise (phase 9) ;
– l'opération d'entrées-sorties se termine alors par la restitution du contexte d'exécution du programme utilisateur qui reprend donc son exécution (phase 10).

Le schéma temporel d'exécution est donné par la figure 9.27.

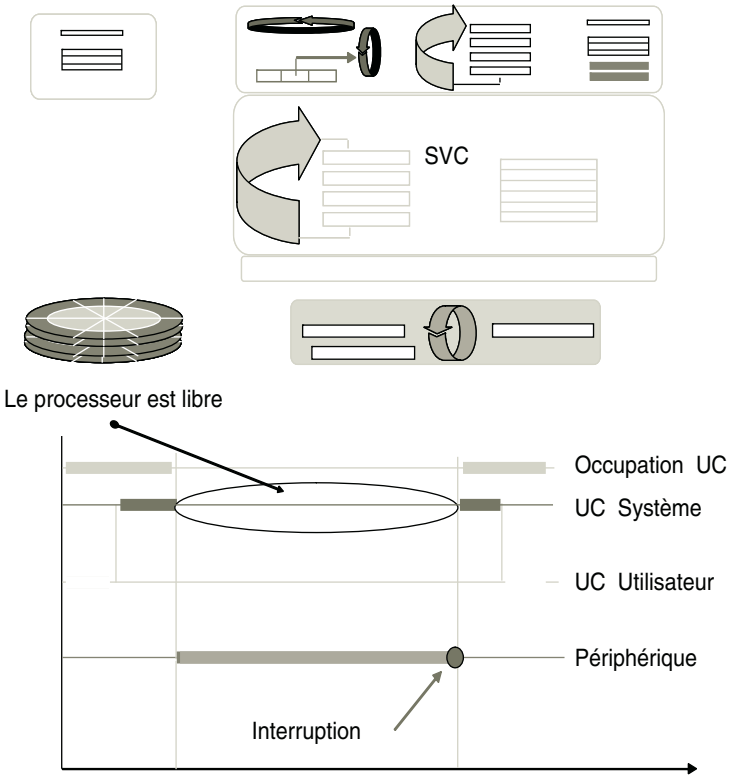


Figure 9.27 Schéma temporel d'exécution avec un mécanisme de DMA.

Ce schéma indique tout particulièrement que pendant tout le temps de l'opération d'entrées-sorties le processeur central est libre. C'est cette constatation qui rend possible la construction des systèmes d'exploitation multiprogrammés. En effet c'est le système d'exploitation qui connaît très exactement le moment où une opération d'entrées-sorties est déclenchée (pilote). De la même manière le système d'exploitation connaît parfaitement la fin d'une opération d'entrées-sorties (programme d'inter-

ruption). Ainsi le système d'exploitation a la connaissance du temps pendant lequel le processeur est libéré lors d'une opération d'entrées-sorties. Dans ces conditions s'il existe d'autres programmes utilisateurs dans la mémoire centrale le système d'exploitation peut attribuer le processeur à un autre programme durant l'opération d'entrées-sorties du premier utilisateur.

9.5 CONCLUSION

Cette constatation nous permet de conclure ce chapitre sur la fonction de communication et d'introduire les chapitres sur les systèmes d'exploitation qui assureront donc des fonctions de gestion et de partage des ressources matérielles d'un ordinateur entre un ou plusieurs utilisateurs. Parmi les fonctions importantes seront traitées :

- la gestion du processeur, c'est-à-dire le partage du processeur entre plusieurs programmes utilisateurs ;
- la gestion de la mémoire, c'est-à-dire le partage de cette ressource entre plusieurs programmes en mémoire en étant capable d'assurer une gestion harmonieuse et sécurisée de cette ressource ;
- le système de gestion de fichiers et des entrées-sorties qui pilote les échanges avec les périphériques.

Chapitre **10**

Exercices corrigés

LA FONCTION D'EXÉCUTION

10.1 Révision

Cet exercice ne donnera pas lieu à une correction particulière, les réponses se trouvent facilement en parcourant le chapitre sur la fonction d'exécution.

1. Rappelez les différentes étapes de l'exécution d'une instruction machine en indiquant la fonction de chacune de ces étapes.
2. Est-ce que l'étape de chargement d'une instruction est dépendante du type de l'instruction ?
3. Quel est le rôle du séquenceur ? Quels sont les avantages et les inconvénients du séquenceur câblé par rapport au séquenceur microprogrammé ?

10.2 Microcommandes

Soit une machine dont les instructions ont le format « codeopération, modeadressage, registre, champopérande » avec, par exemple, Im pour modeadressage immédiat, I pour modeadressage indirect, R1 pour registre R1. Par ailleurs la microcommande InCo permet l'incrémentation du compteur ordinal.

1. Nous disposons de la machine matérielle décrite dans le cours (machine à un bus de la figure 7.5) et nous conservons le formalisme du cours pour indiquer la nature des signaux positionnés. Soient les instructions suivantes :

```
Load Im, R1, x      (charger dans R1 la valeur x)
Load D, R1, x       (chargement direct dans R1)
Load I, R1, x       (chargement indirect de R1)
```

Pour chacune de ces instructions donnez la séquence des microcommandes permettant leur exécution. Quelles conclusions peut-on tirer en matière de temps d'exécution ?

Même question pour :

```
Add D, R1, x      (additionner le contenu de R1 avec le contenu de x
                  ➥ le résultat étant dans R1)
```

2. Nous disposons à présent d'une machine ayant le même langage machine que précédemment mais avec une architecture plus complexe à 3 bus (figure 10.1). De la même manière que précédemment, InCo permet l'incrémentation du compteur ordinal. Pour passer d'un bus à l'autre on doit exécuter une opération NOP (Non Operation), plus précisément passer de A à B se fait en exécutant un NOP et passer de A à C se fait en exécutant deux NOP.

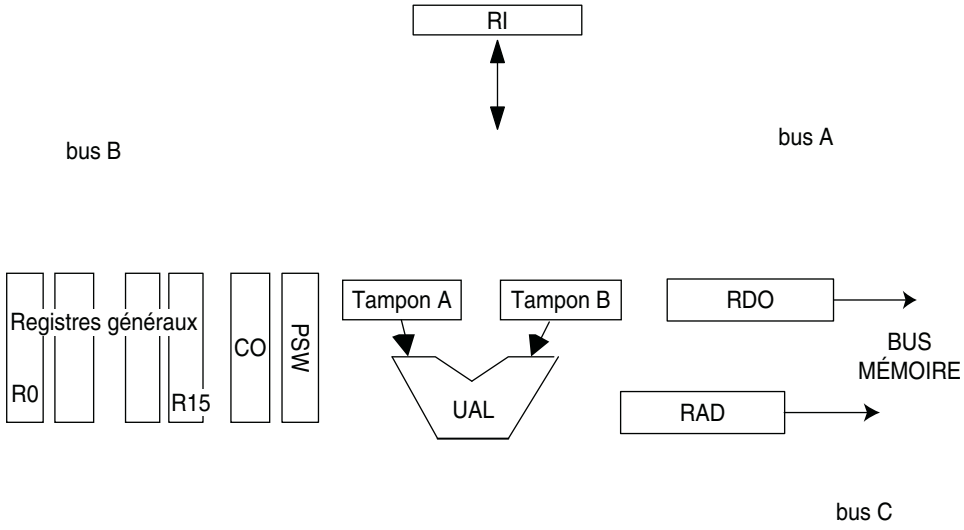


Figure 10.1 Architecture à 3 bus.

Soient les instructions :

```
Load D, R1, x     (chargement direct dans R1)
Add D, R1, x      (additionner le contenu de R1 avec le contenu de x
                  ➥ le résultat étant dans R1)
```

Donnez la séquence des microcommandes permettant leur exécution.

10.3 CISC/RISC

Soit le programme suivant à exécuter : C = A + B (prendre le contenu de A lui ajouter le contenu de B placer le résultat dans C).

1. Dans le cas où l'on dispose d'une machine CISC munie d'un jeu d'instructions du type « codeopération, modeadressage, registre, champopérande », donnez la liste des instructions machine permettant de réaliser ce programme.
2. Dans le cas où l'on dispose d'une machine RISC, rappelez le jeu type d'instruction dont on dispose et donnez la séquence d'instructions machine permettant de réaliser ce programme.

LA FONCTION DE MÉMORISATION

10.4 Cache à correspondance directe

Soit une mémoire de mots de 32 bits, adressée avec des adresses de 32 bits. Le cache à correspondance directe contient 4 Ko de données utiles. Une entrée du cache contient un bloc de 1 mot mémoire.

1. Calculez la taille réelle du cache.
2. On considère que lors de l'exécution d'un programme, le processeur accède aux mots mémoire suivants dont les adresses sont :

   $(00000000)_{16}$, $(00000008)_{16}$, $(00000001)_{16}$, $(00001000)_{16}$, $(FFFF0008)_{16}$, $(00000000)_{16}$

   Le cache est initialement vide. Représentez l'évolution du cache en notant les défauts et les succès.

10.5 Calcul de la taille réelle d'un cache

Soit une mémoire centrale de 1 Go, composée de mots de 64 bits. On désire réaliser une mémoire cache pour améliorer les performances du processeur disposant de cette mémoire centrale en lui ajoutant un dispositif de mémoire cache travaillant par blocs de 8 mots de 64 bits et de capacité utile égale à 64 Ko. Quelle est la taille réelle de ce cache à correspondance directe ?

10.6 Cache associatif et remplacement de lignes

Soit un processeur qui dispose d'une mémoire cache associative de quatre entrées. Les adresses mémoire sont sur 16 bits. Chaque mot mémoire fait 32 bits. La mémoire centrale est adressable par octet. Chaque entrée du cache contient un bloc de quatre mots.

1. Quelle est la taille de l'étiquette ?
2. Soit la suite de références suivantes, qui correspondent aux accès mémoire demandés par le processeur, en termes d'adresses d'octets, dans le temps. Les adresses sont données en hexadécimal (base 16).

Temps	0	1	2	3	4	5	6	7	8
Adresse	001F	0A1F	013A	001D	1B1E	0014	013B	1B32	1137

– Donnez l'évolution des quatre entrées du répertoire du cache et notez les défauts dans les deux cas suivants :
  1. la politique de remplacement est FIFO ;
  2. la politique de remplacement est LRU.

10.7 Cache à correspondance directe

Un processeur dispose d'une mémoire cache à correspondance directe de quatre entrées. Les adresses mémoire sont sur 16 bits. Chaque mot mémoire fait 32 bits. La mémoire centrale est adressable par octet. Chaque entrée du cache contient un bloc de quatre mots.

1. Quelle est la capacité de la mémoire centrale exprimées en Ko ? En Kmots ?
2. Quelle est la taille de l'étiquette ?
3. Quelle est la taille réelle du cache ?

LA FONCTION DE COMMUNICATION

10.8 Questions de cours

Cet exercice est fait pour que vous trouviez dans le cours les éléments de réponse.

1. Quel est le rôle des pilotes ?
2. Qu'appelle-t-on basculement de contexte ?
3. Comment est prise en compte une interruption, du point de vue logiciel et du point de vue matériel ? Est-ce que le mécanisme d'interruptions utilise le basculement de contexte ?
4. Quelles sont les différences entre les modes d'échanges par liaison programmée, pilotée par les interruptions, pilotées par les interruptions avec utilisation d'un DMA.

10.9 Entrées-sorties programmées et entrées-sorties par interruption

Une unité périphérique de type imprimante est considérée. ETAIMP est le registre d'état de l'imprimante tel que le bit de poids fort de ETAIMP est à 1 si l'imprimante est prête et à 0 sinon. SORIMP est le registre de données de l'imprimante. Cette machine représente les nombres signés selon le format du complément à 2.

1. On souhaite écrire un programme réalisant une opération d'entrées-sorties programmée qui permet le transfert de 80 caractères depuis la mémoire à partir de l'adresse EMPL vers cette imprimante. Le registre RB contient l'adresse de la donnée dans la mémoire principale (initialement EMPL). Le registre R1 contient le nombre de caractères restants à transférer. On vous donne en plus des instructions déjà introduites, l'instruction suivante qui permet de tester le signe du contenu d'un registre R : TESTS Rg1 R. TESTS positionne le bit S du registre d'état PSW, tel que celui-ci est à 1 si le contenu de R est négatif et 0 sinon.

2. On suppose à présent que la fin de transfert de chaque caractère par l'unité d'échange vers l'imprimante est signalée par une interruption. Écrivez le code de la routine d'interruption associée.

10.10 Performances des opérations d'entrées-sorties

1. On considère trois périphériques : un souris, une unité de disquette et une unité de disque dur. Les trois périphériques sont gérés par entrées-sorties programmées. Une opération d'interrogation d'un périphérique coûte 100 cycles horloge. L'horloge processeur fonctionne à une fréquence de 50 MHz.
   – La souris doit être interrogée 30 fois par seconde pour être sûr de ne manquer aucun mouvement de l'utilisateur.
   – La disquette transfère des données au processeur par blocs de deux octets et possède un débit de 50 Ko/seconde.
   – Le disque transfère des données au processeur par blocs de quatre octets et possède un débit de 2 Mo/seconde.

   Calculez la fraction de temps processeur consommée pour la gestion de chacun des périphériques.
2. On considère à présent que l'unité de disque est gérée par DMA. L'initialisation du DMA par le processeur nécessite 1 000 cycles horloge. Le traitement de l'interruption en fin de transfert DMA nécessite 500 cycles horloge. Chaque transfert DMA concerne 4 Ko de données et le disque est actif à 100 %.

   Quelle est la fraction de temps processeur consommée par la gestion de l'unité DMA ?

10.11 Gestion des interruptions

On considère une machine admettant huit niveaux d'interruptions matérielles numérotés de 0 à 7, le niveau d'interruptions 0 étant le plus prioritaire et le niveau 7 le moins prioritaire. Le processeur dispose de deux broches, une broche INT sur laquelle lui parvient le signal d'interruption, une broche INTA avec laquelle il acquitte les interruptions. Les huit niveaux d'interruptions sont gérés par un contrôleur d'interruptions.

1. À l'instant 0, le contrôleur d'interruption reçoit les interruptions 2, 5, 4, 6. Quelle interruption est délivrée au processeur ? Que fait le processeur ?
2. Durant le traitement par le processeur de l'interruption délivrée selon vous à la question 1, le contrôleur reçoit l'interruption 1. Que se passe-t-il ? Aucune autre interruption n'est délivrée au contrôleur. Donnez l'ordre de service de ces interruptions par le processeur.
3. Lorsque le processeur prend en compte l'interruption qui lui est délivrée à la question 1, le compteur ordinal CO contient la valeur 400, qui est l'adresse en mémoire centrale de la prochaine instruction à exécuter pour le programme en cours. Lorsque le processeur prend en compte l'interruption 1 de la question 2, le compteur ordinal CO contient la valeur 145, qui est l'adresse en mémoire centrale de la prochaine instruction à exécuter pour le programme en cours.

La table des vecteurs d'interruptions du processeur est la suivante :

Numéro IRQ	Adresse de la routine à exécuter en mémoire centrale
0	100
1	120
2	140
3	160
4	180
5	200
6	220
7	240

Les adresses en mémoire centrale sont les adresses des mots mémoire ; l'incrémentation du CO, l'incrémentation ou la décrémentation du RSP s'effectue par pas de 1.

En reprenant l'ordre de service des interruptions par le processeur tel que vous le donnez en réponse à la question 2, donnez l'évolution du registre CO, du registre RSP et de la pile.

Vous adopterez la convention suivante :

(RSP –> 1002), PILE : (1000 –> 200, 1001 –> 400), (CO->300), qui signifie :

RSP →	1002	
	1001	400
	1000	200
	adresse	pile

CO	300

SYNTHÈSE

10.12 Exercice de synthèse

On considère l'architecture de processeur suivante (figure 10.2). Les registres du processeur sont sur 8 bits. Les nombres signés sont représentés selon la convention du complément à 2.

Les portes à activer sont représentées par un rond sur la flèche.

Le format des microcommandes est :

– entrée du contenu d'un bus vers un registre : nombusnomregistreEn (exemple AtamponAEn) ;

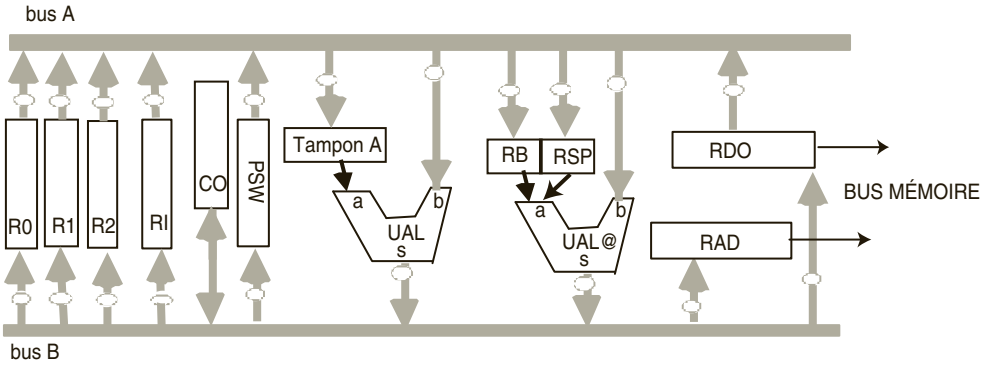


Figure 10.2 Architecture à 2 bus.

– sortie du contenu d'un registre vers un bus : nombusnomregistreSor (exemple BCOSor) ;
– entrée sur une entrée de l'UAL@ : nombusentréeUAL@En ou nomregistreentréeUAL@En (exemples RBUAL@aEn, AUAL@bEn) ;
– entrée sur une entrée de l'UAL : nombusentréeUALEn (exemple AUALbEn) ;
– sortie des UAL : nombussortieUALSor et nombussortieUAL@Sor (exemple BUALsSor).

Cette architecture comporte deux Unités Arithmétiques et Logiques : UAL et UAL@. UAL est utilisée pour réaliser les calculs et opérations logiques du processeur. UAL@ est uniquement utilisée pour calculer des adresses notamment dans le cadre du mode d'adressage basé (addition du contenu du registre de base RB et du déplacement X).

Le passage du bus A vers le bus B s'effectue en activant une opération NOP sur l'Unité Arithmétique et Logique (UAL), ce qui a pour effet de copier le contenu du tampon A sur la sortie de l'UAL. Ainsi, pour passer le contenu du bus A au bus B, les étapes sont :

– mettre le contenu du bus A dans tampon A : AtamponAEn ;
– activer l'opération Nop : NOP ;
– sortir la sortie de l'UAL sur le bus B : BUALsSor.

On considère par ailleurs que la mémoire centrale est chargée avec le programme suivant dont les instructions sont données en langage d'assemblage, selon le format établi dans le chapitre 6, partie Langage machine.

1. Complétez la colonne commentaire pour expliquer ce que réalise chaque instruction, puis concluez en expliquant ce que fait ce programme.
2. L'instruction `ADD Rg2 R2 R1` est chargée dans le registre RI. Donnez la suite de microcommandes correspondant à son exécution sur l'architecture de processeur de l'exercice.
3. Le registre d'état PSW contient un ensemble d'indicateurs S, C, O, Z. Rappelez leur rôle.

Adresse mémoire	Mot mémoire	Commentaire éventuel
`A :`		Valeur de la case A non initialisée
	`IN D A`	L'instruction IN permet la lecture au clavier d'une valeur pour A
	`LOAD Im R1 – 1`	
	`LOAD D R2 A`	
	`JMP Addition`	
`Fin`	`STORE D R2 A`	
	`STOP`	Fin de l'exécution
`Addition`	`ADD Rg2 R2 R1`	
	`JMP Fin`	

4. L'instruction `IN D A` lit la valeur 1 au clavier. Expliquez quelle est la valeur contenue dans le registre PSW à la suite de l'exécution du programme.
5. L'instruction `IN D A` lit la valeur – 128 au clavier. Expliquez quelle est la valeur contenue dans le registre PSW à la suite de l'exécution du programme.
6. Lorsque l'opération d'addition produit un overflow, on souhaite écrire le résultat contenu dans R2 au sommet de la pile plutôt qu'à l'adresse A. Modifiez le code du programme pour permettre cette opération.
7. L'instruction `JMP Addition` est la prochaine instruction à exécuter. Donnez la suite de microcommandes permettant son chargement dans RI et correspondant ensuite à son exécution sur l'architecture de processeur de l'exercice.
8. L'instruction `IN D A` permet la lecture d'une valeur au clavier, cette valeur étant stockée à l'emplacement A. Que déduisez-vous de la gestion des entrées-sorties avec le clavier, de l'utilisation de cette instruction IN d'entrées-sorties ?
9. Donnez la suite de microcommandes correspondant à l'exécution de l'instruction `LOAD B R1 X` sur l'architecture de processeur de l'exercice.

SOLUTIONS

10.1 Révision

1. Relisez le paragraphe 7.3.1.
2. Relisez le paragraphe 7.3.2, section les micro-instructions.
3. Relisez le paragraphe 7.3.2.

10.2 Microcommandes

1. Étapes de l'exécution d'une instruction machine :

Load Im, R1, x	COsor, RADen, Lec, InCo RDOsor, RIen RIsor, R1en (x considéré comme une valeur est placée dans R1)
Load D, R1, x	Cosor, RADen, Lec, InCo RDOsor, RIen RIsor, RADen, Lec (x considéré comme une adresse est placé dans RAD) RDOsor, R1en
Load I, R1, x	Cosor, RADen, Lec, InCo RDOsor, RIen RIsor, RADen, Lec (x considéré comme une adresse est placé dans RAD) RDOsor, RADen, Lec (le contenu de x considéré comme une adresse est placé dans RAD) RDOsor, R1en

L'étape de Fetch a toujours le même nombre de microcommandes. À chaque fois que le mode d'adressage évolue (Im, Direct, Indirect) on ajoute une microcommande qui fait une référence mémoire de plus.

Add D, R1, x	Cosor, RADen, Lec, InCo RDOsor, RIen RIsor, RADen, Lec (x considéré comme une adresse est placé dans RAD) RDOsor, Yen (Y est le registre d'entrée de l'UAL) R1sor, add, Zen (Le contenu de R1 est placé sur le bus donc sur la deuxième entrée de l'UAL, add déclenche l'addition dont le résultat est placé dans Z) Zsor, R1en

2. Étape de chargement d'une instruction :

Load D, R1, x	COsor, NOP, RADen, Lec, InCo RDOsor, RIen RIsor, NOP, NOP, RADen, Lec RDOsor, NOP, NOP, R1en
Add D, R1, x	COsor, NOP, RADen, Lec, InCo RDOsor, RIen R1sor, TamponAen RIsor, NOP, NOP, RADen, Lec RDOsor, TamponBen, add, UALsor, R1en

10.3 CISC/RISC

Dans le cas de la machine CISC on obtient :

```
Load D, R1, A
Add D, R1, B
Store D, R1, C
```

Dans le cas d'une machine RISC on obtient :

```
Load D, R1, A
Load D, R2, B
Load Im, R3, 0
Add R3, R2, R1
Store D, R3, C
```

On voit donc qu'il y a plus d'instructions machine à exécuter dans le cas de la machine RISC. Cependant il faut également noter que les instructions des machines RISC sont toutes de même longueur et que donc on peut plus facilement avoir un pipeline efficace. Par ailleurs l'instruction Add ne fait référence qu'à des registres et pas du tout à la mémoire centrale ce qui est plus rapide. Enfin dans une machine RISC le séquenceur est câblé ce qui est un facteur d'accélération de l'exécution. En résumé… Il est bien délicat de comparer les performances.

10.4 Cache à correspondance directe

1. Taille réelle du cache :

Entrée du cache en bits = 1 (bit validation) + n bits (étiquette) + m bits (donnée)
= 1 + n + 32 bits

La taille de l'étiquette se déduit de celle de l'index et de celle de l'adresse :

taille étiquette (n) = taille de l'adresse en bits – nombre bits pour désigner un octet – nombre bits constituant l'index

La taille de l'index est fonction du nombre d'entrées du cache.

nombre entrées du cache = taille des données utiles en octets / taille d'une entrée en donnée utile en octets

$= (4 \times 1\,024)$ octets / 4 octets
$= 1\,024$ entrées
$= 2^{10}$ entrées

La taille de l'index est donc de 10 bits.

Une entrée contient un bloc de 4 octets : le nombre de bits pour désigner un octet dans une entrée est donc égal à 2. D'où :

– Taille étiquette (n) = 32 – 2 – 10 = 20 bits.
– Entrée du cache en bits = 1 + 20 + 32 = 53 bits.
– Taille du cache = 53 × 1 024 = 53 Kbits.

2. Évolution du cache :

– $(00000000)_{16}$: défaut.

Index	V	Étiquette	Donnée
0	1	$(00000)_{16}$	contenu du mot d'adresse $(00000000)_{16}$

– $(00000008)_{16}$: défaut.

Index	V	Étiquette	Donnée
0	1	$(00000)_{16}$	contenu du mot d'adresse $(00000000)_{16}$
1	0		
2	1	$(00000)_{16}$	contenu du mot d'adresse $(00000008)_{16}$

– $(00000001)_{16}$: succès. On accède à l'entrée 0 du cache et les étiquettes sont égales.
– $(00001000)_{16}$: défaut.

Index	V	Étiquette	Donnée
0	1	$(00001)_{16}$	contenu du mot d'adresse $(00001000)_{16}$
1	0		
2	1	$(00000)_{16}$	contenu du mot d'adresse $(00000008)_{16}$

– $(FFFF0008)_{16}$: défaut.

Index	V	Étiquette	Donnée
0	1	$(00001)_{16}$	contenu du mot d'adresse $(00001001)_{16}$
1	0		
2	1	$(FFFF0)_{16}$	contenu du mot d'adresse $(FFFF0008)_{16}$

– $(00000000)_{16}$: défaut.

Index	V	Étiquette	Donnée
0	1	$(00000)_{16}$	contenu du mot d'adresse $(00000000)_{16}$
1	0		
2	1	$(FFFF0)_{16}$	contenu du mot d'adresse $(FFFF0008)_{16}$

10.5 Calcul de la taille réelle d'un cache

Calcul de l'entrée du cache en bits :

Entrée du cache en bits = 1 (bit validation) + n bits (étiquette) + m bits (donnée)
= 1 + n + (8 × 64) bits

La taille de l'étiquette se déduit de celle de l'index et de celle de l'adresse :

taille étiquette (n) = taille de l'adresse en bits – nombre bits pour désigner un octet – nombre bits constituant l'index

La taille de l'index est fonction du nombre d'entrées du cache.

Nombre entrées du cache = taille des données utiles en octets / taille d'une entrée en donnée utile en octets
= 64 × 1 024 octets / 8 × 8 octets
= 1 024 entrées
= 2^{10} entrées

La taille de l'index est donc de 10 bits.

Une entrée contient un bloc de 8 mots de 8 octets : le nombre de bits pour désigner un octet dans une entrée est donc égal à 6.

La taille de l'adresse est fonction du nombre d'octets adressables en mémoire centrale : la mémoire centrale a une capacité de 1 Go = 2^{30} octets : l'adresse est sur 30 bits.

– Entrée du cache en bits = 1 + 14 + (64 × 8) = 527 bits.
– Taille du cache = 527 × 1 024 = 527 Kbits.

10.6 Cache associatif et remplacement de lignes

1. Pour un cache associatif, l'adresse est utilisée pour la recherche d'une ligne, en deux parties : étiquette – désignation d'un octet de la ligne.
   Chaque ligne de cache contient quatre mots de 32 bits, soit 16 octets. Il faut donc 4 bits pour désigner chacun des octets d'une ligne.
   En conséquence, l'étiquette a une taille de 16 – 4 = 12 bits.
2. Politique FIFO : la ligne remplacée est la plus anciennement chargée dans le cache.

temps	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
adresse	001F	0A1F	013A	001D	1B1E	0014	013B	1B32	1137
Ligne 1	**001**	001	001	**001**	001	**001**	001	**1B3**	1B3
Ligne 2		**0A1**	0A1	0A1	0A1	0A1	0A1	0A1	**113**
Ligne 3			**013**	013	013	013	**013**	013	013
Ligne 4					**1B1**	1B1	1B1	1B1	1B1
Défaut/ Succès	D	D	D	S	D	S	S	D	D

– Les trois premiers chiffres de l'adresse correspondent à la valeur d'étiquette. Il y a succès lorsque l'étiquette de l'adresse correspond à l'étiquette stockée dans une des lignes du cache.
– Au temps 7, il y a défaut, toutes les lignes du cache sont occupées. La ligne la plus anciennement chargée est la ligne 1 qui est remplacée.
– Au temps 8, il y a défaut, toutes les lignes du cache sont occupées. La ligne la plus anciennement chargée est la ligne 2 qui est remplacée.

Politique LRU : la ligne remplacée est la moins récemment accédée dans le cache.

temps	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
adresse	001F	0A1F	013A	001D	1B1E	0014	013B	1B32	1137
Ligne 1	**001**	001	001	**001**	001	**001**	001	001	001
Ligne 2		**0A1**	0A1	0A1	0A1	0A1	0A1	**1B3**	1B3
Ligne 3			**013**	013	013	013	**013**	013	013
Ligne 4					**1B1**	1B1	1B1	1B1	**113**
Défaut/ Succès	D	D	D	S	D	S	S	D	D

– Au temps 7, il y a défaut, toutes les lignes du cache sont occupées. La ligne la moins récemment accédée parmi celles présentes dans le cache est la ligne 2 qui est remplacée.
– Au temps 8, il y a défaut, toutes les lignes du cache sont occupées. La ligne la moins récemment accédée parmi celles présentes dans le cache est la ligne 4 qui est remplacée.

10.7 Cache à correspondance directe

1. La capacité de la mémoire centrale est égale à 2^{16} octets, soit 4 Koctets, soit 1 Kmots.
2. Une ligne de cache contient quatre mots de 32 bits, soit 16 octets. Le cache comporte quatre entrées. L'étiquette a donc une taille égale à : 16 – 2 – 4 = 10 bits.
3. La taille réelle du cache est égale à 4 × (1 + 10 + 128) = 556 bits.

10.8 Questions de cours

Les éléments de réponse sont dans le cours.

10.9 Entrées-sorties programmées et entrées-sorties par interruption

1. Programme de transfert :

   Cet exemple montre bien que dans le cas de la liaison programmée la vitesse globale de fonctionnement est déterminée par la vitesse du périphérique.

Initialisation des registres RB et R1	
RB ← EMPL R1 ← 80	LOAD Im RB EMPL LOAD Im R1 80
Boucle de transfert d'un caractère	
ATTENTE : attente imprimante prête on boucle tant que ETATIMP < 0 R2 ← ETATIMP test R2 < 0 ? Si oui, retour à ATTENTE transfert du caractère son adresse est donnée par RB SORIMP ← caractère décrémentation de R1, incrémentation de RB si R1 = 0, fin sinon retour à ATTENTE	ATTENTE : LOAD D R2 ETATIMP TESTS Rg1 R2 JMPN ATTENTE LOAD B R3 0 STORE D R3 SORIMP ADD Im RB 1 ADD Im R1 – 1 JMPP ATTENTE STOP

2. La routine d'interruption est :

masquage des Interruptions décrémenter R1 si R1 = 0 aller à fin SORIMP ← caractère pointer sur le caractère suivant fin : RTI	DI ADD Im R1 – 1 JMPZ FIN LOAD B R3 0 STORE D R3 SORIMP ADD Im RB 1 FIN: RTI

Dans ce dernier cas on considère qu'à chaque fois qu'un caractère est écrit il y a émission d'une interruption. Il faut donc initialiser le mécanisme d'interruption et on le fait en plaçant dans le tampon de sortie le premier caractère à imprimer. Pour placer ce premier caractère on utilise le mode d'échanges par liaison programmée : on teste si le périphérique est libre, dès qu'il est libre on place le premier caractère à imprimer et on lance l'impression de ce caractère. À la fin de l'impression une interruption est émise, la routine d'interruption s'exécute : à partir de ce moment opération d'entrées-sorties est pilotée par les interruptions.

10.10 Performances des opérations d'entrées-sorties

1. Fraction de temps processeur consommée :
   - Cas de la souris :

     Il faut $30 \times 100 = 3\,000$ cycles par seconde pour interroger la souris ce qui représente $3\,000/(50 \times 10^6) = 0{,}006$ % des cycles du processeur.
   - Cas de la disquette :

     La disquette transfère les données à un débit de 50 Ko/s. Comme les transferts s'effectuent par blocs de 2 octets, il faut 25 Ko interrogations par seconde.

Ces 25 Ko interrogations par seconde représentent $25 \times 2^{10} \times 100 = 25{,}6 \times 10^5$ cycles par secondes, ce qui constitue $25{,}6 \times 10^5/(50 \times 10^6) = 5$ % des cycles du processeur.

– Cas du disque :

Le disque transfère les données à un débit de 2 Mo/s. Comme les transferts s'effectuent par blocs de 4 octets, il faut 500 Ko interrogations par seconde.

Ces 500 Ko interrogations par seconde représentent $500 \times 2^{10} \times 100 = 51{,}2 \times 10^6$ cycles par secondes, ce qui constitue $51{,}2 \times 10^6/(50 \times 10^6) = 100$ % des cycles du processeur !

2. Chaque transfert par DMA prend 4 Ko/2 Mo = 2×10^{-3} secondes.

Comme le disque transfère continuellement ; il faut $1\,000 + 500/2 \times 10^{-3} = 750 \times 10^3$ cycles par seconde, ce qui représente $750 \times 10^3/(50 \times 10^6) = 1{,}5$ % des cycles du processeur.

10.11 Gestion des interruptions

1. L'interruption 2 est délivrée au processeur car c'est l'interruption la plus prioritaire. Le processeur stoppe le traitement en cours, acquitte l'interruption et, avec son numéro, indexe la table des vecteurs d'interruptions. Il récupère l'adresse de la routine d'interruption et charge le registre CO avec cette adresse. Auparavant, il a sauvegardé le registre CO du programme utilisateur interrompu dans la pile.
2. L'interruption 1 est plus prioritaire que l'interruption 2 en cours de traitement. Le contrôleur délivre cette interruption au processeur qui interrompt le traitement de l'interruption 2 et démarre le traitement de l'interruption 1. L'ordre des services des requêtes d'interruptions est : 2 1 2 4 5 6.
3. État initial : (RSP –> 1002), PILE : (1000 –> 200 ; 1001 –> 400), CO –> 400)

   Arrivée interruption 2 : (RSP –> 1003), PILE : (1000 –> 200 ; 1001 –> 400 ; 1002 –> 400), CO –> 140)

   Arrivée interruption 1 : (RSP –> 1004), PILE : (1000 –> 200 ; 1001 –> 400 ; 1002 –> 400 ; 1003 –> 145), CO –> 120)

   Fin interruption 1, reprise interruption 2 : (RSP –> 1003), PILE : (1000 –> 200 ; 1001 –> 400 ; 1002 –> 400), CO –> 145)

   Fin interruption 2, traitement interruption 4 : (RSP –> 1003), PILE : (1000 –> 200 ; 1001 –> 400 ; 1002 –> 400), CO –> 180)

   Fin interruption 4, traitement interruption 5 : (RSP –> 1003), PILE : (1000 –> 200 ; 1001 –> 400 ; 1002 –> 400), CO –> 200)

   Fin interruption 5, traitement interruption 6 : (RSP –> 1003), PILE : (1000 –> 200 ; 1001 –> 400 ; 1002 –> 400), CO –> 220)

   Fin interruption 6, reprise du programme utilisateur : (RSP –> 1002), PILE : (1000 –> 200 ; 1001 –> 400), CO –> 400)

10.12 Exercice de synthèse

1.

Adresse mémoire	Mot mémoire	Commentaire éventuel
`A :`		Valeur de la case A non initialisée
	`IN D A`	L'instruction IN permet la lecture au clavier d'une valeur pour A
	`LOAD Im R1 – 1`	R1 est chargé avec la valeur – 1
	`LOAD D R2 A`	R2 est chargé avec le contenu du mot d'adresse A
	`JMP Addition`	Saut à l'instruction nommée Addition
`Fin`	`STORE D R2 A`	R2 est écrit à l'adresse A en mémoire centrale
	`STOP`	Fin de l'exécution
`Addition`	`ADD Rg2 R2 R1`	R2 = R2 + R1
	`JMP Fin`	Saut à l'instruction nommée Fin

Ce programme décrémente d'une unité la valeur placée dans le mot d'adresse A.

2. `AR2Sor, ATamponAEn, AR1Sor, AUALbEn, ADD, BUALsSor, BR2En`
3. Les indicateurs S, C, O, Z permettent de positionner les propriétés du dernier calcul réalisé par l'UAL. Ainsi S permet d'indiquer si le résultat produit par l'UAL est positif ou négatif, O permet d'indiquer l'occurrence d'un dépassement de capacité, C d'un carry. Enfin, Z permet d'indiquer si le résultat produit par l'UAL est nul ou non.
4. Lors de l'exécution du programme, l'instruction `ADD Rg2 R2 R1` effectue l'opération 1 – 1 = 0. Les indicateurs du registre PSW sont alors positionnés comme suit :
   - Z indique un résultat nul ;
   - S indique un résultat positif ;
   - C indique qu'il n'y a pas de carry ;
   - O indique qu'il n'y a pas de dépassement de capacité.
5. Lors de l'exécution du programme, l'instruction `ADD Rg2 R2 R1` effectue l'opération – 128 – 1. Posons cette opération en binaire.

   – 128 1000 0000
   – 1 1111 1111
   ———————
   1 0111 1111

   Les indicateurs du registre PSW sont alors positionnés comme suit :
   - Z indique un résultat non nul ;
   - S indique un résultat positif ;

– C indique qu'il y a carry ;
– O indique qu'il y a dépassement de capacité.

6.

Adresse mémoire	Mot mémoire
A :	
	IN D A
	LOAD Im R1 – 1
	LOAD D R2 A
	JMP Addition
Fin	STORE D R2 A
	STOP
Addition	ADD Rg2 R2 R1
	JMPO Overflow
	JMP Fin
Overflow	**PUSH Rg1 R2**
	STOP

7. FETCH : BCOSor, BRADEn, LEC, ARDOSor, ATamponAEn, NOP, BUALsSor, BRIEn
   EXECUTION : ARISor, ATamponAEn, NOP, BUALsSor, BCOEn
8. Le jeu d'instructions utilise des instructions spécifiques d'entrées-sorties. L'espace d'adresses des entrées-sorties est donc séparé de celui de la mémoire.
9. FETCH : BCOSor, BRADEn, LEC, ARDOSor, ATamponAEn, NOP, BUALsSor, BRIEn
   EXECUTION : ARISor, AUal@bEn, RBUal@En, ADD, BUAL@sSor, BRADEn, LEC, ARDOSor, ATamponAEn, NOP, BUALsSor, BR1En

PARTIE 3

LES SYSTÈMES D'EXPLOITATION

L'ensemble des chapitres de cette partie est centré autour de la notion de *système d'exploitation multiprogrammé*. Après avoir présenté le rôle fondamental du système d'exploitation vis-à-vis de l'utilisateur d'un ordinateur et vis-à-vis de la machine physique, ainsi que les notions de base attachées au système d'exploitation, nous nous intéressons à chacune des principales fonctions que comporte un système d'exploitation.

Le chapitre 11 introduit la notion de système d'exploitation et décrit l'architecture générale d'un système. Le chapitre 12 présente les notions de processus et d'ordonnancement ainsi que les problèmes de communication et de synchronisation entre processus. Le chapitre 13 est quant à lui consacré à la gestion de la mémoire centrale et aborde les notions liées plus particulièrement à la pagination et à la mémoire virtuelle. Le chapitre 14 présente ce qui a trait à la gestion des supports de masse tels que le disque dur et s'intéresse au service de gestion de fichiers. Le chapitre 15 constitue une introduction aux réseaux ; il aborde les notions fondamentales relatives aux connexions filaires et radio. Enfin cette partie s'achève sur un ensemble d'exercices corrigés.

Mots-clés : système d'exploitation, processus, gestion de la mémoire, système de gestion de fichiers, synchronisation réseaux filaires, Ethernet, Internet, Wi-Fi.

Chapitre **11**

Introduction aux systèmes d'exploitation multiprogrammés

Le *système d'exploitation* est un ensemble de programmes qui réalise l'interface entre le matériel de l'ordinateur et les utilisateurs, d'une part afin de construire au-dessus du matériel une machine virtuelle plus facile d'emploi et plus conviviale, d'autre part afin de prendre en charge la gestion des ressources de la machine et le partage de celles-ci. Dans ce chapitre, nous allons définir plus précisément les rôles d'un système d'exploitation dans un environnement multiprogrammé et les différentes fonctions qui composent ce système d'exploitation. Nous présentons également les différents types de systèmes d'exploitation multiprogrammés existants à l'heure actuelle. Enfin, les notions de base sur lesquelles repose le fonctionnement du système d'exploitation sont décrites.

11.1 RÔLE ET DÉFINITION D'UN SYSTÈME D'EXPLOITATION MULTIPROGRAMMÉ

Le chapitre 9 a présenté les différents modes de réalisation des entrées-sorties au niveau du matériel. Du mode programmé au mode par DMA avec interruption, la caractéristique principale de cette évolution est la libération du processeur de la prise en charge des entrées-sorties. En effet, lorsqu'une opération d'entrées-sorties est réalisée par DMA, le processeur est totalement libéré de la gestion du transfert de données, une fois l'initialisation du DMA réalisée, alors qu'avec un mode d'entrées-sorties programmées, le processeur est sollicité pour le transfert de chaque octet de donnée.

Dans le cadre d'une machine monoprogrammée, c'est-à-dire une machine pour laquelle un seul programme utilisateur est présent en mémoire centrale, cela veut dire que le processeur reste inactif à chaque fois que ce programme utilisateur réalise une opération d'entrées-sorties. Une telle inaction, illustrée sur le chronogramme de la figure 11.1, n'est pas souhaitable. Sur cette figure, nous avons considéré un programme utilisateur Prog A avec un profil d'exécution qui est tel que ce programme réalise 50 % de calcul, puis 50 % d'entrées-sorties. Le processeur est donc inactif la

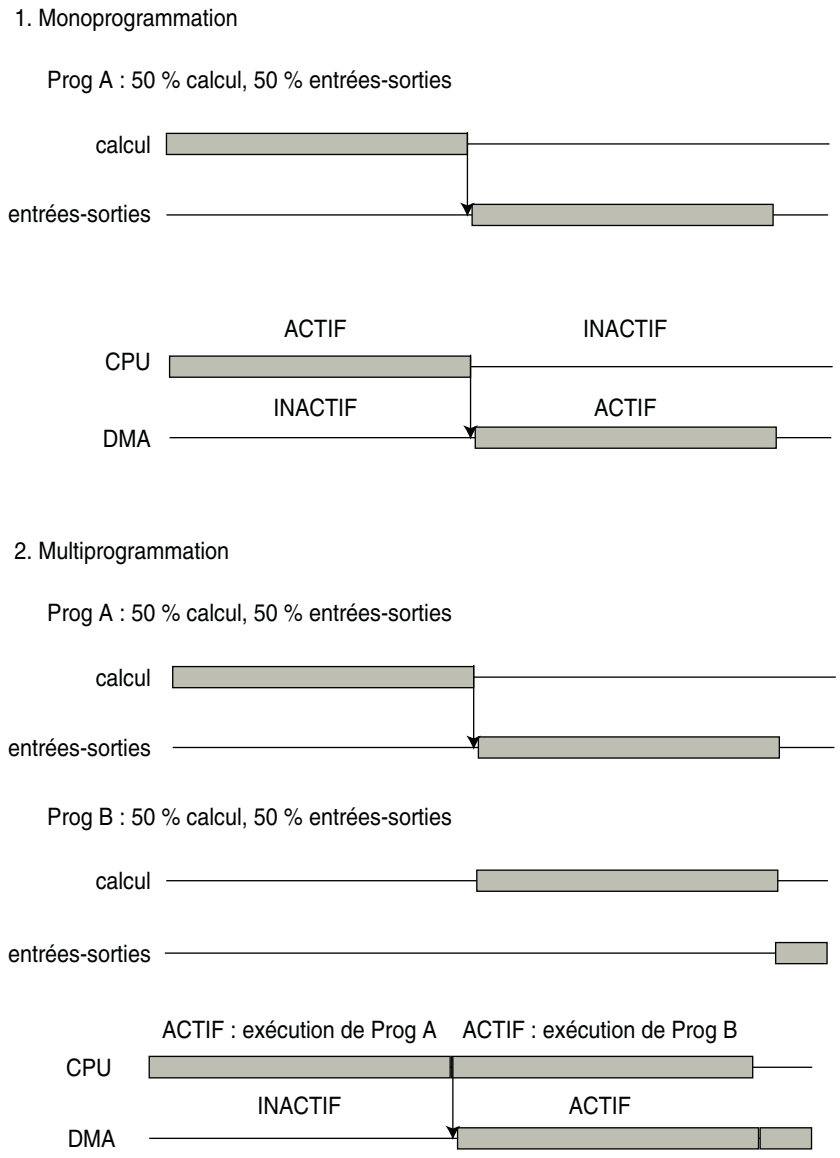


Figure 11.1 Chronogramme d'activité pour le processeur et le DMA.

moitié du temps. Pour remédier à cette inaction, la machine doit devenir multiprogrammée, c'est-à-dire qu'au moins un autre programme utilisateur doit être placé en mémoire centrale, qui sera exécuté pendant l'opération d'entrées-sorties du premier. Ainsi, dans notre exemple, nous plaçons en mémoire centrale, un second programme Prog B, ayant le même profil que le programme Prog A. Prog B est exécuté par le processeur durant l'opération d'entrées-sorties de Prog A ; le processeur est maintenant totalement occupé.

L'exemple de la figure 11.1. est un exemple théorique et idéalisé. Dans la réalité, pour parvenir à occuper à près de 100 % le processeur, il faut placer en mémoire centrale un très grand nombre de programmes.

11.1.1 Un premier rôle : assurer le partage de la machine physique

La machine physique et ses différents composants, s'ils offrent des mécanismes permettant de faciliter leur partage entre différents programmes, ne sont malgré tout pas conçus pour supporter et gérer d'eux-mêmes ce partage. C'est là le premier rôle du système d'exploitation dans un environnement multiprogrammé que de gérer le partage de la machine physique et des ressources matérielles entre les différents programmes. Cette gestion doit assurer l'équité d'accès aux ressources matérielles et assurer également que les accès des programmes à ces ressources s'effectuent correctement, c'est-à-dire que les opérations réalisées par les programmes sont licites pour la cohérence des ressources : on parle alors de *protection des ressources*.

Le partage des ressources va concerner principalement le processeur, la mémoire centrale et les périphériques d'entrées-sorties. Plus précisément, les questions suivantes vont devoir être résolues :

– dans le cadre du partage du processeur : parmi tous les programmes chargés en mémoire centrale, lequel doit s'exécuter ?
– dans le cadre du partage de la mémoire centrale : comment allouer la mémoire centrale aux différents programmes ? Comment disposer d'une quantité suffisante de mémoire pour y placer tous les programmes nécessaires à un bon taux d'utilisation du processeur ? Comment assurer la protection entre ces différents programmes utilisateurs ? Par protection, on entend ici veiller à ce qu'un programme donné n'accède pas à une plage mémoire allouée à un autre programme ;
– dans le cadre du partage des périphériques : dans quel ordre traiter les requêtes d'entrées-sorties pour optimiser les transferts ?

11.1.2 Un second rôle : rendre conviviale la machine physique

Tout au long de la seconde partie de cet ouvrage, ont été présentées les caractéristiques des ressources matérielles processeur, mémoire, périphériques – composant la machine physique. Chaque ressource a ses propriétés et son mode de gestion déterminé par le constructeur de la machine. Ainsi par exemple, tel périphérique est géré par interruption alors qu'un autre est géré par DMA. Se servir de la machine physique et utiliser à travers un programme ses ressources nécessitent de connaître

les particularités de gestion de chacune des ressources physiques utilisées. Cela est évidemment très fastidieux et compliqué pour l'utilisateur de la machine.

Faciliter l'accès à la machine physique constitue le second rôle du système d'exploitation. Par le biais d'une interface de haut niveau, composée d'un ensemble de primitives attachées à des fonctionnalités qui gèrent elles-mêmes les caractéristiques matérielles sous-jacentes et offrent un service à l'utilisateur, le système d'exploitation construit au-dessus de la machine physique, une machine virtuelle plus simple d'emploi et plus conviviale. Ainsi, pour réaliser une opération d'entrées-sorties, l'utilisateur fera appel à une même primitive `ECRIRE(données)` quel que soit le périphérique concerné. C'est la primitive `ECRIRE` et la fonction de gestion des entrées-sorties du système d'exploitation à laquelle cette primitive est rattachée qui feront la liaison avec les caractéristiques matérielles. On appelle *driver* une telle fonction de gestion d'entrées-sorties rattachée à un périphérique spécifique.

Programmation sur l'ENIAC[a] (1946)

Les premiers ordinateurs tels que l'ENIAC ne comportaient pas de systèmes d'exploitation. La programmation se faisait directement en langage machine et un seul programme à la fois pouvait être exécuté par la machine. L'absence de système d'exploitation obligeait le programmeur à charger manuellement le programme, instruction par instruction, dans les registres du processeur et à gérer lui-même les opérations d'entrées-sorties, ce qui l'obligeait à connaître les moindres détails du dialogue avec chaque type de périphérique. Une évolution décisive verra le jour avec l'apparition des *moniteurs d'enchaînement*, capable d'enchaîner automatiquement un ensemble de travaux soumis par un opérateur : c'est l'ancêtre du système d'exploitation.

a. L'ENIAC (*Electronic Numerical Integrator And Computer*) a été construit de 1943 à 1946 par John Mauchley et J. Presper Eckert à l'université de Pennsylvanie. Cette machine qui comportait 18 000 tubes à vide et 1 500 relais, pesait 30 tonnes, consommait 140 kW et occupait une surface au sol de 160 m^2. Elle comportait 20 registres de 10 chiffres décimaux et était programmée à l'aide de 6 000 commutateurs.

11.1.3 Définition du système d'exploitation multiprogrammé

Le système d'exploitation est donc un ensemble de programmes qui réalise l'interface entre le matériel de l'ordinateur et les utilisateurs. Il a deux objectifs principaux (figure 11.2) :

– construction au-dessus du matériel d'une machine virtuelle plus facile d'emploi et plus conviviale ;
– prise en charge de la gestion de plus en plus complexe des ressources et partage de celles-ci.

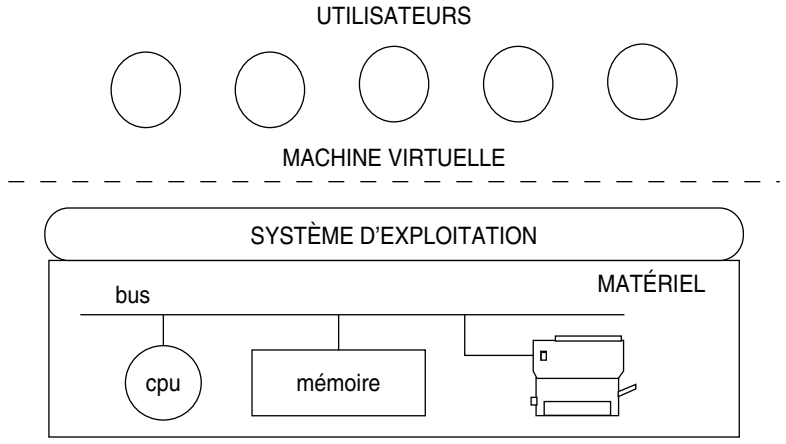


Figure 11.2 Place du système d'exploitation.

Comme son nom le suggère, le système d'exploitation a en charge l'exploitation de la machine pour en faciliter l'accès, le partage et pour l'optimiser.

11.2 STRUCTURE D'UN SYSTÈME D'EXPLOITATION MULTIPROGRAMMÉ

11.2.1 Composants d'un système d'exploitation

Les différentes fonctionnalités

Le système d'exploitation réalise donc une couche logicielle placée entre la machine matérielle et les applications. Le système d'exploitation peut être découpé en plusieurs grandes fonctionnalités présentées sur la figure 11.3.

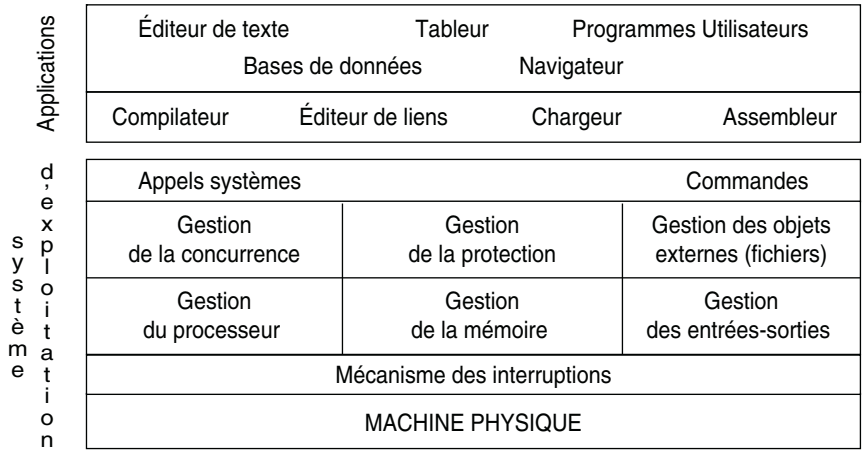


Figure 11.3 Fonctionnalités du système d'exploitation.

➤ La fonctionnalité de gestion du processeur

Le système doit gérer l'allocation du processeur aux différents programmes pouvant s'exécuter. Cette allocation se fait par le biais d'un *algorithme d'ordonnancement* qui planifie l'exécution des programmes. Selon le type de système d'exploitation, l'algorithme d'ordonnancement répond à des objectifs différents.

➤ La fonctionnalité de gestion de la mémoire

Le système doit gérer l'allocation de la mémoire centrale entre les différents programmes pouvant s'exécuter, c'est-à-dire qu'il doit trouver une place libre suffisante en mémoire centrale pour que le chargeur puisse y placer un programme à exécuter, en s'appuyant sur les mécanismes matériels sous jacents abordés au chapitre 8. Comme la mémoire physique est souvent trop petite pour contenir la totalité des programmes, la gestion de la mémoire se fait selon le principe de la *mémoire virtuelle* : à un instant donné, seules sont chargées en mémoire centrale, les parties de code et données utiles à l'exécution.

➤ La fonctionnalité de gestion des entrées-sorties

Le système doit gérer l'accès aux périphériques, c'est-à-dire faire la liaison entre les appels de haut niveau des programmes utilisateurs (exemple `getchar()`) et les opérations de bas niveau de l'unité d'échange responsable du périphérique (unité d'échange clavier) : c'est le pilote d'entrées-sorties (*driver*) qui assure cette correspondance.

➤ La fonctionnalité de gestion des objets externes

La mémoire centrale est une mémoire volatile. Aussi, toutes les données devant être conservées au-delà de l'arrêt de la machine, doivent être stockées sur une mémoire de masse non volatile (disque dur, disquette, cédérom…). La gestion de l'allocation des mémoires de masse ainsi que l'accès aux données stockées s'appuient sur la notion de *fichiers* et de *système de gestion de fichiers* (SGF).

➤ La fonctionnalité de gestion de la concurrence

Comme plusieurs programmes coexistent en mémoire centrale, ceux-ci peuvent vouloir communiquer pour échanger des données. Par ailleurs, il faut synchroniser l'accès aux données partagées afin de maintenir leur cohérence. Le système offre des outils de communication et de synchronisation entre programmes.

➤ La fonctionnalité de gestion de la protection

Le système doit fournir des mécanismes garantissant que ses ressources (processeur, mémoire, fichiers) ne peuvent être utilisées que par les programmes auxquels les droits nécessaires ont été accordés. Il faut notamment protéger le système et la machine des programmes utilisateurs (mode d'exécution utilisateur et superviseur).

Les routines

Les fonctionnalités du système d'exploitation utilisent les mécanismes offerts par le matériel de la machine physique pour réaliser leurs opérations. Notamment, le système

d'exploitation s'interface avec la couche matérielle, par le biais du mécanisme des interruptions, qui lui permet de prendre connaissance des événements survenant sur la machine matérielle.

Par ailleurs, le système d'exploitation s'interface avec les applications du niveau utilisateur par le biais des *fonctions prédéfinies* que chacune de ses fonctionnalités offre. Ces fonctions que l'on qualifie *de routines systèmes* constituent les points d'entrées des fonctionnalités du système d'exploitation et sont appelables depuis les applications de niveau utilisateur. Ces appels peuvent se faire à deux niveaux :

– dans le code d'un programme utilisateur à l'aide d'un *appel système*, qui n'est autre qu'une forme d'appel de procédure amenant à l'exécution d'une routine système ;
– depuis le prompt de *l'interpréteur de commandes* à l'aide d'une *commande*. L'interpréteur de commandes est un outil de niveau utilisateur qui accepte les commandes de l'utilisateur, les analyse et lance l'exécution de la routine système associée.

➤ Exemples de routines système

Sous un système d'exploitation tel que MVS, l'appel à la routine du système (*SVC Call*) `GETMAIN` permet de demander l'allocation d'un espace mémoire et appartient donc à la fonctionnalité de gestion de la mémoire centrale. L'appel `EXCP` (*Execute Channel Program*) permet de demander l'exécution d'une opération d'entrées-sorties et appartient donc à la fonctionnalité de gestion des entrées-sorties.

Sous un système d'exploitation tel que Linux, l'appel à la fonction `CREAT (nom_fich, mode)` de la fonctionnalité de gestion des fichiers permet de créer un nouveau fichier `nom_fich` selon un mode d'accès `mode` qui peut être l'écriture, la lecture ou les deux combinés.

11.2.2 La norme POSIX pour les systèmes ouverts

La norme POSIX (*Portable Operating System Interface*) définit l'ensemble des services et primitives que doit offrir un système d'exploitation dit ouvert pour permettre l'écriture d'applications portables entre systèmes différents.

Un *système ouvert* est un système capable de dialoguer avec n'importe quel autre type de système. Un prérequis à ce dialogue est que les systèmes présentent une interface commune, c'est-à-dire un ensemble d'appels systèmes et de commandes identiques.

La norme POSIX est constituée de plusieurs extensions. Parmi celles-ci, l'extension 1003.1 définit l'ensemble minimal des services que doit offrir un système pour être qualifié de système ouvert.

11.3 PRINCIPAUX TYPES DE SYSTÈMES D'EXPLOITATIONS MULTIPROGRAMMÉS

Les systèmes d'exploitation multiprogrammés peuvent être classés selon différents types qui dépendent des buts et des services offerts par les systèmes. Trois grandes

classes de systèmes peuvent être définies : les systèmes à traitements par lots, les systèmes multi-utilisateurs interactifs et les systèmes temps réels.

11.3.1 Les systèmes à traitements par lots

Les systèmes à traitement par lots ou systèmes batch constituent en quelque sorte les ancêtres de tous les systèmes d'exploitation. Ils sont nés de l'introduction sur les toutes premières machines de deux programmes permettant une exploitation plus rapide et plus rentable du processeur, en vue d'automatiser les tâches de préparation des travaux à exécuter. Ces deux programmes sont d'une part le chargeur dont nous avons déjà parlé, et dont le rôle a été initialement de charger automatiquement les programmes dans la mémoire centrale de la machine depuis les cartes perforées ou le dérouleur de bandes et d'autre part, le moniteur d'enchaînement de traitements, dont le rôle a été de permettre l'enchaînement automatique des travaux soumis en lieu et place de l'opérateur de la machine.

Le principe du traitement par lots s'appuie sur la composition de lots de travaux ayant des caractéristiques ou des besoins communs, la formation de ces lots visant à réduire les temps d'attente du processeur en faisant exécuter les uns à la suite des autres ou ensemble, des travaux nécessitant les mêmes ressources. Ce mode de fonctionnement est directement tiré de la manière dont les programmeurs travaillaient au début de l'informatique : par exemple, pour compiler un programme FORTRAN, il fallait d'abord demander au chargeur de lire depuis le dérouleur de bandes, le compilateur FORTRAN, puis de lire le programme à compiler. Si trois travaux devaient être réalisés, deux compilations FORTRAN et une compilation COBOL, mieux valaient placer dans un même lot les deux compilations FORTRAN nécessitant comme ressource préalable le chargement du compilateur FORTRAN, et dans un autre lot la compilation COBOL nécessitant comme ressource particulière le compilateur COBOL, que de mélanger les trois types de travaux dans un ordre aléatoire, entraînant par exemple le chargement du compilateur FORTRAN pour le premier travail, puis le chargement du compilateur COBOL pour le second travail et de nouveau le chargement du compilateur FORTRAN…

Par ailleurs, dans un système à traitements par lots, la caractéristique principale est qu'il n'y a pas d'interaction possible entre l'utilisateur et la machine durant l'exécution du programme soumis. Le programme est soumis avec ses données d'entrées et l'utilisateur récupère les résultats de l'exécution ultérieurement, soit dans un fichier, soit sous forme d'une impression. Là encore, ce mode de fonctionnement est directement influencé par le mode de travail des premiers temps de l'informatique : le programmeur soumettait son programme sous forme de cartes perforées avec ses données à l'opérateur de la machine, qui faisait exécuter le programme et rendait les résultats de l'exécution ensuite au programmeur.

L'objectif poursuivi dans les systèmes à traitements par lots est de maximiser l'utilisation du processeur et le débit des travaux, c'est-à-dire le nombre de travaux traités sur une tranche de temps.

Un exemple de système d'exploitation orienté batch est le système MVS. Dans ce système, le traitement par lots (*job scheduling*) est assuré par une entité particulière du système, le sous-système JES (*Job Entry Subsytem*). Le sous-système JES (figure 11.4) est responsable de la prise en compte des travaux et de leur exécution auprès du système MVS. Les travaux soumis, avec leur fichier de données d'entrées (SYSIN) sont stockés dans une zone particulière de disque appelée le SPOOL. Les résultats d'une exécution (SYSOUT), de même, sont stockés sous forme d'un fichier dans la même zone de SPOOL. Le sous-système JES est lui-même organisé sous forme de plusieurs initiateurs qui constituent chacun un espace adresse destiné à l'exécution d'un travail batch. Chaque initiateur comprend lui-même plusieurs classes de travaux, auxquelles sont associés des profils d'exécutions. Un travail `JOB` soumis par la commande `SUBMIT JOB` est attaché à un initiateur et à une classe dans cet initiateur, puis placé dans le SPOOL en attente d'être exécuté. Le travail sera exécuté lorsque l'initiateur associé à la classe deviendra actif, et qu'il sera le travail le plus prioritaire du sous-système JES. En effet, par exemple, un initiateur peut devenir actif seulement la nuit, pour exécuter lorsque le maximum de ressource processeur est libre, des gros travaux soumis de jour par les utilisateurs.

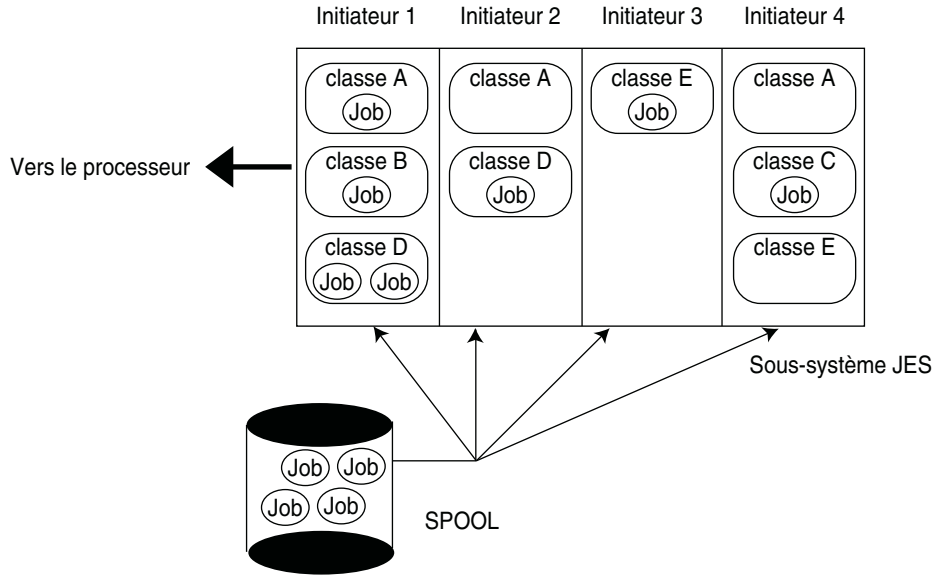


Figure 11.4 Le sous-système JES du système d'exploitation MVS.

Initiateurs et classes MVS

Un initiateur est un espace adresse dans lequel un travail peut s'exécuter. Un initiateur est actif (*active*) si un travail est en train de s'exécuter dans celui-ci. Il est inactif (*inactive*) sinon. Un initiateur drainé (*drained*) est un initiateur

désactivé par l'opérateur : les travaux qui entrent dans cet initiateur ne sont pas exécutés jusqu'à ce que l'initiateur soit réactivé. Un initiateur comprend un certain nombre de classes, caractérisées par une priorité qui est donnée par l'ordre d'apparition de la classe dans la liste de celles-ci. Ainsi, l'initiateur A comprend trois classes A, W, M, la classe A est plus prioritaire que la classe W qui est elle-même plus prioritaire que la classe M. Dans cet initiateur A, on constate que le travail de nom DE106RL est en train de s'exécuter au sein de la classe M. L'étape du travail en exécution (*stepname*) est l'étape EX (se reporter au paragraphe 11.4.3).

```
Display  Filter  View  Print  Options  Help

SDSF INITIATOR DISPLAY                                          LINE 1 – 12 (12)
COMMAND INPUT ===>                                              SCROLL ===> CSR
PREFIX=*  DEST=(ALL)  OWNER=*

NP  ID  STATUS     CLASS  JOBNAME  STEPNAME  PROC   TYPE  JNUM  C      JOBID
    A   ACTIVE     AWM    DE106RL  EX        STEP1  JOB   2893  M  0047  JOB0289
    B   ACTIVE     MDW    DE3162C  ST094           JOB   2894  D  0046  JOB0289
    C   INACTIVE   DM                                             004A
    D   ACTIVE     AWM    A99PRD   STEP5     PROC1  JOB   2896  W  004B  JOB0289
    E   INACTIVE   WM                                             004C
    F   ACTIVE     CSP    DPSSYS   ALLOC           JOB   2895  C  004D  JOB0289
    G   INACTIVE   Q                                              004E
    H   DRAINED    D
    I   DRAINED    P
    J   DRAINED    W
    K   DRAINED    S
    L   DRAINED    Z
```

Figure 11.5 Les initiateurs et les classes MVS.

11.3.2 Les systèmes interactifs

La particularité d'un système d'exploitation interactif est qu'au contraire des systèmes précédents, l'utilisateur de la machine peut interagir avec l'exécution de son programme. Typiquement, l'utilisateur lance l'exécution de son travail et attend derrière le clavier et l'écran, le résultat de celle-ci. S'il s'aperçoit que l'exécution n'est pas conforme à son espérance, il peut immédiatement agir pour arrêter celle-ci et analyser les raisons de l'échec.

Puisque l'utilisateur attend derrière son clavier et son écran et que par nature, l'utilisateur de la machine est un être impatient, le but principal poursuivi par les systèmes interactifs va être d'offrir pour chaque exécution le plus petit temps de réponse possible. Le *temps de réponse d'une exécution* est le temps écoulé entre le moment où l'exécution est demandée par l'utilisateur et le moment où l'exécution est achevée. Pour parvenir à ce but, la plupart des systèmes interactifs travaillent en temps partagé.

Un système en temps partagé permet aux différents utilisateurs de partager l'ordinateur simultanément, tout en ayant par ailleurs la sensation d'être seul à utiliser la

machine. Ce principe repose notamment sur un partage de l'utilisation du processeur par les différents programmes des différents utilisateurs. Chaque programme occupe à tour de rôle le processeur pour un court laps de temps (le *quantum*) et les exécutions se succèdent suffisamment rapidement sur le processeur pour que l'utilisateur ait l'impression que son travail dispose seul du processeur.

