# Systèmes d'exploitation et réseaux

Bart Lamiroy


Version 1.5 du 30 septembre 2004

Ce document a été réalisé sous XEmacs avec LaTeX.
Version du 30 septembre 2004, © Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

# Table des matières

**Introduction** **1**

**1 Systèmes d'exploitation, définition et description** **4**
1.1 Architecture d'un micro-ordinateur . . . . . 5
1.2 Rôle d'un système d'exploitation . . . . . 11
1.2.1 Le noyau . . . . . 12
1.2.2 Les fonctions système . . . . . 13
1.3 Les processus . . . . . 14
1.3.1 La multiprogrammation . . . . . 15
1.3.2 Rôle de l'OS . . . . . 17
1.3.3 Mise en œuvre . . . . . 18
1.3.4 Les algorithmes d'ordonnancement . . . . . 20
1.3.5 Les processus légers – *threads* . . . . . 23
1.3.6 Communication entre processus . . . . . 27
1.3.7 Résumé . . . . . 44
1.4 La mémoire . . . . . 44
1.4.1 Le modèle de base . . . . . 44
1.4.2 Le chargement des processus . . . . . 45
1.4.3 La multiprogrammation avec partitions fixes . . . . . 46
1.4.4 La multiprogrammation avec partitions variables . . . . 48
1.4.5 La mémoire virtuelle . . . . . 53
1.4.6 La segmentation . . . . . 58
1.5 Les systèmes de fichiers . . . . . 62
1.5.1 Opérations de haut niveau . . . . . 62
1.5.2 Définition d'un système de fichiers et mise en œuvre . . 63
1.5.3 Structure bas niveau . . . . . 64

**2 Réseaux et systèmes d'exploitation interconnectés** **66**
2.1 Qu'est-ce qu'un réseau ? . . . . . 67
2.1.1 Communication et protocoles . . . . . 67
2.2 Réseaux d'échange de données . . . . . 69

2.2.1 Classifications . . . . . 69
2.2.2 Transmission de signal et multiplexage . . . . . 73
2.2.3 Le modèle en couches ISO–OSI . . . . . 77
2.2.4 Les protocoles « Liaison » . . . . . 83
2.3 Internet et TCP/IP . . . . . 89
2.3.1 Historique et concepts de base . . . . . 89
2.3.2 Le modèle de couches de TCP/IP . . . . . 91
2.3.3 La couche réseau : IP – Internet Protocol . . . . . 92
2.3.4 Les couches Transport et Session . . . . . 109
2.4 Applications réparties et communicantes . . . . . 119
2.4.1 Les sockets et IPC . . . . . 119

**A L'utilisation de la pile et des registres d'état** **131**

**B Les sémaphores en POSIX** **134**

**Références bibliographiques** **137**

# Table des figures


1.1 Les couches constituant un système ordinateur . . . . . 4
1.2 Schéma de l'architecture d'un micro-ordinateur . . . . . 5
1.3 Représentation d'un programme . . . . . 8
1.4 Multiprogrammation et mémoire . . . . . 15
1.5 Graphe d'état d'un processus (non-préemptif) . . . . . 16
1.6 Graphe d'état d'un processus (préemptif) . . . . . 17
1.7 Système d'exploitation à micro-noyau . . . . . 18
1.8 Changement de contexte par interruption . . . . . 21
1.9 Ordonnancement de processus avec priorité . . . . . 23
1.10 Partage de ressources par alternance . . . . . 31
1.11 L'exclusion mutuelle par attente active par PETERSON (1981). 32
1.12 Les trois organisations de base pour un système monotâche. . 45
1.13 Principe d'allocation de partitions de taille variable. . . . . 49
1.14 Gestion de la mémoire par liste chaînée. . . . . 50
1.15 Réallocation de processus avec augmentation de la zone tas–pile. 52
1.16 Correspondance entre mémoire virtuelle et mémoire réelle. . . 54
1.17 Mémoire virtuelle à deux niveaux. . . . . 56
1.18 Structuration d'une image mémoire d'un processus. Différence entre mémoire paginée (a) et mémoire segmentée (b). . . . . 60
1.19 Structure bas niveau d'un disque dur. . . . . 64

2.1 Réseau point-à-point, réseau en étoile, réseau maillé . . . . . 70
2.2 Réseau en anneau, réseau en bus . . . . . 70
2.3 Réseau hétérogène. . . . . 71
2.4 Signal binaire simple. . . . . 75
2.5 Transformée de Fourier du signal binaire simple avec superposition de bande passante (partie réelle). . . . . 75
2.6 Signal binaire simple effectivement transmis. . . . . 75
2.7 Principe de multiplexage en fréquence par transformée de Fourier. . . . . 77
2.8 Principe de multiplexage en fréquence. . . . . 78
2.9 Principe de multiplexage en temps. . . . . 79

2.10 Principe de l'encapsulation des données dans le modèle à couches ISO–OSI . . . . . 80
2.11 La communication pair-à-pair de la couche transport ISO–OSI . 81
2.12 Interférence dans un réseau à diffusion . . . . . 86
2.13 Automate de transition de l'algorithme CSMA/CD. . . . . 87
2.14 Format d'une trame Ethernet . . . . . 88
2.15 Réseau interconnecté. . . . . 93
2.16 Format d'un datagramme IP. . . . . 99
2.17 Principe du calcul d'une somme en complément à 1. . . . . 101
2.18 Format d'un datagramme ARP pour la liaison Ethernet–IP. . . 105
2.19 Format d'un datagramme ICMP. . . . . 106
2.20 Format de l'entête de base IPv6 . . . . . 108
2.21 Format d'un datagramme UDP. . . . . 111
2.22 Format d'un segment TCP. . . . . 112
2.23 Ouverture de session TCP. . . . . 115
2.24 Utilité d'un numéro de séquence aléatoire en TCP. . . . . 115
2.25 Exemple d'échange avec une fenêtre glissante de 4. . . . . 118
2.26 Fermeture de session TCP. . . . . 118

# Liste des exemples

1.1 Threads en JAVA . . . . . 24
1.2 Threads Posix en C . . . . . 25
1.3 Utilisation de moniteurs en JAVA . . . . . 38
1.4 Les signaux Posix . . . . . 42
2.1 Algorithme de calcul de *checksum* . . . . . 101
2.2 La commande `ping` . . . . . 106
2.3 Le serveur séquentiel non connecté – en C . . . . . 122
2.4 Le serveur parallèle connecté – en JAVA . . . . . 127

# Introduction

Ce document est le support de cours du module SI141 (Systèmes d'exploitation et réseaux) de l'École des Mines de Nancy.

Le module « Systèmes d'exploitation et réseaux » a pour but de fournir une introduction globale au fonctionnement des systèmes d'exploitation et des réseaux actuels. Compte tenu de la particularité de la formation et du large spectre du cours, de nombreux détails techniques seront éludés. L'accent est mis sur la compréhension des méchanismes globaux, et certaines considérations algorithmiques, bien que, pour comprendre la raison d'être de certaines fonctionnalités d'un système d'exploitation ou d'un protocole réseau, le contexte de l'architecture et le comportement du matériel ne peut pas être complètement perdu de vue. L'objectif est de fournir un tour d'horizon suffisamment complet et détaillé pour que le lecteur puisse ensuite facilement aborder une lecture plus technique et la placer dans son contexte.

Se basant sur une définition générique des fonctionnalités qu'un OS [1] doit pouvoir fournir, on étudie les deux systèmes les plus répandus sur des micro-ordinateurs et stations de travail : Unix et Windows–NT/XP.

Dans une deuxième partie, on introduit l'interconnexion de systèmes d'exploitation à travers de réseaux (principalement de type `TCP/IP`) que l'on étudie à différents niveaux d'abstraction : les protocoles mis en œuvre, les services systèmes, les applications autour du WWW.

**Note :** Dans la mesure du possible, tous les exemples d'algorithmes dans ce livre sont écrits en langage JAVA et en `C`. Le choix de présenter les deux vient du fait qu'en général, les solutions en JAVA sont élégantes, simples et permettent d'aborder l'essentiel. En revanche, le fait que JAVA soit exécuté par une *Machine Virtuelle*, qui est en quelque sorte, son propre système d'exploitation, le distancie parfois un peu trop des questions techniques sous-jacentes. D'où le choix d'également proposer un exemple équivalent en `C`, qui lui, est plus proche de l'environnement système.

[1] On utilisera de préférence l'abréviation anglophone OS (*Operating System*) au lieu du peu usité SE pour désigner le système d'exploitation

Pour ce qui concerne les sources `C`, le lecteur trouvera facilement des similarités avec JAVA. Pour l'unique exemple d'assembleur, il s'agit d'une approximation simplifiée de l'assembleur des processeurs `x86` d'Intel, mais qui est suffisamment commenté pour que le lecteur puisse se contenter de lire les commentaires uniquement.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

## Remerciements

Je tiens ici à remercier toutes les personnes qui ont contribué à faire de ce document ce qu'il est maintenant, et notamment tous les élèves de l'École des Mines de Nancy qui m'ont indiqué les passages peu compréhensibles, les fautes de frappe, de syntaxe ou de sémantique et les exemples erronnées.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

# Chapitre 1

# Systèmes d'exploitation, définition et description

Dans ce chapitre nous allons décrire les principales fonctionnalités requises pour un système d'exploitation (ou OS) moderne.

Le système d'exploitation est un logiciel qui a pour objectif de libérer l'utilisateur d'un ordinateur de la complexité (et la redondance) de la programmation du matériel.

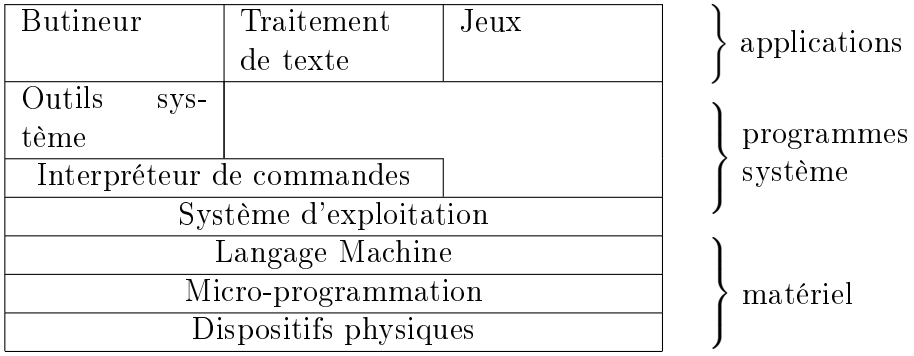


FIG. 1.1 – Représentation d'un ordinateur et ses logiciels par un ensemble de couches. Le système d'exploitation (ou plus généralement l'ensemble des programmes système) sert d'interface entre les applications et la couche matérielle. (inspiré de [1])

Dans la FIG. 1.1 on fait la distinction entre l'OS des programmes système qui fournissent des services aux applications de plus haut niveau. Cette distinction entre les deux types de programmes (l'OS d'une part et les programmes système de l'autre) est subtile, et deviendra clair au fil de ce chapitre. Pour une lecture plus aisée, on peut considérer que cette partie sera

consacrée à l'OS proprement dit, tandis que les parties suivantes traitant de Linux et Windows–NT/XP concerneront détailleront certains aspects OS, mais également les programmes système.

## 1.1 Architecture d'un micro-ordinateur

Afin de mieux comprendre la nécessité d'un système d'exploitation, nous rappelons brièvement la structure et le fonctionnement dans la couche matérielle. Cette architecture est majoritairement due aux travaux du mathématicien américain John VON NEUMANN (1903-1957), qui prévoit, en outre, que la mémoire ne fait aucune distinction entre données et instructions. Reste néanmoins que la présentation faite ici a principalement pour but de faire comprendre les concepts principaux du fonctionnement interne d'un ordinateur. Elle est simplifiée, et sur plusieurs points en déphasage avec les technologies modernes (bus multiples, DMA, AGP, *etc.*) ne doit pas être prise comme une représentation exacte et fidèle de la réalité, bien que le principe de fonctionnement y corresponde en grandes lignes.

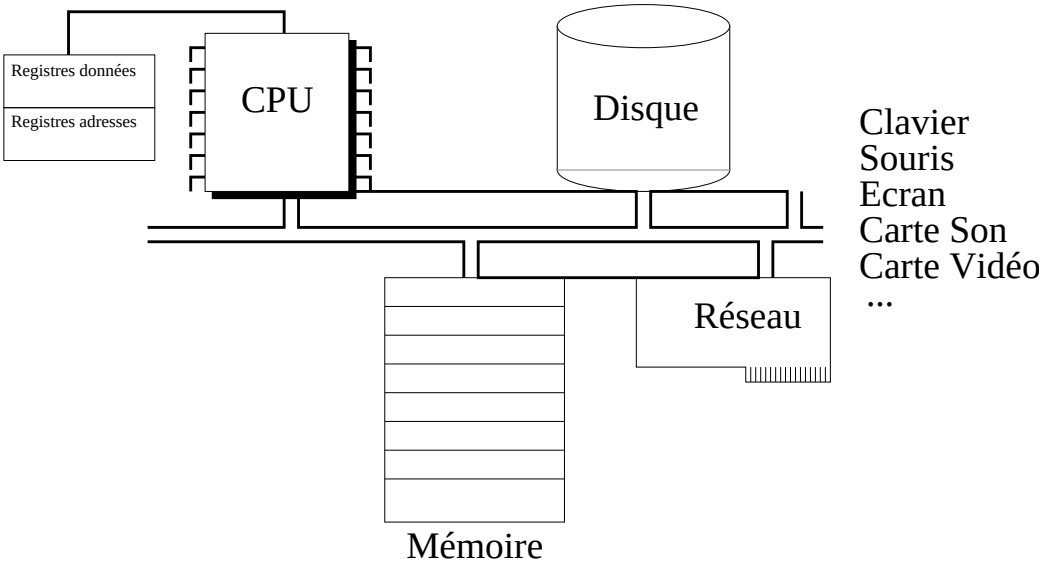


FIG. 1.2 – Représentation d'un micro-ordinateur. Une unité centrale (UC ou CPU– *central processing unit*) disposant d'un ensemble de registres, et communiquant via un bus à la mémoire et aux divers périphériques

**Le processeur,** l'unité centrale ou CPU fait le traitement de l'information en exécutant un programme qui se trouve en mémoire. Pour cela il lit, une à

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

une, les instructions et les données depuis la mémoire pour les traiter. Cette lecture et ces traitements se font par succession de petites étapes, rythmées par des impulsions régulières (les *tops*) de l'horloge interne. C'est la fréquence de ces impulsions, et par corollaire la fréquence des petits traitements élémentaires, qui donne la vitesse du processeur (p. ex. 2 GHz, pour les architectures `x86` actuelles).

Le processeur dispose également de petites zones de mémoire très rapides, appelées *registres*. Ces registres peuvent contenir un mot binaire ou *mot machine* qui représente, soit une donnée de traitement élémentaire, soit l'adresse d'une zone dans la mémoire[1], lui permettant ainsi de stocker des résultats intermédiaires et y accéder de manière plus efficace et rapide que s'il s'agissait de la mémoire vive classique.

Le mot machine est l'unité de traitement de base de l'UC, et contient typiquement une instruction avec ses paramètres, une donnée élémentaire (entier ou flottant), ou une adresse de mémoire. Souvent d'autres données élémentaires peuvent être codées sur moins d'un mot machine (caractère, booléen) ou plus (entiers longs ou flottants à double précision). À l'origine, les mots machine étaient de taille quelconque (36 bits pour l'IBM 701, en 1953, 18 bits pour le DEC PDP-1, en 1961, ou 27 bits pour l'Electrologica X8 en 1968). Maintenant ils sont systématiquement des multiples de 8 et puissances de 2 : 8, 16, 32, 64 ou 128 bits.

Les mots machine forment, comme on l'a déjà vu, l'unité de traitement de base de l'UC. Afin d'exécuter un programme, logé dans la mémoire de l'ordinateur, le processeur agit comme suit :

1. Il charge dans un registre particulier l'adresse de la première instruction à exécuter[2], appelons la `@A1`. Le registre en question s'appelle *compteur ordinal*, *program counter* ou encore *instruction pointer* (`%eip`). Il contient l'adresse de l'instruction courante à exécuter.
2. Via le bus, il charge le mot machine se trouvant à `@A1`. Ce mot machine est une suite de *bits*, dont la première partie correspond à un code instruction, et la deuxième partie contient des paramètres. Soit il s'agit de petits drapeaux binaires, paramétrisant l'exécution de la commande, soit ce sont des opérandes de l'instruction. Dans ce cas ils représentent des registres (dans lesquels on suppose que les valeurs des opérandes ont été chargées au préalable) ou des adresses mémoire censées référencer les valeurs exactes.
3. Il exécute la commande ainsi chargée, ce qui résulte en une modification du *compteur ordinal*, ou la valeur de `@A1`. Le processeur réitère ensuite

[1] *cf.* p. 7 pour la définition d'une adresse mémoire.
[2] Comment sait-il où et comment obtenir cette adresse ?

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

les étapes 2. et 3. jusqu'à la fin du programme.

**La mémoire** vive[3] contient indifféremment des instructions à exécuter ou des données à traiter. Il est important de noter que strictement rien ne permet de déterminer, *a priori*, si une zone de la mémoire contient une instruction ou une donnée.

La mémoire est organisée comme un tableau de *mots* binaires. Chacun de ses mots est référencé par son indice dans ce tableau, et peut être lu ou écrit en fournissant cet indice. Cet indice est appelé *adresse mémoire*. Il est à noter que ces mots binaires ne sont pas forcément de la même taille qu'un *mot machine*. Généralement leur taille est un diviseur entier, puissance de 2, du mot machine, permettant de référencer des données élémentaires de taille inférieure à un mot machine.

Il appartient au programme, exécuté par le processeur, de savoir si l'adresse mémoire qu'il manipule référence une entité d'un seul mot mémoire, ou de plusieurs mots consécutifs, et d'agir en conséquence. On note que la taille d'une instruction plus ses paramètres est standardisé à une taille fixe, ce qui facilite l'incrémentation du compteur ordinal.

**La structure d'un programme** logé dans la mémoire, et exécuté par le processeur a généralement la forme présentée dans FIG. 1.3. Cet exemple est volontairement simplifié et ne tient pas compte de toutes les conventions de liaison, notamment de stockage des adresses de retour des fonctions. Ceci est en dehors de la portée de ce cours.

Utilisation des registres d'état assembleur.c

```
// Déclaration d'une fonction prenant 2 paramètres et retournant un entier
int additionne(int a, int b)                                          additionne
{
  int resultat ;          // Création d'une variable locale
  resultat = a + b ;      // Calcul + affectation α la variable locale
  return resultat ;       // Retour du contenu de la variable locale
}

// Déclaration de variables globales initialisées
int arg1 = 5 ;                                                              10
```

[3] On peut distinguer la mémoire vive ou RAM (*Random Access Memory*), dans laquelle on peut écrire et lire à son gré, de la mémoire morte ou ROM (*Read Only Memory*) qui est préprogrammée et que l'on ne peut que lire. Cette différence mise à part, les deux se traitent exactement de la même façon, et nous ne considérerons, dans la suite, que la RAM.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

```
int arg2 = 1;

// Déclaration de la fonction principale
int main()                                                    main
{
  arg1 = additionne(arg1, arg2); // appel de fonction, passage de paramètres et
                                 // récupération + affectation du résultat
}
```

Les noms de registres `%cs`, `%eip` et `%esp` correspondent respectivement à l'adresse de base (*Code Segment*), le compteur ordinal (*Extended Instruction Pointer*), et le pointeur de pile (*Extended Stack Pointer*).

| | | |
|---|---|---|
| Données statiques | `0000 0000 0000 0101`<br>`0000 0000 0000 0001` | `// int arg1 = 5`<br>`// int arg2 = 1` |
| Instructions | `subl $4, %esp`<br>`movl 12(%ebp), %eax`<br>`addl 8(%ebp), %eax`<br>`movl %eax, -4(%ebp)`<br>`ret`<br><br><br>`pushl arg2`<br>`pushl arg1`<br>`call additionne`<br>`addl $16, %esp`<br>`movl %eax, arg1` | La fonction `additionne`<br><br>`// int resultat (pile)`<br>`// 'b' en registre A`<br>`// ajout de 'a' au registre A`<br>`// copie A en pile (affectation)`<br>`// return (valeur en A)`<br><br>La fonction `main`<br><br>`// ajout argument 2 à la pile`<br>`// ajout argument 1 à la pile`<br>`// appel de fonction`<br>`// nettoyage de la pile`<br>`// copie du résultat dans arg1` |
| Pile | ...<br>`0000 0000 0000 0110`<br>`0000 0000 0000 0101`<br>`0000 0000 0000 0001` | Emplacements vides<br>`// resultat`<br>`// argument 1`<br>`// argument 2` |

FIG. 1.3 – Exemple simplifié de la représentation d'un programme en mémoire, chaque instruction écrite en langage d'assemblage a une représentation binaire unique qui n'est pas reprise ici pour des questions de lisibilité. La version complète et entièrement commentée se trouve en Annexe A, p. 131.

Ce qui est important de noter est qu'un programme comporte 3 parties distinctes, la première, contenant les données statiques, connues à la compila-

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

tion ; la seconde, contenant les instructions à exécuter ; et la troisième, la pile contenant les données intermédiaires résultant de l'exécution du programme (résultats intermédiaires ne pouvant pas être stockés dans des registres, paramètres d'appel de fonction, adresses de retour des fonctions, *etc.*).

Trois registres sont fondamentaux à l'exécution du programme :

`%cs` contient l'adresse de base (*i.e.* le premier mot mémoire) du programme.

`%eip` est le compteur ordinal. Ce registre s'incrémente au fur et à mesure que le programme s'exécute. Il est initialisé à l'adresse de la fonction `main`.

Des instructions de type `call` le font pointer sur une adresse passée en paramètre (branchement).

`%esp` est le pointeur de pile, il évolue dynamiquement avec l'utilisation de la pile, au fur et à mesure des allocations de données intermédiaires.

Il existe d'autres informations liées à l'exécution du programme (le registre d'état, le tas – ou *heap* –, *etc.*). Nous les évoquerons dans la suite au gré des besoins.

**Le bus** est le « tuyau » de communication entre le processeur, la mémoire et les périphériques, permettant le transfert de données entre eux. Dans les systèmes actuels, le processeur et la mémoire partagent un bus spécialisé. Un second bus relie tous les périphériques avec la mémoire et le processeur. Il est gère par un *arbitre de bus* qui en alloue son utilisation (exclusive) à qui en fait la demande (processeur ou périphérique). Lorsque le bus est occupé par une partie du matériel, les autres ne peuvent l'utiliser[4], notamment le processeur. Par contre, les périphériques ont généralement un accès direct à la mémoire (DMA – Direct Memory Access), ce qui fait que, lorsqu'un transfert disque/mémoire est en cours, le processeur peut continuer à exécuter les instructions qu'il a dans son *pipeline*[5]. En général, les périphériques ont la priorité sur l'utilisation du bus. On dénomme le phénomène où le processeur est en attente de la libération du bus *vol de cycle*. Dans les systèmes actuels, on distingue aussi le bus interne (souvent bus IDE) d'autres bus (PCI, USB, Firewire, ...) qui y sont rattachés, formant ainsi un grand bus virtuel.

[4] La vitesse du bus est généralement inférieure à la vitesse de calcul du processeur (p. ex. 100Mhz. pour un bus, et 2Ghz pour le processeur). Ce qui implique que, lorsque le processeur fait appel au bus, il reste, *a priori*, inoccupé pendant un certain nombre de cycles.

[5] Les processeurs modernes disposent de moyens d'exécuter plusieurs instructions en parallèle par le biais d'un *pipeline*. De ce fait, ils peuvent se passer du bus pendant un nombre de cycles, le temps que ce dernier se libère, et qu'ils puissent charger les instructions suivantes.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

**Les périphériques** sont tous les autres parties du matériel qui constituent l'ordinateur : clavier, souris, disques, carte réseau, *etc.* Il n'est pas utile d'entrer dans les détails de la gestion des périphériques. Notons seulement que le processeur y accède à travers le bus, et qu'il communique aux contrôleurs du matériel en question en leur envoyant des instructions. Comme ces périphériques sont souvent très lents par rapport au processeur (5 à 7 ms pour accéder à un disque ($\approx$ 150–200 Hz.), 75 Hz. pour la vitesse de rafraîchissement d'un écran, ...) ils utilisent un système d'interruptions pour prévenir le processeur de la fin de l'opération demandée.

Globalement, on peut décrire le fonctionnement des interruptions comme suit :

1. Lorsqu'un périphérique est prêt à envoyer des données au processeur, il l'avertit le contrôleur d'interruption, via les circuits de contrôle dédiés, qu'il veut lancer une interruption (IRQ : *Interrupt ReQuest*).
2. Le contrôleur d'interruption identifie l'origine de la requête par un *vecteur d'interruption*[6], et signale l'arrivée d'une interruption au processeur.
3. Dès que le processeur est prêt à recevoir cette interruption (en général lorsqu'il a terminé l'instruction en cours), il lit le vecteur d'interruption, et envoie un acquittement au contrôleur.
4. Le contrôleur d'interruption efface le vecteur, et est prêt à gérér d'autres interruptions.
5. En même temps, le processeur consulte une table interne qui fait correspondre le *vecteur d'interruption* à une adresse mémoire qui référence des instructions permettant de gérer l'interruption.
6. L'UC sauvegarde dans la pile[7] le compteur ordinal et le registre d'état courant.
7. L'adresse correspondant au vecteur d'interruption remplace le contenu du compteur ordinal, et l'exécution continue à l'endroit spécifié par ce dernier. Si la fonction de gestion de l'interruption ne modifie pas le pointeur de pile, ou si elle le stocke quelque part, elle peut ensuite retrouver l'endroit où le processus actif se trouvait avant l'interruption.

Tout ceci est géré par les circuits logiques du processeur, et échappe complètement au contrôle du programmeur. L'arrivée d'une interruption fait donc brutalement changer le flux des instructions. Il est à noter que souvent, il existe une commande dans le langage de la machine permettant de masquer

[6] Sur les architectures `x86` le *le vecteur d'interruption* est une valeur sur 8 bits.

[7] *cf.* p. 9 pour la définition et le rôle de la pile

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

les interruptions (ou certaines interruptions). Dans ce cas les interruptions sont ignorées jusqu'à la désactivation du masquage.

C'est la procédure appelée par l'interruption qui va ensuite sauvegarder le reste des registres du processeur (notamment le pointeur de pile) avant de s'exécuter. À la fin de l'exécution, cette même procédure restaure l'état du processeur tel qu'il était avant l'arrivée de l'interruption, et le flux d'instructions reprend son cours normal.

## 1.2 Rôle d'un système d'exploitation

Suite à la description du fonctionnement de la couche matérielle de l'ordinateur, on peut faire plusieurs constatations :

1. Comment fait-on pour amorcer l'exécution d'un programme ? Lors de l'allumage la mémoire est vide, les interruptions éventuelles provenant du clavier n'ont pas de procédure d'interruption chargée en mémoire. *Idem* pour les disques.
2. Chaque fois que l'on écrit un programme, on doit prendre soin d'y inclure toutes les procédures de gestion des périphériques afin de pouvoir les utiliser.
3. Il existe plusieurs situations (principalement lors d'attentes d'entrées/sorties) où le processeur reste inactif, ce qui est gênant d'un point de vue de l'utilisation optimale des ressources.
4. Tout programme s'exécutant a, *a priori*, un contrôle total de la machine. Ceci veut dire qu'un bogue dans l'un des gestionnaires de ressources, ou dans le programme lui-même, peut détruire une partie des données, ou, envoyer des commandes néfastes à des périphériques, les endommageant.
5. Comment avoir un environnement de travail interactif (et non prédétermine en fonctionnant par lots) avec plusieurs intervenants qui potentiellement auront besoin de partager des ressources uniques ?

La réponse à toutes ces questions est l'utilisation d'un système d'exploitation. Globalement, un OS a deux types rôle à jouer : un rôle actif et un passif. La partie active d'un OS concerne la gestion de l'enchaînement et l'ordonnancement des processus (§ 1.3). Elle est constamment opérationnelle et hors contrôle des différents applicatifs. La partie passive, en revanche, permet de séparer des actions potentiellement dangereuses (principalement l'accès aux ressources – mémoire, disque) des actions courantes (calcul) afin de préserver l'intégrité du système. Elle regroupe donc tous les services auxquels les différents processus peuvent faire appel : accès à un périphérique,

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

demande de ressources (mémoire), création, activation ou arrêt de processus. Ces services sont disponibles, mais ne se déclanchent qu'à la demande expresse d'une application, à travers d'un appel système (13). On appelle *le noyau* de l'OS la partie active (ordonnanceur) ainsi que les parties réalisant les services passifs. L'interface entre le noyau et les applications est constitué de la bibliothèque des *fonctions système*, par le biais duquel celles-ci peuvent faire appel aux services du noyau.

### 1.2.1 Le noyau

Le noyau est donc le programme qui est lancé lorsque l'ordinateur démarre. Il est à différencier de l'interface que l'utilisateur perçoit à l'issue du démarrage. Celui-ci est une application lancée par le noyau et qui permet d'interagir avec lui, par le biais des appels système. Ceci explique la distinction des couches *Système d'exploitation*, *Interpréteur de commandes* et *Outils système* dans FIG. 1.1.