Un exemple de système interactif fonctionnant en temps partagé est le système Unix ou le système Linux. Par ailleurs, beaucoup de systèmes à traitements par lots ont été modifiés pour offrir également un service de traitement interactif : c'est le cas du système MVS évoqué dans le paragraphe précédent qui supporte également un sous-système en temps partagé appelé TSO (*Time-Sharing Option*).

11.3.3 Les systèmes temps réel

Les *systèmes temps réel* (figure 11.6) sont des systèmes liés au contrôle de *procédé* pour lesquels la caractéristique primordiale est que les exécutions de programmes sont soumises à des contraintes temporelles, c'est-à-dire qu'une exécution de programme est notamment qualifiée par une date butoir de fin d'exécution, appelée *échéance*, au-delà de laquelle les résultats de l'exécution ne sont plus valides. Des exemples de système temps réel sont par exemple le pilotage automatique d'un train

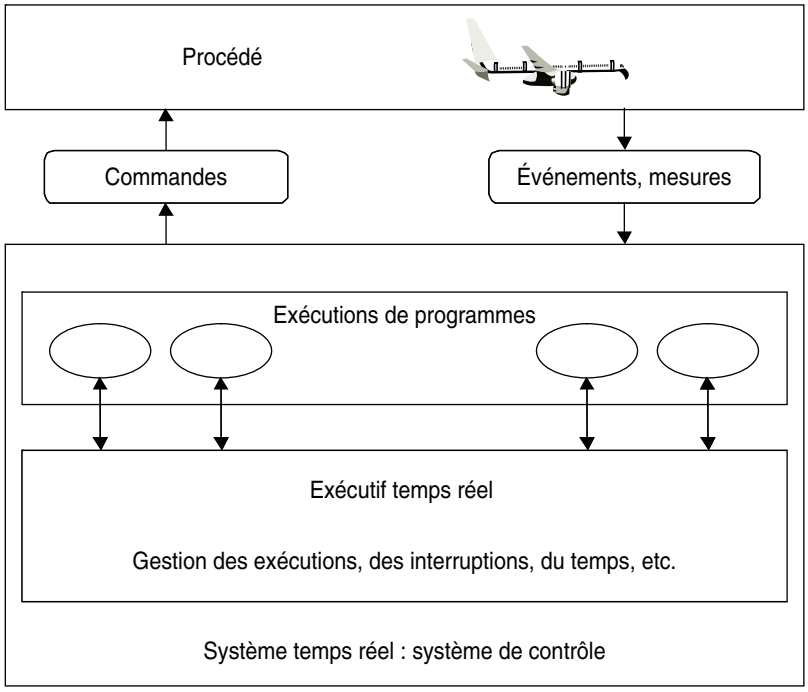


Figure 11.6 Système temps réel.

tel que EOLE ou encore le dispositif de surveillance d'un réacteur de centrale nucléaire. Le procédé désigne ici soit le train EOLE, soit le réacteur nucléaire. Dans le cas même de EOLE, le système temps réel sera qualifié de *système embarqué*, puisqu'il est installé à bord du train. Dans de nombreux cas par ailleurs, le système temps réel est également qualifié de *système réactif*, car ce système reçoit des informations du procédé auquel il se doit de réagir dans les temps impartis. Dans le cas de EOLE, par exemple, le système recevra des informations relatives à l'approche d'une station et devra entreprendre des actions de freinage pour être à même de s'arrêter à temps.

Il ne s'agit pas de rendre le résultat le plus vite possible, mais simplement à temps. L'échelle du temps relative à la contrainte temporelle varie d'une application à l'autre : elle peut être par exemple de l'ordre de la microseconde dans des applications de contrôle radars, mais peut être de l'ordre de l'heure pour une application de contrôle chimique. Par contre, il est souvent primordial de respecter la contrainte temporelle, sous peine de graves défaillances, pouvant mettre en péril le procédé lui-même et son environnement. Tout retard de réaction vis-à-vis d'une situation anormale au sein du réacteur nucléaire, peut évidemment mener à une situation catastrophique !

Pour être en mesure de respecter les contraintes temporelles associées aux exécutions de programmes, le système temps réel doit offrir un certain nombre de mécanismes particuliers, dont le but est de réduire au maximum tout indéterminisme au niveau des durées des exécutions et de garantir par là même que les contraintes temporelles seront respectées. Par exemple, le processeur sera de préférence alloué à l'exécution la plus urgente et les choix de conception du système lui-même seront tels que les ressources physiques comme la mémoire ou encore les périphériques seront gérés avec des méthodes simples, limitant les fluctuations et les attentes indéterminées.

Le système temps réel est souvent qualifié *d'exécutif temps réel*. Des exemples de tels systèmes sont les exécutifs LynxOS de la société Lynx Real-Time Systems ou encore VxWORKS de la société Wind River Systems qui sont des systèmes exclusivement dédiés au temps réel et conçus en tant que tels. Notons qu'il existe également des systèmes temps réel, dérivés de systèmes interactifs préexistants : c'est le cas par exemple du système RTLinux, dérivé temps réel du système Linux.

11.4 NOTIONS DE BASE

Comme nous l'avons déjà évoqué précédemment, le système d'exploitation s'interface avec les applications du niveau utilisateur par le biais de *fonctions prédéfinies* qualifiées *de routines systèmes* et qui constituent les points d'entrées des fonctionnalités du système. Ces appels peuvent être de deux natures et se faire soit par le biais d'un appel système, soit par le biais d'une commande du langage de commandes.

L'exécution des routines systèmes s'effectue sous un mode privilégié, appelé *mode superviseur* ou *mode maître*.

11.4.1 Modes d'exécutions et commutations de contexte

Modes d'exécutions

Un programme utilisateur s'exécute par défaut selon un mode qualifié de *mode esclave* ou *mode utilisateur* : ce mode d'exécution est un mode pour lequel les actions pouvant être entreprises par le programme sont volontairement restreintes afin de protéger la machine des actions parfois malencontreuses du programmeur. Notamment, le jeu d'instructions utilisables par le programme en mode utilisateur est réduit, et spécialement les instructions permettant la manipulation des interruptions est interdite.

Le système d'exploitation, quant à lui, s'exécute dans un mode privilégié encore appelé *mode superviseur*, pour lequel aucune restriction de droits n'existe.

Le codage du mode esclave et du mode maître est réalisé au niveau du processeur, dans le registre d'état de celui-ci.

Modes d'exécutions du processeur 68000 de Motorola

Le registre d'état du processeur 68000 est un registre 16 bits, dans lequel le bit 13 permet le codage du mode d'exécution du processeur, soit utilisateur pour lequel le jeu d'instructions utilisables vis-à-vis du processeur est réduit (interdiction notamment d'utilisation des instructions permettant la modification du registre d'état et interdiction de certaines opérations sur la pile), soit superviseur pour lequel toutes les instructions sont utilisables.

Commutations de contexte

Aussi, lorsqu'un programme utilisateur demande l'exécution d'une routine du système d'exploitation par le biais d'un appel système, ce programme quitte son mode courant d'exécution (le mode esclave) pour passer en mode d'exécution du système, c'est-à-dire le mode superviseur. Ce passage du mode utilisateur au mode superviseur constitue une *commutation de contexte* : elle s'accompagne d'une *opération de sauvegarde du contexte* utilisateur, c'est-à-dire principalement de la valeur des registres du processeur (CO, PSW). Lorsque l'exécution de la fonction système est achevée, le programme repasse du mode superviseur au mode utilisateur. Il y a de nouveau une opération de commutation de contexte avec *restauration du contexte utilisateur* sauvegardé lors de l'appel système, ce qui permet de reprendre l'exécution du programme utilisateur juste après l'appel (figure 11.7).

Trois causes majeures provoquent le passage du mode utilisateur au mode superviseur (figure 11.8) :

– le fait que le programme utilisateur appelle une fonction du système. C'est une demande explicite de passage en mode superviseur ;
– l'exécution par le programme utilisateur d'une opération illicite (division par 0, instruction machine interdite, violation mémoire…) : c'est *la trappe ou l'exception*. L'exécution du programme utilisateur est alors arrêtée ;

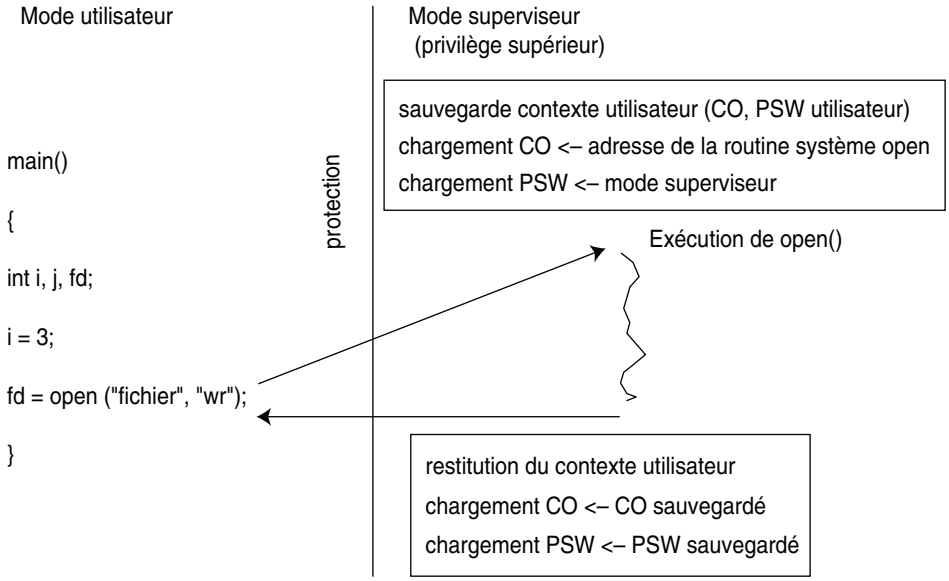


Figure 11.7 Commutations de contexte.

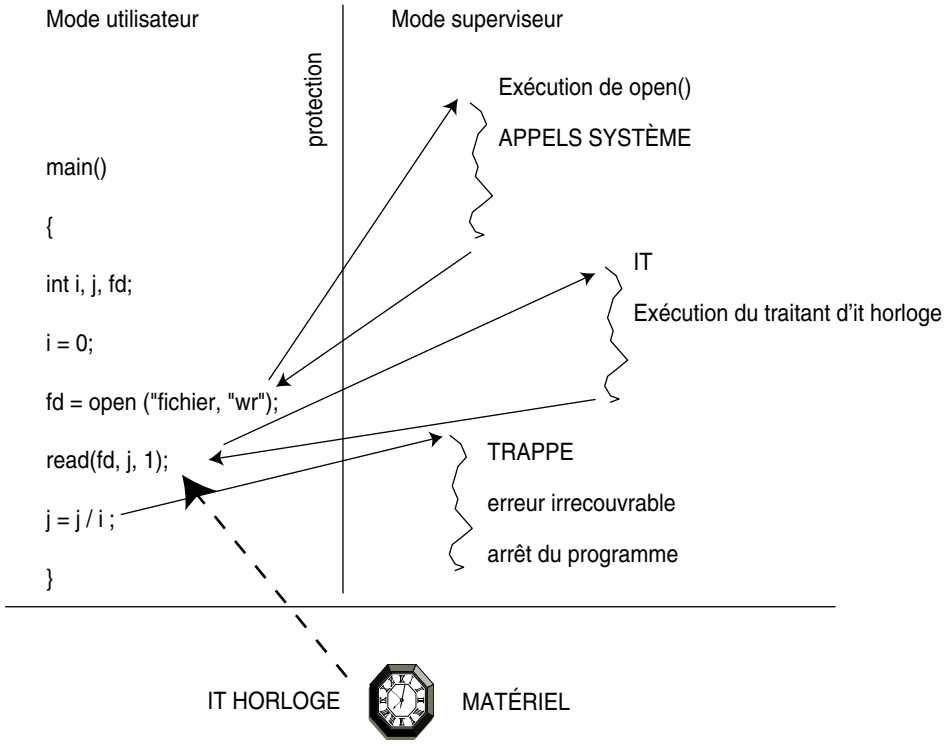


Figure 11.8 Trois causes de commutations de contexte.

– la prise en compte d'une interruption par le matériel et le système d'exploitation. Le programme utilisateur est alors stoppé et l'exécution de la routine d'interruption associée à l'interruption survenue est exécutée en mode superviseur.

Trappes et appels systèmes sont parfois qualifiés *d'interruptions logicielles* pour les opposer aux *interruptions matérielles* levées par les périphériques du processeur.

11.4.2 Gestion des interruptions matérielles et logicielles

Prise en compte d'une interruption matérielle

Comme il a été décrit au chapitre 7, une interruption est un signal permettant à un dispositif externe au processeur (un périphérique d'entrées-sorties notamment) d'interrompre le travail courant du processeur pour aller réaliser un traitement particulier lié à la cause de l'interruption, appelé procédure, programme ou traitant d'interruption. Si le niveau d'exécution du processeur est plus bas que celui de l'interruption, le processeur accepte l'interruption avant de décoder l'instruction machine suivante et relève son niveau d'exécution afin qu'aucune autre interruption de niveau inférieure ou égale ne puisse l'interrompre.

Les traitants d'interruptions sont des routines exécutées en mode superviseur, chargées en mémoire centrale au moment du chargement du système d'exploitation et dont les adresses en mémoire centrale sont contenues dans une table également placée en mémoire centrale, la table des vecteurs d'interruptions.

Lors de la survenue d'une interruption – par exemple l'interruption numéro 3 tel que le montre la figure 11.9 – le programme utilisateur en cours d'exécution est

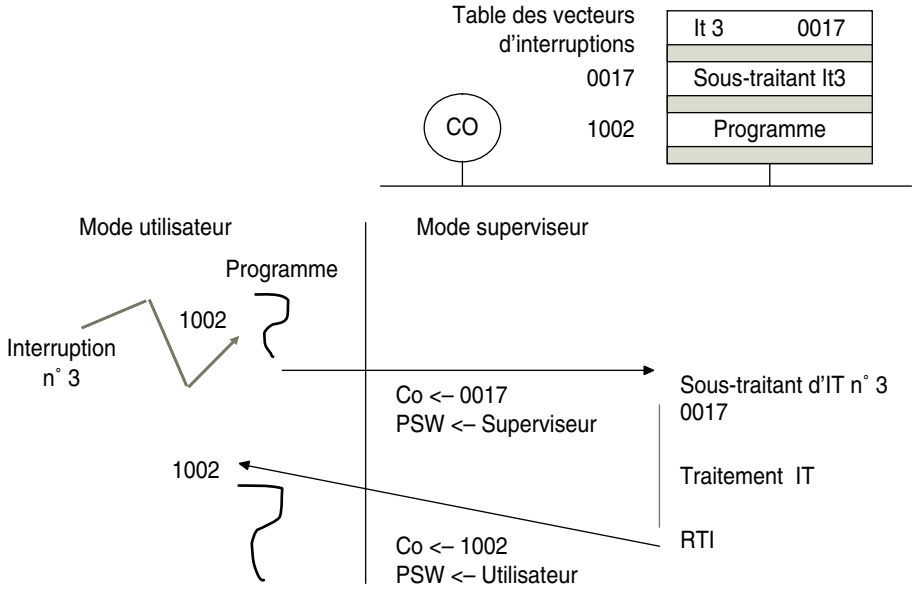


Figure 11.9 Prise en compte d'une interruption.

stoppé. Il quitte le mode utilisateur et entre en mode superviseur afin d'aller exécuter la routine d'interruption associée à l'interruption prise en compte. Le changement de mode d'exécution s'accompagne d'une sauvegarde du contexte utilisateur dont le but est de mémoriser les informations relatives à l'exécution de ce programme afin de pouvoir reprendre celle-ci, une fois l'interruption traitée. Cette sauvegarde est en partie réalisée par le matériel, qui commute le registre d'état du processeur et sauvegarde la valeur du compteur ordinal pour le programme utilisateur. Une sauvegarde complémentaire des registres généraux de la machine est réalisée par le système d'exploitation. Puis le compteur ordinal est chargé avec l'adresse du traitant d'interruption associé à l'interruption en cours de prise en compte (par exemple, l'adresse 0017).

Une fois le traitant d'interruption exécuté, le contexte utilisateur sauvegardé est restauré (instruction machine RTI, *ReTurn Interrupt*) et le programme utilisateur reprend son exécution en mode esclave, à l'instruction où il avait été dérouté.

Gestion d'une trappe

L'occurrence d'une trappe est levée lorsque le processeur rencontre une situation anormale en cours d'exécution, par exemple une division par 0, un débordement, l'utilisation d'une instruction interdite ou encore un accès mémoire interdit.

Le traitement d'une trappe est très comparable à celui des interruptions, évoqué au paragraphe précédent. La différence essentielle réside dans le fait que l'occurrence d'une trappe est synchrone avec l'exécution du programme. Aussi, de la même manière que pour les interruptions, une trappe est caractérisée par un numéro et une routine de traitement de l'erreur lui est associée dont l'adresse en mémoire est rangée dans une table qui peut être la table des interruptions ou bien être une table différente.

Le traitement associé à la trappe consiste souvent à arrêter le programme en cours d'exécution et à créer une image mémoire de son exécution qui pourra être utilisée par le programmeur pour comprendre le problème survenu (fichier core sous Linux).

Exécution d'une fonction du système

L'appel a une fonction du système peut être réalisée soit depuis un programme utilisateur par le biais d'un appel de procédure, qualifié dans ce cas d'appel système, soit par le biais d'une commande. La commande est prise en compte par un programme particulier, l'interpréteur de commandes, qui à son tour invoque la fonction système correspondante. Ainsi, sous Unix, l'appel système `chdir(nom_repertoire)` permet de changer le répertoire courant du programme en cours d'exécution. La commande `cd nom_repertoire` effectue le même travail depuis l'interpréteur de commandes (*le shell*). L'appel système `chdir` ou la commande `cd` aboutissent tous les deux à l'exécution de la routine système `_chdir`.

L'invocation d'une fonction système tient à la fois de l'appel de procédure et du traitement d'interruption. Elle est similaire à un appel de procédure, dans le sens où des paramètres d'appels sont souvent passés au système depuis le mode utilisateur et un résultat retourné au mode utilisateur à l'issue de l'exécution, mais elle en diffère grandement dans le sens où l'invocation de la fonction système entraîne un change-

ment de mode d'exécution. Par ailleurs, elle se rapproche du traitement des interruptions car l'appel à la fonction du système se traduit en fait par la levée d'une interruption logicielle particulière, entraînant le changement de mode et le branchement à l'adresse de la fonction. On pourrait finalement dire que l'invocation d'une fonction système est une levée d'interruption avec passage de paramètres et de résultat entre les deux modes d'exécution du processeur.

Plus précisément, l'ensemble des appels système disponibles est regroupé dans une ou plusieurs bibliothèques, avec lesquelles l'éditeur de liens fait la liaison comme nous l'avons vu au chapitre 3. Chaque fonction de la bibliothèque comprend une instruction permettant le changement de mode d'exécution, puis des instructions assurant le passage des paramètres depuis le mode utilisateur vers le mode superviseur. Souvent, des registres généraux prédéfinis du processeur servent à ce transfert de paramètres (par exemple, les registres R0, R1 et R2). Enfin, de manière similaire à ce qui est fait pour une exception, la fonction lève une trappe en passant au système un numéro spécifique à la routine appelée. Le système cherche dans une table, l'adresse de la routine identifiée par le numéro et se branche sur son code. À la fin de l'exécution de la routine, la fonction de la bibliothèque retourne les résultats de l'exécution via les registres réservés à cet effet, puis restaure le contexte du programme utilisateur.

Réalisation d'un appel système sous Linux

Lorsqu'un processus exécute un appel système, il appelle la fonction correspondante dans une bibliothèque C. Cette fonction traite les paramètres d'appels et fait passer le processus en mode superviseur. Plus précisément, sur une architecture de type x86, les paramètres de l'appel système sont passés via les registres eax, ebx, ecx et edx du processeur, puis une trappe est déclenchée en activant l'interruption 0x80. Cette trappe provoque le passage du processus en mode superviseur et l'exécution de la fonction `system_call`. Cette fonction utilise le numéro de l'appel système (par exemple 12 pour `_chdir`) passé via le registre eax pour appeler la fonction correspondant à l'appel système.

Exemple : les interruptions sur le système IBM 370

➤ Les différents types d'interruptions

Sur l'architecture IBM 370, six sortes d'interruptions sont répertoriées :

– les interruptions matérielles liées à la gestion des entrées-sorties ;
– les interruptions matérielles liées à la signalisation d'un mauvais fonctionnement de l'ordinateur (problème d'alimentation électrique ou problème de refroidissement, etc.) ;

– l'interruption de redémarrage ;
– les interruptions externes levées par l'opérateur de la machine ou un autre processeur de la machine ;
– les interruptions programmes (*program check*) qui correspondent à la levée de trappes suite à une erreur incorrigible dans un programme. Le travail fautif est arrêté et provoque un ABEND (*abnormal end*) ;
– les appels superviseur (*SVC Call*) qui correspondent aux appels aux routines du système. Il y a 256 sortes de SVC portant les numéros 0 à 255.

➤ Les états du processeur

Le processeur fonctionne selon deux états différents : l'état problème (*problem program*) qui correspond au mode esclave et l'état superviseur. Cet état est codé dans le 16e bit du registre d'état du processeur (PSW) qui en compte 64 au total (1 pour l'état problème, 0 pour l'état superviseur). Par ailleurs, les bits 35 à 63 contiennent l'adresse de la prochaine instruction à exécuter dans le programme et constituent donc l'adresse de reprise du programme interrompu (CO).

➤ Mécanisme de traitement d'une interruption

Ce mécanisme comporte principalement quatre étapes :
– le PSW courant est sauvegardé dans une zone fixe de la mémoire. Un nouveau PSW spécifique du type de l'interruption levée est chargé. Il y a 6 couples ancien PSW/nouveau PSW correspondant aux 6 types d'interruptions mentionnées plus haut. Ces 6 couples sont créés au démarrage du système ;
– le nouveau PSW branche le processeur vers une première routine d'interruption appelée routine de premier niveau ou FLIHs (*First Level Interrupt Handler*) particulière du type d'interruption levée. Cette routine est résidente en mémoire. Elle sauvegarde les registres du processeur dans une zone réservée de la mémoire centrale ;
– une routine de traitement d'interruption de second niveau ou SLIH (*Second Level Interrupt Handler*) est maintenant appelée pour répondre exactement à l'interruption. Selon le type de l'interruption, il s'agit soit d'une routine du superviseur d'entrées-sorties, soit d'une routine du superviseur de terminaison responsable de la reprise en cas d'erreur, soit d'une routine spécifique aux interruptions externes, soit d'une routine SVC ;
– le PSW sauvegardé à la première étape est restauré, une fois l'exécution de la routine d'interruption de second niveau achevée.

11.4.3 Langage de commande

Définition

Le langage de commande constitue l'interface de niveau utilisateur avec le système d'exploitation. Les commandes composant le langage sont analysées par

l'outil interpréteur de commandes qui appelle la routine système appropriée en assurant le passage des paramètres éventuels via les registres du processeur réservés à cet effet.

Chaque système d'exploitation a son langage de commandes propre. Nous prenons ci-après deux exemples : d'une part un exemple de langage de commandes dans le cadre d'un système d'exploitation orienté batch, avec la présentation du langage JCL du système MVS, d'autre part un exemple de langage de commandes dans le cadre d'un système d'exploitation interactif, avec la présentation du langage Shell du système Unix.

Exemples : Shell Unix et langage JCL de MVS

➤ Le langage JCL de MVS

Le langage JCL (*Job Control Language*) est un langage de commande orienté batch qui permet de décrire au sous-système JES les travaux à exécuter.

Un ordre JCL est constitué par un ou plusieurs enregistrements de 80 octets encore appelés cartes, par référence aux cartes perforées sur lesquelles les ordres JCL étaient initialement codés. Un ordre JCL a la forme :

```
//nom commande paramètre1, paramètre2, …, commentaires.
```

Dans un système orienté batch, l'utilisateur définit dans un fichier de travail (le Job) l'ensemble des commandes (les étapes du job) qu'il souhaite voir exécuter en leur associant les fichiers de données en entrée et en sortie (les DD).

Trois commandes ou cartes principales sont définies :

– JOB : cette commande indique le début d'un travail, défini par un nom et une classe d'exécution au sein du sous-système JES. Un travail est lui-même constitué par plusieurs étapes (au maximum 255) auxquelles sont associés des fichiers de données ;
– EXEC : cette commande définit le début d'une étape au sein d'un travail et indique le nom du programme à exécuter au cours de cette étape ;
– DD : cette commande permet de définir les caractéristiques d'un fichier utilisé par l'étape. Il y a autant d'ordres DD dans une étape qu'il y a de fichiers utilisés par celle-ci. Un ordre DD est caractérisé par un nom, le *DDname*. Parmi ces entrées DD, SYSIN correspond aux données en entrée, SYSOUT aux données en sortie tandis que SYSPRINT correspond à l'enregistrement des traces d'exécution.

Voici un exemple de travail défini à l'aide du langage JCL. Le job porte le nom `DSE25CPY` et sera soumis à la classe A de l'outil `JES`. L'étape 1 du travail demande l'exécution d'un programme nommé `IDCAMS`, qui effectue la destruction de deux fichiers `'MDP1.DB90.TEMP01'`et `'MDP1.DB90.TEMP02'`. L'étape 2 du travail demande l'exécution d'un programme nommé `IEBCOPY`, qui effectue la copie d'un fichier `SYS4.KF31.LOADTP` dans le fichier `DSE25.UTIL.LOAD.KF` puis la copie du fichier `SYS4.KF31.LOADBT` dans le même fichier `DSE25.UTIL.LOAD.KF`.

```
//DSE25CPY JOB (), 'DELETE COPY', CLASS = A, MSGCLASS = X,
//                NOTIFY = DSE25, MSGLEVEL = (1, 1)
//*-------------------------------------------------------------
//*      DESTRUCTION DE FICHIERS TEMPORAIRES
//*-------------------------------------------------------------
//STEP01      EXEC PGM = IDCAMS
//SYSPRINT    DD SYSOUT = *
//SYSOUT      DD SYSOUT = *
//SYSIN       DD *
DEL 'MDP1.DB90.TEMP01'
DEL 'MDP1.DB90.TEMP02'
/*
//-------------------------------------------------------------
//       COPIE DE BIBLIOTHEQUES
//-------------------------------------------------------------
//STEP02      EXEC PGM = IEBCOPY, REGION = 6144K
//SYSPRINT    DD SYSOUT = *
//SYSOUT      DD SYSOUT = *
//SYSUT1      DD UNIT = WORK, SPACE = (CYL, (1, 1))
//SYSUT2      DD UNIT = WORK, SPACE = (CYL, (1, 1))
//I1          DD DISP = SHR, DSN = SYS4.KF31.LOADTP
//I2          DD DISP = SHR, DSN = SYS4.KF31.LOADBT
//O1          DD DISP = SHR, DSN = DSE25.UTIL.LOAD.KF
//SYSIN       DD*
COPY INDD = I1, OUTDD = O1
COPY INDD = I2, OUTDD = O1
/*
//
```

➤ Le shell d'Unix

Sous Unix, l'utilisateur dispose de nombreuses commandes. Une commande est constituée par un nom, suivi éventuellement d'un ensemble d'arguments. L'exécution des commandes s'effectue en mode interactif, c'est-à-dire que l'utilisateur frappe une commande, qui s'exécute immédiatement et rend le résultat avec le plus court temps de réponse possible.

L'exécution des commandes en mode interactif s'inscrit pour chaque utilisateur dans le cadre d'une *session de travail*. La session de travail est ouverte après que l'utilisateur se soit connecté au système, en fournissant à celui-ci son identifiant (*le login*) et son mot de passe. La session se termine lorsque l'utilisateur se déconnecte. La prise en charge des commandes lancées par l'utilisateur au cours d'une session est assurée par l'interpréteur de commande (le shell) qui est démarré par le système au début de la session. Cet outil attend les commandes de l'utilisateur pour les exécuter, cette attente étant matérialisée par un caractère spécial ($, >, #) appelé le *prompt ou l'invite* (figure 11.10).

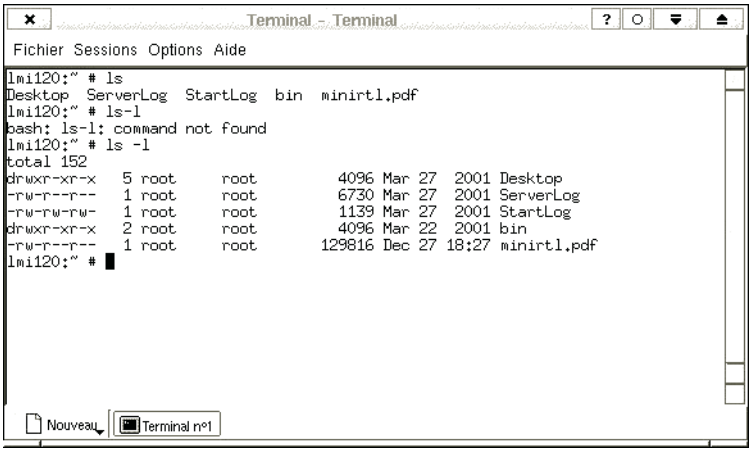


Figure 11.10 Session utilisateur avec invite du shell lmi 20 :~ #.

Les principales commandes du shell concernent essentiellement :

– la délivrance d'informations générales sur le système (commande `date` délivrant la date et l'heure du système, commande `who` délivrant la liste des utilisateurs connectés…);
– la gestion des fichiers (commandes `cp` pour la copie de fichiers, `cd` pour le changement de répertoire, `rm` pour la destruction de fichier, `rmdir` et `mkdir` pour la destruction et la création de répertoire, `ls` pour lister les fichiers d'un répertoire…);
– la gestion des exécutions de programmes (commandes `batch` pour le lancement de commandes en différé, `ps` pour obtenir des informations sur les exécutions de programmes en cours…);
– la communication entre utilisateurs (commande `mail` pour l'envoi de courrier électronique, commande `talk` pour assurer le dialogue entre deux utilisateurs connectés…).

11.5 GÉNÉRATION ET CHARGEMENT D'UN SYSTÈME D'EXPLOITATION

11.5.1 Génération d'un système d'exploitation

Le processus de génération d'un système d'exploitation désigne l'opération consistant à configurer le système en fonction de la spécificité du site sur lequel il est appelé à s'exécuter. Cette génération est effectuée par un utilitaire particulier, SYSGEN, livré avec le système, qui rend le système opérationnel pour la machine cible à partir des paramètres fournis par l'administrateur et des bibliothèques de distribution. Les paramètres fournis par l'administrateur de la machine concernent par exemple la définition du type de l'unité centrale, la définition de la quantité de mémoire centrale

disponible, la liste des périphériques disponibles et le choix des différentes options du système d'exploitation.

La génération d'un système tel que MVS s'effectue en deux étapes. La première étape (STAGE1) concerne la définition des paramètres. Un JCL est créé en fonction de ces paramètres, qui est exécuté lors de la seconde étape du processus (STAGE2). Cette exécution génère à partir des bibliothèques de distribution fournies par IBM, le système d'exploitation en utilisant des opérations de copies de bibliothèques, d'assemblage et d'édition des liens.

11.5.2 Chargement d'un système d'exploitation

Les programmes composant le système d'exploitation généré sont conservés sur un support de masse non volatile. Lorsque l'ordinateur commence à fonctionner, il exécute un premier code placé dans une zone de mémoire morte (ROM). Ce premier programme, appelé *programme d'amorçage* ou *boot-strap*, effectue tout d'abord un test du matériel de la machine, puis il charge à partir du support de masse, un programme d'amorce plus sophistiqué. Cette amorce est placée dans une zone particulière du support de masse, par exemple, dans le bloc 0 du disque qualifié alors de *bloc d'amorçage*. Ce disque constitue *le disque système*. Une fois cette amorce placée en mémoire centrale, elle s'exécute et charge à leur tour les programmes du système d'exploitation.

Le chargement d'un système MVS se déroule en trois étapes. La première étape appelée IML (*initial microcode load*) effectue un test de la mémoire de l'ordinateur puis elle charge à partir d'un disque dont l'adresse est conservée dans une mémoire ROM, une image de la configuration matérielle de la machine (l'IOCDS) lui permettant notamment de connaître l'adresse de la console opérateur et l'adresse du disque système (le SYSRES). La deuxième étape appelée IPL (*initial program load*) est déclenchée depuis la console système par la fonction LOAD; elle permet le chargement depuis le SYSRES du texte de chargement (texte d'IPL) placé dans le bloc 0 du SYSRES. Ce texte d'IPL recherche sur le disque le fichier SYS1.NUCLEUS qui contient des modules d'initialisation du système. L'exécution de ces modules constitue la dernière étape de l'initialisation du système, le NIP (*nucleus initialization program*), qui met en place les différents composants du système en utilisant notamment les descriptions faites dans le fichier système PARMLIB. Le chargement complet du système est une opération qui peut durer une demi-heure sur un système de moyenne importance. Il faut noter que l'opération d'IML n'est nécessaire que si la machine a été complètement mise hors tension.

11.6 CONCLUSION

Ce premier chapitre consacré aux systèmes d'exploitation multiprogrammés nous a permis de définir un système d'exploitation comme étant un ensemble de programmes qui réalisent l'interface entre le matériel de l'ordinateur et les utilisateurs. Les deux objectifs principaux de cette interface sont :

– construire au-dessus du matériel d'une machine virtuelle plus facile d'emploi et plus conviviale ;
– prendre en charge de la gestion de plus en plus complexe des ressources et partage de celles-ci.

Les fonctionnalités du système d'exploitation sont accessibles par le biais des commandes ou des appels système.

Le mode superviseur est le mode d'exécution du système. C'est un mode d'exécution privilégié qui autorise notamment l'appel à des instructions interdites en mode utilisateur (manipulation des interruptions). Ce mode assure la protection du système d'exploitation. Le passage du mode utilisateur vers le mode superviseur est soit provoqué par un appel système, soit par une trappe, soit par l'arrivée d'une interruption. Il s'accompagne de commutations de contexte qui consiste en la séquence : sauvegarde du contexte utilisateur – changement de mode d'exécution – restitution du contexte utilisateur.

Chapitre **12**

Gestion de l'exécution des programmes : le processus

Nous commençons à présent l'étude du fonctionnement d'un système multiprogrammé. Dans ce chapitre, nous nous intéressons à la fonction d'exécution qui recouvre principalement deux notions : celle de *processus* qui correspond à l'image d'un programme qui s'exécute et celle d'*ordonnancement* qui correspond au problème de l'allocation du processeur et donc du partage du processeur entre différents processus. Enfin, nous terminons cette partie en abordant les problèmes de synchronisation et de communication entre processus.

12.1 NOTION DE PROCESSUS

12.1.1 Définitions

Comme nous l'avons étudié au chapitre 3, la chaîne de production de programme transforme un programme écrit dans un langage de haut niveau en un programme dit exécutable, écrit en langage machine. Ce programme exécutable est stocké sur le disque. À l'issue du chargement, il est placé en mémoire centrale pour pouvoir être exécuté.

Imaginons que le programme à exécuter est placé en mémoire centrale à partir de l'emplacement d'adresse $0A10_{16}$. Le processeur commence l'exécution du programme : la première instruction de celui-ci est chargée dans le registre instruction[1] (RI) et le

1. Une instruction est supposée avoir une longueur de 4 octets.

compteur ordinal (CO) contient l'adresse de la prochaine instruction à exécuter soit $0A14_{16}$. Lorsque l'instruction courante a été exécutée, le processeur charge dans le registre RI l'instruction pointée par le CO, soit par exemple l'instruction `add Im R1 5` et le compteur ordinal prend la valeur $0A18_{16}$. L'exécution de l'instruction `add Im R1 5` modifie le contenu du registre PSW puisque c'est une instruction arithmétique : les drapeaux de signe, de nullité etc. sont mis à jour. Ainsi, à chaque étape d'exécution du programme, le contenu des registres du processeur évolue. De même le contenu de la mémoire centrale peut être modifié par des opérations d'écriture ou de lecture dans la pile (instructions push, pop) ou encore des opérations de modification des données (instruction store).

On appelle *processus* l'image de l'état du processeur et de la mémoire au cours de l'exécution d'un programme. Le programme est statique et le processus représente la dynamique de son exécution. Plus précisément, les définitions suivantes permettent de caractériser ce qu'est un processus :

– un processus est un programme en cours d'exécution auquel est associé un environnement processeur (CO, PSW, RSP, registres généraux) et un environnement mémoire (zone de code, de données et de pile) appelés *contexte du processus*;
– un processus est l'instance dynamique d'un programme et incarne le fil d'exécution de celui-ci dans un espace d'adressage protégé, c'est-à-dire sur une plage mémoire dont l'accès lui est réservé.

Par ailleurs, un *programme réentrant* est un programme pour lequel il peut exister plusieurs instances d'exécutions simultanées (n par exemple), c'est-à-dire que le programme peut être exécuté n fois en même temps. Il y a alors n processus correspondant à n exécutions différentes et indépendantes du même programme, chaque processus évoluant à son propre rythme.

12.1.2 États d'un processus

Lors de son exécution, un processus est caractérisé par un état (figure 12.1) :

– lorsque le processus obtient le processeur et s'exécute, il est dans l'état *élu*. L'état élu est l'état d'exécution du processus;
– lors de cette exécution, le processus peut demander à accéder à une ressource, par exemple il demande à lire des données depuis le disque. Le processus ne peut pas poursuivre son exécution tant qu'il n'a pas obtenu la ressource. Selon notre exemple, le processus doit attendre la fin de l'opération d'entrées-sorties disque pour disposer des données sur lesquelles il pourra effectuer les calculs suivants de son code. Le processus quitte alors le processeur et passe dans l'état *bloqué*. L'état bloqué est l'état d'attente d'une ressource autre que le processeur;
– lorsque le processus a enfin obtenu la ressource qu'il attendait, celui-ci peut potentiellement reprendre son exécution. Cependant, nous nous sommes placés dans le cadre de systèmes multiprogrammés, c'est-à-dire qu'il y a plusieurs programmes en mémoire centrale et donc plusieurs processus. Lorsque le processus est passé dans l'état bloqué, le processeur a été alloué à un autre processus. Le processeur

n'est donc pas forcément libre. Le processus passe alors dans l'état *prêt*. L'état prêt est l'état d'attente du processeur.

Le passage de l'état prêt vers l'état élu constitue l'*opération d'élection*. Le passage de l'état élu vers l'état bloqué est l'*opération de blocage*. Le passage de l'état bloqué vers l'état prêt est l'*opération de déblocage*.

Un processus est toujours créé dans l'état prêt. Un processus se termine toujours à partir de l'état élu (sauf anomalie).

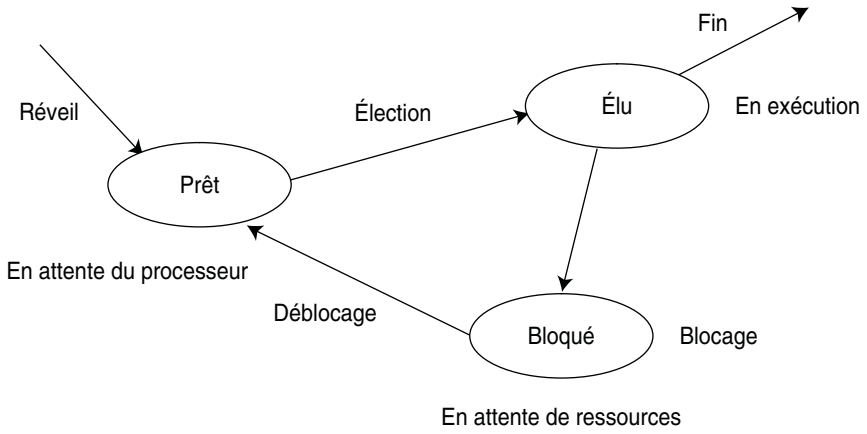


Figure 12.1 Diagramme d'états d'un processus.

12.1.3 Bloc de contrôle du processus

Le chargeur a pour rôle de monter en mémoire centrale le code et les données du programme à exécuter. En même temps que ce chargement, le système d'exploitation crée une structure de description du processus associé au programme exécutable : c'est le *PCB ou Process Control Block* (bloc de contrôle du processus).

Le bloc de contrôle d'un processus contient les informations suivantes (figure 12.2) :

- un identificateur unique du processus (un entier);
- l'état courant du processus (élu, prêt, bloqué);
- le contexte processeur du processus : la valeur du CO, la valeur des autres registres du processeur;
- le contexte mémoire : ce sont des informations mémoire qui permettent de trouver le code et les données du processus en mémoire centrale;
- des informations diverses de comptabilisation pour les statistiques sur les performances du système;
- des informations liées à l'ordonnancement du processus;
- des informations sur les ressources utilisées par le processus, tels que les fichiers ouverts, les outils de synchronisation utilisés, etc.

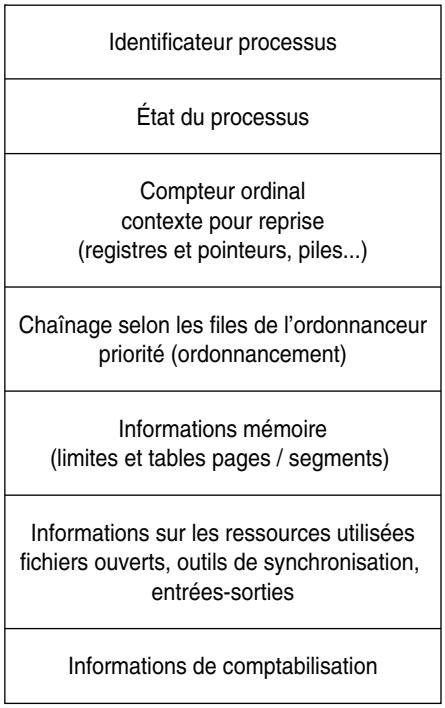


Figure 12.2 Bloc de contrôle de processus.

Le PCB permet la sauvegarde et la restauration du contexte mémoire et du contexte processeur lors des opérations de commutations de contexte.

12.1.4 Opérations sur les processus

Le système d'exploitation offre généralement les opérations suivantes pour la gestion des processus : création de processus, destruction de processus, suspension de l'exécution et reprise de celle-ci.

Création de processus

Un processus peut créer un ou plusieurs autres processus en invoquant un appel système de création de processus. Le processus créateur est généralement appelé le processus père tandis que le ou les processus créés constituent quant à eux les processus fils. Ces processus fils peuvent à leur tour créer des processus, devenant ainsi également des processus père. Ainsi, au fil des opérations de création initiées par les processus, se développe un arbre de filiation entre processus.

Les opérations de création de processus admettent notamment les variantes suivantes selon les systèmes d'exploitation :

– le processus créé hérite ou non de données et du contexte de son processus créateur. Ainsi, le nouveau processus peut exécuter un programme différent de celui

de son créateur (système Dec Vms) ou être constitué comme un clone de celui-ci avec le même code et les mêmes données (système Unix) ;
– le processus créé peut s'exécuter parallèlement à son père. Dans certains systèmes, le processus père doit attendre la terminaison de ses fils pour pouvoir reprendre sa propre exécution.

Exemple

Sous Linux ou Unix, l'appel système `fork` permet à un processus de créer un autre processus qui est une exacte copie de lui-même au moment de l'appel. Le processus fils hérite du code et des données de son père, hormis son identificateur. Ce code hérité peut être modifié pour un autre code par le biais d'un appel à une des routines systèmes de la famille `exec`.

Destruction de processus

La destruction d'un processus intervient :
– lorsque le processus a terminé son exécution. Dans ce cas, le processus s'autodétruit en appelant une routine système de fin d'exécution (par exemple `exit()` sous Unix) ;
– lorsque le processus commet une erreur irrécouvrable. Dans ce cas, une trappe est levée et le processus est terminé par le système (cf. paragraphe 11.4.2) ;
– lorsqu'un autre processus demande la destruction du processus, par le biais d'un appel à une routine système telle que `kill` sous Unix.

Lors de la destruction d'un processus, le contexte de celui-ci est démantelé : les ressources allouées au processus sont libérées et son bloc de contrôle est détruit.

Suspension d'exécution

La suspension d'exécution est une opération qui consiste à momentanément arrêter l'exécution d'un processus pour la reprendre ultérieurement. Lors de l'opération de suspension, le contexte du processus est sauvegardé dans son PCB afin de pouvoir reprendre l'exécution, là où elle a été suspendue. Le processus suspendu passe dans l'état bloqué. Lors de sa reprise d'exécution, le processus franchit la transition de déblocage et entre dans l'état prêt.

Sous les systèmes Linux ou Unix, l'appel système `sleep(duree)` permet de suspendre l'exécution d'un processus pour un temps égal à `duree` secondes.

12.1.5 Un exemple de processus : les processus Unix

Le système Unix est entièrement construit à partir de la notion de processus. Au démarrage du système, un premier processus est créé, le processus 0. Ce processus 0 crée à son tour un autre processus, le processus 1 encore appelé processus `init`. Ce processus `init` lit le fichier `/etc/inittab` et crée chacun des deux types de processus décrits dans ce fichier : d'une part, les processus démons (dont le nom est suffixé par un « d ») qui sont des processus système responsables d'une fonction particulière (`inetd` surveille le réseau, `lpd` gère les imprimantes, `crond` gère un échéancier) et

d'autre part, les processus getty qui surveillent les terminaux. Lorsqu'un utilisateur vient se loguer sur un terminal, un processus login est créé qui lit le nom de l'utilisateur et son mot de passe. Ce processus login vérifie la validité de ces informations en utilisant le fichier des mots de passe. Si les informations sont valides, le processus login ouvre une session pour l'utilisateur et crée un processus shell, interpréteur de commandes (cf. paragraphe 11.4.3). Cet interpréteur de commandes exécute alors les commandes et programmes de l'utilisateur en créant à chaque fois un nouveau processus (figure 12.3).

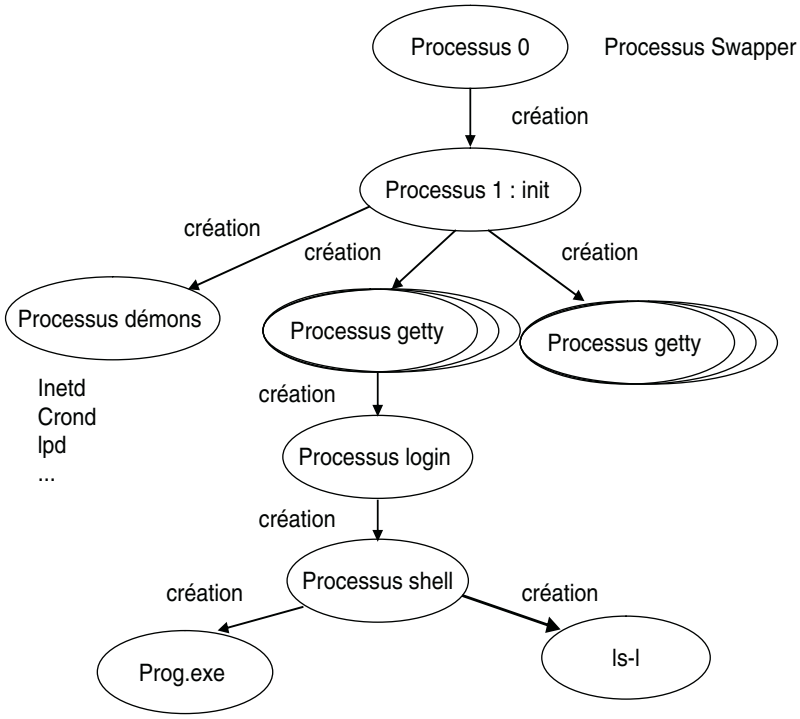


Figure 12.3 Arborescence des processus Unix.

Un processus Unix est décrit par un bloc de contrôle qui est divisé en deux parties (figure 12.4) :

– chaque processus dispose d'une entrée dans une table générale du système, la table des processus. Cette entrée contient les informations sur le processus qui sont toujours utiles au système quel que soit l'état du processus : l'identificateur du processus (pid), l'état du processus, les informations d'ordonnancement, les informations mémoire, c'est-à-dire l'adresse des zones mémoire allouées au processus ;
– chaque processus dispose également d'une autre structure, la Zone U. Cette Zone U contient d'autres informations concernant le processus, mais ce sont des informations qui peuvent être temporairement déplacées sur le disque, notamment lorsque le processus est dans l'état bloqué depuis un certain temps.

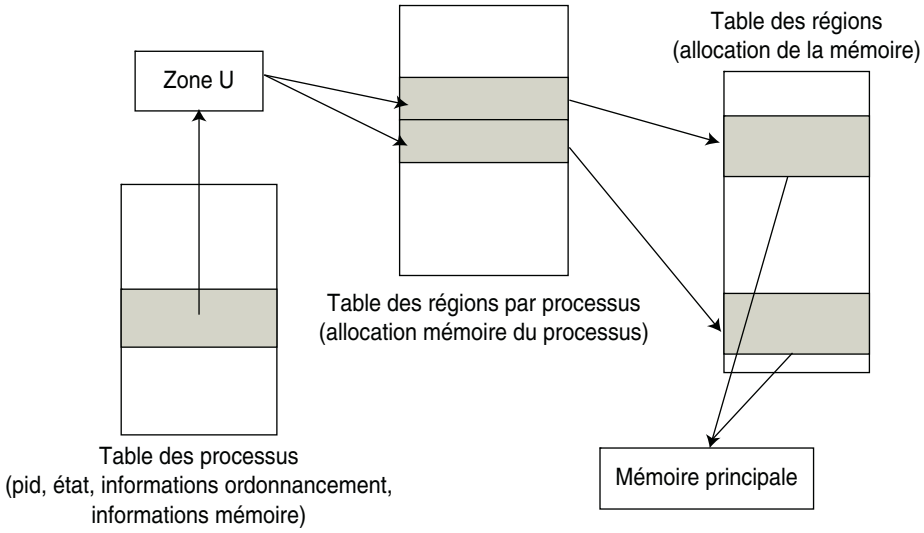


Figure 12.4 Bloc de contrôle d'un processus Unix.

Un processus Unix évolue entre trois modes au cours de son exécution : le mode utilisateur qui est le mode classique d'exécution, le mode noyau en mémoire qui est le mode dans lequel se trouve un processus élu passé en mode superviseur, un processus prêt ou un processus bloqué (endormi) et le mode swappé qui est le mode dans lequel se trouve un processus bloqué (endormi) déchargé de la mémoire centrale. En effet, le système Unix décharge de la mémoire centrale les processus endormis depuis trop longtemps (ils sont alors dans l'état endormi swappé). Ces processus réintègrent la mémoire centrale lorsqu'ils redeviennent prêts (transition de l'état prêt swappé vers l'état prêt).

Un processus Unix qui se termine passe dans un état dit zombi. Il y reste tant que son PCB n'est pas entièrement démantelé par le système. La figure 12.5 résume les états d'un processus Unix.

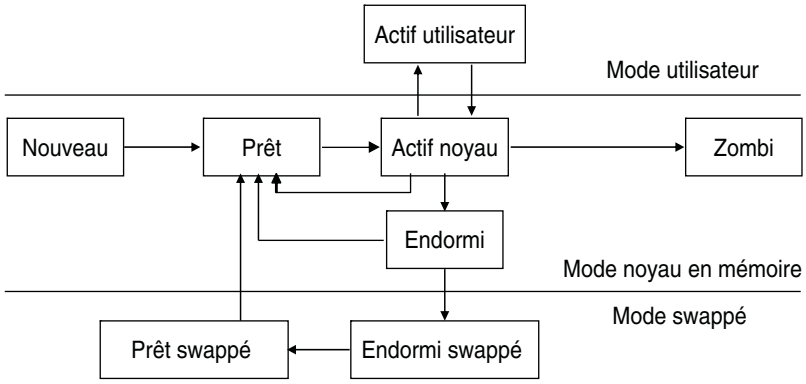


Figure 12.5 Diagramme d'états simplifié d'un processus Unix.

12.2 ORDONNANCEMENT SUR L'UNITÉ CENTRALE

La *fonction d'ordonnancement* gère le partage du processeur entre les différents processus en attente pour s'exécuter, c'est-à-dire entre les différents processus qui sont dans l'état prêt.

Dans le contexte d'une machine multiprogrammée, plusieurs processus sont présents en mémoire centrale. Imaginons la situation suivante à un instant t : le processus P1 est élu et s'exécute sur le processeur. Les processus P2 et P4 sont dans l'état bloqué car ils attentent tous les deux la fin d'une opération d'entrées-sorties avec le disque. Les processus P3, P5 et P6 quant à eux sont dans l'état prêt : ils pourraient s'exécuter mais ils ne le peuvent pas car le processeur est occupé par le processus P1. Lorsque le processus P1 quitte le processeur parce qu'il a terminé son exécution, les trois processus P3, P5 et P6 ont tous les trois le droit d'obtenir le processeur. Mais le processeur ne peut être alloué qu'à un seul processus à la fois : il faut donc choisir entre P3, P5 et P6. Par ailleurs, à un moment donné l'un des deux processus P2 ou P4, terminera son opération d'entrées-sorties. Ce processus passera alors dans l'état prêt et sera peut-être plus prioritaire que le processus courant. Il faudra alors préempter celui-ci. C'est le rôle de la fonction d'ordonnancement que de choisir un des trois processus P3, P5 ou P6 pour lui allouer le processeur ou de choisir d'arrêter le processus courant pour allouer le processeur à un nouveau processus prêt.

12.2.1 Ordonnancement préemptif et non préemptif

La figure 12.6 reprend le graphe d'états tel qu'il a été présenté sur la figure 12.1. Sur cette figure, nous nous intéressons plus particulièrement aux transitions pouvant exister entre l'état prêt (état d'attente du processeur) et l'état élu (état d'occupation du processeur). Le passage de l'état prêt vers l'état élu constitue l'opération d'*élection* : c'est l'allocation du processeur à un des processus prêts. Le passage de l'état élu vers l'état prêt a été ajouté par rapport à la figure 12.1 : il correspond à une *réquisi-*

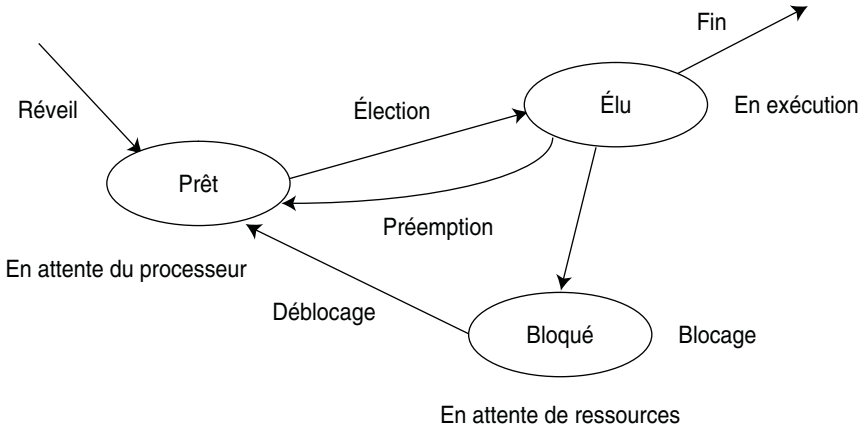


Figure 12.6 Opérations d'élection et de préemption.

tion du processeur, c'est-à-dire que le processeur est retiré au processus élu alors que celui-ci dispose de toutes les ressources nécessaires à la poursuite de son exécution. Cette réquisition porte le nom de *préemption*.

Ainsi selon si l'opération de réquisition est autorisée ou non, l'ordonnancement est qualifié d'*ordonnancement préemptif* ou *non préemptif* :

– si l'ordonnancement est non préemptif, la transition de l'état élu vers l'état prêt est interdite : un processus quitte le processeur s'il a terminé son exécution ou s'il se bloque ;
– si l'ordonnancement est préemptif, la transition de l'état élu vers l'état prêt est autorisée : un processus quitte le processeur s'il a terminé son exécution, s'il se bloque ou si le processeur est réquisitionné.

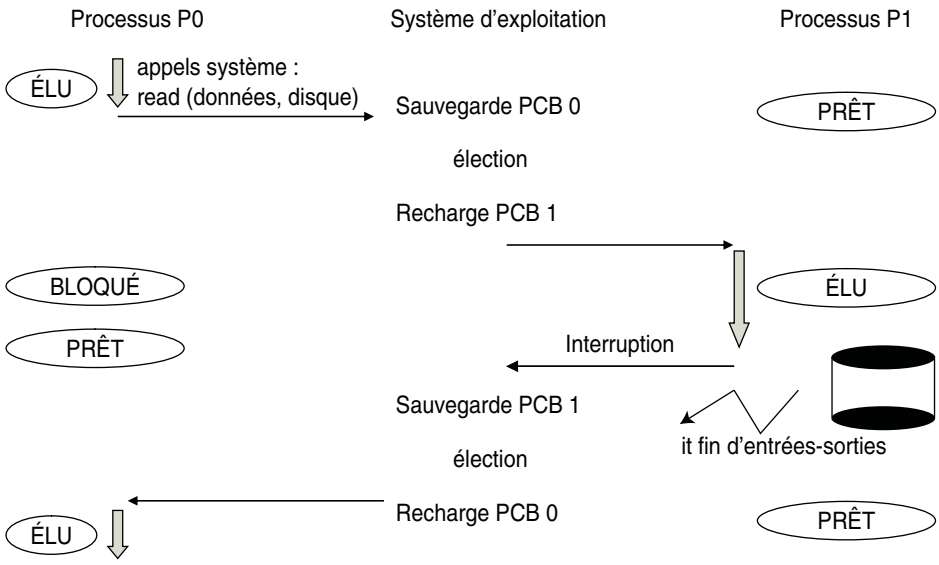


Figure 12.7 Déroulement des opérations d'ordonnancement.

La figure 12.7 schématise le déroulement des opérations d'ordonnancement préemptif avec deux processus P0 et P1. Initialement le processus P0 est élu et s'exécute. Le processus P1 est dans l'état prêt. Le processus P0 fait un appel système, par exemple `read (données, disque)` pour demander la lecture de données depuis un disque. Il y a commutation de contexte avec changement de protection pour aller exécuter le code de l'appel système (passage en mode superviseur) au cours duquel le processus P0 se bloque dans l'attente de la fin de l'opération d'entrées-sorties avec le disque. Il y a donc une opération d'élection et le processus P1 est élu : le contexte processeur associé au processus P0 est sauvegardé dans le PCB du processus P0 (PCB0) et le processeur est chargé avec le PCB du processus 1 (PCB1). Le processus P1 commence son exécution. Au cours de cette exécution, l'opération d'entrées-sorties au bénéfice du processus P0 se termine et le disque envoie donc une interruption pour signaler la

fin de cette opération d'entrées-sorties. Le processus P1 est dérouté pour aller exécuter le traitant d'interruption correspondant ; il y a commutation de contexte avec changement de protection (passage en mode superviseur). Au cours du traitant d'interruption, l'état du processus P0 est modifié et devient égal à prêt, puis une opération d'ordonnancement est lancée : le processus P0 est de nouveau élu. Le contexte processeur associé au processus P1 est sauvegardé dans le PCB du processus P1 (PCB1) et le processeur est chargé avec le PCB du processus 0 (PCB0).

Les opérations d'ordonnancement prennent place lors de tout changement d'états des processus.

12.2.2 Entités systèmes responsable de l'ordonnancement

Les processus prêts et les processus bloqués sont gérés dans deux files d'attentes distinctes qui chaînent leur PCB. Le module *ordonnanceur* (*scheduler*) trie la file des processus prêts de telle sorte que le prochain processus à élire soit toujours en tête de file. Le tri s'appuie sur un critère donné spécifié par la *politique d'ordonnancement*.

Dans un contexte multiprocesseur, le *répartiteur* (*dispatcher*) alloue un processeur parmi les processeurs libres à la tête de file de la liste des processus prêts et réalise donc l'opération d'élection. Dans un contexte monoprocesseur le rôle du répartiteur est très réduit puisqu'il n'a pas à choisir le processeur à allouer : il est alors souvent intégré dans l'ordonnanceur. La figure 12.8 représente les modules évoqués ici.

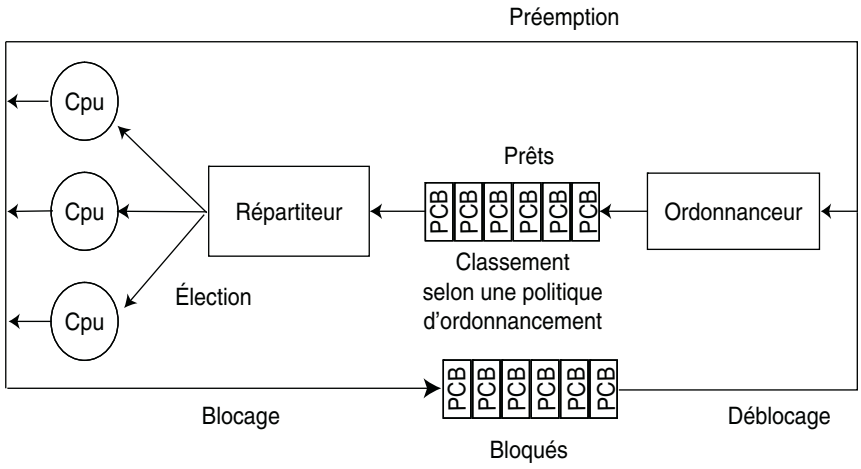


Figure 12.8 Ordonnanceur et répartiteur.

12.2.3 Politiques d'ordonnancement

La politique d'ordonnancement détermine quel sera le prochain processus élu. Selon si la préemption est autorisée ou non, la politique d'ordonnancement sera de type préemptive ou non.

Objectifs des politiques

Les objectifs à atteindre en matière d'ordonnancement diffèrent selon les types de systèmes considérés. Aussi, telle ou telle autre politique se révélera plus appropriée à un certain type de système plutôt qu'à un autre. Plus précisément :

- pour un système à traitements par lots, le but recherché est de maximiser le débit du processeur ou capacité de traitement, c'est-à-dire le nombre de travaux traités sur une certaine tranche de temps ;
- pour un système en temps partagé, le but recherché est de maximiser le taux d'occupation du processeur tout en minimisant le temps de réponse des processus ;
- pour un système temps réel, le but recherché est de respecter les contraintes temporelles des processus.

Différents critères sont utilisés pour mesurer les performances des politiques d'ordonnancement :

- le taux d'occupation du processeur, c'est-à-dire le rapport entre le temps d'utilisation du processeur par les processus sur le temps total écoulé ;
- la capacité de traitement du processeur, c'est-à-dire le nombre de processus exécuté sur un intervalle de temps donné ;
- le temps d'attente des processus, c'est-à-dire le temps passé par un processus dans la file d'attente des processus prêts ;
- le temps de réponse des processus, c'est-à-dire le temps écoulé entre la soumission du processus et sa fin d'exécution.

Les mesures sont généralement effectuées sur un temps moyen pour un ensemble de processus.

Présentation des politiques

Nous présentons à présent les politiques d'ordonnancement les plus courantes. Pour chacune des politiques évoquées, nous donnons un exemple sous forme d'un *diagramme de Gantt*. Un diagramme de Gantt permet de représenter l'assignation dans le temps du processeur aux processus. Le temps est représenté horizontalement, de manière croissante de la gauche vers la droite, à partir d'une origine arbitraire fixée à 0. Le temps est représenté par unité et pour chacune d'elles, figure le nom du processus affecté au processeur.

➤ Politique Premier Arrivé, Premier Servi

La première politique est la politique du « Premier Arrivé, Premier Servi ». Les processus sont élus selon l'ordre dans lequel ils arrivent dans la file d'attente des processus prêts. Il n'y a pas de réquisition, c'est-à-dire qu'un processus élu s'exécute soit jusqu'à ce qu'il soit terminé, soit jusqu'à ce qu'il se bloque de lui-même.

L'avantage de cette politique est sa simplicité. Son inconvénient majeur réside dans le fait que les processus de petit temps d'exécution sont pénalisés en terme de temps de réponse par les processus de grand temps d'exécution qui se trouvent avant eux dans la file d'attente des processus prêts. La figure 12.9 donne un exemple d'application de cette politique.

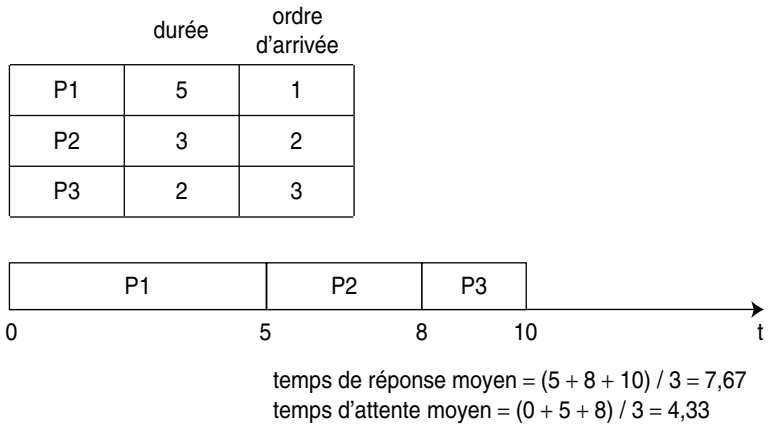


Figure 12.9 Politique « Premier Arrivé, Premier Servi ».

➤ Politique Plus Court d'Abord

Une deuxième politique d'ordonnancement est la politique du « Plus Court d'Abord ». Dans cette politique, l'ordre d'exécution des processus est fonction de leur temps d'exécution. Le processus de plus petit temps d'exécution est celui qui est ordonnancé en premier. La politique est sans réquisition.

La stratégie mise en œuvre dans cette politique remédie à l'inconvénient cité pour la politique précédente du « Premier Arrivé, Premier Servi ». On montre d'ailleurs que cette politique est optimale dans le sens où elle permet d'obtenir le temps de réponse moyen minimal pour un ensemble de processus donné. La difficulté majeure de cette politique réside cependant dans la connaissance a priori des temps d'exécution des processus. Cette connaissance n'est pas disponible dans un système interactif. C'est pourquoi cette politique est essentiellement mise en œuvre dans les systèmes de traitement par lots où les utilisateurs soumettent leurs travaux en fournissant une estimation du temps d'exécution nécessaire. La figure 12.10 donne un exemple d'application de cette politique.

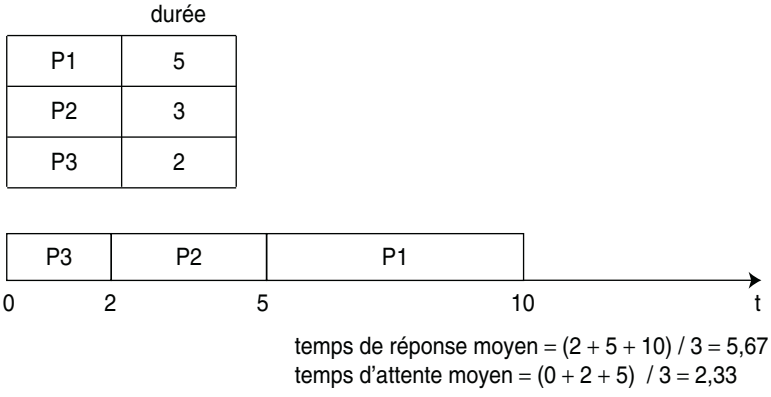


Figure 12.10 Politique « Plus Court d'Abord ».

➤ Politique par priorité

Une troisième politique très courante est la politique par priorité constante. Dans cette politique, chaque processus possède une *priorité*. La priorité est un nombre positif qui donne un rang à chaque processus dans l'ensemble des processus présents dans la machine. À un instant donné, le processus élu est le processus prêt de plus forte priorité.

Selon les systèmes d'exploitation, les numéros de priorités les plus bas représentent les valeurs de priorité les plus élevées ou vice-versa.

Cette politique se décline en deux versions selon si la réquisition est autorisée ou non. Si la réquisition est admise, alors le processus couramment élu est préempté dès qu'un processus plus prioritaire, donc possédant une priorité plus forte devient prêt.

Un inconvénient de cette politique est le risque de *famine* pour les processus de petite priorité si il y a de nombreux processus de haute priorité. Par famine, on entend le fait que les processus de plus forte priorité monopolisent le processeur au détriment des processus de plus faible priorité qui de ce fait ne peuvent pas s'exécuter (ils ont « faim » du processeur qu'ils n'obtiennent pas). On dit aussi qu'il y a *coalition* des processus de forte priorité contre les processus de faible priorité. La figure 12.11 donne un exemple d'application de cette politique.