Le système d'exploitation à proprement dire, c'est le noyau. On interagit avec lui à travers l'interpréteur de commandes, éventuellement par le biais d'outils système, ou directement par l'appel de fonctions système.

Les fonctionnalités du noyau opèrent en mode non protégé (également appelé mode *noyau* ou mode *réel*), gèrent l'ordonnancement des différentes tâches (ou processus), l'allocation de la mémoire, et le contrôle des entrées/sorties avec les problèmes de conflits d'accès, de priorité ou d'interblocage qui peuvent en résulter.

Les fonctionnalités de plus haut niveau opèrent en mode protégé (ou mode utilisateur), et constituent une sorte de machine virtuelle ou étendue. Elle doivent faire appel aux fonctionnalités bas niveau pour l'utilisation des ressources critiques. Le système fournit aux applications des interfaces de programmation moins contraignantes et difficiles à utiliser que l'accès direct au matériel. Typiquement, on y trouve des outils à l'organisation des systèmes de fichiers sur disque, l'utilisation des périphériques, de lancement de processus, ou d'utilisation de mémoire dynamique.

Lors de l'exécution d'un programme en mode protégé, l'appel à une fonction système, le fait temporairement basculer en mode non protégé, le temps des opérations critiques, puis de revenir en mode protégé, à l'issue de l'appel en question.

On peut distinguer, dans des OS couramment utilisés deux types, en fonction du périmètre du noyau :

– des systèmes dits, monolithiques, comme Unix, intègrent une grande quantité de fonctionnalités dans le noyau et fournissent un gros système relativement lourd à maintenir, mais efficace à l'exécution. Le

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

noyau fournit ainsi tout, un ordonnanceur, des pilotes de périphérique, la gestion de la mémoire, les systèmes de fichiers, les politiques d'authentification, *etc.*.

– des systèmes dits à micro-noyau, qui tentent de réduire au minimum leur noyau (fonctionnalités bas niveau), et dans lesquels les autres fonctionnalités sont implantés comme des services auxquels on fait appel ou qui se font appel mutuellement. Ces systèmes ont l'avantage d'être très modulaires, mais il y a une surcharge liée à la gestion de la communication entre modules qui tend à pénaliser les temps d'exécution. Comme exemples, on peut citer Mach et Chorus, qui sont des mirconoyaux opérationnels. Ici, le noyau se contente uniquement de fournir des pilotes de périphérique, un ordonnanceur et un méchanisme de communication entre processus. Ensuite, des processus en mode utilisateur, se chargent du reste.

Microsoft a également introduit la classe des systèmes d'exploitation *client-serveur* qui sont en réalité des systèmes monolithiques allégés autour duquel on implante le maximum de fonctionnalités externes, comme dans l'approche micro-noyau. Les parties considérées comme trop cruciales restent implémentées dans le noyau lui-même.

### 1.2.2 Les fonctions système

Les fonctions système, comme expliqué précédemment, servent comme point d'entrée aux services du noyau, et permettent, par la même occasion de basculer le processeur du mode protégé en mode noyau.

#### 1.2.2.1 Pricipe de base

Les processeurs actuels associent un niveau de protection aux différentes zones mémoire (rendu possible grâce à la segmentation *cf.* § 1.4.6). Lorsque le processeur exécute des instructions provenant de ces zones mémoire, il les exécute dans le mode de protection associé[8]. Un nombre d'instructions (accès au bus, gestion des interruptions, mémoire brute) ne peuvent être utilisés dans le mode de protection le moins élevé. Par ailleurs, lorsque le processeur est dans un mode particulier, il ne peut accéder qu'aux instructions de mode équivalant ou supérieurs au sien.

[8]Sur la famille des processeurs $x86$ il y a 4 niveaux de protection, 0 étant la plus faible, 3 la plus forte

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

#### 1.2.2.2 Mécanisme

Le niveau de protection est déterminé par la zone mémoire dans lequel se trouve l'instruction courante. Une instruction de branchement ne peut aboutir que lorsqu'elle débouche dans une zonde mémoire de niveau de protection équivalente ou plus forte. Sinon, elle est refusée et provoque une erreur de segmentation (*cf.* p. 58).

En revanche, il est possible d'invoquer des instructions appartenant à une zone moins protégée par le biais des `trap`. En appliquant le shéma des vecteurs d'interruption, il est possible de définir au préalable, un tableau de « *points d'entrée* » qui sont des adresses de fonctions de zones moins protégées et autorisées à être invoquées depuis des zones plus protégées. L'instruction `trap()` permet, en fournissant le numéro de point d'entrée correspondant, d'effectuer un branchement à l'un de ces endroits bien précis et définis à l'avance par le système d'exploitation. Il s'agit ici, en quelque sorte, d'une définition d'une interface très rigide d'interaction entre les couches de protection du noyau.

C'est bien évidemment le système d'exploitation, en démarrant en mode noyau, qui détermine les *traps* une fois pour toutes. Les adresses des points d'entrée ainsi définis correspondent alors aux *fonctions systèmes* qui sont exportés et accessibles par les niveaux supérieurs de protection.

#### 1.2.2.3 Mise en œuvre

Comme les outils (programmes, logiciels) qui s'exécutent sur le système sont généralement écrits dans des langages de haut niveau (`C`, `C++`, JAVA ...) il n'est pas très réaliste d'exiger des développeurs de maîtriser les *traps* et d'effectuer les appels systèmes en langage d'assemblage à chaque fois qu'ils en ont besoin. Le système fournit donc une ou plusieurs bibliothèques (`win32.dll` sous Windows–NT/XP, `libc.so` sous Linux, ...) encapsulant tous les arcanes bas niveau dans des fonctions d'un langage de plus haut niveau. Cela permet aussi de décliner un même *trap* en plusieurs appels spécifiques, de tester la validité des paramètres, et de garantir de la portabilité des applications, indépendamment, parfois, des choix du système sousjacent.

## 1.3 Les processus

Le mode de fonctionnement des micro-ordinateurs, tel que nous l'avons vu dans les sections précédentes, présente un obstacle à l'utilisation optimale de la puissance de calcul du processeur. Afin de fonctionner pleinement, celui-ci

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

doit communiquer avec des périphériques qui sont plusieurs ordres de grandeurs plus lentes que lui. Cette communication se fait de façon asynchrone, et le périphérique sollicité prévient le processeur de la fin de la tâche par une interruption. Dans un contexte de traitement par lots, où tout traitement se déroule de façon séquentielle, le processeur doit attendre de précieux cycles afin d'obtenir la réponse de sa requête auprès du périphérique.

### 1.3.1 La multiprogrammation

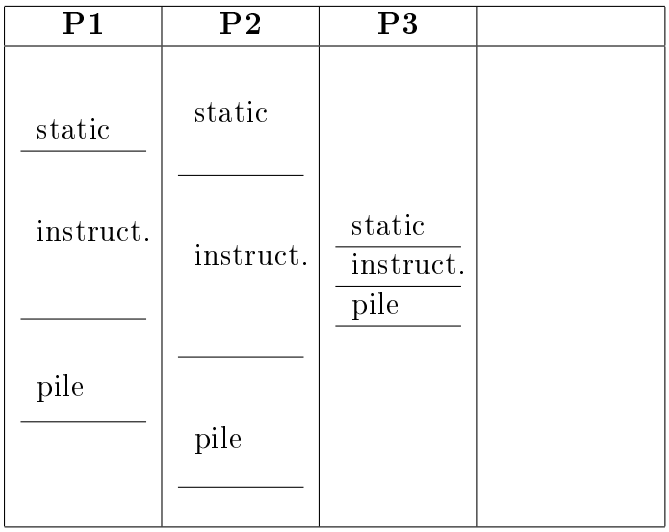


FIG. 1.4 – Représentation de la mémoire RAM comme un tableau bidimensionnel dans lequel sont stockés différents processus, selon le schéma vu en FIG. 1.3

La solution couramment adoptée pour optimiser l'attente des processus est l'utilisation de processus selon le principe de la multiprogrammation (ou *multi-tasking* en anglais). Un processus est une *unité d'exécution principale* et qui correspond à une tâche qui doit être effectuée par le processeur. C'est typiquement un programme qui s'exécute, mais dans le cas général il n'y a pas d'équivalence entre processus et programme ; ce dernier pouvant être constitué de plusieurs processus collaboratifs.

Le principe de la multiprogrammation, introduit par IBM sur leur OS/360, et ensuite étendu au partage de temps (ou *time-sharing*) au MIT au début des années 1960, consiste à loger simultanément plusieurs processus en mémoire (*cf.* FIG. 1.4). L'idée de base est ensuite de commuter entre les processus (*i.e.* exécuter un processus, puis le suspendre pour en exécuter un autre, *etc.*) selon les besoins.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

Ceci mène à une sorte de pseudo-parallélisme des différents processus. Conceptuellement, vu du processus, il est le seul à être exécuté sur le processeur. En réalité, le processeur n'exécute qu'un processus à la fois, puis le suspend pour une certaine durée, en exécute un autre, le suspend, *etc.*

On distingue principalement deux types de systèmes de multiprogrammation : les systèmes non préemptifs et les préemptifs. Les premiers sont maintenant largement obsolètes.

**La multiprogrammation non préemptive** est représentée dans FIG. 1.5. À chaque processus est associé un état par l'OS : *élu*, *prêt*, *bloqué* avec différentes transitions entre les états. Le(s) processus élu(s) (il peut y en avoir plusieurs dans le cas d'un système multiprocesseur) sont exécutés par le(s) processeur(s). Lorsqu'un processus élu doit attendre l'arrivée d'une interruption d'entrée/sortie, le système le bascule dans l'état *bloqué* et l'ordonnanceur choisit, parmi les processus en attente dans l'état *prêt*, un autre processus qui passe en état *élu* et qui s'exécute jusqu'à être bloqué à son tour.

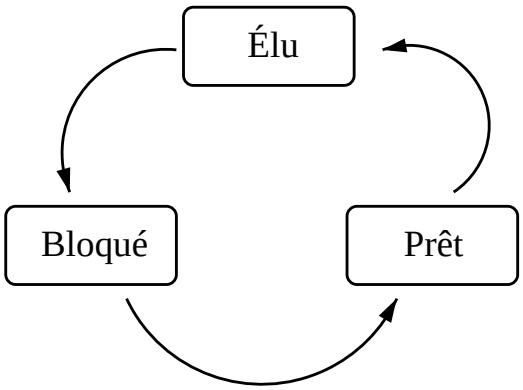


FIG. 1.5 – La multiprogrammation non préemptive : un processus élu s'exécute sur le processeur jusqu'à ce qu'il soit bloqué sur une entrée/sortie.

Il est clair, dans ce schéma, qu'il est absolument impossible de prévoir le temps de traitement total nécessaire à l'exécution d'un processus donné. Un processus nécessitant un grand nombre d'E/S passera plus de temps en attente qu'un processus faisant uniquement du calcul. Il convient à des situations où tout le traitement concerne des tâches de traitement de lots ou l'ordre et la réactivité d'exécution importent peu. Ce mode de fonctionnement ne convient, par contre, absolument pas à des environnements interactifs (nécessitant justement beaucoup d'E/S – souris, clavier, *etc.*).

Dans certains cas (lorsqu'on contrôle parfaitement les processus qui tourneront sur le système ... souvent des systèmes embarqués) on peut avoir

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

recours à des systèmes non préemptifs. Cela impose une rigeur de bonne conduite aux processus, mais allège la tâche du système.

**La multiprogrammation préemptive** introduit une nouvelle transition dans le graphe d'état, permettant de faire passer un processus de l'état *élu* à l'état *prêt* sans que celui-ci soit nécessairement bloqué par une demande d'E/S (*cf.* FIG. 1.6).

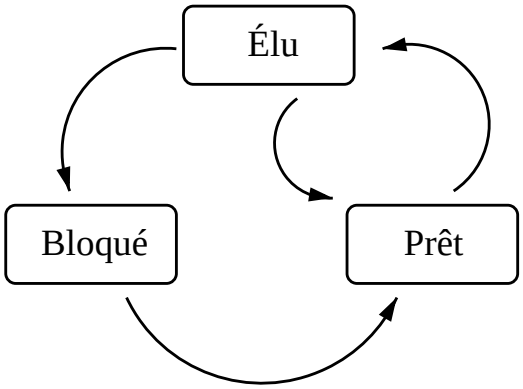


FIG. 1.6 – La multiprogrammation préemptive : un processus élu s'exécute sur le processeur jusqu'à ce qu'il soit bloqué sur une entrée/sortie ou que l'ordonnanceur le fasse basculer en état *prêt*.

Dans cette situation, l'OS joue un rôle beaucoup plus actif que dans le modèle non préemptif. Il peut décider, à tout moment d'interrompre le processus actuel en cours (soit sur l'entrée d'une interruption E/S, soit parce que son quantum de temps d'exécution a expiré, p. ex.) pour en activer un autre. Ceci permet donc de gérer plus facilement l'ordre d'exécution et la répartition des processus sur le(s) processeur(s).

Par contre, ce modèle nécessite plus d'intervention de la part de l'OS. Notamment, il doit disposer d'un *ordonnanceur* qui décide quand interrompre le processus en cours et quel processus élire par la suite. Les différentes stratégies d'ordonnancement seront abordées p. 20.

### 1.3.2 Rôle de l'OS

Le système d'exploitation, comme décrit dans FIG. 1.1, fournit d'une part l'interface entre les outils système de haut niveau et des applications utilisateur, et d'autre part l'interface avec les couches matérielles.

Il doit donc permettre de lancer et de créer des processus, afin que les couches supérieures puissent s'exécuter. Ceci mène naturellement à une struc-

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

ture arborescente d'hiérarchie de processus. Un processus père peut lancer plusieurs processus fils (p. ex. la fonction système `fork` sous Unix) qui à leur tour peuvent lancer des processus fils, *etc.* Ces processus doivent ensuite pouvoir communiquer des informations entre eux.

Le système doit également gérer le chargement, le déchargement de processus en mémoire. Il doit pouvoir les commuter sans perte du flot d'exécution intra-processus. Il doit aussi décider le l'ordre d'exécution, la priorité et le temps alloué à chaque processus (*quantum*) avant de le suspendre, p. ex.. Pour cela, il utilise un *ordonnanceur* qui prend les décisions quant à quel processus relancer, et un noyau d'une taille minimale qui s'occupe de gérer les interruptions et l'occupation en mémoire des processus.

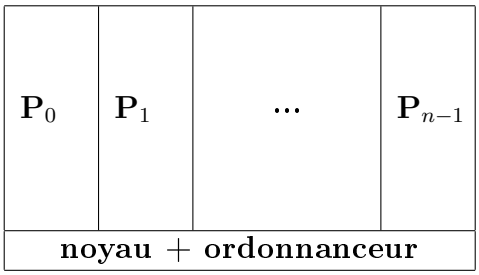


FIG. 1.7 – Représentation d'un OS à micro-noyau. Le cœur de l'OS est constitué d'un petit noyau contenant un ordonnanceur et un gestionnaire des interruptions et de chargement/déchargement des processus, sur lequel s'appuient les processus qui s'exécutent.

### 1.3.3 Mise en œuvre

Afin de pouvoir réaliser la commutation entre processus, le système stocke un ensemble d'informations, relatives à chaque processus dans sa *table de processus*. Cet ensemble d'informations contient notamment l'endroit en mémoire où ils se trouvent (leur adresse de base) plus d'autres informations décrivant leur état.

Regardons maintenant le graphe d'état des processus pour les systèmes préemptifs (FIG. 1.6). Dans ce graphe on observe 4 transitions possibles entre les états. On notera les transitions de la façon suivante :

$T_1$ la transition *élu – bloqué*,

$T_2$ la transition *bloqué – prêt*,

$T_3$ la transition *élu – prêt*,

$T_4$ la transition *prêt – élu*.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

**La transition $T_1$,** comme nous l'avons vu, est provoquée par l'attente d'un processus lors d'une requête E/S. Or comme les ressources des périphériques sont uniquement accessibles à travers des appels système[9], il est facile pour l'OS de gérer cette transition : lors d'une demande de ressources, le système lance la commande bas niveau appropriée, puis bascule le processus demandeur en un état *bloqué*, puis interroge l'ordonnanceur pour savoir quel processus activer.

**La transition $T_2$,** est provoquée par l'arrivée d'une interruption signalant la fin d'une opération d'E/S. Là encore, la gestion est conceptuellement simple. Comme décrit p. 10, l'arrivée d'une interruption provoque instantanément un changement de contexte[10], et l'exécution d'une procédure de gestion de l'interruption. C'est cette procédure qui se charge de changer l'état du processus auquel l'interruption était destinée de *bloqué* en *prêt*. Ensuite, l'OS interroge éventuellement l'ordonnanceur pour savoir quel processus activer, ou relance le processus, arrêté par l'arrivée de l'interruption (dans le cas contraire, le système aura bien-sûr pris soin de changer l'état de celui-ci de *élu* en *prêt*).

**La transition $T_3$,** concerne le cas où le système décide d'interrompre un processus en cours pour en activer un autre. Conceptuellement ceci se fait de façon identique à la réalisation de la transition $T_1$. En effet, un système préemptif garde le contrôle de ces processus en armant une horloge, qui envoie des interruptions avec une période régulière. Un processus dispose de cette façon d'un *quantum* de temps pendant lequel il dispose du processeur. L'arrivée de l'interruption redonne la main au système qui fait basculer le processus de son état *élu* en état *prêt*. Ensuite, l'OS interroge l'ordonnanceur pour savoir quel processus activer ensuite.

**La transition $T_4$,** s'opère lorsque l'OS a repris la main après une interruption. L'ordonnanceur lui indique le processus à faire passer en état *élu*, ce que fait le système d'exploitation.

[9] Effectuer un appel système consiste à invoquer une fonction bien définie qui « rend la main » au système d'exploitation de façon temporaire, afin qu'il fasse une opération en mode noyau et qu'il rende la main (ainsi que le résultat de l'opération) par la suite.

[10] On appelle *changement de contexte* le changement brusque du flot d'exécution d'un programme pour aller exécuter un autre programme. Cette terminologie s'applique principalement à la commutation de processus, lorsqu'on désactive un processus et on en active un autre.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

**Le chargement et le déchargement** d'un processus actif est relativement simple (pourvu que ce dernier se trouve déjà en mémoire quelque part). On a déjà vu qu'un processus est caractérisé par trois registres du processeur (en réalité il y en a quelques uns de plus, mais pour des raisons de clarté de l'exposé on les ignore) : `%cs`, `%eip` et `%esp` correspondant respectivement à l'adresse de base (*Code Segment*), le compteur ordinal (*Extended Instruction Pointer*[11]), et le pointeur de pile (*Extended Stack Pointer*). Lors de son exécution, le programme écrit et lit également des données intermédiaires dans les autres registres. Pour que la commutation entre processus reste transparente du point de vue des processus, *tous* ces paramètres doivent pouvoir être sauvegardés et restitués.

Considérons d'abord le déchargement d'un processus actif. On a vu que ceci est systématiquement provoqué par l'arrivée d'une interruption, et qu'une partie de la gestion des interruptions est complètement gérée par la couche matérielle (*cf.* p. 10). L'UC stocke notamment le compteur ordinal sur la pile, puis le remplace pour exécuter la fonction de gestion de l'interruption. Ensuite c'est à la fonction de gestion de l'interruption de faire le nécessaire (*cf.* FIG. 1.8).

Le chargement d'un programme n'est pas régi par des interruptions, comme le cas du déchargement. C'est toujours une décision du système, prise à un instant où c'est lui qui occupe le processeur. Lorsqu'il décide d'activer un processus $p_i$, il restaure, depuis les informations de sa table de processus, tous les registres contenant des données temporaires ainsi que le pointeur de pile `%esp` et l'adresse de base `%cs` du processus. Ensuite (puisqu'il connaît l'état de la pile, maintenant), il dépile le compteur ordinal et le restaure également. À ce moment précis c'est le processus restauré qui s'exécute.

### 1.3.4 Les algorithmes d'ordonnancement

Jusqu'ici nous nous sommes uniquement intéressés aux aspects techniques et algorithmiques de l'activation ou de la désactivation des processus. Nous avons vu, pourtant, que l'ordonnanceur joue un rôle important dans la commutation des processus. C'est lui qui décide quel processus élire pour l'exécution.

Il existe une littérature abondante sur le sujet de l'ordonnancement, et il s'agit d'un sujet de recherche tout-à-fait d'actualité. Néanmoins, pour des raisons d'efficacité en termes de surcharge du système, les implémentations des différents OS actuels se contentent d'utiliser les algorithmes les plus simples.

[11] Le terme *Extended* est un héritage de l'époque où les processeurs ne foncionnaient qu'avec des adresses sur 16 bits, et signifie que le registre fonctionne avec 32 bits.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

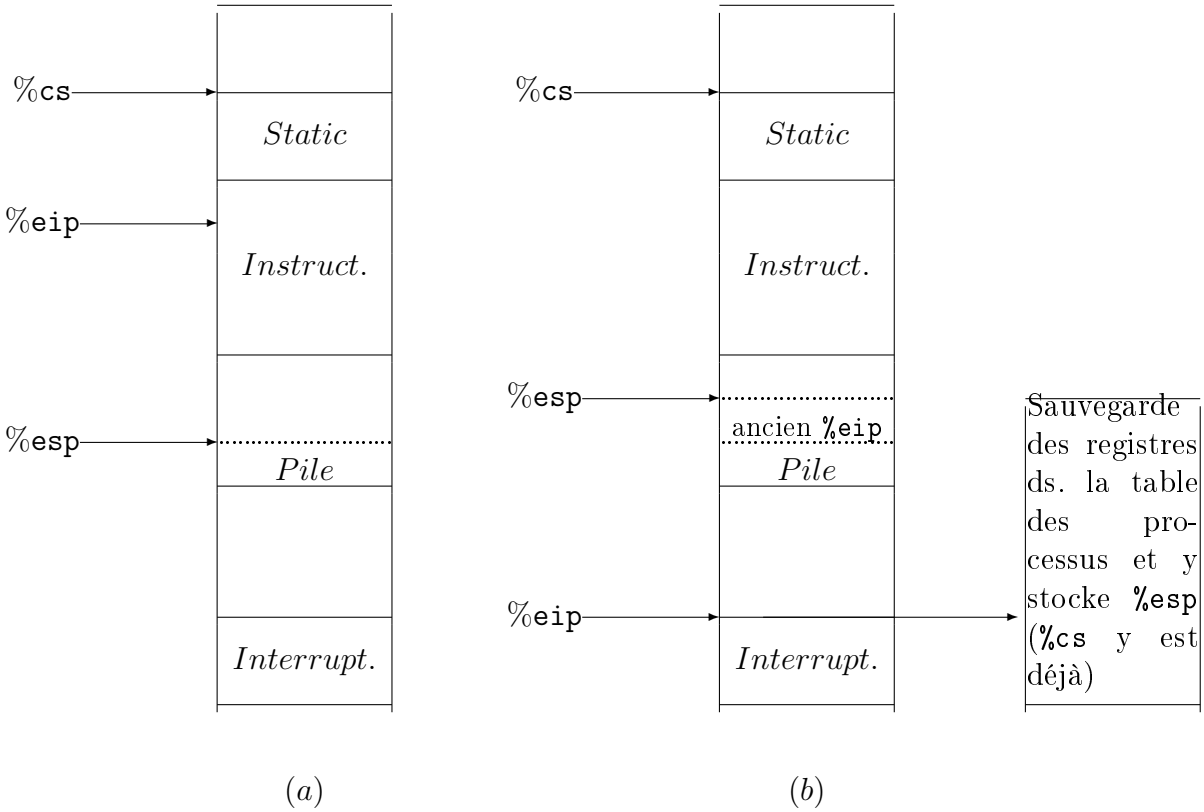


FIG. 1.8 – Schéma représentant le déchargement d'un processus actif en deux étapes. *(a)* Le processus actif s'apprête à exécuter l'instruction désignée par son compteur ordinal (`%eip`). *(b)* L'arrivée d'une interruption fait sauvegarder ce compteur dans la pile, et le fait ensuite pointer sur la fonction de gestion des interruptions qui, elle, sauvegarde l'état du processus dans la table du système.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

Nous nous bornerons qu'à l'étude d'un seul algorithme et une dérivée. Dans des environnements temps réel, en revanche, la politique d'ordonnancement des processus devient cruciale, dû principalement aux nécessités de garanti de temps de réponse.

**L'ordonnancement circulaire,** également appelé tourniquet, est l'algorithme d'ordonnancement le plus simple et le plus robuste. Chaque processus dispose d'un quantum pendant lequel on lui accorde l'exclusivité du processeur. Si le processus se bloque, s'il termine avant la fin de son quantum ou si son quantum expire, le système passe la main au processus suivant. L'OS doit donc seulement mémoriser une liste de processus prêts. Lorsque le premier a utilisé son quantum, il est retiré de la tête de la liste et réinséré en fin. Le processus qui le suivait dans la liste et maintenant le premier, et est exécuté. Des processus qui passent d'un état *bloqué* à *prêt* sont également insérés en fin de liste.

On note, au passage, qu'il existe une relation subtile entre la réactivité du système (la vitesse à laquelle un processus reprend la main) et son efficacité (la surcharge provoquée par l'exécution de tâches de gestion diverses, empêchant les processus de tourner). Si le changement de contexte requiert $c$ ms à s'opérer, et le quantum est fixé à $q$ ms, la surcharge introduite par le système est de $\frac{c}{c+q}$. Il y a donc intérêt à avoir une valeur pour $q$ qui soit la plus grande possible. Par contre plus $q$ croît, plus la réactivité du système diminue.

**L'ordonnancement avec priorité** est une variante du schéma précédent. Chaque processus dispose d'une priorité. L'ordonnanceur choisira d'activer systématiquement le processus ayant la plus haute priorité. Souvent même, les processus avec le même niveau de priorité sont regroupés dans une même file sur laquelle on applique l'ordonnancement du tourniquet, jusqu'à ce qu'il n'y ait plus de processus à ce niveau de priorité, puis on descend, pour exécuter les processus de priorité inférieure, *etc.* (*cf.* FIG. 1.9).

Afin d'éviter qu'un processus très privilégié s'accapare tout le temps CPU il existe des méthodes d'ajustement dynamique de la priorité. L'un des moyens est de diminuer la priorité du processus en cours après chaque quantum qui expire, jusqu'à ce qu'il y ait un processus plus prioritaire que lui, et d'opérer le changement de contexte ensuite. Un autre moyen est de privilégier les processus demandant beaucoup d'E/S (typiquement les processus interactifs), en modulant leur priorité par la fraction de quantum non utilisée lors de son exécution précédente. Le facteur de modification de la priorité d'un processus ayant utilisé tout son quantum est de 1, tandis qu'un proces-

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

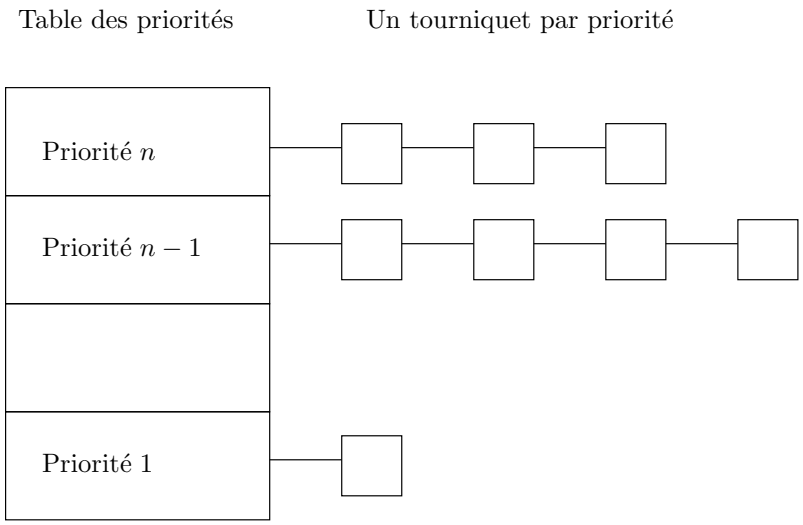


FIG. 1.9 – Ordonnancement avec différentes classes de priorité.

sus bloqué sur une E/S au bout d'une fraction $f$ de son quantum se verra attribuer un facteur de modification de $1/f$.

### 1.3.5 Les processus légers – *threads*

Jusqu'à maintenant nous avons considéré qu'un processus était l'équivalent d'un programme comprenant trois parties : les données statiques, les instructions et la pile ; le tout dans un bloc contigu de mémoire. Souvent, il existe une quatrième partie, qui s'appelle le *tas*, et qui est une zone de mémoire de laquelle le programme peut puiser de façon non organisée des blocs de mémoire, contrairement à la pile, dans laquelle on ne peut qu'empiler ou dépiler des objets par le dessus. C'est là notamment que des langages de haut niveau puisent les ressources lorsqu'ils font appel à `malloc()` en `C`, ou encore `new` en `C++` et JAVA. Le processus dispose en plus d'un nombre d'autres ressources (p. ex. le contenu des registres, les fichiers ouverts, ...) qui résident dans la table de description du processus gérée par le système.