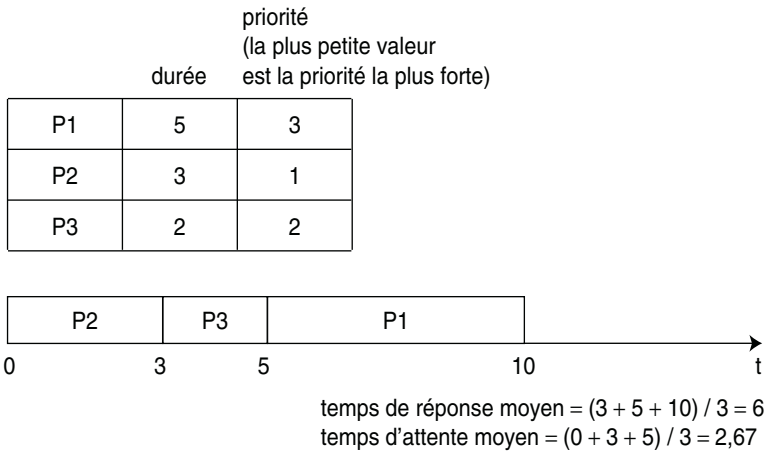


Figure 12.11 Politique par priorité.

Nous présentons sur la figure 12.12 un exemple d'application de cette politique à priorité dans le cadre d'un système temps réel. Les deux processus considérés ici sont périodiques, c'est-à-dire qu'ils s'exécutent à intervalles réguliers. Ainsi le processus P1 a une période égale à 5 : il devient donc prêt aux instants t = 0, t = 0 + 5 (5), t = 5 + 5 (10), t = 10 + 5 (15), et ainsi de suite.

La politique appliquée appelée *Rate Monotonic* est une politique classique du temps réel. Chaque processus reçoit une priorité fixe qui est inversement proportionnelle à sa période : le processus de plus petite période est donc le processus le plus

	durée	période
P1	4	5
P2	2	10

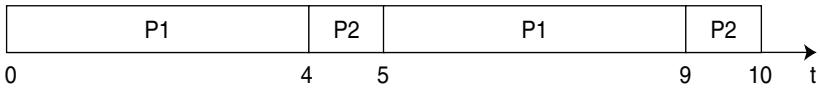


Figure 12.12 Politique Rate Monotonic.

prioritaire. Ici donc, le processus P1 est le plus prioritaire. A l'instant 0, les deux processus sont prêts : P1 est élu. Il s'exécute durant 4 unités de temps, puis c'est le processus P2 qui devient élu. À l'instant t = 5, le processus P1 redevient prêt : il préempte P2 car le processus P1 est plus prioritaire. Le processus P1 s'exécute donc entre les instants t = 5 et t = 9, puis le processus P2 est réélu et achève son exécution.

➤ Politique du tourniquet (round robin)

La politique par tourniquet est la politique mise en œuvre dans les systèmes dits en temps partagé. Dans cette politique, le temps est effectivement découpé en tranches nommées *quantums de temps*. La valeur du quantum peut varier selon les systèmes entre 10 à 100 ms.

Lorsqu'un processus est élu, il s'exécute au plus durant un quantum de temps. Si le processus n'a pas terminé son exécution à l'issue du quantum de temps, il est préempté et il réintègre la file des processus prêts mais en fin de file. Le processus en tête de file de la file des processus prêts est alors à son tour élu pour une durée égale à un quantum de temps maximum. La figure 12.13 donne un exemple d'application de cette politique.

La valeur du quantum constitue un facteur important de performance de la politique, dans la mesure où elle influe directement sur le nombre de commutations de

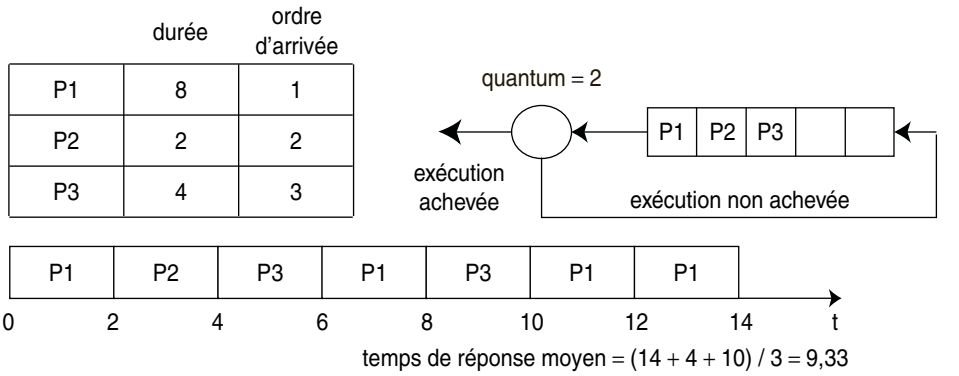


Figure 12.13 Politique du tourniquet.

contexte entre processus. En effet, plus la valeur du quantum sera faible, plus le nombre de commutations de contexte entre les processus à ordonnancer sera important et pourra de ce fait devenir non négligeable. Ainsi, si la valeur du quantum de temps est de 10 ms et que le temps de commutation de contexte est de 3 ms, alors ces opérations de commutations de contexte représentent pour une durée de 20 ms, 45 % d'attente. Au contraire, si le quantum est de 20 ms, alors les opérations de commutations de contexte ne représentent plus que 30 % d'attente.

12.2.4 Exemples

Les systèmes actuels combinent très souvent deux des politiques que nous avons vues : celles des priorités fixes et celles du tourniquet. La file des processus prêts est en fait divisée en autant de sous files qu'il existe de niveaux de priorité entre les processus. Chaque file de priorité Fi est gérée en tourniquet avec éventuellement un quantum Qi de temps propre. Pour remédier au problème de famine des processus de plus faible priorité, un mécanisme d'*extinction de priorité* peut être mis en œuvre. Dans ce cas, la priorité d'un processus baisse au cours de son exécution.

Si la politique est mise en œuvre sans extinction de priorité, un processus en fin de quantum regagne toujours la file d'attente dont il est issu. Sur la figure 12.14, les

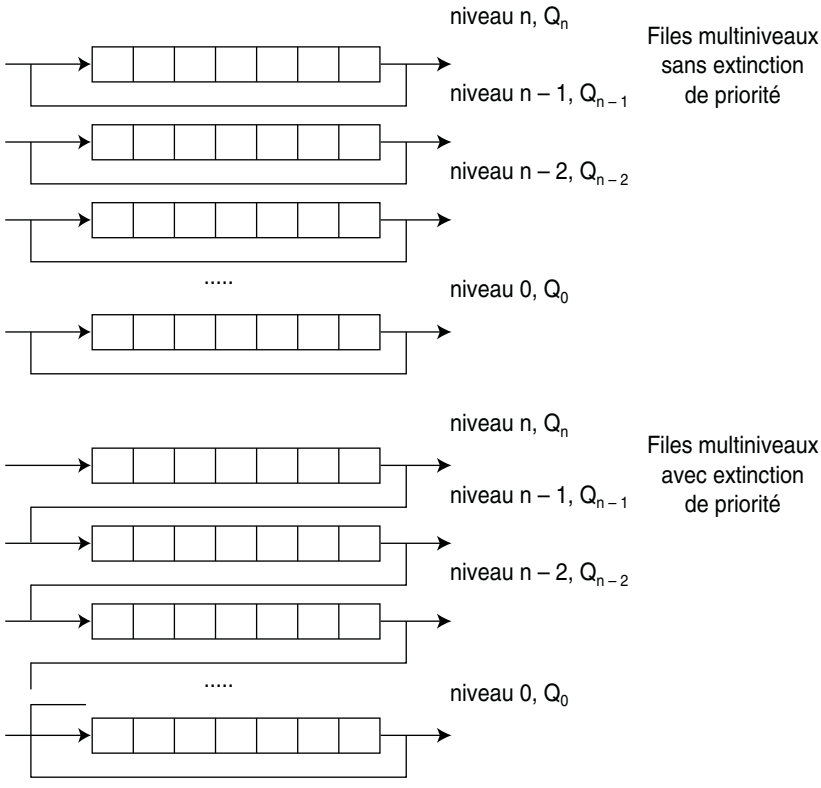


Figure 12.14 Politique sans extinction ou avec extinction de priorité.

processus de niveau n – 1 ne peuvent s'exécuter que lorsqu'il n'y aura plus aucun processus dans la file de niveau supérieure n. De même les processus de niveau n – 2 ne peuvent s'exécuter que lorsqu'il n'y a plus aucun processus dans les files de niveau supérieur n et n – 1. Ainsi, plus les processus sont dans les files de priorités inférieures, plus leurs chances de s'exécuter sont réduites.

Si la même politique est mise en œuvre mais avec extinction de priorité, alors lorsqu'un processus de niveau n a terminé son quantum, il ne regagne pas la file de niveau n, mais la file de niveau juste inférieure n – 1. Ainsi sa priorité baisse.

Ordonnancement sous Linux

➤ Principe de l'ordonnancement

Dans le système Linux, chaque processus est qualifié par une priorité. Par ailleurs, conformément à la norme pour les systèmes d'exploitation ouverts POSIX 1003.1, le système Linux offre trois politiques d'ordonnancement différentes, SCHED_FIFO, SCHED_RR et SCHED_OTHER. Un processus créé est attaché à l'une des politiques par un appel à la fonction système `sched_setscheduler()`.

– La politique SCHED_FIFO élit à tout instant le processus de plus forte priorité parmi les processus attachés à cette classe.
– La politique SCHED_RR est une politique de type tourniquet entre processus de même priorité.
– La politique SCHED_OTHER implémente une politique à extinction de priorité, qui est la politique d'ordonnancement classique du système Unix.

Les processus attachés aux politiques SCHED_FIFO et SCHED_RR sont plus prioritaires que les processus attachés à la politique SCHED_OTHER, c'est-à-dire que les processus SCHED_OTHER ne s'exécutent que lorsqu'il n'y a plus aucun processus SCHED_FIFO ou SCHED_RR à exécuter.

➤ Appels systèmes liés à l'ordonnancement

Le système Linux, conformément à la norme POSIX 1003.1, offre un ensemble de primitives systèmes pour la gestion de l'ordonnancement, plus précisément :

– les primitives `sched_setscheduler()` et `sched_getscheduler()` permettent d'attribuer une des trois politiques d'ordonnancement vues précédemment à un processus ou de connaître la politique affectée à un processus ;
– les primitives `setpriority()` et `getpriority()` permettent de modifier la priorité d'un processus ou de connaître la priorité affectée à un processus ;
– la primitive `sched_rr_get_interval()` permet de connaître le quantum de temps attribué à un processus de type SCHED_RR.

Ordonnancement sous MVS

Dans le système MVS, les travaux à ordonnancer sont également qualifiés par une priorité. Ces travaux de deux types différents sont décrits au sein du système par des blocs de contrôle : d'une part les TCB (*Task Control Block*) représentent les processus dans un espace adresse tels que les processus utilisateurs ; d'autre part, les SRB (*Service*

Request Block) décrivent les demandes d'exécutions de routines systèmes dans un espace adresse système. Les TCB et SRB sont attachés à un espace adresse auquel est également associée une priorité et qui est également décrit par un bloc de contrôle, l'ASCB (*Address Space Control Block*).

Au niveau de l'ordonnanceur, une première file d'attente est constituée par le chaînage de tous les ASCB présents en mémoire, chaînés par ordre de priorité croissant. Ensuite, pour chaque ASCB, il existe une file de tous les TCB et SRB attachés à cet espace adresse, également chaînés par priorité croissante. Les SRB sont prioritaires par rapport au TCB.

À un instant donné, l'ordonnanceur choisit d'allouer le processeur au SRB ou à défaut au TCB le plus prioritaire de l'espace adresse de plus haute priorité (figure 12.15).

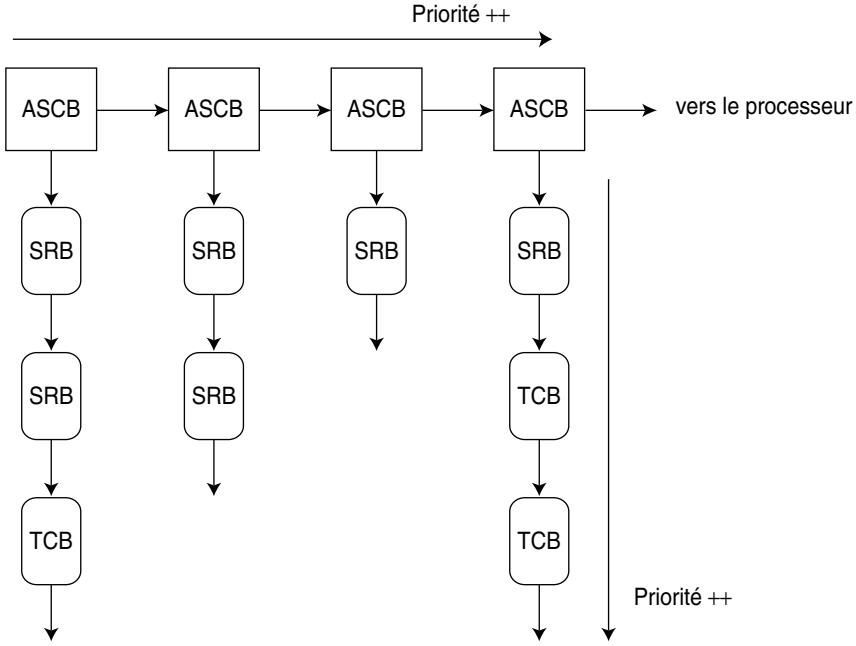


Figure 12.15 Chaînage des ASCB, TCB et SRB au sein de l'ordonnanceur MVS.

12.3 SYNCHRONISATION ET COMMUNICATION ENTRE PROCESSUS

Dans un système multiprocessus, l'ordonnanceur alloue le processeur à chaque processus selon un algorithme d'ordonnancement : la politique choisie conditionne l'ordre d'exécution des processus et très souvent, les exécutions des processus s'entrelacent les unes avec les autres. Chaque processus dispose d'un espace d'adressage propre et indépendant, protégé par rapport aux autres processus. Malgré tout, les processus peuvent avoir besoin de communiquer entre eux pour échanger des données.

Cet échange de données peut se faire par le biais d'une zone de mémoire partagée, par le biais d'un fichier ou encore en utilisant les outils de communication offerts par le système d'exploitation. Dans tous ces cas, les processus ne sont plus indépendants et ils effectuent des accès concurrents aux ressources logicielles que sont la mémoire partagée, le fichier ou encore les outils de communication. Plus généralement, une ressource désigne toute entité dont a besoin un processus pour s'exécuter. La ressource peut être matérielle comme le processeur ou un périphérique ou elle peut être logicielle comme une variable.

Une ressource est caractérisée par un état qui définit si la ressource est libre ou occupée et par son nombre de points d'accès, c'est-à-dire le nombre de processus pouvant l'utiliser en même temps. Notamment, on distinguera la notion de *ressource critique* qui correspond à une ressource ne pouvant être utilisée que par un seul processus à la fois. Un processeur par exemple correspond à une ressource critique ; en effet, il n'est pas possible d'allouer celui-ci à deux processus simultanément. Une imprimante est un autre exemple d'une telle ressource.

L'utilisation d'une ressource par un processus s'effectue en trois étapes. La première étape correspond à l'étape d'allocation de la ressource au processus qui consiste à donner la ressource au processus qui la demande. Une fois que le processus a pu obtenir la ressource, il l'utilise durant un certain temps (deuxième étape) puis il rend la ressource : c'est la dernière étape de restitution de la ressource. Les phases d'allocation et de restitution d'une ressource doivent assurer que la ressource est utilisée conformément à son nombre de points d'accès. Par ailleurs, l'étape d'allocation de la ressource peut se révéler bloquante pour le processus qui l'effectue si tous les points d'accès de la ressource demandée sont occupés. Au contraire, l'étape de restitution d'une ressource par un processus peut entraîner le déblocage d'un autre processus en attente d'accès à cette ressource.

Plusieurs schémas de synchronisation entre processus ont été définis afin de garantir une bonne utilisation des ressources par les processus et d'une manière plus générale une communication entre processus cohérente et sans perte de données. Ces schémas sont : l'exclusion mutuelle, l'allocation de ressources, le producteur-consommateur et les lecteurs-rédacteurs.

12.3.1 L'exclusion mutuelle

Un exemple simple

Nous allons mettre en lumière ce premier problème de synchronisation à l'aide d'un exemple simple. Pour cela, considérons donc le petit programme de réservation de place (dans un avion, un train…) donné ci-après.

```
Réservation :
si nbplace > 0
alors
    réserver une place;
    nbplace = nbplace – 1;
fsi
```

Le programme `Réservation` peut être exécuté par plusieurs processus à la fois (autrement dit, le programme est réentrant). La variable `nbplace`, qui représente le nombre de places restantes dans l'avion par exemple, est ici une variable d'état du système (de réservation).

On considère l'exécution de deux processus Client_1 et Client_2, alors que `nbplace` est égale à 1. Client_1 s'exécute en premier, et entame une réservation. Maintenant, l'exécution de Client_1 est commutée par l'ordonnanceur juste après le test de la valeur de la variable `nbplace` (`nbplace` = 1). Client_2 s'exécute à son tour, teste `nbplace` qu'il trouve également égale à 1 et donc effectue une réservation en décrémentant d'une unité la variable `nbplace`. `nbplace` devient égale à 0. Maintenant que le processus Client_2 a terminé son exécution, Client_1 reprend la main. Comme Client_1 avait trouvé la variable `nbplace` comme étant égale à 1 juste avant d'être commuté, il continue son exécution en décrémentant à son tour `nbplace`. De ce fait, `nbplace` devient égale à – 1 ce qui est incohérent ! Une même place a été allouée à deux clients différents !

Ressource critique, section critique et exclusion mutuelle

La variable `nbplace` doit être accédée par un seul processus à la fois pour que sa valeur reste cohérente : ici en l'occurrence le processus Client_1 qui a commencé la réservation en premier doit terminer son opération avant que le processus Client_2 puisse commencer à s'exécuter. Ainsi, Client_2 trouvera la variable `nbplace` égale à 0 lors du test et n'effectuera pas de réservation. `nbplace` constitue donc une ressource critique.

Le code d'utilisation d'une ressource critique est appelé *section critique*. La section critique doit au moins offrir la propriété essentielle de l'*exclusion mutuelle* qui garantit que la ressource critique manipulée durant la section critique ne peut effectivement être utilisée que par un seul processus à la fois. Pour ce faire, la section critique est précédée par un prélude et est suivie par un postlude qui assurent cette propriété d'exclusion mutuelle.

Pour garantir l'exclusion mutuelle, il faut donc entourer l'utilisation de la variable `nbplace` d'un prélude et d'un postlude. Le prélude prend la forme d'une protection qui empêche un processus de manipuler la variable `nbplace` si un autre processus le fait déjà. Ainsi le processus Client_2 est mis en attente dès qu'il cherche à accéder à la variable `nbplace` déjà possédée par le processus Client_1. Le postlude prend la forme d'une fin de protection qui libère la ressource `nbplace` et la rend accessible au processus Client_2.

Réalisation d'une section critique à l'aide des interruptions matérielles

Nous rappelons que le mécanisme sous-jacent à l'ordonnancement des processus peut être la survenue d'une interruption horloge. Aussi une solution pour réaliser l'exclusion mutuelle est de masquer les interruptions dans le prélude et de les démasquer dans le postlude. Ainsi les interruptions sont masquées dès qu'un processus accède à la ressource critique et aucun événement susceptible d'activer un autre processus ne peut être pris en compte.

```
Réservation :
masquer_it;   — prélude de section critique
Si nbplace > 0
alors
    réserver une place;
    nbplace = nbplace – 1;
fsi
démasquer_it; — postlude de section critique
```

Cependant, cette solution est moyennement satisfaisante car elle empêche l'exécution de tous les processus y compris ceux ne désirant pas accéder à la ressource critique. De plus, le masquage et le démasquage des interruptions sont des opérations réservées au mode superviseur et ne sont donc pas accessibles pour les processus utilisateurs. Ce mode de réalisation d'une section critique est donc uniquement utilisé dans des parties de code sensibles du système d'exploitation, comme par exemple, la manipulation des files d'attente de l'ordonnanceur.

L'outil sémaphore

Une autre solution est d'utiliser un outil de synchronisation offert par le système d'exploitation : les *sémaphores*.

➤ Présentation de l'outil sémaphore

Un sémaphore `sem` est une structure système composée d'une file d'attente `L` de processus et d'un compteur `K`, appelé niveau du sémaphore et contenant initialement une valeur `val`. Cette structure ne peut être manipulée que par trois opérations `P(sem)`, `V(sem)` et `init(sem, val)`. Ces trois opérations sont des opérations indivisibles c'est-à-dire que l'exécution de ces opérations s'effectue interruptions masquées et ne peut pas être interrompue. Un outil sémaphore peut être assimilé à un distributeur de jetons dont le nombre initial est fixé par l'opération `init`. L'acquisition d'un jeton par un processus donne le droit à ce processus de poursuivre son exécution. Sinon, le processus est bloqué.

L'opération init

L'opération `init` a pour but d'initialiser le sémaphore, c'est-à-dire qu'elle met à vide la file d'attente `L` et initialise avec la valeur val le compteur `K`. On définit ainsi le nombre de jetons initiaux dans le sémaphore.

```
init(sémaphore sem, entier val)
debut
masquer_it;
    sem.K  : = val;
    sem.L  : = vide;
démasquer_it;
fin
```

L'opération P (sem)

L'opération P(sem) attribue un jeton au processus appelant s'il en reste au moins un et sinon bloque le processus dans la file sem.L. L'opération P est donc une opération éventuellement bloquante pour le processus élu qui l'effectue. Dans le cas du blocage, il y a réordonnancement et un nouveau processus prêt est élu. Concrètement, le compteur K du sémaphore est décrémenté d'une unité. Si la valeur du compteur devient négative, le processus est bloqué.

```
P(sémaphore sem)
début
masquer_it;
sem.K : = sem.K – 1;
si sem.K < 0
alors
    ajouter ce processus à sem.L;
    bloquer ce processus;
fsi
démasquer_it;
fin
```

L'opération V (sem)

L'opération V(sem) a pour but de rendre un jeton au sémaphore. De plus, si il y a au moins un processus bloqué dans la file d'attente L du sémaphore, un processus est réveillé. La gestion des réveils s'effectue généralement en mode FIFO, c'est-à-dire que c'est le processus le plus anciennement endormi qui est réveillé. L'opération V est une opération qui n'est jamais bloquante pour le processus qui l'effectue.

```
V(sémaphore sem)
début
masquer_it;
sem.K : = sem.K + 1;
si sem.K < = 0    — il y a au moins un processus dans la file L
alors
    sortir un processus de sem.L;
    réveiller ce processus; — passage à l'état prêt
fsi
démasquer_it
fin
```

➤ Réalisation d'une section critique à l'aide des sémaphores

La réalisation d'une section critique à l'aide de l'outil sémaphore s'effectue en utilisant un sémaphore que nous appelons mutex, dont le compteur K est initialisé à 1.

Le prélude de la section critique correspond à une opération P(mutex).

Le postlude de la section critique correspond à une opération V(ùutex).

```
init (mutex, 1);
Réservation :
P(mutex);     — prelude de section critique
si nbplace > 0
alors
    reserver une place;
    nbplace = nbplace – 1;
fsi
V(mutex);     — postlude de section critique
```

12.3.2 Le schéma de l'allocation de ressources

Présentation du schéma

Le schéma d'allocation de n exemplaires de ressources critiques à un ensemble de processus est une généralisation du schéma précédent, dans lequel on considère à présent que l'on a n sections critiques. Le sémaphore `res` d'allocation de ressources est initialisé au nombre d'exemplaires de ressources initialement disponibles, soit n, c'est-à-dire que l'on fait correspondre un jeton par exemplaire de ressources. Pour un processus, acquérir un exemplaire de ressource est alors synonyme d'acquérir le jeton correspondant, soit une opération `P(res)`. Rendre la ressource est synonyme de rendre le jeton associé, soit une opération `V(res)`.

```
init (res, n);
Allocation : P(res);
    Utilisation de la ressource res
Restitution : V(res);
```

Interblocage

Considérons à présent la situation suivante pour laquelle deux processus P1 et P2 utilisent tous deux deux ressources critiques R1 et R2 selon le code suivant. Les deux ressources R1 et R2 sont gérées par le biais de deux sémaphores SR1 et SR2 initialisés à 1.

```
Processus P1                    Processus P2
début                           début
P(SR1);                         P(SR2);
P(SR2);                         P(SR1);
Utilisation de R1 et R2;        Utilisation de R1 et R2;
V(SR2);                         V(SR1);
V(SR1);                         V(SR2);
fin                             fin
```

Initialement, les deux ressources R1 et R2 sont libres. Le processus P1 commence son exécution, demande à accéder à la ressource R1 en faisant l'opération P(SR1) et obtient R1 puisque la ressource est libre. Maintenant, le processus P1 est commuté par l'ordonnanceur au bénéfice du processus P2. Le processus P2 commence donc à

s'exécuter, demande à accéder à la ressource R2 en faisant l'opération P(SR2) et obtient R2 puisque la ressource est libre. Le processus P2 poursuit son exécution et demande à présent à accéder à la ressource R1 en effectuant l'opération P(SR1). Cette fois, le processus P2 est stoppé puisque R1 a été allouée au processus P1. Le processus P2 passe donc dans l'état bloqué ; l'ordonnanceur est invoqué et celui-ci réélit le processus P1. Le processus P1 reprend son exécution et demande à présent à accéder à la ressource R2 en effectuant l'opération P(SR2). Le processus P1 est également stoppé puisque R2 a été allouée au processus P2.

La situation dans laquelle les processus P1 et P2 se trouvent maintenant est donc la suivante : le processus P1 possède la ressource R1 et attend la ressource R2 détenue par le processus P2. Pour que le processus P1 puisse reprendre son exécution, il faut que le processus P2 libère la ressource R2 en effectuant l'opération V(SR2). Mais le processus P2 ne peut lui-même pas poursuivre son exécution puisqu'il est bloqué en attente de la ressource R1 possédée par le processus P1. On voit donc clairement que les deux processus P1 et P2 sont bloqués dans l'attente l'un de l'autre et que rien ne peut les sortir de cette attente.

Une telle situation est qualifiée d'interblocage. Elle correspond à une situation où au moins deux processus, possédant déjà un ensemble de ressources, sont en attente infinie les uns des autres pour l'acquisition d'autres ressources. Une telle situation, lorsqu'elle se produit, ne peut être résolue que par la destruction des processus impliqués dans l'interblocage. Une façon d'éviter l'occurrence d'une telle situation est de veiller à ce que les processus utilisant les mêmes ressources demandent celles-ci dans le même ordre. Ici, donc les opérations P(SR2) et P(SR1) pour le processus P2 doivent être inversées.

12.3.3 Le schéma lecteurs-rédacteurs

Dans ce problème, on considère la situation où un fichier ou une zone de mémoire commune est accédée simultanément en lecture et en écriture. Il y a donc deux types de processus, d'une part des processus lecteurs et d'autre part des processus rédacteurs.

Dans ce schéma, deux objectifs doivent être atteints. D'un côté, le contenu du fichier ou de la zone de mémoire commune doit évidemment rester cohérent, donc les écritures ne doivent pas avoir lieu en même temps. Par ailleurs, on souhaite que les lecteurs lisent toujours une information stable, c'est-à-dire que les lecteurs ne doivent pas lire une information en cours de modification.

Pour parvenir à réaliser ces deux objectifs, il faut donc, à un instant donné, sur la zone de mémoire commune ou sur le fichier partagé :

– soit une seule écriture en cours ;
– soit une ou plusieurs lectures en cours, les lectures simultanées ne gênant pas puisqu'une lecture ne modifie pas le contenu de l'information.

Le rédacteur

Un processus rédacteur exclut donc les autres rédacteurs ainsi que tous les lecteurs. Autrement dit, un rédacteur accède donc toujours seul au fichier ou à la zone de

mémoire commune, c'est-à-dire qu'il effectue des accès en exclusion mutuelle des autres rédacteurs et des lecteurs. L'accès du processus rédacteur peut donc être géré selon le schéma d'exclusion mutuelle vu précédemment, en utilisant un sémaphore appelé ici `acces` et initialisé à 1.

```
REDACTEUR
init (acces, 1);
m'assurer que l'accès au fichier est libre :  P(acces);
                                          entrer en écriture
libérer l'accès au fichier :                  V(acces);
```

Le lecteur

Un lecteur exclut les rédacteurs mais pas les autres lecteurs. Seul le premier lecteur doit donc s'assurer qu'il n'y a pas d'accès en écriture en cours. Par ailleurs, seul le dernier lecteur doit libérer la ressource et réveiller un éventuel rédacteur.

Il faut en conséquence compter le nombre de lecteurs qui accèdent au fichier ou à la zone de mémoire partagée. On utilise pour cela une variable `nl`, initialisée à 0. Cette variable `nl` est accédée en concurrence par tous les lecteurs qui vont soit incrémenter cette variable (un lecteur de plus), soit la décrémenter (un lecteur de moins). Pour que le contenu de la variable reste cohérent, il faut que `nl` soit accédée en exclusion mutuelle par tous les lecteurs. L'accès à la variable est donc gardé par un sémaphore `mutex` initialisé à 1.

```
LECTEUR
— accès à nl pour compter un lecteur de plus
P(mutex);
nl : = nl + 1;
si (nl = 1)
alors
— je suis le premier lecteur; je me place en exclusion mutuelle
➥ d'un rédacteur éventuel
    P(acces);
fsi
V(Mutex);
accès en lecture
— accès à nl pour compter un lecteur de moins
P(mutex);
nl : = nl – 1;
si (nl = 0)
alors
— je suis le dernier lecteur; je libère la ressource et je réveille
➥ d'un rédacteur éventuel en attente
    V(acces);
fsi
V(mutex);
```

Utilisation de ce schéma au niveau du système d'exploitation

Certains systèmes d'exploitation offrent au niveau du système de gestion de fichiers, un paramètre permettant de configurer l'ouverture d'un fichier, soit en mode partagé, soit en mode exclusif. Lorsqu'un processus ouvre un fichier en mode partagé, alors il est mis en attente pour accéder à ce fichier seulement si un autre processus a ouvert ce même fichier en mode exclusif. Ce processus se comporte comme un lecteur. Lorsqu'un processus ouvre un fichier en mode exclusif, alors il est mis en attente pour accéder à ce fichier seulement si un autre processus a ouvert ce même fichier en mode exclusif ou en mode partagé. Autrement dit, ce processus accède seul au fichier, tout comme un rédacteur.

Exemple

Le système MVS permet de configurer l'ouverture d'un fichier en mode exclusif (OLD) ou en mode partagé (SHR). La ligne suivante, extraite d'un JCL, demande l'ouverture en mode exclusif du fichier physique `MDP1.ARTEMIS.V70.PRO003£.MULTILIB`.

```
000011 //A70DLMA DD DSN = MDP1.ARTEMIS.V70.PRO003£.MULTILIB, DISP = OLD
```

Un accès concurrent à ce fichier réalisé par un autre job entraîne l'affichage du message suivant :

```
DSLIST – Data Sets Matching MDP1.ARTEMIS.V70.PRO003£.  Data set in use
```

indiquant que l'accès au fichier est verrouillé.

12.3.4 Le schéma producteur-consommateur

Présentation du problème

On considère maintenant deux processus communiquant par un tampon de n cases. D'un côté, un processus producteur produit des messages qu'il dépose dans la case du tampon pointée par l'index de dépôt `i`. De l'autre côté, un processus consommateur prélève les messages déposés par le processus producteur dans la case pointée par l'index de retrait `j`. Le tampon est géré selon un mode FIFO circulaire en consommation et en production, c'est-à-dire que :

– le producteur dépose les messages depuis la case 0 jusqu'à la case n – 1, puis revient à la case 0 ;
– le consommateur prélève les messages depuis la case 0 jusqu'à la case n – 1, puis revient à la case 0.

Le processus producteur et le processus consommateur sont totalement indépendants dans le sens où ils s'exécutent chacun à leur propre vitesse. Dans ce cas, pour qu'aucun message ne soit perdu, les trois règles suivantes doivent être respectées :

– le producteur ne doit pas produire si le tampon est plein ;
– le consommateur ne doit pas faire de retrait si le tampon est vide ;
– le producteur et le consommateur ne doivent jamais travailler dans une même case.

Une solution au problème

Dans ce problème, deux types de ressources distinctes peuvent être mis en avant :

– le producteur consomme des cases vides et fournit des cases pleines ;
– le consommateur consomme des cases pleines et fournit des cases vides.

Il y a donc des ressources cases vides et des ressources cases pleines. Le problème peut maintenant se rapprocher d'un problème d'allocation de ressources critiques tel que nous l'avons vu précédemment. On associe donc un sémaphore à chacune des ressources identifiées, initialisé au nombre de cases respectivement vides ou pleines initialement disponibles (n et 0). Soient donc les deux sémaphores `vide` initialisé à n et `plein` initialisé à 0.

Le producteur s'alloue une case vide par une opération `P(vide)`, remplie cette case vide et de ce fait génère une case pleine qu'il signale par une opération `V(plein)`. Cette opération réveille éventuellement le consommateur en attente d'une case pleine.

```
PRODUCTEUR
index i de dépot : = 0
P(vide);
déposer le message dans tampon(i);
i : = i + 1 mod n;
V(plein);
```

Le consommateur s'alloue une case pleine par une opération `P(plein)`, vide cette case pleine et de ce fait génère une case vide qu'il signale par une opération `V(vide)`. Cette opération réveille éventuellement le producteur en attente d'une case vide.

```
CONSOMMATEUR
index j de retrait : = 0
P(plein);
retirer le message de tampon(j);
j : = j + 1 mod n;
V(vide);
```

Utilisation de ce schéma au niveau du système d'exploitation

Les systèmes d'exploitation offrent directement des outils de communication mettant en œuvre au niveau des primitives d'accès à ces outils, le schéma producteur-consommateur.

Le système Linux par exemple offre deux outils de ce type :

– d'une part, les tubes (*pipe*) qui permettent exclusivement la communication entre processus qui sont issus d'un même processus père, créateur du tube ;
– d'autre part, les files de messages (*Messages queues* – MSQ) qui permettent à n'importe quel processus connaissant l'identifiant de la file de communiquer avec d'autres processus.

12.4 CONCLUSION

Un processus est un programme en cours d'exécution auquel est associé un environnement processeur (CO, PSW, RSP, registres généraux) et un environnement mémoire appelés contexte du processus.

L'ordonnanceur est un programme système dont le rôle est d'allouer le processeur à un processus prêt. La politique d'ordonnancement détermine le choix d'un processus à élire parmi tous ceux qui sont prêts. Les exécutions de processus ne sont pas indépendantes : les processus peuvent vouloir communiquer et accéder de manière concurrente à des ressources. Le sémaphore S est un outil système de synchronisation assimilable à un distributeur de jetons et manipulable par seulement trois opérations atomiques : `P(S)`, `V(S)` et `init(S)`.

Il existe plusieurs schémas typiques de synchronisation à partir desquels sont élaborés des outils de communication entre processus : l'exclusion mutuelle, l'allocation de ressources, les lecteurs-rédacteurs, le producteur-consommateur.

Chapitre **13**

Gestion de la mémoire centrale

Nous commençons à présent l'étude de la gestion de la mémoire centrale. Dans le cadre d'un système multiprogrammé, plusieurs programmes sont chargés dans la mémoire centrale, qui doit donc être partagée entre ceux-ci. Ce partage engendre trois grands problèmes à résoudre : tout d'abord, il faut définir un espace d'adressage séparé pour chaque processus, allouer de la place en mémoire centrale pour le code et les données de ce processus et protéger cet espace d'adressage vis-à-vis des accès des autres processus. Par ailleurs, le nombre de processus nécessaires à une occupation optimale du processeur va très vite rendre insuffisante la quantité de mémoire physique disponible sur la machine. Une gestion de la mémoire visant à ne charger que les parties utiles à un instant donné de chaque processus est donc mise en place pour palier ce problème : c'est le concept de *mémoire virtuelle* que nous expliquons dans la deuxième partie de ce chapitre. Enfin, nous terminons en abordant les notions de s*wapping*.

13.1 MÉMOIRE PHYSIQUE ET MÉMOIRE LOGIQUE

La chaîne de production de programmes telle que nous l'avons étudiée en première partie de cet ouvrage produit un programme exécutable relogeable, c'est-à-dire dont toutes les adresses sont calculées à partir d'une origine fixée à 0.

Lors d'une opération de chargement dynamique (cf. paragraphe 3.3.1), les adresses du programme exécutable relogeable ne sont pas translatées de la valeur de l'adresse

d'implantation en mémoire centrale. Cette translation s'effectue seulement au moment de l'exécution, c'est-à-dire au moment où le processeur veut accéder à l'emplacement de la mémoire physique désigné par l'adresse manipulée. À ce moment-là, l'adresse relogeable est additionnée avec la valeur de l'adresse d'implantation maintenue dans un registre dit registre de translation.

L'ensemble des adresses du programme exécutable, générées par le processeur au moment de l'exécution des instructions, est appelé *espace d'adresses logiques ou virtuelles*. L'ensemble des adresses physiques réellement occupées par le programme exécutable suite au chargement en mémoire centrale est appelé *espace d'adresses physiques*.

Pour que le processeur puisse accéder à la mémoire centrale, il est nécessaire de *convertir* les adresses logiques générées par le processeur en adresses physiques, placées sur le bus et présentées au dispositif de sélection de la mémoire physique. Cette conversion est réalisée par un dispositif matériel déjà évoqué au chapitre 8, la MMU (*Memory Management Unit*). Dans l'exemple pris ici et illustré par la figure 13.1, la MMU contient le registre de translation. Ce registre de translation est toujours chargé avec l'adresse d'implantation du programme exécutable correspondant au processus couramment actif. Nous verrons dans la suite de ce chapitre qu'il existe d'autres schémas de conversion, notamment les schémas de conversions liés à la segmentation et à la pagination.

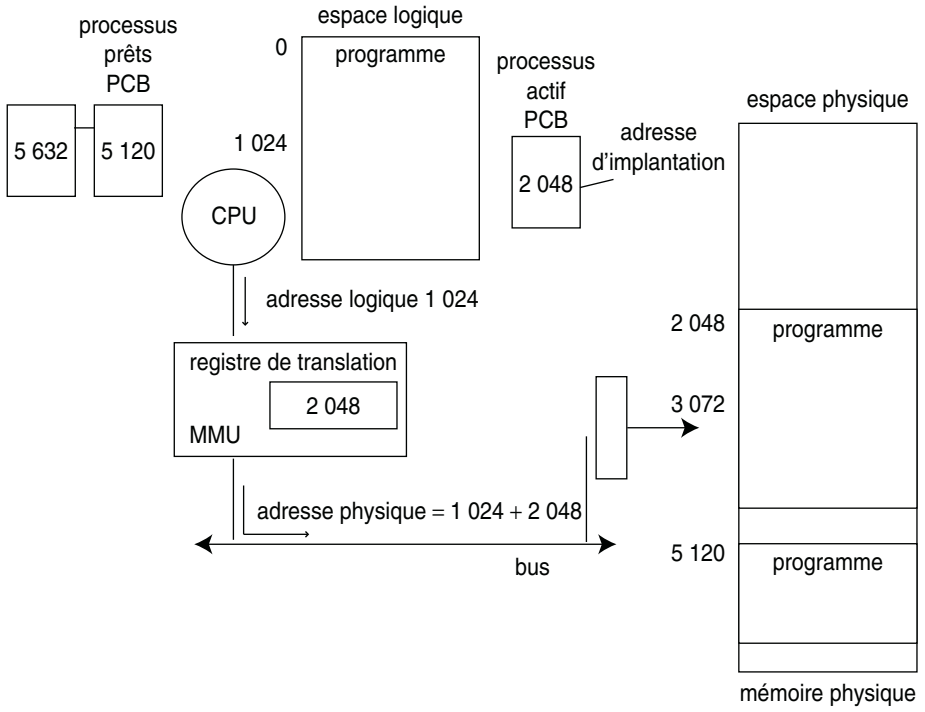


Figure 13.1 Espace d'adresses logiques et espace d'adresses physiques.

13.2 ALLOCATION DE LA MÉMOIRE PHYSIQUE

Dans un système multiprogrammé, la mémoire centrale abrite d'une part le code et les données du système d'exploitation, d'autre part le code et les données des programmes utilisateurs. La mémoire est donc divisée en deux grandes zones, celle réservée au système d'exploitation qui est généralement placé en mémoire haute (depuis les adresses 0), en tous les cas toujours au voisinage du vecteur d'interruptions, d'autre part celle réservée aux programmes des utilisateurs. Ainsi sous le système Linux, 1 Go de mémoire centrale est réservée au système d'exploitation et 3 Go sont dédiés aux utilisateurs.

Lorsqu'un utilisateur demande l'exécution d'un programme, le chargeur place en mémoire centrale dans la zone réservée aux programmes utilisateurs le code et les données de ce programme. Pour ce faire, le chargeur doit trouver une place libre suffisamment grande dans cette zone. Lorsque le processus correspondant a terminé son exécution, la mémoire centrale doit être libérée. L'attribution de place mémoire aux programmes utilisateurs et la libération de celle-ci en fin d'exécution des processus s'appuient sur différentes stratégies d'allocation et de libération mémoire.

Les stratégies d'allocation / libération de la mémoire centrale peuvent être divisées en deux grandes familles :

– pour la première famille, un programme est un ensemble de mots contigus insécable. L'espace d'adressage du processus est linéaire. On trouve ici les méthodes d'allocations en *partitions variables* que nous allons étudier en premier;
– pour la seconde famille, un programme est un ensemble de mots contigus sécable, c'est-à-dire que le programme peut être divisé en plus petits morceaux, chaque morceau étant lui-même un ensemble de mots contigus. Chaque morceau peut alors être alloué de manière indépendante. On trouve ici les mécanismes de *segmentation* et de *pagination*.

13.2.1 Allocation contiguë de la mémoire physique

Notion de partitions variables

Dans cette méthode d'allocation, le programme est considéré comme un espace d'adressage insécable. La mémoire physique est découpée en zones disjointes de taille variable, adaptables à la taille des programmes, qui sont dénommées *partitions*.

Initialement, la zone réservée aux programmes utilisateurs est vide et constitue une unique *zone libre* (figure 13.2). Au fur et à mesure des chargements de programmes, la zone libre se réduit et à l'instant t, elle n'occupe plus qu'une fraction de la mémoire centrale. Lorsque les exécutions des processus se terminent, la mémoire est libérée : il se crée alors pour chaque zone libérée, une nouvelle zone libre. Finalement, la mémoire centrale se retrouve constituée d'une ensemble de zones allouées et de zones libres réparties dans toute la mémoire.

Les zones libres sont souvent organisées en une liste circulaire doublement chaînée de zones libres. Chaque zone libre ZL est caractérisée par une adresse d'implantation (adresse(ZL)) en mémoire centrale et une taille en octet (taille(ZL)).

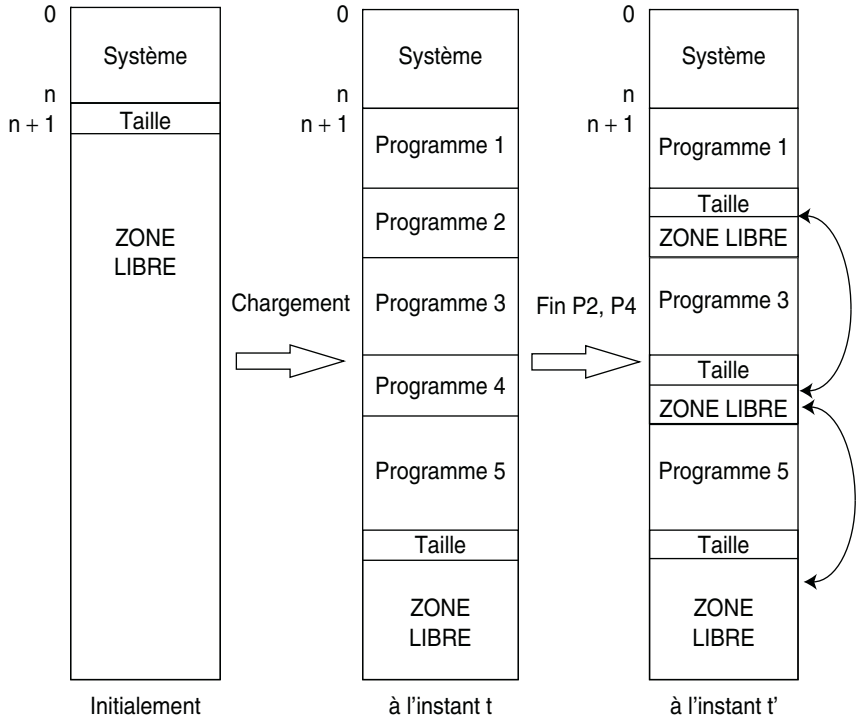


Figure 13.2 Évolution de la mémoire centrale gérée en partitions variables.

Gestion de la mémoire sur l'OS-MFT

Sur le système d'exploitation OS-MFT, ancêtre de MVS, la mémoire réelle était divisée au plus en 15 partitions de taille fixe, attribuées à 15 utilisateurs différents. La taille de chaque partition, ainsi que son adresse origine étaient fixées au démarrage du système. L'occupation de la mémoire pouvait être très mauvaise, un travail de petite taille pouvant être placé dans une partition bien plus grande que lui, générant une perte de place qualifiée de *fragmentation interne*.

Méthodes d'allocations d'une zone libre

Charger un nouveau programme consiste à trouver une zone libre suffisamment grande pour pouvoir y placer ce programme, tout en minimisant la place perdue en mémoire centrale et le temps de choix d'une zone libre. Trois stratégies existent pour le choix de la zone libre : stratégie de la première zone libre (*First Fit*), stratégie de la meilleure zone libre (*Best Fit*) et stratégie de la plus mauvaise zone libre (*Worst Fit*).

➤ Stratégie de la première zone libre

Une première stratégie pour choisir une zone libre est de prendre la première zone libre suffisamment grande trouvée au cours du parcours de la liste chaînée des zones, c'est-à-dire la première zone libre ZL telle que taille (ZL) ≥ taille (programme à charger).

Sur la figure 13.3, le programme 7 dont la taille est égale à 80 Ko est ainsi placé dans la zone libre de 120 Ko ce qui crée une nouvelle zone libre résiduelle de 40 Ko.

Cette méthode optimise la rapidité du choix d'une zone, mais elle ne conduit pas forcément à la meilleure utilisation possible de la mémoire centrale.

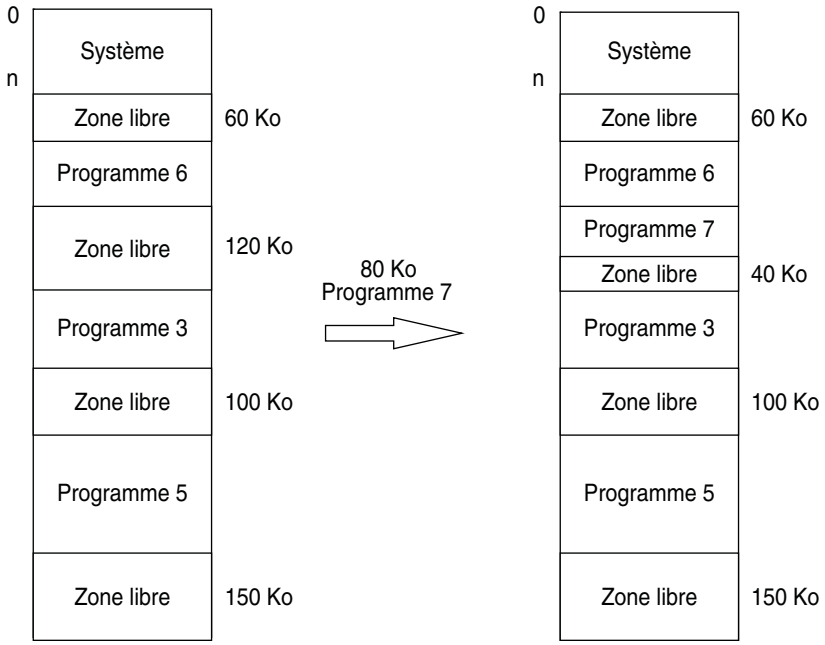


Figure 13.3 Stratégie de la première zone libre.

➤ Stratégie de la meilleure zone libre

Une seconde stratégie pour choisir une zone libre est de prendre la zone libre dont la taille est la plus proche de celle du programme à allouer, donc celle engendrant la plus petite zone libre résiduelle. C'est la zone libre Z_i pour laquelle pour tout i, taille (Z_i) – taille (programme à charger) est minimum.

Sur la figure 13.4, le programme 7 dont la taille est égale à 80 Ko est ainsi placé dans la zone libre de 100 Ko ce qui crée une nouvelle zone libre résiduelle de 20 Ko.

Cette méthode optimise l'utilisation de la mémoire centrale. Elle crée cependant des zones libres résiduelles de taille de plus en plus petite, qui ne sont pas forcément utilisables pour une nouvelle allocation. Par contre, elle est beaucoup moins rapide en terme de choix de la zone libre, puisque l'ensemble des zones libres de la mémoire centrale doit être examiné afin de choisir celle dont la taille est la plus proche de celle du programme à charger.

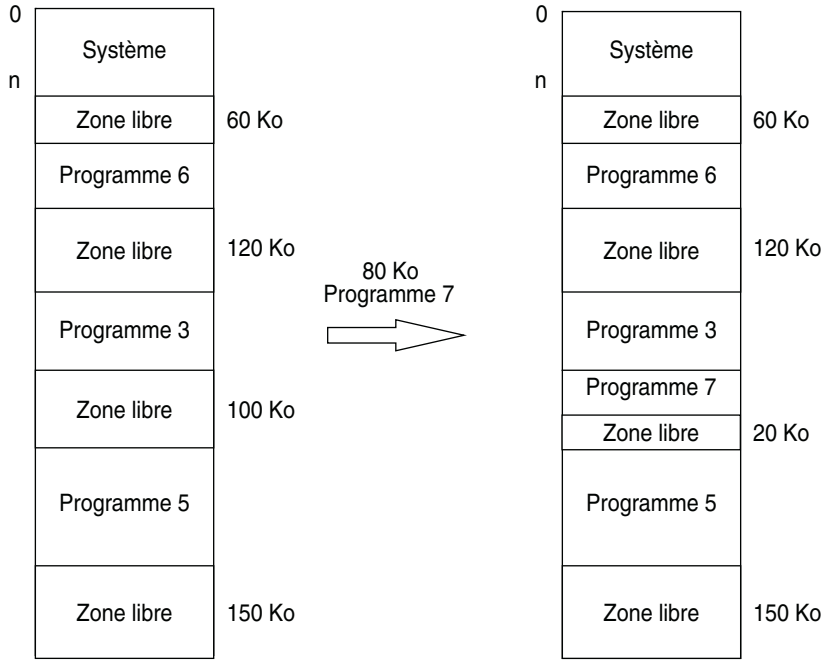


Figure 13.4 Stratégie de la meilleure zone libre.

➤ Stratégie de la plus mauvaise zone libre

Une dernière stratégie pour choisir une zone libre est de prendre la zone libre dont la taille est la plus éloignée de celle du programme à allouer, donc celle engendrant la plus grande zone libre résiduelle. C'est la zone libre Z_i pour laquelle pour tout i, taille (Z_i) – taille (programme à charger) est maximum.

Sur la figure 13.5, le programme 7 dont la taille est égale à 80 Ko est placé dans la zone libre de 150 Ko ce qui crée une nouvelle zone libre résiduelle de 70 Ko.

Cette méthode présente le même inconvénient que la méthode de la meilleure zone libre en terme de vitesse de choix de la zone libre. Par contre, les zones résiduelles créées suite à une allocation, sont davantage utilisables pour de nouvelles allocations.

Fragmentation externe et compactage

➤ Fragmentation externe

Au fur et à mesure des opérations d'allocations et de désallocations, la mémoire centrale devient composée d'un ensemble de zones occupées et de zones libres éparpillées dans toute l'étendue de la mémoire centrale. Ces zones libres peuvent être chacune trop petites pour permettre l'allocation de nouveaux programmes, alors que la somme totale de l'espace libre en mémoire centrale est suffisant : on parle alors de *fragmentation externe* de la mémoire centrale.

Ainsi, sur la figure 13.6, la mémoire centrale comporte 4 zones libres respectivement de 60 Ko, 120 Ko, 20 Ko et 150 Ko mais aucune d'elles n'est assez grande pour

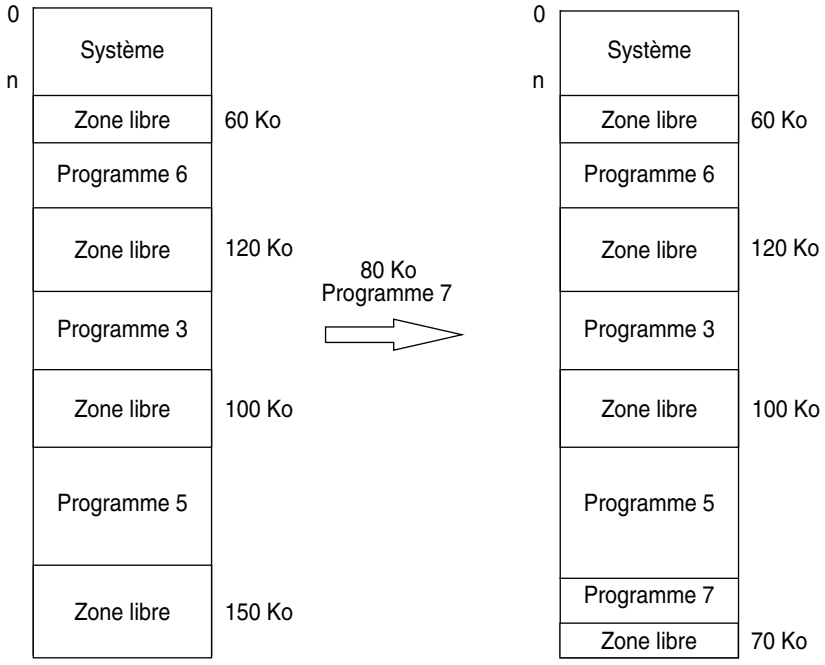


Figure 13.5 Stratégie de la plus mauvaise zone libre.

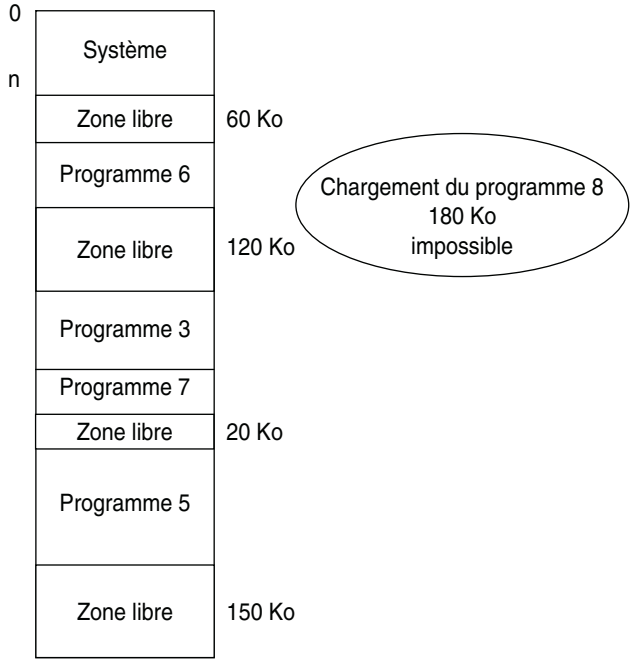


Figure 13.6 Fragmentation de la mémoire centrale.

contenir le programme 8 de 180 Ko. Pourtant l'ensemble des 4 zones libres forme un espace de 60 + 120 + 20 + 150 = 350 Ko suffisant pour y placer le programme 8. Il y a fragmentation externe.

La méthode d'allocation par partitions variables n'engendre pas directement le problème de fragmentation interne évoqué dans l'exemple des partitions fixes du système OS-MFT, puisque la taille des partitions allouées s'adapte à celle des programmes. Cependant, si la zone résiduelle créée à l'issue d'une allocation est trop petite (quelques mots) pour permettre une nouvelle allocation, le système peut choisir d'allouer les mots résiduels plutôt que de les considérer comme une nouvelle zone libre. Il existe alors bel et bien une fragmentation interne.

➤ Compactage de la mémoire centrale

Une solution au problème de la fragmentation externe est apportée par l'opération de *compactage de la mémoire centrale*. Le compactage de la mémoire centrale consiste à déplacer les programmes en mémoire centrale de manière à ne créer qu'une seule et unique zone libre. Le compactage de la mémoire centrale est une opération coûteuse. Il n'existe pas d'algorithme simple permettant d'optimiser le nombre d'octets déplacés lors d'une telle opération. Par ailleurs elle suppose un chargement dynamique des programmes : déplacer un programme consiste alors à changer la valeur d'adresse d'implantation chargée dans le registre de translation.

La figure 13.7 illustre le principe du compactage de la mémoire centrale qui permet ici de charger le programme 8 dont la taille est égale à 180 Ko.

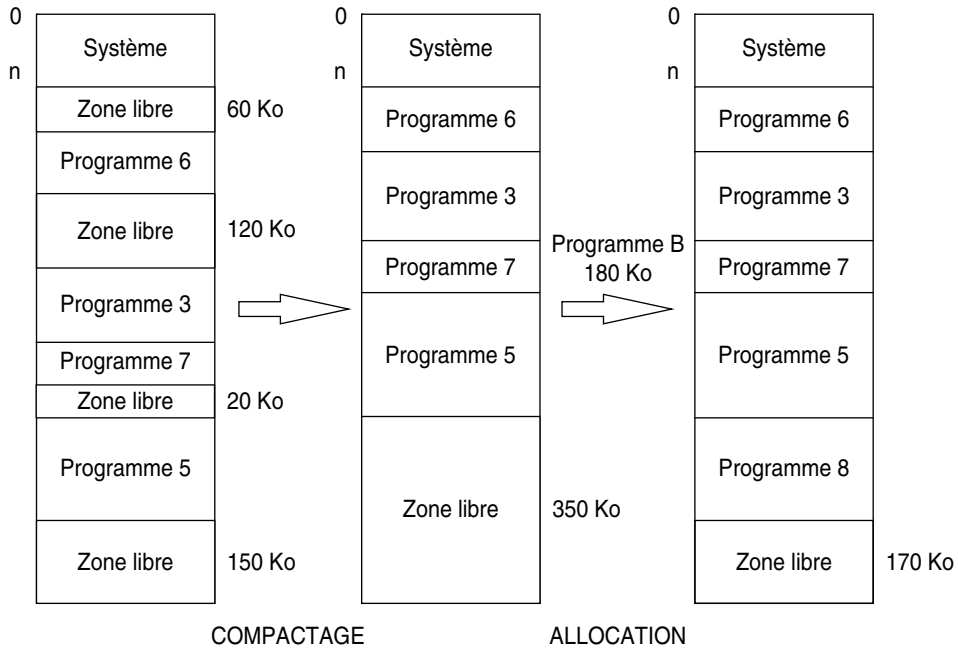


Figure 13.7 Compactage de la mémoire centrale.

Protection des partitions

Chaque partition allouée à un programme constitue un espace d'adressage protégé pour le processus correspondant qui doit être gardé des accès mémoire des autres processus.

Cette protection est réalisée en vérifiant que chaque adresse logique générée par le processeur pour le compte d'un processus est bien inférieure strictement à la taille de la partition allouée au processus concerné. La taille limite de la partition allouée à un processus fait partie du contexte du processus au même titre que la valeur de l'adresse d'implantation adr en mémoire centrale de cette partition. Les valeurs de ces grandeurs pour le processus actif sont chargées dans des registres de la MMU, respectivement dans le registre de translation et dans le registre limite (figure 13.8).

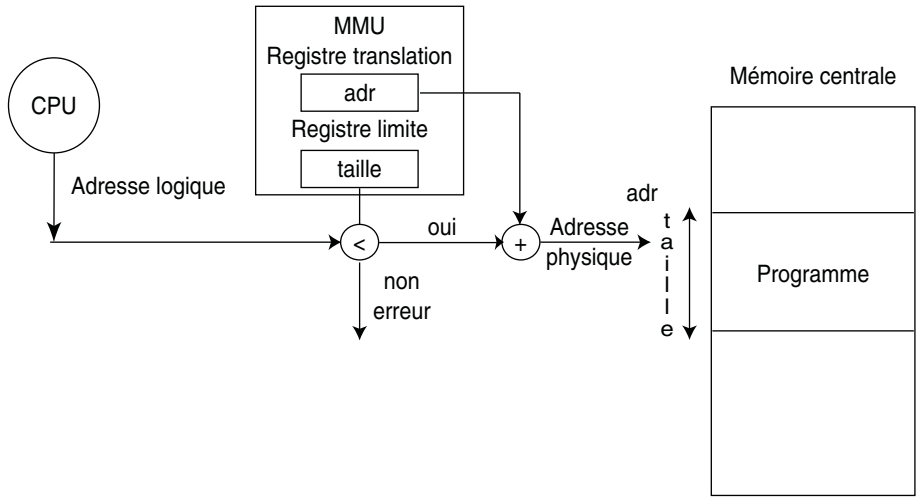


Figure 13.8 Protection des partitions.

13.2.2 Allocation non contiguë de la mémoire physique

La méthode d'allocation par partitions variables présente deux difficultés :

– comme le programme à charger en mémoire centrale est considéré comme un ensemble de mots insécable, elle nécessite de trouver en mémoire centrale une zone libre d'un seul tenant suffisamment grande pour y placer le programme. Si la taille du programme est importante, trouver une telle zone libre peut se révéler difficile ;

– les allocations et désallocations successives engendrent une fragmentation externe de la mémoire centrale, nécessitant une opération de compactage de la mémoire centrale qui est excessivement coûteuse.

Une solution à la première difficulté évoquée ci-dessus est de considérer que l'ensemble des mots constituant un programme est maintenant un ensemble sécable. Ainsi, le programme à charger est divisé en un ensemble de morceaux, chaque

morceau étant lui-même un ensemble de mots contigus insécable. Chaque morceau du programme est alors alloué en mémoire centrale indépendamment des autres :

– Si le programme est divisé en un ensemble de morceaux de taille fixe et égale, le mécanisme d'allocation de la mémoire centrale est alors celui de la *pagination*. Chaque morceau est appelé *page*.
– Si le programme est divisé en un ensemble de morceaux de taille variable, le mécanisme d'allocation de la mémoire centrale est alors celui de la *segmentation*. Chaque morceau est appelé *segment*.

La pagination

➤ Principe de la pagination

Dans le mécanisme de pagination, l'espace d'adressage du programme est découpé en morceaux linéaires de même taille appelés *pages*. L'espace de la mémoire physique est lui-même découpé en morceaux linéaires de même taille appelés *case* ou *cadre de page*. La taille d'une case est égale à la taille d'une page. Cette taille est définie par le matériel, comme étant une puissance de 2, variant selon les systèmes d'exploitation entre 512 octets et 8 192 octets. Ainsi, sur l'IBM 370, la taille des pages est fixée à 4 Ko tandis que sur une machine DEC la taille des pages est de 512 mots.

Dans ce contexte, charger un programme en mémoire centrale consiste à placer les pages dans n'importe quelle case disponible. Pour connaître à tout moment quelles sont les cases libres en mémoire centrale à un instant t, le système maintient une table appelée *table des cadres de pages* ou *table des cases* qui indique pour chaque case de la mémoire physique, si la case est libre ou occupée, et si elle est occupée, quelle page et quel processus la possèdent.

Ce mécanisme d'allocation de la mémoire centrale n'engendre pas de fragmentation externe. Il peut provoquer une fragmentation interne dans la mesure où la taille du programme à allouer n'est pas forcément un multiple de la taille des pages.

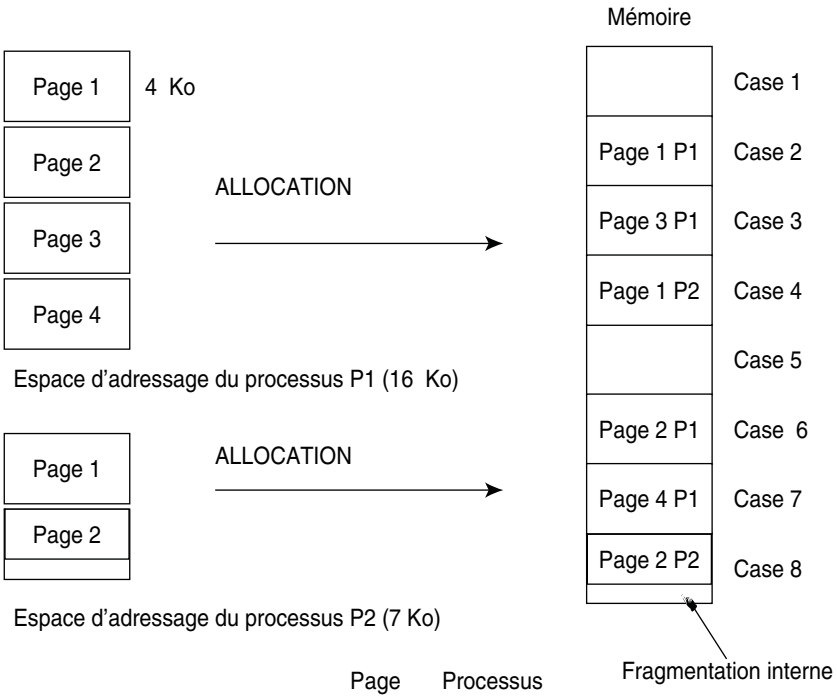
La figure 13.9 donne un exemple d'application de ce mécanisme d'allocation pour deux processus P1 et P2 dont les espaces d'adressage sont respectivement égaux à 16 Ko et 7 Ko. Les pages et les cases ont-elles une taille de 4 Ko.

➤ Adresse logique et table des pages

Conversion adresse logique – adresse physique

L'espace d'adressage du processus étant découpé en pages, les adresses générées dans cet espace d'adressage sont des *adresses paginées*, c'est-à-dire qu'un octet est repéré par son emplacement relativement au début de la page à laquelle il appartient. L'adresse d'un octet est donc formée par le couple <numéro de page p à laquelle appartient l'octet, déplacement d relativement au début de la page p>. Pour une adresse logique de m bits, en considérant des pages de 2^n octets, les m – n premiers bits correspondent au numéro de page p et les n bits restants au déplacement d dans la page.

Les octets dans la mémoire physique ne peuvent être adressés au niveau matériel que par leur adresse physique. Pour toute opération concernant la mémoire, il faut donc convertir l'adresse paginée générée au niveau du processeur en une adresse



	Page	Processus
Case 1	– 1 (libre)	
Case 2	1	P1
Case 3	3	P1
Case 4	1	P2
Case 5	– 1 (libre)	
Case 6	2	P1
Case 7	4	P1
Case 8	2	P2

Table des cases

Figure 13.9 Principe de la pagination.

physique équivalente. L'adresse physique d'un octet s'obtient à partir de son adresse logique en remplaçant le numéro de page p par l'adresse physique d'implantation de la case contenant la page p et en ajoutant à cette adresse physique d'implantation, le déplacement d de l'octet dans la page. C'est la MMU qui effectue cette conversion.