Lorsque plusieurs utilisateurs lancent une même application sur une même machine, il est important de distinguer toutes ces zones mémoire, et il est donc important de pouvoir disposer de plusieurs instances (processus) d'un même programme. En revanche, il peut être tout aussi utile de faire effectuer des tâches en parallèle par un processus principal, tout en partageant certaines ressources mémoire entre le processus parent et ses fils, surtout que cela facilite grandement le partage et l'échange de données que, d'un point de vue système, le changement de contexte demande beaucoup moins de sur-

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

charge. Dans ce cas, on parle de threads. Chaque thread est un processus, du point de vue du système d'exploitation, mais il ne dispose uniquement de sa propre pile. Tous les autres ressources (zone statique, tas, *etc.*) sont partagées avec le processus parent et les fils de celui-ci.

**Exemple 1.1 : Threads en JAVA**

Cet exemple montre la facilité de créer des threads en JAVA. Il suffit de définir une classe héritant de `Thread` et surchargeant la fonction `run()`. Ensuite l'appel à la fonction membre `start()` fait le nécessaire de démarrer le thread, et d'exécuter `run()`.

Dans cet exemple, deux threads concurrents ajoutent une valeur aléatoire entre 0 et 5 à une variable partagée `compteur`, jusqu'à ce qu'elle dépasse la valeur 100. A l'exécution, notez des situations de concurrence, où un thread peut manipuler la valeur de `compteur` alors qu'en même temps, l'autre la modifie.

```
public class ThreadExemple extends Thread {

  // Facteur d'attente avant exécution du thread dans l'intervalle
  // [0, RandomSleepFactor[ millisecondes. Ici : 1 seconde
  private static final int RandomSleepFactor = 1000;

  // La variable partagée par tous les threads
  private static int compteur = 0;

  // La fonction principale, se contentant de créer 2 threads
  // et de les démarrer.
  public static void main(String[] args) {

    ThreadExemple t1 = new ThreadExemple("Premier thread");
    ThreadExemple t2 = new ThreadExemple("Second thread");

    t1.start();
    t2.start();
  }

  // Le constructeur ... sans beaucoup d'intérêt
  public ThreadExemple(String nom) {  super(nom);  }

  // Le cœur du processus :
  //   1/ on prend la valeur de la variable partagée (oldcompteur)
```

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

```
  //    2/ on crée une variable temporaire (newcompteur)
  //       qui prend la valeur de la variable partagée incrémentée
  //       d'une valeur aléatoire.
  //    3/ on affiche les valeurs lues et calculées
  //    4/ on attend un peu
  //    5/ on met jour la variable partagée
  public void run() {

    // On itère jusqu'à ce que quelqu'un fasse dépasser
    // compteur la valeur 100.
    while(compteur < 100) {

      // Une copie somme toute anodine
      int oldcompteur = compteur;
      // Attention ... newcompteur se sert de compteur et
      // non pas de oldcompteur ! Notez la différence fondamentale.
      int newcompteur = (int) (compteur + 5*Math.random());

      // L'affichage devrait permettre de voir s'il y a eu
      // un accès concurrent ... sans garantie aucune, bien évidemment
      System.err.println(getName() + ": oldcompteur = " + oldcompteur);
      System.err.println(getName() + ": newcompteur = " + newcompteur);
      System.err.println(getName() + ": compteur = " + compteur);

      // On provoque un peu le sort en se mettant en veille...
      try {
          sleep((int)(RandomSleepFactor * Math.random()));
      }
      catch(InterruptedException e) {}

      // Seulement maintenant la variable partagée est mise à jour.
      compteur = newcompteur;
    }

    System.err.println(getName() + " fini !");
  }
}
```

**Exemple 1.2 : Threads Posix en C**

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

On reprend le même exemple que précédemment en JAVA, mais avec les threads Posix[12]. On laisse le soin au lecteur de transposer cet exemple en C++.

Il est très utile d'observer l'exécution de ce programme, puisque la non-synchronisation des entrées-sorties fait que l'on voit bien le problème de concurrence d'accès à la variable partagée `compteur`.

Threads Posix – C threadexemple.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>

/* Facteur d'attente avant exécution du thread dans l'intervalle
   [0, RandomSleepFactor[ millisecondes. Ici : 1 seconde */

static int RandomSleepFactor = 1000;
static int compteur = 0;

void thread_function( void *ptr );

int main(int argc, char** argv) {

    pthread_t thread1, thread2;
    int  v1, v2;

    v1 = pthread_create( &thread1, NULL, (void*)&thread_function,
                        (void*) "Premier Thread");
    v2 = pthread_create( &thread2, NULL, (void*)&thread_function,
                        (void*) "Second Thread");

    /* On attend que les threads se terminent ... */
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    fprintf(stderr,"Le premier thread termine avec : %d\n",v1);
    fprintf(stderr,"Le second thread termine avec : %d\n",v2);
```

[12] Posix est une norme qui sera détaillé dans le prochain chapitre.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

```
    exit(0);
}

void thread_function (void *ptr) {                                    thread_function

    int oldcompteur;
    int newcompteur;

    double tmpvalue;                                                          40

    // attention α l'appel de fonctions non ré-entrantes (non thread-safe)
    // comme p.ex strcpy()
    char* nom = (char*)ptr;

    while(compteur < 100) {

      oldcompteur = compteur;
      tmpvalue = 5.0 * rand()/(double)(RAND_MAX);
      newcompteur = compteur + (int)(tmpvalue);                               50

      fprintf(stderr,"%s : oldcompteur = %d\n",nom,oldcompteur);
      fprintf(stderr,"%s : newcompteur = %d\n",nom,newcompteur);
      fprintf(stderr,"%s : compteur = %d\n",nom,compteur);

      tmpvalue = RandomSleepFactor * rand()/(double)(RAND_MAX);
      usleep((int)(tmpvalue));

      compteur = newcompteur;
    }                                                                         60

    fprintf(stderr,"%s fini !\n",nom);
}
```

### 1.3.6 Communication entre processus

Jusqu'à maintenant nous avons considéré les processus comme des entités isolées n'ayant pour seul objectif leur exécution sur le processeur, et n'interagissant pas avec le système ou avec d'autres processus. Pourtant, on a vu

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

que les *threads* partagent des zones de la mémoire entre eux, dans lesquels ils peuvent tous lire et écrire. De plus, des processus peuvent avoir besoin d'utiliser d'autres ressources communes.

Par exemple, la commande Unix suivante

```
$ cat f1 f2 f3 | grep "text"
```

exécute la commande `cat`, réalisant la concaténation du contenu des fichiers $f1$ $f2$ $f3$ passés en paramètre, et envoie son résultat via un *pipe* (l'opérateur `|`) à la commande `grep` qui effectue une recherche séquentielle et affiche toutes les lignes contenant le texte fourni comme paramètre. Dans ce cas les deux processus (`cat` et `grep`) partagent le *pipe* qui est une sorte de fichier virtuel dans lequel l'un lit et l'autre écrit. La généralisation de ce cas au partage de ressources conduit à une série de problèmes complexes d'interblocage ou de corruption de données qui nécessitent l'instauration de méthodes de synchronisation et d'exclusion mutuelle. C'est ce que nous allons étudier dans cette section[13].

**Un exemple** de corruption de données, fourni par TANENBAUM [1] concerne l'utilisation d'un spoule d'impression. Quand un processus veut imprimer, il doit placer un fichier dans le spoule d'impression. Régulièrement, un *démon*[14] regarde s'il y a des fichiers à imprimer, et les enlève de la queue d'impression.

Supposons maintenant que le répertoire de spoule en question ressemble à un tableau avec un nombre d'entrées illimitées, et que les processus voulant l'utiliser disposent de deux variables *prochain* et *libre*, pointant respectivement sur le prochain fichier à imprimer et la prochaine place libre dans le spoule. Deux processus $A$ et $B$ veulent imprimer en même temps. $A$ lit la valeur de *libre* et la stocke dans un registre. Sa valeur est égale à $n$. À ce moment l'usage du processeur lui est retiré (pour une raison quelconque) et l'ordonnanceur l'alloue à $B$. $B$ lit la valeur de *libre* et la stocke dans un registre. Sa valeur est toujours égale à $n$. $B$ incrémente la valeur du registre, le stocke dans *libre*, et écrit son fichier à l'emplacement $n$. Par la suite $A$ retrouve l'accès au processeur, et continue son exécution. Son état interne n'est

[13] *Stricto sensu* il serait nécessaire de distinguer les problèmes de synchronisation et d'exclusion mutuelle. Néanmoins, on peut résoudre les uns avec des solutions prévues pour les autres, l'élégance parfois mise à part.

[14] On appelle couramment *démon* un processus (généralement faisant partie des outils système) qui est inactif pendant la plupart du temps (il se trouve en un état *bloqué* en attendant qu'un événement particulier le fait passer en état *prêt*). À son activation il effectue une tâche bien particulière, puis se remet en état *bloqué* en attendant l'événement suivant.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

plus cohérent avec la valeur de *libre* (qui est maintenant à $n+1$), mais il n'a aucun moyen de s'en apercevoir (le changement de contexte est transparent pour les processus). Il incrémente la valeur de son registre, le stocke dans *libre* (ce qui ne change pas sa valeur), et écrit son fichier à l'emplacement $n$, écrasant par la même occasion le fichier de $B$. Le démon d'impression n'en saura jamais rien, et imprimera uniquement le fichier de $A$.

Les situations de ce type, où deux processus ou plus lisent et écrivent des données partagées, et où le résultat dépend de l'ordonnancement des processus, sont qualifiées d'accès concurrents ou *race conditions*.

#### 1.3.6.1 Les sections critiques

Pour éviter les problèmes dans les situations décrites ci-dessus, il faut trouver un moyen d'interdire la lecture ou l'écriture des données partagées à plus d'un processus à la fois. Il faut une *exclusion mutuelle* (`mutex`) qui empêche les autres processus d'accéder à la ressource partagée si un processus est déjà en train de l'utiliser.

Le problème peut aussi être spécifié de façon formelle. La plupart du temps, les processus effectuent des calculs et des opérations qui ne conduisent pas à des situations de conflit. Seules certaines parties bien spécifiques du code font appel à des ressources partagées. On appelle ces parties les *sections critiques*. Le problème des conflits d'accès seraient résolus si on pouvait garantir que jamais deux processus puissent se trouver en même temps dans une section critique relative à une ressource commune.

Cette condition est suffisante pour éviter les conflits d'accès, mais elle ne permet pas à des processus parallèles (ou pseudo-parallèles dans le cas de multiprogrammation) de coopérer correctement et efficacement. Pour cela les quatre conditions suivantes sont nécessaires :

1. Deux processus ne peuvent pas être en même temps en section critique relative à une ressource commune.
2. Aucune hypothèse doit être faite sur les vitesses relatives des processus et sur le nombre de processeurs.
3. Aucun processus suspendu en dehors de sa section critique ne doit bloquer les autres.
4. Aucun processus ne doit attendre trop longtemps avant de pouvoir entrer dans sa section critique.

**Le masquage des interruptions** pourrait, à première vue, sembler une solution, certes brutale, mais efficace pour éviter qu'un processus soit interrompu pendant l'exécution de sa section critique. Non seulement cette

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

solution est périlleuse (aucun système raisonnablement conçu permettrait à l'utilisateur lambda de masquer les interruptions, courant ainsi le risque de ne jamais pouvoir reprendre la main sur le processus), mais en plus elle est inefficace dans le cas où plusieurs processus tournent sur plusieurs processeurs en parallèle. Le masquage est seulement valable pour un processeur donné d'une part, et même si ce n'était pas le cas, il ne pourrait pas empêcher un processus actif sur un autre processeur de faire des accès à la ressource partagée.

Il reste à noter que le masquage des interruptions est couramment utilisé par le noyau, lorsqu'il exécute des tâches critiques et sensibles.

#### 1.3.6.2 L'attente active

Les solutions de l'attente active consistent à gérer les problèmes d'accès concurrent sans intervention ou aide de la part du système d'exploitation. Elles sont citées ici par mesure de complétude et pour l'aspect algorithmique de la solution. On évoquera en fin de cette section deux critiques que l'on peut émettre contre leur utilisation.

L'allocation des ressources par alternance (FIG. 1.10) est une proposition au problème qui est simple et que nous proposons ici pour bien comprendre l'idée de l'attente active. Elle n'est pas applicable aux cas généraux, puisqu'elle suppose que les processus accédant aux ressources s'exécutent à la même vitesse, et font des accès en alternance. Elle repose sur un compteur `tour` qui indique l'identifiant de processus qui a le droit d'utiliser la ressource et qui est partagée entre les deux processus. Lorsqu'un processus a terminé sa section critique, il incrémente `tour`, modulo le nombre de processus, pour que sa valeur pointe sur le processus suivant. Lorsqu'un processus est en attente de l'allocation de la ressource, il exécute une boucle infinie. C'est le fait d'exécuter des commandes inutiles (*i.e.* la boucle infinie) qui forme la caractéristique principale de l'attente active : un processus en attente occupe le processeur avec des opérations inutiles.

En dehors de l'hypothèse de l'alternance la solution présentée dans FIG. 1.10 ne marche pas. Supposons, par exemple, que le processus (a) est un processus très rapide, et qu'il vient de terminer sa section critique. (b) est très lent, est actuellement en section non critique, et n'aura pas besoin d'entrer en section critique avant très longtemps. (a) termine rapidement sa section non critique, et veut à nouveau entrer en section critique. Il ne pourra pas, puisque la variable `tour` n'est pas positionnée correctement.

**La solution de PETERSON** (1981) offre un algorithme élégant et simple à l'exclusion mutuelle. Elle est représentée dans FIG. 1.11. La première solution

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

```
while (TRUE) {
  while (tour != 0) /* attente */
  section_critique() ;
  tour = 1 ;
  section_noncritique() ;
}
```

(a)

```
while (TRUE) {
  while (tour != 1) /* attente */
  section_critique() ;
  tour = 0 ;
  section_noncritique() ;
}
```

(b)

FIG. 1.10 – Exemple de partage de ressources par alternance entre deux processus [1].

formelle, datant de 1965 et due à T. DEKKER, était beaucoup trop complexe pour effectivement être mise en œuvre.

Dans cette solution, chaque processus doit, avant d'utiliser des variables partagées, appeler `entrer_region()` en lui fournissant son identifiant de processus en paramètre. Cet appel ne retournera la main que lorsqu'il n'y a plus de risque. Dès que le processus n'a plus besoin de la ressource, il doit appeler `quitter_region()` pour indiquer qu'il sort de sa section critique et que les autres processus peuvent accéder à la ressource partagée. L'idée principale de l'algorithme est d'utiliser deux variables différentes, l'une indiquant que l'on s'apprête à utiliser une ressource commune (`interesse[]`), et l'autre (`tour`) qui, comme dans le cas de l'alternance, indique le processus qui utilisera la ressource au prochain tour. Lorsqu'un processus entre en section critique, il notifie les autres de ses intentions en positionnant correctement son entrée dans le tableau `interesse[]`, puis il s'approprie le prochain tour. Tant que lui, est le prochain utilisateur (`tour == process`), mais l'autre processus a également l'intention d'utiliser (ou utilise déjà) la ressource, le processus courant attend que l'autre libère la ressource. En fin de section critique, en appelant `quitter_region()`, le processus sortant signale aux autres qu'il n'utilise plus la ressource.

Il est à noter que toute la partie traitée dans ce cours concernant les exclusions mutuelles, repose fondamentalement sur le respect des règles d'utilisation de tous les processus voulant accéder aux ressources partagées. Ici, par exemple, on utilise un tableau partagé (`interesse[]`) dans lequel chaque processus indique son intention d'utiliser la ressource partagée. La règle implicite est, bien-sûr, que chaque processus n'écrit qu'à l'indice du tableau qui lui est dédié, et non pas aux autres endroits.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

```
#define N     2                          /* nombre de processus */

 /* Les variables partagées par tous les processus */
int tour;                                      /* à qui le tour */
int interesse[N];                         /* initialisé à FALSE */

void entrer_region(int process)  /* numéro du processus appelant */
                                  /* ici process vaut 0 ou 1 */
{
   int autre;                  /* numéro de l'autre processus */

   autre = 1-process;
   interesse[process] = TRUE;          /* indiquer son intérêt */
   tour = process;           /* positionner le drapeau d'accès */
   while(tour == process && interesse[autre] == TRUE);
}

void quitter_region(int process)  /* numéro du processus appelant */
                                  /* ici process vaut 0 ou 1 */
{
   interesse[process] = FALSE;          /* indiquer la sortie de
                                          la section critique */
}
```

FIG. 1.11 – L'exclusion mutuelle par attente active par PETERSON (1981).

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

Le lecteur intéressé est invité à tester la solution de Peterson en y introduisant, à des endroits aléatoires[15], des changements de contexte, et vérifier qu'effectivement l'exclusion mutuelle est respectée.

**Exercice :** Que se passe-t-il lorsque l'on intervertit les deux lignes

```
interesse[process] = TRUE ;
tour = process ;
```

dans la fonction `entrer_region()` ?

En étudiant de près les problèmes liés à l'exclusion mutuelle et la corruption de données qui peut en découler, on constate que le point critique se trouve entre la lecture d'une variable et son écriture. Si un changement de contexte intervient entre ces deux actions, et un autre processus écrit dans la variable après que la première l'ait lue, le premier se trouve dans un état incohérent, et peut, ensuite provoquer des actions néfastes (comme l'exemple du spoule d'impression, p. 28). Cette analyse a conduit, sur certaines architectures, à introduire une *commande atomique*[16] au niveau du langage d'assembleur, permettant à la fois de lire et d'écrire une variable en une instruction.

**L'instruction TSL** (*Test and Set Lock*) prend comme paramètres un registre et une adresse mémoire (partagée par tous les processeurs) appelée *drapeau*. L'instruction charge le contenu du drapeau dans le registre, et en même temps, elle met la valeur du drapeau à 1. Cette instruction est garantie comme étant atomique, c'est-à-dire que l'arrivée d'une interruption ne sera acquittée par le processeur que lorsque l'opération du TSL sera terminée (on rappelle qu'une instruction machine peut prendre plusieurs cycles horloge à s'exécuter, si la demande d'interruption arrive avant, elle ne sera acquittée qu'à la fin du dernier cycle de la commande TSL). L'autre avantage de cette instruction est qu'elle occupe le bus pendant toute son exécution. De ce simple fait, elle empêche d'éventuels autres processeurs d'accéder en même temps au *drapeau*.

Du fait de cette atomicité, les fonctions `entrer_region()` et `quitter_region()` peuvent êtres écrites en assembleur de la façon suivante :

[15] L'endroit le plus vicieux étant entre les deux évaluations de la boucle *while*

[16] On appelle *commande atomique*, une commande, fournie par le système d'exploitation, dont on garantit qu'elle ne peut pas être interrompue par un changement de contexte. Il s'agit généralement de fonctions très petites, à l'intérieur desquelles le système masque les interruptions.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

```
entrer_region:
  TSL  registre,drapeau     | copier drapeau dans registre,
                            | et mettre drapeau à 1
  CMP  registre,#0          | comparer le registre à 0
  BNZ  entrer_region        | si différent boucler
  RTS                       | retour à l'appelant

quitter_region:
  MOVE #0,drapeau           | écrire 0 dans le drapeau
  RTS                       | retour à l'appelant
```

Notez qu'entre la commande `TSL` et `CMP` on peut avoir un changement de contexte sans que cela ne remette en cause la solution.

Par contre il existe deux critiques que l'on peut émettre contre solutions de type attente active. L'une est qu'elle consomme des ressources du processeur. Un processus qui est en attente, effectue une boucle `while`, gaspillant ainsi des cycles du processeur. Une autre critique est qu'il existe une faille dans ce système. On l'appelle l'*inversion de priorité*.

Supposons que l'on ait deux processus $H$ et $B$ partageant une même ressource. $H$ est un processus prioritaire pour le système, et se trouve actuellement en attente d'une interruption (E/S ou horloge démon, p. ex.). $B$ est un processus ordinaire, qui vient d'entrer dans sa section critique. À ce moment précis, $H$ est activée par l'interruption qu'il attendait. Il ne peut pas entrer en section critique puisqu'elle est occupée par $B$. L'attente active fait qu'il boucle jusqu'à obtention des ressources. Malheureusement, le système, considérant $H$ comme prioritaire, ne changera jamais le contexte pour activer $B$, ce qui conduit à un interblocage.

#### 1.3.6.3 L'attente passive

L'attente passive demande une intervention de la part du système (qui doit par conséquent fournir les appels au noyau nécessaires), et évite une surcharge inutile du processeur. Il évite également le problème d'*inversion de priorité*, puisque l'idée de base consiste à basculer le processus en attente de l'accès à une ressource partagée en mode *bloqué*, jusqu'à ce que la ressource se libère, ce qui la fait basculer en mode *prêt*. Dans l'exemple précédent $H$ restait actif à attendre, $B$ qui ne pouvait jamais s'activer pour cause de priorité inférieure. Dans le contexte d'attente passive, $H$ est forcé en mode *bloqué* (priorité ou non), et $B$ peut sortir de sa section critique, libérant ensuite la place pour $H$.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

Il existe une quantité importante de solutions pour implanter des méthodes d'attente passive. On peut montrer qu'elles sont toutes équivalentes d'un point de vue formel (ce qui ne les rend pas toutes également efficaces ou faciles à utiliser dans la pratique). On se contentera d'étudier en détail les sémaphores, qui sont les plus répandus dans des langages « anciens » et les moniteurs, qui sont intégrés dans des langages plus « modernes » comme ADA ou JAVA.

**Les sémaphores** sont introduit par DIJKSTRA en 1965 pour pallier des problèmes d'une méthode plus ancienne basée sur des appels de `sleep()` (basculement en mode *bloqué*) et `wakeup()` (basculement en mode *prêt*) directs, qui pouvaient engendrer des interblocages. Ils sont maintenant largement utilisés dans les systèmes d'exploitation modernes.

Un *sémaphore* est un compteur d'accès à une ressource partagée, pouvant prendre toute valeur positive, et sur laquelle on peut effectuer deux opérations : `up()` et `down()`. Ces opérations sont garanties *atomiques* par l'OS.

La commande `up()`, incrémente le sémaphore. La commande `down()` décrémente la valeur du sémaphore si elle est strictement supérieure à zéro, puis rend la main à l'appelant[17]. Si la valeur du sémaphore vaut zéro, le processus bascule en mode *bloqué* jusqu'à ce que la valeur du sémaphore redevienne positive et que le processus soit réveillé par le système. Lors de son réveil, le processus décrémente la valeur du sémaphore et continue son exécution.

L'atomicité des primitives est primordiale pour des raisons évoquées précédemment. Dans le cas d'un système multiprocesseur, il est nécessaire de protéger chaque sémaphore par une variable verrou et de se servir d'une instruction `TSL` pour éviter que plusieurs processeurs accèdent au même sémaphore en même temps. Il est fondamental de noter que l'utilisation ici, dans ce cas précis, d'une instruction `TSL` ne génère pas les mêmes problèmes que dans le cas précédent. L'attente active ne durera que quelques microsecondes puisque les opérations sur les sémaphores sont très sommaires. En plus il n'y a pas de risque d'inversion de priorité, puisque les processus concurrents s'exécutent sur des processeurs différents, et se trouvent (de part de l'atomicité des opérations) dans un contexte où les interruptions sont masquées.

Un exemple d'utilisation des sémaphores est repris dans l'exemple p. 36. Le lecteur intéressé trouvera d'autres exercices classiques à la fin de cette section. Le problème présenté concerne celui du producteur et du consommateur, et conviendrait, par exemple, à la modélisation d'une communication

[17] Dans beaucoup d'ouvrages on réfère aux fonctions originales utilisées par Dijkstra `P()` et `V()` au lieu de `down()` et `up()`. Pour les néerlandophones, parmi les lecteurs, sachez que `P()` vient de *passeren* (passer) et `V()` de *vrijgeven* (libérer)

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

par pipe, entre autres.

Deux processus partagent une mémoire tampon de taille fixe : le premier, le *producteur*, y met des informations, tandis que l'autre, le *consommateur*, les lit. Les points délicats à régler, outre ceux de l'accès concurrent tel que l'on l'a vu pour le spoule d'impression p. 28, concernent la synchronisation des processus : que se passe-t-il lorsque la mémoire tampon est pleine et le producteur veut produire des informations, ou, au contraire, lorsque le mémoire est vide et le consommateur veut en retirer ?

La solution consiste à utiliser trois sémaphores : un pour l'exclusion mutuelle (`mutex`), et deux pour la synchronisation (`plein` et `vide`). `plein` contient le nombre d'emplacements de mémoire occupés (initialisé à 0) et `vide` le nombre d'emplacements inoccupés (initialisé à `N`). Lorsque le producteur veut ajouter un objet dans le tampon, il effectue un `down(&vide)`, se bloquant lorsqu'il n'y a plus d'espaces libres. Lorsqu'il a effectivement ajouté l'objet, il fait un `up(&plein)` pour signaler éventuellement au consommateur bloqué qu'une case s'est remplie. Le consommateur, lui, fait l'opération inverse : `down(&plein)` et `up(&vide)`.

```
*/ ATTENTION !! Ceci est du pseudo POSIX-C !
   Les fonctions down() et up() n'existent pas en tant que telles
   et les manipulations des sémaphores sont syntaxiquement
   légèrement plus verbeuses dans la réalité. Elles se basent sur
   les fonctions semget() et semop() et font intervenir les
   structures sembuf.