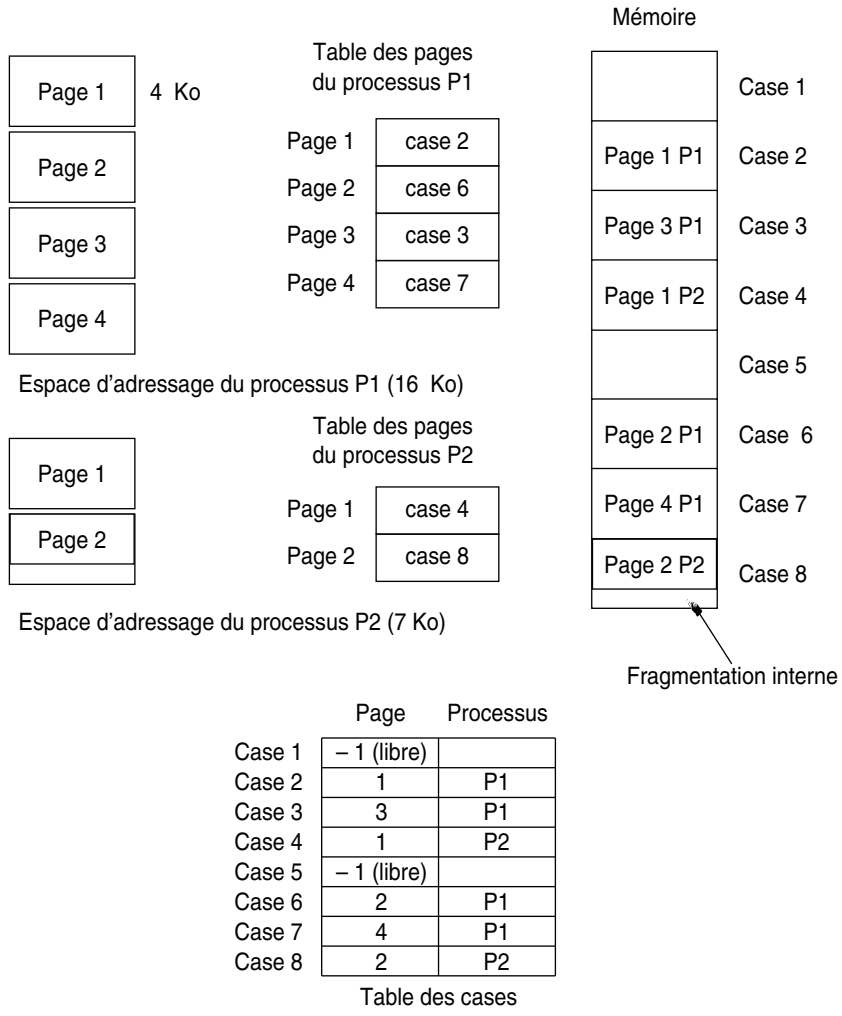
Pour toute page, il faut donc connaître dans quelle case de la mémoire centrale celle-ci a été placée. Cette correspondance s'effectue grâce à une structure particulière appelée la *table de pages*.

La table des pages

Dans une première approche, la table des pages est une table contenant autant d'entrées que de pages dans l'espace d'adressage d'un processus. Chaque processus a sa propre table des pages. Chaque entrée de la table est un couple <numéro de page, numéro de

case physique dans laquelle la page est chargée> ou <numéro de page, adresse d'implantation de case physique dans laquelle la page est chargée>.

Dans l'exemple de la figure 13.10, le processus P1 a 4 pages dans son espace d'adressage, donc la table des pages a 4 entrées. Chaque entrée établit l'équivalence



	Page	Processus
Case 1	– 1 (libre)	
Case 2	1	P1
Case 3	3	P1
Case 4	1	P2
Case 5	– 1 (libre)	
Case 6	2	P1
Case 7	4	P1
Case 8	2	P2

Table des cases

Figure 13.10 Table des pages.

numéro de page, numéro de case dans laquelle la page est chargée relativement au schéma de la mémoire centrale. Le processus P2 quant à lui a un espace d'adressage composé de 2 pages, donc sa table des pages a 2 entrées. Par ailleurs, l'espace d'adressage du processus P2 étant de 7 Ko, la page 2 du processus a une taille égale à 3 Ko. Il s'ensuit un phénomène de fragmentation interne en mémoire physique au niveau de la case 8 dans laquelle la page 2 est chargée.

Puisque chaque processus dispose de sa propre table des pages, chaque opération de commutation de contexte se traduit également par un changement de table des pages, de manière à ce que la table active corresponde à celle du processus élu.

Deux approches existent pour la réalisation de la table des pages :

– la table des pages est une structure matérielle réalisée grâce à des registres de la MMU. L'ensemble des registres composant la table des pages est sauvegardé avec le contexte processeur dans le PCB du processus ;
– la table des pages est une structure logicielle placée en mémoire centrale. L'adresse en mémoire centrale de la table des pages du processus actif est placée dans un registre de la MMU, le PTBR (*page-table base register*). Chaque processus sauvegarde dans son PCB la valeur de PTBR correspondant à sa table.

Dans la première approche, accéder à un emplacement mémoire nécessite seulement un accès à la mémoire, celui nécessaire à la lecture ou l'écriture de l'octet recherché puisque la table des pages est stockée dans des registres du processeur. Cependant, cette solution ne peut convenir que pour de petites tables des pages, n'offrant par exemple pas plus de 256 entrées.

La deuxième approche permet de réaliser des tables de pages de très grande taille. Cependant, accéder à un emplacement mémoire à partir d'une adresse paginée <p, d> nécessite maintenant deux accès à la mémoire (figure 13.11) :

– un premier accès permet de lire l'entrée de la table des pages correspondant à la page p cherchée et délivre une adresse physique c de case dans la mémoire centrale[1] ;
– un second accès est nécessaire à la lecture ou l'écriture de l'octet recherché à l'adresse c + d.

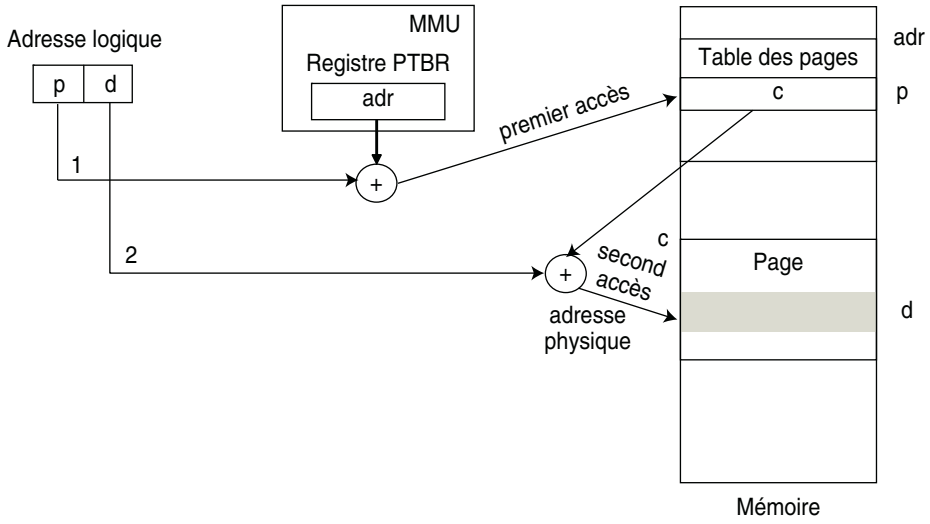


Figure 13.11 Traduction d'une adresse paginée en adresse physique.

1. Si l'entrée de la table des pages contient le numéro nc de la case dans laquelle la page est chargée, l'adresse d'implantation c est obtenue en multipliant nc par la taille en octet d'une case.

Pour accélérer les accès à la mémoire centrale et compenser le coût lié à la pagination, un cache associatif (*TLB look-aside buffers*) est placé en amont de la mémoire centrale. Ce cache associatif contient les couples <numéro de page p, adresse d'implantation de la case contenant p> les plus récemment accédés. Lorsque la MMU doit effectuer une conversion d'adresse paginée, elle cherche tout d'abord dans le cache si la correspondance <numéro de page p, adresse d'implantation de la case contenant p> n'est pas dans le cache. Si non, elle accède à la table des pages en mémoire centrale et place le nouveau couple référencé dans le cache. Si oui, elle effectue directement la conversion : un seul accès mémoire est alors nécessaire pour accéder à l'octet recherché (figure 13.12).

La taille des pages est souvent choisie comme étant un compromis entre la taille de la table des pages et la fragmentation interne possible. Ainsi, opter pour une petite taille de pages (inférieure à 4 Ko) diminue le problème de fragmentation interne, mais conduit à une plus grande table des pages pour chaque processus. Au contraire, augmenter la taille des pages augmente le risque de fragmentation interne, mais amène à de plus petites tables des pages.

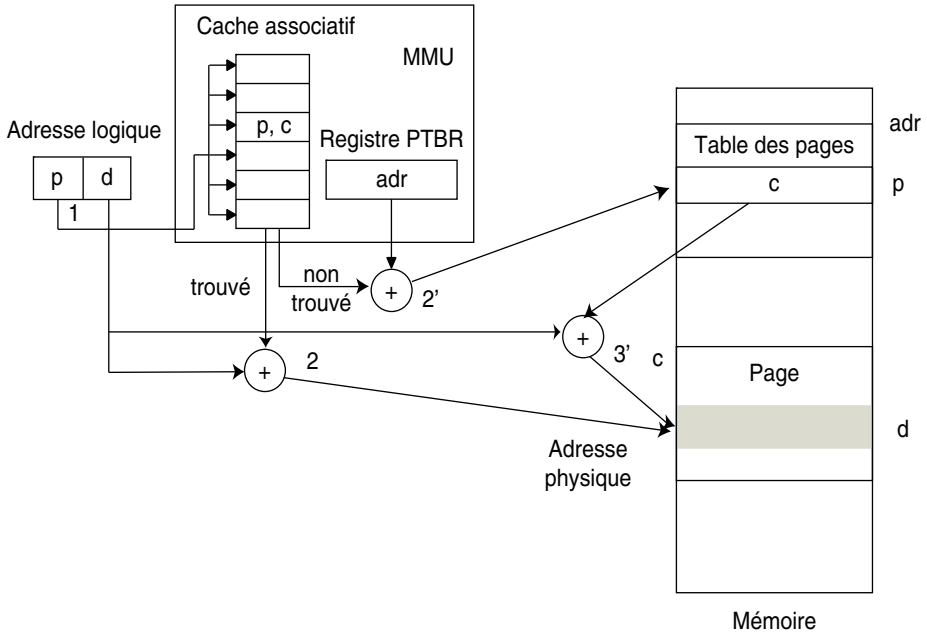


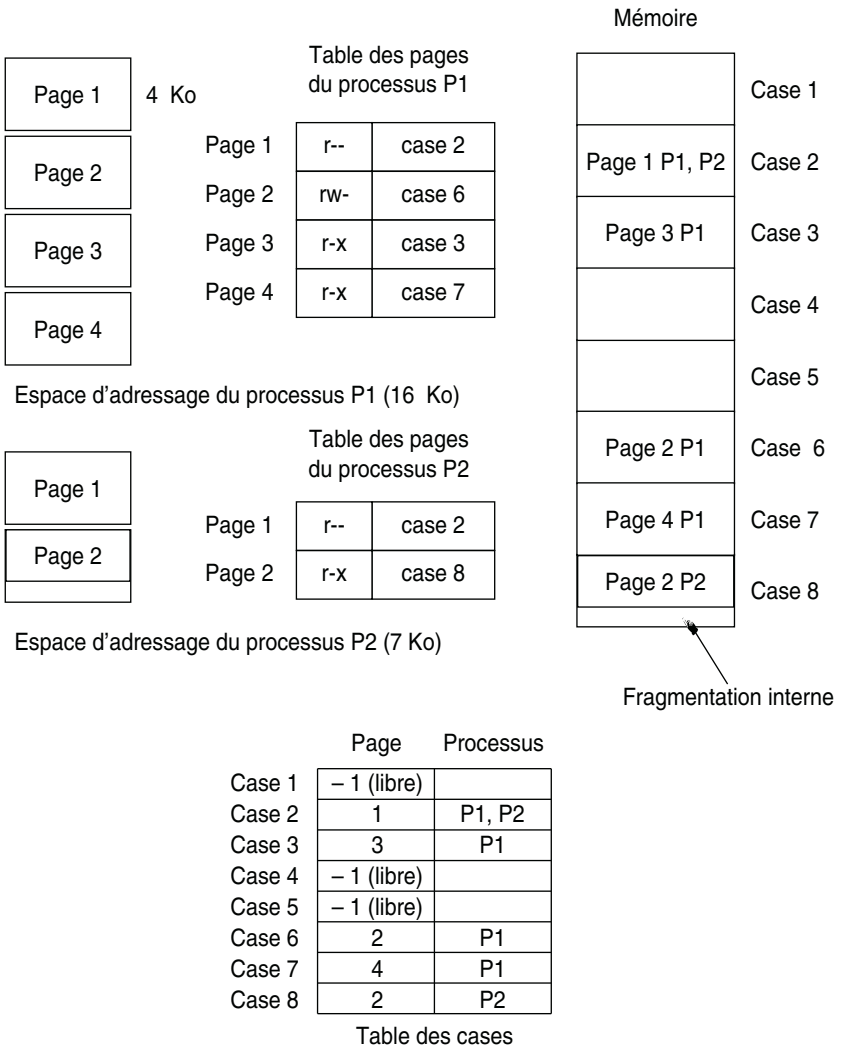
Figure 13.12 Traduction d'une adresse paginée en adresse physique avec ajout d'un cache associatif.

➤ Protection de l'espace d'adressage des processus

Des bits de protection sont associés à chaque page de l'espace d'adressage du processus et permettent ainsi de définir le type d'accès autorisés à la page. Ces bits de protection sont mémorisés pour chaque page, dans la table des pages du processus. Classi-

quement, 3 bits sont utilisés pour définir respectivement l'autorisation d'accès en lecture (r), écriture (w) et exécution (x).

Lors d'un accès à une page, la cohérence du type d'accès avec les droits associés à la page est vérifiée et une trappe est levée par le système d'exploitation si le type d'accès réalisé est interdit. Par exemple, sur la figure 13.13, la page 1 du processus P1 est définie avec le seul accès en lecture (r--) autorisé. La demande d'exécution d'une instruction du type `STORE D R1 page1, d`, demandant l'écriture du contenu du registre R1 dans la cellule mémoire située à l'emplacement d de la page 1 provoque une trappe et l'arrêt de l'exécution du processus P1.



	Page	Processus
Case 1	– 1 (libre)	
Case 2	1	P1, P2
Case 3	3	P1
Case 4	– 1 (libre)	
Case 5	– 1 (libre)	
Case 6	2	P1
Case 7	4	P1
Case 8	2	P2

Table des cases

Figure 13.13 Entrées de la table des pages avec les bits de protection et partage des pages entre processus.

Par ailleurs, chaque processus ne peut avoir accès qu'à sa propre table des pages et donc à ses propres pages. De ce fait, chaque espace d'adressage est protégé vis-à-vis des accès des autres processus. Malgré tout, il peut être souhaitable que des pages soient accessibles par plusieurs processus différents. C'est par exemple souhaitable pour éviter la duplication en mémoire centrale du code des bibliothèques ou la duplication de code exécutable réentrant comme le code de l'éditeur de texte utilisé couramment par les utilisateurs de la machine. Pour que deux processus puissent partager un ensemble de pages, il faut que chacun des deux processus référence cet ensemble de pages dans sa table des pages respective. Ainsi, sur la figure 13.13, les processus P1 et P2 référencent tous les deux la même page 1, située dans la case 2 de la mémoire centrale.

➤ Pagination multiniveaux

L'espace d'adresses logiques supporté par les systèmes d'exploitation actuels est très grand, de l'ordre de 2^{32} à 2^{64} octets. Dans de telles conditions, la table des pages d'un processus peut devenir également de très grande taille et comporter jusqu'à un million d'entrées. Il n'est dès lors plus envisageable de charger de manière contiguë la table

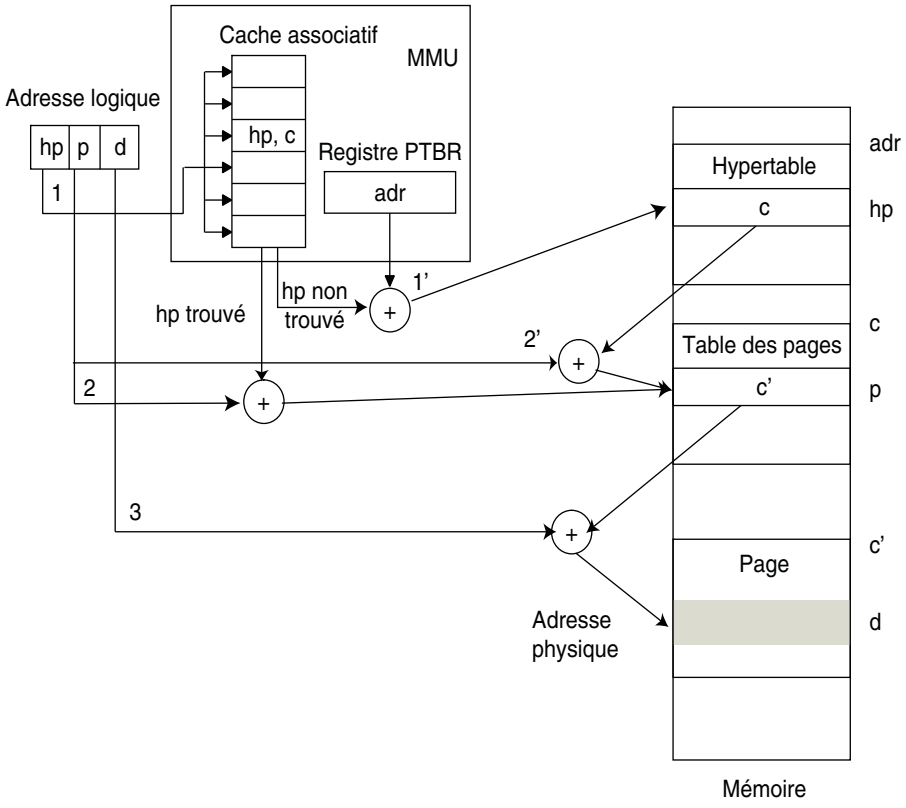


Figure 13.14 Pagination à deux niveaux.

des pages d'un processus. Une solution consiste alors à paginer la table des pages elle-même.

Ainsi, les processeurs de l'architecture Intel x86 peuvent adresser 4 Go. La taille des pages est de 4 Ko et une entrée de la table des pages occupe 4 octets. Dans ces conditions, la table des pages peut contenir 1 048 576 entrées ce qui représente une occupation mémoire de 4 Mo.

Dans ce contexte, l'adresse paginée devient un triplet <hp, p, d>, où hp désigne l'entrée d'une hypertable des pages, dont chaque entrée correspond à une page contenant une partie de la table des pages du processus. p désigne une entrée de cette partie de la table des pages qui permet d'accéder à la page p de l'espace d'adressage du processus et d un déplacement dans la page p.

L'hypertable des pages est elle-même placée en mémoire centrale et son adresse d'implantation en mémoire centrale est repérée par un registre matériel de la MMU. La conversion de l'adresse paginée <hp, p, d> en adresse physique et l'accès à l'octet mémoire désigné comporte maintenant une étape supplémentaire (figure 13.14) :

– un premier accès permet de lire l'entrée de l'hypertable des pages correspondant à la page hp cherchée et délivre une adresse physique c de case dans la mémoire centrale, qui contient une partie de la table des pages du processus ;
– un second accès à la case c permet de lire l'entrée p de la table des pages et de récupérer l'adresse physique de la case c'contenant la page p ;
– un troisième accès est nécessaire à la lecture ou l'écriture de l'octet recherché à l'adresse c' + d.

Pour accélérer la conversion de l'adresse logique vers l'adresse physique, un cache associatif contient les couples <hp, p> les plus récemment accédés.

➤ Un exemple : gestion mémoire sur le système MVS/370

Sur l'architecture IBM 370, l'espace virtuel d'un processus peut atteindre 16 Mo. La taille d'une page et d'une case est fixée à 4 Ko. Une table des pages de processus peut donc contenir jusqu'à 2^{12} entrées, chaque entrée mesurant 2 octets et donc occuper 8 Ko. Un second niveau de pagination est donc mis en place pour supporter la taille de la table des pages des processus.

L'espace d'adressage d'un processus est découpé en morceaux ou hyperpages constitués chacun de 16 pages du processus. Une table des pages est associée à chaque hyperpage. L'hypertable des pages comporte 256 entrées de 4 octets, contenant chacune l'adresse d'une table des pages pour une hyperpage. L'adresse en mémoire centrale de cette table est maintenue pour le processus courant dans le registre de contrôle 1, CR1, appelé aussi STOR (*segment table origin register*).

Une adresse virtuelle sur 24 bits se décompose donc en trois parties :

– les 8 premiers bits constituent le numéro d'hyperpage hp ;
– les 4 suivants donnent le numéro de page p de l'hyperpage ;
– les 12 derniers donnent le déplacement d dans la page.

Pour accélérer la conversion de l'adresse virtuelle vers l'adresse réelle, un cache associatif de 8 à 100 postes est placé en amont de l'hypertable.

La segmentation

La pagination constitue un découpage de l'espace d'adressage du processus qui ne correspond pas à l'image que le programmeur a de son programme. Pour le programmeur, un programme est généralement constitué des données manipulées par ce programme, d'un programme principal, de procédures séparées et d'une pile d'exécution.

La *segmentation* est un découpage de l'espace d'adressage qui cherche à conserver cette vue du programmeur (figure 13.15). Ainsi, lors de la compilation, le compilateur associe un segment à chaque morceau du programme compilé. Un *segment* est un ensemble d'emplacements mémoire consécutifs non sécable. À la différence des pages, les segments d'un même espace d'adressage peuvent être de taille différente.

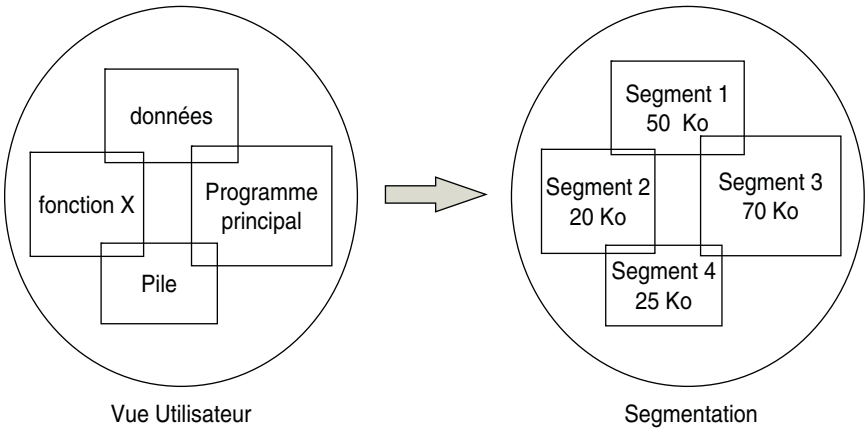


Figure 13.15 Segmentation d'un programme.

D'une manière générale, on trouvera un segment de code, un segment de données et un segment de pile.

➤ Adresse logique et table des segments

Conversion adresse logique – adresse physique

D'une manière similaire à ce qui se passe avec la pagination, la segmentation de l'espace d'adressage d'un processus génère des *adresses segmentées*, c'est-à-dire qu'un octet est repéré par son emplacement relativement au début du segment auquel il appartient. L'adresse d'un octet est donc formée par le couple <numéro de segment s à laquelle appartient l'octet, déplacement d relativement au début du segment s>.

Pour toute opération concernant la mémoire, il faut ici encore convertir l'adresse segmentée générée au niveau du processeur en une adresse physique équivalente.

L'adresse physique d'un octet s'obtient à partir de son adresse segmentée en remplaçant le numéro de segment s de l'adresse segmentée par l'adresse physique d'implantation du segment en mémoire centrale et en ajoutant à cette adresse physique d'implantation, le déplacement d de l'octet dans le segment. C'est la MMU qui effectue cette conversion.

Il faut donc connaître pour tout segment, l'adresse d'implantation dans la mémoire centrale du segment : cette correspondance s'effectue grâce à une structure particulière appelée la *table des segments*.

La table des segments

La table des segments est une table contenant autant d'entrées que de segments dans l'espace d'adressage d'un processus. Chaque entrée i de la table est un couple <adresse adr d'implantation du segment i, taille t du segment i>. Sur la figure 13.16, le processus P1 a 4 segments dans son espace d'adressage, donc la table des segments a 4 entrées. Chaque entrée établit l'équivalence entre d'une part le numéro de segment, d'autre part l'adresse d'implantation du segment et sa taille, relativement au schéma de la mémoire centrale.

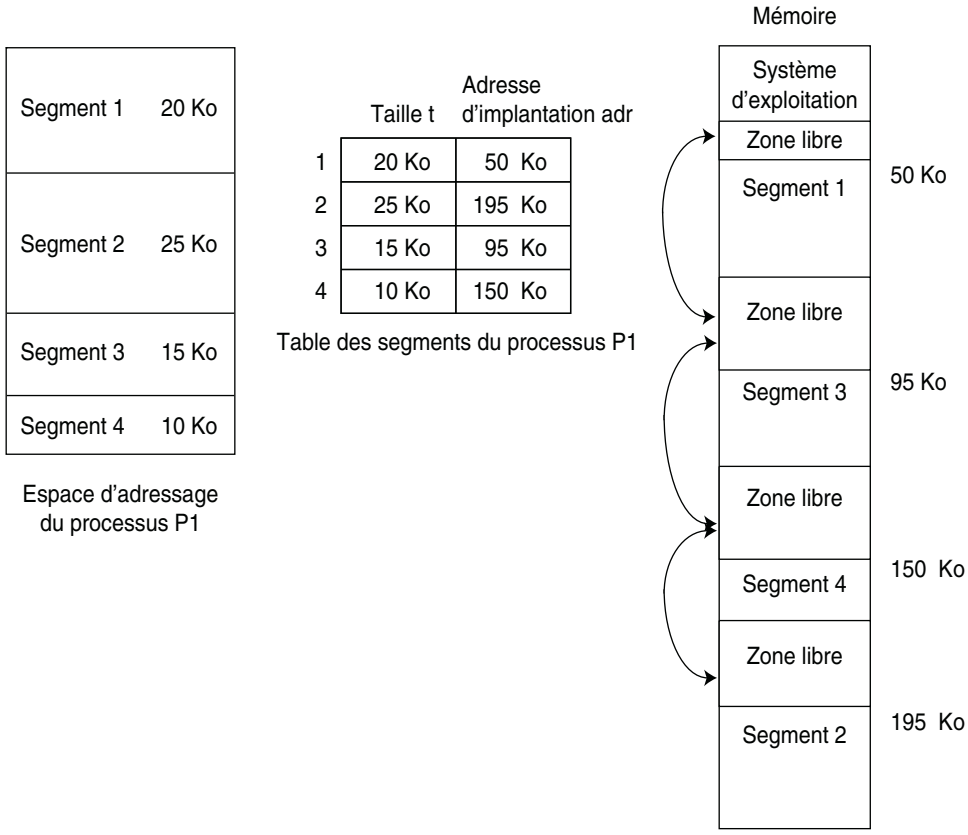


Figure 13.16 Table des segments.

Tout comme la table des pages, la table des segments peut être réalisée à l'aide de registres de la MMU, mais avec les mêmes limites au niveau de la taille de la table. Aussi, la table des segments est plus souvent placée en mémoire centrale. Un registre STBR (*segment-table base register*) contient l'adresse d'implantation en mémoire centrale de la table de segments du processus actif, tandis qu'un registre STLR (*segment-table length register*) contient le nombre de segments existants dans l'espace d'adressage du processus actif. Les valeurs contenues dans les registres STLR et STBR sont sauvegardées pour chaque processus dans son PCB.

La conversion d'une adresse segmentée <s, d> avec s, numéro de segment, et d déplacement dans le segment, suit les étapes suivantes (figure 13.17) :

– s est comparé à la valeur max_s contenue dans le registre STLR. Si s est supérieur à cette valeur, alors une trappe est levée car le segment adressé n'existe pas ;
– sinon s est additionné au contenu du registre LTBR de manière à indexer l'entrée de la table concernant le segment s. On récupère alors l'adresse d'implantation adr du segment s en mémoire centrale ;
– le déplacement d est alors comparé à la taille t du segment. Si d est supérieur à t, alors une trappe est levée car le déplacement est en dehors du segment. Sinon, le déplacement d est ajouté à l'adresse d'implantation du segment pour générer l'adresse physique.

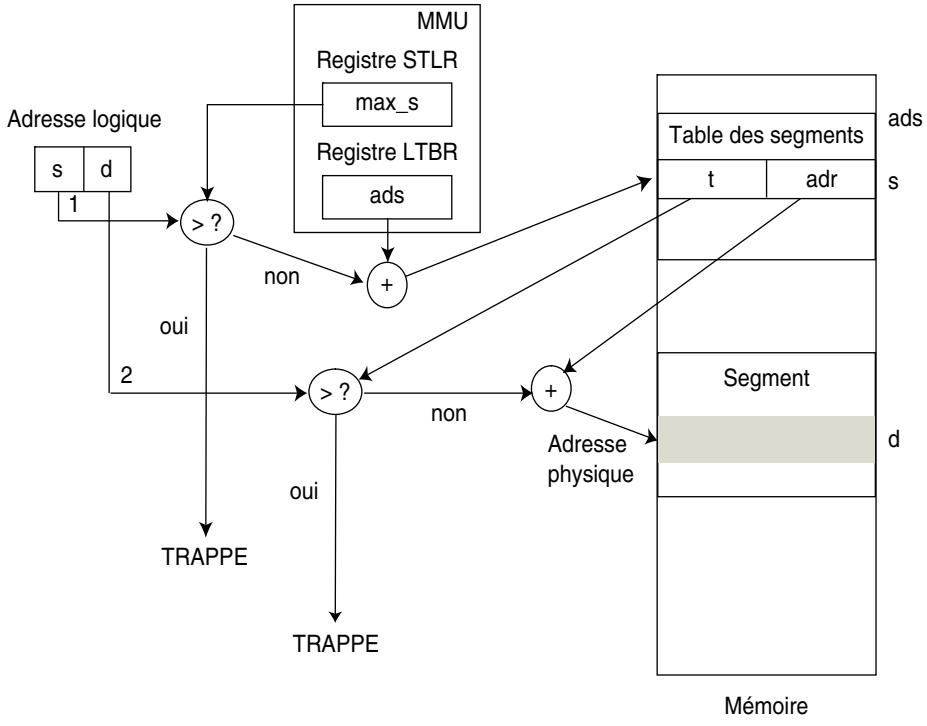


Figure 13.17 Traduction d'une adresse segmentée en adresse physique.

L'allocation des segments en mémoire centrale s'effectue selon le même principe que pour l'allocation de partitions variables. Pour allouer un segment de taille t, il faut trouver une zone libre dont la taille soit au moins égale à la taille du segment. Elle engendre les mêmes problèmes de fragmentation externe.

➤ Protection de l'espace d'adressage d'un processus

Des bits de protection sont associés à chaque segment de l'espace d'adressage du processus et permettent ainsi de définir le type d'accès autorisés au segment. Ces bits de protection sont mémorisés pour chaque segment, dans la table des segments du processus. Classiquement, 3 bits sont utilisés pour définir respectivement l'autorisation d'accès en lecture (r), écriture (w) et exécution (x).

Lors d'un accès à un segment, la cohérence du type d'accès avec les droits associés au segment est vérifiée et une trappe est levée par le système d'exploitation si le type d'accès réalisé est interdit.

Par ailleurs, chaque processus ne peut avoir accès qu'à sa propre table des segments et donc à ses propres segments. De ce fait, chaque espace d'adressage est protégé vis-à-vis des accès des autres processus. Pour que deux processus puissent partager un ensemble de segments, il faut que chacun des deux processus référence cet ensemble de segments dans sa table des segments respective.

➤ Pagination des segments

Une solution, très largement répandue, au problème de la fragmentation externe, est de combiner pagination et segmentation, c'est-à-dire de paginer les segments.

Dans le cas où pagination et segmentation sont simultanément employées, une table des segments est définie pour chaque espace d'adressage de processus. Chaque segment est à son tour paginé, il existe donc une table des pages pour chaque segment. Ainsi une entrée de la table des segments ne contient plus l'adresse du segment correspondant en mémoire physique mais contient l'adresse de la table des pages en mémoire physique pour ce segment.

L'adresse d'un octet dans l'espace d'adressage du processus devient un couple <s, d>, le déplacement d étant à son tour interprété comme étant un couple <numéro de page p, déplacement d' dans cette page>. Les mécanismes de traduction de l'adresse logique vers l'adresse physique concernant la segmentation et la pagination se superposent l'un à l'autre. Ainsi (figure 13.18) :

- s est comparé à la valeur max_s contenue dans le registre STLR. Si s est supérieur à cette valeur, alors une trappe est levée car le segment adressé n'existe pas ;
- sinon s est additionné au contenu du registre LTBR de manière à indexer l'entrée de la table concernant le segment s. On récupère alors l'adresse d'implantation adr_t de la table des pages du segment s en mémoire centrale ;
- le déplacement d est alors décomposé en un numéro de page p et un déplacement d' dans la page p. Le numéro de page p est ajouté à l'adresse adr_t de la table des pages du segment s pour indexer dans la table l'entrée concernant la page p.

L'adresse adr d'implantation en mémoire centrale de la case contenant la page p est ainsi récupérée ;

– enfin, le déplacement d' est ajouté à adr pour constituer l'adresse physique de l'octet recherché.

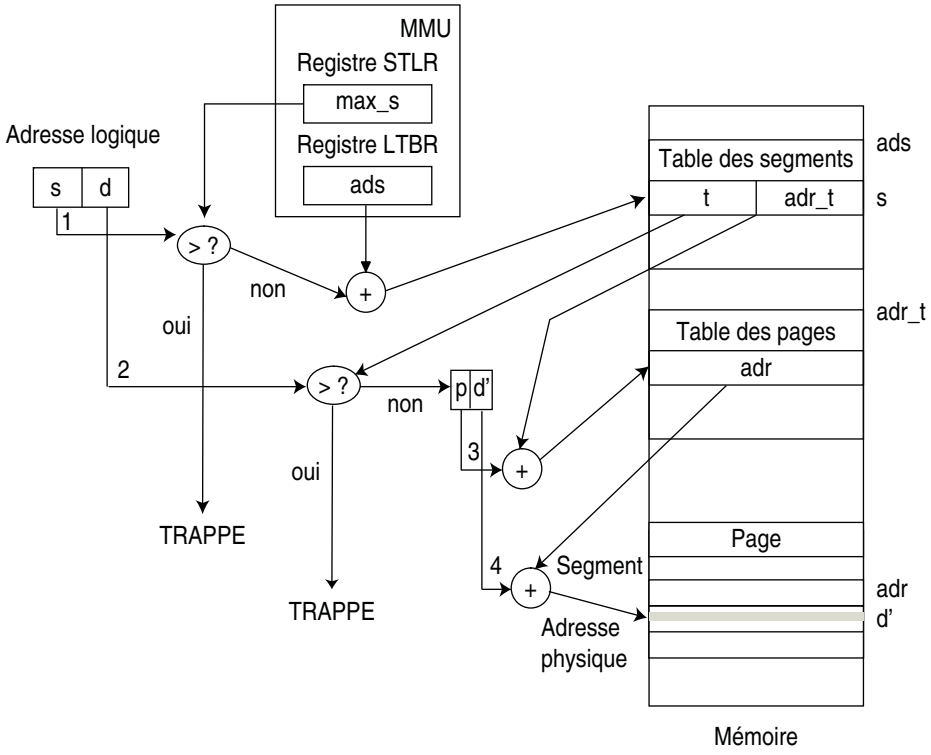


Figure 13.18 Pagination des segments.

13.3 MÉMOIRE VIRTUELLE

13.3.1 Principe de la mémoire virtuelle

La multiprogrammation implique de charger plusieurs programmes en mémoire centrale de manière à obtenir un bon taux d'utilisation du processeur. Supposons comme sur la figure 13.19 que l'exécution des processus P1, P2, P3 soit nécessaire pour obtenir ce taux d'utilisation du processeur satisfaisant. On peut remarquer qu'une fois les pages des processus P1 et P2 chargées dans la mémoire, il ne reste pas assez de cases libres pour charger la totalité des pages du programme 3. Ce dernier ne peut donc pas être chargé.

Lorsque l'on regarde l'exécution d'un processus, on s'aperçoit qu'à un instant donné le processus n'accède qu'à une partie de son espace d'adressage, par exemple

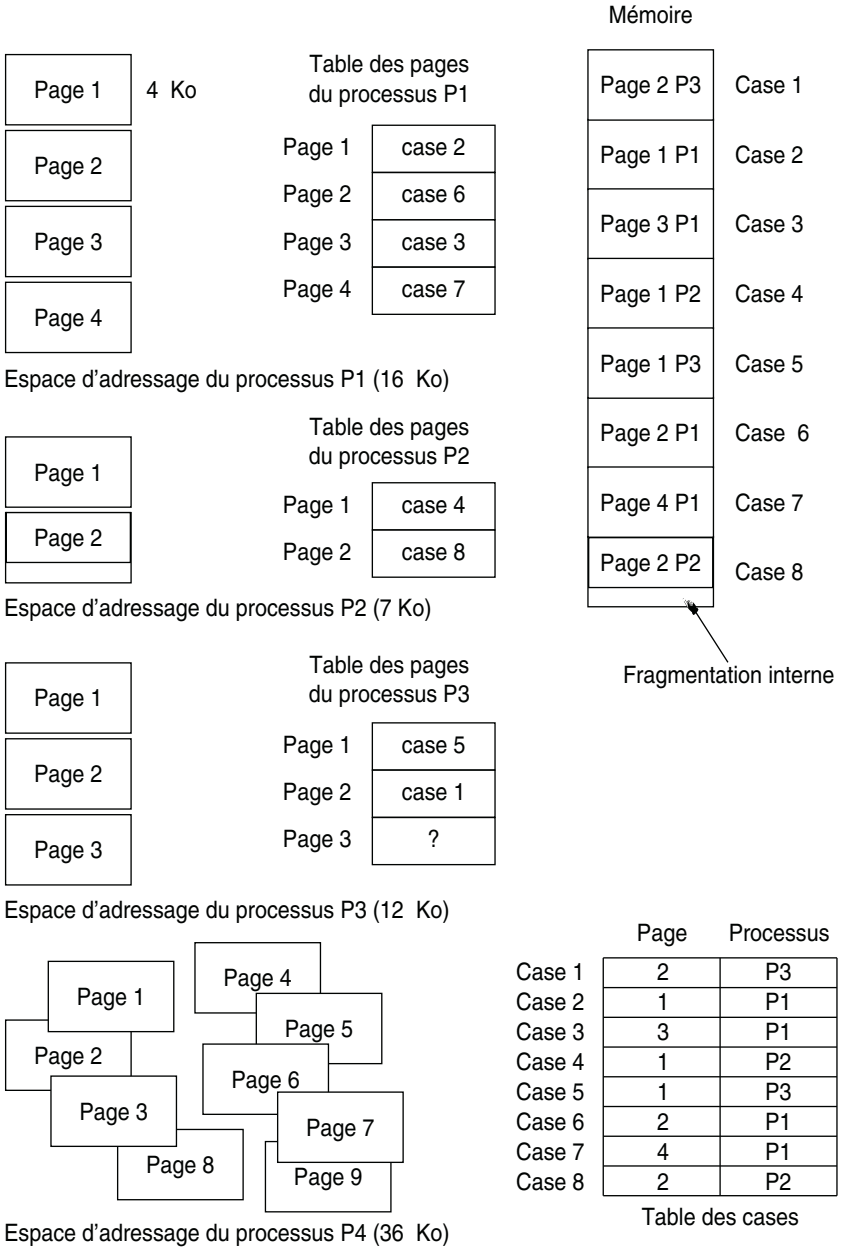


Figure 13.19 Insuffisance de la mémoire physique.

la page de code couramment exécutée par le processeur et la page de données correspondante. Les autres pages de l'espace d'adressage ne sont pas accédées et sont donc inutiles en mémoire centrale. Une solution pour pouvoir charger plus de processus dans la mémoire centrale est donc de ne charger pour chaque processus

que les pages couramment utilisées. Ainsi, sur la figure 13.20, seules les pages 1 et 3 du processus P1 sont chargées ainsi que la page 1 du processus P2 et les pages 1 et 2 du processus P3.

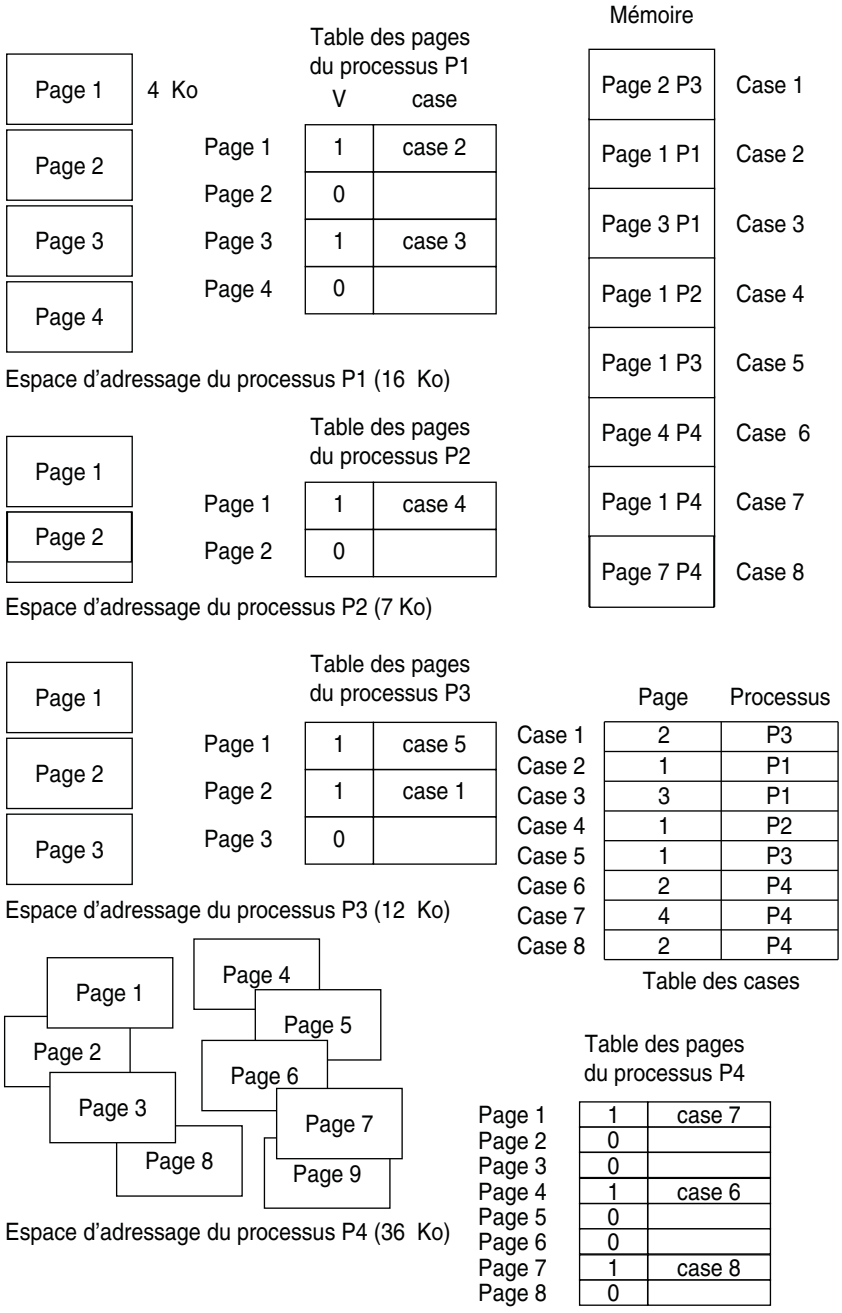


Figure 13.20 Mémoire virtuelle.

Par ailleurs, on remarquera sur cette même figure, que l'espace d'adressage du processus P4 est plus grand que la mémoire physique disponible pour les programmes utilisateurs. Le *principe de la mémoire virtuelle*, qui consiste donc à ne charger à un instant donné en mémoire centrale que la partie couramment utile de l'espace d'adressage des processus, permet de résoudre ce problème et autorise donc la constitution de programmes dont la taille n'est plus limitée par celle de la mémoire physique.

Puisque les pages d'un espace d'adressage de processus ne sont pas toutes chargées en mémoire centrale, il faut que le processeur puisse détecter leur éventuelle absence au moment où s'effectue la conversion de l'adresse logique vers l'adresse physique. Chaque entrée de la table des pages comporte alors un champ supplémentaire, le bit validation V, qui est à 1 si la page est effectivement présente en mémoire centrale, 0 sinon.

La figure 13.20 montre les valeurs des bits de validation pour les tables des pages des quatre processus P1, P2, P3 et P4, en tenant compte des chargements de leurs pages en mémoire centrale. Ainsi pour le processus P1, la page 1 est chargée dans la case 2, le bit de validation est à 1. La page 3 est chargée dans la case 3, le bit de validation est à 1. Par contre les pages 2 et 4 ne sont pas présentes en mémoire centrale et donc le bit de validation est à 0 : dans ce cas, le champ numéro de case n'a pas de signification.

Puisque l'ensemble des pages des processus ne sont plus chargées en mémoire centrale, il est maintenant nécessaire de définir comment les cases de la mémoire physique vont être réparties entre les processus. Deux stratégies existent :

– l'allocation équitable ou fixe répartit équitablement les cases de la mémoire centrale entre les processus qui disposent donc tous du même nombre de cases quelle que soit la taille de leur espace d'adressage. Il peut en résulter une mauvaise utilisation de la mémoire centrale car un processus avec un petit espace d'adressage peut disposer de beaucoup plus de cases que nécessaire à son exécution ;
– l'allocation proportionnelle répartit les cases de la mémoire centrale proportionnellement à la taille des processus, attribuant ainsi d'autant plus de cases à un processus que son espace d'adressage est important.

Le principe de la mémoire virtuelle est couramment implémenté avec la pagination à la demande, c'est-à-dire que les pages des processus ne sont chargées en mémoire centrale que lorsque le processeur demande à y accéder. La mémoire virtuelle peut également être implémentée sur un système de segmentation à la demande, mais elle est plus difficile à mettre en œuvre du fait de la longueur variable des segments et donc plus rare. C'est pourquoi, nous avons délibérément choisi ici de nous placer dans le seul contexte de la pagination.

13.3.2 Le défaut de page

Que se passe-t-il à présent lorsqu'un processus tente d'accéder à une page de son espace d'adressage qui n'est pas en mémoire centrale ? La MMU accède à la table des pages pour effectuer la conversion de l'adresse logique vers l'adresse physique et teste la valeur du bit de validation. Si la valeur du bit V est à 0, cela signifie que la

page n'est pas chargée dans une case et donc la conversion ne peut pas être réalisée. Une trappe appelée *défaut de page* est levée qui suspend l'exécution du processus puis initialise une opération d'entrées-sorties afin de charger la page manquante en mémoire centrale dans une case libre. Les pages constituant l'espace d'adressage du processus sont stockées dans une zone particulière du disque communément appelée

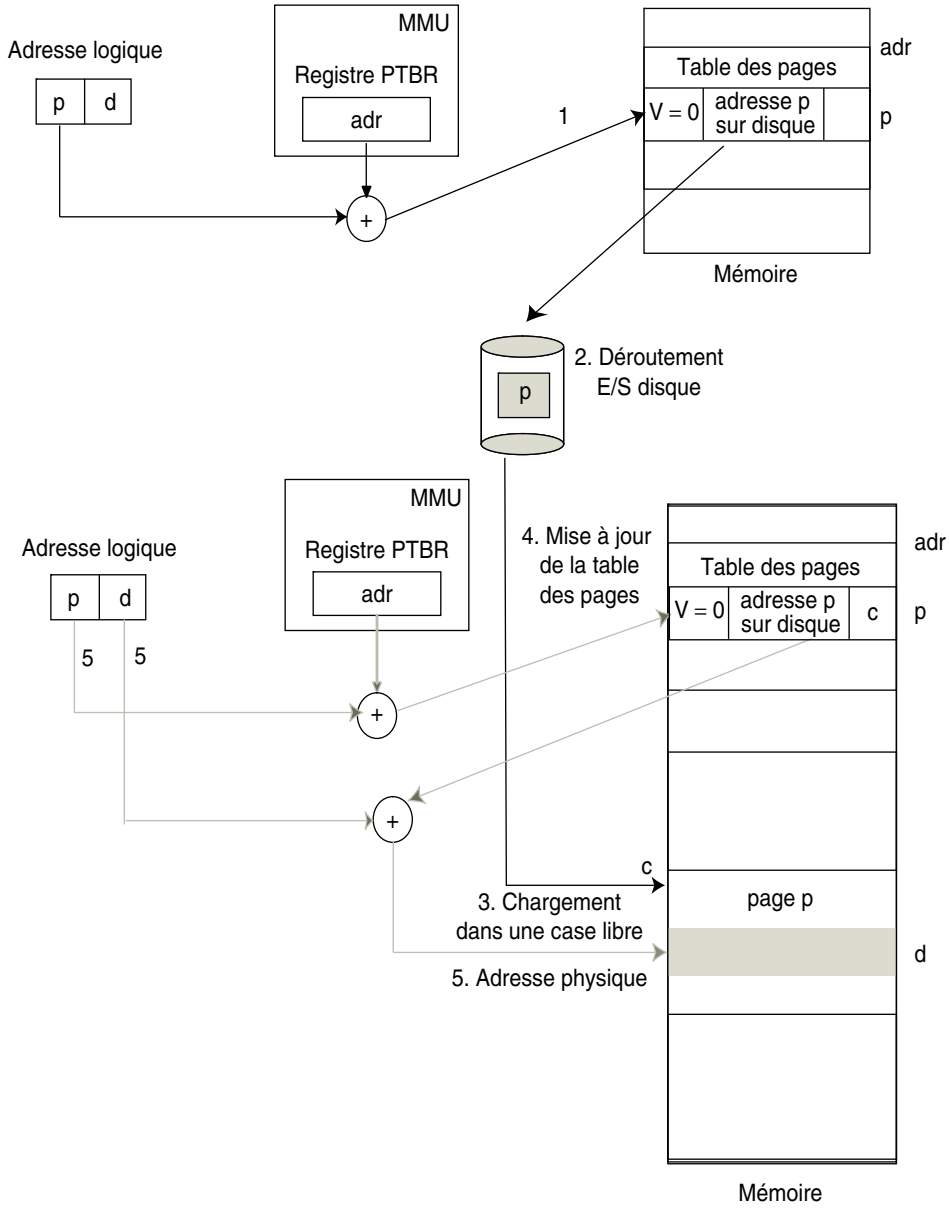


Figure 13.21 Défaut de page.

zone de swap. L'adresse de chaque page sur le disque est mémorisée dans l'entrée correspondante de la table des pages. Ainsi, sur un processeur de type Pentium, l'exception 14 correspond au défaut de pages.

Avec ce principe de pagination à la demande, le mécanisme de conversion d'une adresse logique <p, d> vers l'adresse physique correspondante devient (figure 13.21) :

1. accès à l'entrée p de la table des pages du processus actif et test de la valeur du bit de validation V;
2. si la valeur du bit V est à 0, alors il y a défaut de page. Une opération d'entrées-sorties est lancée pour charger la page dans la mémoire centrale (l'adresse de la page sur le disque est stockée dans la table des pages);
3. la page est placée dans une case libre, trouvée par l'intermédiaire de la table des cases;
4. la table des pages du processus est mise à jour c'est-à-dire que le bit de validation V passe à 1 et le champ numéro de case est renseigné avec l'adresse d'implantation de la case abritant maintenant la page p;
5. la conversion de l'adresse logique vers l'adresse physique est reprise selon le principe vu au paragraphe 13.2.2.

13.3.3 Le remplacement de pages

Lors d'un défaut de page, la page manquante est chargée dans une case libre. Cependant, la totalité des cases de la mémoire centrale peut être occupée. Il faut donc libérer une case de la mémoire physique pour y placer la nouvelle page. Lors de la libération d'une case, la page victime contenue dans cette case doit être sauvegardée sur le support de masse si elle a été modifiée lors de son séjour en mémoire centrale. Un bit M de modification mis à 1 si la page est modifiée est ajouté à chaque entrée de la table des pages afin d'être à même de savoir si la page doit être réécrite sur le disque avant d'être écrasée par la nouvelle page (figure 13.22).

Le choix de la page victime lors du traitement d'un défaut de page pour un processus peut se faire soit localement à ce processus, soit globalement sur l'ensemble des processus. Dans le premier cas, la page victime doit forcément appartenir à l'espace

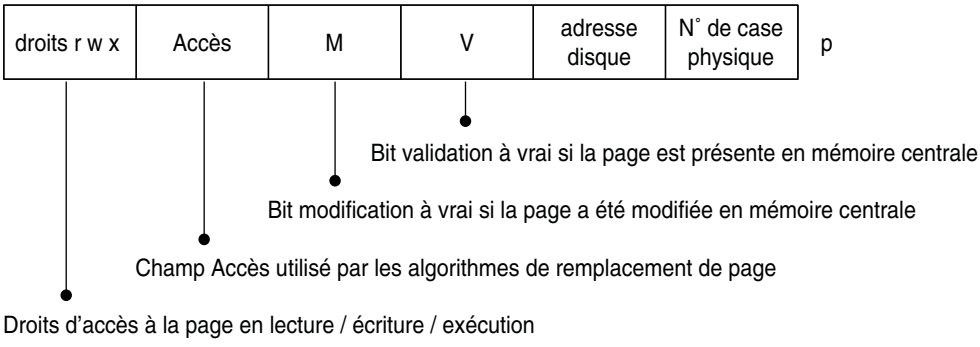


Figure 13.22 Format d'une entrée de la table des pages.

d'adressage du processus en défaut de page. Dans le second cas, la page victime peut appartenir à l'espace d'adressage de n'importe lequel des processus présents en mémoire centrale. Ce dernier cas est le plus souvent mis en œuvre car il donne de meilleurs résultats quant à l'utilisation de la mémoire centrale.

Le mécanisme de conversion d'une adresse logique <p, d> vers l'adresse physique correspondante devient maintenant :

1. accès à l'entrée p de la table des pages du processus actif et test de la valeur du bit de validation V;
2. si la valeur du bit V est à 0, alors il y a défaut de page.
3. une case libre est recherchée. Si il n'y a pas de cases libres, une page victime est choisie. Si la valeur du bit M pour cette page est à 1, alors une opération d'entrées-sorties est lancée pour sauvegarder la page victime;
4. une opération d'entrées-sorties est lancée pour charger la page dans la mémoire centrale dans la case libre ou libérée;
5. la table des pages du processus est mise à jour c'est-à-dire que le bit de validation V passe à 1 et le champ numéro de case est renseigné avec l'adresse d'implantation de la case abritant maintenant la page p;
6. la conversion de l'adresse logique vers l'adresse physique est reprise selon le principe vu au paragraphe 13.2.2.

Le choix optimal pour la page victime consiste à libérer une case en déchargeant de la mémoire centrale une page qui ne servira plus du tout, de façon à ne pas provoquer de défaut de page ultérieur sur cette même page. Évidemment, il n'est pas possible de savoir si telle ou telle page d'un processus est encore utile ou pas.

Plusieurs algorithmes, appelés *algorithmes de remplacement de pages*, ont été définis qui tendent plus ou moins vers l'objectif optimal. Ce sont les stratégies : FIFO (*First In, First Out*), LRU (*Least Recently Used*), algorithme de la seconde chance, LFU (*Least Frequently Used*) et MFU (*Most Frequently Used*).

L'évaluation de ces différentes stratégies s'effectue en comptant sur une même suite de références à des pages, le nombre de défaut de pages provoqués. Une telle suite de références à un même ensemble de pages est appelée *chaîne de références.* Ainsi dans les paragraphes suivants, nous donnons un exemple de chaque stratégie sur la chaîne de références 8, 1, 2, 3, 1, 4, 1, 5, 3, 4, 1, 4, 3 où chaque chiffre i correspond à un accès à la page i. Dans chaque exemple, nous supposons une mémoire centrale composée de trois cases initialement vides. La page victime apparaît en italique. La lettre D signale l'occurrence d'un défaut de page.

Remplacement FIFO

Avec cet algorithme, c'est la page la plus anciennement chargée qui est remplacée. Une implémentation de cet algorithme peut être réalisée en conservant dans la table des pages, la date de chargement de chaque page. La page choisie est alors celle pour laquelle la date de chargement est la plus ancienne.

La mise en œuvre de cet algorithme est simple, mais ses performances ne sont pas toujours bonnes. En effet, l'âge d'une page n'est pas le reflet de son utilité.

La figure 13.23 donne un exemple de mise en œuvre de cet algorithme.

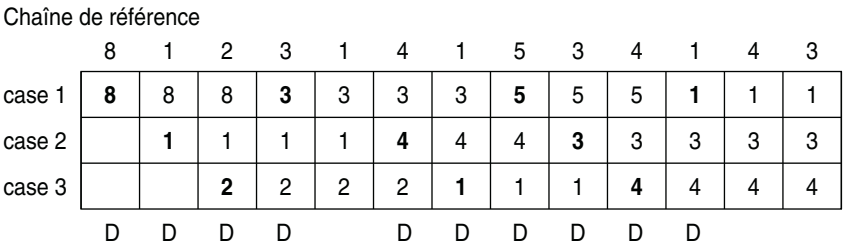


Figure 13.23 Remplacement FIFO.

Remplacement LRU

Avec cet algorithme, c'est la page la moins récemment utilisée qui est remplacée. Cette stratégie utilise le principe de la localité temporelle selon lequel les pages récemment utilisées sont celles qui seront référencées dans le futur. La page la moins récemment utilisée est donc jugée comme étant celle devenue la plus inutile.

Une implémentation de cet algorithme peut être réalisée en conservant dans la table des pages, la date de dernier accès de chaque page. La page choisie est alors celle pour laquelle la date d'accès est la plus ancienne. La figure 13.24 donne un exemple de mise en œuvre de cet algorithme.

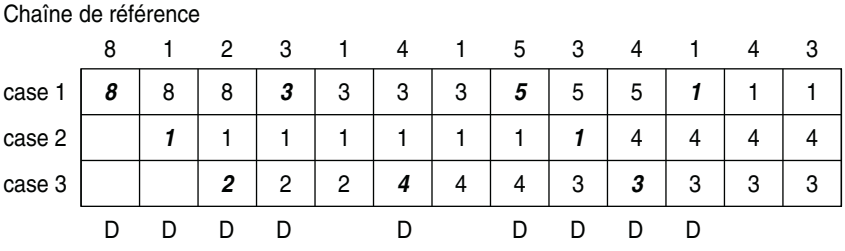


Figure 13.24 Remplacement LRU.

L'algorithme LRU est l'un des algorithmes les plus utilisés car il est considéré comme très bon. Cependant sa mise en œuvre est coûteuse et nécessite le recours à des compteurs matériels spécifiques pour la délivrance des dates d'accès aux pages.

Algorithme de la seconde chance

Un bit de référence est associé à chaque page dans la table des pages des processus. Ce bit de référence est mis à 1 à chaque référence à la page.

Le principe de l'algorithme de la seconde chance est le suivant :

– la page la plus anciennement chargée est sélectionnée comme étant a priori la page victime ;
– si la valeur du bit de référence est à 0, alors la page est effectivement remplacée ;
– si la valeur du bit de référence est au contraire à 1, alors une seconde chance est donnée à la page. Son bit de référence est remis à 0, et la page la plus anciennement chargée suivante est sélectionnée.

Cet algorithme constitue une bonne approximation de l'algorithme LRU. Il est beaucoup moins coûteux à implémenter car de nombreux matériels offrent directement un bit de référence associé à chaque case de la mémoire centrale.

Remplacement LFU et MFU

Avec l'algorithme LFU, c'est la page la moins fréquemment utilisée qui est remplacée. Cet algorithme présente un problème vis-à-vis des pages abondamment référencées sur un court laps de temps. La valeur du compteur pour ces pages étant élevée, elles ne sont pas retirées de la mémoire centrale, même si elles ne sont plus jamais référencées.

Avec l'algorithme MFU, c'est au contraire la page la plus fréquemment utilisée qui est remplacée. L'argument à la base de cet algorithme est qu'une page ayant un petit nombre de références vient sans doute d'être chargée en mémoire centrale et doit donc y rester.

L'implémentation de ces deux algorithmes est réalisée à l'aide d'un compteur associé à chaque entrée de table des pages qui mémorise le nombre de références à une page. Cependant, les performances de ces deux algorithmes ne sont pas très bonnes et ils sont rarement utilisés.

13.3.4 Performance

Les performances d'un système informatique peuvent être sensiblement affectées par le mécanisme de pagination à la demande, si le nombre de défauts de pages est trop important. En effet, dans un système utilisant la pagination à la demande, le temps d'accès effectif à un mot de la mémoire centrale augmente proportionnellement avec le nombre de défaut de pages : soit p, $0 \leq p \leq 1$ la probabilité d'un défaut de page, soit ta, le temps d'accès à un mot de la mémoire centrale sans défaut de page alors le temps effectif d'accès est $(1 - p) \times ta + p \times$ temps de défaut de page.

Le temps de défaut de page est surtout impacté par le temps nécessaire pour réaliser l'opération d'entrées-sorties permettant de charger la page manquante en mémoire centrale. Un temps moyen pour réaliser une telle opération d'entrées-sorties avec les technologies de disque actuelles est de 25 ms. Ainsi, le temps effectif d'accès devient, si l'on considère un temps d'accès à la mémoire centrale égal à 100 ns :

$$(1 - p) \times 100 + p \times 25\,000\,000 = 100 + 24\,999\,900 \times p$$

Pour obtenir une dégradation des performances inférieure à 10 %, il faut que la probabilité p soit inférieure à 0,000 000 4, c'est-à-dire que moins d'un accès mémoire sur 2 500 000 provoque un défaut de page.

13.3.5 Exemples

Le dérobeur de pages sous Linux

L'entrée d'une page dans une table des pages d'un processus Linux contient les champs suivants :

– un bit V de validation indique si la page est présente ou non en mémoire centrale ;
– un bit M de modification indique si la page a été modifiée en mémoire centrale ;
– le champ case contient l'adresse de la case physique contenant la page ;
– un champ accès indique si la page a été accédée ou non ;
– un compteur âge permet de mémoriser l'âge de la page en mémoire centrale.

Les champs accès et âge sont utilisés par le processus « dérobeur de pages » (processus kswapd) pour choisir des pages victimes. Le dérobeur de pages est réveillé dès que l'espace mémoire libre tombe sous une marque de limite inférieure. Le dérobeur de pages libère alors des cases jusqu'à ce que l'espace mémoire libre passe au-dessus d'une marque de limite supérieure. Les marques de limite inférieure et supérieure sont des paramètres système configurés par l'administrateur.

Une page est victime du dérobeur de pages si elle a atteint un âge donné (paramètre système) sans être référencée. Plus précisément :

– à chaque fois qu'une page est référencée, l'âge de la page devient égal à 0 et le bit référence est mis à vrai ;
– à chacun de ses passages, le dérobeur de pages met à faux le bit référence s'il est à vrai puis incrémente l'age de la page.

Une page est victime si son bit de référence est à faux et si elle a atteint l'âge limite. Le remplacement est de type global.

La mémoire virtuelle sous l'OS MVS/370

La gestion de la mémoire virtuelle pour le système d'exploitation MVS/370 est assez similaire à celle mise en œuvre sous Linux. Chaque page est caractérisée par un bit de référence qui est positionné par le matériel à chaque référence à la page et par un âge évoluant de 0 à 255 comptabilisé dans le compteur UIC (*unreferenced interval count*) d'une taille d'un octet.

À intervalle régulier, le système inspecte l'ensemble des pages présentes en mémoire centrale :

– si le bit de référence est positionné, celui-ci est remis à faux et l'UIC de la page revient à 0 ;
– si le bit de référence n'est pas positionné, l'UIC de la page est incrémenté d'une unité.

La page victime est celle pour laquelle la valeur de l'UIC est la plus grande, ce qui correspond à la page la moins récemment utilisée (LRU). Le remplacement est ici aussi de type global.

13.3.6 Notion d'écroulement

On appelle *écroulement*, une haute activité de pagination. Un processus s'écroule lorsqu'il passe plus de temps à paginer qu'à s'exécuter. La figure 13.25 illustre ce phénomène. Elle représente le taux d'utilisation du processeur en fonction du degré de multiprogrammation, c'est-à-dire en fonction du nombre de processus chargés en mémoire centrale.

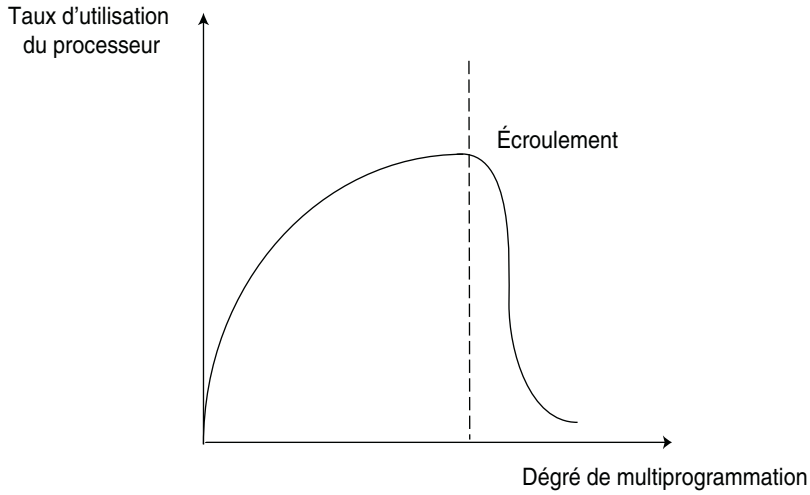


Figure 13.25 Écroulement.

Sur cette figure, on voit clairement que l'utilisation du processeur augmente jusqu'à un certain seuil au-delà duquel cette utilisation chute complètement. Cette chute correspond à une trop grande activité de pagination des processus qui passent le plus clair de leur temps en opérations d'entrées-sorties car ils n'ont pas suffisamment de cases mémoires disponibles pour contenir les pages relatives à leur espace de travail courant.

Le phénomène d'écroulement se produit surtout si une stratégie de remplacement global est mise en œuvre. En effet, avec une telle stratégie, un processus peut s'étendre en mémoire centrale au détriment des autres. Les autres processus qui voient leur nombre de pages diminuer en mémoire centrale vont commettre des défauts de pages et passer dans l'état bloqué, en attente de la fin de l'opération d'entrées-sorties leur permettant d'obtenir les pages souhaitées. Le nombre de processus prêts va donc diminuer et le système automatiquement va réintroduire de nouveaux processus en mémoire centrale pour maintenir le taux d'utilisation du processeur à un bon niveau. Ces nouveaux processus vont à leur tour provoquer des défauts de pages puisqu'ils vont demander le chargement en mémoire centrale des pages nécessaires au démarrage de leur exécution. L'effet précédent en est donc encore renforcé, entraînant petit à petit le système jusqu'à l'écroulement de ses performances.

13.4 SWAPPING DES PROCESSUS

L'opération de *vidage* (*swap-out*) d'un processus consiste à transférer sur disque dans la zone de swap, l'ensemble des pages présentes en mémoire centrale de l'espace d'adressage de ce processus. Le processus victime de l'opération de vidage est un processus endormi (état bloqué) en attente d'une ressource pour la poursuite de son exécution. L'intérêt de cette opération est de désengorger d'un seul coup la mémoire centrale de pages inutiles plutôt que de le faire petit à petit par vol de pages. Le vidage de processus peut être aussi déclenché en cas de situation de crise au niveau de l'occupation de la mémoire centrale. L'opération inverse de *swap-in* consiste à ramener les pages du processus depuis la zone de swap lorsque celui-ci redevient actif.

Sous le système Unix, le processus système 0, premier processus créé lors du boot du système, devient ensuite le processus swapper. Lorsqu'il est réveillé par le noyau, le processus swapper examine les processus présents en mémoire centrale et transfère ceux qui sont endormis depuis le plus longtemps. Par ailleurs, il examine également les processus transférés dans la zone de swap et remonte en mémoire centrale ceux qui sont devenus prêts et pour lesquels il existe suffisamment d'espace en mémoire centrale.

13.5 CONCLUSION

L'allocation en partitions variables considère le programme comme un ensemble d'adresses insécables. Ce type d'allocation pose un problème de fragmentation et nécessite des opérations de compactage de la mémoire centrale.

La pagination découpe l'espace d'adressage du programme en pages et la mémoire physique en cases de même taille. Une adresse générée par le processeur est de la forme <numéro de page, déplacement dans la page>. La table des pages du processus permet de traduire l'adresse paginée en adresse physique.

La segmentation découpe l'espace d'adressage du programme en segments correspondant à des morceaux logiques du programme. Une adresse générée par le processeur est de la forme <numéro de segment, déplacement dans le segment>. La table des segments du processus permet de traduire l'adresse segmentée en adresse physique.

Remarque : Segmentation et pagination sont très souvent associées.

Lorsque le principe de la mémoire virtuelle est appliqué, les pages d'un processus ne sont chargées en mémoire centrale que lorsque le processus y accède. Lorsqu'un processus accède à une page non présente en mémoire centrale, il se produit un défaut de page. La page manquante est alors chargée dans une case libre. Si aucune case n'est libre, le système utilise un algorithme de remplacement de pages pour choisir une case à libérer.

L'écroulement est la situation pour laquelle un ou plusieurs processus passent plus de temps à paginer qu'à s'exécuter.

Chapitre **14**

Système de gestion de fichiers

La mémoire centrale de l'ordinateur est une mémoire volatile, c'est-à-dire que son contenu s'efface lorsque l'alimentation électrique de l'ordinateur est interrompue. Cependant, les programmes et les données stockés dans la mémoire centrale et en premier lieu le code et les données même du système d'exploitation, ont besoin d'être conservés au-delà de cette éventuelle coupure : c'est le rôle rempli par le *système de gestion de fichiers* qui assure la conservation des données sur un support de masse non volatile, tel que le disque dur, la disquette, le CD-ROM ou encore la bande magnétique. Le système d'exploitation offre à l'utilisateur une unité de stockage indépendante des propriétés physiques des supports de conservation : *le fichier.* Ce concept de fichier recouvre deux niveaux. D'une part, le *fichier logique* représente l'ensemble des données incluses dans le fichier telles qu'elles sont vues par l'utilisateur. D'autre part, *le fichier physiqu*e représente le fichier tel qu'il est alloué physiquement sur le support de masse. Le système d'exploitation gère ces deux niveaux de fichiers et assure notamment la correspondance entre eux, en utilisant une structure de *répertoire*.

14.1 LE FICHIER LOGIQUE

14.1.1 Définition

Le fichier logique correspond à la vue que l'utilisateur de la machine a de la conservation de ses données. Plus précisément, le fichier logique est :

– d'une part, un type de données standard défini dans les langages de programmation, sur lequel un certain nombre d'opérations spécifiques peuvent être réalisées.

Ces sont les opérations de création, ouverture, fermeture et destruction de fichier. Dans le programme, le fichier est identifié par un nom. Les opérations de création ou d'ouverture du fichier logique effectuent la liaison du fichier logique avec le fichier physique correspondant sur le support de masse. Au contraire, les opérations de fermeture et de destruction rompent cette liaison ;

– d'autre part, un ensemble d'*enregistrements* ou *articles.* Un enregistrement est un type de données regroupant des données de type divers liées entre elles par une certaine sémantique inhérente au programme qui les manipule. L'enregistrement constitue pour le programme une unité logique de traitement. Les enregistrements du fichier logique sont accessibles par des opérations spécifiques de lecture ou d'écriture que l'on appelle les *fonctions d'accès*. Différents modes d'accès aux enregistrements peuvent être définis. Les principaux sont le mode *séquentiel*, le mode *indexé* et le mode *direct*.

Exemple

Dans le langage Pascal, le type de données représentant un fichier est le type `file of`. Un enregistrement est défini comme une structure de type `record`. Un fichier est un ensemble d'enregistrements.

Définition de l'enregistrement `element` comportant 3 champs `classe`, `professeur` et `nb_eleves`.

```
type element = record
    classe : string [12];
    professeur : string [25];
    nb_eleves : integer;
end;
```

Définition du fichier de nom `nom_fich` comme étant un ensemble d'élément `element`.

```
nom_fich : file of element;
```

14.1.2 Les modes d'accès

Les modes d'accès définissent la manière dont les enregistrements composant un fichier sont accessibles. Le nombre et le type de modes d'accès disponibles sont fonction du système d'exploitation. Les modes d'accès les plus courants sont l'accès séquentiel, l'accès indexé et l'accès direct. Ils définissent la sémantique des opérations de lecture et d'écriture d'enregistrements au niveau du fichier. Selon le mode d'accès associé à un fichier, celui-ci peut être accessible en lecture seule, en écriture seule ou en lecture et écriture simultanée.

Accès séquentiel

Le mode d'accès séquentiel traite les enregistrements d'un fichier dans l'ordre où ils se trouvent dans ce fichier, c'est-à-dire les uns à la suite des autres. Dans ce contexte, une opération de lecture d'un enregistrement délivre l'enregistrement courant et se

positionne sur le suivant. Ainsi, lire le troisième enregistrement du fichier nécessite au préalable la lecture du premier enregistrement ainsi que du second. L'opération d'écriture d'un nouvel enregistrement place obligatoirement cet enregistrement en fin de fichier. Ce mode d'accès est très simple et il découle naturellement du modèle de la bande magnétique. Avec ce mode, un fichier est soit accessible en lecture seule, soit en écriture seule.

Accès indexé

Le mode d'accès indexé, encore appelé accès aléatoire, permet d'accéder directement à un enregistrement quelle que soit sa position dans le fichier. Un champ commun à tous les enregistrements sert de clé d'accès. Une structure d'accès (l'index) ajoutée aux données du fichier permet de retrouver un enregistrement en fonction de la valeur du champ d'accès.

Dans la fonction de lecture, la valeur du champ d'accès recherché est spécifiée et l'opération de lecture délivre directement l'enregistrement pour lequel la clé correspond. La fonction d'écriture place le nouvel enregistrement en fin de fichier et met à jour la structure de données permettant de le retrouver ultérieurement. Une opération de modification d'un enregistrement peut être disponible et permet de réécrire un enregistrement préalablement lu depuis le fichier sans avoir à le rechercher de nouveau.

Avec ce mode, un fichier est soit accessible en lecture seule, soit en écriture seule, soit tout à la fois en lecture et en écriture.

Accès direct

Dans le mode d'accès direct, encore appelé accès relatif, l'accès à un enregistrement se fait en spécifiant sa position relative par rapport au début du fichier. Ce mode d'accès constitue un cas particulier de l'accès aléatoire pour lequel la clé d'accès est la position de l'enregistrement dans le fichier.

Modes d'accès du système MVS

La méthode d'accès VSAM (*Virtual Storage Access Method*) est une méthode d'accès générale, qui sur les systèmes MVS 370, remplace et unifie les anciennes méthodes d'accès IBM (QSAM, BSAM, ISAM, BDAM…). La méthode d'accès VSAM gère en fait 4 modes d'accès distincts :

– le mode KSDS (*key-sequenced data set*) est un mode d'accès de type indexé. Dans cette organisation, les enregistrements sont repérés par une clé et classés physiquement par ordre de clés croissantes. L'accès à un enregistrement passe par un index qui indique la position de l'enregistrement en fonction de la clé fournie ;
– le mode ESDS (*entry_sequenced data set*) est un mode d'accès de type séquentiel, sans clé ;
– le mode RRDS (*relative record data set*) est un mode d'accès de type relatif ou direct. Un enregistrement est accédé en fournissant son numéro d'ordre dans le fichier ;

– le mode LDS (*linear data set*) est un mode d'accès qui efface la notion d'enregistrement. Le fichier se présente simplement comme une suite d'octets découpés en paquets d'une taille de 4 096 octets. Cette organisation est similaire à celle mise en œuvre par le système Unix.

Parallèlement à la méthode d'accès VSAM, le système 370 gère un autre mode d'accès situé à mi-chemin entre le mode séquentiel et le mode direct. Dans cette organisation appelée *organisation partitionnée*, le fichier est découpé en membres, stockés les uns derrière les autres, selon leur ordre de création. Un index placé en début de fichier pointe sur chacun des membres. Les enregistrements placés dans un membre sont ensuite accédés de manière séquentielle.

14.1.3 Exemples

Manipulation de fichiers avec le langage Ada

Le langage Ada permet la définition de fichiers accessibles en mode séquentiel et en mode direct.

Voici un premier programme illustrant la définition et la manipulation de deux fichiers A et B, définis comme étant à accès séquentiel. Les fichiers contiennent des enregistrements de type `T_ENRG`, composés de deux champs `nom_eleve` et `note`. Les opérations `OPEN` et `CREATE` associent les fichiers logiques A et B à un fichier physique, respectivement `Notes_Eleves` qui est ouvert en mode lecture (`IN_FILE`) et `Copie_Eleves` qui est ouvert en mode écriture (`OUT_FILE`). Les enregistrements lus dans le fichier A en mode séquentiel sont écrits dans le fichier B.

```
Fichier séquentiel
type T_ENRG is     — définition de l'enregistrement
record
    nom_eleve : string[30];
    note : integer
end record;
— instanciation du paquetage SEQUENTIAL_IO permettant l'utilisation du mode
➥ séquentiel. Le paquetage contient les fonctions d'accès.
with SEQUENTIAL_IO;
package T_IO is new SEQUENTIAL_IO(ELEMENT_TYPE = > T_ENRG);
— déclaration de deux fichiers A et B de type séquentiel
A, B : T_IO.FILE_TYPE; ENRG : T_ENRG;
BEGIN
— ouverture du fichier A en mode lecture seule (IN_FILE)
T_IO.OPEN(A, IN_FILE, "Notes_Eleves");
— ouverture du fichier B en mode écriture seule (OUT_FILE)
T_IO.CREATE(B, OUT_FILE, "Copie_Eleves");
— lecture d'un enregistrement dans A et copie dans B jusqu'à atteindre
➥ la fin de fichier pour A
```

```
WHILE (NOT T_IO.END_OF_FILE(A))
    T_IO.READ(A, ENRG);
    T_IO.WRITE(B, ENRG);
END;
— fermeture des fichiers
T_IO.CLOSE(A);
T_IO.CLOSE(B);
END;
```

Le second programme illustre la définition et la manipulation de trois fichiers A, B et C, définis comme étant à accès direct. Les ordres de lecture (READ) et écriture (WRITE) spécifient le numéro relatif de l'enregistrement concerné.

```
Fichier en mode direct
type T_ENRG is
record
    nom_eleve : string[30];
    note : integer
end record;
— instanciation du paquetage DIRECT_IO permettant l'utilisation du mode
➥ direct. le paquetage contient les fonctions d'accès.
with DIRECT_IO;
package T_IO is new DIRECT_IO(ELEMENT_TYPE = > T_ENRG)
A, B, C : T_IO.FILE_TYPE; ENRG : T_ENRG;
BEGIN
— ouverture du fichier A en mode lecture seule (IN_FILE)
T_IO.OPEN(A, IN_FILE, "Notes_Eleves");
— ouverture du fichier B en mode écriture seule (OUT_FILE)
T_IO.OPEN(B, OUT_FILE, "Copie_Eleves");
— ouverture du fichier C en mode lecture/écriture (INOUT_FILE)
T_IO.OPEN(C, INOUT_FILE, "Modif_Eleves");
— lecture dans le fichier A de l'enregistrement numéro 2 du fichier
T_IO.READ(A, ENRG, 2);
— écriture dans le fichier B de l'enregistrement numéro 2 du fichier A
T_IO.WRITE(B, ENRG);
— lecture dans le fichier C de l'enregistrement numéro 4 du fichier
T_IO.READ(C, ENRG, 4);
ENRG.note : = 0;
— écriture dans le fichier C de l'enregistrement numéro 4 du fichier
➥ après modification
T_IO.WRITE(C, ENRG, 4);
T_IO.CLOSE(A);
T_IO.CLOSE(B); T_IO.CLOSE(C);
END;
```

Manipulation de fichiers avec le langage Cobol

Le programme suivant illustre la déclaration de deux fichiers FIC1 et FIC2, en langage Cobol, sous un système MVS 370. Les deux fichiers utilisent la méthode d'accès VSAM. Le premier fichier FIC1 est un fichier séquentiel. Le second fichier FIC2 est un fichier avec une organisation de type séquentielle indexée. Dans ce fichier FIC2, un enregistrement est composé de deux champs dont le premier sur 5 caractères correspond à la clé. L'exécution de ce programme Cobol est lancée par l'intermédiaire d'un job écrit en JCL qui effectue la liaison des fichiers logiques FIC1 et FIC2 avec le fichier physique correspondant.

```
000100   IDENTIFICATION DIVISION.
000200   PROGRAM-ID VSAM.
000300   ENVIRONMENT DIVISION.
000400   INPUT-OUTPUT SECTION.
000500   FILE-CONTROL.
000600        SELECT FIC1 ASSIGN UNIT1.
000700        SELECT FIC2 ASSIGN UNIT2
000800             ORGANIZATION INDEXED
000900             ACCESS DYNAMIC
001000             RECORD KEY IS FIC2-ENR1-CLE.
001100             FILE STATUS IS F-S.
001200   DATA DIVISION.
001300   FILE SECTION.
001400   FD FIC1
001500        RECORDING MODE f
001600        LABEL RECORD STANDARD
001700        DATA RECORD IS FIC1-ENR.
001800   01   FIC1-ENR.
001900        02   FIC1-ENR0              PIC X(80).
002000        02   FIC1-ENR1 REDEFINES    FIC1-ENR0.
002100             03   FIC1-ENR1-CHP1    PIC 9(5).
002200             03   FILLER            PIC X(75).
002300   FD   FIC2
002400        RECORDING MODE f
002500        LABEL RECORD STANDARD
002600        DATA RECORD IS FIC2-ENR.
002700   01   FIC2-ENR.
002800        02   FIC2-ENR0              PIC X(80).
002900        02   FIC2-ENR1 REDEFINES    FIC2-ENR0.
003000             03   FIC2-ENR1-CLE     PIC X(5).
003100             03   FIC2-ENR1-LIB     PIC X(75).
```

14.2 LE FICHIER PHYSIQUE

Le fichier physique correspond à l'entité allouée sur le support permanent et contient physiquement les enregistrements définis dans le fichier logique. Nous prendrons ici comme support de mémoire secondaire le disque dur.

14.2.1 Structure du disque dur

Un disque est constitué d'un ensemble de *plateaux* (jusqu'à 20), empilés verticalement sur un même axe, formant ainsi ce que l'on appelle une *pile* de disques. Chaque plateau est composé d'une ou deux *faces*, divisées en *pistes* qui sont des cercles concentriques. Selon les modèles, le nombre de pistes par face varie de 10 à plus de 1 000. Chaque piste est elle-même divisée en secteurs (de 4 à 32 secteurs par piste), le *secteur* constituant la plus petite unité de lecture/écriture sur le disque. La taille d'un secteur varie généralement entre 32 à 4 096 octets, avec une valeur courante de 512 octets. On parle également de *cylindre*, celui-ci étant constitué de l'ensemble des pistes de même rayon sur l'ensemble des plateaux. L'opération consistant à créer à partir d'un disque vierge, l'ensemble des pistes et des secteurs s'appelle *l'opération de formatage*.

L'accès à une face d'un disque s'effectue par l'intermédiaire d'un bras, supportant une tête de lecture/écriture. Le bras avance ou recule pour se positionner sur la piste à lire/écrire tandis que le disque tourne. Une adresse sur le disque est donc de la forme (face ou bras, piste ou cylindre, secteur).

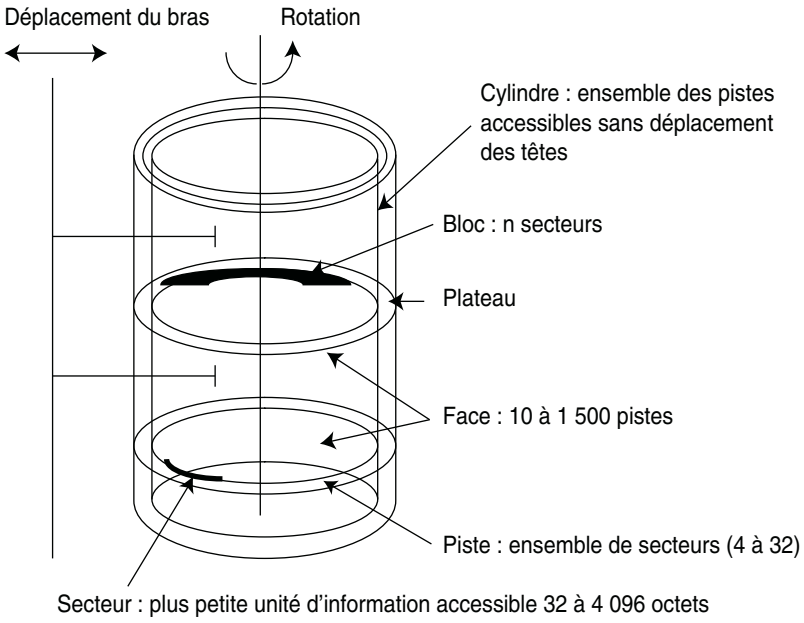


Figure 14.1 Structure du disque dur.

La plus petite unité accessible physiquement sur le disque est donc le secteur. Pour optimiser les opérations de lecture et écriture, les secteurs sont regroupés en *bloc*. Un bloc est constitué de plusieurs secteurs et constitue la plus petite unité d'échange entre le disque et la mémoire centrale. Sa taille dépend du périphérique d'entrées-sorties et est fixée par le matériel. La figure 14.1 illustre la structure d'un disque.

14.2.2 Méthodes d'allocation de la mémoire secondaire

Les enregistrements composant le fichier logique doivent être écrits dans les secteurs composant les blocs du disque, pour former ainsi le fichier physique correspondant au fichier logique.

Le fichier physique est donc constitué d'un ensemble de blocs physiques qui doivent être alloués au fichier. Différentes méthodes d'allocation de la mémoire secondaire ont été définies qui se rapprochent grandement des méthodes d'allocation de la mémoire centrale évoquées au chapitre précédent. Ce sont principalement les méthodes de l'allocation contiguë, par zones, par blocs chaînés ou indexée.

Par ailleurs, pour pouvoir allouer des blocs aux fichiers, il faut connaître à tout moment l'ensemble des blocs libres et donc gérer l'espace libre sur le disque.

Allocation contiguë

Cette allocation exige qu'un même fichier occupe un ensemble de blocs physiques contigus. Cette méthode a l'avantage de la simplicité mais elle demande à ce que lors de la création d'un fichier la taille finale de celui-ci soit évaluée afin qu'un espace suffisant puisse être réservé. Il en découle souvent une sous-utilisation de l'espace disque.

L'allocation d'un fichier requiert de trouver un espace libre sur le disque suffisamment grand c'est-à-dire dont le nombre de blocs est au moins égal au nombre de blocs du fichier. Les méthodes *First Fit*, *Best Fit* et *Worst Fit* évoquées lors du chapitre précédent à propos de l'allocation de la mémoire centrale par partitions variables sont appliquées également ici pour choisir l'espace libre à allouer. Ainsi, sur la figure 14.2, l'allocation du fichier fich_5 d'une taille évaluée maximale à 4 blocs attribuera les blocs libres 4, 5, 6 et 7 dans le cas d'une stratégie de type *First Fit*, les blocs libres 13, 14, 15 et 16 dans le cas d'une stratégie *Best Fit* et les blocs libres 21, 22, 23 et 24 dans le cas d'une stratégie *Worst Fit*.

Il découle de cette méthode d'allocation ces mêmes problèmes de fragmentation qui se traduisent par une difficulté croissante pour trouver des espaces libres suffisants à une nouvelle allocation. Il faut alors avoir recours à une opération de compactage pour regrouper les espaces libres dispersés et insuffisants en un seul espace libre exploitable. Cette opération est évidemment très coûteuse puisqu'elle nécessite le déplacement des blocs du disque, c'est-à-dire la lecture de chaque bloc et sa réécriture.

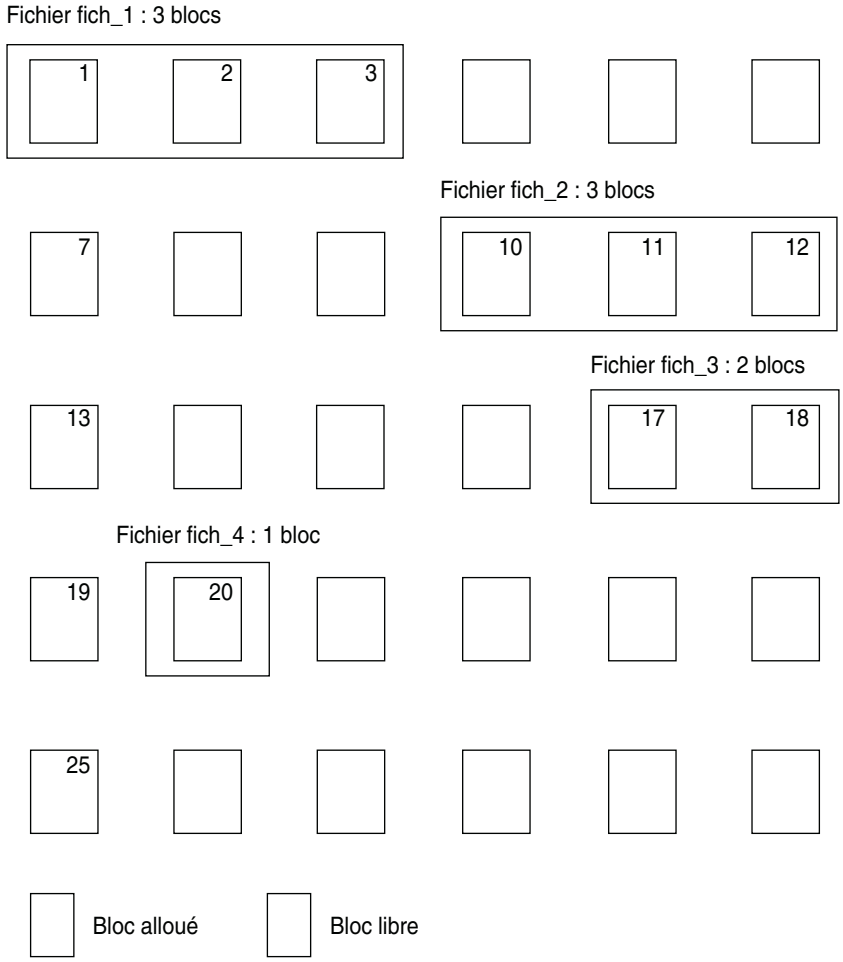


Figure 14.2 Allocation contiguë.

L'opération de compactage du disque est réalisée sur les systèmes de type DOS et Windows par le biais de l'opération de défragmentation.

Une dernière difficulté est celle de l'extension d'un fichier si les blocs disques voisins ne sont pas libres. Ainsi, sur la figure 14.3, l'utilisateur souhaite étendre le fichier fich_1 avec un nouveau bloc. Cependant, le bloc voisin au dernier bloc de fich_1 est alloué au fichier fich_5. Une solution peut être alors de déplacer le fichier déjà existant dans une zone libre plus importante autorisant de ce fait son extension, mais cette solution est excessivement coûteuse. La solution plus généralement utilisée dans ce cas est de générer une erreur.

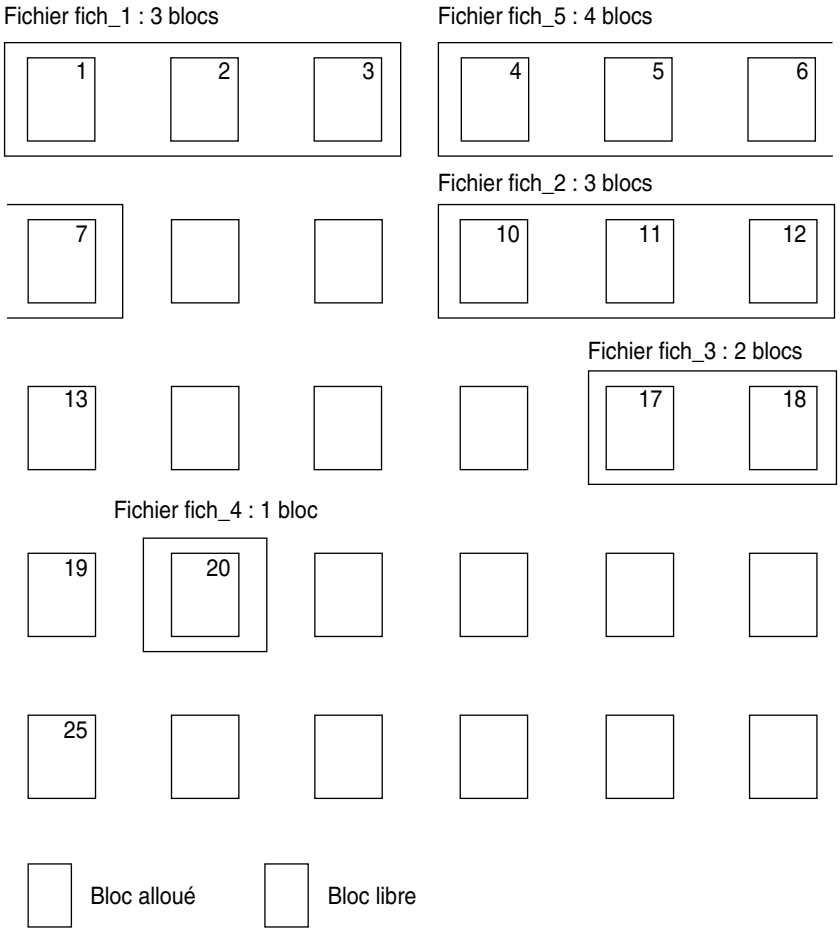


Figure 14.3 Difficultés de l'allocation contiguë.

Cette méthode d'allocation supporte les modes d'accès séquentiel et direct. Par ailleurs, elle offre de bonnes performances pour l'accès aux différents blocs d'un même fichier, car leur contiguïté minimise le nombre de repositionnements nécessaires des têtes de lecture/écriture. Cependant, elle aboutit à une mauvaise utilisation de l'espace disque qui fait qu'elle tend de plus en plus à être abandonnée.

Allocation par zones

La méthode d'allocation par zones constitue une variante de l'allocation contiguë, qui autorise un fichier à être constitué de plusieurs zones physiques distinctes. Une zone doit être allouée dans un ensemble de blocs contigus mais les différentes zones composant le fichier peuvent être dispersées sur le support de masse. Il en résulte une extension plus facile d'un fichier. Lors de la création du fichier, la première zone appelée *zone primaire* est allouée et les zones suivantes appelées *zones secondaires*

sont allouées au fur et à mesure de l'extension du fichier. La taille de la zone primaire et celle des zones secondaires sont généralement définies au moment de la création du fichier. Le nombre de zones secondaires autorisées pour un fichier est souvent limité. La figure 14.4 illustre cette méthode.

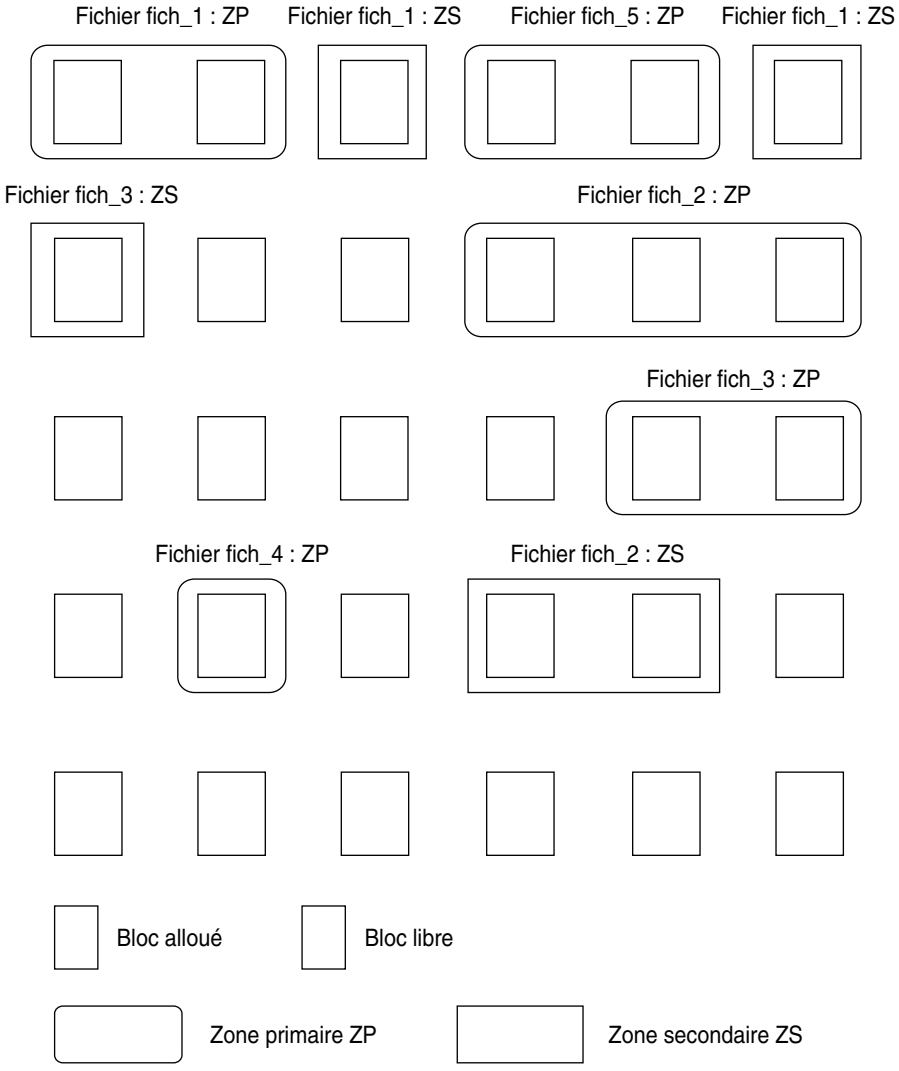


Figure 14.4 Allocation par zones.

Cette méthode d'allocation est celle mise en œuvre sur les systèmes IBM de type MVS. Dans ce système, un fichier est constitué par une zone primaire d'une taille P et d'au plus 15 zones secondaires d'une taille S. Les tailles P et S peuvent être explicitées en terme de pistes (environ 50 Ko), de cylindres (environ 15 pistes) ou de

blocs (4 Ko ou plus). Voici l'exemple d'un travail batch écrit en JCL permettant la création d'un fichier logique nommé FICHIER lié au fichier physique de nom DSE25.OUT. Le paramètre SPACE = (TRK, (5, 1)) indique que le fichier est composé d'une zone primaire d'une taille de 5 pistes (TRK) et de zones secondaires dont la taille est d'une piste.

```
000001 //DSE25BP  JOB (A99, DSE), 'ALLOC-DSN', CLASS = D, MSGCLASS = X,
000002 //                  NOTIFY = DSE25, MSGLEVEL = (1, 1)
000003 //*
000004 //*-------------------------------------------------------
000005 //* ALLOCATION DATASET
000006 //*-------------------------------------------------------
000007 //*
000008 //STEP1     EXEC PGM = IEFBR14
000009 //* fichier physique DSE25.OUT
000010 //* disposition (DISP) : création (NEW), inscription
000011 //* au répertoire en fin de job (CATLG), destruction du fichier
000012 //* si probleme (DELETE)
000013 //* données du fichier (DCB) : longueur enregistrement
000014 //* de taille fixe (132)
000015 //FICHIER   DD DSN = DSE25.OUT, SPACE = (TRK, (5, 1)),
000016 //          DISP = (NEW, CATLG, DELETE), UNIT = ARTAROT1,
000017 //          DCB = (LRECL = 132, RECFM = FB, BLKSIZE = 0)
000018 //
```

La méthode d'allocation par zones diminue les performances d'accès aux blocs d'un même fichier puisque le passage d'une zone à un autre non contiguë peut nécessiter un repositionnement important du bras. Elle minimise par ailleurs les problèmes de fragmentation externe sans cependant les résoudre complètement. Ainsi, l'utilitaire de gestion du disque de MVS, DFDSS (*data facility dataset services*), offre entre autres une commande de compactage DEFRAG.

Allocation par blocs chaînés

Dans cette méthode, un fichier est constitué comme une liste chaînée de blocs physiques, qui peuvent être dispersés n'importe où sur le support de masse. Chaque bloc contient donc l'adresse du bloc suivant dans le fichier.

Étendre un fichier est alors simple : il suffit d'allouer un nouveau bloc physique puis de le chaîner au dernier bloc physique du fichier. Ainsi, il n'est plus nécessaire de connaître à la création du fichier, la taille maximale de celui-ci. Par ailleurs, les problèmes de fragmentation externe disparaissent.

Cette méthode d'allocation souffre malheureusement de deux inconvénients. Le premier est que le seul mode d'accès utilisable est le mode d'accès séquentiel. En effet, accéder à un enregistrement donné du fichier nécessite de parcourir la liste chaînée des blocs constituant ce fichier, donc de lire les uns à la suite des autres

chacun des blocs du fichier pour connaître l'adresse du suivant. Le second inconvénient est la place occupée dans chaque bloc par le chaînage de la liste, fonction de la taille des adresses physiques des blocs. La figure 14.5 illustre cette méthode.

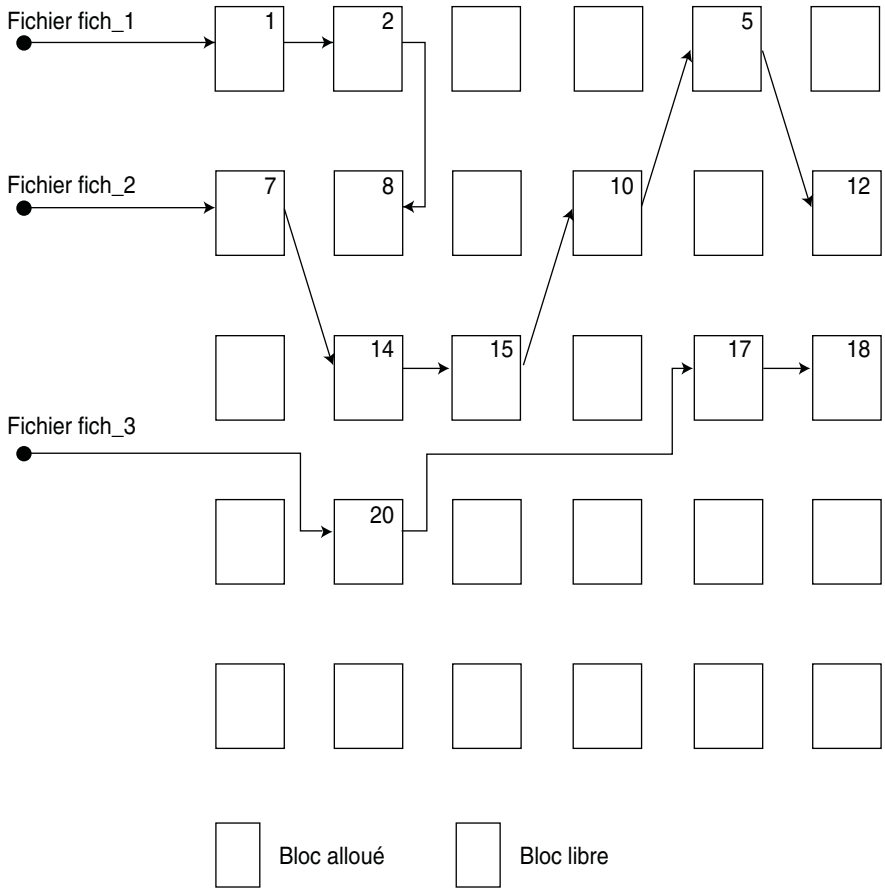


Figure 14.5 Allocation par blocs chaînés.

Une variante de l'allocation par blocs chaînés est la solution retenue par les systèmes DOS/Windows. L'ensemble des chaînages des blocs (*cluster*s) de fichiers est regroupé dans une table appelée FAT (*File Allocation Table*). Cette table est composée d'autant d'entrées qu'il y a de blocs de données sur le disque. Chaque entrée de la table correspond à un bloc du disque et contient :

– si le bloc appartient à un fichier et n'est pas le dernier bloc de ce fichier, le numéro du bloc suivant du fichier ;
– si le bloc appartient à un fichier et est le dernier bloc de ce fichier, une valeur de fin de fichier ($FF8_{16}$) ;
– si le bloc n'appartient pas à un fichier, une valeur de bloc libre (000_{16}).

Ainsi la figure 14.6 donne la FAT correspondant à l'allocation chaînée de la figure 14.5.

On distingue à l'heure actuelle deux systèmes de FAT, la FAT 16 pour laquelle le numéro de bloc est codé sur 16 bits autorisant ainsi 65 536 blocs sur le disque et la FAT 32 pour laquelle le numéro de bloc est codé sur 32 bits autorisant ainsi 2 Gigablocs sur le disque. Notons que ce système a été conçu à l'origine pour gérer l'allocation sur des disquettes de faible capacité et tend aujourd'hui à montrer ses limites du fait de l'augmentation de la capacité des disques. Il est aujourd'hui remplacé par le système NTFS (*NT File Systems*).

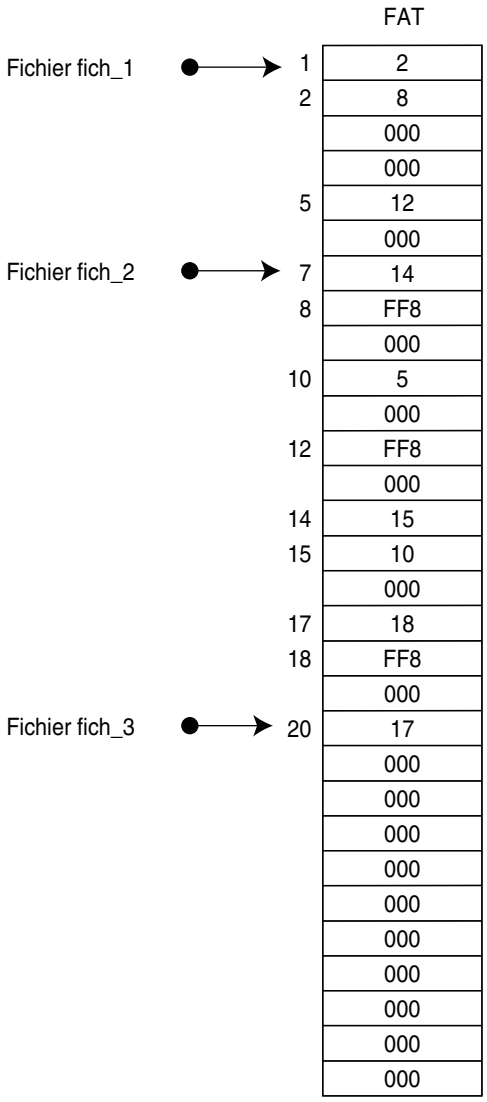


Figure 14.6 Table d'allocation Windows (FAT).

Allocation indexée

L'allocation indexée vise à supprimer les deux inconvénients de la méthode d'allocation chaînée. Dans cette méthode, toutes les adresses des blocs physiques constituant un fichier sont rangées dans une table appelée index, elle-même contenue dans un bloc du disque. À la création d'un fichier, l'index est créé avec toutes ses entrées initialisées à vide et celles-ci sont mises à jour au fur et à mesure de l'allocation des blocs au fichier. Ce regroupement de toutes les adresses des blocs constituant un fichier dans une même table permet de réaliser des accès directs à chacun de ces blocs. Par ailleurs, les blocs alloués au fichier contiennent à présent exclusivement des données du fichier. La figure 14.7 illustre cette méthode.

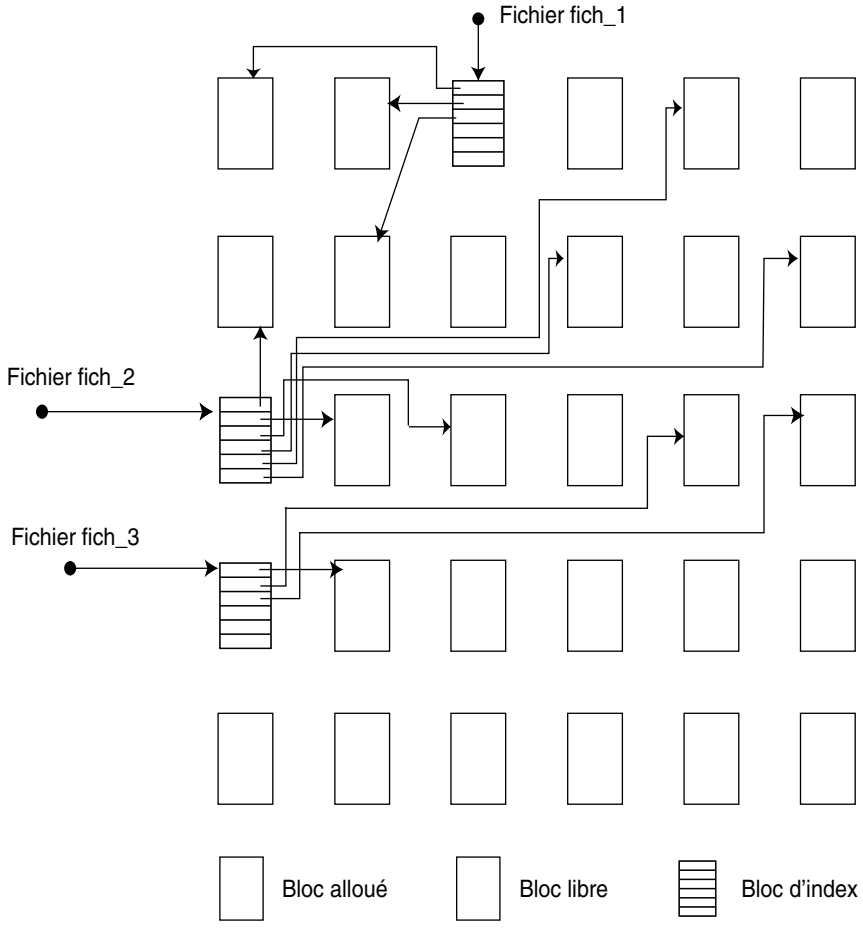


Figure 14.7 Méthode d'allocation indexée.

Un problème est inhérent cependant à la taille de la table d'index qui est conditionnée par celle d'un bloc physique et par le nombre de blocs existants sur le disque. Si le bloc est grand, le nombre d'entrées utilisées dans le bloc d'index peut

être faible ce qui conduit à un problème de fragmentation interne et à une perte de place, notamment en ce qui concerne l'allocation des fichiers de petite taille. Au contraire, le nombre d'entrées disponibles dans le bloc d'index peut se révéler être insuffisant vis-à-vis du nombre de blocs requis pour le fichier.

Dans ce dernier cas, une solution est d'utiliser un index multiniveaux. Le premier bloc d'index ne contient pas directement des adresses de blocs de données, mais il contient des adresses de blocs d'index, qui eux contiennent les adresses des blocs de données du fichier. Ainsi, pour lire un bloc de données d'un fichier, le système doit d'abord lire le premier bloc d'index, puis le bloc d'index de second niveau avant de lire le bloc de données lui-même. Il s'ensuit une évidente perte de performances qui s'accroît encore si ce schéma est étendu à 3 ou 4 niveaux (figure 14.8).

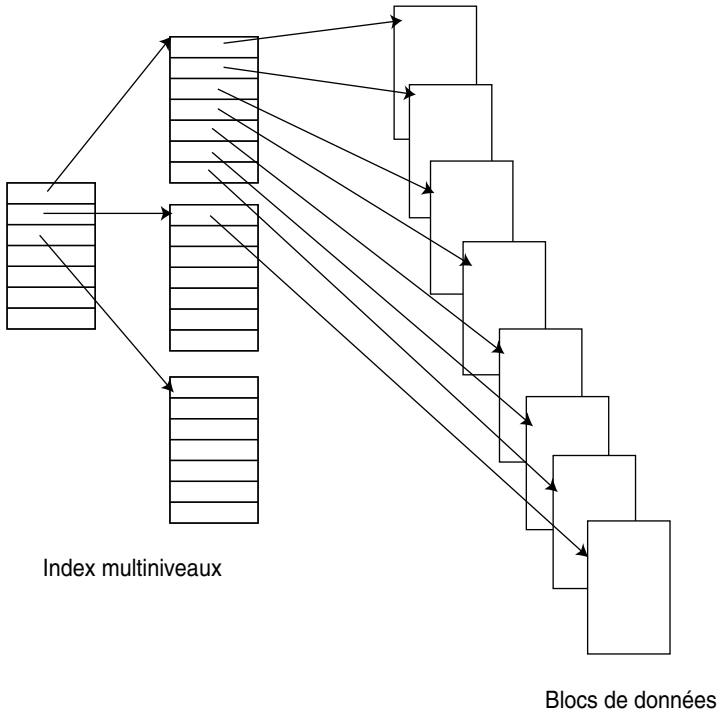


Figure 14.8 Index multiniveaux.

Cette méthode d'allocation indexée est mise en œuvre pour l'allocation de fichiers dans le système Unix. Dans ces systèmes, un premier index composé de 13 entrées est maintenu dans le descripteur du fichier. Les 10 premières entrées de cet index contiennent les adresses des 10 premiers blocs de données du fichier. La 11^{e} entrée par contre pointe, elle, sur un bloc d'index, qui contient les adresses des p (p = taille bloc en octets/taille en octets d'une adresse de bloc) blocs de données suivants du fichier, introduisant ainsi un premier niveau d'indirection dans l'index du fichier. La 12^{e} entrée de la table engendre un niveau d'indirection supplémentaire. Elle pointe

également sur un bloc d'index, qui contient les adresses de p blocs d'index dont les entrées pointent sur les p^2 blocs de données suivants du fichier. Enfin, la dernière entrée de la table ajoute encore un niveau d'index supplémentaire (figure 14.9).

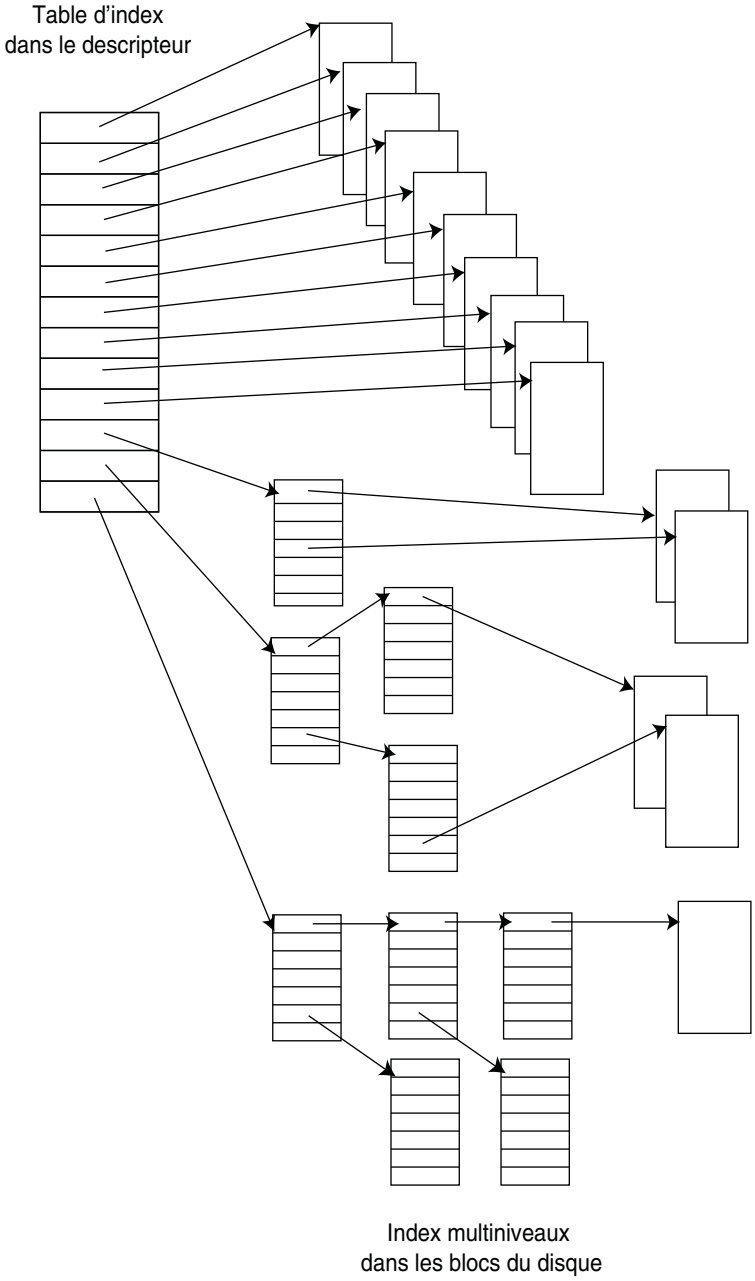


Figure 14.9 Index dans le système Unix.

Ainsi, si l'on considère des blocs de données de 1 024 octets et des adresses de blocs de 4 octets, la taille maximale d'un fichier est de 10 + 256 + 65 536 + 16 777 216 blocs. Chaque niveau d'indirection additionnel nécessite un accès disque supplémentaire pour obtenir un bloc de données. Ainsi, un petit fichier d'au plus 10 blocs a ses blocs de données accessibles en une seule opération d'entrées-sorties. Par contre, un très gros fichier dont la taille est supérieure à 10 + 256 + 65 536 blocs demande 4 accès disque pour obtenir un bloc situé au-delà du 65 803^{e} bloc. Aussi, un mécanisme de cache (*buffer cache d'Unix*) qui conserve en mémoire centrale les blocs disque les plus récemment accédés est mis en œuvre pour limiter les accès au disque.

Gestion de l'espace libre

Le système maintient une liste d'espace libre, qui mémorise tous les blocs disque libres, c'est-à-dire les blocs non alloués à un fichier.

Lors de la création ou de l'extension d'un fichier, le système recherche dans la liste d'espace libre la quantité requise d'espace et l'alloue au fichier. L'espace alloué est supprimé de la liste.

Lors de la destruction d'un fichier, l'espace libéré est intégré à la liste d'espace libre.

Il existe différentes représentations possibles de l'espace libre. Les principales sont la représentation de l'espace libre sous forme d'un vecteur de bits et la représentation de l'espace libre sous forme d'une liste chaînée des blocs libres.

Le système de la FAT évoqué plus haut, intègre directement la gestion de l'espace libre sur le disque, sans mécanisme additionnel.

➤ Gestion de l'espace libre par un vecteur de bits

Dans cette méthode, l'espace libre sur le disque est représenté par un vecteur binaire dans lequel chaque bloc est figuré par un bit. La longueur de la chaîne binaire est donc égale au nombre de blocs existants sur le disque. Dans cette chaîne, un bit à 0 indique que le bloc correspondant est libre. Au contraire, un bit à 1 indique que le bloc correspondant est alloué. Ainsi, le vecteur de bits correspondant à l'état du disque de la figure 14.5 est 11001011010101101101000000000.

➤ Gestion de l'espace libre par liste chaînée

Dans cette méthode, l'espace libre sur le disque est représenté par une liste chaînée de l'ensemble des blocs libres du disque. Avec cette méthode, la recherche sur disque de n blocs consécutifs peut nécessiter le parcours d'une grande partie de la liste chaînée. Une variante de la méthode consiste à indiquer pour chaque premier bloc d'une zone libre, le nombre de blocs libres constituant la zone, puis l'adresse du premier bloc de la zone libre suivante. La figure 14.10 illustre cette méthode de représentation de l'espace libre.

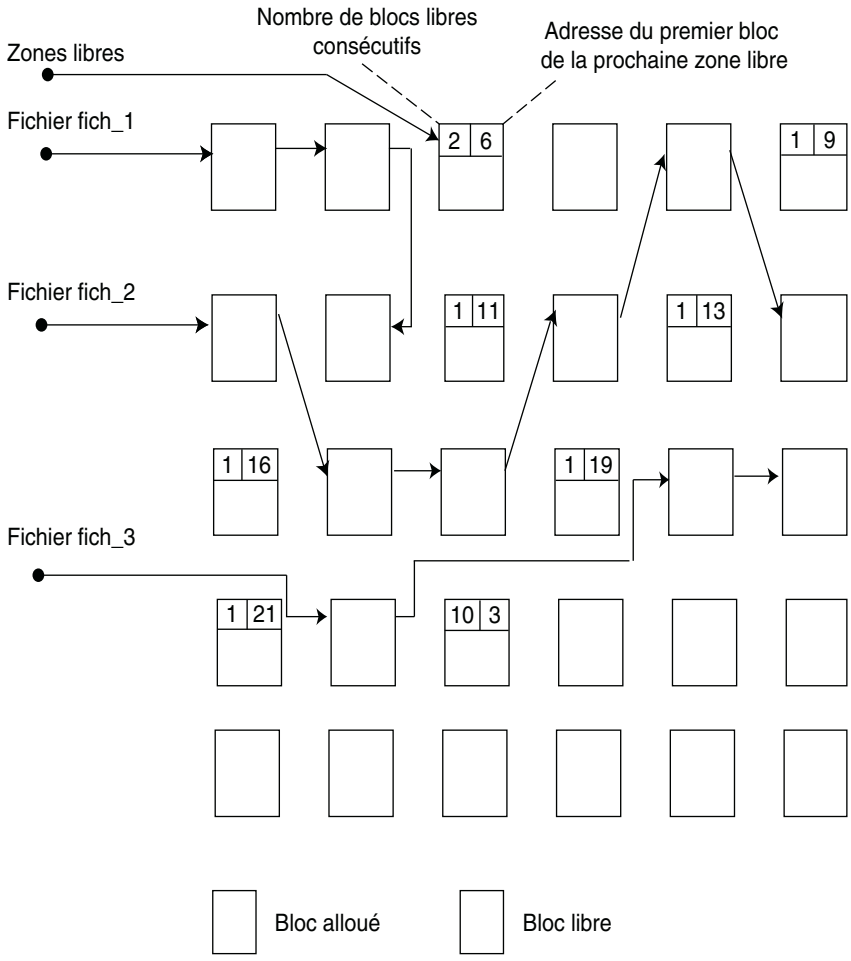


Figure 14.10 Gestion de l'espace libre par liste chaînée.

14.3 CORRESPONDANCE FICHIER LOGIQUE-FICHIER PHYSIQUE

14.3.1 Notion de répertoire

Définition

Le système de gestion de fichiers effectue la correspondance entre fichiers logiques et fichiers physiques par le biais d'une table appelée *répertoire* qui contient des informations de gestion des fichiers, dont notamment, pour chaque fichier existant sur le disque, le nom logique du fichier et son adresse physique sur le disque. Plus précisément, une entrée de répertoire concernant un fichier donné, contient généralement les informations suivantes :

– le nom logique du fichier;
– le type du fichier si les fichiers sont typés. Les principaux types de fichiers sont les fichiers texte, les fichiers objet, les fichiers exécutables, les fichiers de traitement par lots contenant des commandes pour l'interpréteur de commandes, les fichiers binaires correspondant par exemple à des images ou à des impressions. Le type d'un fichier est souvent codé dans son nom logique, à l'aide d'une extension séparée du nom proprement dit du fichier par un point. Ainsi, les extensions.exe pour les fichiers exécutables, .o pour les fichiers objets, .doc pour les fichiers texte issus de l'application Word, .gif ou .jpeg pour des fichiers binaires de type images;
– l'adresse physique du fichier, c'est-à-dire une information permettant d'accéder aux blocs physiques alloués au fichier. Cette information dépend de la méthode d'allocation mise en œuvre sur le disque. Ainsi, dans le cas d'une allocation contiguë ou dans le cas d'une allocation par blocs chaînés, l'information mémorise l'adresse du premier bloc alloué au fichier. Dans le cas d'une allocation par zones, cette information est une structure contenant l'adresse de la zone primaire et les adresses des zones secondaires allouées au fichier. Enfin, dans le cas d'une allocation indexée, cette information est constituée par l'adresse du bloc d'index;
– la taille en octets ou en blocs du fichier;
– la date de création du fichier;
– le nom du propriétaire du fichier;
– les protections appliquées au fichier, à savoir si le fichier est accessible en lecture, écriture ou en exécution et éventuellement par quels utilisateurs.

Le système de gestion de fichiers offre des primitives permettant de manipuler les répertoires; ce sont les opérations permettant de lister le contenu d'un répertoire, de changer de répertoire, de créer un répertoire ou de détruire un répertoire.

Ainsi sous le système Unix, la commande `mkdir nom_rep` crée un répertoire. La commande `rmdir nom_rep` détruit le répertoire. Enfin, la commande `ls – l nom_rep` permet d'afficher l'ensemble des fichiers connus du répertoire avec leurs caractéristiques.

Structure des répertoires

Les différentes structures de répertoires existantes se distinguent par le nombre de niveaux qu'elles présentent. Les répertoires à un niveau groupent tous les fichiers d'un support de masse dans une même table. S'ils sont simples, les répertoires à un niveau posent des difficultés quand le nombre de fichiers augmente et lorsque plusieurs utilisateurs différents stockent leurs fichiers sur un même support de masse car tous les noms de fichiers doivent être différents. On préfère alors les répertoires à deux niveaux où chaque utilisateur possède un répertoire propre, appelé *répertoire de travail*. La structure à deux niveaux se généralise facilement dans une structure à n niveaux pour laquelle chaque utilisateur hiérarchise son propre répertoire en autant de sous-répertoires qu'il le désire. Cette structure en arbre est composée d'un répertoire initial appelé la racine, souvent symbolisée par « / », d'un ensemble de nœuds

constitués par l'ensemble des sous-répertoires et d'un ensemble de feuilles qui sont les fichiers eux-mêmes.

Dans cette structure à n niveaux, le nom complet d'un fichier encore appelé *path-name* est constitué de son nom précédé du chemin dans la structure de répertoires depuis la racine. Ce chemin est composé des noms de chacun des répertoires à traverser depuis la racine, séparés par un caractère de séparation (par exemple / sous Unix et \ sous DOS) Ainsi, sur la figure 14.11, le nom complet du fichier F3 de l'utilisateur Arthur est /Arthur/pg/F3.

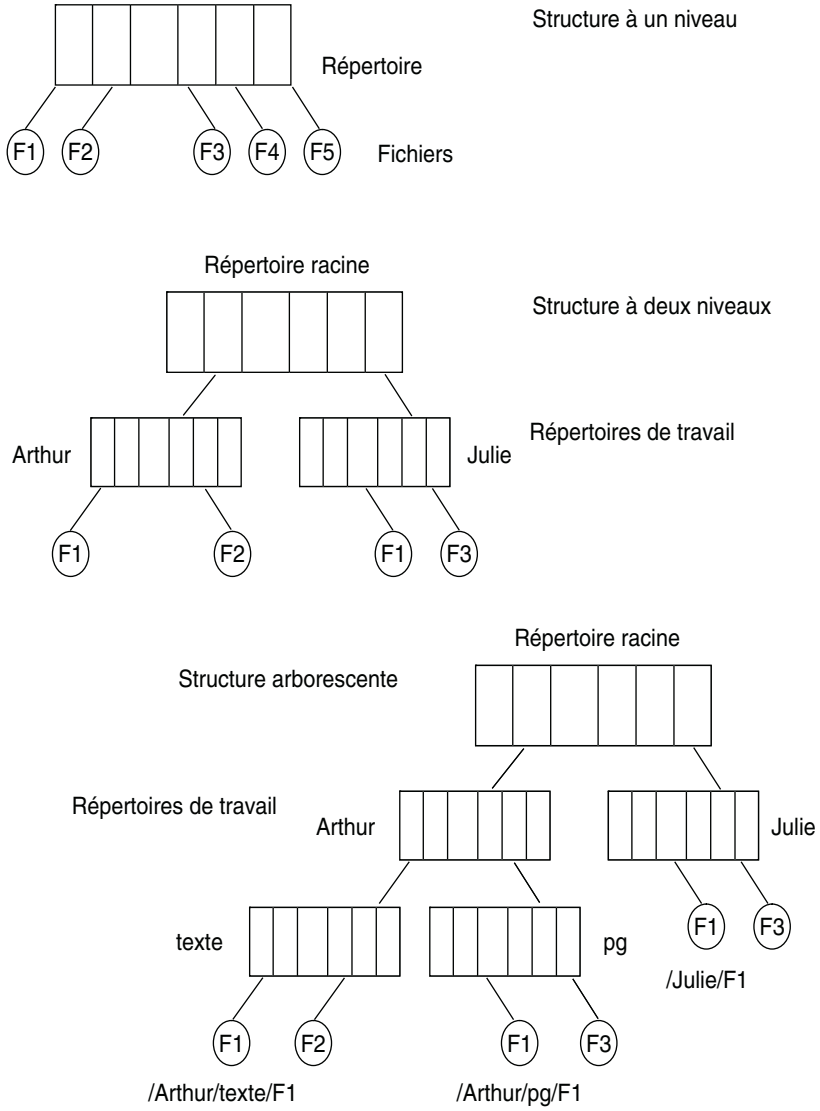


Figure 14.11 Structures de répertoire.

Notion de volumes ou partitions

Le système de gestion de fichiers d'un ordinateur peut comporter des milliers de fichiers répartis sur plusieurs giga-octets de disque. Gérer ces milliers de fichiers dans un seul ensemble peut se révéler difficile. Une solution couramment mise en œuvre est alors de diviser l'ensemble du système de gestion de fichiers en morceaux indépendants appelés *volumes* ou *partitions*. Chaque partition constitue alors un disque virtuel, auquel est associé un répertoire qui référence l'ensemble des fichiers présents sur la partition. Le disque virtuel ainsi créé peut physiquement correspondre à une seule partie de disque physique, ou bien regrouper plusieurs parties de disque, placées sur un même disque ou des disques différents (figure 14.12).

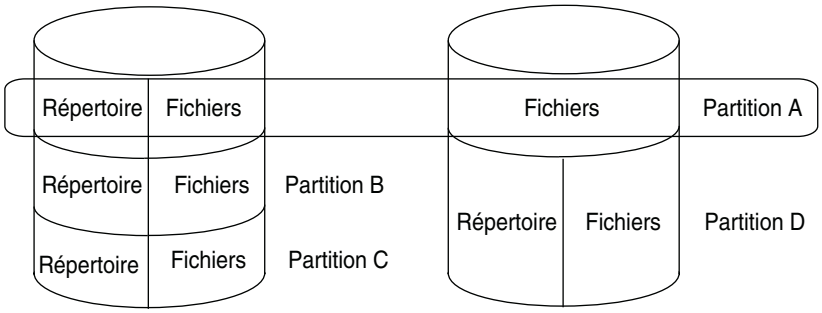


Figure 14.12 Partitions.

Chaque partition est repérée par un nom appelé *label*. Pour pouvoir être accessible, la partition doit être connectée à l'arborescence de fichiers de la machine, en un point d'ancrage qui correspond à un répertoire. Rendre accessible le contenu d'une partition à l'utilisateur de l'ordinateur en liant une partition à un répertoire constitue l'opération de *montage* de la partition. Cette opération est souvent accomplie à l'initialisation du système.

Ainsi, sous le système Linux, la disquette constitue une partition qui doit être liée au système de gestion de fichiers de la machine pour pouvoir être accessible. Ceci est réalisé par la commande `mount label_disquette/rep_disquette` qui lie ici la disquette à un répertoire placé directement sous la racine et appelé `rep_disquette`.

Exemples

➤ Partition FAT

Une partition principale de type FAT, sur les ordinateurs hébergeant les systèmes d'exploitation de la famille Windows, est divisée en six parties. On y trouve (figure 14.13) :

– un secteur d'amorçage placé dans le premier secteur du disque qui contient notamment la table des partitions, table dans laquelle sont stockées les informations relatives aux différentes partitions existantes sur le disque ;

– un secteur de boot qui contient des informations sur la structure du disque (taille d'un secteur, nombre de secteurs par piste, etc.) et un programme permettant le chargement du système d'exploitation en mémoire centrale ;
– un exemplaire de la FAT ;
– une copie optionnelle de la FAT, cette redondance permettant d'assurer une certaine tolérance aux pannes en cas d'accès inaccessible au premier exemplaire de la FAT ;
– le répertoire racine ou principal (*root directory*) ;
– la zone de données et de sous-répertoires.

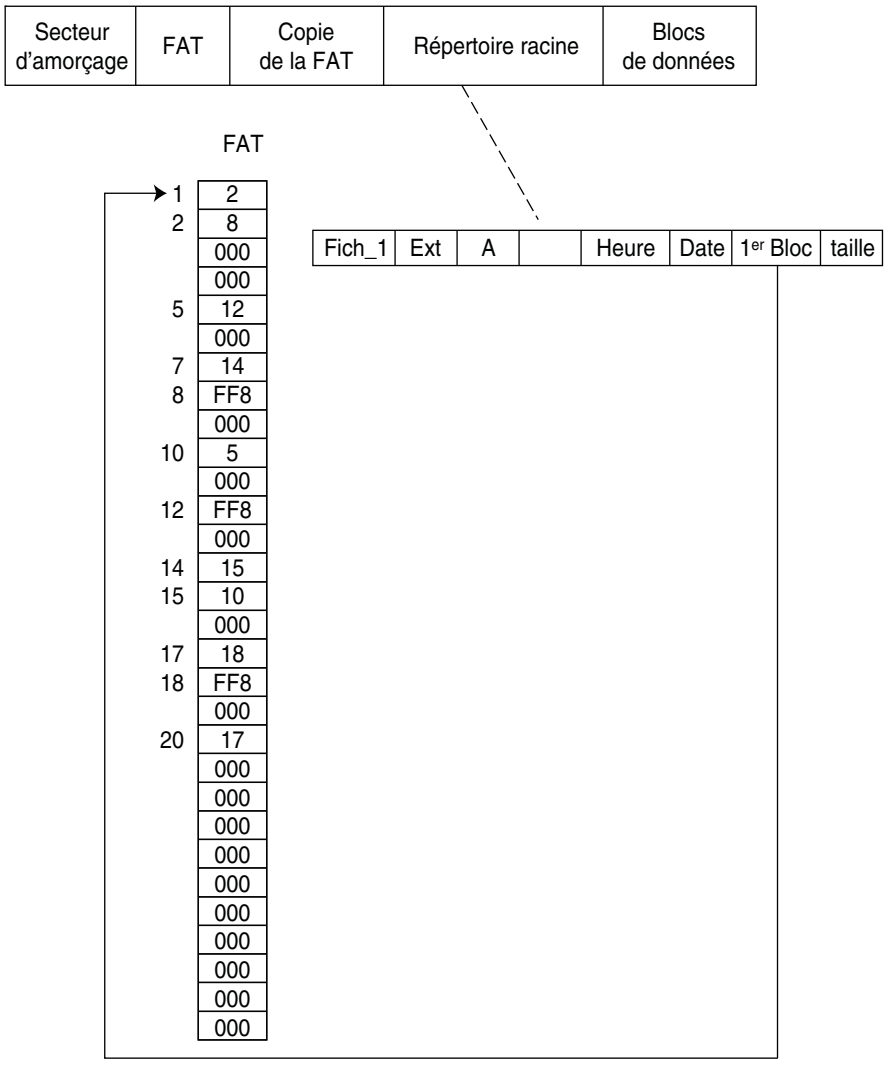


Figure 14.13 Partition FAT.

Le répertoire racine comprend 112 entrées de 32 octets chacune. Une entrée comporte les éléments suivants :

– le nom du fichier sur 8 octets ;
– l'extension du nom du fichier sur 3 octets ;
– les attributs du fichier sur un octet. Les attributs du fichier permettent de définir si le fichier est un fichier caché, un fichier système, un répertoire, une archive ou si le fichier est accessible en lecture seule ;
– un champ de 10 octets réservés à DOS ;
– l'heure de dernière écriture du fichier sur 2 octets ;
– la date de dernière écriture sur 2 octets ;
– le numéro du premier bloc constituant le fichier sur 2 octets, les autres blocs étant accessibles via la FAT ;
– la taille en octets du fichier sur 2 octets.

➤ Partition Unix

Sous le système Unix, un fichier est représenté physiquement par un descripteur stocké sur le disque, appelé *i-nœud* (*inode*). L'i-nœud d'un fichier contient toutes les informations concernant ce fichier c'est-à-dire notamment l'identificateur du propriétaire du fichier, le type du fichier, les permissions d'accès au fichier, les dates d'accès au fichier et enfin la table d'index de 13 entrées permettant de trouver les blocs de données alloués au fichier.