   L'équivalent de l'exemple ci-dessous, en code compilable et
   exécutable se trouve en Annexe B, p. 134.
*/
```

Solution avec des sémaphores pour le problème du producteur et du consommateur en `C`. semaphores.c

```
#define N 100        /* nombre d'emplacements */

typedef int semaphore ;        /* c'est plus propre */

/* Les sémaphores partagés par tous les processus */
semaphore mutex = 1 ;          /* controle zone critique */
semaphore vide = N ;           /* nb. emplacements libres */
semaphore plein = 0 ;          /* nb. emplacements utilisés */
```

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

```
void producteur(void)                                          10 producteur
{
  int objet ;       /* un objet de type arbitraire */

  while (TRUE) {
    produire_objet(&objet) ;
    down(&vide) ;   /* décrément des places libres */
    down(&mutex) ; /* section critique */
    mettre_objet(objet) ; /* utilisation ressource */
    up(&mutex) ;    /* fin section critique */
    up(&plein) ;    /* incrément des places occupées */            20
  }
}

void consommateur(void)                                        consommateur
{
  int objet ;       /* un objet de type arbitraire */

  while (TRUE) {
    down(&plein) ; /* décrément des places occupées */
    down(&mutex) ; /* section critique */                         30
    retirer_objet(&objet) ; /* utilisation ressource */
    up(&mutex) ;    /* fin section critique */
    up(&vide) ;     /* incrément des places vides */
  }
}
```

**Exercice :** Que se passe-t-il lorsque l'on intervertit les deux lignes

```
down(&vide) ;
down(&mutex) ;
```

dans la fonction `producteur()` ?

**Les compteurs d'évènements** sont sensiblement comparables aux sémaphores et ont été introduits par REED et KANODIA en 1979. Au lieu d'incrémenter ou de décrémenter des valeurs entières, les compteurs d'évènements ne font que les incrémenter, certaines primitives système permettant d'attendre qu'une valeur dépasse un certain seuil avant de se réveiller. Les primitives (atomiques) système, sont alors :

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

`read(e)` qui donne la valeur courante du compteur `e`,

`advance(&e)` qui incrémente la valeur courante du compteur `e`,

`await(e,v)` qui attend que la valeur du compteur `e` dépasse `v`.

**Exercice :** Refaites l'algorithme du producteur et du consommateur avec des compteurs d'évènements. En faut-il autant que des sémaphores ? Pourquoi ?

**Les moniteurs** sont une solution aux problèmes de synchronisation (HOARE, 1974 ; BRINCH HANSEN 1975) de plus haut niveau pour éviter les erreurs de programmation qui peuvent apparaître avec des sémaphores ou les compteurs d'évènements. L'idée d'un moniteur est d'encapsuler un ensemble de variables d'état et d'en fournir l'accès uniquement à travers quelques fonctions bien définies avec la restriction que seul un processus à la fois peut activer une fonction du moniteur. Principalement introduit pour éviter les pièges liés à l'ordre d'incrémentation et décrémentation des sémaphores et le risque de non respect du contrat de confiance entre les utilisateurs de ressources communes, les moniteurs sont donc des constructions de plus haut niveau, qui sont entièrement gérés par le compilateur lui-même, le programmeur se bornant simplement à indiquer les zones critiques.

**Exemple 1.3 : Utilisation de moniteurs en JAVA**

L'utilisation des moniteurs en JAVA se fait à travers le mot clé `synchronized`. Le moniteur garantit pour toute instance d'une classe (donc individuellement pour chaque objet instancié) qu'à un instant donné, seul un thread se trouve à exécuter du code dans l'ensemble des blocs marqués `synchronized`. Tout autre thread souhaitant exécuter un bloc synchronisé doit attendre que le thread en cours sorte de son bloc synchronisé, ou qu'il le libère temporairement en se mettant en attente d'une ressource par la commande `wait()`.

Tout bloc étiqueté comme `synchronized` doit généralement se terminer par un appel à `notify()` (resp. `notifyAll()`) pour signaler aux processus en attente de la ressource, donc ayant invoqué la commande `wait()` (resp. à tous les processus en attente d'une ressource) que celle-ci est libérée (même si ce n'est pas la ressource qui les concerne, dans le cas de `notifyAll()`). Attention ! Il convient de distinguer deux aspects de la synchronisation :

1. L'exclusion d'accès mutuel aux zones `synchronized`, qui est géré par le moniteur, et qui échappe à tout contrôle du programmeur.
2. Les couples `wait()` – `notify()` qui permettent d'affiner l'accès aux ressources critiques. Dans les cas où l'utilisation de `wait()` est superflue, il n'y a pas lieu de terminer le bloc par `notify()`.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

Dans cet exemple, on définit une classe qui compte et décompte des accès, en garantissant qu'on ne décompte pas plus que l'on n'incrémente (un sémaphore donc).

Sémaphore – JAVA Counter.java

```
public class Counter {

  // La valeur du compteur
  private int valeur = 0 ;

  // L'incrément (qui retourne la valeur avant incrémantation,
  // α l'instar de l'opérateur postfixe ++). Synchronisé, pour
  // éviter les accès concurrents.
  public synchronized int incr() {                                  incr
    int anciennevaleur = valeur ;                                   10
    valeur += 1 ;

    // libération de la ressource.
    notifyAll() ;
    return anciennevaleur ;
  }

  // Le décrément (qui retourne la valeur avant décrémentation,
  // α l'instar de l'opérateur postfixe –). Synchronisé, pour
  // éviter les accès concurrents. Se met en attente si la valeur      20
  // ne peut être décrémentée.
  public synchronized int decr() {                                  decr

    // Si la valeur ne peut être décrémentée, on libère la
    // ressource (via un appel α wait()) et on attend un signal
    // de libération de la ressource provenant d'un autre
    // thread, pour retester la valeur, et éventuellement se
    // réattribuer la ressource.
    while(valeur == 0)
      try {                                                         30
        wait() ;
      }
      catch(InterruptedException e) {}

    int anciennevaleur = valeur ;
```

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

```
    valeur -= 1 ;

    // libération de la ressource.
    notifyAll() ;
    return anciennevaleur ;                                    40
  }

}
```

**L'envoi de messages** est une approche nouvelle à la synchronisation et l'accès aux ressources partagées qui profite de l'apparition des systèmes distribués (et, en moindre mesure client/serveur). En effet, toutes les solutions présentées précédemment s'appliquent parfaitement à des situations de multiprogrammation sur des monoprocesseurs. Déjà, leur utilisation sur des ordinateurs multiprocesseurs demande une adaptation avec une commande `TSL` pour éviter des conflits. Dans un contexte réparti cette adaptation n'a plus aucun effet, et il convient de trouver d'autres solutions. Les modèles à base de messages permettent d'envoyer des requêtes d'accès, de blocage ou de libération de ressources à des « *fournisseurs de services* », indépendamment du fait que ceux-ci se trouvent sur la même machine ou éloignés et accessibles via un réseau. Nous en aborderons quelques cas dans la partie consacrée aux systèmes réparties (§ 2.4).

#### 1.3.6.4 Quelques exemples classiques

Afin de conclure cette partie sur la synchronisation de processus et le partage de ressources, nous proposons ici quelques problèmes devenus très classiques en guise d'exercice.

> **Exercice :** Résoudre les problèmes énoncés ci-dessous par intermédiaire d'un des systèmes décrits dans les paragraphes précédents.

Les textes sont repris à l'identique de [1].

### Le problème des philosophes – Dijkstra, 1965

Cinq philosophes sont assis autour d'une table. Chaque philosophe a devant lui un plat de spaghettis tellement glissants qu'il faut deux fourchettes pour pouvoir les manger. Une fourchette sépare deux assiettes consécutives (il y a donc autant de fourchettes que d'assiettes ou de philosophes).

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

Un philosophe passe son temps à penser et à manger. Lorsqu'un philosophe a faim, il tente de s'emparer des deux fourchettes qui sont de part et d'autre de son assiette, l'une après l'autre. L'ordre n'importe pas. S'il obtient les deux fourchettes, il mange pendant un certain temps, puis les repose, et se remet à penser.

La question est la suivante : comment écrire un programme dans lequel chaque philosophe est un processus indépendant (*i.e.* rien, mis à part la disponibilité des fourchettes et son propre état interne, n'impose à chaque philosophe quand il doit penser et/ou manger) et qui permette à chaque philosophe de se livrer à ces activités sans que le programme se bloque ?

### Le modèle des lecteurs et des rédacteurs – Courtois et *al.*, 1971

Imaginez une grande base de données, où plusieurs processus tentent de lire et d'écrire des informations. On peut accepter que plusieurs processus lisent en même temps dans la base. Mais si un processus est en train de modifier la base en y écrivant des données, aucun autre processus, pas même un lecteur, ne doit être autorisé à y accéder.

Il existe deux solutions : l'une donne priorité aux lecteurs (*i.e.* tant qu'il y a des lecteurs, aucun rédacteur n'a accès à la base), l'autre aux rédacteurs (*i.e.* une demande d'accès en écriture empêche de nouveaux lecteurs de lire la base).

### Le démon de requêtes réseau

Un cas plus proche de la réalité concerne un service de gestion de connexions réseau. Le système gère un pilote qui attend l'arrivée de données via une carte réseau, selon le type de données, elles sont ensuite transmises à des processus de gestion spécialisés. Chacun de ces processus spécialisés dispose de $n$ emplacements mémoire pour gérer des données éventuelles. En l'absence de données, le service ne fait rien, et ne surcharge pas le système. Lorsqu'une donnée arrive, le pilote de la carte réseau analyse les données, et les transmet au processus associé. Celui-ci se réveille, et les traite. Si entre temps d'autres données arrivent, les données sont empilées sur les emplacements libres, ou jetées si aucun emplacement ne reste disponible. Le processus spécialisé gère les données tant que des emplacements restent occupés, et se rendort lorsqu'il n'en reste plus.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

#### 1.3.6.5 Les signaux

Beaucoup de systèmes (notamment ceux respectant Posix) fournissent la possibilité aux processus de s'envoyer des *signaux*. Les signaux sont une implémentation *soft* des interruptions. Leur principe est le suivant. Un processus, s'il le souhaite, définit quels sont les signaux auxquels il veut réagir (c'est le système qui lui fournit la liste de signaux disponibles). Il associe une fonction de traitement à chaque signal, à l'aide des fonctions systèmes associés. Ensuite il s'exécute normalement. À tout moment de sont exécution, il peut être interrompu par un signal. À l'instar de ce qui se passe avec les interruptions, le processus cesse son flot d'exécution classique, et se déroute vers la fonction de traitement du signal, à l'issue de laquelle il reprend l'exécution là où il l'avait laissée.

**Exemple 1.4 : Les signaux Posix**

Dans cet exemple en `C` (les signaux n'existent pas en JAVA), on écrit un programme dont la fonction principale consiste à signaler au système d'exploitation qu'il est en mesure de traîter des signaux de type `SIGUSR1`, et que la fonction de traitement est `handler()`, puis à'effectuer une boucle infinie d'attente.

La fonction `handler()` se contente d'afficher l'heure de réception du signal.

À chaque réception de signal `SIGUSR1`, le programme interrompt sa boucle, exécute la fonction `handler()`, puis reprend l'exécution de la boucle.

Les signaux – `C` signaux.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>

#include <signal.h>

pid_t pid = 0;

void handler(int s)                                    handler
{                                                      10
  int heures, minutes, secondes;
  struct timeval t;

  gettimeofday(&t,0);
```

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

```
  heures = (t.tv_sec/60/60)%24 ;
  minutes = (t.tv_sec/60)%60 ;
  secondes = t.tv_sec%60 ;

  fprintf(stderr,"Le processus %d vient de recevoir un signal α %dh%d :%d\n",
          pid, heures, minutes, secondes);

  /* Selon les systèmes, il est nécessaire de réarmer la gestion des
     signaux α l'issue du handler */
  signal(SIGUSR1, handler);
}


int main()                                                          main
{
  /* appel système permettant de récuperer l'identifiant du processus */
  pid = getpid();

  fprintf(stderr, "Lancement du processus %d\n",pid);

  /* signaler au système que la fonction handler() gère les interruptions
     de type SIGUSR1 */
  signal(SIGUSR1, handler);

  for(;;) pause();

  exit(0);
}
```

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

### 1.3.7 Résumé

Dans cette partie, dédiée aux processus, nous avons évoqué un ensemble de services qu'un système d'exploitation multitâches doit pouvoir assurer :

1. fournir les fonctions de création et l'exécution des processus,
2. gérer le changement de contexte entre processus,
3. ordonnancer l'alternance (équitable, de préférence) entre les différents processus,
4. fournir des moyens de synchronisation ou d'exclusion d'accès aux ressources partagées.

## 1.4 La mémoire

Jusque là, nous n'avons considéré que des points liés à l'exécution de tâches ; leur mise en œuvre, leur ordonnancement, et la communication entre eux avec les problèmes de synchronisation et d'exclusion mutuelle que cela soulève. Nous ne sommes que très peu intéressés par leur présence en mémoire : leur chargement et leur manipulation par le système d'exploitation.

### 1.4.1 Le modèle de base

La représentation de base, pour un système monoprocesseur et monotâche, est montrée dans FIG. 1.12. En général, le système d'exploitation se trouve au niveau des premières adresses de la zone mémoire de la RAM. Pour des systèmes avec un OS embarqué (consoles de jeu, téléphones mobiles, *etc.*) le système se trouve souvent dans une partie non modifiable (ROM).

Afin de garantir de façon transparente mais flexible, l'amorçage d'un système quelconque, on retrouve souvent une combinaison des deux approches (RAM et ROM). La ROM contient alors un système minimal permettant de piloter les périphériques de base (clavier – disque – écran) et de charger le code d'amorçage à un endroit bien précis (d'un disque, p. ex.). Ce code est exécuté lors de la mise sous tension de l'ordinateur, et le « vrai » système d'exploitation, se trouvant dans la zone d'amorçage sur le disque, est ensuite chargé, prenant le relais du système minimal[18].

Ce schéma peut être étendu à un système d'exploitation multitâches, puisque c'est l'OS qui gère le chargement et le déchargement des processus

[18]Sur des architectures de type PC-Intel, le minisystème dans la ROM s'appelle BIOS (*Basic Input Output System*).

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

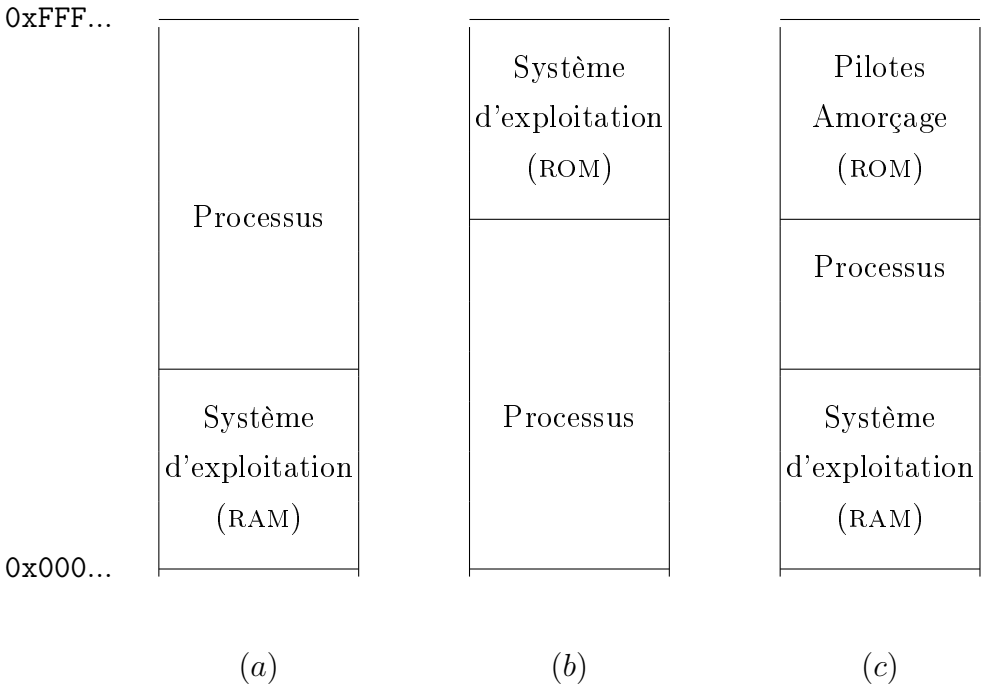


FIG. 1.12 – Les trois organisations de base pour un système monotâche.

en mémoire. La question à laquelle il convient à répondre est la suivante : « *Comment le système d'exploitation gère-t-il l'espace occupé par le(s) processus présent(s) en mémoire ?* ». Il faudra notamment savoir comment un processus est chargé et exécuté à un endroit précis, et comment peut-on être sûr que cet espace n'était pas déjà occupé par un autre processus.

### 1.4.2 Le chargement des processus

Sauf dans des cas très particuliers, que nous n'aborderons pas ici, on ne sait pas, *a priori*, à quel endroit de la mémoire un programme atterrira lorsqu'il sera exécuté. Jusqu'à présent (*cf.* FIG. 1.3) nous avons présenté l'image des processus comme une zone mémoire figée et structurée. Qu'en est-il en réalité ? Quelle est la relation entre un programme exécutable, compilé, stocké sur un disque, et cette image en mémoire ? Que se passe-t-il lorsque le système décide d'exécuter un programme stocké sur disque.

#### 1.4.2.1 Le code d'amorçage

Selon le principe de VON NEUMANN, l'image de données et d'instructions sont indissociables : c'est une suite de bits. Bien qu'un processus ait une structure identifiée (*cf.* FIG. 1.4), les différentes parties ont une taille

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

variable, et rien ne garantit que, dans la zone des instructions, la première ligne correspond effectivement à la première instruction à exécuter. Sans information supplémentaire, le système est donc incapable d'exploiter un fichier brut, puisqu'il lui manque des informations fondamentales. Chaque système exigera donc qu'au début de chaque fichier d'exécutable, figurent au moins (d'une façon ou d'une autre, et très dépendante du système en question, et selon une structure très codifiée) les informations nécessaires pour identifier la première ligne à exécuter. Ce sont ces informations que l'on appelle code d'amorçage. La partie du système d'exploitation qui s'occupe de récupérer le fichier du disque, de préparer les zones mémoires nécessaires et initialise l'état du processus ainsi crée avec les données d'amorçage, s'appelle le *chargeur* ou *loader*.

#### 1.4.2.2 La translation d'adresses

Reste l'incertitude de l'endroit de la mémoire où sera stocké le processus. C'est seulement au chargement que l'on sait quelle sera la zone mémoire (et donc, par conséquent, les adresses réelles des variables et des différents branchements `if` – `while` – appels de fonctions) qui contiendra les instructions. Il y a deux solutions au problème, selon le type de processeur qui est utilisé. S'il n'offre pas de services particuliers, le chargeur devra parcourir tout le code du programme et translater les adresses qu'il y trouve. Pour cela, on suppose que le programme a été compilé avec l'hypothèse de base que la zone commence à l'adresse `0x000`. Il suffit donc de rajouter à chaque adresse trouvé l'adresse de chargement du processus. Faut-il encore identifier les adresses ! Il est donc nécessaire que le chargeur analyse les instructions et identifie les arguments, ce qui constitue une surcharge significative. Une solution plus élégante (mais qui nécessite que le processeur dispose du jeu d'instructions adéquat) est de faire faire le travail au processeur, en utilisant le registre d'adresse de base `%cs`, et de lui signaler de systématiquement translater tout argument de type adresse par la valeur stockée dans `%cs`. Ce travail étant intégré dans les circuits du processeur, cela crée une surcharge négligeable.

### 1.4.3 La multiprogrammation avec partitions fixes

Revenons maintenant au problème de la sélection de la zone mémoire qui recevra le processus. Une première solution consiste à *partitionner* la mémoire en $n$ parties de tailles fixes (pas nécessairement identiques). Ce partitionnement est fait une fois pour toutes, à chaque démarrage du système (p. ex. MFT sur le système OS/360 d'IBM).

Chaque nouvelle tâche est ensuite placée dans une file d'attente, et chargée

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

dans la plus petite partition qui peut la contenir lorsque celle-ci se libère. Toute mémoire non utilisée dans cette partition est perdue pour d'éventuelles autres processus dans d'autres partitions.

Il existe un certain nombre de variantes sur ce schéma : on peut dès le lancement d'un processus, déterminer la partition qu'il va occuper, et gérer une file d'attente par partition ; ou, au contraire, choisir de n'avoir qu'une seule liste d'attente, et d'appliquer des stratégies plus ou moins complexes pour attribuer au mieux une partition libre à un processus en attente (premier dans la liste, plus grand pouvant être contenu dans la partition, plus grand avec interdiction de non sélection $k$ fois, ...).

#### 1.4.3.1 Inconvénients

Cette approche a l'avantage d'être très simple dans sa mise en œuvre, et de nécessiter très peu d'interventions de la part du système. Le chargement de processus et le changement de contexte s'en trouvent donc efficacement et rapidement gérés. C'est une approche parfaitement adaptée au traitement de tâches par lots. Par contre il ne convient pas à des environnements interactifs, et cela pour les raisons suivantes :

Les processus interactifs sont souvent de petite taille. Si on les loge dans des petites partitions, on ne gaspille pas de mémoire, et on peut se garder soit de la place pour de plus grandes partitions, pour lancer des processus plus grands, soit de la place pour d'autres petites partitions, et avoir plusieurs processus interactifs en même temps.

Dans le premier cas, tous les petits processus en attente doivent être stockés sur le disque (*cf.* le va-et-vient, p. 48). Chaque changement de contexte provoquera donc une copie du processus actif sur le disque et le chargement du premier processus en attente. Ce qui nuit terriblement à l'interactivité et l'efficacité. Ceci sera d'autant plus le cas que le nombre de petites partitions est limité.

Dans le second cas, les changement de contexte pour les petits processus sera probablement moins pénalisant, puisque beaucoup d'entre eux peuvent rester en mémoire. Par contre, une application volumineuse a toute chance de se faire refuser l'exécution par faute de partition de taille suffisante disponible.

On est donc constamment limité par le nombre de partitions et la taille de la partition la plus grande, qui est nécessairement nettement inférieure à la taille totale de la mémoire disponible.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

### 1.4.4 La multiprogrammation avec partitions variables

La solution la plus flexible est d'adopter un système avec des partitions de taille variable et un système de va-et-vient qui permet d'utiliser le disque comme mémoire secondaire et d'y stocker un nombre de processus inactifs ou en attente.

#### 1.4.4.1 Le va-et-vient

Conceptuellement le va-et-vient, ou *swap*, se comporte exactement comme la mémoire vive, à la différence près qu'on ne peut y exécuter des processus (pour exécuter un processus sur le *swap*, il faut le charger en mémoire vive), ainsi que quelques considérations liées au médium de stockage, qui impose un accès et un stockage par blocs, mais que l'on peut négliger.

Un processus qui est inactif (soit bloqué, soit préempté) peut donc être placé dans le swap. Les stratégies de choix des processus à écrire sur disque sont sensiblement identiques à celles mises en œuvre pour l'ordonnancement des processus. La gestion du swap en tant que « zone mémoire » est également comparable à ce qu'on verra dans la section suivante. Il est à noter qu'il existe deux types de solution : soit on alloue une place fixe dans le swap, pour chaque processus créé, soit on utilise le swap comme une grande zone de stockage dans laquelle les processus sont écrits selon les besoins et à un endroit déterminé en fonction de l'occupation au moment de l'écriture (à l'instar des méthodes de partitionnement de la mémoire fixe ou variable).

#### 1.4.4.2 Principe des partitions variables

L'idée principale consiste à allouer une zone mémoire à un processus selon ses besoins. La représentation FIG. 1.13 montre l'état de la mémoire dans un cas de figure ou 3 processus (A, B et C) se lancent consécutivement. Ensuite A se termine, et un autre processus est lancé (D). Les zones grisées représentent la mémoire inoccupée. À l'observation de cette représentation, plusieurs questions importantes se soulèvent :

- Comment gérer la mémoire libre ? Puisqu'il n'y a aucun contrôle sur l'ordre de lancement/terminaison ou sur la taille des processus, on conçoit facilement que la mémoire puisse rapidement devenir *fragmentée*.
- Comment détermine-t-on l'emplacement d'un processus, et que se passe-t-il s'il n'y a pas assez de mémoire disponible ?
- Comment détermine-t-on la taille d'un processus ? Est-ce que cette taille doit être fixe tout au long de son exécution ?

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

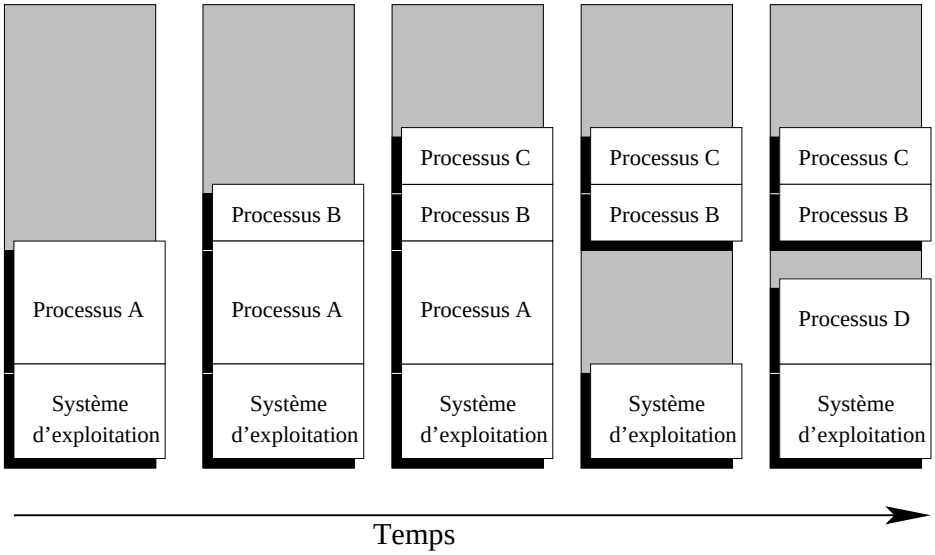


FIG. 1.13 – Principe d'allocation de partitions de taille variable.

#### 1.4.4.3 Gestion de la mémoire libre

Le système doit, à chaque instant, savoir quelles sont les zones de mémoire disponibles, afin de pouvoir décider où loger les nouveaux processus. Il existe plusieurs méthodes, que nous nous contentons d'énumérer simplement. Chacune d'entre elles demande ses propres mises en œuvre algorithmiques particulières et ses structures de données adéquates.

La solution la plus facile à mettre en œuvre, consiste à diviser la mémoire en petites cellules de taille fixe, allant de quelques mots mémoire à quelques kilo-octets. À chaque cellule, le système fait correspondre un bit dans un tableau interne, qu'il met à 1 si la cellule est occupée, ou à 0 si elle est libre. Cette approche présente l'inconvénient d'être relativement gourmande en espace mémoire, si on veut contrôler assez finement l'utilisation de la mémoire, et en plus, elle oblige le système à parcourir une partie non négligeable du tableau, à chaque fois qu'il doit trouver un espace libre d'une taille donnée, ce qui induit un surcoût d'exécution à chaque chargement en mémoire d'un processus.

Une autre solution consiste à gérer une liste chaînée d'emplacements libres, avec comme information, à chaque nœud de liste, l'adresse du premier mot mémoire libre et la taille de la zone. L'insertion d'un processus consiste à mettre à jour la taille et la première adresse du nœud libre qu'il va occuper. À la fin du processus, il suffit de mettre à jour (en l'agrandissant) le nœud libre adjacent s'il existe (et éventuellement de fusionner les deux nœud

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

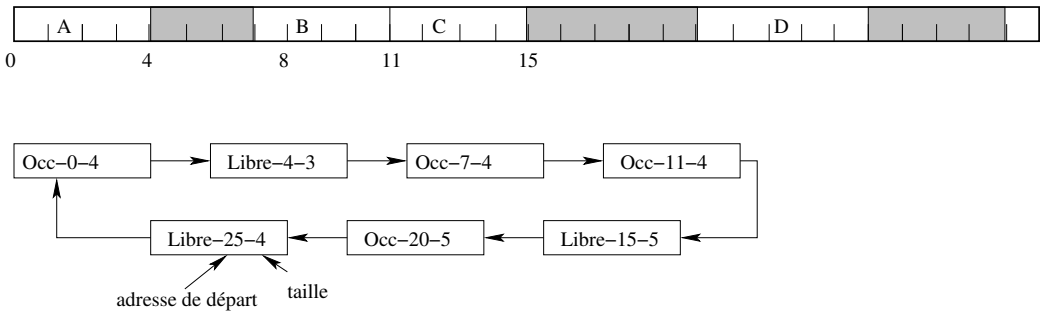


FIG. 1.14 – Gestion de la mémoire par liste chaînée.

adjacents, s'il y a lieu) ou de créer un nœud dans la liste s'il n'existe pas de zones adjacentes libres. Une variante, un peu plus flexible consiste à stocker dans la liste chaînée aussi bien les zones occupées que les zones libres (comme représenté dans FIG. 1.14). Dans ce cas-là, les seules opérations à effectuer consistent à scinder une case libre en deux, ou de fusionner des cases libres et occupées entre elles, suivant les cas.

Dans ce modèle, il reste néanmoins relativement complexe de déterminer quelle zone mémoire choisir pour loger le prochain processus. Doit-on prendre la première zone libre (et de taille suffisante), la plus grande, la plus petite ... ?

Il existe d'autres approches de gestion de mémoire, basées sur un découpage récursif, sous forme d'arbre binaire, par exemple, pour laquelle toute une série de variantes à été développée. Sachez juste qu'elles existent, et que de façon générale, la gestion de la mémoire n'est pas seulement un problème de bas niveau, mais qu'il existe un nombre de problèmes algorithmiques intéressants à résoudre (même pour les approches ci-dessus avec *bitmap* ou liste chaînée) afin d'obtenir une méthode de gestion efficace. Comme ces opérations sont susceptibles d'être effectuées très souvent, leur efficacité est primordiale pour limiter la surcharge induite par le système d'exploitation.

#### 1.4.4.4 Taille d'un processus

Un point clé, jusqu'à maintenant passé sous silence concerne la taille d'un processus. Afin de pouvoir le charger en mémoire, le système doit savoir de combien d'espace mémoire un processus doit pouvoir disposer (ne serait-ce que pour savoir s'il reste assez de mémoire disponible). Ensuite, une fois chargé, est-ce qu'un processus garde une taille constante, ou est-il autorisé à s'agrandir, et réclamer plus de mémoire qu'initialement prévu ?

Revenons sur la description de la représentation en mémoire d'un processus (p. 8). On y identifiait trois zones principales : les données statiques,

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

les instructions et la pile. Par construction même, les deux premières parties ont une taille fixe[19]. Restent donc la pile et le tas (une quatrième partie dont nous n'avons que brièvement évoqué l'existence).

La pile contient toutes les variables intermédiaires qui apparaissent et disparaissent au gré de l'exécution du programme. Principalement on y stocke : les variables locales d'une fonction, les paramètres d'appel d'une fonction, des résultats de calcul intermédiaire. Plus la pile est grande, plus la profondeur d'appel des fonctions (*i.e.* le nombre d'appels de fonction imbriqués) pourra être grande. Selon le flot d'exécution d'un même programme (dépendant des données d'entrée qui sont fournies) la profondeur d'appel peut varier. Par exemple, calculer $n!$ de façon récursive sollicitera d'autant plus la pile que $n$ est grand. Il est donc impossible de savoir, *a priori*, quelle est la taille qu'il faut allouer à la pile.

Le tas contient toutes les données qui sont dynamiquement allouées par le processus (par des appels `malloc()` en `C` ou `new` en JAVA et `C++`, p. ex.). Sa taille dépend uniquement des paramètres d'exécution et des données d'entrée fournies. Là encore, il est impossible de connaître les besoins en termes de mémoire *a priori*.

La solution consiste à fixer arbitrairement une taille pour la pile et le tas, basée sur des estimations raisonnables, et l'utilisation habituelle du processus en question. Lorsqu'un processus a utilisé toutes ses ressources, on peut envisager deux solutions. Soit le système met fin au processus, en lui annonçant qu'il ne dispose plus de ressources. Soit le système reloge le processus dans une nouvelle zone de la mémoire, en lui réservant plus de ressources.

La réallocation de mémoire pose un certain nombre de problèmes, car les données stockées dans le tas sont référencées directement par leur adresse en mémoire. On se heurte donc au même problème de translation des adresses comme dans le cas du chargement des processus (p. 45). De ce fait, la réallocation des ressources ne doit pas modifier l'emplacement des objets dans le tas par rapport à l'adresse de base `%cs`. Le déplacement de la pile pose généralement moins de problèmes, puisqu'on y accède uniquement via le pointeur de pile `%esp`.