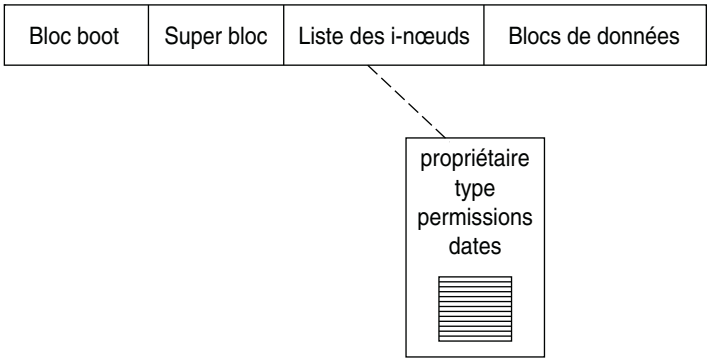


Figure 14.14 Partition Unix.

Dans ce contexte, un répertoire Unix est un fichier à part entière, décrit par un i-nœud, qui contient comme données, la liste des fichiers appartenant au répertoire, avec le numéro d'i-nœud qui leur est associé.

Une partition du système Unix a le format suivant. On y trouve (figure 14.14) :

– le bloc de boot occupe le premier secteur de la partition. Ce bloc contient le code permettant d'initialiser le système ;

– le super bloc contient des informations sur le système de gestion de fichiers associé à la partition, telles que la taille du système de gestion de fichiers, le nombre de blocs libres et l'adresse du premier bloc libre dans la liste des blocs libres, le nombre d'i-nœuds disponibles et l'adresse du premier i-nœud libre dans la liste des i-nœuds ;
– la liste des i-nœuds alloués à des fichiers. Parmi, ces i-nœuds se trouve l'i-nœud du répertoire racine de la partition qui est le répertoire à partir duquel la partition est accessible à l'utilisateur après l'opération de montage ;
– les blocs de données constituant les fichiers.

➤ Répertoire MVS

Le répertoire MVS encore appelé VTOC (*Volume Table Of Contents*) donne la liste des fichiers présents sur le disque, leurs caractéristiques et leur emplacement physique. La table est constituée d'entrées appelées DSCB (*DataSet Control Blocks*) qui décrivent soit des fichiers, soit des zones d'espaces libres. Il y a six types de DSCB :

– le type 1 concerne un fichier du disque et décrit la zone primaire allouée au fichier ainsi que des deux premières zones secondaires ;
– le type 2 indique l'emplacement de l'index du fichier si celui-ci est à accès séquentiel indexé ;
– le type 3 complète le DSCB de type 1 en décrivant les 13 zones secondaires supplémentaires ;
– le type 4 conserve des informations spécifiques sur la structure du disque ;
– le type 5 décrit les espaces libres disponibles sur le disque. Un DSCB de type 5 peut décrire au maximum 26 espaces libres ;
– le type 6 permet de décrire des zones partagées entre fichiers.

14.3.2 Réalisation des opérations

Fonctions du système de gestion de fichiers

Les fonctions systèmes composant le système de gestion de fichiers peuvent être groupées selon deux catégories : les fonctions liées aux opérations sur les fichiers et les fonctions liées à l'accès aux enregistrements.

➤ Fonctions liées aux opérations sur les fichiers

Ces fonctions permettent soit la création d'un fichier, soit l'ouverture d'un fichier, soit sa fermeture et sa destruction.

Création de fichier

Lors de la demande de création d'un fichier caractérisé par un nom logique et divers attributs tels que le type, la taille, le mode d'accès, le système d'exploitation vérifie qu'aucun autre fichier de même nom n'existe dans le répertoire de création du fichier. Si tel est le cas, il ajoute une entrée au répertoire pour le nouveau fichier, puis

alloue à partir de la liste des blocs libres de l'espace physique à ce nouveau fichier, en fonction de la méthode d'allocation en vigueur sur le disque. L'entrée du répertoire concernant ce nouveau fichier est mise à jour avec les attributs fournis par le biais de la fonction système et également avec les informations concernant l'allocation physique qui vient d'être réalisée. Dans le cas où un fichier de même nom existait déjà ou si l'allocation physique n'est pas possible, une erreur est générée.

Sous Unix, l'appel système `creat (const char * nom_fichier, int mode)` provoque la création d'un fichier dont le nom est `nom_fichier`, selon le mode `mode` qui peut être lecture seule, écriture seule ou lecture et écriture.

Ouverture de fichier

Lors de la demande d'ouverture d'un fichier caractérisé par un nom logique et effectuée selon un certain mode (lecture, écriture ou lecture/écriture), le système d'exploitation recherche dans le répertoire devant contenir le fichier, s'il existe une entrée correspondante. Si oui, il vérifie la compatibilité du mode d'ouverture demandé avec les droits d'accès associés au fichier, puis recherche l'adresse physique du début de fichier. Toutes les informations concernant le fichier, c'est-à-dire les informations contenues dans l'entrée de répertoire correspondant au fichier sont alors copiées dans une structure en mémoire centrale, appelée *descripteur de fichier* ou encore *bloc de contrôle de fichier (BCF)*. Ainsi, lors des accès ultérieurs à ce même fichier, le système d'exploitation n'aura pas besoin d'accéder de nouveau au disque pour les connaître. Le système effectue l'association entre le fichier logique et le fichier physique par le biais de ce bloc de contrôle. Dans le cas où aucun fichier du nom spécifié n'existe dans le répertoire ou si le mode d'ouverture ne correspond pas aux droits d'accès associés au fichier, une erreur est générée.

Sous Unix, l'appel système `int open (const char * nom_fichier, int mode)` provoque l'ouverture du fichier dont le nom est `nom_fichier`, selon le mode `mode` qui peut être lecture seule, écriture seule ou lecture et écriture. L'i-noeud disque décrivant ce fichier est copié en mémoire centrale. La fonction système renvoie à l'utilisateur un descripteur qui établit la liaison avec la copie de l'i-nœud.

Fermeture de fichier

L'opération de fermeture de fichier rompt la liaison entre le fichier logique et le fichier physique. Le BCF est copié sur le disque, dans l'entrée de répertoire correspondante, s'il a été modifié durant son séjour en mémoire centrale, puis il est détruit.

Sous Unix, l'appel système `int close (int desc)` provoque la fermeture du fichier accessible via le descripteur `desc`. L'i-nœud mémoire est recopié sur le disque si besoin est.

Destruction de fichier

Lors d'une demande de destruction de fichier, caractérisé par un nom logique, le système d'exploitation recherche dans le répertoire devant contenir le fichier, s'il existe une entrée correspondante. Si oui, il détruit cette entrée après avoir désalloué les blocs assignés au fichier. Ces blocs sont réintégrés à la liste des blocs libres du disque.

Sous Unix, la commande `rm nom_fichier` provoque la destruction du fichier `nom_fichier`. L'i-nœud disque alloué au fichier est réintégré à la liste des i-nœuds libres.

➤ Fonctions liées à l'accès aux enregistrements

Pour assurer les opérations de lecture/écriture, le système d'exploitation positionne un index logique de lecture/écriture dans le fichier, qui mémorise l'emplacement dans le fichier du prochain enregistrement délivré. À l'ouverture du fichier, cet index est placé au début du fichier et pointe sur le premier enregistrement de celui-ci. Il est mémorisé dans le BCF du fichier.

Pour réaliser la lecture d'un enregistrement du fichier, le système d'exploitation utilise le BCF du fichier pour récupérer l'index de position courante dans le fichier et calcule la position permettant de lire l'enregistrement demandé. Puis, il effectue la lecture du bloc contenant l'enregistrement.

Afin d'améliorer les performances du système de gestion de fichiers, les opérations de lecture et écriture d'enregistrements ne se traduisent pas systématiquement par des opérations d'entrées-sorties disque. En effet, le système d'exploitation utilise un mécanisme de cache composé d'un nombre fixe de tampons qui permettent de mémoriser en mémoire centrale les blocs les plus récemment lus. Ainsi, un bloc à lire est tout d'abord recherché dans le cache et l'opération d'entrées-sorties n'est réalisée que si le bloc n'y est pas trouvé. D'une manière similaire, les opérations d'écriture sont réalisées dans le cache et le bloc n'est réellement écrit sur le disque que lorsque le tampon le contenant doit être libéré pour y mémoriser un nouveau bloc.

Déroulement d'une opération d'entrées-sorties

En prenant exemple sur le fonctionnement du système MVS 370, nous décrivons à présent la réalisation d'une opération d'entrées-sorties sur un disque, telle que la lecture d'un enregistrement.

➤ Interface des dispositifs d'entrées-sorties

Rappels matériels

Sur les gros systèmes de type MVS, la réalisation des opérations d'entrées-sorties est réalisée au niveau matériel par un processeur d'entrées-sorties appelé *sous-système canal*. Ce sous-système canal exécute un programme placé en mémoire centrale, appelé *programme canal* et composé de *mots de commande canal* (CCW), qui décrit l'opération d'entrées-sorties à réaliser. Le programme canal est construit par le processeur principal, qui ensuite laisse la réalisation de l'opération d'entrées-sorties à la charge entière du sous-système canal. Ce dernier signale par interruption la fin de l'opération.

Le sous-système canal dialogue avec le contrôleur du disque. Le contrôleur du disque traduit les CCW du programme canal à l'unité disque qui lui est connectée. La figure 14.15 illustre cette architecture matérielle.

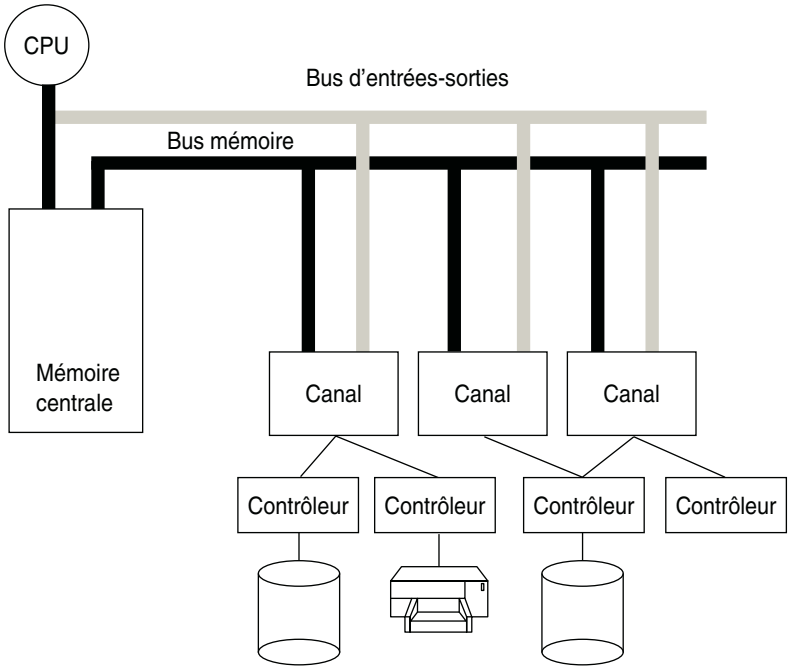


Figure 14.15 Interface matérielle du disque.

Interface système et étapes de l'opération d'entrées-sorties (lecture d'un enregistrement)

L'interface du système d'exploitation avec le dispositif d'entrées-sorties matériel est composée de trois parties, la dernière citée étant la plus proche du matériel et du sous-système canal : la méthode d'accès, le pilote du superviseur d'entrées-sorties (*IOCS driver*) et le superviseur d'entrées-sorties (*IOCS*).

La *méthode d'accès* est appelée par l'opération de lecture d'un enregistrement par exemple. Cette méthode d'accès construit le programme canal. Puis, elle décrit dans un bloc de contrôle d'entrées-sorties (*IOB input/output block*), les paramètres de l'opération d'entrées-sorties c'est-à-dire notamment l'identification du fichier concerné, l'identification de l'unité disque et du contrôleur concernés, l'adresse du programme canal en mémoire centrale. Enfin, elle passe la main au pilote du superviseur d'entrées-sorties.

Le *pilote du superviseur* d'entrées-sorties prépare l'environnement nécessaire à l'opération d'entrées-sorties. Notamment, il contrôle les données fournies par la méthode d'accès, puis il traduit les adresses virtuelles du programme canal en adresses réelles et fixe les pages du programme canal en mémoire centrale car le sous-système canal ne connaît pas la mémoire virtuelle. Enfin, le pilote lance le superviseur d'entrées-sorties.

Le *superviseur d'entrées-sorties* réalise l'interface avec le sous-système canal. Il met la demande d'entrées-sorties transmise par le pilote dans la file d'attente corres-

pondant au contrôleur de l'unité disque concernée, puis lance l'opération d'entrées-sorties au niveau du sous-système canal, si celui-ci est libre. Par ailleurs, le processus initiateur de l'opération d'entrées-sorties est mis dans l'état bloqué et l'ordonnanceur est invoqué pour réallouer le processeur principal.

Lorsque l'opération d'entrées-sorties est terminée, le matériel émet une interruption qui est captée par le superviseur d'entrées-sorties. La routine d'interruption analyse les résultats de l'opération d'entrées-sorties, puis la fin l'opération est signalée au pilote du superviseur. Le superviseur d'entrées-sorties lance l'opération d'entrées-sorties suivante dans la file d'attente du contrôleur. Le processus initiateur de l'opération d'entrées-sorties est débloqué et placé dans l'état prêt et l'ordonnanceur est invoqué pour réallouer le processeur principal. Lors de l'élection du processus, la méthode d'accès achève son exécution et renvoie l'enregistrement souhaité. La figure 14.16 illustre cet ensemble d'événements.

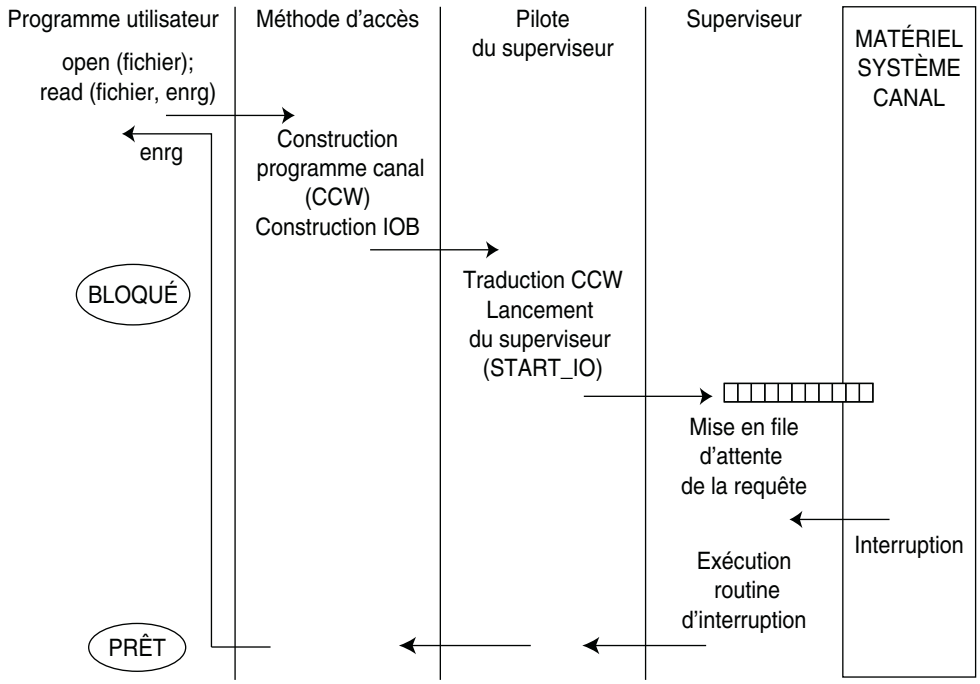


Figure 14.16 Interface système du disque.

➤ Ordonnancement des requêtes disque

Le superviseur d'entrées-sorties place chaque nouvelle requête d'entrées-sorties dans la file d'attente du contrôleur du disque concerné.

L'accès à un secteur du disque se décompose en deux parties :

– le *temps de positionnement* du bras correspond au temps nécessaire pour que le bras portant la tête de lecture vienne se positionner sur la piste contenant le secteur ;

– le *temps de latence* correspond au temps nécessaire pour qu'une fois la tête placée sur la bonne piste, le secteur passe sous la tête de lecture. Ce temps de latence dépend de la vitesse de rotation du disque.

Le temps de positionnement est la partie la plus pénalisante pour la réalisation d'une opération d'entrées-sorties sur le disque. Afin de réduire cette pénalité induite par les mouvements du bras, le système ordonnance les requêtes de la file d'attente du contrôleur de manière à réduire ces mouvements. Les principaux algorithmes d'ordonnancement sont : FCFS (*First Come, First Served*), SSTF (*Shortest Seek Time First*) et SCAN (ascenseur ou balayage).

Nous les présentons chacune avec un exemple appliqué à la file de requêtes disque suivante, où chaque nombre représente un numéro de piste à atteindre : 50, 110, 25, 105, 12, 100, 40, 45, 10, 80, 88. Le bras est initialement sur la piste 90. Le disque comporte 150 pistes par plateau.

Algorithme FCFS

Avec cet algorithme, les requêtes disque sont servies selon leur ordre de soumission. Cette politique, si elle est simple à mettre en œuvre, ne minimise pas les mouvements du bras. Sur la file de requêtes définie dans l'exemple, le déplacement total du bras est égal à 624 pistes. La figure 14.17 illustre l'ordre de service des requêtes et le déplacement du bras qui s'ensuit.

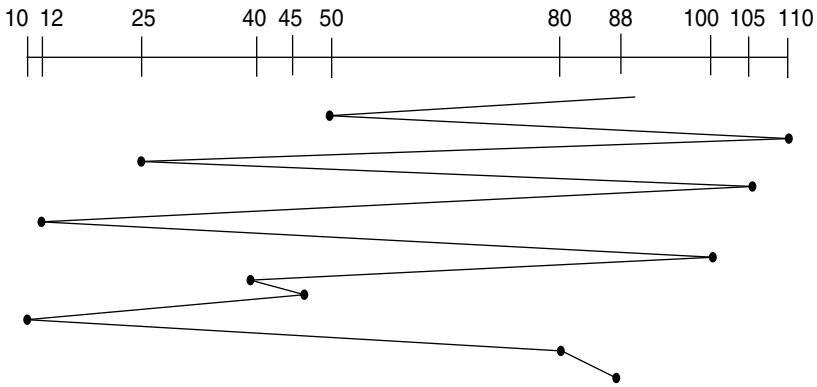


Figure 14.17 Algorithme FCFS.

Algorithme SSTF

Avec cet algorithme, la prochaine requête disque servie est celle dont la position est la plus proche de la position courante, donc celle induisant le moins de déplacement du bras. Le problème de cette politique est le risque de famine pour des requêtes excentrées par rapport au point de service courant. Sur la file de requêtes définie dans l'exemple, l'ordre de service est 88, 80, 100, 105, 110, 50, 45, 40, 25, 12, et 10 et le déplacement total du bras est égal à 140 pistes. La figure 14.18 illustre l'ordre de service des requêtes et le déplacement du bras qui s'ensuit.

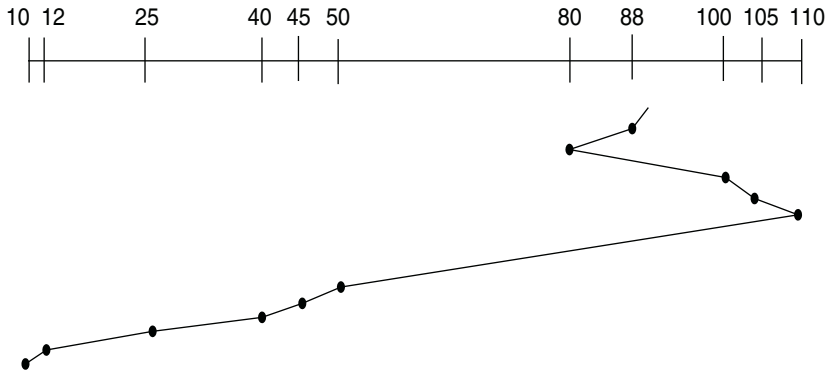


Figure 14.18 Algorithme SSTF.

Algorithme SCAN

C'est un algorithme de type balayage. La tête de lecture/écriture démarre à une extrémité du disque et parcourt celui-ci jusqu'à l'autre extrémité, en servant les requêtes au fur et à mesure de chaque piste rencontrée. Quand la tête de lecture/écriture est parvenue à l'autre extrémité du disque, son mouvement s'inverse et le service continue en sens opposé. Sur la file de requêtes définie dans l'exemple, l'ordre de service est 100, 105, 110, 88, 80, 50, 45, 40, 25, 12, et 10 si l'on suppose un parcours initialement ascendant des pistes et le déplacement total du bras est égal à 195 pistes. La figure 14.19 illustre l'ordre de service des requêtes et le déplacement du bras qui s'ensuit.

Une variante de cette politique est la politique C-SCAN. Le principe de cet algorithme est le même que le précédent. La seule différence consiste dans le mouvement

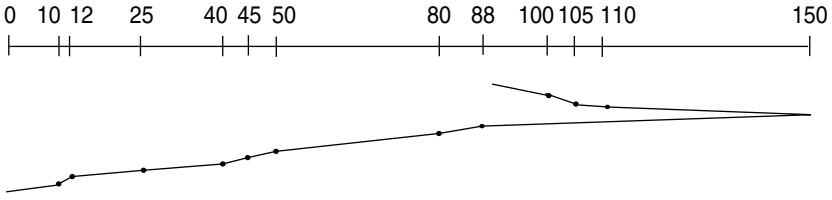


Figure 14.19 Algorithme SCAN.

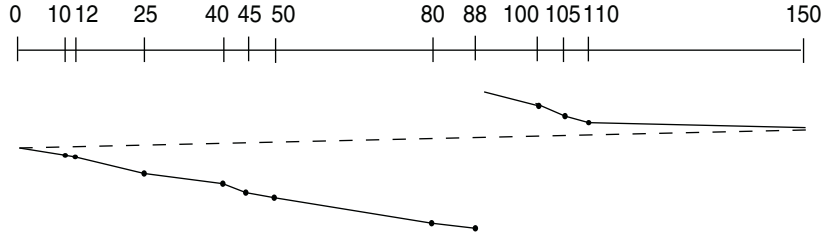


Figure 14.20 Algorithme C-SCAN.

du bras lorsqu'il arrive à l'autre extrémité du disque. Au lieu de repartir en sens inverse, le bras va ici se repositionner sur l'extrémité initiale de son parcours. Ainsi, le parcours des pistes se fait toujours dans le même sens. Cette politique offre pour chaque requête disque un temps de traitement plus uniforme que la politique précédente. Sur la file de requêtes définie dans l'exemple, l'ordre de service est 100, 105, 110, 10, 12, 25, 40, 45, 50, 80 et 88 et le déplacement total du bras est égal à 298 pistes.

Une dernière variante de ces politiques est la politique LOOK. Le bras parcourt de la même manière l'ensemble du disque en sens montant ou descendant, mais cette fois il s'arrête dans son balayage dès qu'il a rencontré la requête la plus extrême dans le sens montant ou descendant.

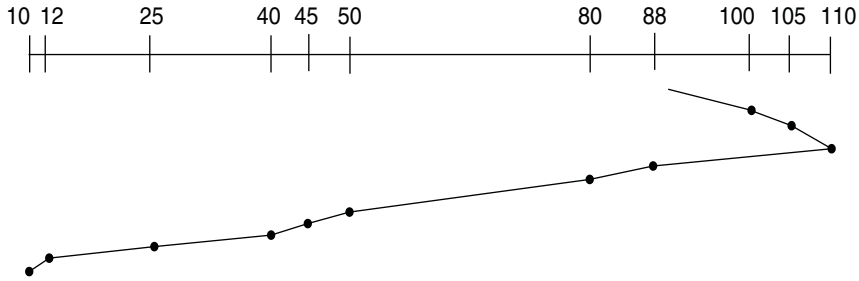


Figure 14.21 Algorithme LOOK.

14.4 PROTECTION

La protection du système de gestion de fichiers recouvre deux aspects d'une part, la protection contre les accès inappropriés et, d'autre part, la protection contre les dégâts physiques.

14.4.1 Protection contre les accès inappropriés

La protection du système de gestion de fichiers contre les accès inappropriés peut être réalisée de plusieurs façons. Une première solution est d'associer un mot de passe à chaque fichier que l'utilisateur souhaite protéger. C'est une solution souvent retenue sur les systèmes mono-utilisateur mais elle est plus difficilement mise en œuvre sur les systèmes multi-utilisateurs. Ainsi, la protection des fichiers par mot de passe est une méthode usuellement mise en œuvre dans les systèmes IBM. Les fichiers peuvent être protégés en écriture par un mot de passe ou en lecture/écriture. Ce mot de passe est stocké dans l'entrée de la VTOC correspondant au fichier.

Une autre solution consiste à définir des *droits d'accès* associés aux fichiers, tels que le droit le lire le fichier, le droit d'écrire le fichier, le droit d'exécuter le fichier ou encore le droit de détruire le fichier. Une liste, appelée *liste d'accès,* est ensuite construite. Elle regroupe pour chaque fichier, les identifiants des utilisateurs avec les

droits qui leur sont associés. Lorsqu'un utilisateur demande l'accès à un fichier, le système parcourt la liste d'accès associé au fichier à la recherche de l'identifiant de l'utilisateur et vérifie que le type de l'opération que souhaite réaliser l'utilisateur est conforme avec les droits qu'il possède.

Évidemment, la taille d'une liste d'accès peut se révéler très importante si le fichier est utilisé par de nombreux utilisateurs différents. Pour remédier à ce problème, des groupes d'utilisateurs auxquels sont associés des droits peuvent être construits. Un utilisateur hérite des droits du groupe auquel il appartient.

Ainsi, le système de gestion de fichiers du système d'exploitation Unix définit trois groupes : le « propriétaire » désigne l'utilisateur qui a créé le fichier, le « groupe » regroupe des utilisateurs appartenant à un même groupe de travail (par exemple, les enseignants, les étudiants…), enfin les « autres » assemblent tous ceux qui ne sont ni le propriétaire, ni le groupe. La définition de ces trois groupes et la gestion des appartenances des utilisateurs aux groupes sont réalisées par l'administrateur de la machine. Pour chaque groupe, trois droits d'accès sont définis : le droit d'accès en lecture (r), le droit d'accès en écriture (w) et les droits d'accès en exécution (x). Voici la commande `ls – l` permet de lister les fichiers du répertoire courant en affichant les droits qui leur sont associés. Ainsi le fichier `essai.c` est accessible en lecture et écriture par le propriétaire, et en lecture seule pour le groupe et les autres. L'exécution de la commande `chmod a + w essai.c` demande l'ajout du droit d'accès en écriture (+ w) pour tous (a).

```
> ls – l
-rw-r--r--    1 delacroi 6401 Jan 8 1997 eleve.c
-rwxr-xr-x    1 delacroi 24576 Dec 15 1998 essai
-rw-r--r--    1 delacroi 67 Dec 15 1998 essai.c
> chmod a + w essai.c
> ls – l essai.c
-rw-rw-rw-    1 delacroi 67 Dec 15 1998 essai.c
```

14.4.2 Protection contre les dégâts physiques

Le système de gestion de fichiers peut être endommagé ou détruit de multiples manières : les coupures d'électricité, l'écrasement des têtes de lecture/écriture sur le disque, les poussières ou encore des températures extrêmes peuvent altérer le support physique de stockage.

La protection contre ces dégâts physiques est assurée par le recours *aux techniques de redondance*, qui consiste à dupliquer et généralement à stocker dans des endroits différents, les données maintenues sur le support physique. Cette redondance peut être de deux types, la *redondance interne* et la *redondance par sauvegarde périodique*.

Redondance interne

La redondance interne consiste à maintenir l'information en double exemplaires sur le support physique. La première version appelée *version primaire* est la version

couramment utilisée. La seconde version appelée *version secondaire* est une copie de la version primaire et est utilisée lorsque la version primaire devient inaccessible. Le système assure la cohérence de contenu entre la version primaire et la version secondaire. Ainsi, les systèmes DOS/Windows maintiennent au niveau de chaque partition deux exemplaires de la FAT. Le premier exemplaire de la FAT est celui qui est couramment recopié en mémoire centrale pour permettre l'accès aux fichiers. Toute modification de cette copie de la FAT est recopiée sur le disque dans la version primaire et dans la version secondaire. Si jamais la version primaire se révèle inaccessible, la version secondaire est lue et copiée dans la version primaire.

Redondance par sauvegardes périodiques

L'acquisition de la redondance par sauvegarde périodique consiste à copier l'ensemble du système de gestion de fichiers sur un second support. Ainsi, la sauvegarde des disques durs est-elle souvent réalisée sur bandes magnétiques. Cette opération est la plupart du temps lancée automatiquement par le système d'exploitation, une fois que l'administrateur de la machine a configuré les paramètres du programme responsable de celle-ci.

La *sauvegarde complète* consiste à copier régulièrement l'intégralité du système de gestion de fichiers. Le problème d'une telle opération est qu'elle peut être très longue à réaliser et prendre ainsi plusieurs heures durant lesquelles le système est inactif. La *sauvegarde incrémentale* consiste à ne copier que les fichiers qui ont été modifiés depuis la dernière sauvegarde réalisée. Ceci permet évidemment de réduire le temps nécessaire à la réalisation de cette opération.

Les deux types de sauvegarde sont très souvent utilisés simultanément mais à rythmes différents. Par exemple, une sauvegarde incrémentale est programmée tous les soirs de la semaine hormis le dimanche, jour où a lieu une sauvegarde complète. Une restauration, suite à une perte de données, s'effectue alors en utilisant la sauvegarde complète du dimanche, actualisée ensuite avec les sauvegardes incrémentales réalisées jusqu'à la veille de la restauration.

14.5 CONCLUSION

Le système de gestion de fichiers assure la conservation des données sur un support de masse non volatile avec comme unité de stockage, le fichier.

Le concept de fichier recouvre deux niveaux. D'une part, le fichier logique représente l'ensemble des données incluses dans le fichier telles qu'elles sont vues par l'utilisateur. D'autre part, le fichier physique représente le fichier tel qu'il est alloué physiquement sur le support de masse. Différentes méthodes d'allocation permettent d'attribuer des blocs physiques au fichier.

Le système d'exploitation gère ces deux niveaux de fichiers et assure notamment la correspondance entre eux, en utilisant une structure de répertoire. Cette structure

de répertoire est généralement arborescente et attribue à chaque utilisateur un répertoire de travail dans lequel il dépose ses fichiers.

La protection du système de gestion de fichiers recouvre deux aspects. D'une part la protection contre les accès inappropriés est assurée par le biais de la définition de droits d'accès associés au fichier. D'autre part, la protection contre les dégâts physiques est assurée par les techniques de redondance et de sauvegardes.

Chapitre **15**

Introduction aux réseaux

Ce chapitre constitue une introduction des notions relatives aux réseaux. Après avoir exposé le rôle grandissant du réseau dans la société actuelle et présenté une classification courante des différents types de réseaux, nous consacrons une première partie aux principes fondamentaux des réseaux filaires dans laquelle nous décrivons l'architecture de ces réseaux ainsi que les modes de transmissions des données mises en œuvre. Une seconde partie traite des notions liées aux réseaux sans fil, en exposant leurs particularités vis-à-vis des réseaux filaires. Dans chaque cas, des réseaux concrets tels que Ethernet, Bluetooth ou Wi-Fi sont pris à titre d'exemples. Ce chapitre s'achève en abordant la problématique de l'interconnexion de réseaux et à travers elle l'Internet.

15.1 DÉFINITION

Un réseau est un ensemble de matériels et de logiciels distribués permettant d'offrir des services (par exemple le téléphone ou la diffusion de musiques ou de films) ou de permettre le transport de données.

Il n'existe pas une manière unique de classer les réseaux, mais généralement on retient deux critères pour les caractériser : la taille et le mode de transmission de l'information.

La taille

Selon ce critère, on classe les réseaux en :

– *Réseaux personnels* ou PAN (*Personal Area Network*) : c'est typiquement le réseau permettant de connecter l'ordinateur à ses périphériques (souris, clavier, imprimante…).

– *Réseaux locaux* ou LAN (*Local Area Network*) : ce sont des réseaux qui recouvrent une surface géographique réduite (par exemple un bâtiment). Ils permettent un partage des ressources informatiques avec des débits importants (entre 10 et 100 Mbits/s).
– *Réseaux métropolitains* ou MAN (*Metropolitan Area Network*) : ils couvrent une surface géographique de l'ordre de la ville. Ils sont généralement utilisés pour fédérer des réseaux locaux, assurer la diffusion d'informations pour des circonscriptions géographiques importantes telles des villes ou des campus. Un exemple important est le réseau de télévision par câble dans certaines grandes villes.
– *Réseaux longues distances* ou WAN (*Wide Area Network*) : ce sont des réseaux qui couvrent des surfaces correspondant à un pays. Les débits sont variables (de quelques Kbits/s à quelques Mbits/s).
– *Réseaux domestiques :* ces réseaux commencent à apparaître et permettront à des appareils domestiques, dans un appartement par exemple, de communiquer entre eux.
– *Inter-réseaux :* il existe un très grand nombre de réseaux dans le monde, chacun ayant des architectures matérielles et logicielles différentes. Le besoin de communiquer au travers de réseaux hétérogènes a provoqué le développement de l'interconnexion de réseaux au moyen de machines spécialisées (les passerelles ou *gateway*). L'objet des passerelles est d'assurer les traductions, tant au plan matériel que logiciel, permettant à l'information de circuler d'un réseau à un autre. Ainsi deux machines raccordées à deux réseaux hétérogènes pourront communiquer, c'est-à-dire échanger de l'information. Un problème important est la traduction des adresses des machines (l'adresse d'une machine étant déterminée par le type du réseau auquel la machine est connectée). L'Internet est un exemple très important d'inter-réseau.

Mode de transmission

On distingue deux grands modes de transmission : la *diffusion* et le *point à point*.

– Un réseau à diffusion dispose d'un seul canal de transmission partagé par tous les utilisateurs du réseau. L'information est généralement transmise sous formes de *paquets* reçus par l'ensemble des machines connectées au réseau. Dans le paquet, on trouve d'une part, l'information à transmettre, d'autre part, une information (dite d'*adressage*) permettant au récepteur de reconnaître que cette information est pour lui. L'adressage permet, soit d'atteindre toutes les machines du réseau (diffusion *broadcast*) soit d'atteindre un sous-ensemble de machines (diffusion *multicast*).
– À l'inverse, le réseau point à point, permet la connexion d'un très grand nombre de machines. Les machines sont reliées une à une. Dans ce type de réseau, l'information, pour aller d'une machine à une autre, passe par plusieurs machines intermédiaires. Ainsi il peut y avoir plusieurs *routes* pour aller d'une machine à une autre. Dans ce type de réseau, le calcul de la meilleure route est donc un facteur important d'amélioration de l'efficacité de la transmission. La transmission point à

point s'appelle aussi transmission *unicast.* Ces réseaux sont aussi appelés *réseaux de commutation* indiquant par là une fonction fondamentale (la commutation) utilisée pour la transmission des informations dans de tels réseaux.

Enfin on peut noter l'utilisation de plus en plus importante des réseaux sans fil. Ce mode de transmission n'est pas une nouveauté, mais l'amélioration de la qualité des transmissions permet le développement important de ce type de réseaux.

On peut raisonnablement classer les réseaux sans fil en trois catégories : les réseaux personnels, les LAN et les WAN.

15.2 LES RÉSEAUX FILAIRES

Les réseaux filaires ou à transmission des données par des supports guidés occupent une place très importante. Nous présentons ces réseaux en précisant les aspects concernant l'architecture des réseaux filaires et les modalités de circulation de l'information dans les réseaux.

15.2.1 Architecture des réseaux filaires

Les supports physiques

Le rôle des supports de transmission est de transporter un flux de données binaires brutes d'une machine à une autre. Il existe plusieurs types de supports qui se différencient par leurs caractéristiques comme la bande passante, le délai de transmission, le coût mais aussi la simplicité d'installation. Les supports de transmission guidée sont construits à partir des paires torsadées, du câble coaxial et de la fibre optique.

La paire torsadée est composée de deux fils de cuivre isolés. Ces fils sont enroulés de manière hélicoïdale. Ce type de support est très largement utilisé dans les systèmes téléphoniques. La bande passante dépend du diamètre des fils de cuivre et de la distance à parcourir. Il est possible de transporter des informations à plusieurs Mbits/s sur de courtes distances (quelques kilomètres). Elle permet le transport des signaux analogiques et numériques. Ainsi par ses performances, son coût et la simplicité d'utilisation, elle est très largement utilisée.

Le câble coaxial est constitué de deux conducteurs concentriques séparés par un isolant. Le conducteur extérieur est une tresse de cuivre appelé *blindage* qui est relié à la terre. Le tout est protégé par une gaine isolante. Le câble coaxial permet des débits plus élevés que la paire torsadée, il est peu sensible aux perturbations électromagnétiques extérieures et le taux d'erreurs sur un tel câble est très faible. Il est très utilisé en transmission numérique dans les réseaux locaux où l'on peut atteindre des débits de 10 Mbits sur des distances de l'ordre du kilomètre. En liaison analogique, le câble coaxial est utilisé pour réaliser des liaisons longues distances.

La transmission des données par *fibre optique* permet d'envisager des bandes passantes de l'ordre de 50 000 Gbits/s (50 Tbits/s). Actuellement on est limité à une bande passante de l'ordre de 10 Gbits/s. Cette limitation est essentiellement due à la difficulté d'effectuer les transformations des signaux numériques en signaux optiques.

Il est intéressant de comparer la fibre optique au câble de cuivre. La fibre présente de nombreux avantages :

– la bande passante importante (bien plus que celle du câble de cuivre) la rend bien adaptée aux réseaux à hautes performances ;
– la faible atténuation du signal permet un nombre limité de répéteurs (un répéteur tous les 50 km environ) pour les liaisons longues distances. Le coût d'installation est donc beaucoup plus faible qu'avec un support à base de fil de cuivre où il faut un répéteur tous les 5 km environ ;
– elle est fine et légère, insensible à la corrosion et aux surtensions électriques.

Par contre il s'agit d'une technologie récente qui nécessite des compétences que tout le monde n'a pas. Par ailleurs elle ne supporte pas facilement d'être pliée, et s'appuyant sur de la transmission optique, elle ne transmet que des flux unidirectionnels. Il est clair que dans un proche avenir toutes les lignes de communications fixes dépassant quelques mètres de longueurs seront réalisées à partir de fibres optiques.

Topologies physiques des réseaux

La topologie physique d'un réseau spécifie la manière dont les nœuds composant le réseau sont connectés. Les topologies de base sont toutes des variantes d'une liaison point à point (figure 15.1).

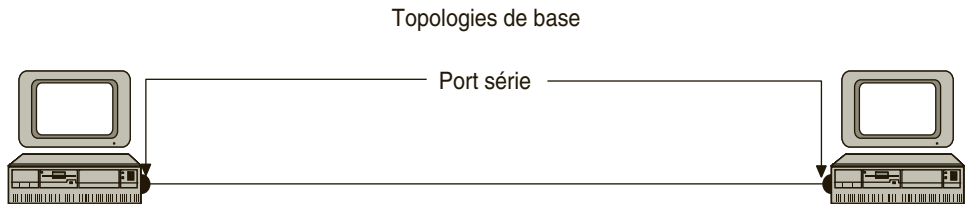


Figure 15.1 Liaison point à point.

La topologie en bus est la plus simple des topologies de base (figure 15.2). Dans une telle topologie, l'information est diffusée sur tout le réseau (réseau à diffusion) et chaque station accède directement au réseau (canal de diffusion unique). Cette topologie induit des conflits d'accès à la ressource unique (canal de transmission) qui se traduisent par des *collisions*. Une telle topologie nécessite de définir une politique d'accès au réseau pour éviter les conflits.

Dans *la topologie en anneau* (figure 15.2) chaque station est connectée à la suivante par une liaison point à point. Les messages circulent, dans un seul sens, sur l'anneau. Chaque station reçoit le message. Si le message lui est destiné alors la station recopie celui-ci et le régénère, sinon la station se contente de le régénérer. Ce type de topologie permet des débits élevés sur de grandes distances (régénération du signal par les stations) mais est sensible à la rupture de l'anneau.

La topologie en étoile est construite à partir d'un nœud qui émule plusieurs liaisons point à point (figure 15.3).

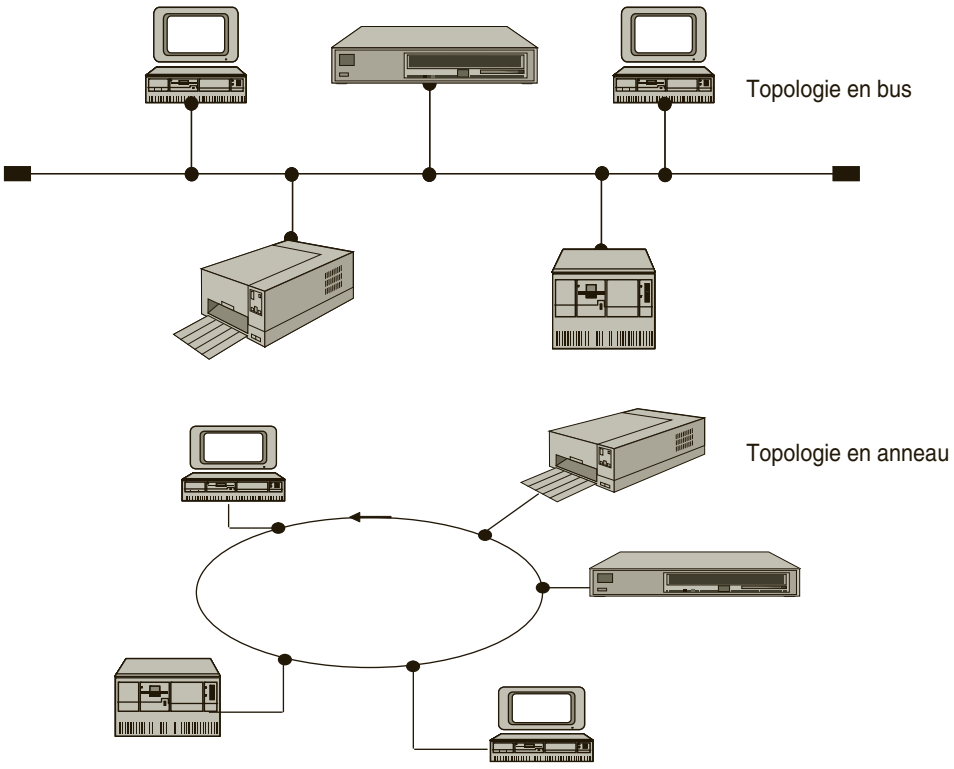


Figure 15.2 Topologie en bus et en anneau.

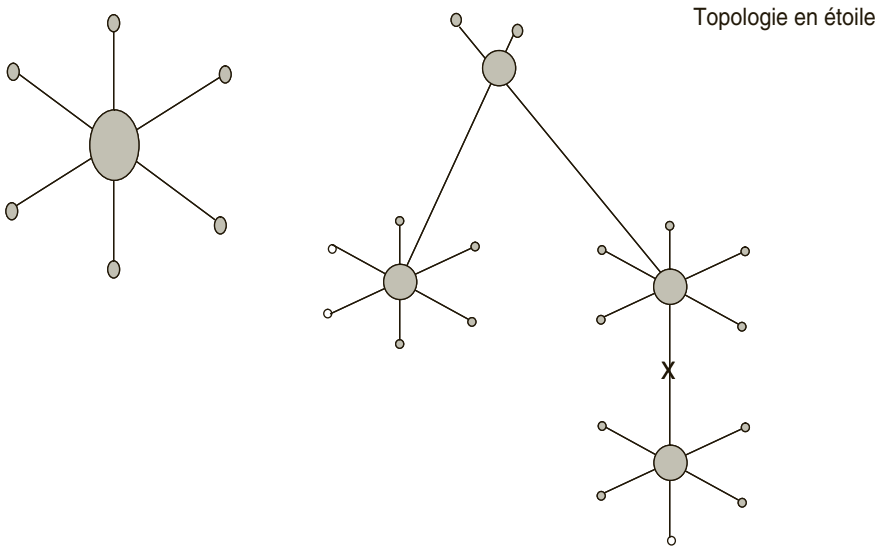


Figure 15.3 Topologie en étoile.

Dans cette topologie, tous les messages transitent par le point central de l'étoile qui joue le rôle de *concentrateur*. Le concentrateur est un composant actif qui examine tous les messages et les retransmet au bon destinataire. Cette topologie est bien adaptée à la distribution de postes téléphoniques dans une entreprise. Elle est performante, insensible à la défaillance d'un nœud mais sensible à la défaillance du concentrateur.

Pour permettre l'extension des réseaux on développe des topologies arborescentes (figure 15.3). Ces topologies sont construites à partir de plusieurs réseaux en étoiles reliés par des concentrateurs jusqu'au nœud de base (la racine du réseau). Cette topologie, arborescente, est très utilisée pour la construction des réseaux locaux. Ces réseaux présentent des faiblesses à cause de l'existence des concentrateurs : la défaillance d'un nœud l'isole du reste du réseau. Une solution consiste à mettre en place des chemins de secours. On évolue alors vers les réseaux *maillés* (figure 15.4).

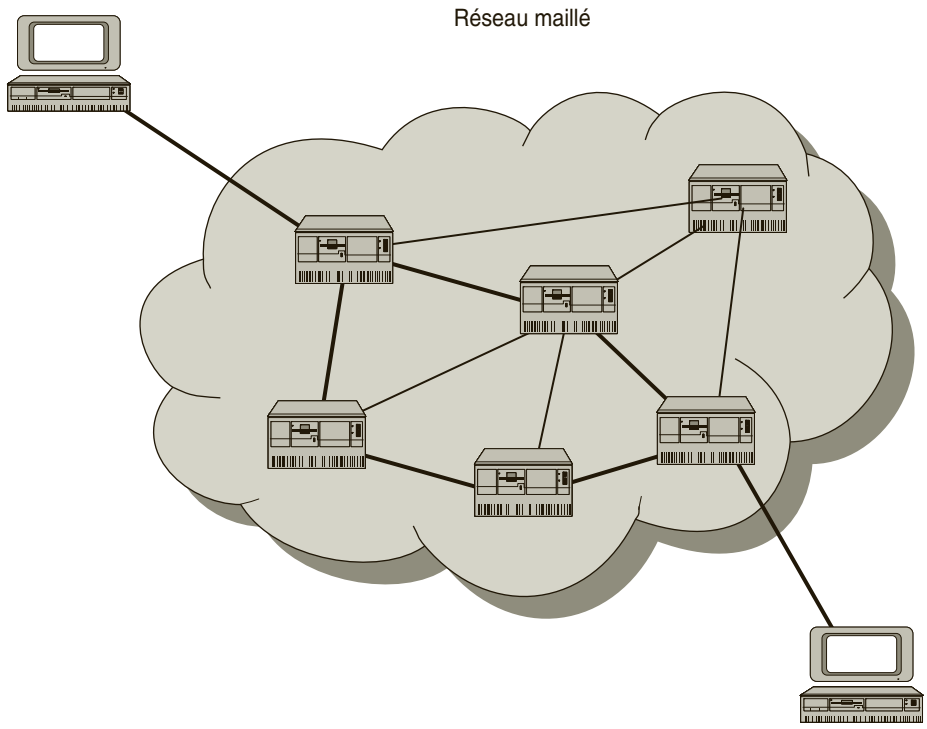


Figure 15.4 Réseau maillé.

Dans ce type de topologie, il existe plusieurs chemins pour relier une station à une autre station. Une telle topologie est très résistante à la défaillance d'un nœud et permet une répartition des charges entre les nœuds. Les nœuds sont reliés par des liaisons point à point. Le nombre de liaisons qui lient un nœud aux autres définit la *connectivité* du nœud.

Cette topologie est utilisée pour la construction des WAN. Ainsi on peut voir un WAN comme composé des stations *hôtes* (qui abritent les applications des utilisateurs) reliés entre elles par un *sous-réseau*. Le sous-réseau est constitué de nœuds (*routeurs*) et de lignes de transmission qui acheminent les informations d'un routeur à l'autre. Les liaisons sont à base de fil de cuivre, de fibres optiques ou même de liaisons radio.

Lors de la mise en place d'un réseau, il est indispensable de se préoccuper des coûts d'installation et de mettre en place une architecture permettant d'optimiser les performances. Ces choix s'appuient sur une mutualisation des ressources que l'on réalise en utilisant des matériels (*concentrateurs* et *multiplexeurs*) et des fonctions logicielles permettant de minimiser les lignes de transmissions entre les hôtes (la *commutation*).

En informatique traditionnelle (figure 15.5), les concentrateurs permettent l'utilisation d'une seule ligne de transmission pour l'accès à *n* terminaux. Le concentrateur possède une logique programmée lui permettant de distribuer la bonne information au bon terminal (les messages envoyés par le serveur et à destination des terminaux utilisent tous la même liaison de transmission).

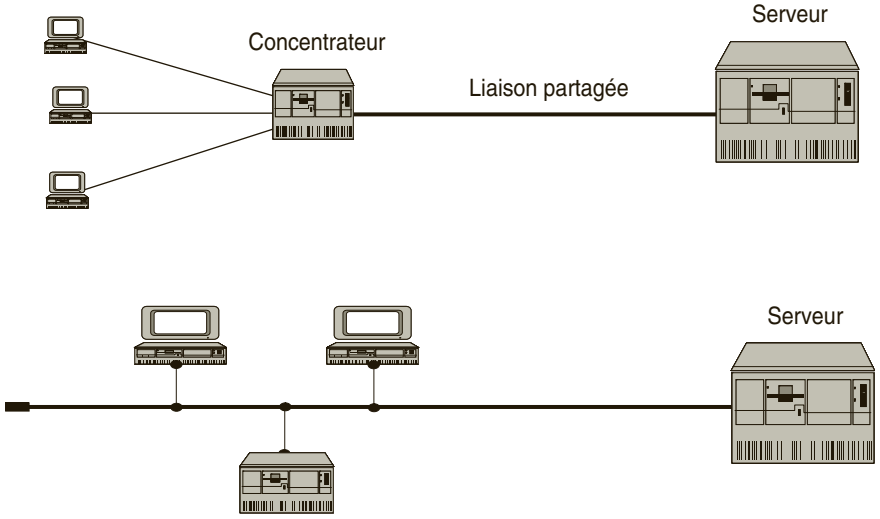


Figure 15.5 Concentrateurs.

L'arrivée des LAN tend à faire disparaître les concentrateurs (figure 15.5 en bas). La fonction de concentration est assurée par un ordinateur particulier du LAN. Cette architecture est utilisée dans le raccordement d'un réseau local à un WAN. Le LAN accède alors au sous-réseau du WAN par le biais d'un routeur qui joue également la fonction de concentration.

Le *multiplexage* (figure 15.6) permet le partage d'un lien unique par plusieurs utilisateurs.

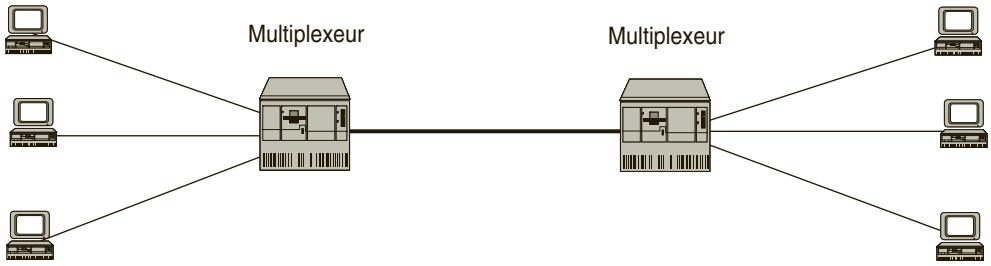


Figure 15.6 Multiplexeurs.

Le multiplexeur simule *n* liaisons point à point. Chaque station est reliée au multiplexeur par une liaison d'entrée, l'opération de regroupement des liaisons d'entrée correspond au multiplexage. La liaison point à point qui relie les multiplexeurs transporte alors un message composite. Le démultiplexage permet de restituer aux bons destinataires les données des différentes liaisons d'entrée.

On distingue le multiplexage *fréquentiel* (*FDM, Frequency Division Multiplexing*) et le multiplexage *temporel*.

Si l'on veut permettre la connexion de tous les hôtes d'un réseau, on est amené à établir un très grand nombre de liaisons physiques. Ainsi pour relier deux machines il faut une liaison, pour trois machines, il faut trois liaisons, etc. On montre facilement que pour relier N machines, il faut N(N – 1)/2 liaisons physiques. Si l'on applique ce résultat au réseau téléphonique qui comporte plusieurs centaines de millions d'abonnés dans le monde, il faudra de l'ordre de 10^{15} liens physiques chez chaque abonné pour qu'il puisse atteindre n'importe quel autre abonné du réseau.

La fonction de commutation permet à chaque abonné à partir d'un seul lien de communiquer avec n'importe quel autre abonné par simple *commutation* (figure 15.7).

Un tel réseau est dit *réseau à commutation*. Nous verrons plus en détail dans la partie transmission des données cette fonction de commutation.

Accès aux réseaux

Dans cette partie, nous présentons les moyens mis en œuvre par un opérateur réseau pour collecter le trafic des utilisateurs. Cette desserte des utilisateurs s'appelle la *boucle locale*. Au travers de cette boucle locale, un utilisateur accède aux réseaux des opérateurs permettant les transmissions haut débit.

La *fibre optique* est une première solution pour réaliser la boucle locale. Elle implique le câblage intégral du réseau de distribution en fibre optique. Cette technique, dite FITL (*Fiber In The Loop*) permet d'obtenir de hauts débits jusqu'au

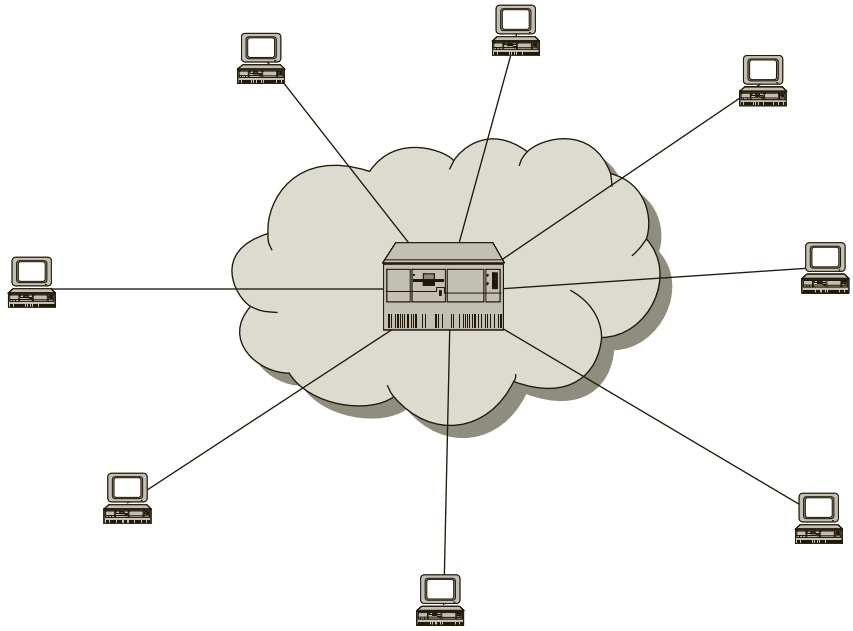


Figure 15.7 Principe de la commutation.

terminal utilisateur. Ce câblage est très onéreux et plusieurs solutions s'offrent pour réduire les coûts :

– FTTC (*Fiber To The Curb*) : le câblage optique est réalisé jusqu'à un point proche de l'immeuble. Le reste du câblage est réalisé par l'utilisateur final ;
– FTTN (*Fiber To The Node*) : le câblage optique est réalisé jusqu'à un répétiteur dans l'immeuble lui-même ;
– FTTH (*Fiber To The Home*) : le câblage est réalisé jusqu'à la porte de l'utilisateur ;
– FTTT (*Fiber To The Terminal*) : le câblage est réalisé jusqu'à la prise du terminal de l'utilisateur final.

La solution est d'autant plus onéreuse que l'on câble en fibre optique à proximité de l'utilisateur. La tendance actuelle est de câbler en fibre optique jusqu'à des points répartis dans les quartiers, la desserte des utilisateurs se fait par d'autres techniques moins onéreuses. Le câblage métallique permet de prendre en charge des débits de quelques mégabits, une solution consiste donc à câbler en fibre optique jusqu'à quelques kilomètres de l'utilisateur puis à raccorder l'utilisateur par du câblage métallique. Cette solution moins onéreuse est facilement réalisable dans les villes mais difficile hors agglomération où il faudra trouver d'autres solutions.

Les *paires métalliques* sont les supports les plus utilisés en raison des coûts même si les supports en fibre de verre et les supports hertziens se développent énormément. Les liaisons métalliques sont utilisées depuis très longtemps pour le transport de la parole en téléphonie. Ces dernières années, la mise en concurrence des télécommu-

nications a posé le problème du partage de cette ressource. L'ART (Autorité de Régulation des Télécommunications) a permis le partage de cette ressource par le *dégroupage* (figure 15.8) de la boucle locale.

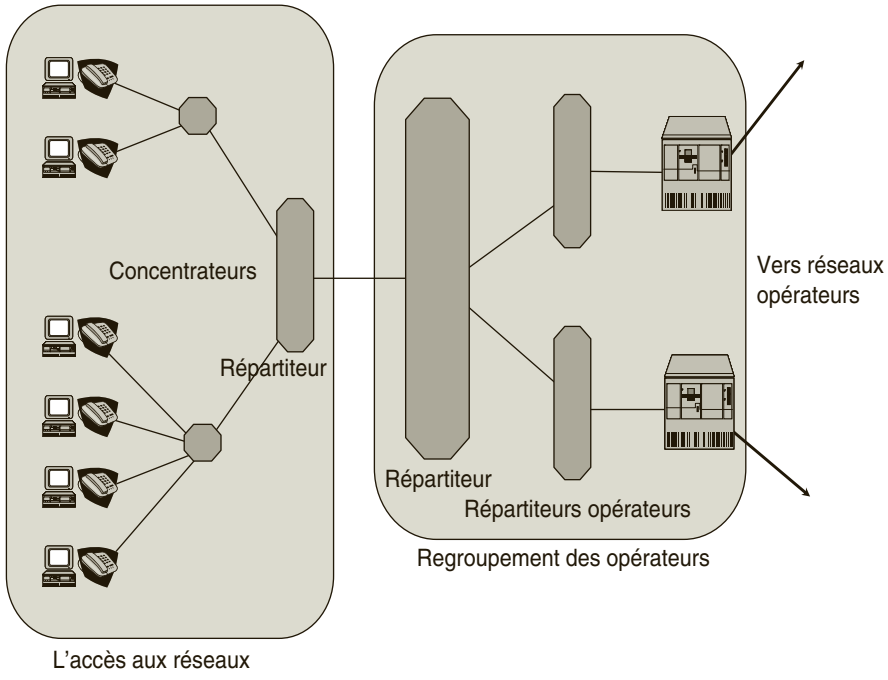


Figure 15.8 Dégroupage de la boucle locale.

On retrouve une topologie en étoile permettant aux utilisateurs d'accéder aux réseaux des opérateurs *via* leur téléphone, les concentrateurs, les sous-répartiteurs reliés aux répartiteurs principaux qui distribuent eux-mêmes sur les répartiteurs des opérateurs. Pour l'accès aux hauts débits il est possible d'utiliser les paires métalliques en s'appuyant sur les techniques DSL (*Data Subscriber Line*). Le sigle DSL indique que l'on utilise une ligne d'abonné pour les données.

La bande passante pour la voix nécessite 4 kHz alors que la bande passante possible pour la paire torsadée dépasse le MHz. La technologie ADSL (*Asymetric data rate Digital Subscriber Line*) réserve une bande passante pour le service de la voix en même temps qu'elle ouvre deux canaux de données, l'un dit montant qui offre une bande passante de 32 à 640 Kbits/s et l'autre dit descendant qui offre une bande passante de 1,5 à 8,2 Mbits/s.

L'accès haut débit au réseau *via* la ligne téléphonique (figure 15.9) nécessite l'installation d'un modem qui intègre la fonction (*splitter*) de séparation des canaux.

Le modem offre généralement des accès de type Ethernet et USB. Du côté de l'opérateur, le DSLAM (*Digital Subscriber Line Access Multiplex*) est un multiplexeur qui assure l'interface entre les connexions de l'utilisateur et le réseau haut

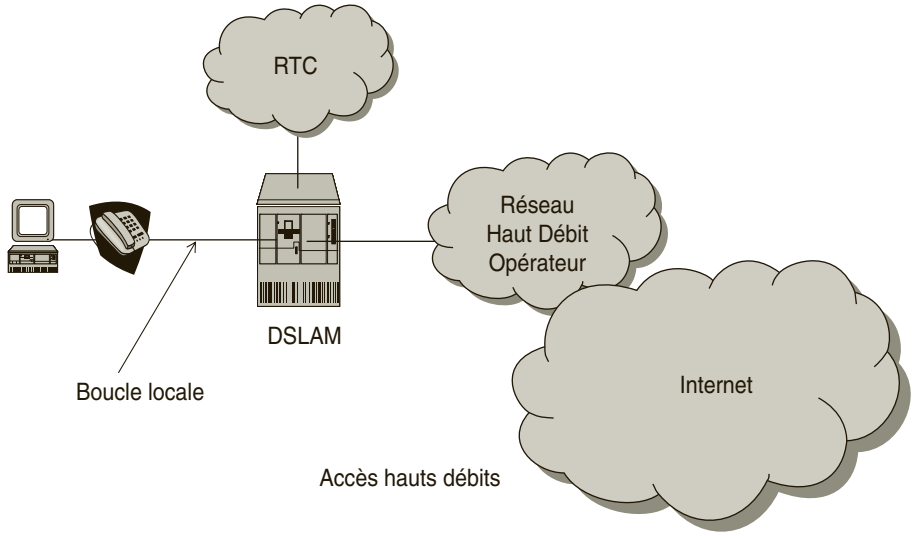


Figure 15.9 L'accès aux hauts débits.

débit de l'opérateur. Enfin pour que les transferts soient efficaces il ne faut pas que l'accès au fournisseur d'accès soit éloigné de plus de quelques kilomètres.

Il existe d'autres méthodes pour réaliser la boucle locale. À titre indicatif on notera l'utilisation des réseaux câblés de télévision (CATV) lorsqu'ils existent, et l'utilisation du réseau électrique. Dans ce dernier cas, les tests effectués dans de nombreux pays sont divergents. En tout état de cause, comme pour l'ADSL, il ne faut pas que l'accès aux fournisseurs soit éloigné de plus de quelques kilomètres.

15.2.2 Circulation des informations

Nous présentons dans cette partie les différents éléments permettant à des hôtes de correspondre entre eux au travers d'un réseau de transmission de données. Dans ce qui précède nous avons classé les réseaux en deux grandes catégories pour ce qui concerne la transmission des données : les réseaux à diffusion et les réseaux point à point.

Les réseaux à diffusion correspondent le plus souvent aux réseaux locaux. Ils permettent l'interconnexion des machines au travers d'un canal unique.

Les réseaux point à point, permettent l'interconnexion d'un très grand nombre de machines et correspondent plutôt aux WAN. De plus nous avons vu qu'afin de mutualiser les ressources, les WAN s'appuient sur la fonction commutation pour interconnecter les hôtes au travers du sous-réseau. Les informations à transporter sont découpées en *paquets* qui circulent au travers du sous-réseau. Ainsi un sous-réseau est composé de lignes de transmissions reliant les routeurs. Lorsque des routeurs veulent communiquer, ils devront le faire au travers de routeurs intermédiaires s'ils ne sont pas reliés directement par des lignes de transmission. Lorsqu'un paquet arrive sur un routeur intermédiaire, il est reçu intégralement, puis mis en attente jusqu'à ce que la liaison de sortie se libère. Ce mode de fonctionnement, le plus

général dans les WAN, est appelé transmission en mode différé (*store and forward*) ou à *commutation de paquets*. Dans ce mode de transmission, les paquets peuvent utiliser des chemins différents pour aller d'un hôte à un autre. Ce choix se fait par utilisation *d'algorithmes de routages* dont l'objet est de trouver le meilleur chemin pour aller d'un point à un autre.

Afin d'assurer la transmission des données il est nécessaire de préciser les notions d'adressage (qui permet d'identifier les machines et les applications du réseau), et les modalités d'acheminements des informations dans le réseau.

Adressage

Les réseaux permettent la communication d'informations. Il est donc nécessaire d'identifier d'une part les machines qui doivent communiquer et d'autre part les applications (programmes informatiques) qui produisent et reçoivent les informations.

Une adresse est une suite de caractères identifiant de manière unique et sans ambiguïté un point de raccordement à un réseau : adressage physique.

L'adresse est un identifiant qui permet l'acheminement des informations, au travers d'un ou plusieurs réseaux, de messages d'un correspondant émetteur vers un correspondant destinataire. Pour identifier un correspondant final il faut pouvoir identifier :

– le réseau auquel il est connecté ;
– le point d'accès auquel il est raccordé.

Ces deux champs d'adresse permettent d'identifier une machine d'un utilisateur dans le réseau. C'est l'adresse réseau d'une machine.

La machine maintenant identifiée il reste à déterminer quel est le processus sur cette machine qui est en charge de réceptionner le message pour le traiter. L'identification du processus dépend du système d'exploitation, il est donc nécessaire d'identifier les processus sans référence aux systèmes d'exploitation. Pour envoyer une information à un processus destinataire, le processus émetteur construit une structure de données contenant l'adresse réseau de l'ordinateur distant et le numéro du processus distant, et il fournit cette information au réseau. Le réseau connaissant ainsi l'adresse du destinataire pourra acheminer une information grâce aux mécanismes d'acheminement dont il dispose.

Des organismes de normalisation définissent l'adressage. L'ISO définit l'adressage dit NSAP (*Network Service Access point*) au travers d'une suite de caractères répartis en plusieurs champs :

– l'AFI (*Autority Format Identifier*) qui désigne l'autorité gestionnaire du domaine d'adressage ;
– l'IDI (*Initial Domain Identification*) qui identifie le domaine d'adressage ;
– DSP (*Domain Specific Part*) qui correspond à l'adresse effective de l'abonné.

Cette adresse peut être complétée par l'adresse du terminal (numéro du réseau et numéro de l'abonné dans le réseau).

Nous verrons un exemple du mécanisme d'adressage choisi dans le cas de l'internet.

Acheminement des données

Acheminer les données d'un point à un autre consiste à assurer le transit des paquets de données du point d'émission au point de réception. L'acheminement est réalisé au travers des algorithmes de routage qui permettent de déterminer un chemin au travers de l'ensemble des routeurs du sous-réseau. Chaque routeur contient des tables de routage qui indiquent la route à suivre pour atteindre un destinataire. On distingue de nombreux modes (ou protocoles) de routage :

- le routage statique ou fixe : on construit dans chaque nœud une table indiquant pour chaque destination l'adresse du nœud suivant. Ce routage est simple, mais il n'existe pas de solution de secours en cas de rupture d'un lien. Il convient bien aux petits réseaux. Il est mis en place par l'administrateur du réseau au moment de la configuration du réseau ;
- le routage par diffusion : l'information est routée simultanément vers plusieurs utilisateurs. Le message est dupliqué, cette technique oblige l'émetteur à connaître tous les destinataires. Cette technique surcharge le réseau ;
- le routage par inondation : chaque nœud envoie le message sur toutes les lignes de sortie, sauf celle d'où provient le message. Chaque message comporte un compteur de saut qui est initialisé au démarrage de l'émission et est décrémenté au passage de chaque nœud. Le message est détruit quand le compteur arrive à zéro. Cette technique (compteur de sauts) évite la surcharge du réseau. Cette technique est très robuste et garantit de toujours trouver le chemin le plus court ;
- le routage par le chemin le plus court : chaque nœud tient à jour des tables de routage indiquant quel est le chemin le plus court pour atteindre le routeur de destination. Chaque lien a un coût affecté ou calculé.

Ainsi les logiciels réseaux, s'appuyant sur les supports physiques, garantissent une qualité de l'acheminement au travers :

- du contrôle d'erreur qui permet de vérifier que tous les messages émis sont bien reçus par le destinataire ;
- du contrôle de flux, il s'agit ici de ne pas submerger le réseau à cause d'émetteurs trop rapides. Ce mécanisme implique l'existence de moyens de communication permettant à un destinataire d'informer l'émetteur qu'il doit bloquer momentanément l'émission pour ne pas saturer le destinataire. Ce contrôle de flux peut être exercé au niveau de chaque routeur ;
- d'un choix efficace de la route pour aller d'un émetteur à un destinataire. C'est la fonction de routage.