La solution généralement adoptée est représentée FIG. 1.15. On stocke dans la même zone mémoire et la pile et le tas. En général, la pile croît du bas vers le haut, tandis que le tas grandit du haut vers le bas. Lorsque les

[19] Des programmes qui modifiaient leur propre jeu d'instructions ont existé, et existent probablement encore dans certains laboratoires de recherche. Le consensus actuel est de considérer que la réalisation (d'un point de vue système d'exploitation) de ce genre de processus automodifiants est trop complexe pour avoir une utilité dans l'immédiat. Les systèmes d'exploitation imposent donc que la partie correspondant aux instructions soit non-modifiable.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

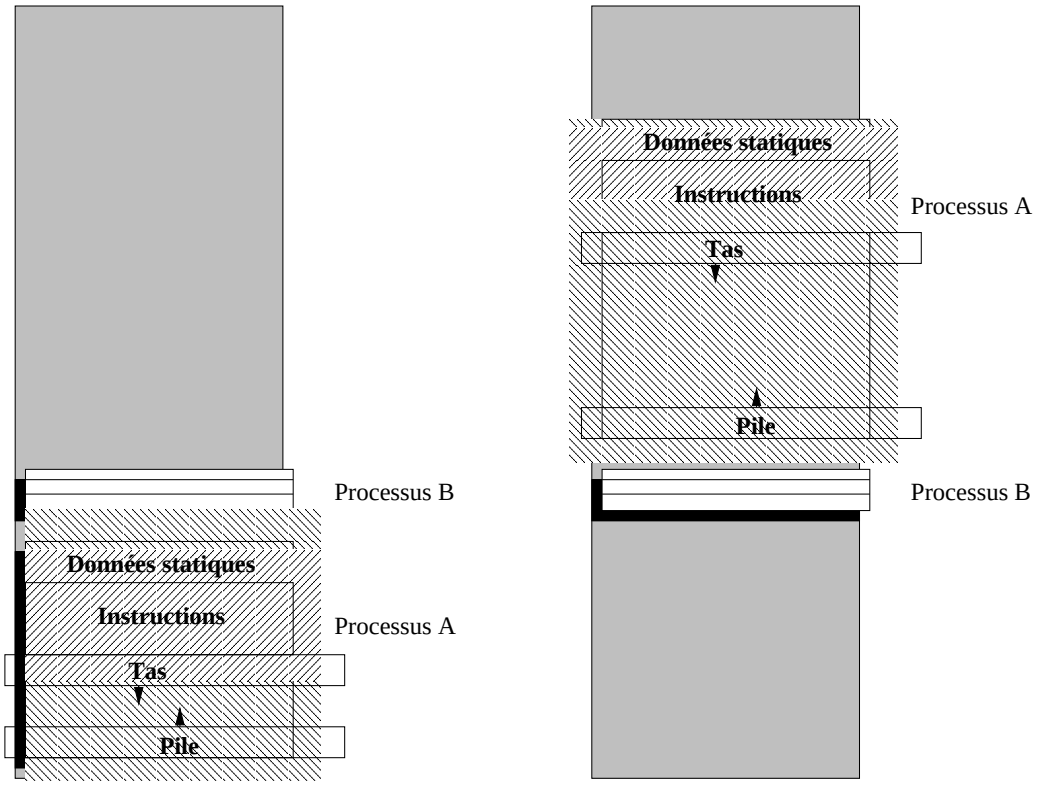


FIG. 1.15 – Réallocation de processus avec augmentation de la zone tas–pile.

deux parties se rencontrent on réalloue une nouvelle zone pour le processus. Les objets présents dans le tas restent à la même position relative par rapport à l'adresse de base ; il suffit de mettre le pointeur de pile à jour, et de recopier les données de la pile dans son nouvel emplacement.

#### 1.4.4.5 Limites

On peut relever deux limites majeures de cette approche. Premièrement, le fait que le processus doit résider dans sa totalité en mémoire. Les besoins des grosses applications sont maintenant telles, qu'il est facilement concevable qu'un logiciel contienne plus d'instructions et de données que la mémoire disponible. Pourtant, il n'exécute qu'une instruction à la fois, donc il n'y a pas de raison de ne pas pouvoir l'exécuter. Deuxièmement, la solution de réallocation de ressources n'est pas vraiment satisfaisante. L'organisation du tas, le besoin de recopier tout le programme pour seulement agrandir la place pour quelques structures de données bien précises, et le manque de protection efficace des zones mémoire, donnent une impression de bricolage.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

Le premier point sera résolu par la notion de mémoire virtuelle, le second par son extension : la segmentation.

### 1.4.5 La mémoire virtuelle

La mémoire virtuelle a pour but principal de pouvoir exécuter des processus sans qu'ils soient logés en mémoire en leur totalité. Tout particulièrement dans les sections concernant la mémoire virtuelle et la segmentation, on aborde des problèmes très techniques qui touchent toutes les couches du système d'exploitation (et même certaines applications hors OS, comme les compilateurs, pour la partie segmentation) jusqu'au matériel (MMU, cache, *etc.*). Nous ne nous attarderons pas sur les aspects techniques, et nous nous contenterons simplement de donner les principes de fonctionnement généraux.

#### 1.4.5.1 La pagination

Au niveau du CPU, une adresse mémoire est généralement égale à un mot machine. Les mots machines des stations de travail actuelles sont soit de 32 bits, soit de 64 bits. Cela veut dire qu'une adresse peut prendre, dans le premier cas $2^{32}$ valeurs différentes, et dans le second cas $2^{64}$. En d'autres termes, si on suppose que chaque adresse représente un octet (ce qui est généralement le cas), la taille de la mémoire totale que l'on peut adresser est de $2^{32}$ octets dans le premier cas (4 Giga-octets) et $2^{64}$ octets dans le second cas (64 Exa-octets = 16,8 millions de Tera-octets). On appellera l'espace ainsi adressable *mémoire virtuelle*. Il est rare, voire impossible pour les architectures 64 bits, de disposer de cette quantité de mémoire vive. Par contre, il est moins rare d'avoir un espace disque atteignant (et même dépassant, dans le cas d'une architecture 32 bits) la taille de la mémoire virtuelle. En considérant l'ensemble de l'espace de stockage réel disponible (swap + RAM) il est envisageable de vouloir exécuter des applications qui ne tiennent pas entièrement dans la RAM, d'autant plus que celle-ci peut être partagée entre différents processus, réduisant ainsi la place disponible.

L'idée de base est donc de dire que le système dispose d'un *espace d'adressage virtuel*. C'est l'espace potentiellement couvert par un pointeur et que l'on a appelé *mémoire virtuelle* ci-dessus. On « découpe » cet espace en *pages* logiques (d'où le terme de *pagination*). On prendra dans la suite de ce document l'hypothèse que les pages ont une taille de 4Ko (4 Kilo-octets, ou $2^{12}$ octets). Ainsi, le système dispose d'un espace d'adressage virtuel de $n$ pages.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

**Exemple :** sur une architecture de 32 bits, on utilise une taille de page de 4Ko. L'espace d'adressage de 4Go ($2^{32}$) est ainsi partagée en 1 048 576 ($2^{20}$) pages de 4Ko ($2^{12}$) chacune. ($2^{20} \times 2^{12} = 2^{32}$)

Avec ce système, on peut donner une nouvelle interprétation à une adresse mémoire. Au lieu de considérer qu'une adresse représente un indice absolu de 32 bits, on peut le séparer en un numéro de page de 20 bits, et un déplacement (*offset* en anglais), à l'intérieur de cette page, de 12 bits.

$$\text{Adresse} = \underbrace{0000\,0110\,0101}_{\text{page } 81}\ \underbrace{0100\,1101\,0100\,1000\,1001}_{\text{offset } 316553}$$

Quel est le but de cette manœuvre ? Bien, si on découpe la mémoire réelle de la même façon, c'est-à-dire en pages de la même taille, on peut charger en mémoire réelle les pages effectivement utilisées au fur et à mesure des besoins, et les enlever lorsqu'on n'en a plus besoin. La condition nécessaire, est de pouvoir disposer d'un moyen de savoir quelles pages de la mémoire virtuelle se trouvent actuellement en mémoire et où elle se trouvent. C'est le rôle de l'unité de gestion de la mémoire du système, ou le MMU (*Memory Management Unit*), comme représenté dans FIG. 1.16.

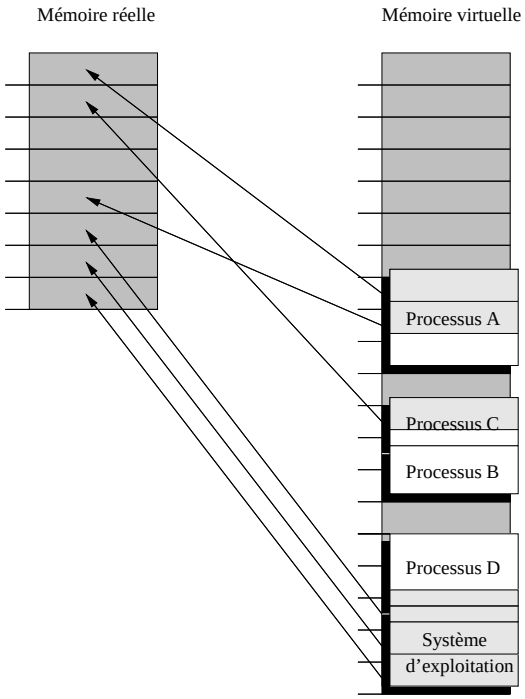


FIG. 1.16 – Correspondance entre mémoire virtuelle et mémoire réelle.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

FIG. 1.16 représente, à droite, la mémoire virtuelle, vue par le système d'exploitation. Avec les zones occupées par des processus, et des zones libres, et, à gauche, la mémoire réelle (qui est plus petite), ainsi que les correspondances entre différentes pages réelles et virtuelles.

Chaque fois que le processeur manipule une adresse mémoire (qui est une adresse virtuelle), l'instruction est interceptée par le MMU, qui fait partie intégrante du processeur. Au lieu d'accéder à la mémoire réelle directement à l'adresse spécifiée, le MMU va traduire l'adresse virtuelle en son correspondant réel avant de placer la requête effectivement sur le bus. Pour cela, le MMU doit satisfaire un certain nombre de conditions :

- Il doit savoir quelles pages virtuelles sont actuellement chargées en mémoire et où elles se trouvent. (*cf.* les parties grisées dans FIG. 1.16)
- Il doit pouvoir communiquer avec le système d'exploitation pour le prévenir si ce dernier veut accéder à une page qui n'est pas chargée en mémoire.
- Il doit exister des moyens pour charger des pages virtuelles en mémoire, ou encore pour libérer des pages mémoire lorsqu'elles ne sont plus utiles.

En fait, le MMU dispose d'une table de correspondance. Supposons pour l'instant qu'il dispose, dans cette table, d'un registre par page virtuelle. Ce registre est constitué de deux parties : un bit de présence, et un numéro de page. Lorsque le processeur veut accéder à une adresse virtuelle, le MMU va décomposer l'adresse virtuelle en un numéro de page $p$ et un offset $o$. Ensuite, il regarde dans sa table de correspondance, à l'indice $p$.

Si le bit de présence vaut 1, cela signifie que la page est actuellement en mémoire. Il lit donc le numéro de la page réelle $r$ correspondant, et il remplace l'adresse virtuelle ($p$, $o$) en une adresse réelle ($r$, $o$) par simple substitution de la partie page.

Si le bit de présence vaut 0, il provoque un *défaut de page* par l'intermédiaire d'une interruption. Le traitement de cette interruption permet de charger la page en mémoire (éventuellement en éliminant une autre page de la mémoire), de remettre à jour la table de correspondance du MMU, et ensuite de continuer le processus qui a provoqué le défaut de page.

Le seul détail que nous avons passé sous silence est la taille du tableau de correspondance. En effet, dans l'exemple (tout à fait réaliste) que nous avons développé, la table devrait comporter dans les 1 048 576 entrées. Il est irréaliste de considérer qu'un processeur puisse disposer de ce nombre de registres. En réalité, on utilise une table à plusieurs niveaux, qui, elle, réside en mémoire vive, et dont les registres du MMU sont des points d'entrée. L'inconvénient majeur est que de cette façon, l'efficacité du principe de la mémoire virtuelle est largement réduite, puisque, pour connaître la position

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

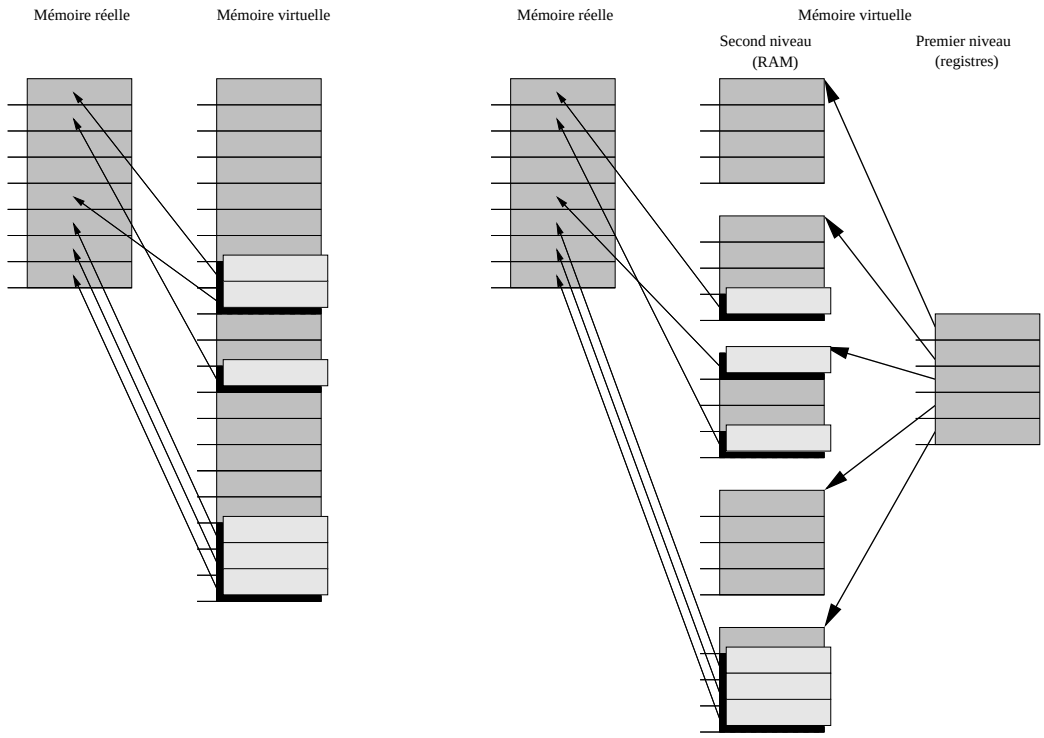


FIG. 1.17 – Mémoire virtuelle à un niveau (à gauche) : les registres du MMU permettent de faire la correspondance directement. Mémoire virtuelle à deux niveaux (à droite) : les registres du MMU contiennent des entrées sur des tableaux stockés en RAM.

exacte d'une adresse mémoire, il y a besoin d'accéder à la table de correspondance qui se trouve elle même en mémoire, et non plus dans une table de registres. Ceci implique des occupations du bus et des accès mémoire, ralentissant considérablement chaque opération d'accès mémoire virtuelle.

#### 1.4.5.2 Cache de pagination

Pour être souple et efficace, la taille des pages doit rester relativement petite. Par contre, cela impose que la taille de la table de correspondance soit grande, et donc que finalement on pénalise trop l'accès à la mémoire (chaque consultation d'une adresse virtuelle est ralentie par un facteur 2 ou 3 par rapport à un système non paginé).

On constate, en outre, que l'on n'utilise finalement qu'un petit nombre de pages entre lesquelles on bascule fréquemment, et que lorsqu'une page est déchargée de la mémoire, on n'a rarement besoin de la recharger dans l'immédiat.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

La solution la plus efficace est donc d'introduire une antémémoire rapide (*cache*) qui mémorise toutes les pages virtuelles récemment accédées ainsi que leur correspondant réel. Ainsi lors de l'accès à une adresse virtuelle, on essayera d'abord de la traduire par l'intermédiaire de cette antémémoire (c'est fait par des circuits intégrés, et en parallèle, donc sans perte de temps). Si la page était présente on ne sollicite pas les tables (plus lentes) du MMU, et on fait la traduction immédiatement. Sinon, on fait une recherche dans les tables du MMU, et on met à jour l'antémémoire pour d'éventuelles références futures. Cette mise à jour suppose que l'on éjecte une page qui était déjà présente dans le cache. Les algorithmes de remplacement de cache s'apparentent à ceux qui gèrent le remplacement des pages en général.

#### 1.4.5.3 Gestion des défauts de page

En général, la provocation d'un *défaut de page* suscite le remplacement d'une page existante par une nouvelle page, s'il n'y a plus de place de libre en mémoire. Dans ce cas, il est nécessaire de prendre une décision quant à quelle page enlever, ce qui n'est pas nécessairement un choix facile. De plus, les défauts de page sont systématiquement provoqués par un processus en cours d'exécution, et il n'est peut-être pas souhaitable de prendre du temps précieux à la prise de cette décision. Il serait donc préférable qu'il y ait systématiquement quelques pages de libre en mémoire.

C'est pour cette raison que les systèmes disposent, en général, d'un démon de pagination. Le démon veille à ce qu'il reste suffisamment de pages libres, et se réveille régulièrement pour enlever les pages devenues obsolètes. Les algorithmes mis en œuvre se servent en général du dernier instant de lecture ou d'écriture dans la page, pour décider, laquelle est la moins fréquemment utilisée. Il faut savoir qu'il existe de fortes présomptions (anomalie de Brady) pour supposer qu'il n'existe pas de critère optimal de remplacement de page. Les systèmes utilisent donc un certain nombre d'heuristiques pour décider si une page doit être enlevée ou non.

Reste à noter qu'il existe beaucoup de problèmes techniques vicieux à résoudre. Par exemple, si un processus est en mode bloqué, en attente d'une interruption d'entrée/sortie, on aurait tendance à vouloir enlever sa page de la mémoire. Or si ledit processus a demandé une écriture en DMA[20] à un périphérique, ce dernier écrira dans la page mémoire sans passer par le système. Si ce dernier a remplacé le contenu de cette page en attendant, on peut avoir des cas sévères de corruption de données.

[20] *Direct Memory Access*, *cf.* remarque p.1.1

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

### 1.4.6 La segmentation

Un certain nombre de points ont été occultés lors de la présentation des modèles de mémoire précédents.

- L'un concerne la sécurité des différentes zones de mémoire. Comment le système peut-il s'assurer qu'un processus utilise à bon escient son espace mémoire alloué, et notamment, comment peut-il s'assurer que certaines zones de mémoire restent correctement protégées en lecture, écriture et/ou exécution ?
  Par exemple, certaines ressources du système peuvent être accessibles en lecture seule, ou encore peut-on vouloir éviter que certaines parties de la mémoire correspondant à des instructions, soient écrasées par des données, ou encore que des données soient interprétées comme des instructions[21].
- Un autre point concerne la gestion dynamique de la mémoire. Certains types de programmes gèrent de grandes quantités de mémoire dont on ne peut prévoir à l'avance l'évolution (et surtout pas lors de l'écriture du programme). Prévoir une trop grande quantité de mémoire peut être pénalisante pour les performances générales du système, en prévoir une trop petite quantité peut limiter l'utilisation du programme. Comme nous l'avons déjà évoqué à la page p. 52 et la FIG. 1.15, le schéma de base code + tas/pile, stockés dans un même espace de processus n'est pas très flexible, surtout qu'à l'intérieur du tas, on peut avoir besoin de réallouer des grandes zones devenues trop exiguës.
- Un dernier point concerne celui des bibliothèques chargées dynamiquement lors de l'exécution du code. De plus en plus, les systèmes d'exploitation modernes s'appuient sur des bibliothèques dynamiques pour rendre les logiciels moins dépendants des différentes versions des services fournis par le système[22]. Ces bibliothèques ont une interface d'appel bien définie, mais ne sont pas incluses dans le logiciel qui les utilise. C'est lors de leur utilisation à l'exécution, que le système les charge en mémoire et les exécute. Ceci permet de les mettre à jour sans que le logiciel appelant ne doive être recompilé, et permet également d'avoir des fichiers moins grands pour le code binaire de ces logiciels. La difficulté de mise en œuvre consiste à noter que ces bibliothèques ne savent pas à l'avance à quel *offset* (*cf.* p. 54) ils se trouveront dans l'espace du processus, ce qui complique considérablement l'édition de

[21] C'est ce qui se passe lorsque Unix signale une « *Segmentation Fault* » lors de l'exécution de certains programmes contenant du code erroné.

[22] Ces bibliothèques dynamiques sont les fameux `.dll` sous Windows, pour *Dynamically Linked Library*, et `.so` sous les Unix, pour *Shared Object*.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

liens à la volée que nécessite ce chargement de bibliothèques.

#### 1.4.6.1 Principe

La solution adoptée par la segmentation de la mémoire, consiste à ne plus considérer la mémoire comme un tableau monodimensionnel linéaire, mais plutôt comme un espace bidimensionnel, ou encore comme un espace de segments de mémoire indépendants. Au lieu de disposer d'une zone de mémoire linéaire, un processus dispose maintenant d'un ensemble de segments, chaque segment correspondant un espace de mémoire linéaire classique. Une adresse est alors interprété comme un segment et un décalage dans le segment. Un Intel Pentium©, par exemple, considère les 14 premiers bits d'une adresse de 32 bits comme étant le numéro du segment :

$$\text{Adresse} = \underbrace{0000\,0110\,0101\,01}_{\text{segment } 405}\underbrace{00\,1101\,0100\,1000\,1001}_{\text{offset } 54409}$$

La différence de fond avec la pagination est qu'un processus peut contenir un nombre illimité (pour peu que le terme *illimité* puisse avoir un sens) de segments, et que chaque segment correspond à une partie bien précise : une pile, une fonction/procédure, un tableau, *etc.* De plus, contrairement aux pages, un segment n'a pas une taille fixe, mais variable. Au lieu de voir l'image mémoire d'un processus comme un bloc contigu d'instructions et de données, découpé en tranches de taille constante, comme dans le cas d'une mémoire paginée, un processus devient un ensemble de segments de taille variable et indépendants, comme le montre la FIG. 1.18.

#### 1.4.6.2 Avantages

1. Le premier point est le découpage en parties *logiques* plutôt qu'en tranches arbitraires[23]. Ceci confère deux avantages importants. Puisque chaque entité logique commence au début de son segment, son *offset* est toujours `0x0000`. Un segment peut donc très facilement être déplacé en mémoire, sans que cela affecte la structure du programme (à condition d'avoir, à l'instar des mémoires paginées, une table qui garde une trace des segments et leur placement effectif en mémoire ; ce qui va de soi). La réallocation de mémoire à la volée ne pose plus aucun problème, puisqu'il suffit de déplacer le segment à un endroit en mémoire qui lui

[23] Il est à noter que la FIG. 1.18 est inexacte et qu'en général un segment est plus grand qu'une page

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

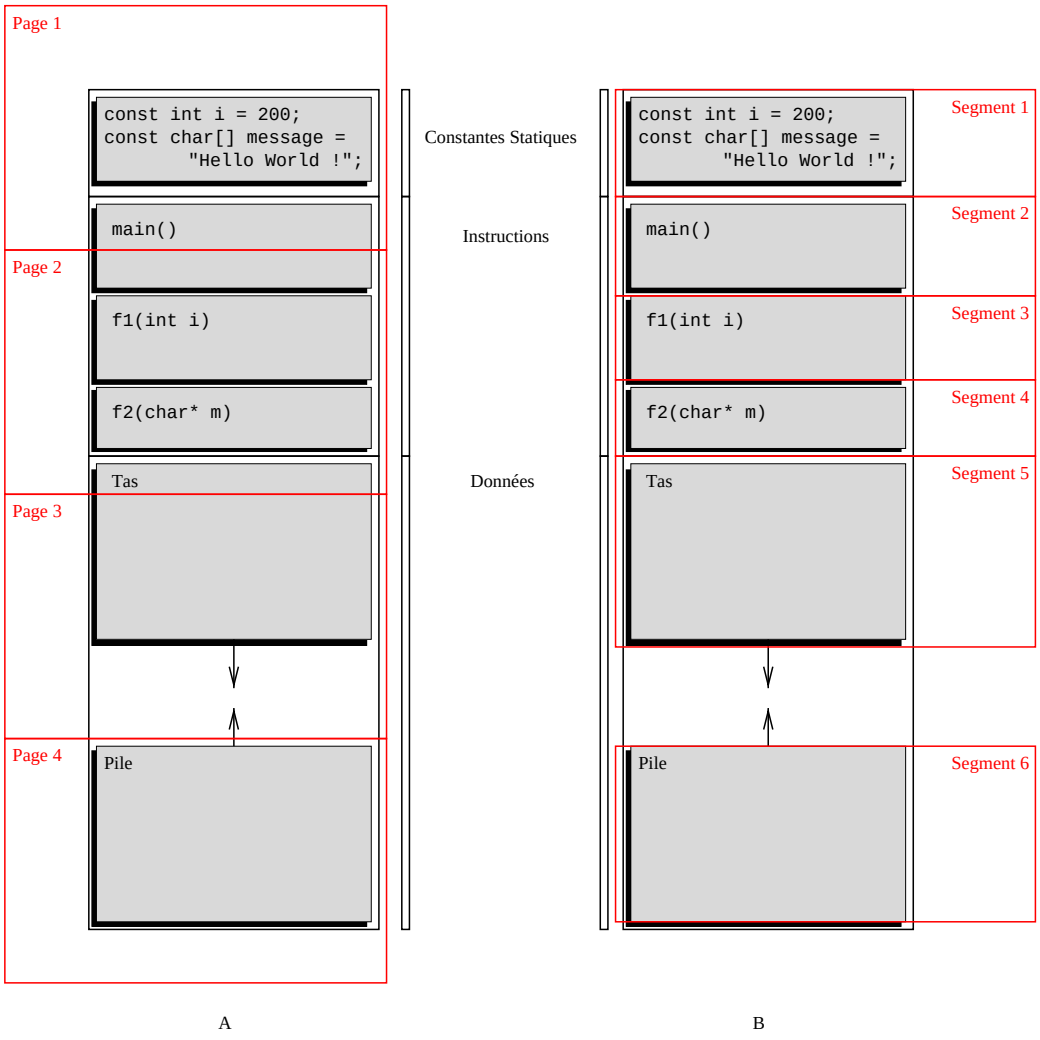


FIG. 1.18 – Structuration d'une image mémoire d'un processus. Différence entre mémoire paginée (a) et mémoire segmentée (b).

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

permet d'avoir plus de place. En plus, l'édition de liens est grandement facilitée, puisque l'appel à une fonction revient à faire un branchement au début d'un segment donné, sans qu'on ait à se préoccuper d'où se situe ce segment.

2. Étant donné qu'un segment représente une entité *logique* : donnée, code, constante, ... ; il est très facile d'imaginer une façon de protéger ces données. En associant à chaque segment un drapeau de type *lecture–écriture–exécution* on peut prévenir un certain nombre d'erreurs ou de corruptions de mémoire, en vérifiant, à chaque opération sur la mémoire que celle-ci est bien conforme aux droits d'accès spécifiés.
3. Les bibliothèques partagées, ou les programmes multi-entrants[24] n'ont plus besoin d'être stockés plusieurs fois en mémoire si plusieurs processus les utilisent. Il suffit de partager entre eux les mêmes segments.

#### 1.4.6.3 Inconvénients

Le principal inconvénient provient du fait que le système ne peut pas savoir, *a priori* quels sont les segments qu'utilise un processus, quelle est leur taille, la protection qui leur est affectée, *etc.* C'est donc l'exécutable qui doit informer le système de sa structure interne au moment du chargement. Ceci est fait par un entête de description (*cf.* le format ELF[25] sous Linux et d'autres Unices, ou le format PE[26] sous les Windows postérieurs à 3.11) qui décrit la structure du binaire.

**Exercice :** Sous Linux, la commande `readelf` permet d'afficher l'information concernant la structure d'un binaire ELF. Essayez la commande suivante :
```
> readelf --segments /bin/more
```

Par conséquent, c'est donc au programmeur qu'incombe la tâche de gérer la manipulation des segments et de décider quelle partie tombe dans quel segment, *etc.* Toutefois, les compilateurs se chargent de cette besogne et l'utilisation des segments est transparente pour l'utilisateur lambda.

#### 1.4.6.4 Mise en œuvre

Nous ne traiterons pas de la mise en œuvre de la gestion des segments dans ce cours. Sachez que les segments sont souvent plus grands qu'une page et

[24] Du code *multi-entrant* est du code qui a vocation à être exécuté en parallèle par plusieurs processus (*cf. threads* ou la commande `fork()`)
[25] ELF : Executable and Linking Format
[26] PE : Portable Executable Format

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

que les principes de mémoire paginée et segmentée ne sont pas incompatibles. On peut donc aboutir à des situations où, au final, une adresse mémoire peut être interprétée comme ayant trois parties :

$$\text{Adresse} = \underbrace{0000\,0110\,0101\,01}_{\text{segment } 405}\,\underbrace{00\,1101\,0100}_{\text{page } 212}\,\underbrace{1000\,1001}_{\text{offset } 137}$$

Bien évidemment, comme pour la pagination, ce modèle n'est pas possible sans intervention de la couche machine. Le processeur qui exécute le système effectue lui-même la traduction d'une adresse virtuelle en adresse physique en consultant un ensemble de tables (qui elles sont gérées par le système) et vérifie que les droits de lecture–écriture–exécution sont bien respectés. Comme pour la pagination, la structure des tables peut devenir très compliquée, avec plusieurs niveaux et des systèmes d'antémémoire.

Notons, pour conclure, que le modèle présenté ici a été simplifié. Par exemple, il n'est pas nécessaire d'associer un segment à chaque procédure, mais plusieurs fonctions et procédures peuvent être regroupées dans un même segment.

## 1.5 Les systèmes de fichiers

Une autre forme de mémoire que le système doit gérer, est la mémoire dite *permanente* ou mémoire de masse. En d'autre termes : les disques (et par extension les CDROM, clés USB, bandes, ZIP, DAT, *etc.*). Ici encore le système d'exploitation sert d'intermédiaire entre le haut niveau (des applications souhaitant utiliser de la mémoire permanente) et le bas niveau (les pilotes des disques).

Comme il existe beaucoup de formats différents, selon les systèmes d'exploitation et les supports utilisés, nous ne fournirons qu'une description succincte de ce qu'un *système de fichiers* fournit comme services aux applications de haut niveau, et comment cela se traduit sur un support de type disque magnétique. Dans le chapitre suivant, les systèmes de fichiers d'Unix et Windows–NT/XP seront ensuite présentés en plus de détails.

### 1.5.1 Opérations de haut niveau

Le système fournit aux applications une mémoire permanente dans laquelle on peut stocker de l'information. Globalement parlant, le client n'a pas à savoir comment le système s'y prend. Ce qu'on lui offre sont des opérations du style :

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

- organiser les données et pouvoir les nommer de façon non-ambigüe,
- stocker et détruire des données,
- lire des données,
- écrire des données,

Dans certains cas se rajoutent des opérations de type :

- gérer des attributs d'une donnée ou d'un ensemble de données (droits d'accès, expiration de validité, *etc.*),
- avoir différents types de stockage (stockage caractère par caractère, bloc par bloc, *etc.*).

Dans la plupart des cas, le système d'exploitation permet donc ces opérations de façon entièrement transparente vis-à-vis du support sous-jacent. On retrouve presque partout la notion de fichier (document, *file*, ...) qui est la notion atomique du stockage permanent de stocker un ensemble de données propre à une application, ainsi que la notion de répertoire (dossier, *directory*, ...) qui est une collection de fichiers ou d'autres répertoires. On retrouve alors une structure arborescente avec un répertoire dit *racine* contenant un ensemble de fichiers et de sous-répertoires. Le nommage unique des fichiers se fait par concaténation des noms de répertoire formant le chemin depuis la racine au fichier en question.

La manipulation des fichiers se fait alors en fournissant simplement leur nom canonique en paramètre aux fonctions de base permettant de les créer, de les détruire, de les modifier, *etc.* (dans le principe ... en pratique, on utilise plutôt des descripteurs de fichiers que l'on obtient à partir du nom canonique).

### 1.5.2 Définition d'un système de fichiers et mise en œuvre

L'ensemble des données sur disque peut donc être vu comme un énorme catalogue de fichiers, référencés de façon non-ambigüe. C'est donc de fait une structure de données complexe qui doit pouvoir gérer à la fois les notions haut niveau : nom d'un fichier, ses attributs, son type (si cela a un sens pour le système), sa taille, l'endroit physique sur le support, les opérations en cours sur le fichier (gestion d'accès concurrents, modifications en cours, mais non transcrits sur le support, *etc.*), ... Mais le système doit aussi gérer les aspects bas-niveau : parties du support occupés et libres, sections endommagés, tampons de transfert, fragmentation, *etc.*.

Il y a, actuellement, peu de consensus sur ce point, ne serait-ce que à cause des différents supports utilisés et les besoins de différents types d'utilisateur. On peut avoir besoin de favoriser des accès rapides en lecture/écriture, ou alors préférer la fiabilité et la possibilité de retracer une opération si elle échoue (*rollback*), avec option de trace de toutes les opérations effectuées ; on

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

peut favoriser des fichiers de grande taille, de petite taille, minimiser l'espace inutilisé sur disque ou optimiser les temps de recherche d'un nom, p. ex.. Tous ces critères sont antagonistes, et il s'agit d'un problème d'optimisation complexe que de satisfaire chacun d'entre eux au mieux.

Nous détaillerons donc les mises en œuvre des quelques systèmes de fichiers lors de l'étude de cas du chapitre suivant§ **??** et § **??**) . Comme, par contre, la plupart des systèmes de fichiers sont faits pour des supports magnétiques type disque dur, nous détaillerons dans la section suivante, la structure bas niveau, telle qu'elle est aperçue par le système à travers le contrôleur de disque.

### 1.5.3 Structure bas niveau

Un disque dur classique est un ensemble de $\delta$ disques magnétiques parallèles entre lesquels peuvent se déplacer des têtes de lecture, comme le montre FIG. 1.19. Les disques effectuent une rotation autour de leur axe vertical (à une vitesse de l'ordre de 10000 RPM), tandis que les têtes peuvent se déplacer en translation horizontale. Chaque disque contient un nombre $\rho$ de *pistes*, qui sont des cercles concentriques. Une piste est donc tout ce qu'une tête de lecture peut lire lorsqu'elle reste immobile. On appelle *cylindre* l'ensemble de pistes de même diamètre sur tous les disques (un disque dur a donc au total $\rho \times \delta$ pistes et $\rho$ cylindres).

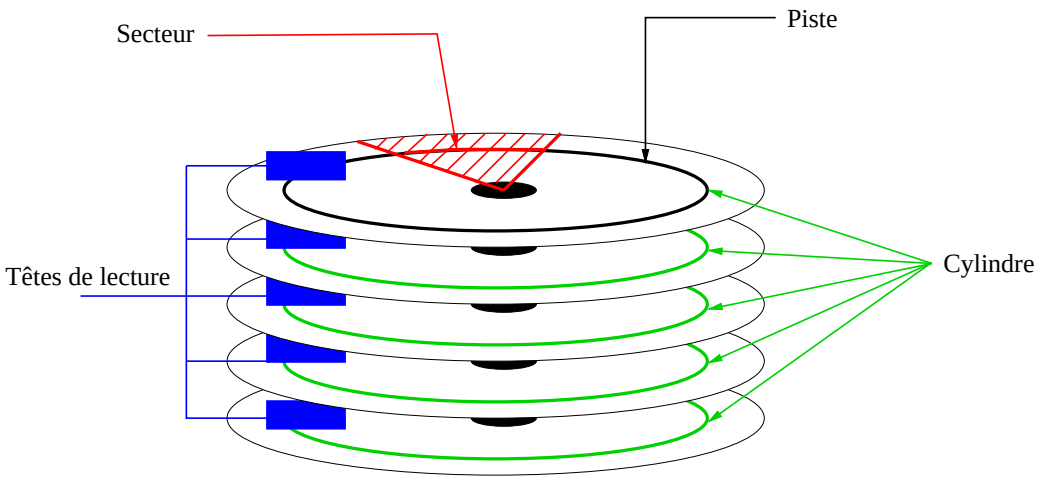


FIG. 1.19 – Structure bas niveau d'un disque dur.

Lorsqu'il est formaté, le disque dur contient un ensemble $\sigma$ de secteurs. Les secteurs forment une subdivision régulière des pistes en $f$ parts, de sorte qu'un disque dur formaté contient $\sigma = f \times \rho \times \delta$ secteurs. Le secteur est

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

la plus petite unité qui puisse être lue par le contrôleur de disque (et, par conséquent, par le système). Souvent, par contre, le système raisonne en termes de *blocs* (pour diverses raisons, l'une étant de se rendre indépendant du matériel utilisé, car tous les disques n'ont pas nécessairement la même taille de données par secteur) qui est de taille équivalente à un multiple entier de la taille d'un secteur.

Toute la difficulté des parties bas niveau d'un système de fichiers provient maintenant de faire correspondre des secteurs à des blocs et des blocs à des fichiers, en faisant en sorte que les temps d'accès soient optimaux, que la cohérence entre la représentation *logique* du système corresponde à la réalité *physique* sur le disque, même en cas de panne brusque du système, *etc.*.

Ces derniers points sont importants à l'égard de la fiabilité du système. En effet, pour des raisons d'efficacité, il existe souvent une mémoire tampon (*cache*) entre le contrôleur de périphérique et le système d'exploitation. Comme les accès aux périphériques sont généralement lents, toute modification physique du système de fichiers n'est pas toujours répercutée immédiatement, mais optimisée en fonction de la charge et des besoins. De ce fait, un retrait prématuré d'un périphérique ou un arrêt brutal du système a de grandes chances d'intervenir à un moment ou toutes les modifications n'ont pas encore été répercutées physiquement sur le support en question. Créant à coup sûr une perte de données, voire même un risque certain de corruption des structures logiques du système de fichiers, pouvant le rendre complètement inutilisable si l'opération intervient à un moment critique d'écriture.

Tous ces risques ont fait qu'il existe maintenant des systèmes de fichiers dits *à journal*. En échange d'un certain coût en temps, ces systèmes de fichiers intègrent une approche transactionnelle, inspirée des SGBD. Toute modification est découpée en étapes qui donnent lieu à un enrgegistrement dans un journal sur le disque, permettant ainsi, en cas de conflit, de retracer toute opération non achevée au moment de l'incident et d'annuler toute écriture incomplète (voire de la terminer dans certains cas) et de garantir que le système reste cohérent.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

# Chapitre 2

# Réseaux et systèmes d'exploitation interconnectés

Ce chapitre aborde l'interconnexion de systèmes d'exploitation. Jusqu'à maintenant, nous avons étudié les OS en tant que gestionnaire de ressources d'un ordinateur individuel. Avec l'émergence de l'Internet, on ne conçoit plus guère des ordinateurs seuls, mais plutôt en tant que entité communicante dans un réseau.

Ce chapitre est organisé comme suit :

1. Une brève introduction définit de façon intuitive la notion de réseau, aborde les principaux critères de classification de différents réseaux d'échange de données.
2. Ensuite, nous étudions en détail le protocole `TCP/IP`, sous-jacent à Internet.
3. Puis, nous abordons l'utilisation d'un réseau du point de vue d'un utilisateur du système d'exploitation qui cherche à exploiter ses possibilités de communication.
4. Le chapitre dédié aux systèmes répartis montre comment un système d'exploitation peut devenir un maillon d'une entité plus grande et coopérative ; fournissant des services à des clients, ou dépendant de services fournies par des serveurs.
5. Finalement nous évoquons les défis liés à la sécurité du système lorsqu'un OS devient connecté à un réseau.

## 2.1 Qu'est-ce qu'un réseau ?

La question de définir correctement la notion de « réseau » est probablement assez philosophique, et la réponse que l'on obtiendra dépendra très fortement de l'utilisation que l'on veut en faire. On parle, pêle-mêle de *réseau routier*, *réseau d'amis*, *réseaux d'influence*, *réseau téléphonique*, *réseau de distribution* et bien évidemment d'Internet. Une définition probablement assez exacte[1] se rapporterait à la définition mathématique du graphe, pour aboutir à quelque chose dans le genre : *ensemble de nœuds munis de moyens d'échange entre eux.* Ce qu'est un nœud ou en quoi consiste un échange, ce qu'on échange et comment reste à déterminer en fonction du réseau considéré.

Sans pour autant faire une étude socio-philosophique de la notion de réseau, il est intéressant de remarquer, pour la suite de l'exposé, qu'un réseau n'est pas une notion absolue, et que, par exemple, un réseau de distribution peut être construit au dessus d'un réseau de transport, le réseau de transport étant lui-même construit au dessus d'un réseau routier, ferroviaire, aérien et maritime.

Il peut paraître étrange, dans un cours d'informatique, de s'attarder à ce genre de réflexions. Nous verrons par la suite, que ces parallèles dans le monde réel, facilitent grandement les notions de protocole, d'encapsulation et du concept d'Internet lui-même.

### 2.1.1 Communication et protocoles

Si on revient à la définition générique d'un réseau, on y fait référence à des moyens d'échange. Si on admet qu'on n'échange jamais rien sans raison, on peut assimiler tout échange à une sorte de communication. Un réseau étant construit pour échanger avec un but précis, un réseau est donc nécessairement conçu pour servir de moyen de communication. On pourrait s'amuser à identifier la notion précise de communication pour les réseaux cités ci-dessus, ce qui nous intéresse plutôt ici (dans le but ultime d'en arriver à Internet, rappelons) est d'introduire la notion, et la nécessité de *protocole*.

Le but du réseau est de permettre la communication. Il est fondamental de dissocier les deux, néanmoins. Le réseau est simplement un moyen de communication, un médium. La communication, elle, en est indépendante. En revanche, le réseau impose que la communication se conforme à des règles afin de pouvoir s'établir. Ce sont ces règles que l'on appellera *protocole*.

Prenons deux exemples : le réseau ferroviaire, et le réseau postal.

[1] L'auteur se trouve dans le train Paris-Nancy au moment d'écrire cette phrase, et doute fortement qu'un jour il pensera à chercher la définition dans un dictionnaire digne de ce nom.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

**Le réseau ferroviaire** permet de « communiquer » des marchandises. Il dispose d'une infrastructure (les moyens d'échange) qui consiste en un maillage de gares, reliées par des rails. Il ne suffit toutefois pas de simplement déposer les marchandises sur les rails. Il faut conditionner les marchandises dans des volumes standards, les charger dans des wagons (qui respectent d'autres standards de longueur, de taille d'essieu, *etc.*) et de munir ces wagons d'une force motrice. Des marchandises hors gabarit, des wagons hors normes ou des wagons sans force motrice ne peuvent pas faire aboutir la « communication ».

**Le réseau postal** permet également de « communiquer » des marchandises, certes d'un gabarit différent que dans le cas précédent. Ce qui est important, en comparaison avec le cas ferroviaire, est que dans le réseau postal, la notion d'infrastructure devient virtuelle. Il n'y a plus de lien physique entre l'émetteur et le récepteur de la communication. Il n'y a pas non plus besoin de munir sa marchandise d'une force motrice. Le protocole de communication stipule un gabarit pour la marchandise, et la mention d'une adresse. Point important avec le cas précédent (et intimement lié à l'absence d'un lien physique entre l'émetteur et le récepteur) est le fait qu'à l'émission, il n'y a aucune garantie d'aboutissement de la communication. Le fait que l'adresse soit erronée ne gène en aucun cas le départ de la marchandise, ce n'est qu'à partir d'une certaine durée de présence dans le réseau que l'on pourra déterminer si la communication peut s'établir.

Le but de ces exemples étaient d'introduire de façon intuitive deux notions essentielles : *l'encapsulation* et *le protocole de communication*.

**L'encapsulation** est une notion très forte et essentielle. Elle consiste à dire que dès lors que l'objet d'échange se conforme aux contraintes physiques du réseau (on n'essayera pas de transmettre de l'eau via le réseau téléphonique ou du 220V à travers les égouts), le réseau lui-même n'a que faire de ce qui est transmis. Il se borne à une tâche bien spécifique qui est l'échange ou la transmission, selon un ensemble de règles. Le comportement du réseau est complètement indépendant de l'objet transporté.

**Le protocole de communication** est un ensemble de règles d'empaquetage des objets à transporter et de règles de conduite, éventuellement combiné à des échanges de meta-données préliminaires ou postérieures à l'échange de données.

Cette introduction en la matière un peu inhabituelle a pour but de montrer qu'un réseau, les protocoles et l'échange des données sont, en effet, des

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

concepts très génériques, dont le modèle mathématique sous-jacent est très facilement généralisable à un grand nombre de situations. Dans le reste du chapitre on se consacrera uniquement à des réseaux d'échange de données numériques, et nous n'aborderons pas en détail les modélisations formelles qui peuvent les sous-tendre.

## 2.2 Réseaux d'échange de données

Cette section, sans pour autant rentrer dans des détails de modélisation formelle, donne quelques caractéristiques générales de réseaux d'échange de données numériques. On aborde en première instance les différentes classifications habituelles des types de réseaux, on introduit la notion de multiplexage, et on termine par la modèle en couches ISO des réseaux d'échange de données.

### 2.2.1 Classifications

Même à l'intérieur des réseaux d'échange des données, il y beaucoup de différents cas de figures d'utilisation, de contraintes ou de déploiement. On donne ici quelques critères de distinction entre ces différents types réseaux, sans pour autant vouloir être exhaustif ou prétendre qu'il s'agit d'une partition stricte, rigoureuse et immuable.

#### 2.2.1.1 Topologie

Le premier type de classification concerne la topologie du réseau. Souvent, lorsqu'on peut faire des hypothèses sur la façon dont les points d'entrée du réseau sont interconnectés physiquement, certaines solutions techniques sont plus appropriées que d'autres (p. ex. les réseaux de type *bus* sont idéaux pour des transmissions type diffusion, tandis que les structures en anneau conviennent très bien à des réseaux à jeton), bien qu'elles ne les excluent pas.

On distingue cinq classes principales de topologies de réseaux, plus une sixième, qui est hybride :

- Les réseaux *point-à-point* sont les plus simples d'un point de vue de la gestion des communications : chaque nœud est directement relié à tout autre nœud du réseau. La communication d'un nœudà un autre consiste donc simplement à transmettre sur le médium directement connecté à l'interlocuteur.
- Les réseaux *en étoile* sont organisés autour d'un nœud central par lequel toutes les communications passent obligatoirement. C'est typiquement

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

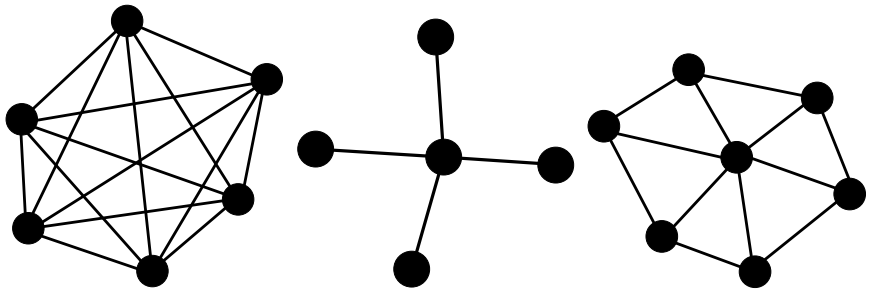

FIG. 2.1 – *De gauche à droite :* réseau point-à-point, réseau en étoile, réseau maillé.

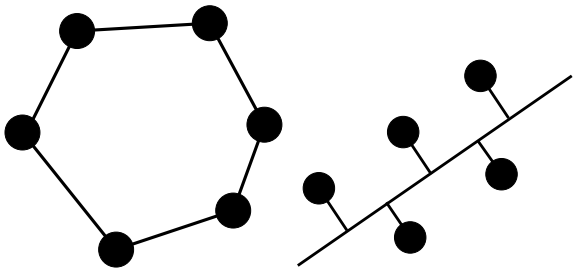

FIG. 2.2 – *De gauche à droite :* réseau en anneau, réseau en bus.

le cas dans ce qu'on appelle (faussement) la boucle locale du réseau téléphonique, ou encore dans les réseaux de terminaux d'un gros calculateur de type *mainframe*.

- Les réseaux *maillés* sont des réseaux où chaque nœud est connecté directement à ses voisins les plus proches. Si un nœud veut communiquer avec un autre, plus éloigné, il doit faire transiter la communication par un de ses voisins.
- Les réseaux *en anneau* portent bien leur nom. Ils forment une boucle fermée, et chaque nœud a exactement deux voisins. La communication doit se faire en faisant transiter le message de voisin en voisin, jusqu'à destination.
- Les réseaux *en bus* sont construit autour d'une *épine dorsale* à laquelle tous les nœuds sont connectés. La communication sur ce genre de réseaux ne peut être faite que par diffusion : tous les nœuds reçoivent tous les messages simultanément (ou presque).

Le sixième type de réseau est tout simplement l'interconnexion d'un certain nombre de ce classes de réseaux. On l'appelle réseau hétérogène. Il est clair que les différents types de réseaux répondent à différents types de besoins, et soulèvent différents problèmes de mise en œuvre des communications (diffi-

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

culté d'ajouter un nouveau nœud dans un réseau point à point ou en anneau, gestion de l'acheminement dans un réseau maillé, gestion du partage d'accès à un réseau en bus, *etc.*).

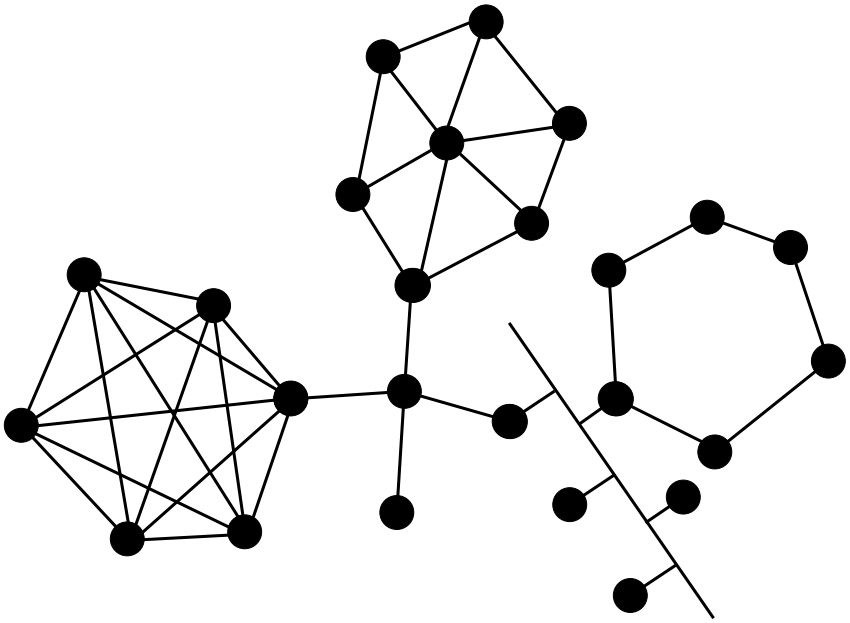

FIG. 2.3 – Réseau hétérogène.

#### 2.2.1.2 Taille

On distingue des réseaux également selon leur taille physique, non tellement en nombre de nœuds, mais en couverture géographique et débit de transmission. On parle ainsi de WAN pour *Wide Area Network* ou réseaux à large couverture ou réseaux globaux, et de LAN pour *Local Area Network* ou réseaux locaux. La séparation entre les deux n'est pas toujours bien franche. L'ordre de grandeur d'un bâtiment concerne forcément un LAN, celui d'un pays un WAN, la taille d'une ville étant dans une zone de floue. Dans ce cas, c'est l'utilisation, le débit ou le flux qui détermineront la classification (on parle d'ailleurs aussi de MAN pour *metropolitan area network*.

Depuis le développement des technologies dites *sans fil* (Bluetooth, WiFi, ...) combiné au développement des terminaux mobiles (téléphones, organiseur, *etc.*) le terme PAN (*Personnal Area Network*) est maintenant aussi rentré dans le jargon lié aux réseaux. La portée d'un PAN ne dépasse pas quelques dizaines de mètres.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

#### 2.2.1.3 Média

Il existe différents types de média de transmission. Les plus fréquents sont les câbles en cuivre (coaxial, torsadé), les fibres optiques ou les ondes hertziennes. Tous peuvent s'appliquer à un même type de réseau et/ou de protocole (p. ex. Ethernet) mais répondent à des besoins différents de coût d'équipement, de débit ou de qualité et de flexibilité de service.

#### 2.2.1.4 Commutation

Le type de mise en relation entre les nœuds communicants est un critère important pour la disponibilité du réseau. On appelle *commutation* l'établissement d'un lien entre deux communicants. On parle de commutation par circuits (ou commutation physique) ou commutation par paquets (ou commutation virtuelle)[2].

La commutation par circuits est ce qui se passe dans le réseau téléphonique classique[3] (on fait abstraction de l'utilisation de plus en plus répandue de *voix-sur-IP*). Lors de l'établissement de la communication il existe un chemin physique entre le deux nœuds par lequel le signal transite. On parle également, dans ce cas de communication en mode connecté. Chaque nœud intermédiaire dispose d'un certain nombre de circuits qu'il peut utiliser pour connecter l'un de ses voisins à un autre. La communication entre deux points éloignés s'établit lorsque tous les nœuds intermédiaires ont établi une connexion avec leur voisin approprié (et ont donc « consommé » l'un de leurs circuits), formant ainsi une liaison physique entre les deux points. À la fin de la communication les nœud intermédiaires libèrent leur circuit pour d'éventuelles autres communications. Lorsqu'un nœud n'a plus de circuit de disponible, la communication ne peut s'établir[4]. Le grand avantage des réseaux à commutation de circuits est la disponibilité totale de la bande passante du médium de communication. Les deux parties communicantes peuvent être assurées d'une liaison de qualité constante (débit, temps de réponse, latence[5]). L'inconvénient majeur est l'impossibilité d'éventuelles autres parties d'utiliser le médium si les deux nœuds initiaux ne l'utilisent pas tout le temps

[2] En anglais on parle de *switching* : *circuit switching* ou *paquet switching*

[3] On trouve souvent l'acronyme RTC lorsqu'on parle du *Réseau Téléphonique Commuté*

[4] Finalement, dans le principe de fonctionnement, peu de choses ont changé depuis « Le 22 à Asnières » de Fernand Raynaud, si ce n'est que la disparition de l'intervention humaine dans l'établissement des circuits.

[5] Le facteur de latence est le temps qu'une donnée passe dans le réseau entre l'émetteur et le récepteur ; c'est la différence entre le temps de réception et le temps d'émission. Il est dissocié du débit, et dans certains cas il peut être préférable d'avoir une faible latence plutôt que d'avoir un haut débit.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

ou en deçà des capacités de transfert (p. ex. au téléphone, même si vous ne parlez pas, la communication reste établie).

L'autre possibilité de commutation est la commutation par paquets. Au lieu d'établir une liaison physique entre deux communicants, on établit une liaison virtuelle (on parle de communication en mode déconnecté). La liaison virtuelle consiste en une séquence de nœuds qui doit être traversée pour arriver au destinataire. Cette séquence peut être statique (établie lors de la demande de connexion et identique pendant la durée de la communication) ou dynamique (variable pendant la communication). Ensuite, les données à transmettre sont séparées dans des « paquets » et transmis au nœud suivant, qui transmet au suivant, *etc.* On peut tirer un parallèle avec le réseau de distribution de la Poste. L'avantage de ce genre de commutation est l'utilisation optimale des ressources du réseau, puisqu'un nœud gère simplement une pile de paquets entrants et les redistribue. Il n'a pas à réserver un circuit pour une communication donnée. Les nœuds sont donc toujours assurés de pouvoir établir une communication. Par contre, ils ne sont plus assurés d'avoir une qualité de service constante. Un autre effet de bord est que les nœuds intermédiaires doivent fournir plus de travail. Dans le cas d'une commutation de circuits, une fois le circuit établi, un nœud est une simple connexion entre deux fils. Dans le cas d'une commutation par paquets, chaque paquet doit être analysé et retransmis, ce qui demande du matériel plus « intelligent », fournissant plus de calcul, et, par conséquent, plus lent mais plus coûteux.

### 2.2.2 Transmission de signal et multiplexage

Cette section n'existe que par simple intérêt de culture générale. Elle aborde, dans rentrer trop dans les détails le principe de *Multiplexage*, ou, « comment s'y prendre pour transmettre plusieurs signaux sur un même médium et en même temps ? ». Elle peut être sautée par le lecteur non intéressé.

Pour bien faire, cette section devrait comporter la bases du traitement du signal, ce qui est largement hors de la portée de ce cours. Nous nous contenterons de seulement aborder les principes de base d'un point de vue conceptuel. Les lecteurs vraiment intéressés se réfèreront à la bibliographie sur le sujet à la fin de ce polycopié.

#### 2.2.2.1 Principes de base de la transmission d'un signal

Historiquement, on a commencé par transmettre des signaux analogiques sur différents média. On s'est alors aperçu que chaque médium se comporte comme un *filtre* et qu'il dispose d'une *bande passante*. En d'autres termes,

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

un médium transmet bien les fréquences d'un signal lorsqu'elles se trouvent dans une certaine plage $\left[f_{\min}..f_{\max}\right]$. Les fréquences en dehors de cette plage sont trop déformées pour être utilisables ou alors tout simplement non transmises. C'est cette plage que l'on appelle *bande passante*, et le fait de déformer ou d'annuler certaines fréquences forme un *filtre*.

Il faut savoir également, que depuis Joseph Fourier en 1807, on sait qu'un signal $s(t)$ dans le domaine temporel a son équivalent $S(f)$ dans le domaine fréquentiel, et que chacune des deux représentations caractérise complètement le signal. La représentation en fréquence consiste à décomposer le signal en une superposition de différentes fonctions de base. Typiquement, ce sont les séries de Fourier qui servent aux signaux périodiques. Ceux-ci peuvent être représentés comme une superposition d'une onde sinusoïdale fondamentale de fréquence $\omega$ et de divers harmoniques.

$$s(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \cos\left(n\omega t\right) + b_n \sin\left(n\omega t\right)\right) \tag{2.1}$$

Pour les signaux non périodiques, on se sert alors d'une intégrale. La méthode consiste alors à représenter le signal par une superposition d'ondes sinusoïdales de toutes les fréquences possibles : la transformée de Fourier.

$$S(f) = \int_{-\infty}^{\infty} s(t) e^{2i\pi f t} dt \tag{2.2}$$

Les problèmes commencent alors lorsqu'on veut transmettre des données binaires. Le signal théorique est, dans ce cas, une série de transitions brusques de 0 à 1 (*cf.* FIG. 2.4).

Ces transitions font intervenir des fréquences infinies (comme le montre sa transformée de Fourier FIG. 2.5), et comme les support utilisés ont une bande passante finie, il est évident que ce signal ne peut être transmis en tant que tel.