Services avec et sans connexion

Dans la transmission de données, on peut offrir deux types de services : avec connexion et sans connexion.

Un service avec connexion correspond au modèle téléphonique (composition du numéro distant, conversation, raccrochage). Dans le cas d'un service avec connexion, une connexion est établie, utilisée puis libérée. Ainsi les messages tran-

sitent *via* « un tuyau » construit entre l'émetteur et le destinataire. Tous les messages utilisent le même chemin, la route est calculée à la connexion.

Un service sans connexion correspond plus au modèle du système postal. Chaque message contient l'adresse du destinataire. La route est calculée pour chaque message. De ce fait, avec un tel service, on ne peut garantir que les messages émis dans un certain ordre arrivent dans le même ordre. L'ordre d'arrivée va dépendre des chemins suivis par chacun des messages.

Enfin chaque service est caractérisé par une *qualité de service*. Un service est dit fiable s'il ne perd jamais de données. Cette qualité de service est réalisée au moyen d'un mécanisme de notification par acquittement des données reçues. Cette technique provoque une surcharge du réseau et induit des délais qui justifient que parfois, on se dispense de la qualité de service. Un cas justifié de l'utilisation d'un service avec connexion et fiable est le transfert de fichiers d'un émetteur vers un destinataire. Dans ce cas, il ne faut en effet ne pas perdre d'information.

Tous les échanges ne nécessitent pas un service avec connexion. La diffusion d'information sous forme de petits messages autonomes ne nécessite pas de service avec connexion. De plus la livraison en mode fiable n'est pas non plus nécessairement requise. Un service sans connexion et non fiable (sans acquittement) est appelé transmission de *datagrammes*. On peut enfin réaliser une transmission pilotée par un service sans connexion (messages courts) avec fiabilité (équivalent d'une lettre recommandée). On a alors une transmission par *datagrammes acquittés*.

Enfin il existe un autre service de type *question-réponse*. On envoie un seul datagramme et l'on attend une réponse. Ce type de service est très utilisé dans le modèle client-serveur : le client émet une requête, le serveur répond à cette requête.

Bien que pouvant paraître étrange la communication non fiable se justifie par le fait que la fiabilité induit un ralentissement qui peut être inacceptable pour certaines applications : les applications en temps réels et multimédias par exemple.

15.2.3 Exemple de réseau filaire

Un réseau personnel : Ethernet

Ethernet est un réseau local (LAN) à diffusion dont les spécifications sont définies par la norme IEEE 802.3. Il est construit sur la base d'un canal unique et l'accès matériel se fait par le biais d'une carte Ethernet (figure 15.10). La diffusion s'appuie sur la méthode d'accès CSMA/CD qui permet la diffusion des messages à toutes les stations. Lorsqu'une station souhaite émettre, elle écoute le réseau, si le réseau est silencieux (aucune émission en cours) la station émet. Si le réseau est occupé, la station diffère son émission jusqu'à ce que le réseau soit libre.

Malheureusement cela ne garantit pas que deux stations ne détectent pas le silence du réseau en même temps. Dans ce cas, les deux stations émettent simultanément leur message. Le message résultant sur le réseau est pollué par l'autre (collision), le message est inexploitable. Aussi lorsqu'une station détecte une collision, elle cesse d'émettre.

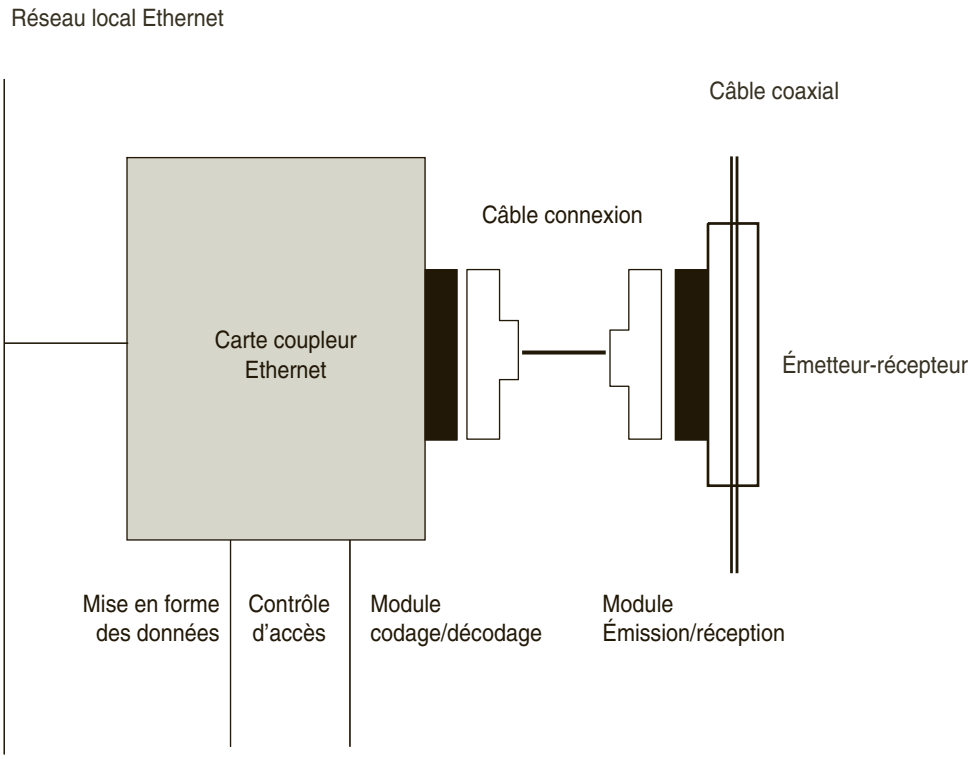


Figure 15.10 Ethernet accès au bus.

Pour détecter une collision, une station écoute le réseau en même temps qu'elle émet. Lorsqu'elle décèle une perturbation de son message (le niveau électrique ne correspond pas à l'émission), elle arrête l'émission et positionne un temporisateur aléatoire. À l'échéance du temporisateur, la station écoute le réseau et émet le message si le réseau est libre.

La figure 15.11 présente le schéma général d'un réseau Ethernet. En fait, il existe plusieurs versions de l'Ethernet qui se différencie par la norme, le type de câblage, les débits, les supports physiques. La figure 15.11 est donc une forme « générique » des réseaux Ethernet à diffusion.

La forme générale de la trame 802.3 commune à toutes les versions d'Ethernet est donnée par le tableau 5.1.

Tableau 15.1

Adresse Destination 6 octets	Adresse Source 6 octets	Longueur des données utiles 2 octets	Données utiles 46 octets ou bourrage	Bourrage Si données utiles < 46 octets	FCS 4 octets

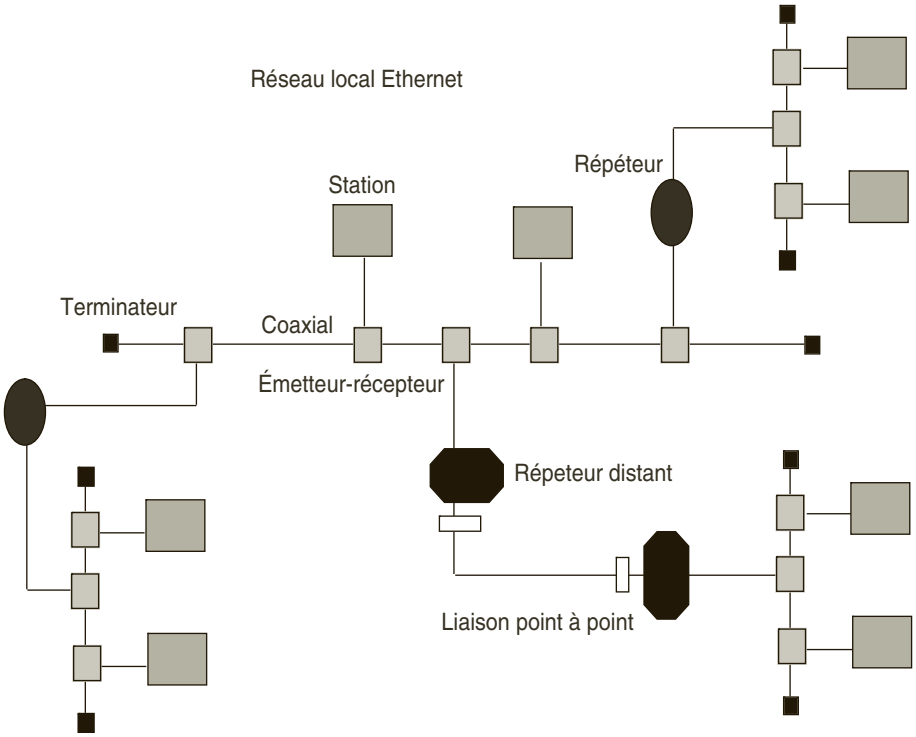


Figure 15.11 Ethernet architecture standard.

On trouve les différentes versions suivantes d'Ethernet :

– *Ethernet épais, IEEE 802.3 10 base 5* – Cette version fonctionne à 10 Mbits/s en bande de base sur câble coaxial d'une longueur maximale de 500 mètres par segment. Chaque station est équipée d'une interface NIC (*Network Interface Card*) généralement appelée carte Ethernet qui gère l'algorithme CSMA/CD. Un connecteur DB15 permet le raccordement du câble au transceiver (MAU : *Medium Attachement Unit*). Le MAU contient l'électronique qui assure la détection des collisions. On peut placer un maximum de 100 stations actives par segment de 500 m. On peut construire un réseau avec au maximum cinq segments et donc couvrir une surface de 2,5 km. Cette version d'Ethernet présente une grande difficulté de câblage et de ce fait est de moins en moins utilisée.
– *Ethernet fin, IEEE 802.3 10 base 2* – Elle correspond à une version économique (par rapport à 10 base 5) réalisée avec du coaxial fin. Dans cette version, les fonctions du transceiver sont intégrées à la carte. Le raccordement se fait par l'intermédiaire d'un T BNC (*Barrel Neck Connection*). La longueur d'un segment est de 185 m et chaque segment peut supporter 30 stations. Cette version économique est utilisée pour réaliser de petits réseaux.
– *Ethernet sur paire torsadée, IEEE 802.3 10 base T* – Le principe ici est d'utiliser le réseau téléphonique pour la réalisation du réseau. Le réseau devait passer d'une

topologie en bus à une topologie en étoile (réseau téléphonique), assurer la diffusion des messages et détecter les collisions. La solution consiste à émuler le bus par un boîtier : le hub, chargé de concentrer les connexions et d'assurer la diffusion des messages. Ce type de réseau fonctionne à 1 Mbits/s et la distance maximale entre une station et le hub est de 250 m. C'est cette architecture qui a donné naissance au 10 base T (*twisted pair*).

– *Ethernet à 100 Mbits/s* – Elle correspond à l'évolution de la version 10 base T (*Fast Ethernet*) et résulte des travaux du groupe de travail IEEE 802.14. Elle reprend la méthode d'accès CSMA/CD et conserve la taille des trames de 10 base T.
– *Ethernet Gigabit* – C'est une évolution du 100 Mbits/s, il fonctionne en diffusion mais aussi en commutation. Cette version Ethernet reprend donc les principes de la commutation mise en place dans les réseaux point à point. Le commutateur ne diffuse pas la trame mais la réémet vers le port unique correspondant au destinataire. La fonction en mode commuté permet de ne pas utiliser la fonction de détection de collision ainsi que l'émission et la réception simultanées (*full duplex*). Cette version Ethernet fonctionne selon différents modes : en full duplex (de commutateur à commutateur ou de commutateur à station) et en half duplex pour les stations directement raccordées au répéteur Ethernet Gigabit (*hub*). Les équipements Gigabit Ethernet combinent des ports à 10, 100, et plusieurs connexions sur fibre optique à 1 000 Mbits/s.

15.3 LES RÉSEAUX SANS FIL

Un réseau sans fil (*wireless network*) est un réseau permettant à plusieurs ordinateurs de communiquer sans liaison filaire. La communication s'effectue en utilisant des ondes du spectre électromagnétiques de la zone radio ou infrarouge à la place de signaux électriques véhiculés par l'intermédiaire de câbles.

Les réseaux sans fil permettent de relier facilement des ordinateurs distants d'une dizaine de mètres à quelques kilomètres sans nécessiter d'aménagements des infrastructures existantes. Ils accroissent la liberté de communication des équipements nomades qui peuvent se raccorder à un réseau filaire existant sans avoir recours à l'utilisation de câbles.

Les réseaux sans fil sont aptes à véhiculer des données texte, mais aussi de la voix et de la vidéo. De ce fait, ils permettent l'utilisation d'applications diverses comme la messagerie électronique, le Web ou la téléphonie.

Ils posent cependant deux difficultés :

– l'utilisation d'ondes radioélectriques ne permet pas le confinement à un espace géographique donné des informations circulant sur le réseau. Il est donc plus aisé à un pirate d'écouter ce réseau ;
– la transmission par ondes radioélectriques est sensible aux interférences. Comme de nombreuses applications utilisent ce type de transmission, il est nécessaire de définir par réglementation des plages de fréquences affectées à chaque type d'utilisation.

Les réseaux sans fils peuvent être répartis en quatre classes, selon leur zone de couverture, c'est-à-dire selon le périmètre dans lequel une connexion est possible :

– les réseaux personnels sans fil WPAN (*Wireless Personal Area Network*) concernent les réseaux d'une portée de quelques dizaines mètres. Ce type de réseau sert généralement à relier des périphériques tels qu'une imprimante, un scanner ou un assistant personnel (PDA) à un ordinateur sans liaison filaire. La principale technologie de ce type est la technologie *Bluetooth*, lancée par Ericsson en 1994, proposant un débit théorique de 1 Mbits pour une portée maximale d'une dizaine de mètres ;
– les réseaux locaux sans fil WLAN (*Wireless Local Area Network*) sont des réseaux dont la portée est d'environ une centaine de mètres. Il existe plusieurs technologies de ce type, la plus connue étant celle du Wi-Fi (*Wireless Fidelity*). Elles offrent des débits allant jusqu'à 54 Mbits sur une distance de plusieurs centaines de mètres ;
– les réseaux métropolitains sans fil WMAN (*Wireless Metropolitan Area Network*) offrent un débit utile de 1 à 10 Mbits/s pour une portée de 4 à 10 kilomètres ;
– les réseaux étendus sans fil WWAN (*Wireless Wide Area Network*) possèdent une zone de couverture à l'échelle d'un pays (centaine de kilomètres). Ils offrent un débit maximum de 170 kbit/s.

15.3.1 Architecture des réseaux sans fil

L'architecture d'un réseau sans fil (figure 15.12) s'appuie sur des composants similaires à ceux d'un réseau filaire, cependant ceux-ci sont adaptés afin de permettre une transmission des signaux sous la forme d'ondes de la zone radio.

L'équipement (imprimante, téléphone mobile, ordinateur portable, PDA, etc.) héberge une carte réseau qui réalise l'interface entre l'infrastructure du réseau sans fil et l'équipement. Cette carte réseau dispose d'une antenne dont le rôle est d'assurer la conversion entre les signaux numériques et les ondes radio.

Actuellement, les principales cartes réseaux disponibles sont les suivantes :

– Carte PCI, gérées par le bus PCI de l'équipement ;
– PC-Card et CardBus développées dans les années 90 par l'association PCMCIA (*Personal Computer Memory Card International Association*). Ce type de carte a la taille d'une carte de crédit et sert à doter un équipement de petite taille ;
– Mini-PCI, est une version réduite de la carte PCI, destinée aux ordinateurs portables ;
– CompactFlash, introduite en 1994, est très petite et quatre fois moins encombrante qu'une PC-Card. Elles sont actuellement disponibles sur certains PDA ;
– les cartes adaptatrices PCMCIA permettent l'utilisation d'une carte PC-Card ou CardBus sur l'interface PCI d'un équipement fixe. Ainsi la même carte peut être utilisée sur un ordinateur portable ou un ordinateur fixe ;
– Les modules USB Wi-Fi se branchent sur l'interface USB de l'équipement. Ces modules peuvent prendre la forme de clés USB (*dongles*) et associent alors souvent à la carte réseau une mémoire de stockage de 64 Mo ou davantage.

Deux types d'architectures sont définis : le *mode infrastructure* et le *mode ad-hoc*.

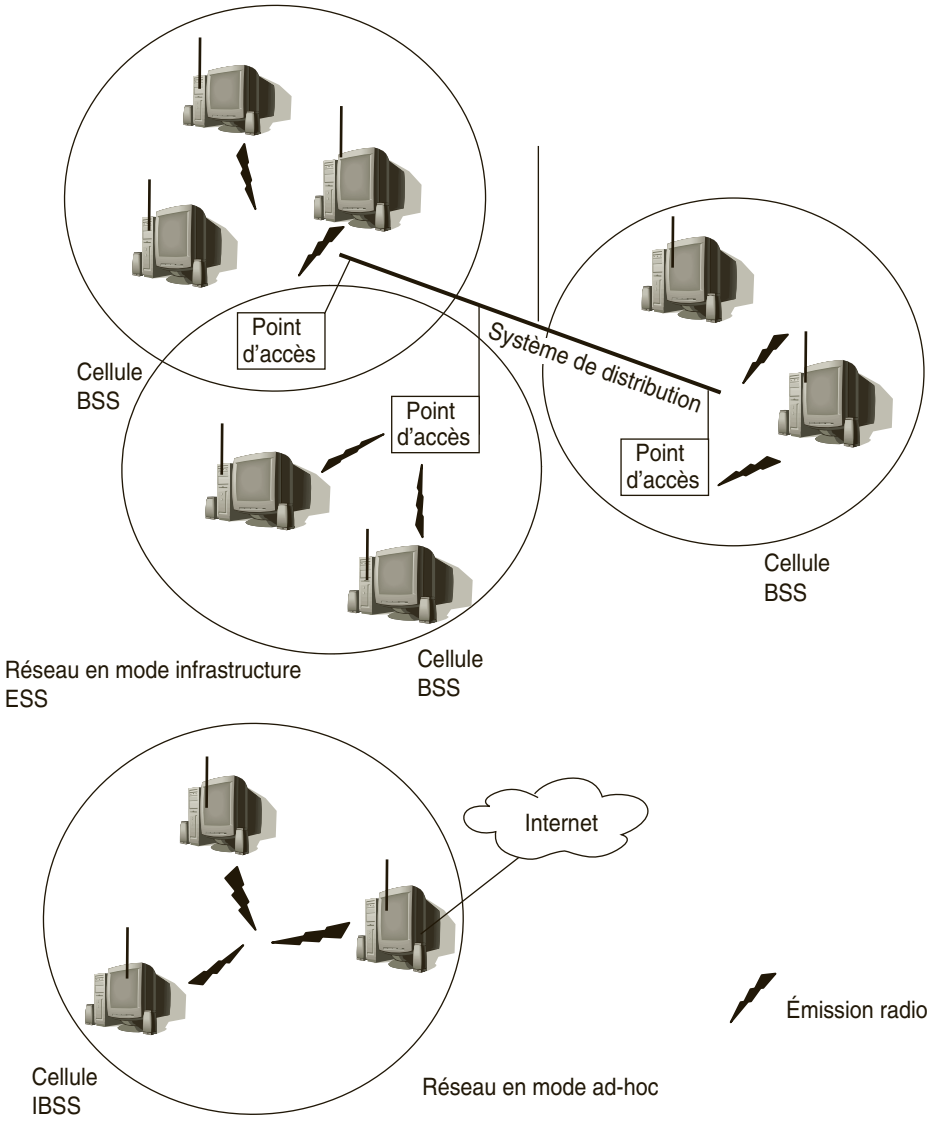


Figure 15.12 Architecture des réseaux sans fil.

Dans le mode infrastructure, des points d'accès (*AP, Access Point*) permettent l'échange d'informations entre les équipements du réseau sans fil. Dans une *cellule*, c'est-à-dire dans la zone de couverture d'un point d'accès, toute communication entre deux équipements de cette cellule passe ainsi obligatoirement par l'intermédiaire du point d'accès. Les cellules ainsi définies peuvent être disjointes ou se recouvrir.

– La topologie BSS (*Basic Service Set*) comporte un seul point d'accès. Celui-ci peut gérer jusqu'à 30 équipements qui se partagent le support de transmission ainsi que le débit ;

– la topologie ESS (*Extended Service Set*) comporte plusieurs points d'accès, connectés entre eux par l'intermédiaire d'un système de distribution (DS, *Distribution System*) qui assurent l'échange de trames entre les différents points d'accès. Ce système de distribution est le plus souvent implémenté sous la forme d'un réseau Ethernet mais il peut également prendre la forme d'un autre réseau sans fil. Il peut aussi au travers d'une passerelle servir d'interface vers un autre réseau de type filaire tel qu'internet.

Dans le mode ad-hoc ou IBSS (*Independant Basic Set Service*), les différents équipements communiquent directement entre eux sans passer par un point d'accès. Dans le cas le plus simple, un tel réseau est composé de deux équipements munis de leur carte réseau sans fil. Dans ce type d'architecture, l'un des équipements peut être connecté à un réseau tel qu'Internet et offrir l'accès à ce réseau aux autres équipements qui sont dans sa zone de couverture.

15.3.2 Circulation des informations

Deux bandes de fréquences dites sans licence, c'est-à-dire n'imposant pas à l'utilisateur d'acquérir un droit pour leur usage, sont disponibles pour les réseaux sans fil. Ce sont les fréquences suivantes :

– la bande ISM (*Industrial Scientific and Medical*) composée de trois sous-bandes 902-908 MHz, 2,400-2,4835 GHz et 5,725-5,825 GHz. En Europe, seules les deux premières sous-bandes sont utilisées ;
– la bande U-NII (*Unlicensed National Information Infrastructure*) composée des deux sous-bandes 5,15-5,35 GHz et 5,725-5,825 GHz.

Dans ces bandes de fréquence, un canal correspondant à une certaine largeur de bande est affecté à chaque cellule. Des cellules disjointes peuvent utiliser les mêmes canaux au sein d'une plage de fréquences tandis que des cellules qui se recouvrent doivent utiliser des canaux différents pour éviter les interférences.

Accès au support

Deux protocoles d'accès au support sont définis pour les réseaux sans fil :

– DCF (*Distributed Coordination Function*) : cette méthode d'accès est une méthode avec contention, c'est-à-dire que des collisions peuvent se produire lorsque deux équipements émettent en même temps. Elle est utilisée pour la transmission de données asynchrones. Elle est utilisée en mode infrastructure ou en mode ad-hoc ;
– PCF (*Point Coordination Function*) : cette méthode ne génère pas de collisions dans le sens où c'est le point d'accès qui gère les différentes transmissions de données en donnant à tour de rôle accès au support aux équipements qui ont des données à émettre. Elle est utilisée pour la transmission de données isochrones ou temps réel (voix, vidéo) pour lesquelles les retards de transmission dus aux collisions ne sont pas tolérables. Elle est utilisée seulement en mode infrastructure.

Le DCF étant la méthode principale, nous en présentons brièvement le principe ci-dessous.

➤ DCF : un aperçu des mécanismes mis en œuvre

Le DCF s'appuie sur deux mécanismes distincts :

– le protocole CSMA/CA (*Carrier Sense Multiple Access with Collision Avoidance*). Dans ce protocole, un équipement souhaitant envoyer des données écoute le support de communication avant d'émettre ou diffère sa transmission si le support est occupé ;
– le mécanisme de réservation qui permet à un équipement souhaitant émettre de réserver le support de communication durant la totalité de sa transmission.

Ainsi un équipement A souhaitant émettre suit l'algorithme suivant (figure 15.13) :

1. l'équipement A écoute le support ;
2. le support est libre durant un temps appelé DIFS ; l'équipement A émet les données ;
3. le support n'est pas libre, alors l'équipement A attend que le support se libère ;
4. le support étant libéré, l'équipement A attend un temps DIFS et si le support est toujours libre, retarde encore son émission d'un temps aléatoire $T_{back\text{-}off}$ calculé selon un algorithme appelé algorithme du back-off ;
5. le temporisateur de back-off écoulé, si le support est libre, l'équipement A émet ;
6. les autres équipements (équipement D), en attente pour émettre, bloquent l'écoulement de leur propre temporisateur de back-off. À l'issue de l'émission, et après avoir vérifié que le support est libre au bout d'un temps DIFS, ils reprennent l'écoulement de leur temporisateur.

L'équipement B destinataire quant à lui suit l'algorithme suivant :

1. il vérifie par l'intermédiaire d'un champ de contrôle associé à la trame reçue que la transmission est correcte ;
2. il attend un temps appelé SIFS pour émettre une trame d'acquittement ACK.

L'équipement émetteur en l'absence de réception de cette trame d'acquittement, considère qu'une collision s'est produite et réémet les données.

Le tirage d'un temps aléatoire $T_{back\text{-}off}$ par chaque équipement souhaitant émettre permet d'éviter des collisions, chaque équipement tirant potentiellement une valeur différente. Cependant deux équipements peuvent tirer une valeur égale. Une collision se produit qui n'est détectée que par l'absence de réception d'une trame d'acquittement par l'équipement émetteur (figure 15.14).

Par ailleurs, l'équipement émetteur peut avoir recours à un mécanisme de réservation du support avant toute émission de données vers un équipement destinataire. Ce mécanisme assure l'absence de collision durant l'échange. Il est facultatif et peut être surtout utilisé lors de la transmission de trames de données de grande taille.

Dans ce cas (figure 15.15) :

1. l'équipement émetteur envoie une trame de réservation RTS. Cette trame contient notamment des informations sur le volume de données à émettre et sa vitesse de transmission ;

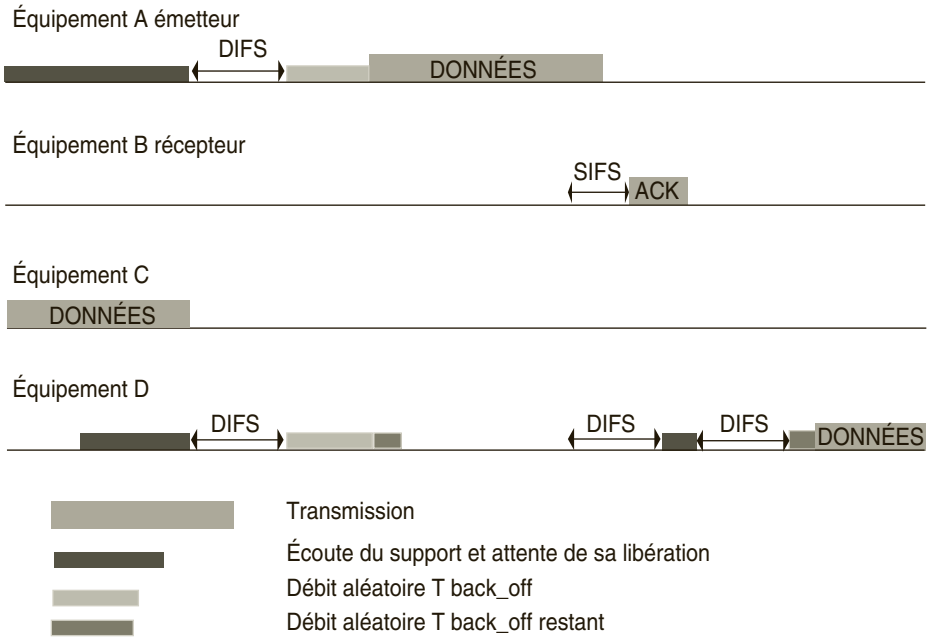


Figure 15.13 Protocole CSMA/CA.

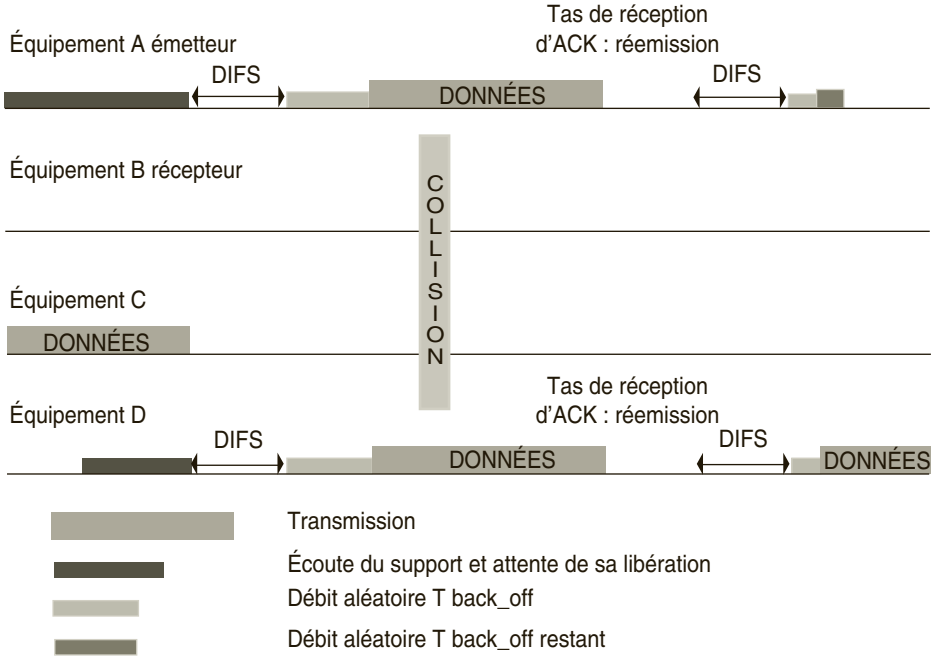


Figure 15.14 Protocole CSMA/CA avec collision.

2. les équipements de la cellule qui entendent la trame extraient de celle-ci les informations de volume et de vitesse et calculent à partir de ces données une durée d'occupation du support ;
3. l'équipement destinataire répond par une trame CTS ;
4. l'équipement émetteur reçoit la trame CTS. Elle est assurée que le support est réservé pour la totalité de sa transmission de données. Elle commence à émettre.

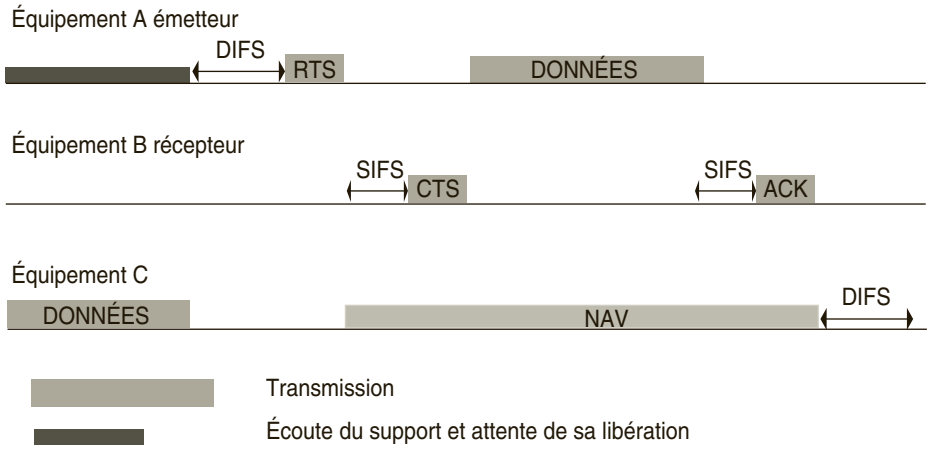


Figure 15.15 Réservation du support par les trames RTS et CTS.

Transmission des données sur le support

Des techniques de transmission particulières, dites *techniques d'étalement de bandes*, sont mises en œuvre sur les réseaux sans fil afin d'atteindre des débits de dizaine de mégabits malgré l'étroitesse des bandes passantes disponibles, ceci afin de rendre les réseaux sans fil concurrentiels par rapport aux réseaux filaires.

La technique FHSS (*Frequency Hopping Spread Spectrum*) est l'une de ces techniques d'étalement de bande. Elle est fondée sur le principe de saut de fréquence. La bande des 2,4 GHz est divisée en 79 canaux de 1 GHz chacun. Afin de minimiser les collisions, un échange de données s'effectue alors par l'intermédiaire de sauts entre les différents canaux, ces sauts se réalisant toutes les 300 ms. Pour communiquer, l'émetteur et le récepteur se mettent au préalable d'accord sur une séquence de sauts. Ce mécanisme permet théoriquement 26 réseaux simultanés, mais dans la pratique, à cause des recouvrements partiels des canaux, seuls 15 réseaux simultanés sont autorisés.

La technique DSSS (*Direct Sequence Spread Spectrum*) est une seconde méthode d'étalement de bande. Elle divise en 14 canaux de 20 MHz la bande de fréquence 2,4 GHz. Une transmission s'effectue en totalité sur un même canal. Cependant, les 14 canaux se recouvrant partiellement, seuls trois réseaux simultanés peuvent émettre sans risque d'interférences sur une même cellule.

Ces deux techniques sont illustrées par la figure 15.16.

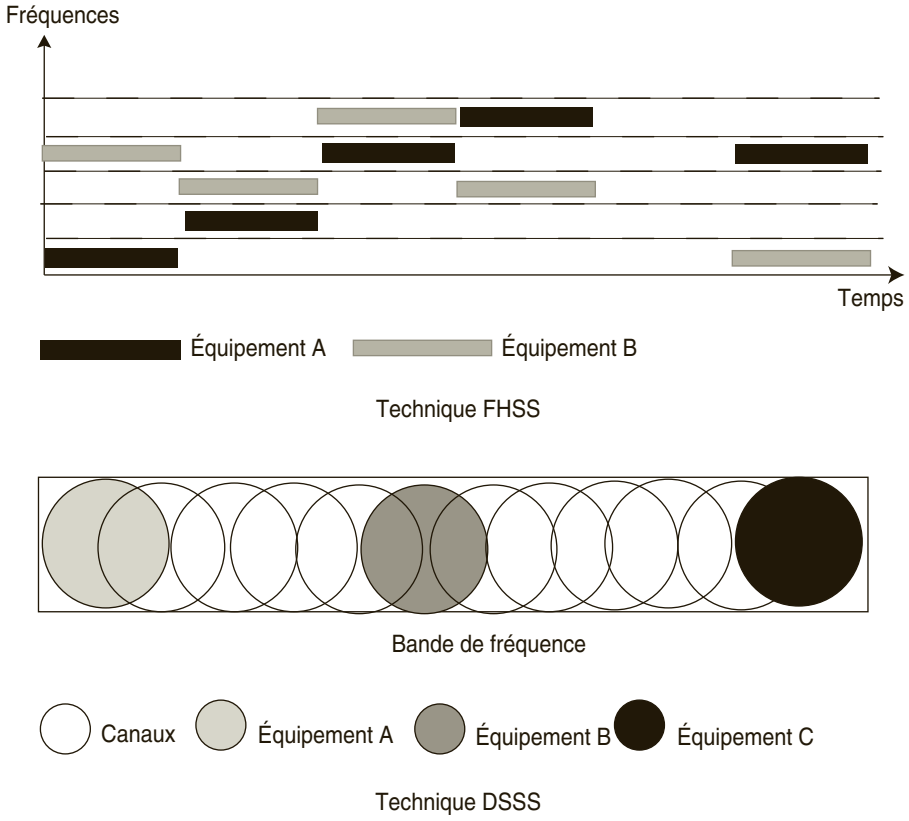


Figure 15.16 Techniques FHSS et DSSS.

Sécurité

Les problèmes de sécurité constituent l'un des points faibles des réseaux sans fil. En effet, tout équipement présent dans la zone de couverture d'une cellule peut en capter les émissions ou même reconfigurer le réseau.

Les mécanismes suivants sont implémentés afin d'empêcher les écoutes clandestines et les tentatives d'accès non autorisés :

– identifiant du réseau SSID (*Service Set ID*) : tout équipement souhaitant intégrer une cellule doit fournir au point d'accès le nom du réseau qu'il souhaite rejoindre, le SSID ;
– ACL (*Access Control List*) : le point d'accès maintient une liste des adresses des cartes réseau sans fil (adresses MAC) autorisées dans la cellule. Cela permet de limiter l'accès au réseau à un certain nombre de machines ;

– protocole WEP (*Wired Equivalent Privacy*) : ce protocole utilise un mécanisme de chiffrement de données et d'authentification qui permet d'éviter les écoutes clandestines et assure également de contrôler l'identité des équipements autorisés dans la cellule.

Dans ces mécanismes, seul le premier est obligatoire. Le SSID étant configuré au point d'accès lors de la construction de celui-ci, il est nécessaire de le modifier lors de la mise en place du réseau, afin qu'un équipement malveillant ne puisse le déduire en fonction de la marque et du constructeur du point d'accès.

15.3.3 Exemples de réseaux sans fil

Un réseau personnel

Bluetooth est une technologie développée en 1994 par Ericsson et répondant à la norme IEEE 802.15 (*Institute of Electrical and Electronics Enginners*).

Ce réseau personnel utilise la bande de fréquence des 2,4 GHz, avec un débit de 1 Mbits/s. Les transmissions s'effectuent selon la technique d'étalement de bandes FHSS que nous avons présentée au paragraphe Y.

Ce réseau possède une zone de couverture d'une dizaine de mètres et autorise le raccordement qu'équipements tels qu'un ordinateur, un PDA, une imprimante, un scanner sans liaison filaire.

La configuration de ce réseau est de type ad-hoc. Deux équipements se connectant constituent un réseau Bluetooth que l'on désigne sous le vocable *piconet (figure 15.17)*. L'équipement initiant la connexion est maître du piconet et les autres équipements jouent le rôle d'esclave. Au maximum, un équipement maître peut établir une connexion avec sept équipements esclaves.

Un équipement peut appartenir à plusieurs piconets, mais il ne peut être l'équipement maître que pour un seul d'entre eux.

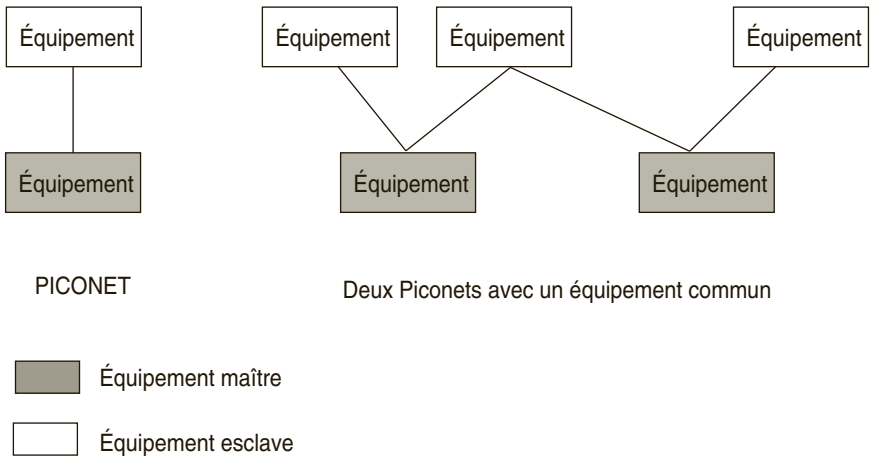


Figure 15.17 Configuration du réseau Bluetooth.

Un réseau local

Les réseaux locaux sans fil répondent à la norme IEEE 802.11. Bien qu'issues de la même norme, de nombreuses implémentations peuvent être distinguées, principalement celles répondant à la certification Wi-Fi et par exemple, le réseau HiperLAN/2.

Wi-Fi est une certification apposée sur des équipements de réseaux sans fil dès lors que ceux-ci répondent à un certain nombre de critères prédéfinis. Cette certification a été mise en place par l'organisme international Wi-Fi Alliance dans les années 2000 afin de rendre interopérables un ensemble d'implémentations diverses de la norme 802.11.

Les réseaux locaux sans fil certifiés Wi-Fi reprennent la plupart des mécanismes que nous avons présentés au paragraphe précédent : techniques d'étalement de bandes, accès au support *via* le protocole CSMA/CA, sécurité grâce au protocole WEP.

HiperLAN/2 (*High Performance Radio LAN*) est une implémentation distincte de réseau local sans fil développée par la division BRAN (*Broadband Radio Access Networks*) de L'ETSI (European *Telecommunications Standards Institute*). Ce réseau utilise la bande des 5 GHz, avec des débits de 54 Mbits/s. Il n'utilise pas le protocole CSMA/CA pour gérer l'accès au support mais le protocole TDMA (*Time Division Multiple Access* expliqué au paragraphe suivant) qui permet de donner à chaque équipement un accès régulier au support à la différence de CSMA/CA. Pour cette raison, ce réseau local est plus performant dans le transfert de données isochrones telles que la voix et la vidéo. Cependant ce réseau est encore au stade de prototype, sans version commercialisée à ce jour.

Un réseau grande distance

Les réseaux satellites, utilisés notamment dans la diffusion télévisée et aussi pour assurer des connexions à Internet, constituent un exemple de réseaux grand distance sans fil.

Dans ce type de réseau, un équipement au sol envoie des signaux radio en direction du satellite par le biais d'une antenne parabolique. Le satellite que l'on peut assimiler à un point d'accès est équipé d'un circuit répéteur spécialisé, le transpondeur, qui capte le signal et le réémet vers un autre équipement terrestre.

La plupart des satellites sont placés à différents endroits de l'orbite géostationnaire, soit à une distance de 35 900 km de la Terre. Ils supportent un trafic bidirectionnel dans la bande de fréquences située entre 450 MHz et 20 GHz. Dans cette bande, les basses fréquences sont réservées à la liaison descendante tandis que les hautes fréquences sont réservées à la liaison montante.

La grande distance que doivent parcourir les signaux radios, soit 35 900 kilomètres, porte le délai de transmission à une valeur située autour de 100 ms et la durée du trajet Terre-Satellite-Terre à 200 ms. De ce fait, les protocoles d'accès mis en œuvre dans la norme 802.11 et notamment le protocole CSMA/CA nécessitant un acquittement pour chaque paquet transmis sont inefficaces.

Les réseaux satellites utilisent des protocoles d'accès au support dédiés. Ces protocoles réservent un canal pour chaque équipement, ce qui permet de réduire

considérablement les problèmes d'interférences entre différentes émissions. Ce sont essentiellement les trois protocoles suivants :

– TDMA (*Time Division Multiple Access*) : la bande de fréquence est divisée en sous-canaux, et chaque sous-canal est assigné à un équipement. Les équipements peuvent donc émettre simultanément, chacun dans la bande de fréquence qui lui est réservée ;
– FDMA (*Frequency Division Multiple Access*) : à chaque équipement est assignée une tranche de temps durant lequel il peut émettre. Les équipements émettent donc à tour de rôle ;
– CDMA (*Code Division Multiple Access*) : chaque équipement voit son signal modulé au moyen d'un code différent. Les transmissions sont alors simultanées et occupent toute la bande de fréquence sans risque d'interférences.

15.4 L'INTERCONNEXION DE RÉSEAUX : INTERNET

Il s'agit d'un inter-réseau c'est-à-dire d'une interconnexion de réseaux comme l'indique la figure 15.18.

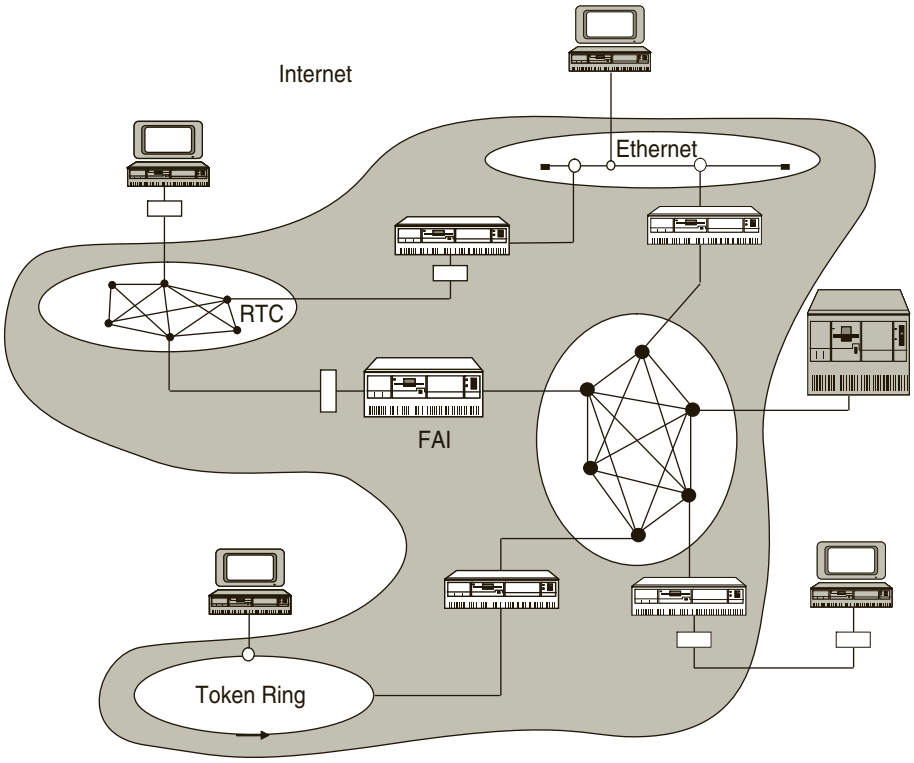


Figure 15.18 Une représentation de l'Internet.

Dans ce qui suit nous allons présenter les caractéristiques essentielles de l'Internet en suivant le mode de présentation que nous avons adopté lors des paragraphes précédents de ce chapitre.

15.4.1 Architecture de l'Internet

Supports physiques et topologies

Le réseau Internet doit être considéré comme un réseau logique d'interconnexion de réseaux physiques. On ne peut donc pas parler de supports physiques et de topologies propres à internet. On doit plutôt considérer que l'Internet intègre tous les supports physiques et toutes les topologies de réseaux existants. C'est ce que symbolise la figure 15.18.

Les accès à l'Internet

Du point de vue des accès, on trouve la réalisation de la boucle locale aussi bien au travers de réseaux filaires (liens RTC hauts débits, réseaux locaux filaires), qu'au travers de réseaux sans fils (liaisons satellites par exemple).

15.4.2 Circulation de l'information

Adressage

Internet est un réseau logique pour lequel est défini le mécanisme d'adressage que rappelle la figure 15.19.

L'adressage Internet permet de spécifier une adresse réseau pour une machine (adresse IP) et l'adresse du processus qui va traiter les données sur la machine dont l'adresse IP est spécifiée.

L'adresse IP est codée sur 32 bits et est organisée en classes d'adresses (A, B, C). Pour chacune des classes, on code le numéro du réseau et le numéro de la machine dans ce réseau :

– classe A : 128 réseaux et 16 777 216 hôtes (7 bits pour le réseau et 24 pour les hôtes) ;
– classe B : 16 384 réseaux et 65 535 hôtes (14 bits pour les réseaux et 16 pour les hôtes) ;
– classe C : 2 097 152 réseaux et 256 hôtes (21 bits pour les réseaux et 8 pour les hôtes).

Il s'agit là du codage de l'adresse correspondant à la première génération IP (IPv4). Le codage de l'adresse est en cours d'évolution vers le IPv6 qui correspond à la nouvelle génération de l'Internet.

Afin d'assurer l'unicité des numéros de réseau, les adresses Internet sont attribuées par un organisme central, le NIC (*Network Information Center*).

Codées sur 32 bits, les adresses Internet sont trop limitées pour relier toutes les machines. Dans la version IPv6, les adresses seront codées sur 128 bits ce qui donne un très grand confort d'utilisation.

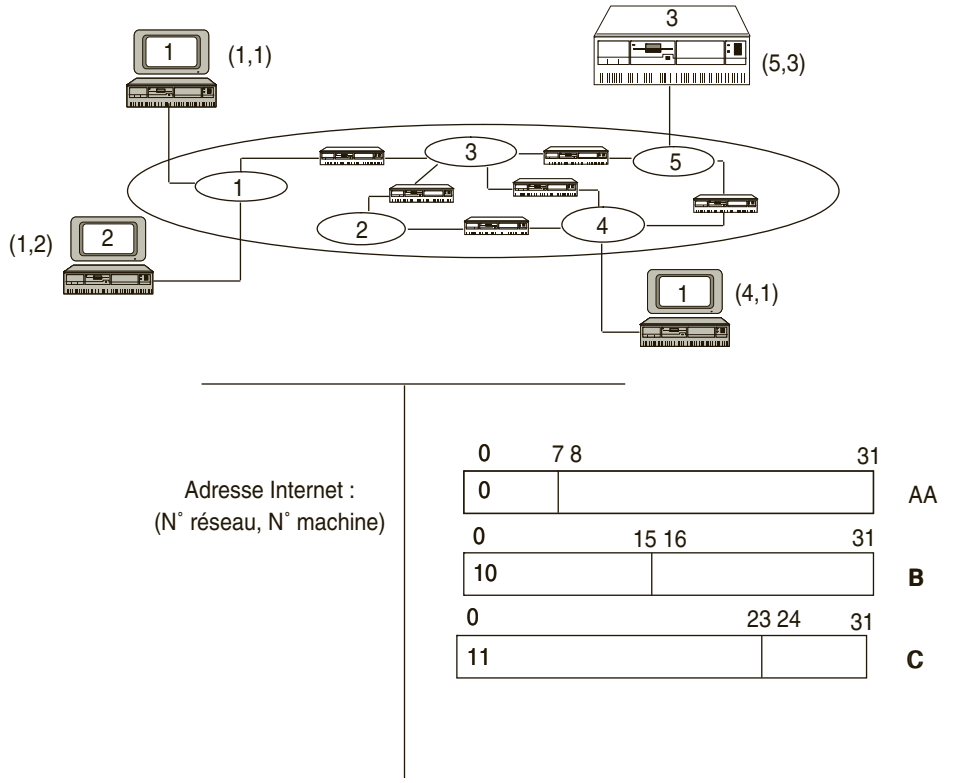


Figure 15.19 Adressage Internet.

Internet est un réseau logique et les adresses des machines sont construites indépendamment des adresses physiques de ces machines. Ainsi pour envoyer un datagramme sur Internet le logiciel réseau doit convertir l'adresse Internet en une adresse physique. Il s'agit de la résolution des adresses qui permet de déterminer l'adresse d'un équipement à partir de l'adresse Internet.

Il existe deux protocoles importants permettant la résolution d'adresse entre le monde IP et le monde Ethernet : ARP (*Adress Resolution Protocol*) et RARP (*Reverse ARP*).

Manipuler les adresses physiques (32 ou 128 bits) dans le corps des applications est complexe, aussi l'Internet définit des adresses logiques construites sur une hiérarchie de domaines permettant de définir des adresses symboliques plus simples à manipuler : `ceante.cnam.fr` est la machine `ceante` du domaine `cnam` appartenant au domaine France.

Pour réaliser la traduction d'une adresse exprimée logiquement (hiérarchie de domaines en adresse Internet codée sur 32 bits), l'Internet utilise des serveurs de noms (*DNS, Domain Name Server*) pouvant répondre aux requêtes de résolution des noms.

La transmission des messages se fait entre des applications qui s'exécutent sur des machines. Les adresses IP permettent d'identifier les machines mais pas les applica-

tions. Le problème est d'identifier un processus indépendamment du système d'exploitation afin de pouvoir faire communiquer des machines hétérogènes au plan des systèmes d'exploitation. Pour identifier un processus d'un point de vue « réseau », on lui attribue un numéro de *port* qui caractérise le processus. Ce numéro de port est associé (dans des tables systèmes) à un nom de processus du système.

Ainsi l'adresse complète d'un processus est un doublet constitué de l'adresse IP de la machine et du numéro de port du processus.

Transmission de l'information

L'Internet permet de définir les modalités de transmission de l'information d'un point à un autre.

Les messages sont découpés en paquets (figure 15.20) qui peuvent être acheminés soit selon un service avec connexion fiable (Mode TCP/IP), soit en mode datagramme (service sans connexion non fiable).

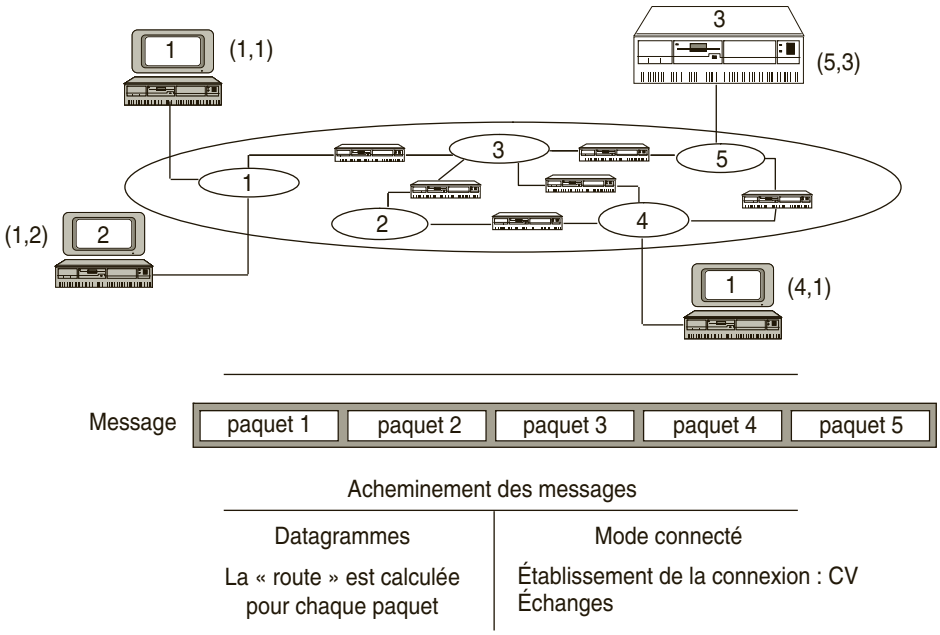


Figure 15.20 L'acheminement des données.

On peut noter une particularité des transmissions des données dans l'Internet. Le protocole de base est *l'Internet Protocol* qui fonctionne en mode datagramme. Pour assurer un service avec connexion fiable, l'Internet fonctionne de bout en bout, c'est-à-dire qu'il n'effectue les vérifications que sur la machine distante (au travers du protocole TCP) et non pas sur chacun des routeurs intervenant dans les réseaux physiques traversés. Si le réseau physique fonctionne bien cela confère à Internet une plus grande efficacité dans la transmission des données.

Pour résumer et conclure, une application voulant transmettre des données à une application située sur une machine distante disposant d'une adresse Internet, utilise :

– une structure de donnée permettant de spécifier l'adresse de l'application distante ;
– une structure de donnée permettant de spécifier les modalités de l'échange (TCP/IP ou mode datagramme) ;
– un ensemble de fonctions permettant de réaliser l'échange soit en mode connecté fiable soit en mode datagramme.

Chapitre **16**

Exercices corrigés

ORDONNANCEMENT DE PROCESSUS

16.1 Algorithmes d'ordonnancement

On considère 5 processus P1, P2, P3, P4 et P5 dont les caractéristiques sont résumées dans le tableau suivant (un petit numéro de priorité indique une priorité forte).

	Ordre d'arrivée	Temps d'exécution	Priorité
P1	1	2	2
P2	2	6	4
P3	3	10	3
P4	4	4	5
P5	5	12	1

Pour chacune des politiques d'ordonnancement « premier arrivé, premier servi », « plus court d'abord », priorité fixe, tourniquet avec un quantum de temps égal à 2 unités, donnez l'ordre de service des processus et le temps de réponse moyen obtenu.

16.2 Ordonnancement par priorité préemptif et non préemptif

On considère 5 processus P1, P2, P3, P4 et P5 dont les caractéristiques sont résumées dans le tableau suivant (un petit numéro de priorité indique une priorité forte).

	Date d'arrivée	Temps d'exécution	Priorité
P1	2	2	1
P2	4	6	4
P3	5	3	2
P4	0	4	5
P5	0	7	3

Représentez l'ordre d'exécution des processus dans le cas d'un ordonnancement par priorité non préemptif, puis dans le cas d'un ordonnancement préemptif. Dans chacun des cas, donnez le temps d'attente moyen.

16.3 Chronogramme d'exécutions

On considère 3 processus P1, P2 et P3 qui effectuent chacun du calcul sur un processeur et des entrées-sorties avec un disque. L'ordonnancement des processus sur le processeur se fait selon une politique de priorité préemptive, avec priorité (P1) > priorité (P2) > priorité (P3). Un seul processus à la fois peut être servi par le disque et la réalisation d'une entrée-sortie est non préemptive. L'ordre de service sur le disque est selon un mode « premier arrivé, premier servi ».

Les processus s'exécutent selon les profils suivants. Ils sont tous les trois prêts en même temps à l'instant 0.

P1	P2	P3
Calcul durant 4 unités Entrées-sorties durant 2 unités Calcul durant 3 unités Entrées-sorties durant 2 unités Calcul durant 1 unité	Calcul durant 3 unités Entrées-sorties durant 1 unité Calcul durant 2 unités	Calcul durant 7 unités

Représentez l'exécution des processus dans le temps en mentionnant les changements d'états intervenant entre l'état prêt, l'état élu et l'état bloqué.

16.4 Ordonnancement sous Unix

Les opérations relatives à l'ordonnancement sont réalisées par le système Unix à chaque fois qu'un processus s'apprête à passer du mode système au mode utilisateur, donc à la suite de l'occurrence soit d'une interruption, soit d'un appel système, soit d'une trappe. La politique d'ordonnancement du système Unix est une politique à multiples niveaux de priorité avec quantum de temps. Le noyau recalcule la priorité d'un processus quand il passe du mode noyau au mode utilisateur. Ce calcul permet une extinction de priorité des processus qui s'exécutent, de manière à permettre l'exécution de tous les processus et à éviter les problèmes de famine. L'algorithme

mis en œuvre s'appuie sur la routine d'interruption horloge qui fait régulièrement passer le processus en mode noyau. Plus précisément :

– à chaque interruption horloge, le système incrémente d'une unité la valeur du compteur « utilisation du processeur » pour le processus élu ;
– toutes les secondes c'est-à-dire toutes les 50 à 100 interruptions horloge, la valeur du compteur « utilisation du processeur » pour tous les processus est divisée par deux ;
– enfin, la priorité des processus est recalculée selon la formule :

priorité du processus = (utilisation du processeur/2)
+ (priorité de base du niveau utilisateur)

Du fait de ce recalcul des priorités, les processus se déplacent dans les files de priorité.

On considère à présent trois processus A, B et C qui ont chacun une priorité initiale de 60. La priorité de base du niveau utilisateur est également la priorité 60. L'interruption horloge se déclenche 60 fois au cours d'une seconde. Sur un intervalle de temps égal à 5 secondes, déterminez l'ordre d'élection des trois processus A, B et C.

16.5 Ordonnancement sous Linux

On considère un système monoprocesseur de type Linux dans lequel les processus partagent un disque comme seul ressource autre que le processeur. Cette ressource n'est accessible qu'en accès exclusif et non requérable, c'est-à-dire qu'une commande disque lancée pour le compte d'un processus se termine normalement avant de pouvoir en lancer une autre. Un processus peut être en exécution, en attente d'entrées-sorties, en entrées-sorties ou en attente du processeur. Les demandes d'entrées-sorties sont gérées à l'ancienneté.

Dans ce système, on considère 4 processus P1, P2, P3, P4 pour lesquels on sait que :

– P1 et P2 sont des processus appartenant à la classe SCHED_FIFO. Dans cette classe, le processeur est donné au processus de plus haute priorité. Ce processus peut être préempté par un processus de la même classe ayant une priorité supérieure ;
– P3 et P4 sont des processus appartenant à la classe SCHED_RR. Dans cette classe, le processeur est donné au processus de plus haute priorité pour un quantum de temps égal à 10 ms. La politique appliquée est celle du tourniquet.

Les processus de la classe SCHED_FIFO sont toujours plus prioritaires que les processus de la classe SCHED_RR.

Les priorités des processus sont égales à 50 pour le processus P1, 49 pour le processus P2, 49 pour le processus P3 et 49 pour le processus P4. La plus grande valeur correspond à la priorité la plus forte.

Les 4 processus ont le comportement suivant :

• Pl :
  – Calcul pendant 40 ms.
  – Lecture disque pendant 50 ms.
  – Calcul pendant 30 ms.

  - Lecture disque pendant 40 ms.
  - Calcul pendant 10 ms.
- P2 :
  - Calcul pendant 30 ms.
  - Lecture disque pendant 80 ms.
  - Calcul pendant 70 ms.
  - Lecture disque pendant 20 ms.
  - Calcul pendant 10 ms.
- P3 :
  - Calcul pendant 40 ms.
  - Lecture disque pendant 40 ms.
  - Calcul pendant 10 ms.
- P4 :
  - Calcul pendant 100 ms.

Établissez le chronogramme d'exécution des quatre processus en figurant les états prêt, élu, en attente d'entrées-sorties et en entrées-sorties.

SYNCHRONISATION DE PROCESSUS

16.6 Producteur(s)-Consommateurs(s)

Soit un système composé de trois processus cycliques Acquisition, Exécution et Impression, et de deux tampons Requête et Avis gérés circulairement, respectivement composés de m et n cases.

Le processus Acquisition enregistre chacune des requêtes de travail qui lui sont soumises par des clients puis il les place dans le tampon Requête à destination du processus Exécution.

Le processus Exécution exécute chaque requête de travail prélevée depuis le tampon Requête et transmet ensuite au processus Impression un ordre d'impression de résultats déposé dans le tampon Avis.

Le processus Impression prélève les ordres d'impression déposés dans le tampon Avis et exécute ceux-ci.

1. Programmez la synchronisation des trois processus à l'aide des sémaphores et des variables nécessaires à la gestion des tampons.
2. On étend le système à trois processus Acquisition, trois processus Exécution et trois processus Impression. Complétez la synchronisation précédente pour que celle-ci demeure correcte.

16.7 Allocations de ressources et interblocage

On considère un système composé de 3 processus respectivement appelés Acquisition, Collecteur et Bilan. On dispose par ailleurs d'une seule ressource disque et d'une seule ressource imprimante.

Le processus Acquisition effectue toutes les t unités de temps la lecture de mesures à partir de trois capteurs et dépose les 3 grandeurs lues dans un tampon de 3 cases. Une fois les 3 dépôts effectués par le processus Acquisition, le processus Collecteur lit le contenu du tampon de 3 cases, enregistre les données collectées sur le disque et imprime celles-ci. Le processus Bilan est quant à lui réveillé toutes les semaines. Il lit depuis le disque toutes les mesures enregistrées sur la semaine, effectue un traitement sur celles-ci et imprime le résultat de ce traitement.

1. Sachant que l'unité de transfert vers l'imprimante est la ligne de texte, que les impressions envoyées sur l'imprimante par les processus comportent toutes plusieurs lignes, montrez que des incohérences peuvent survenir sur les impressions effectuées par les processus de ce système. Proposez une solution.
2. Voici les pseudo-codes des processus Collecteur et Bilan. Les sémaphores `imp` et `disque` sont respectivement initialisés à 1. Montrez que ces codes peuvent conduire à une situation d'interblocage entre ces deux processus puis proposez une solution.

Collecteur	Bilan
début	début
boucle	boucle
lire_mesures(tampon, mesures);	P(disque);
P(imp);	P(imp);
P(disque);	lire_disque(mesures);
enregistrer_disque(mesures);	res = traitement(mesures);
imprimer(mesures);	imprimer(res);
V(disque);	V(imp);
V(imp);	V(disque);
fin boucle	fin boucle
fin	fin

3. Le processus Collecteur ne doit prélever les mesures dans le tampon de 3 cases que lorsque les 3 dépôts ont été effectués par le processus Acquisition. Par ailleurs, le processus Acquisition ne doit pas faire de nouveau dépôt tant que les dépôts du cycle précédent n'ont pas été lus par le processus Collecteur. À l'aide de sémaphores, établissez le schéma de synchronisation permettant de respecter ces 2 contraintes.

16.8 Allocation de ressources et états des processus

On considère un ensemble de deux ressources R à accès exclusif et un type de processus utilisateur de la ressource R dont le pseudo-code est de la forme :

```
processus utilisateur de R
début
    P(R);
    utilisation de R en effectuant un appel système bloquant;
    V(R);
fin
```

1. À quelle valeur doit-on initialiser le sémaphore R permettant de gérer l'accès à l'ensemble des deux ressources de type exclusif ?
2. On considère trois processus P1, P2, P3 utilisateurs de la ressource R selon le code donné ci-dessus. Donnez la suite des états pour chacun de ces processus, sachant que l'ordre d'élection des processus est P1, puis P2, puis P3. Les trois processus sont prêts en même temps, à un instant t = 0. Le temps de blocage sur l'appel système est important au regard du temps nécessaire aux autres opérations.

GESTION DE LA MÉMOIRE CENTRALE

16.9 Gestion de la mémoire par partitions variables

Soit un système qui utilise l'allocation par partitions variables. On considère à l'instant t, l'état d'allocation de la mémoire centrale représenté sur la figure 16.1, où les zones grisées représentent les zones libres.

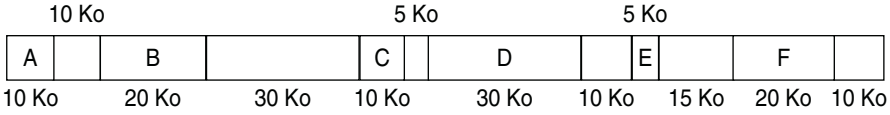


Figure 16.1 Allocation de la mémoire à t.

1. Représentez l'évolution de la mémoire centrale, suite à chacun des événements suivants, en supposant une stratégie de choix *First Fit* :
   1) arrivée du programme G de taille égale à 20 Ko ;
   2) départ du programme B ;
   3) arrivée du programme H de taille égale à 15 Ko ;
   4) départ du programme E ;
   5) arrivée du programme I de taille égale à 40 Ko.
2. Même question mais en supposant une stratégie de choix *Best Fit*.

16.10 Remplacement de pages

Soit la liste des pages virtuelles référencées aux instants t = 1, 2, …, 11 :

3 5 6 8 3 9 6 12 3 6 10

La mémoire centrale est composée de 4 cases initialement vides. Représentez l'évolution de la mémoire centrale au fur et à mesure des accès pour chacune des deux politiques de remplacement de pages FIFO et LRU. Notez les défauts de pages éventuels.

16.11 Mémoire paginée et segmentée

On considère une mémoire segmentée paginée pour laquelle les cases en mémoire centrale sont de 4 Ko. La mémoire centrale compte au total 15 cases numérotées de 1 à 15. Dans ce contexte, on considère deux processus A et B. Le processus A a un espace d'adressage composé de trois segments S1A, S2A et S3A qui sont respectivement de 8 Ko, 12 Ko et 4 Ko. Le processus B a un espace d'adressage composé de deux segments S1B et S2B qui sont respectivement de 16 Ko et 8 Ko Pour le processus A, seules les pages 1 et 2 du segment S1A, la page 2 du segment S2A et la page 1 du segment S3A sont chargées en mémoire centrale respectivement dans les cases 4, 5, 10, 6. Pour le processus B, seules les pages 2 et 3 du segment S1B et la page 1 du segment S2B sont chargées en mémoire centrale respectivement dans les cases 11, 2 et 15.

1. Représentez sur un dessin les structures allouées (table des segments, tables des pages) et la mémoire centrale correspondant à l'allocation décrite.
2. Soit l'adresse logique <S1A, page 1, 12>. Quelle adresse réelle lui correspond-elle ?
3. Soit l'adresse logique <S2B, page 2, 10>. Quelle adresse réelle lui correspond-elle ?
4. Dans ce même contexte, donnez pour chacune des adresses linéaires suivantes, son équivalent en adresse virtuelle, puis son adresse physique correspondante : 4 098 pour le processus A, 12 292 pour le processus A, 8 212 pour le processus B.

16.12 Mémoire virtuelle et ordonnancement de processus

On considère un système fonctionnant selon le principe de la mémoire virtuelle. Les pages des processus sont chargées à la demande c'est-à-dire seulement lorsque le processus demande à accéder à la page. Lors d'un défaut de page, une opération d'entrées-sorties est lancée qui coûte 10 ms par page à charger.

Soient les trois processus P1, P2 et P3 dont les espaces d'adressages sont respectivement composés de 3, 3 et 2 pages. Les 3 processus sont tous les 3 prêts à l'instant t = 0. L'ordonnancement sur le processeur est un ordonnancement par priorité préemptif. Le processus P1 est le processus le plus prioritaire et le processus P3 est le processus le moins prioritaire.