Le signal effectivement transmis sur le support utilisé, ressemblera plutôt à celui représenté dans FIG. 2.6 [6]. En fait, conceptuellement, le changement brusque d'un état 0 à 1, fait intervenir une pente infinie, et donc une fréquence infinie. La bande passante définissant la fréquence maximale qui peut être transmise, elle détermine, en quelque sorte, la pente (théorique) maximale permettant de passer d'un état 0 à un état 1. Il est trivial de voir que, *a fortiori*, la bande passante détermine le nombre de changements d'état

[6] L'application d'un filtre parfait : on prend le signal $s(t)$, on calcule sa transformée de Fourier $S(f)$, on tronque les hautes fréquences, et on opère la transformée inverse pour obtenir le signal $s'(t)$ filtré.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

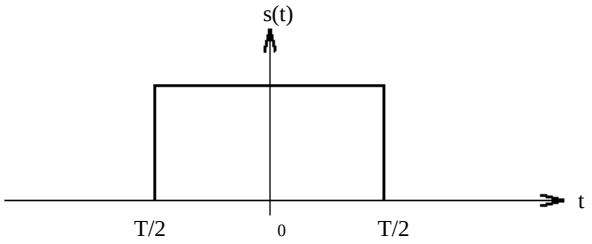


FIG. 2.4 – Signal binaire simple.

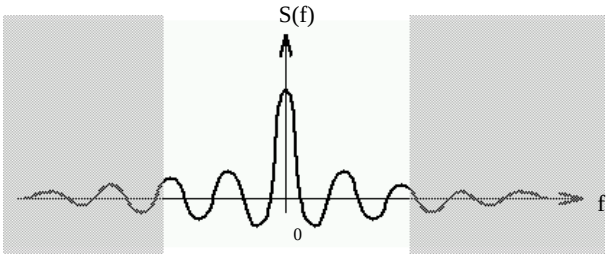


FIG. 2.5 – Transformée de Fourier du signal binaire simple avec superposition de bande passante (partie réelle).

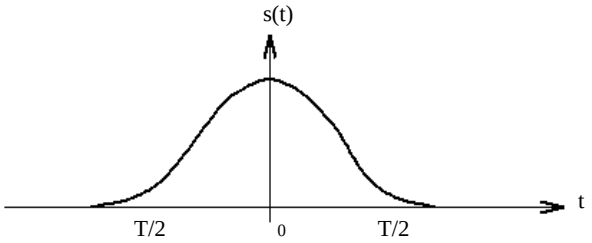


FIG. 2.6 – Signal binaire simple effectivement transmis.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

autorisés en un intervalle de temps donné et qu'il y a donc une relation directe entre la *bande passante* et le débit maximal d'un support de transmission[7].

#### 2.2.2.2 Multiplexage en fréquence

Si on raisonne dans l'espace fréquentiel, la notion de multiplexage en fréquence est relativement facile à concevoir. Comme nous l'avons vu dans FIG. 2.5 un signal est constitué d'un certain nombre de fréquences superposées, mais que le médium utilisé ne laisse passer que celles qui sont dans sa bande passante. Souvent, la bande passante effective permet un débit qui est généralement supérieur à la demande. Par contre, il est fréquent que plusieurs transmissions doivent être émises simultanément. L'idée du multiplexage en fréquence est donc de découper virtuellement la bande fréquentielle $\left[f_{\min}..f_{\max}\right]$ du support en $n$ sous bandes de largeur égale. À chaque bande on fait correspondre une fréquence de référence $f_i$, appelée *porteuse* (*cf.* FIG. 2.7). On dispose donc ainsi, au lieu d'un support de transmission de bande passante $B$, de $n$ supports de bande passante $B/n$.

Faire passer $n$ signaux sur le même support revient alors à coder chaque signal pour une bande passante de largeur $\frac{f_{\max}-f_{\min}}{n}$. Puis, dans le domaine fréquentiel, chaque signal est translaté de $f_i$, où $i$ est le canal virtuel dans lequel il va passer. Les signaux sont superposés et transmis sur le support. À l'arrivée, on applique $n$ filtres passe-bande, correspondant aux bandes virtuelles utilisées pour récupérer les signaux d'origine.

Dans FIG. 2.7 on montre ce qui se passe si on veut émettre le même signal en simultané sur trois canaux différents (bien évidemment on pourrait aussi émettre trois signaux différents) et le signal composite obtenu. En résumé, comme le montre la figure FIG. 2.8, le principe du multiplexage en fréquence consiste à diviser un canal de communication en $n$ nous-canaux indépendants, pour faire passer différents signaux.

#### 2.2.2.3 Multiplexage en temps

Un autre type est le multiplexage temporel. Au lieu de réserver une sous-bande passante pour chaque canal virtuel, comme c'était le cas dans le multiplexage en fréquence, on réserve tout le médium à un seul signal pendant un intervalle de temps fixe, et on permute avec les autres signaux. En quelque sorte, on découpe chaque signal à transmettre en petits paquets, et on transmet alternativement un paquet de chaque signal (*cf.* FIG. 2.9).

[7] La relation n'est pas si directe que ça, car elle fait intervenir d'autres facteurs comme pêle-mêle, les effets de repliement de spectre, le nombre d'états de transition, le bruit et le taux d'échantillonage (*cf.* théorème de Shannon–Nyquist, p. ex.)

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

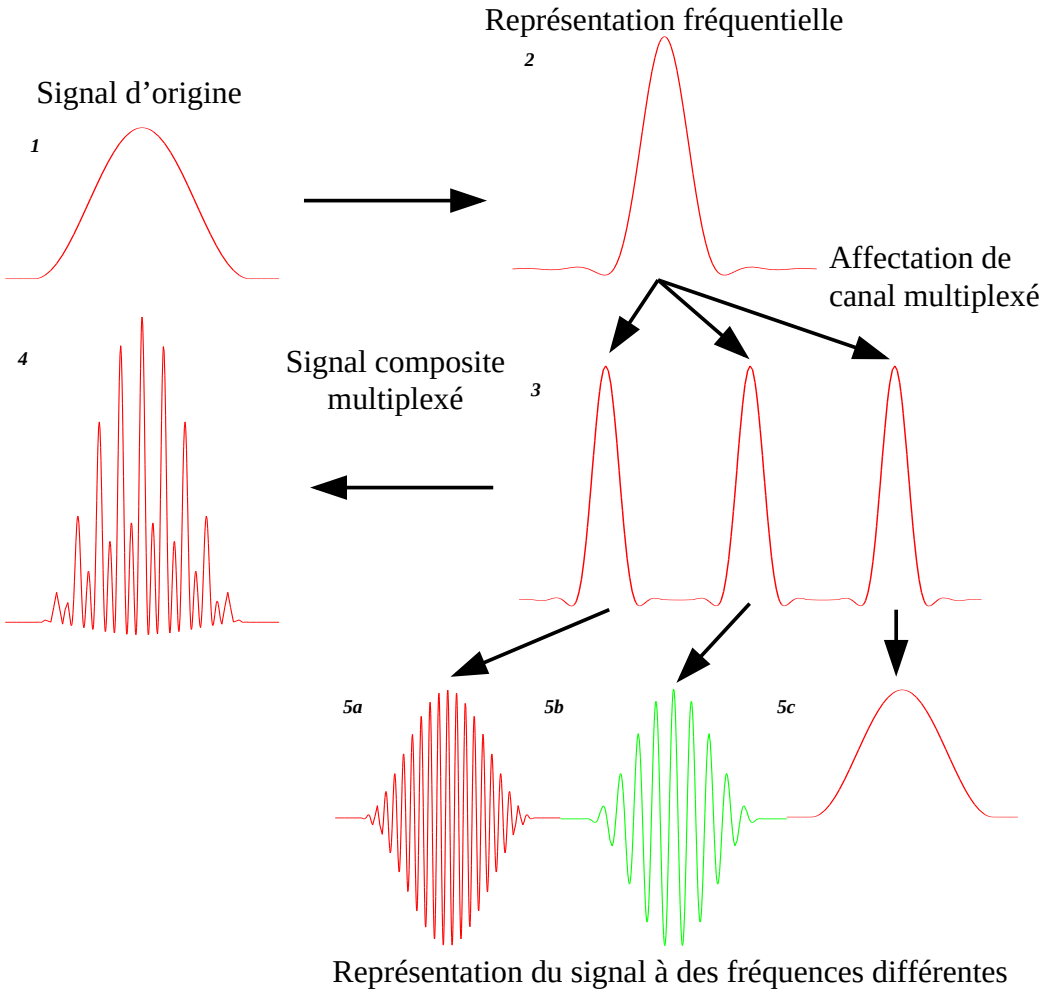


FIG. 2.7 – Principe de multiplexage en fréquence par transformée de Fourier.

On peut, sans prétention aucune, tirer un parallèle entre la commutation et le multiplexage. En effet, la même analyse de performance ou d'utilisation tient dans le cas d'une commutation par circuit et d'un multiplexage en fréquence, ainsi que pour une commutation par paquets et un multiplexage temporel.

### 2.2.3 Le modèle en couches ISO–OSI

La question qui se pose alors, au vue des différentes catégories et choix d'implémentation de réseaux des sections précédentes : « *Comment concevoir un protocole qui marche parfaitement sur un réseau hétérogène de tout point de vue ?* »

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

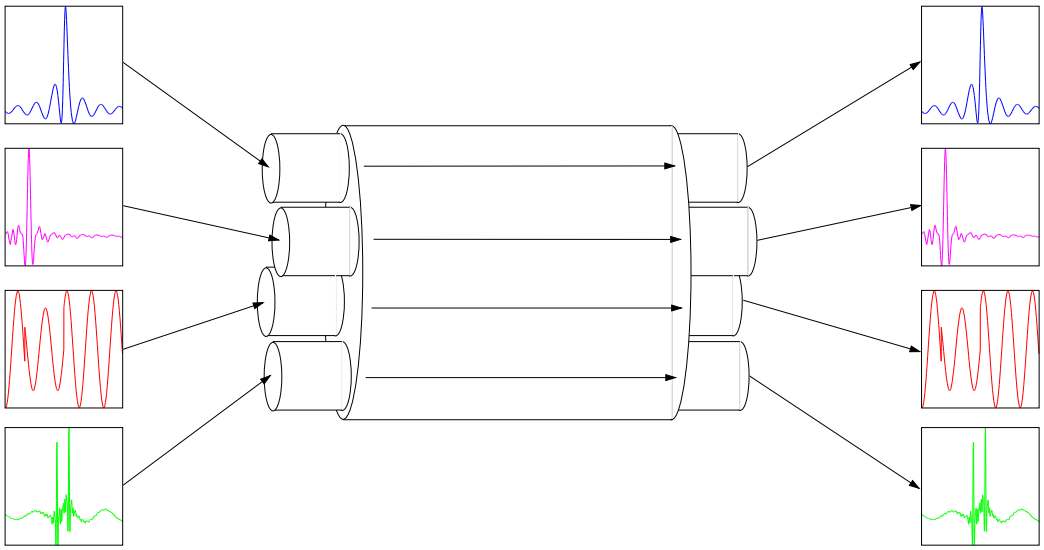

FIG. 2.8 – Principe de multiplexage en fréquence. On subdivise la bande passante globale en sous-bandes, et on alloue une sous-bande à chaque signal à transmettre.

La question est loin d'être triviale. Quels sont les éléments avec lesquels un concepteur de logiciel communicant doit tenir compte ? Est-ce qu'il doit prévoir tous les cas de figure de configuration de réseau, de protocole de transmission intermédiaire, de fréquence d'échantillonnage, *etc.* ? Comment faire si une application, conçue pour un type de réseau, doit être adaptée à une autre configuration, sans avoir à tout réécrire, p. ex. ?

La solution vient du modèle ISO–OSI (*International Standard Organization – Open System Interconnect*) qui décrit comment des logiciels communicants doivent être conçus afin d'abstraire au maximum les concepts qui leur sont nécessaires en tant que « services », mais dont ils ne doivent pas connaître les implémentations ou le fonctionnement interne. On définit ainsi sept niveaux de services, appelées « *couches* », et un principe d'encapsulation strict qui impose à une application qui tourne sur une machine donnée, qu'une couche ne passe des données qu'à la couche lui étant immédiatement supérieure et à celle lui étant immédiatement inférieure. Dans ce contexte là, on parle souvent de *pile*, car une application doit, en quelque sorte, dépiler (*i.e.* parcourir dans l'ordre) toutes les couches pour accéder au réseau. Au niveau de l'échange de données entre applications, une couche donnée ne s'adresse qu'à la couche de même niveau de l'autre côté. Comme le montre FIG. 2.10, si une couche $n$ a des données à transmettre à son correspondant,

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

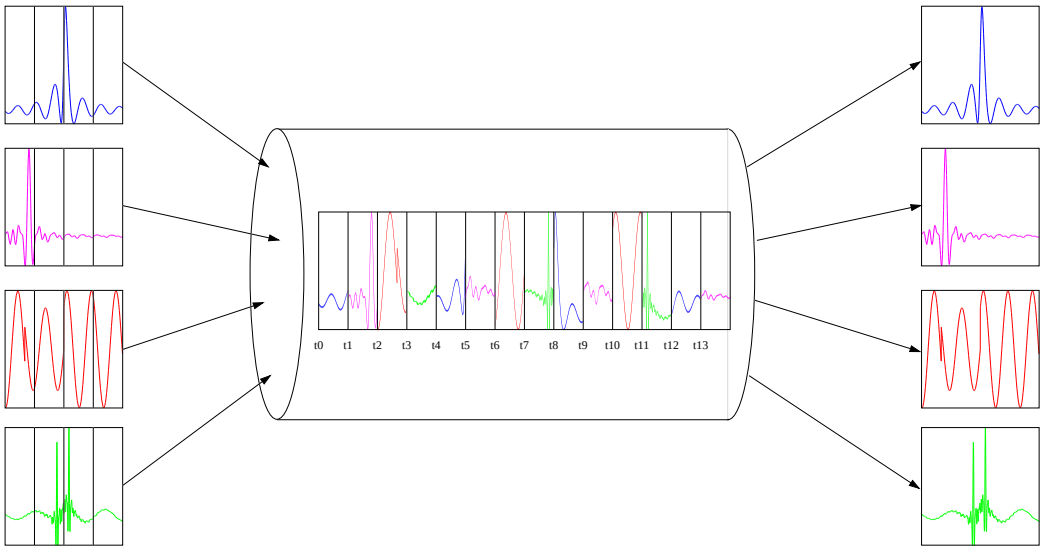


FIG. 2.9 – Principe de multiplexage en temps. On alloue toute la bande passante à chaque signal pendant une durée déterminée, puis on alterne.

elle transmet ces données à la couche $n-1$ qui lui est inférieure, en lui fournissant les méta-données nécessaires à leur transmission. Celle-ci contacte la couche $n-1$ du correspondant avec la requête de transmettre ce qui suit à la couche $n$. Elle encapsule les données et transmet à la couche inférieure, *etc.*

1. *La couche Physique*

   La première couche du modèle gère tout ce qui concerne les caractéristiques physiques et électriques du médium de transmission : les fréquences d'émission, les tensions à appliquer, la taille et forme des connecteurs, *etc.* Elle reçoit de la couche supérieure des bits qu'elle se contente de transformer en un signal approprié. Elle permet également de répondre à des questions sur l'état du médium.

2. *La couche Liaison*

   La couche liaison a comme tâche de rendre le moyen de communication brut, fourni par la couche physique, fiable, et de fournir un moyen de communication *point-à-point* sans erreur de transmission. Elle ne fournit aucun autre service que de transmettre les données qu'elle reçoit de la couche *réseau* aux machines qui sont reliés à elle via la couche physique. Le cas échéant elle est responsable de la ré-émission de données en cas de défaillance de la couche *physique*. Les données sont transmises par unités qui sont appelées *trames* (*frames* en anglais).

3. *La couche Réseau*

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

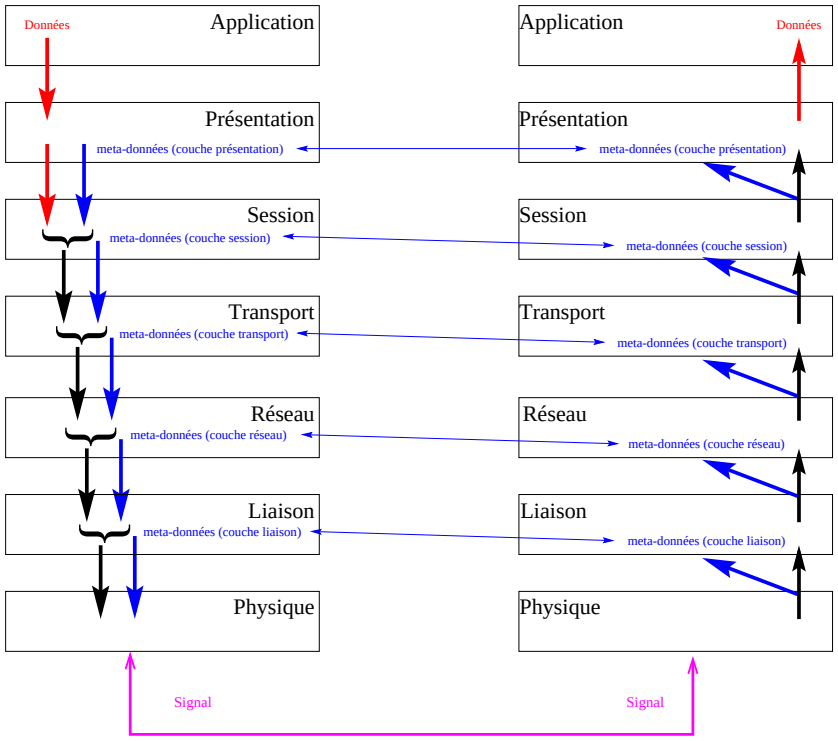


FIG. 2.10 – Principe de l'encapsulation des données dans le modèle à couches ISO–OSI

Cette couche fournit aux couches supérieures le service d'acheminement de données vers un destinataire, en s'appuyant directement sur les liaisons dont il dispose. En effet, la couche *liaison* ne permet les communications qu'entre nœuds directement connectés entre eux. La couche réseau permet la transmission des données à travers différents nœuds et résout les problèmes de routage, ou encore d'adressage et de nommage des nœuds. De ce fait, la couche réseau est responsable de gérer la différence entre protocoles ou l'utilisation de la couche liaison aux endroits stratégiques (points de connexion entre réseaux de différent type), puisqu'elle doit fournir une interface entièrement transparente à l'utilisateur, qui n'a pas à se soucier des types de réseaux intermédiaires par lesquels passent ses données. La couche doit aussi résoudre les problèmes d'acheminement des données en déterminant le meilleur chemin (*routage*), et en l'adaptant éventuellement à l'état actuel du réseau (*gestion du congestionnement*). Finalement, la couche réseau doit permettre (selon les cas) des communications en mode connecté – en établissant un circuit virtuel entre les deux communicants, et en garan-

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

tissant le chemin pendant toute la durée de la communication – ou en mode déconnecté – en routant chaque paquet de données individuellement. En mode déconnecté, une unité de données transmises s'appelle *datagramme*.

4. *La couche Transport*

   La couche Transport fournit un protocole de communication de pair à pair. Bien que la couche réseau assure l'acheminement des données, il n'y a pas d'échange de données direct entre l'émetteur et le récepteur et il y a, explicitement une notion de nœuds intermédiaires, p. ex. La couche transport utilise le réseau en tant que service abstrait et ne s'intéresse qu'à contacter un interlocuteur distant, indépendamment du chemin parcouru (*cf.*. FIG. 2.11).

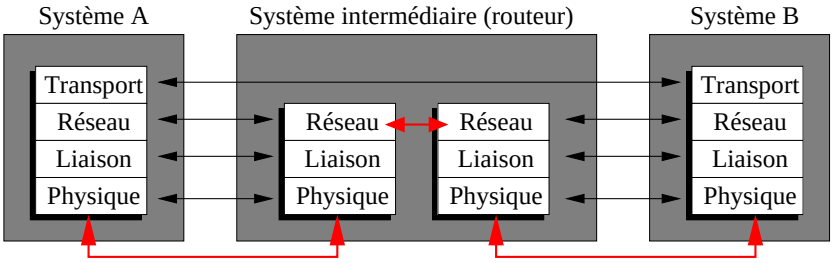


FIG. 2.11 – La communication pair-à-pair de la couche transport ISO–OSI : les flèches en noir représentent les échanges de données entre couches correspondantes ; les flèches en rouge représentent le flot de données.

Le rôle de base de la couche transport est de recevoir des données de la couche session, de redimensionner les données si besoin ait, de les transmettre à la couche réseau pour transmission, et de s'assurer que tous les paquets arrivent correctement à destination, et dans le bon ordre. Elle définit la structure des *segments* transmis afin de pouvoir en contrôler l'intégrité.

La couche transport doit fournir quatre services principaux :

(a) La gestion du flux :

La gestion de flux assure que l'émetteur adapte sa vitesse en fonction des capacités de réception du récepteur, pour ne pas le submerger de données et d'éviter que des nœuds à haut débit engorgent le réseau.

(b) Le multiplexage :

La couche transport peut, selon ses besoins, utiliser une même connexion de la couche réseau pour faire passer plusieurs communications simultanées. Ou, au contraire, si elle a besoin d'une

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

large bande passante, ouvrir plusieurs connexions pour une même communication. Dans tous les cas, la norme spécifie que ce multiplexage doit être complètement transparent pour la couche session.

(c) La gestion de circuits virtuels :
Bien que c'est la couche réseau qui offre la possibilité de faire des communications en mode connecté, c'est la couche transport qui veille à la gestion de circuits crées. Elle se charge de les établir, de les maintenir actifs et de les fermer correctement.

(d) La vérification et correction d'erreurs :
Comme nous l'avons déjà évoqué, la couche transport garantit une fiabilité totale de transmission des données par des systèmes de détection d'erreurs et de correction et/ou de retransmission, le cas échéant.

En résumé, la couche transport fournit un certain nombre de services à la couche session. Le plus utilisé est de fournir un canal de communication fiable, point à point à travers lequel l'ordre des données envoyées est respectée, mais d'autres types de service existent (remise au mieux sans garantie, p. ex.). Le fait que les machines communicantes ont des moyens pour faire tourner plusieurs programmes en même temps (*cf.* la multiprogrammation § 1.3, p. 14), la couche transport doit fournir des mécanismes (entêtes des paquets transmis, système de nommage) pour bien identifier les processus (ou ports) communicants de chaque interlocuteur afin de déterminer quelle donnée appartient à quelle connexion.

5. *La couche Session*

La couche session s'affranchit de la simple notion de transmission, et intègre une première notion de *communication*. Elle se sert de la couche transport pour ouvrir, fermer et gérer des connexions entre processus de différentes machines, mais les considère comme des *séances* (ou sessions), dont la durée de vie peut être supérieure à la connexion proprement dite, et auxquelles elle rajoute des fonctionnalités d'administration comme des journaux de session, de la sécurité (authentification, échange de clés), *etc.*

La couche gère d'éventuelles notions de priorité de communication, de tour de parole, de gestion d'accès concurrentiel et de synchronisation. Une défaillance totale du réseau, notamment, doit rester complètement transparente pour l'utilisateur de la couche session. C'est elle qui se charge d'insérer dans le flot de données des points de contrôle pour la reprise des échanges, par exemple.

6. *La couche Présentation*

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

L'avant-dernière couche du modèle sert de « *traducteur* » entre deux communicants. Elle s'assure que la couche application de la machine A puisse comprendre les représentations des données de la machine B en appliquant des fonctions de conversion ou d'encodage des données (Unicode, grand-boutistes/petit-boutistes, *etc.*). Cela concerne également la compression ou l'encryptage des données.

7. *La couche Application*
   La dernière couche ne doit pas être confondu avec l'application elle-même, mais plutôt comme un ensemble de services ou protocoles fourni aux applications. Les fonctionnalités concernent principalement l'identification aisée des interlocuteurs, l'abstraction du réseau et des protocoles d'échange bas-niveau, l'allocation des ressources, transfert de fichiers, *etc.*

Le tableau 2.1 récapitule les sept couches et leur rôle. Dans les sections suivantes, où nous aborderons la pile de protocoles `TCP/IP`, nous regarderons plus en détail certaines couches. Il faut savoir, néanmoins, que la famille de protocoles en question, qui sont les plus répandus dans l'Internet aujourd'hui, ne suit pas le modèle ISO–OSI *stricto sensu*, mais forme plutôt une simplification de celui-ci. Cela n'enlève en rien de la validité ou la légitimité de l'un ou de l'autre. Parmi les autres protocoles, respectant plus strictement le modèle OSI, on trouve, par exemple, SNA d'IBM, ou encore X.25, protocole très répandu dans les télécom.

### 2.2.4 Les protocoles « Liaison »

La couche liaison assure l'établissement d'une liaison logique entre deux ordinateurs physiquement reliés entre eux et qui veulent communiquer. La couche fournit la possibilité d'envoyer et de recevoir des données, d'en définir le destinataire ou d'en connaître l'expéditeur (parmi les machines physiquement et directement reliés entre elles). Conforme au concept en couches, le protocole de liaison n'a que faire de la technologie de transmission sous-jacente utilisée, et, par exemple, Ethernet, s'applique aussi bien sur un support en cuivre (câble coaxial – BNC, paire torsadée – RJ45), en fibre optique ou sur un support hertzien.

En revanche, dans la normalisation OSI chaque terminal de la couche liaison est identifiable de façon unique par une adresse (codée sur 48 bits) appelée adresse MAC (*Media Access Control address*). Cette adresse contient notamment l'identifiant du fabricant du terminal (généralement la carte réseau).

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

| | | |
|---|---|---|
| 7 | Application | abstraction/transparence du réseau, protocoles de haut niveau, SMTP, HTTP, LDAP, *etc.* |
| 6 | Présentation | représentation des données, compression, cryptographie |
| 5 | Session *(séance)* | gestion de « session » de communication, reprise après défaillance du réseau, synchronisation, gestion d'accès simultané, *etc.* |
| 4 | Transport *(segment)* | fiabilité de transmission, ouverture/fermeture de connexion, multiplexage, contrôle de flux. |
| 3 | Réseau *(datagramme/ datagram)* | transmission à travers un réseau, routage, gestion de congestionnement, adressage, ... |
| 2 | Liaison *(trame/frame)* | émission et réception de paquets de bits (trames) entre machines partageant un même support de transmission. |
| 1 | Physique | caractéristiques physiques du support de transmission, connectique, *etc.* |

TAB. 2.1 – Récapitulatif des couches ISO–OSI et de leur rôle

#### 2.2.4.1 Les réseaux à jeton – Token Ring

Les anneaux à jeton, et plus particulièrement Token Ring, dont la norme IEEE 802.5 est l'incarnation, ou FDDI – ANSI, sont une technique de partage de bande passante déterministe entre plusieurs terminaux partageant une même liaison sur le principe de transmission d'un « *droit de parole* », appelé *jeton*.

La première commercialisation de Token Ring, produit de IBM, date des débuts des années 1970. La technologie a pendant très longtemps été au cœur de l'offre de réseaux locaux d'IBM. Considéré comme conceptuellement supérieure à Ethernet (car plus structuré et, surtout, plus déterministe) elle a, au final, fait les frais de sa solidité au détriment des coûts de développement des commutateurs Ethernet, largement inférieurs aux commutateurs Token Ring.

Pour la petite histoire, l'inventeur du réseau à jeton ne serait pas IBM, mais Olof SODERBLOM technicien à la *Svenska Handelsbanken* qui déposa un

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

brevet en 1967. Il se trouva qu'IBM était, à l'époque, l'entreprise en charge du déploiement et de la maintenance du réseau de cette banque. En 1981, SODERBLOM a fait valoir l'antériorité de son brevet, et s'est vu attribuer dans les $100 millions de droits de licence pour l'exploitation de son invention par des entreprises comme IBM ou Hewlett Packard. Par la suite, le jugement attribuant ces droits de licence a été cassé en 1992.

Le principe de communication de Token Ring est la communication avec jeton. Un réseau à jeton peut être vu comme un anneau, dans lequel chaque nœud est relié à 2 voisins. Un nœud est autorisé à émettre seulement lorsqu'il est en possession du jeton. Lorsqu'un nœud n'a pas besoin d'émettre ou lorsque son quota de temps a expiré, il transmet le jeton à son voisin direct. En réalité, le jeton est une petite trame unique et bien identifiable qui est envoyé par un nœudà son voisin.

Par la suite, cette architecture a évolué, puisqu'elle comportait trop de problèmes techniques (insérer un nœud dans l'anneau obligeait à casser l'infrastructure du réseau, des défaillances techniques pouvaient faire « disparaître » le jeton du réseau, ou au contraire en créer plusieurs, *etc.*). Par la suite, le réseau s'est plus structuré en forme d'étoile, avec un commutateur central, gérant l'attribution des jetons aux nœuds lui étant associé. Plusieurs commutateurs pouvant être connectés en boucle pour agrandir le réseau. Les phases d'auto-négociation pour la régénération de jetons pouvait ainsi être géré de façon plus centralisée, et l'ajout de nœuds pouvait se faire plus facilement.