Le comportement des processus est le suivant :

- Pl :
  - Calcul utilisant la page 1 pendant 20 ms.
  - Calcul utilisant la page 2 pendant 10 ms.
  - Calcul utilisant la page 1 et la page 2 pendant 30 ms.
  - Calcul utilisant la page 3 pendant 30 ms.

- P2 :
  - Calcul utilisant la page 1 et la page 2 pendant 40 ms.
  - Calcul utilisant la page 1 et la page 3 pendant 20 ms.
  - Calcul utilisant la page 1, la page 2 et la page 3 pendant 50 ms.
- P3 :
  - Calcul utilisant la page 1 pendant 20 ms.
  - Calcul utilisant la page 2 pendant 20 ms.

Établissez le chronogramme d'exécution des 3 processus en figurant les états prêt, élu, et bloqué c'est-à-dire en opération d'entrées-sorties pour défaut de page.

16.13 Pagination à la demande

On considère une mémoire paginée pour laquelle les cases en mémoire centrale sont de 1 Ko. La mémoire centrale compte au total pour l'espace utilisateur 20 cases numérotées de 1 à 20. Dans ce contexte, on considère trois processus A, B et C. Le processus A a un espace d'adressage composé de six pages P1, P2, P3, P4, P5 et P6. Le processus B a un espace d'adressage composé de quatre pages, P1 à P4. Le processus C a un espace d'adressage composé de deux pages, P1 et P2. Pour le processus A, seules les pages P1, P5, P6 sont chargées en mémoire centrale respectivement dans les cases 2, 4, 1. Pour le processus B, seule la page P1 est chargée en mémoire centrale dans la case 5. Pour le processus C, seule la page P2 est chargée en mémoire centrale dans la case 12.

1. Représentez sur un dessin les structures allouées pour ce type d'allocation mémoire et la mémoire centrale correspondant à l'allocation décrite.
2. Les trois processus A, B et C sont décrits par un bloc de contrôle qui contient entre autre les informations suivantes :
   - Pour le processus A, compteur ordinal CO = (page P5, déplacement 16), adresse table des pages = 16 ;
   - Pour le processus B, compteur ordinal CO = (page P2, déplacement 512), adresse table des pages = 64 ;
   - Pour le processus C, compteur ordinal CO = (page P1, déplacement 32), adresse table des pages = 128.

   Le compteur ordinal CO contient l'adresse de l'instruction à exécuter.

   Le processus A devient actif. Décrivez le processus de conversion d'adresse pour l'instruction exécutée à sa reprise. Quelle valeur contient le registre PTBR ? Quelle adresse physique correspond à l'adresse virtuelle de l'instruction exécutée ?

   Maintenant le processus A est préempté et le processus B est élu. Décrivez succinctement l'opération de commutation de contexte qui a lieu notamment en donnant les nouvelles valeurs des registres CO et PTBR. Que se passe-t-il lorsque le processus B reprend son exécution ?
3. Chaque entrée de table des pages contient un champ de bits permettant de spécifier les droits d'accès associés à une page. Ce champ est composé de trois bits x, r, w avec la signification suivante :

– x : 0 pas de droit en exécution sur la page, 1 droit en exécution accordé ;
– r : 0 pas de droit en lecture sur la page, 1 droit en lecture accordé ;
– w : 0 pas de droit en écriture sur la page, 1 droit en écriture accordé.

Ce champ « droit » à la valeur 010 pour la page P1 du processus A. Le processus A exécute l'instruction STORE R1 (page P1, déplacement 128) qui effectue l'écriture du contenu du registre processeur R1 à l'adresse (page P1, déplacement 128). Que se passe-t-il ?

SYSTÈME DE GESTION DE FICHIERS

16.14 Modes d'accès

1. On considère un fichier à accès direct. Le dernier enregistrement lu par le système dans ce fichier est l'enregistrement 512. Combien faut-il lire d'enregistrements pour accéder à chacun des enregistrements suivants : enregistrement 513, enregistrement 1 024, enregistrement 77 ?
2. On considère à présent un fichier séquentiel. Le dernier enregistrement lu par le système dans ce fichier est l'enregistrement 512. Combien faut-il lire d'enregistrements pour accéder à chacun des enregistrements suivants : enregistrement 513, enregistrement 1 024, enregistrement 77 ?

16.15 Organisation de fichiers

1. On considère un fichier à allocation contiguë constitué de blocs de 1 024 octets. L'adresse du premier bloc constituant le fichier est mémorisée dans l'entrée du répertoire concernant ce fichier. Le dernier bloc lu par le système dans ce fichier est le bloc 512. Combien faut-il lire de blocs physiques pour accéder à chacun des blocs suivants : bloc 513, bloc 1 024, bloc 77 ?
2. On considère à présent un fichier à allocation chaînée constitué de blocs de 1 024 octets. L'adresse du premier bloc constituant le fichier est mémorisée dans l'entrée du répertoire concernant ce fichier. Le dernier bloc lu par le système dans ce fichier est le bloc 512. Combien faut-il lire de blocs physiques pour accéder à chacun des blocs suivants : bloc 513, bloc 1 024, bloc 77 ?

16.16 Noms de fichiers et droits d'accès

1. La figure 16.2 représente une arborescence de fichiers. Donnez le nom absolu de chacun des fichiers apparaissant dans cette arborescence.
2. Le système de protection des fichiers gère trois niveaux de protection. Chaque niveau de protection définit des droits en lecture (r), écriture (w) et exécution (x) pour les éléments du niveau. Le premier niveau ne contient que le créateur du fichier. Le second niveau correspond au groupe d'utilisateurs auquel appartient le créateur du ficher, enfin le troisième niveau concerne tous les autres utilisateurs en

dehors du créateur et du groupe du créateur. Le système définit deux groupes, les étudiants et les professeurs.

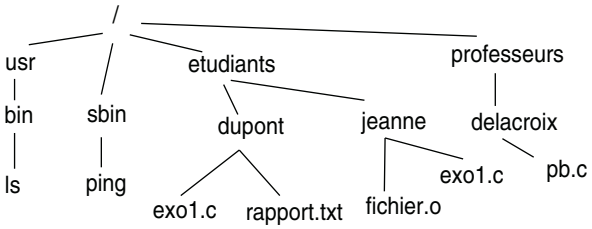


Figure 16.2 Arborescence de fichiers.

Jeanne appartient au groupe des étudiants. Les droits associés au fichier `exo1.c` dont elle est le créateur sont `rwx` pour le premier niveau, `r – x` pour le second niveau, `r – –` pour le troisième niveau. L'utilisateur delacroix appartenant au groupe des professeurs peut-il modifier ce fichier ?

16.17 Algorithmes de services des requêtes disque

On considère un disque composé de 300 pistes numérotées de 0 à 299. Le bras est couramment positionné sur la piste 50. La liste des requêtes (numéro de piste cherchée) à servir donnée selon l'ordre d'arrivée est la suivante :

`62, 200, 150, 60, 12, 120, 250, 45, 10, 100.`

Donnez l'ordre de service des requêtes et le déplacement de bras total en résultant dans le cas d'un service FCFS, d'un service SSTF et d'un service LOOK sens initial montant.

16.18 Fichiers Unix

Un processus Unix lit séquentiellement un fichier de 8 Mo, à raison de 256 octets à la fois. On suppose que les blocs disque sont de 1 024 octets et qu'un numéro de bloc occupe 4 octets. Par ailleurs, le temps d'accès moyen au disque est de 50 ms.

1. Le système ne gère pas de mécanisme de cache au niveau du disque, c'est-à-dire que chaque demande de lecture équivaut à une opération d'entrées-sorties. Donnez le nombre total d'accès disque nécessaire et le temps d'attente en entrées-sorties.
2. Le système gère un mécanisme de cache au niveau du disque, qui mémorise en mémoire centrale les 100 blocs disques les plus récemment accédés. Donnez le nombre total d'accès disque nécessaire et le temps d'attente en entrées-sorties.

16.19 Système de gestion de fichiers FAT

Soit l'allocation du disque C sur un système Windows représentée par la figure 16.3. L'allocation est gérée selon un système de FAT. Les blocs sont numérotés de 1 à 20,

ligne à ligne, de gauche à droite. Les rectangles blancs sont des blocs libres. Les numéros apparaissant dans les blocs colorés correspondent au numéro du fichier auquel le bloc appartient (exemple : 1 pour fichier_1).

Représentez le contenu des vingt premières entrées de la table en fonction de l'allocation figurée par la figure 16.3.

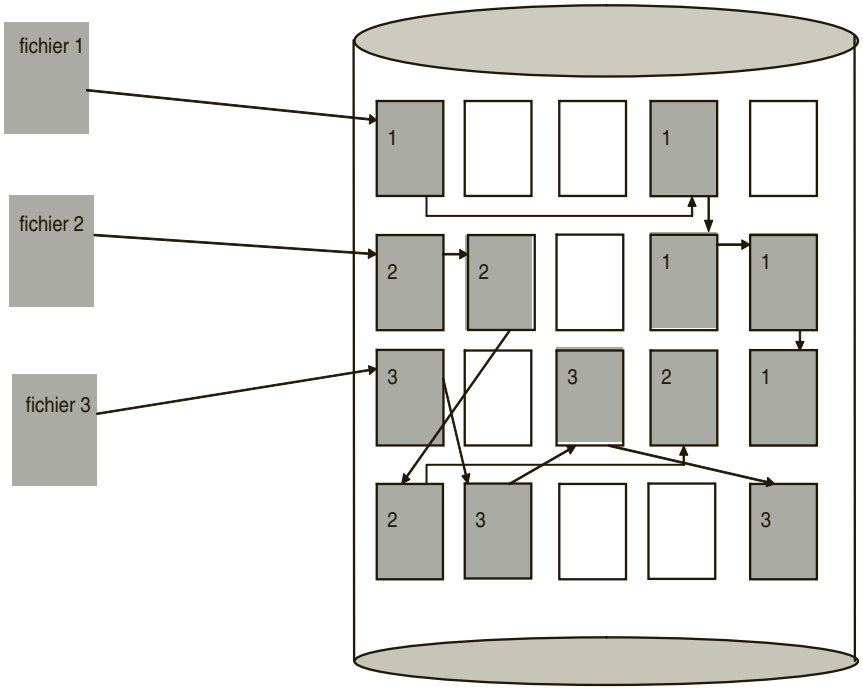


Figure 16.3 Allocation du disque C.

16.20 Système de gestion Unix

On considère un fichier de type UNIX. Le fichier a une taille de 16 Moctets. Les blocs disque sont de 1 024 octets. Un numéro de bloc occupe deux octets.

1. Quel est le nombre de blocs de données du fichier ?
2. Combien comporte-t-il de blocs d'adresses ?

16.21 Synthèse

On considère les trois processus suivants caractérisés par :

– une date de soumission : date de l'arrivée du processus dans la file des processus prêts ;
– un temps d'exécution total ;
– une priorité fixe (plus la valeur de priorité est petite, plus la priorité est grande).

Processus	Date de soumission (t)	Temps d'exécution (s)	Priorité
P1	6	12	1
P2	0	9	3
P3	4	18	2

1. On ordonnance ces trois processus selon une politique de priorité préemptive puis non préemptive. Représentez l'exécution des trois processus pour chacun de ces deux cas. Donnez leur temps de réponse.
2. Les trois processus P1, P2 et P3 font à présent des entrées-sorties en utilisant un disque non requérable. Les demandes d'entrées-sorties des trois processus sont traitées selon un ordre FIFO. Sur le processeur, les processus sont ordonnancés selon un mode de priorité préemptive. Les dates de soumission et les priorités des processus demeurent inchangés.

   Les trois processus ont chacun une exécution définie comme ci-dessous :

P1	P2	P3
4 secondes calcul 5 secondes E/S disque 4 secondes calcul 2 secondes E/S disque 4 secondes calcul	3 secondes calcul 6 secondes E/S disque 6 secondes calcul	6 secondes calcul 5 secondes E/S disque 6 secondes calcul 3 secondes E/S disque 6 secondes calcul

   Représentez l'exécution des trois processus en précisant leurs états prêt, élu, bloqué. Donnez leur temps de réponse.
3. Les trois processus P1, P2 et P3 font des entrées-sorties en utilisant un disque pour lequel l'allocation des blocs physiques est gérée selon un système de gestion de fichiers de type FAT. On suppose que ce disque comporte 30 blocs physiques numérotés de 1 à 30. Quatre fichiers sont alloués sur ce disque.
   - Le fichier 1 nommé fich_1, d'une taille de 5 632 octets, comporte dans l'ordre les blocs 1, 5, 9, 15, 25, 29.
   - Le fichier 2 nommé fich_2, d'une taille de 4 098 octets, comporte dans l'ordre les blocs 3, 2, 7, 11, 30.
   - Le fichier 3 nommé fich_3, d'une taille de 6 272 octets, comporte dans l'ordre les blocs 4, 8, 17, 12, 13, 16, 21.
   - Le fichier 4 nommé fich_4, d'une taille de 3 104 octets, comporte dans l'ordre les blocs 22, 24, 6, 10.

   Donnez la FAT correspondant à cette allocation et donnez l'entrée de répertoire pour chaque fichier, en la complétant avec les attributs connus.
4. Les trois processus P1, P2 et P3 disposent d'un espace d'adressage paginé. La mémoire centrale est composée de dix cases numérotées de 1 à 10. Chaque case a

une capacité de 512 octets. Lors d'un défaut de pages, la page manquante est chargée dans la case libre de plus grand numéro.

La figure 16.4 représente l'allocation de la mémoire centrale à l'instant t pour ces trois processus ainsi que les tables des pages. Complétez-la.

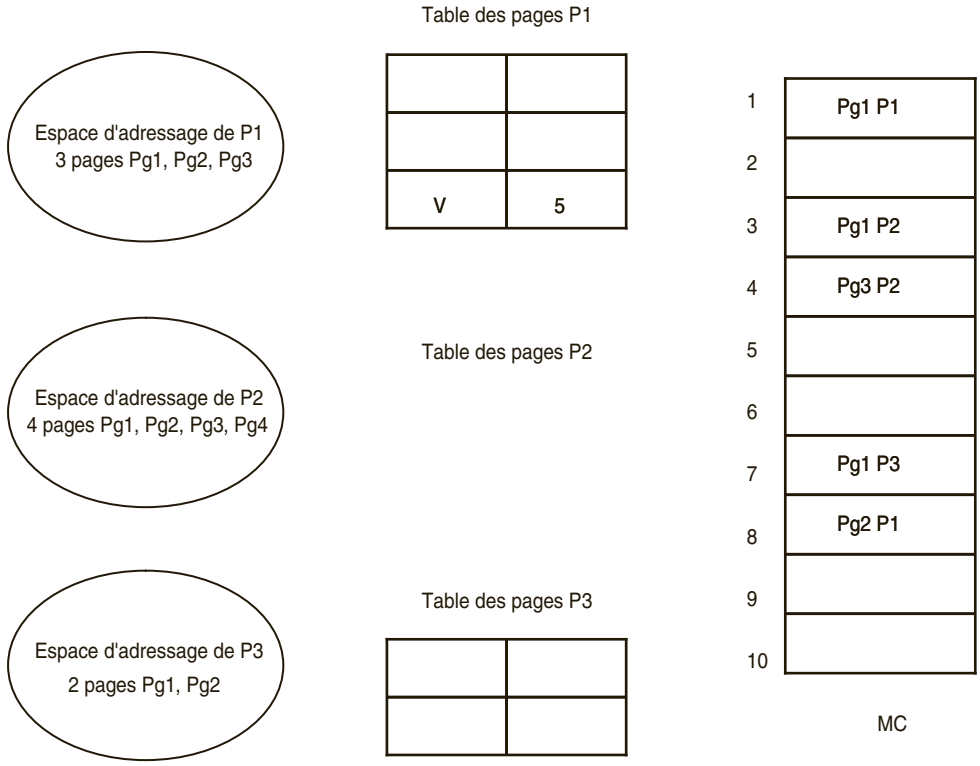


Figure 16.4 Allocation de la mémoire centrale à l'instant t.

– Le processus P1 accède à l'adresse paginée <page 1, deplacement 128>. Donnez l'adresse physique correspondante.
– Le processus P2 accède à l'adresse paginée <page 4, deplacement 64>. Donnez l'adresse physique correspondante.

SOLUTIONS

16.1 Algorithmes d'ordonnancement

- Politique « premier arrivé, premier servi » :
  - ordre de service : P1, P2, P3, P4, P5 ;
  - temps de réponse moyen : (2 + 8 + 18 + 22 + 34)/5 = 16,8.
- Politique « plus court d'abord » :
  - ordre de service : P1, P4, P2, P3, P5 ;
  - temps de réponse moyen : (2 + 6 + 12 + 22 + 34)/5 = 15,2.
- Politique par priorité :
  - ordre de service : P5, P1, P3, P2, P4 ;
  - temps de réponse moyen : (12 + 14 + 24 + 30 + 34)/5 = 22,8.
- Politique par quantum :
  - ordre de service : P1, P2, P3, P4, P5, P2, P3, P4, P5, P2, P3, P5, P3, P5, P3, P5, P5 ;
  - temps de réponse moyen : (2 + 20 + 30 + 16 + 34)/5 = 20,4.

16.2 Ordonnancement par priorité préemptif et non préemptif

La solution est donnée par la figure 16.5.

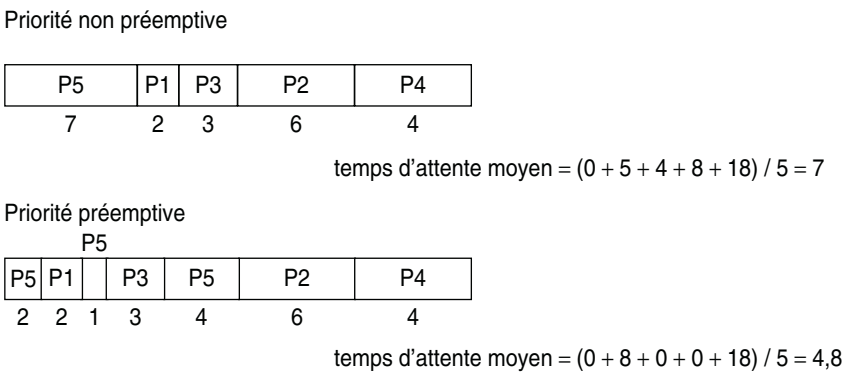


Figure 16.5 Priorité préemptive et non préemptive.

16.3 Chronogramme d'exécutions

La solution est donnée par la figure 16.6.

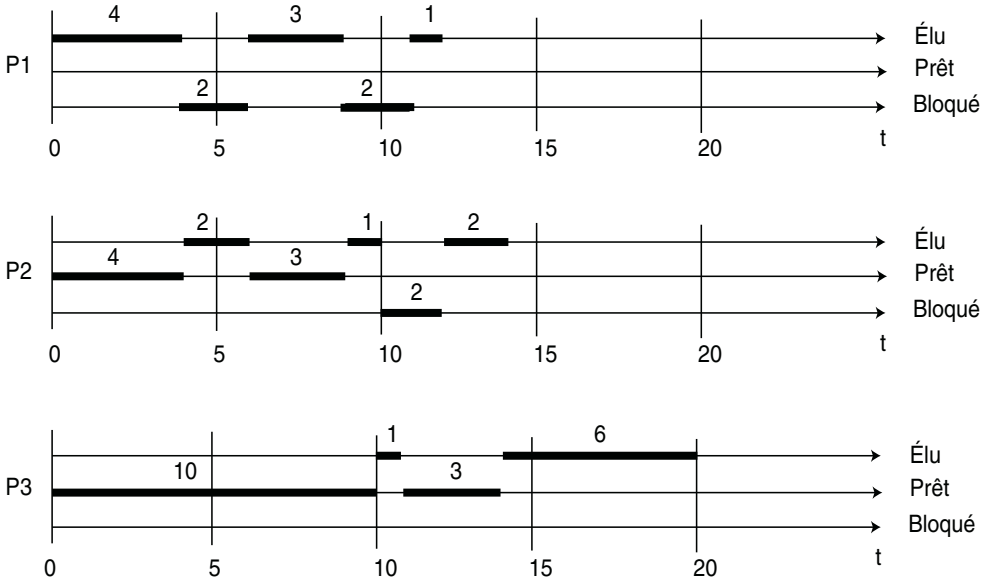


Figure 16.6 Chronogramme d'exécutions.

16.4 Ordonnancement sous Unix

À l'instant 0, le processus A est élu. Chaque seconde, l'interruption horloge survient et le compteur « utilisation du processeur » est incrémenté d'une unité.

Au bout de 60 interruptions (instant t = 1), la priorité des processus est recalculée. Du fait que les processus B et C ne se sont pas encore exécutés leur priorité est inchangée (compteur « utilisation du processeur » est nul pour ces processus). La priorité du processus A par contre baisse et devient égale à 75 (60 + 30/2). C'est donc le processus B qui est à présent élu.

À l'instant 2, de nouveau les priorités sont recalculées. Le compteur « utilisation du processeur » du processus C étant toujours nul, la priorité de ce processus n'est pas modifiée. Les priorités du processus A et B par contre évoluent et deviennent respectivement égales à 67 pour le processus A (60 + 15/2) et à 75 pour le processus B. On voit à présent que le processus A a une priorité qui remonte du fait qu'il n'a pas pu utiliser l'unité centrale. Le processus C est élu.

À l'instant 3, les priorités deviennent pour le processus A égal à 63 (60 + 7/2), pour le processus B égal à 67 et le processus C égale à 75. Le processus A est donc élu.

À l'instant 4, les priorités deviennent pour le processus A égal à 75 (60 + 31/2), pour le processus B égal à 63 et le processus C égale à 67. Le processus B est donc élu.

À l'instant 5, les priorités deviennent pour le processus A égal à 67 (60 + 15/2), pour le processus B égal à 75 et le processus C égale à 63. Le processus C est donc élu.

16.5 Ordonnancement sous Linux

La solution est donnée par la figure 16.7.

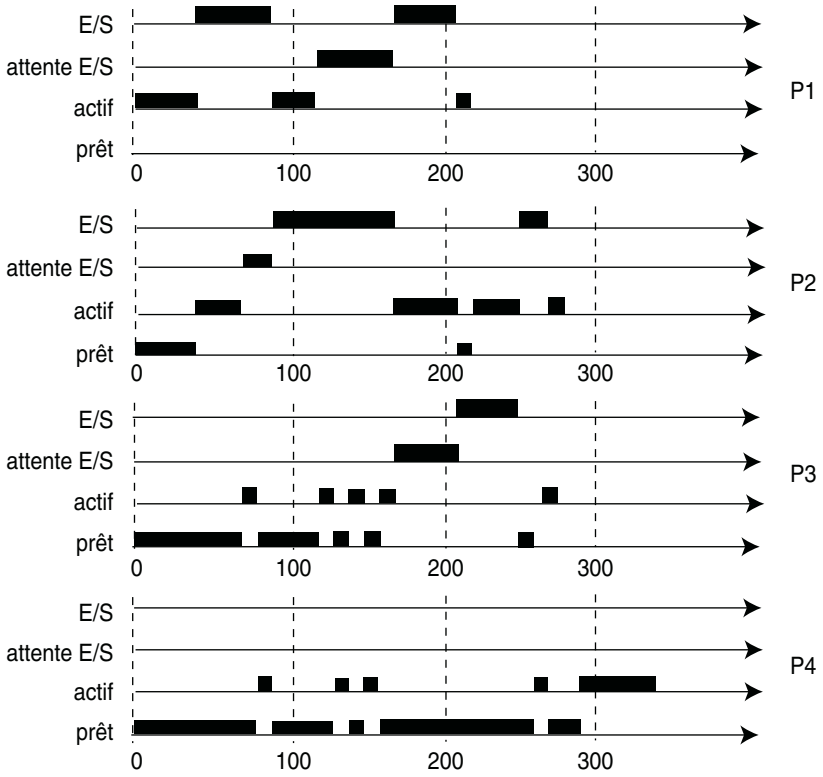


Figure 16.7 Ordonnancement Linux.

16.6 Producteur(s)-Consommateurs(s)

1. On identifie un schéma producteur-consommateur sur chacun des tampons. On utilise pour chacun de ces schémas un couple de sémaphores :
   – `mvide` initialisé à `m` et `mplein` initialisé à 0 (tampon requête) ;
   – `nvide` initialisé à `n` et `nplein` initialisé à 0 (tampon avis).

```
déclarations globales :
requête : tampon (0.m – 1) de messages;
avis : tampon (0.n – 1) de messages;
m, n : entier;
mvide, nvide, mplein, nplein : sémaphores;
init(mvide, m); init (nvide, n); init(mplein, 0); init(nplein, 0);
```

Acquisition	Exécution	Impression

mess : message;
i : index : = 0;
début
boucle
enregistrer_travail(mess);
P(mvide);
requete(i) = mess;
i = i + 1 mod m;
V(mplein);
fin boucle
fin
	mess, res : message;
	j, k : index : = 0;
	début
	boucle
	P(mplein);
	mess = requete(j);
	j = j + 1 mod m;
	V(mvide);
	exécuter_travail(mess, res);
	P(nvide);
	avis(k) = res;
	k = k + 1 mod n;
	V(nplein);
	fin boucle
	fin
		mess : message;
		l : index : = 0;
		début
		boucle
		P(nplein);
		mess = avis(l);
		l = l + 1 mod n;
		V(nvide);
		imprimer_resultat(mess);
		fin boucle
		fin

2. Il faut maintenant gérer les accès concurrents aux tampons avis et requête. En effet :
   - les différents processus Acquisition se partagent l'index i ;
   - les différents processus Exécution se partagent l'index j et k ;
   - les différents processus Impression se partagent l'index k.

   Les variables i, j, k, l sont maintenant globales et les accès à ces variables doivent se faire en exclusion mutuelle. On ajoute donc quatre sémaphores d'exclusion mutuelle initialisés à 1 (un sémaphore par index).

```
déclarations globales :
requête : tampon (0..m – 1) de messages;
avis : tampon (0..n – 1) de messages;
m, n : entier;
i, j, k, l : index sur les tampons;
mvide, nvide, mplein, nplein, muti, mutj, mutk, mutl : sémaphores;

début
init(mvide, m); init (nvide, n); init(mplein, 0); init(nplein, 0);
init(muti, 1); init(mutj, 1); init(mutk, 1); init(mutl, 1);
i = j = k = l = 0;
```

Acquisition	Exécution	Impression

mess : message;
début
boucle
enregistrer_travail(mess);
P(mvide);
P(muti);
requete(i) = mess;
i = i + 1 mod m;
V(muti);
V(mplein);
fin boucle
fin
	mess, res : message;
	début
	boucle
	P(mplein);
	P(mutj)
	mess = requete(j);
	j = j + 1 mod m;
	V(mutj);
	V(mvide);
	exécuter_travail(mess, res);
	P(nvide);
	P(mutk);
	avis(k) = res;
	k = k + 1 mod n;
	V(mutk);
	V(nplein);
	fin boucle
	fin
		mess : message;
		début
		boucle
		P(mutl);
		P(nplein);
		mess = avis(l);
		l = l + 1 mod n;
		V(nvide);
		V(mutl);
		imprimer_resultat(mess);
		fin boucle
		fin

16.7 Allocations de ressources et interblocage

1. Deux processus se partagent l'imprimante : le processus Collecteur et le processus Bilan. Imaginons que le processus Collecteur ait commencé une impression alors que le processus Bilan se réveille et démarre lui-même sa propre impression suite au traitement sur les mesures. Comme chacune des deux impressions comporte plusieurs lignes et que la ligne est l'unité de transfert vers l'imprimante il est très probable que les 2 impressions seront mélangées sur l'imprimante.

   Pour résoudre ce problème, il ne faut pas qu'un processus puisse démarrer une impression si une impression est déjà en cours. Autrement dit, l'imprimante doit être accédée en exclusion mutuelle.

2. Imaginons que le processus Collecteur s'exécute et acquiert l'imprimante par le biais de l'opération `P(imp)`. Le processus Bilan maintenant se réveille, préempte le processus Collecteur et acquiert la ressource disque par le biais de l'opération `P(disque)`. Maintenant les deux processus sont bel et bien en situation d'interblocage. Le processus Bilan poursuivant son exécution est bloqué sur l'opération `P(imp)` car l'imprimante est allouée au processus Collecteur. Le processus Collec-

teur lui-même se bloque sur l'opération `P(disque)` car le disque est alloué au processus Bilan.

Une solution est d'inverser les opérations `P(imp)` et `P(disque)` par exemple chez le processus Bilan de façon à ce que les opérations d'allocation de ressources soient effectuées dans le même ordre chez les 2 processus.

3. On considère deux sémaphores `depot` et `lire` respectivement initialisés à la valeur 1 et 0.

   Les pseudos codes des 2 processus concernant la prise des mesures sur le tampon de 3 cases sont alors de la forme :

Acquisition	Collecteur
`début`	`début`
`boucle`	`boucle`
`attendre (reveil);`	`P(lire);`
`P(depot);`	`lire_mesures(tampon, mesures);`
`ecrire_mesures(tampon, mesures);`	`V(depot)`
`V(lire);`	`P(imp);`
`fin boucle`	`P(disque);`
`fin`	`enregistrer_disque(mesures);`
	`imprimer(mesures);`
	`V(disque);`
	`V(imp);`
	`fin boucle`
	`fin`

16.8 Allocation de ressources et états des processus

1. Le sémaphore R est initialisé au nombre de ressources R initialement disponibles soit 2.
2. Suite des états pour les processus P1, P2 et P3 :

- t = 0 P1 élu P2 prêt P3 prêt :
  - P1 effectue P(R) et obtient un exemplaire de la ressource.
  - P1 se bloque sur l'appel système.
- t = 1 P1 bloqué P2 élu P3 prêt :
  - P2 effectue P(R) et obtient un exemplaire de la ressource.
  - P2 se bloque sur l'appel système.
- t = 2 P1 bloqué P2 bloqué P3 élu :
  - P3 effectue P(R) et se bloque sur cette opération car il n'existe plus d'exemplaires disponibles.

- t = 3 P1 bloqué P2 bloqué P3 bloqué :
  - P1 sort de l'appel système bloquant. P1 prêt puis élu rend la ressource R via l'opération V(R). P1 se termine. P3 est débloqué, acquiert un exemplaire de R et passe à l'état prêt.
- t = 4 P2 bloqué P3 prêt :
  - P3 est élu et se bloque sur l'appel système.
- t = 5 P2 bloqué P3 bloqué :
  - P2 sort de l'appel système bloquant. P2 prêt puis élu rend la ressource R via l'opération V(R). P2 se termine.
- t = 6 P3 bloqué :
  - P3 sort de l'appel système bloquant. P3 prêt puis élu rend la ressource R via l'opération V(R). P3 se termine.

16.9 Gestion de la mémoire par partitions variables

1. La solution est donnée par la figure 16.8.
2. La solution est donnée par la figure 16.9.

16.10 Remplacement de pages

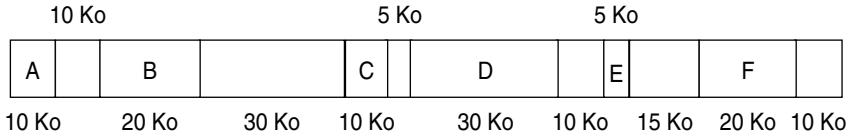
FIFO

Accès	3	5	6	8	3	9	6	12	3	6	10
Case 1	3	3	3	3	3	**9**	9	9	9	9	**10**
Case 2		5	5	5	5	5	5	**12**	12	12	12
Case 3			6	6	6	6	6	6	**3**	3	3
Case 4				8	8	8	8	8	8	**6**	6
Défaut	D	D	D	D		D		D	D	D	D

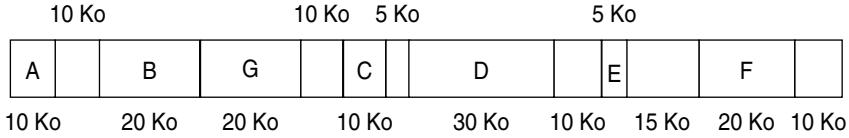
LRU

Accès	3	5	6	8	3	9	6	12	3	6	10
Case 1	3	3	3	3	3	3	3	3	3	3	3
Case 2		5	5	5	5	**9**	9	9	9	9	**10**
Case 3			6	6	6	6	6	6	6	**6**	6
Case 4				8	8	8	8	**12**	12	12	12
Défaut	D	D	D	D		D		D			D

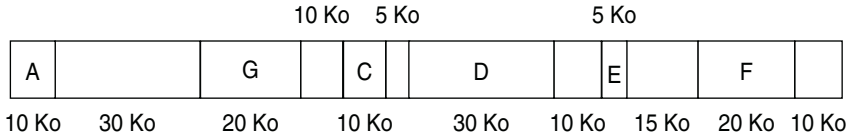
État Initial



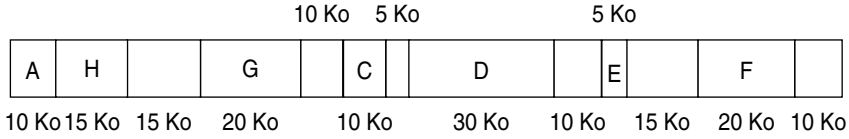
Arrivée du programme G, taille 20 Ko



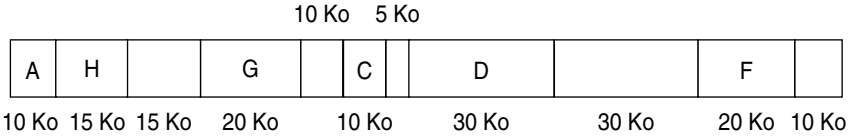
Départ du programme B



Arrivée du programme H, taille 15 Ko

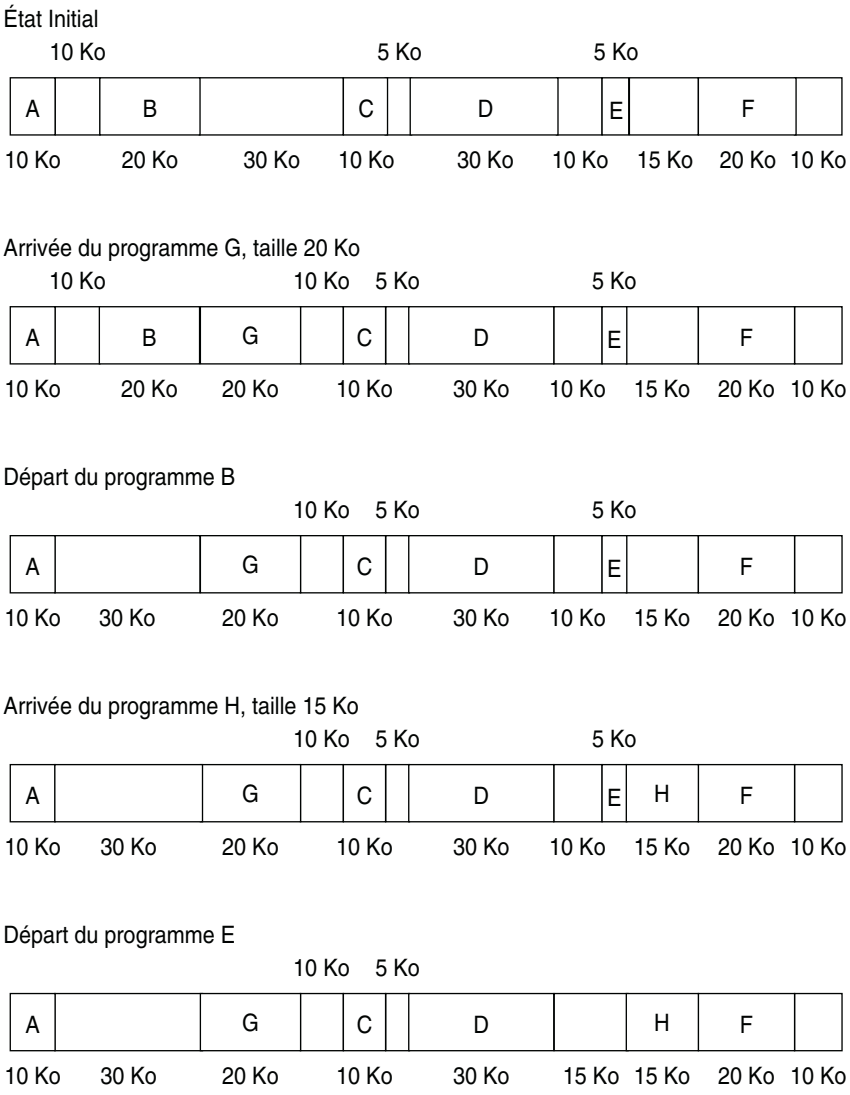


Départ du programme E



Arrivée du programme I, taille 40 Ko. L'état de la mémoire est fragmentéee. Il faut compacter :

Figure 16.8 Stratégie First Fit.



Arrivée du programme I, taille 40 Ko. L'état de la mémoire est fragmentéee. Il faut compacter :

Figure 16.9 Stratégie Best Fit.

16.11 Mémoire paginée et segmentée

1. La solution est donnée par la figure 16.10.

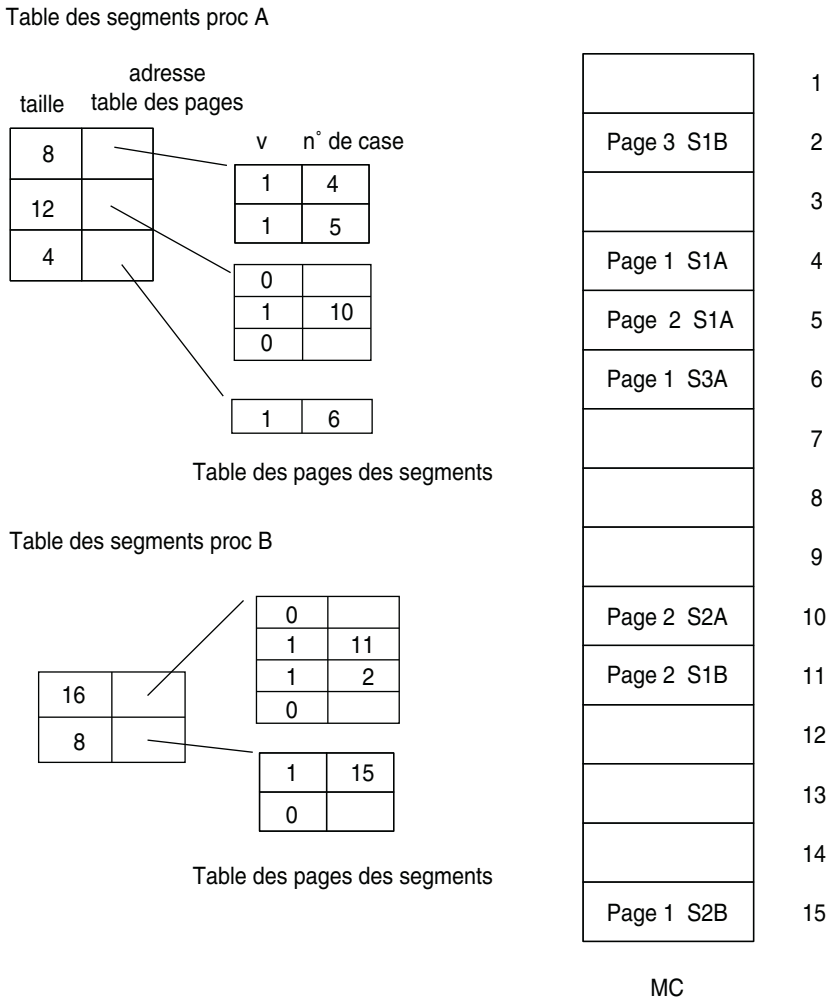


Figure 16.10 Tables des pages et tables des segments pour les processus A et B.

2. adresse logique <S1A, page 1, 12>. = adresse réelle <12 Ko, 12> = 12 Ko + 12 = 12 300.
3. adresse logique <S2B, page 2, 10>. La page 2 du segment 2 du processus B n'est pas en mémoire centrale. Il se produit un défaut de page. Nous pouvons supposer que la page manquante est chargée dans la première case libre, c'est-à-dire la case 1. Il s'ensuit alors que l'adresse physique correspondante est 10.
4. 4 098 pour le processus A : adresse virtuelle <S1, page 2, déplacement 2> = adresse réelle <case 5, déplacement 2> = octet 16 386.

12 292 pour le processus A : adresse virtuelle <S2, page 2, déplacement 4> = adresse réelle <case 10, déplacement 4> = octet 331 780.

8 212 pour le processus B : adresse virtuelle <S1, page 3, déplacement 20> = adresse réelle <case 2, déplacement 20> = octet 4 116.

16.12 Mémoire virtuelle et ordonnancement de processus

La solution est donnée par la figure 16.11.

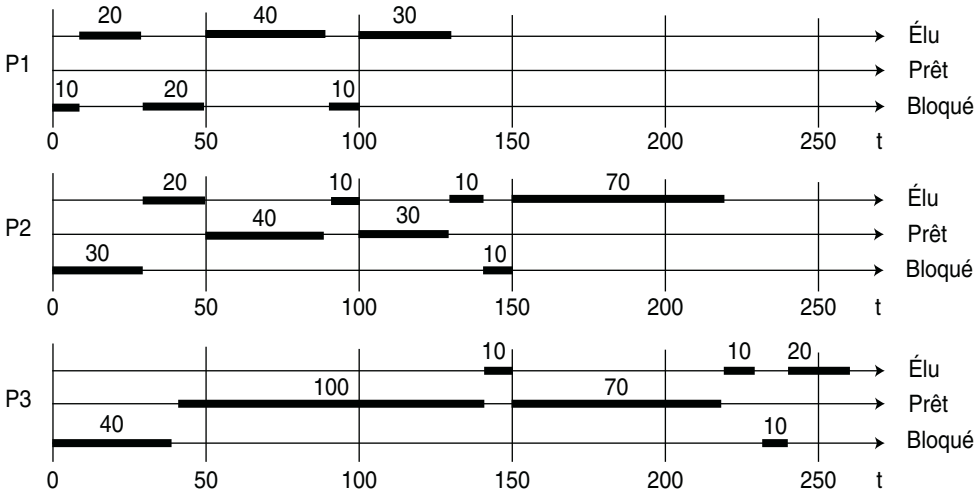


Figure 16.11 Chronogrammes d'exécutions des processus P1, P2 et P3.

16.13 Pagination à la demande

1. La figure 16.12 représente l'allocation de la mémoire centrale et les tables des pages des processus.
2. L'instruction de reprise est celle dont l'adresse correspond au CO = (page P5, déplacement 16). La valeur de PTBR est égale à 16.

   La conversion de l'adresse s'effectue selon le processus suivant :

   1. accès à la table via le PTBR ;
   2. accès à l'entrée 5 de la table des pages du processus A et test du bit V ;
   3. le bit V étant à 1, la page est présente en mémoire centrale ; l'adresse physique correspondante est (case 4, déplacement 16) soit 3 Ko + 16 = 3 098.

   Les opérations suivantes sont réalisées lors de la préemption du processus A par le processus B :

   1. sauvegarde dans le bloc de contrôle du processus A de sa valeur de CO et PTBR

2. restauration du contexte du processus B. Le registre CO est chargé avec l'adresse paginée (page P2, déplacement 512) et le registre PTBR prend la valeur 64.

Lorsque le processus B reprend son exécution :

1. accès à la table via le PTBR ;
2. accès à l'entrée 2 de la table des pages du processus B et test du bit V ;
3. le bit V étant à 0, la page est absente en mémoire centrale : il y a défaut de page ;

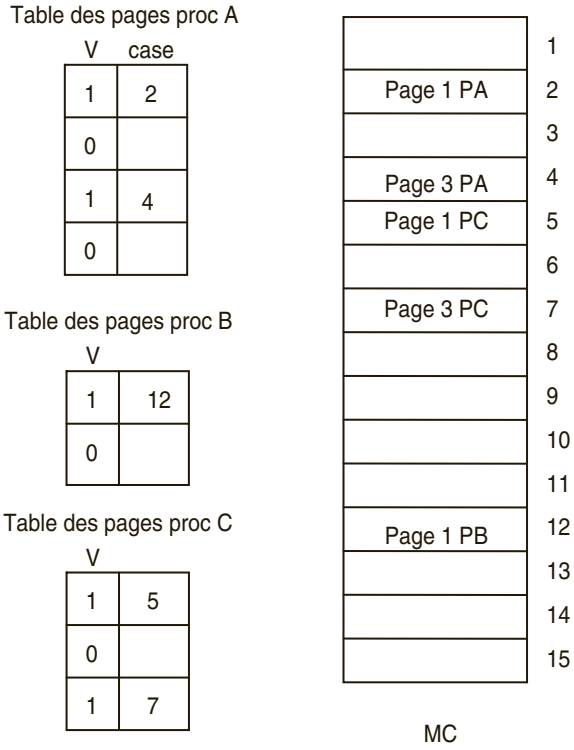


Figure 16.12 Allocation de la mémoire centrale et tables des pages des processus A, B et C.

3. Les droits 010 correspondent à un droit en lecture seule pour la page P1 du processus A. L'instruction STORE étant une opération d'écriture en mémoire centrale, une trappe est levée. L'exécution du processus A est déroutée en mode système et arrêtée.

16.14 Modes d'accès

1. Dans tous les cas, un seul enregistrement doit être lu.
2. Il faut une seule opération de lecture pour lire l'enregistrement 513 qui suit immédiatement l'enregistrement 512. La lecture de l'enregistrement 1 024 nécessite la lecture de 511 enregistrements tandis que la lecture de l'enregistrement 77 nécessite la lecture des 77 premiers enregistrements.

16.15 Organisation de fichiers

1. Dans tous les cas, une seule lecture est nécessaire car l'adresse de chaque bloc sur le disque peut être calculée à partir de l'adresse du premier bloc, de la taille des blocs et du numéro de bloc recherché.
2. La lecture des blocs impose de suivre le chaînage entre les blocs. Il faut donc une seule opération de lecture pour lire le bloc 513 qui suit immédiatement le bloc 512. La lecture du bloc 1 024 nécessite la lecture de 511 blocs tandis que la lecture de l'enregistrement 77 nécessite la lecture des 77 premiers blocs.

16.16 Noms de fichiers

1. Nom absolu de chacun des fichiers apparaissant dans l'arborescence :

```
/usr/bin/ls
/sbin/ping
/etudiants/dupont/exo1.c
/etudiants/dupont/rapport.txt
/etudiants/jeanne/fichier.o
/etudiants/jeanne/exo1.c
/professeurs/delacroix/pb.c
```

2. Non. L'utilisateur delacroix appartient au troisième niveau de protection pour le fichier `exo1.c`. Il ne peut que lire le fichier `exo1.c`.

16.17 Algorithmes de services des requêtes disque

FCFS :

– ordre de service : 62, 200, 150, 60, 12, 120, 250, 45, 10, 100;
– déplacement du bras : $12 + 138 + 50 + 90 + 48 + 108 + 130 + 205 + 35 + 90 = 906$.

SSTF :

– ordre de service : 45, 60, 62, 100, 120, 150, 200, 250, 12, 10;
– déplacement du bras : $5 + 15 + 2 + 38 + 20 + 30 + 50 + 50 + 238 + 2 = 450$.

LOOK montant :

– ordre de service : 60, 62, 100, 120, 150, 200, 250, 10, 12, 45;
– déplacement du bras : $15 + 2 + 38 + 20 + 30 + 50 + 50 + 240 + 2 + 38 = 485$.

16.18 Fichiers Unix

1. Nombre total d'accès disque nécessaire et temps d'attente en entrées-sorties :
   – lecture des 10 premiers blocs : 4×10 accès disque,
   – lecture des 256 blocs suivants (niveau d'indirection 1) : on a deux accès disque par lecture donc 8×256 accès disque,
   – lecture des 7 926 blocs restants (niveau d'indirection 2) : on a trois accès disque par lecture donc $12 \times 7\,926$ accès disque.

Soit un total de 97 200 accès disque et une durée moyenne de lecture égale à 4 860 s.

2. On a un accès disque par blocs de données lors de la lecture des 256 premiers octets du bloc. On a un total de 33 blocs d'adresse à lire, soit un nombre d'accès disque égal à 8 192 + 33 = 9 224 et un temps moyen de lecture égal à seulement 461 s !

16.19 Système de gestion de fichiers

La table d'allocation (FAT) correspondant à l'allocation du disque est la suivante :

Numéro entrée	
1	4
2	libre
3	libre
4	9
5	libre
6	7
7	16
8	libre
9	10
10	15

Numéro entrée	
11	17
12	libre
13	20
14	Fin fichier
15	Fin fichier
16	14
17	13
18	libre
19	libre
20	Fin fichier

16.20 Système de gestion Unix

On considère un fichier de type UNIX. Le fichier a une taille de 16 Moctets. Les blocs disque sont de 1 024 octets. Un numéro de bloc occupe deux octets.

1. Le nombre de blocs est égal à $2^4 \times 2^{20} / 2^{10} = 2^{14}$ blocs soit 16 384 blocs.
2. Un bloc d'adresse comporte 512 entrées. Pour adresser les 16 384 blocs de données, il faut donc :
   - Un bloc d'adresse INDIRECT_1 adresse 512 blocs de données.
   - Il reste 16 384 – 512 – 10 = 15 862 blocs. Il faut 31 + 1 blocs d'adresses au niveau d'indirection 2 pour adresser ces blocs de données.
   - Au total donc, 33 blocs d'adresses sont nécessaires.

16.21 Synthèse

1. La figure 16.13 donne le chronogramme d'exécution des trois processus.
2. La figure 16.14 donne le chronogramme d'exécution des trois processus.

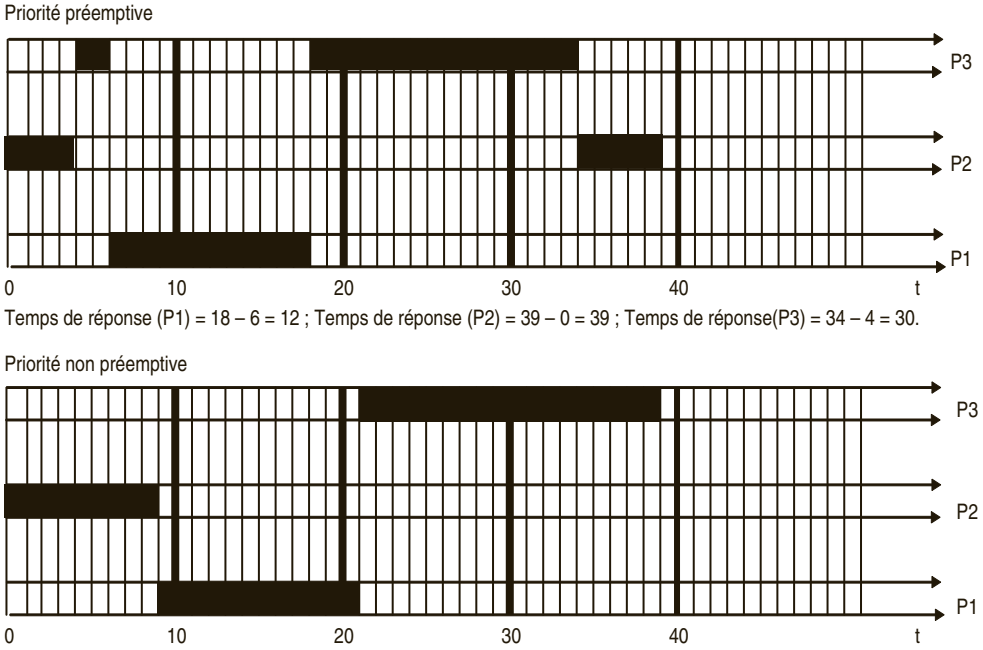


Figure 16.13 Chronogramme d'exécution.

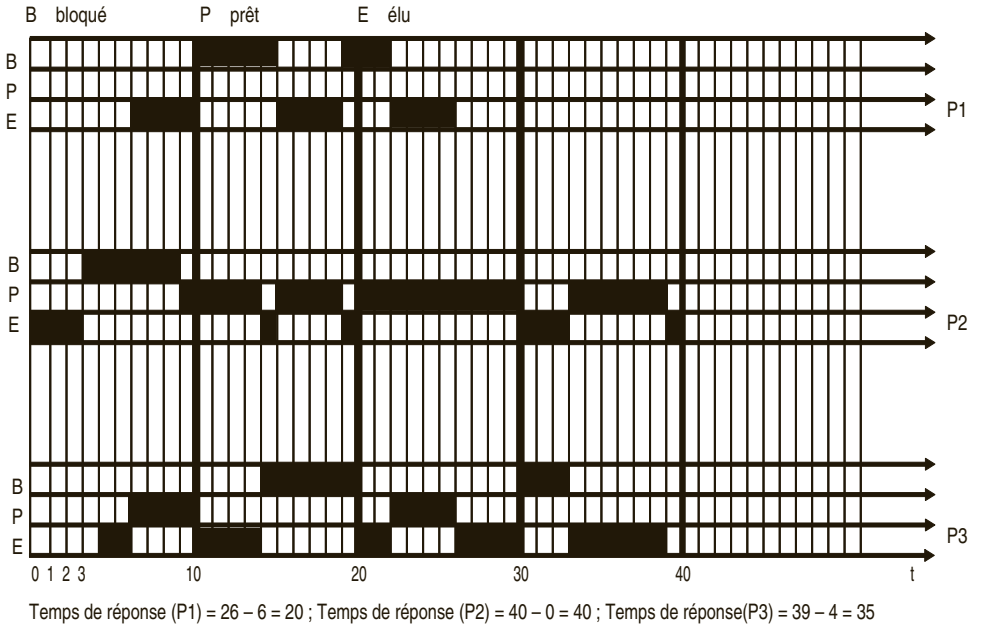


Figure 16.14 Chronogramme d'exécution.

3. La figure 16.15 donne la table d'allocation FAT correspondant à l'allocation décrite.

1	5
2	7
3	2
4	8
5	9
6	10
7	11
8	17
9	15
10	null
11	30
12	13
13	16
14	libre
15	25
16	21
17	12
18	libre
19	libre
20	libre
21	null
22	24
23	libre
24	6
25	29
26	libre
27	libre
28	libre
29	null
30	null

FAT

nom fichier	taille (octets)	adresse premier bloc
fich_1	5 632	1
fich_2	4 098	3
fich_3	6 272	4
fich_4	3 104	22

Entrées de répertoire

Figure 16.15 Table d'allocation FAT.

4. La figure 16.16 donne le schéma d'allocation mémoire complété.
   - Le processus P1 accède à l'adresse paginée <page 1, deplacement 128>. L'adresse physique correspondante est <case 1, déplacement 128>, soit 128.
   - Le processus P2 accède à l'adresse paginée <page 4, deplacement 64>. La page 4 du processus P2 est hors mémoire centrale. Il y a défaut de page. La page

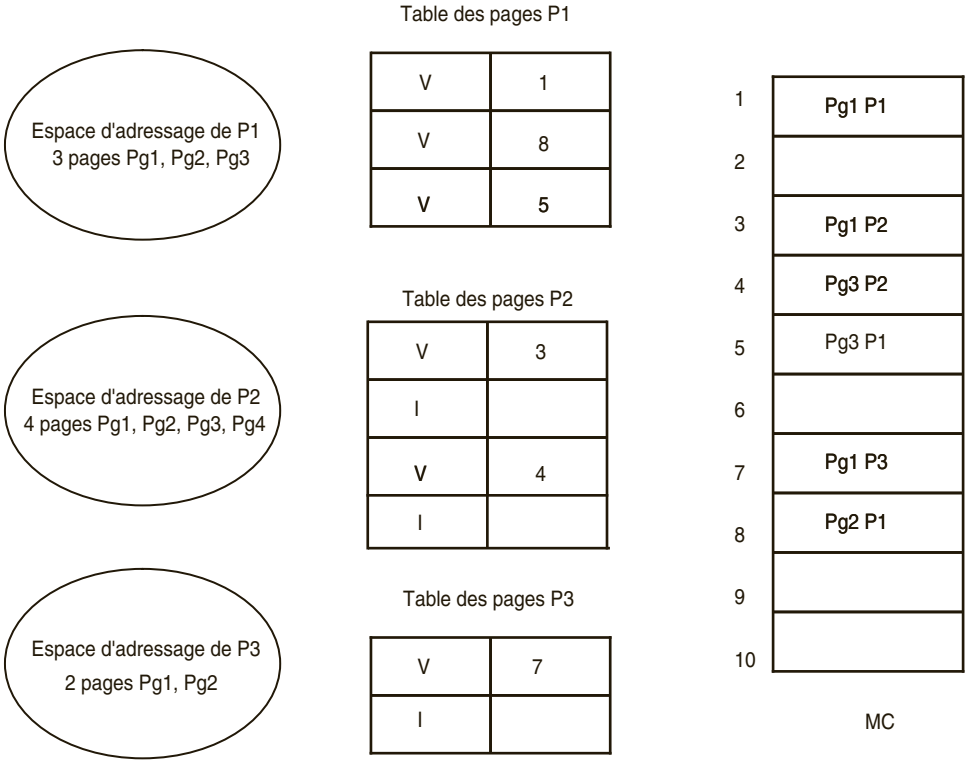


Figure 16.16 Allocation mémoire et tables des pages des processus.

manquante est chargée dans la case libre de plus grand numéro soit la case 10. L'adresse physique correspondante est alors <case 10, déplacement 64> soit $(9 \times 512) + 64$.

Index

A

ACL 406
Activité de programmation 25
Additionneur n bits 94
Adressage 394
  NSAP 394
Adresse 5
  paginée 324
  segmentée 332
ADSL 392
Algorithme 1, 25
  C-SCAN 378
  d'ordonnancement 270
  de la seconde chance 343
  FCFS 377
  LOOK 379
  SCAN 378
  SSTF 377
Allocation 317
  contiguë 355
  indexée 362
  par blocs chaînés 359
  par zones 357
Analyse
  lexicale 38
  sémantique 42
  syntaxique 40
Appel système 271, 280, 296
Apple 218
ART 392
Assembleur 29, 31, 87

B

Bande
  ISM 402
  U-NII 402
Bascule 99
Best Fit 355
Bibliothèque 47, 57, 281
Bit 65, 66
  de carry 73
Bloc
  de contrôle du processus 290
  du disque 355
Bluetooth 400, 407
Bootstrap 19
Bus 3, 134, 210
  AGP 216
  asynchrones 136
  d'adresses 8
  de commandes 9
  de communication 8, 207
  de données 9
  EISA 211
  ISA 211
  MCA 211
  parallèles 210
  PC-AT 211
  PCI 212
  SCSI 236
  série 210
    USB 232
  synchrones 135

C

Câble coaxial 385
Cache 193
  associatif 191, 328
  direct 189
  mixte 192
Carte
  CardBus 400

CompactFlash 400
Mini-PCI 400
PC-Card 400
PCI 400
PCMCIA 400
CDMA 409
Chaîne de production de programme 30, 35
Chargement 59
d'un système d'exploitation 286
dynamique 60, 315
statique 60
Chargeur 18, 31
Circuit
logique 90
séquentiel 99
CISC 198, 200, 202
Coalition 300
Code relogeable 44
Commande 271, 280, 282
Communication 205
Commutation de contexte 277, 296
Compactage 322, 355
Compilateur 29, 31
Compilation 36
séparée 46
Concentrateur 388, 389
Contexte du processus 289
Contrôleur 229
Convention
de la valeur signée 70
du codage DCB 72
du complément à 2 71
Cylindre 354

D

Datagrammes 396
Décodage 141
Défaut de page 340
Dépassement de capacité 73
Descripteur de fichier 373
Diffusion
broadcast 384
multicast 384
Disque 181
DMA 241
DRAM 172
Driver 268, 270
Droit d'accès 379
DSL 392

E

Échange 219
Échéance 275
Écroulement 346
Éditeur
de liens 31
de texte 30
Édition des liens 46
dynamique 57, 61
statique 57
EDO 172
Élection 290, 295
Entrées-sorties 236, 374
asynchrones 241
Espace
d'adressage 289, 304, 317, 324, 332
d'adresses logiques 316
d'adresses physiques 316
État
bloqué 289
d'un processus 289
élu 289
prêt 290
ETCD 224
Ethernet 227, 396
ETSI 408
ETTD 224
Exception 277
Exclusion mutuelle 305, 306
Exécution 129, 154
Extinction de priorité 302

F

Famine 300
FAT (*File Allocation Table*) 360
FDMA 409
FETCH 139
Fibre optique 385, 390
Fichier 270
logique 348
physique 354
First Fit 355
Formatage 354
Fragmentation 355
externe 320, 324, 335, 359
interne 318, 322, 324
Full duplex 399

G

Génération d'un système d'exploitation 285

H

Half duplex 399
HiperLAN 408

I

Index 362
  multiniveaux 363
Instruction 2, 163
  machine 27, 79, 137
Intel 217
Interblocage 309
Internet 409
  adressage 410
  protocole 412
Interpréteur 29
  de commandes 271, 280, 283
Inter-réseaux 384
Interruption 19, 154, 239, 279, 281, 306, 376
  logicielle 279
  matérielle 279

L

Label 369
LAN 384
Langage
  d'assemblage 28, 84
  de haut niveau 29, 36
  évolué 29
  machine 15, 27
Liaison programmée 237
Libération mémoire 317
Lien
  à satisfaire 47, 50, 52
  utilisable 47, 50, 51
Liste d'accès 379

M

Make 62
Makefile 62
MAN 384
Mémoire
  cache 169, 184
  centrale 3, 4, 13, 131, 207, 315
  de travail 171
  morte 180
  virtuelle 195, 270, 339
  vive 171
Mémorisation 168
Micro-instructions 145, 148
Microprocesseur 3, 10, 13, 131, 132
MIMD 167
MIPS 163
MISD 167
MMU 196, 316, 323, 327, 334
Mode
  ad-hoc ou IBSS 402
  d'accès 349
    direct 350
    indexé 350
    séquentiel 349
  d'adressage 80, 81
  d'exécution 277
  esclave 277
  infrastructure 401
  maître 277
  superviseur 277
  utilisateur 277
Mode de transmission
  à commutation de paquets 394
  différé 394
  diffusion 384
  point à point 384
  unicast 385
Mot
  de passe 379
  mémoire 5
Multi-ordinateurs 166
Multiplexeur 390
Multiprocesseur 166

N

Norme IEEE 754 75
Notation de Backus-Naur 37
Numération
  binaire 66
  hexadécimale 66
  octale 66

O

Opérateur
  ET 93
  NON 92
  OU 92
  OU exclusif 93
Ordinateur 1, 127
Ordonnancement 295
  des requêtes disque 376
  non préemptif 296
  préemptif 296

sur l'unité centrale 295
Ordonnanceur 297
Overflow 73

P

Pagination 324
à la demande 339
des segments 335
multiniveaux 330
Paire
métallique 391
torsadée 385
PAN 383
Partition 369
FAT 369
Unix 371
variable 317
Pathname 368
Performance 162
Périphérique 205, 218
interface 206
Piconet 407
Pilote 208
Pipeline 163
Piste 354
Plateau 354
Politique
d'ordonnancement 297
du « Plus Court d'Abord » 299
du « Premier Arrivé, Premier Servi » 298
par priorité constante 300
par tourniquet 301
Porte 92
Processeur 2, 10, 165, 199, 207
matriciel 165
vectoriel 165
Processus 289
Unix 292
Programme 25
d'amorçage 286
machine 154
objet 36
réentrant 289
source 36
Protection 267, 270, 323, 328, 335, 379
contre les accès inappropriés 379
contre les dégâts physiques 380
Protocole CSMA/CA 403
Protocole d'accès
DCF 402, 403
PCF 402
PSW 11

Q

Qualité de service 396
Quantum 301

R

RAM 19, 171
Redondance
interne 380
par sauvegarde périodique 380
Registre 11, 101, 145, 171, 180
Registre d'état (PSW) 73
Remplacement
de pages 342
FIFO 342
LFU 344
LRU 343
MFU 344
Répartiteur 297
Répertoire 366
à deux niveaux 367
à un niveau 367
de travail 367
racine 367
Représentation
d'un nombre signé 69
des caractères 77
des nombres flottants 74
Réseau
à commutation 390
adressage 394
cartes ~ 400
de commutation 385
définition 383
domestique 384
Ethernet 396
filaire 385
grande distance 408
interconnexion 409
inter-réseaux 384
local 384, 408
longues distances 384
maillé 388
métropolitain 384
personnel 383, 407
qualité de service 396
routage 395
sans fil 399

sécurité 406
topologie 386
Ressource critique 305, 306
RISC 198, 201, 202
ROM 19, 171
Routage 395
Routeur 389
Routine système 271, 276
RS232 224

S

Sauvegarde
complète 381
incrémentale 381
Schéma
de l'allocation de ressources 309
lecteurs-rédacteurs 310
producteur-consommateur 312
Secteur 354
Section critique 306, 308
Segmentation 332
Sémaphore 307
Séquencement 150
Séquenceur
câblé 150
microprogrammé 153
Session de travail 284
SIMD 167
SISD 167
SRAM 172
SSID 406
Stratégie de la
meilleure zone libre (*Best Fit*) 318
plus mauvaise zone libre (*Worst Fit*) 318
première zone libre (*First Fit*) 318
Superscalaire 165
SVC 242
Swap 347
Synchronisation 270, 305, 307
Système
à traitement par lots 272
batch 272
d'exploitation 265, 268
interactif 274
de gestion de fichiers 270, 348
embarqué 276
en temps partagé 274
ouvert 271
réactif 276
temps réel 275, 300

T

Table
de vérité 91
des cadres de pages 324
des pages 325
des segments 333
des vecteurs d'interruptions 279
TDMA 409
Technique de transmission
DSSS 405
FHSS 405
Temps de réponse 274
Topologie
BSS 401
en anneau 386
en bus 386
en étoile 386
ESS 402
Traitant d'interruption 279
Transistor 101, 102
bipolaire 103
unipolaire 103
Trappe 277, 280

U

UAL 11, 146
UART 221, 222
Unité
d'échange 16
de contrôle 3
USB 214

V

V24 224
Vecteur d'interruptions 158

W

WAN 384
WEP 407
Wi-Fi 408
WLAN 400
WMAN 400
Worst Fit 355
WPAN 400
WWAN 400

Z

Zone de swap 341, 347

Photocomposition : **SCM**, Toulouse

SCIENCES SUP

3e édition

Alain Cazes
Joëlle Delacroix

ARCHITECTURE DES MACHINES ET DES SYSTÈMES INFORMATIQUES

Cet ouvrage s'adresse aux étudiants de premier cycle informatique et il constitue également un bon ouvrage de référence pour les étudiants d'IUT et les élèves ingénieurs.

Il présente le fonctionnement d'un ordinateur au niveau matériel et au niveau système d'exploitation. Ainsi, l'ordinateur est étudié depuis son niveau le plus haut – celui du langage de programmation et celui de l'interface du système – jusqu'à son niveau le plus bas – celui de l'exécution binaire et des composants électroniques. Pour chaque fonction ou composant de la machine, les **notions de base** sont présentées, puis **les concepts sont approfondis**. Des **exemples** sont donnés s'appuyant sur des architectures à base de processeurs connus, ainsi que sur des systèmes d'exploitation tels que Unix, Linux, Windows ou encore MVS.

Un chapitre est consacré aux réseaux : leurs particularités, leurs usages et leur interconnexion.

L'ouvrage se compose de trois grandes parties :

- la production de programmes ;
- la structure de l'ordinateur ;
- les systèmes d'exploitation.

Chacune de ces parties s'achève par un ensemble d'**exercices corrigés, encore plus nombreux dans cette nouvelle édition**.

ALAIN CAZES
est maître de conférences en informatique au Conservatoire National des Arts et Métiers. Il est docteur en informatique.

JOËLLE DELACROIX
est maître de conférences en informatique au Conservatoire National des Arts et Métiers. Elle est docteur en informatique.

MATHÉMATIQUES
PHYSIQUE
CHIMIE
SCIENCES DE L'INGÉNIEUR
INFORMATIQUE
SCIENCES DE LA VIE
SCIENCES DE LA TERRE

LICENCE 1 2 3 | MASTER 4 5 6 | DOCTORAT 7 8

ISBN 978-2-10-053945-1

www.dunod.com

DUNOD