Les grands avantages de cette approche sont :

1. **Son déterminisme** : on peut parfaitement calculer le temps maximal d'attente pour un nœud avant de pouvoir émettre. Ceci est d'une grande utilité dans des réseaux où l'on doit avoir une garantie de temps de parole.
2. **Son passage à l'échelle** et l'occupation utile du réseau : le réseau est utilisé à 100% (hormis d'éventuels échanges liés à une régénération de jeton en cas d'incident technique) par des communications effectives, et la bande passante est divisée par le nombre de nœuds, et le temps de parole est inversement proportionnel à ce nombre. Ce n'est pas le cas dans une architecture à diffusion où, statistiquement, le taux de communications inefficaces augmente avec le nombre de nœuds, résultant en une dégradation de la bande passante effective avec le nombre de nœuds connectés.
3. **L'architecture en étoile** permet en plus la gestion de nœuds prioritaires et la détection de conflits ou dysfonctionnements, augmentant sensiblement la qualité de service ou la disponibilité du réseau.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

#### 2.2.4.2 Les réseaux à diffusion – Ethernet

Les réseaux à diffusion, couramment appelés CSMA/CD (*Carrier Sense Multiple Access/Collision Detect* – standard IEEE 802.3), sont résolument opposés aux principes structurés des réseaux à jeton. Ils reposent sur le principe d'un médium de communication partagé par tous les nœuds. Lorsqu'un nœud émet, le médium est occupé pour tout le monde (tout le monde reçoit les données, et personne ne peut émettre pendant ce temps).

Historiquement, le premier protocole CSMA était ALOHA, développé par Norm ABRAMSON de l'Université de Hawaï. Il utilisait la radio sur une plage de fréquences commune pour communiquer entre différentes îles de l'archipel. Par la suite, Robert Metcalfe de Xerox (Palo-Alto) à mis au point, vers 1970 la première version CSMA/CD qui est par la suite devenu Ethernet sous l'impulsion du consortium Digital Equipment Corp., Intel et Xerox.

Comme dans le cas de Token Ring, il convient de distinguer la version initiale de Ethernet (qui était *half-duplex*) et les améliorations futures, notamment avec l'introduction de commutateurs (la rendant *full-duplex*), qui corrigent un certain nombre de ses inconvénients.

Son principe de fonctionnement repose donc sur le partage d'un même médium de communication par tous les nœuds, la principale activité de régulation consistant à éviter ou à gérer des « *collisions* », c'est-à-dire des situations où plusieurs nœuds tentent d'émettre en même temps. Ces collisions créent des interférences et rendent le signal reçu par les autres inutilisable (FIG. 2.12).

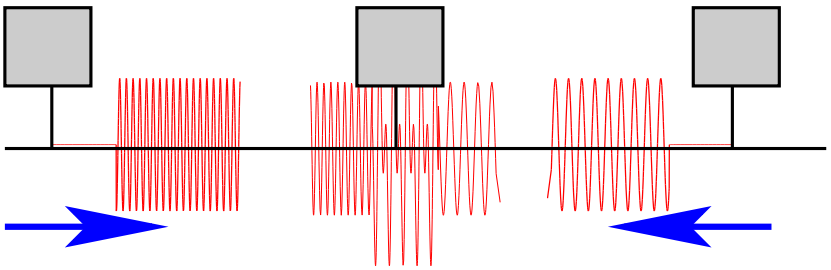

FIG. 2.12 – Interférence dans un réseau à diffusion : deux nœuds émettent un signal (extrémités). Le troisième nœud (milieu) reçoit la somme des deux signaux (*collision*).

Pour éviter cette situation, on utilise l'algorithme suivant :

- Lorsqu'un nœud veut transmettre, il écoute s'il entend une porteuse (*carrier*) sur le médium. La porteuse signale qu'un autre nœud va émettre ou est en train d'émettre.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

- En présence d'une porteuse, on attend que le médium se libère. Dans le cas contraire, on attend pendant un temps déterminé (*Interframe gap,* $9,6\mu s$).
- Si le médium est toujours libre après cette attente, il émet, sinon il recommence à surveiller la porteuse.
- Pendant l'émission, on vérifie si le signal reçu correspond bien à celui émis.
- S'il détecte une collision, il arrête de transmettre et attend un temps aléatoire avant de tenter de retransmettre. Avant de terminer réellement la transmission, en cas de collision, il compète sa trame avec une *jam sequence* (séquence de bits signalant une collision) qui garantira que :
  1. la suite de contrôle à la fin de la trame soit non conforme, signalant aux récepteurs que le message a été brouillé,
  2. la longueur de la trame corresponde bien à la longueur minimum imposée par le protocole.
- À la fin de la transmission, il attend pendant une durée fixe (*Roundtrip time*, $51,2\mu s$) avant de tenter de transmettre d'autres données.

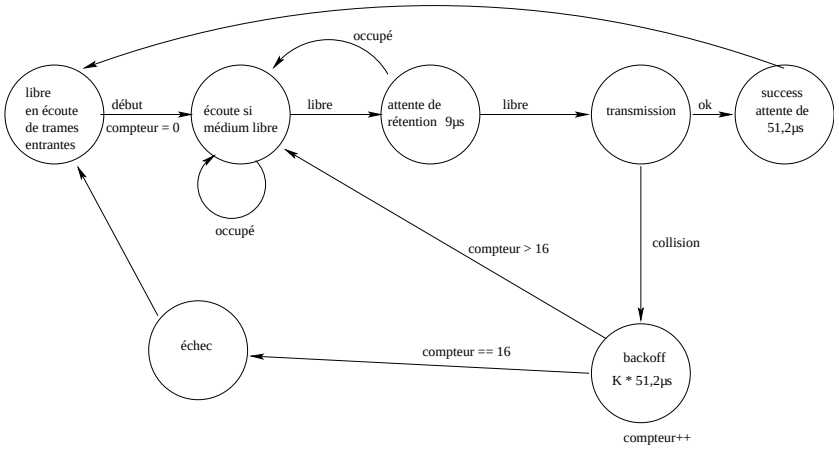


FIG. 2.13 – Automate de transition de l'algorithme CSMA/CD.

Tous les paramètres du protocole (longueur longueur minimale de la trame, *interframe gap*, *roundtrip time*, ...) ont été choisis avec soin pour différentes raisons, en fonction des données technologiques disponibles à l'époque de l'établissement de la norme.

Imposer une longueur minimale de 512 bits (ce qui correspond, par ailleurs, exactement à $51,2\mu s$ sur un réseau à 10Mbit/s, et donc au *roundtrip*) garantit qu'il est impossible d'avoir des *collisions* tardives ; c'est-à-dire que tous les émetteurs détecteront forcément une éventuelle collision pendant l'émission

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

de leur trame, et non pas après (en d'autres termes, le premier bit émis a le temps de faire un aller-retour sur le bus avant que le dernier bit ne soit émis, d'où, justement le terme de *roundtrip*). De fait, une trame émise sans collision détectée par l'émetteur est garantie de ne pas être brouillée par un autre.

L'*interframe gap* garantit que toutes les cartes réseau ont le temps de rebasculer de mode émission en mode réception, sans risque de « rater » une trame.

Il reste à noter que ces étalonnages ont été faits, dans les années 1970, avec les technologies de l'époque (Ethernet 10Mbit/s sur du câble coaxial). Les évolutions ultérieures des technologies sous-jacentes : câblages en paire torsadée permettant d'augmenter la taille du réseau et le débit à 100Mbit/s, câblages en fibres optiques permettant de dépasser 1Gbits/s, a mis à mal ces valeurs, sans pour autant les écarter, pour des raisons de compatibilité et de coexistence avec de vieux matériels.

Autre point important concerne l'attente aléatoire avant ré-émission en cas de collision. Si l'algorithme prévoyait une attente constante, les deux émetteurs qui avaient commencé à émettre en même temps attendraient le même laps de temps avant de recommencer, ce qui résulterait dans une nouvelle collision. Le protocole s'appuie sur le *retrait binaire exponentiel* (*exponential binary backoff*) qui gère de façon élégante les collisions multiples impliquant plus de deux émetteurs. Si une collision est détecté, on comptabilise le nombre de tentatives d'émission $i$, en l'initialisant à 1. On tire un nombre aléatoire $k$ dans l'intervalle $[0 \dots 2^i[$ et on attend pendant $k$ périodes de temps ; une période équivaut à un *roundtrip* : $51,2\mu s$. On tente ensuite de ré-émettre. Si une collision est détectée de nouveau, on incrémente $i$ et on recommence. À partir de 10 tentatives infructueuses, $k$ reste constante (1024). Au bout de 15 tentatives infructueuses, la trame est abandonnée et le réseau est considéré trop saturé.

| Préambule | Destination | Source | Type | Data | CRC |
|---|---|---|---|---|---|
| 8 octets | 2 ou 6 octets | 2 ou 6 octets | 2 octets | de 46 à 1 500 octets | 4 octets |

FIG. 2.14 – Format d'une trame Ethernet

La longueur d'une trame Ethernet est limitée à 1518 octets, et sa structure binaire est représentée dans FIG. 2.14. Il commence par un préambule de 64 bits, l'identifiant comme trame Ethernet (ceci permettant de partager le même médium entre différents protocoles ... Appletalk, par exemple), viennent ensuite l'adresse du destinataire (généralement l'adresse MAC) et

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

celui de l'émetteur. Le champ type est un code identifiant le type de données véhiculé (IP = 0x0800, ARP = 0x0806, ...[8]), viennent ensuite les données, et à la fin une somme de contrôle permettant de vérifier si la trame est bien complète et non brouillée. Comme le protocole impose un « silence » de $9,6\mu s$ il n'y a pas lieu de délimiter la fin de la trame par une séquence particulière.

Il est possible, sur un réseau à diffusion de type Ethernet, de s'adresser à tous les nœuds, plutôt qu'à un destinataire identifié, en mettant tous les bits du champ destinataire à 1. Cette adresse (tous les bits égaux à 1) est appelée adresse de diffusion ou *broadcast*. Tous les nœuds interceptent et considèrent ces trames comme si ils leur étaient destinés directement.

## 2.3 Internet et TCP/IP

**Note :** la famille de protocoles TCP/IP est de loin la plus développée dans ce cours. Il est à remarquer que ce n'est pas la seule approche aux réseaux existante. Il s'avère néanmoins qu'elle domine de loin les autres en termes de déploiement et utilisation, notamment du au fait que c'est une norme ouverte, tandis que la plupart de ses concurrentes sont généralement des solutions propriétaires. Ces types de protocoles sont développés dans une version future de ce document.

### 2.3.1 Historique et concepts de base

L'histoire d'Internet commence aux alentours des débuts des années 1960 en pleine guerre froide. Les unités d'études stratégiques du Pentagone américain, se posaient le problème suivant : « Comment s'assurer que les organes de décision (surtout militaires) américains puissent encore communiquer après une attaque nucléaire globale ? ». Quel qu'en soit le blindage et la protection physique, une attaque nucléaire ciblée réduirait à néant tout réseau de communication concevable, surtout qu'un tel réseau devrait forcément être contrôlé et piloté, rendant ce (ou ces) centre(s) de contrôle des cibles de choix en cas d'attaque.

Le principe d'un réseau répondant à ces besoins était proposé par la RAND Corporation[9] en 1964. Un tel réseau devrait être conçu dès le départ

[8] Il est à noter qu'ici Ethernet et la norme IEEE 802.3 sont incompatibles. Selon la norme IEEE ce champ indique la longueur de la trame, plutôt que son type

[9] La RAND était un des fameux « *think tanks* », laboratoires de recherche généreusement financés par le ministère de la défense américaine, à but de faire émerger des concepts scientifiques leur permettant de gagner une guerre thermo-nucléaire en cas de conflit avec l'URSS. C'est, entre autres, l'un des centres qui à donné naissance aux mathématiques de la décision, de recherche opérationnelle, de l'automatisation du contrôle aérien, *etc.*

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

comme étant défectueux et non fiable, et surtout, il ne devrait pas comporter d'autorité centrale qui le régule. De ce fait, tous les nœuds du réseau devraient être mis sur le même pied d'égalité, chacun émettant, transmettant ou recevant des données de son propre gré. Les données devraient être correctement étiquetés avec une origine et une destination, et étaient censés « trouver leur chemin » à travers ce réseau, sans que la route empruntée ait une quelconque importance, pourvu que les données arrivent à bon port. Les données passeraient d'un nœud à un autre selon des décisions locales de chacun d'entre eux, s'approchant progressivement de leur destination. Le fait qu'à un moment donné, une partie du réseau cesse d'exister, n'empêchera pas le reste de fonctionner.

La première tentative de réalisation d'un tel réseau eut lieu en Grande Bretagne en 1968, mais l'ARPA (Advanced Research Projects Agency, financé par le Pentagone) leur emboîta le pas en mettant en place un réseau formé de gros super-calculateurs (de l'époque) dans des sites de recherche universitaires. Le premier réseau, constitué de 4 machines et des liens de communication dédiés était constitué fin 1969 : ARPANET. Son but était d'expérimenter le contrôle, et l'utilisation des ordinateurs à distance. Grâce au réseau, les chercheurs pouvaient partager facilement de la puissance et du temps de calcul ... denrée rare à l'époque. Le réseau comptait 15 machines en 1971, et 37 en 1972. À ce moment là, les financiers du projet faisaient une constatation étrange ... le gros du trafic d'échange entre les différents nœuds ne concernait plus l'échange de données entre applications de recherche, mais était constitué de petits messages personnels. Parfois il s'agissait de échanges informels ou de discussions entre chercheurs travaillant sur des sujets communs, plus souvent les messages étaient d'ordre purement privé et extra-professsionnels. Qui plus est, c'était cette possibilité d'échange et de communication qui était plébiscité par ses utilisateurs, plutôt que les fonctionnalités initialement prévus.

Pendant les années 70, le réseau grandissait régulièrement. Sa conception, différente des réseaux propriétaires des grandes entreprises, le rendait indépendant d'un grand nombre de facteurs qui auraient pu freiner cette expansion : peu importait qui possédait les machines, quelle était leur marque, leur architecture ou leur système d'exploitation, pourvu qu'elles parlent le protocole du réseau. Et en plus, sa conception purement anarchique rendait tout contrôle d'expansion impossible. Il suffisait qu'un nœud autorise qu'un autre s'y connecte pour que ce dernier faisait partie intégrante du réseau global. Le protocole initial (NCP, « Network Control Protocol ») fut ensuite progressivement (entre 1975 et 1982) supplanté par un standard d'interconnexion mieux adapté : TCP/IP, permettant de s'affranchir complètement des architectures et protocoles utilisés sur les liens individuels entre nœuds. C'est

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

à ce moment là que l'on commençait à utiliser le terme d'*inter-net*, pour accentuer le fait que TCP/IP permettait d'*inter*connecter différents réseaux (*net*works) hétérogènes . Comme les standards utilisés étaient complètement ouverts, qu'il y avait une absence totale de contrôle de qui se branchait sur le réseau, et surtout, que cela ne coûtait rien à la communauté (chaque nœud supportant les frais de ses propres fonctions), rien n'était en mesure de s'opposer à l'expansion exponentielle du réseau (3000 en 1990, 137000 en 1990, 4 millions en 1995 ...).

### 2.3.2 Le modèle de couches de TCP/IP

TCP/IP utilise un modèle en couches comme le modèle OSI (il est important de souligner que le modèle OSI lui est très largement postérieur) et partage un certain parallèle avec ce dernier au niveau de la fonctionnalité de chaque couche (*cf.* TAB 2.2). En termes de modèle OSI, la famille de protocoles TCP/IP couvre les couches 2 à 5 : réseau, transport et session (partiellement). Un certain nombre d'applications connues mais ne faisant pas strictement partie de TCP/IP : telnet, ftp, smtp, ... se chargent des couches supérieures.

<table>
<tr><th colspan="2">Modèle OSI</th><th>TCP/IP</th></tr>
<tr><td>7</td><td>Application</td><td rowspan="2">Telnet | FTP | ⋮</td></tr>
<tr><td>6</td><td>Présentation</td></tr>
<tr><td>5</td><td>Session</td><td rowspan="2">TCP | UDP</td></tr>
<tr><td>4</td><td>Transport</td></tr>
<tr><td>3</td><td>Réseau</td><td>IP | ICMP</td></tr>
<tr><td>2</td><td>Liaison</td><td rowspan="2">ARP<br>pas de contraintes</td></tr>
<tr><td>1</td><td>Physique</td></tr>
</table>

TAB. 2.2 – Comparaison des couches TCP/IP avec le modèle ISO–OSI

Dans ce qui suit, nous aborderons en détail les protocoles de communication IP– Internet Protocol (couche réseau), UDP– User Datagram Protocol (couche transport) et TCP– Transmission Control Protocol (couche transport

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

+ session), ainsi que deux protocoles de support ICMP– Internet Control Message Protocol et ARP– Address Resolution Protocol.

### Guide de lecture

Un certain nombre de figures (dont, p. ex. FIG. 2.16) représentent le format des données du protocole qui circulent sur le réseau ; on parlera de trames, de paquets, de datagrammes, ... Ces tableaux représentent des canevas d'interprétation de leur signal binaire. C'est seulement à des fins de lisibilité et de présentation. Le signal lui même est bien évidemment une suite mono-dimensionnelle de `0` et de `1`, et il convient de lire ces tableaux dans ce sens :

1. Les tableaux se lisent de droite à gauche et de haut en bas. Sauf indiqué autrement, ils sont alignés sur 32 bits, et représentent des blocs de bits qui se suivent dans la transmission.
2. Les blocs de 32 bits sont sous-divisés en parties de longueurs diverses qui encodent telle ou autre caractéristique du paquet de données.

Par exemple, pour le format des datagrammes IP, p. 99, le tableau signifie que les premiers quatre bits encodent la version du protocole utilisée (ce qui signifie dont qu'on peut représenter $2^4 = 16$ versions différentes), que les quatre bits suivants représentent la longueur de l'entête, qu'en suite viennent les 8 bits de service, *etc.*

Il est également à noter que l'objectif ici n'est pas tant de donner les différents formats des entêtes de paquets/datagrammes/trames, mais de montrer en quoi un certain nombre de choix fonctionnels imposent de transmettre les informations en question (que l'on retrouve dans les entêtes) et comment leur présence ou non induit des comportements de fonctionnement ou des nécessités algorithmiques.

## 2.3.3 La couche réseau : IP – Internet Protocol

Rappelons d'abord les objectifs du protocole IP : la transmission de bout en bout d'un réseau de données quelconques, sans connaissance des particularités locales du réseau (principe d'encapsulation) ni de sa topologie, indépendance totale des contraintes physiques des liens de transmission individuels, intégrité des données, tolérance aux pannes éventuelles de nœuds intermédiaires, régulation locale (non centralisée).

La solution apportée par le modèle IP est de considérer le réseau global (Internet) comme un simple assemblage de réseaux locaux. Pour transiter de bout en bout du réseau, les données passent successivement d'un réseau local

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

à un autre par intermédiaire de *passerelles*[10]. Une passerelle est un nœud du réseau qui appartient à plusieurs réseaux locaux, et qui peut assurer le passage d'un réseau à un autre.

Si on reprend la FIG. 2.3, on peut facilement représenter en bleu les réseaux locaux et en rouge les passerelles (*cf.* FIG. 2.15). Il est important

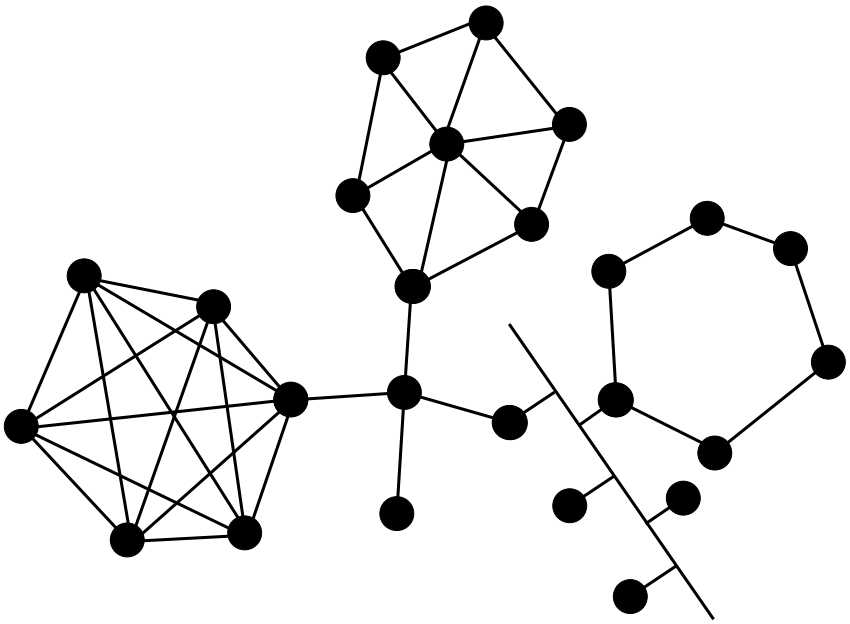

FIG. 2.15 – Réseau interconnecté.

de remarquer qu'une passerelle fait partie de *deux* réseaux (au moins) cela a une importance non négligeable dans la façon dont nous allons par la suite aborder les problèmes d'adressage et de routage dans le réseau. Rappelons également que dans la logique de conception, une passerelle est avant tout un nœud du réseau comme les autres il n'a de statut particulier que par le service qu'il rend.

#### 2.3.3.1 Adressage et classes de réseaux

Avant de commencer à répondre à la question « Comment trouve-t-on son chemin dans un réseau hétérogène de type IP ? », la premier point qui dont être abordé est : comment identifier le nœud auquel on veut envoyer des données ?

[10] *gateways* en anglais

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

La réponse est (au premier abord) d'une déconcertante simplicité : on attribue à chaque nœud du réseau un identifiant unique : un numéro binaire sur 32 bits (ce qui donne de l'ordre de 4 milliard d'adresses). Ce choix appelle immédiatement deux questions fondamentales :

1. En quoi cette numérotation permet-elle de déterminer où se trouve l'interlocuteur recherché (et par conséquent, comment permet-elle de trouver le chemin qui y mène) ?
2. Si on parle de numérotation unique, comment gère-t-on des conflits de numérotation, puisque, par conception le réseau ne doit pas dépendre d'une régulation centralisée ?

La réponse peut paraître surprenante, mais nous aborderons en détail les conséquences de ce choix élégant : on ne considère non pas des nœuds dans le réseau, mais des sous-réseaux interconnectés ; une adresse IP de 32 bits est considéré comme étant composé de deux parties, la première identifiant son réseau, la seconde identifiant le nœud en question au sein de son réseau local. La remise finale des données est laissée aux soins du gestionnaire du réseau destinataire, et le problème est réduit à l'acheminement de données d'un sous-réseau à un autre, plutôt qu'entre nœuds de tout le réseau interconnecté.

Une adresse IP de 32 bits est généralement représentée par quatre entiers de 8 bits (8 bits codent des nombres de 0 à 255 ou, en notation hexadécimale[11], de `00` à `FF`). Par exemple, l'adresse `193.252.71.61` peut aussi être écrit `B3 FB 47 C3` et correspond à la suite de bits `110000011 11111100 01000111 00111101`.

Selon les cas, les 8, 16 ou 24 premiers bits encodent l'adresse du réseau, les autres le nœud au sein du réseau. Ceci a pour effet de réduire la complexité d'identification de la destination d'approximativement $4\,10^9$ nœuds à seulement $2\,10^6$ réseaux. Afin de savoir quelle partie de l'adresse IP encode le réseau, la norme spécifie les règles suivantes :

Si le premier bit de l'adresse vaut 0, alors les 8 premiers bits encodent l'adresse du réseau, on parle de réseaux de classe A. Sinon, si le second bit vaut 0, les 16 premiers bits représentent l'adresse du réseau, et on parle de réseau de classe B. Sinon, avec le troisième bit valant 0 on est en présence d'un réseau de classe C, et les 24 premiers bits encodent son adresse. Sinon, on tombe dans des adresses, soit réservées à la multidiffusion ($4^{\text{ème}}$ bit valant 0), soit à des utilisations réservées non définies. Une adresse IP ayant tous ses bits à 1 signifie « tous les nœuds de ce réseau » et est appelé adresse de diffusion ou *broadcast*. Le tableau 2.3 récapitule les différents types d'adresses

[11] La notation hexadécimale est une représentation de nombres en base 16. Un nombre est représenté avec les chiffres 0 à 9 et les lettres A à F ; A valant 10 en base décimale, F valant 15. Le nombre $10_{\text{base } 16}$ vaut alors $16_{\text{base } 10}$.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

| Classe | Plage binaire | Plage décimale | Nombre | nœuds/réseau |
|---|---|---|---|---|
| Classe A | **00000001** 00000000 00000000 00000001<br>**01111111** 11111111 11111111 11111110 | **1**.0.0.1<br>**127**.255.255.224 | 126 | $2^{24} \approx 17\,10^6$ |
| Classe B | **10000000 00000000** 00000000 00000001<br>**10111111 11111111** 11111111 11111110 | **128.0**.0.1<br>**191.255**.255.224 | 16065 | $2^{16} = 65536$ |
| Classe C | **11000000 00000000 00000000** 00000001<br>**11011111 11111111 11111111** 11111110 | **192.0.0**.1<br>**223.255.255**.224 | 2031616 | $2^8 = 255$ |
| Multipoint | **11100000 00000000 00000000 00000000**<br>**11101111 11111111 11111111 11111111** | **224.0.0.0**<br>**239.255.255.225** | $\approx 25\,10^6$ | |
| Réservé | **11110000 00000000 00000000 00000000**<br>**11111111 11111111 11111111 11111110** | **240.0.0.0**<br>**255.255.255.254** | $\approx 25\,10^6$ | |
| Diffusion | **11111111 11111111 11111111 11111111** | **255.255.255.255** | 1 | |

TAB. 2.3 – Différentes classes de réseaux IP, identifiables à partir des premiers bits de l'adresse.

et les réseaux associés. Notons que le réseau de classe A avec l'adresse 127 est appelé non-routable. Cette adresse est réservée à des réseaux expérimentaux, et une passerelle recevant des données pour ce réseau est censé ne pas les transmettre (sauf configuration explicite). Parmi les adresses de ce réseau particulier, l'adresse 127.0.0.1 signifiant « moi-même ». Elle permet à une machine de s'envoyer des données à elle-même sans qu'elles transitent réellement par le réseau. Cette adresse est communément associé à l'interface virtuelle *loopback*.

En plus, outre l'adresse de diffusion déjà évoqué, il existe d'autres conventions. De ce fait, une adresse de nœud ne peut jamais être composé uniquement de 0 ou de 1. À l'instar du broadcast sur le réseau courant (255.255.255.255), une adresse de nœud compose uniquement de 1 correspond également à une diffusion à toutes les machines, mais alors sur le réseau spécifié dans l'adresse.

Les adresses multipoint servent à la diffusion 1-à-$n$ (par exemple la diffusion de programmes audio ou vidéo, où un émetteur s'adresse à un grand nombre de souscripteurs). Dans ce cas, au lieu d'établir des connexions individuelles avec chaque souscripteur (démultipliant $n$ fois la même donnée), l'émetteur émet sur une adresse multicast (ou multipoint), les passerelles ou nœuds intéressés récupèrent alors l'information, sans qu'elle circule inutilement en plusieurs exemplaires sur le réseau.

**Résumé :** les adresses IP sont des mots binaires de 32 bits, généralement regroupés dans 4 groupes de 8 bits. Elles sont divisés en groupes, selon la classe de réseau à laquelle elles appartiennent. On distingue, dans ces adresses, une partie identifiant le réseau, et une autre, identifiant le nœud

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

dans ce réseau. Cette structuration est appelée à évoluer, et la nouvelle norme IPV6 (*cf.* § 2.3.3.8 p. 107) prend progressivement la place de celle-ci.

**Remarques :** Suite aux remarques faites p. 94, il peut paraître absurde d'esquiver le problème de la régulation centralisée, en substituant des identifiants absolus par des identifiants réseau/nœud, puisque la question de fond demeure : comment éviter des conflits de nommage sans centralisation ? En effet on ne le peut pas vraiment. Une entité centrale, la *Internet Corporation for Assigned Names and Numbers (ICANN)*[12] coordonne depuis 1998 l'affectation des adresses des réseaux IPà qui en fait la demande. Cette organisation délègue à des instances régionales (RIPE pour l'Europe[13]). L'intelligence de l'approche réside principalement dans le découpage réseau/machine. L'acheminement est réduit à l'acheminement entre réseaux, les réseaux eux-mêmes étant responsables de la remise effective aux destinataires.

#### 2.3.3.2 Sous-réseaux

Le découpage en classes de réseaux a un inconvénient qui est la limitation arbitraire du nombre de machines connectés dans un réseau. Il est difficilement envisageable qu'un détenteur d'un réseau de classe A dispose d'un lieu contenant 17 millions d'ordinateurs connectés, et encore moins que toutes ces machines sont reliés à un unique réseau local.

Un administrateur de réseau (classe A–B–C) a la possibilité de définir des *sous-réseaux* au sein de sont réseau IP. À l'instar de ce qui se passe au niveau du protocole de base, avec les classes prédéfinies, un administrateur peut décider de redécouper sa plage d'adressage des nœuds en une partie d'identification de sous-réseau et une partie d'identification des nœuds. Il suffit pour ça de définir un masque qui sera superposé à l'adresse d'un nœud. L'opération logique **et** entre le masque et l'adresse donne le numéro du sous-réseau, la même opération avec le complément du masque donne le numéro du nœud dans ce sous-réseau.

Tous les problèmes d'acheminement, d'identification, *etc.* existant au niveau des réseaux IP, s'appliquent sans discrimination aux réseaux de base et aux sous-réseaux. Une seule règle prévaut, en général : l'acheminement se fait entre réseaux de même niveau. C'est au réseau destinataire de décider s'il délègue à un sous-réseau éventuel la remise finale.

[12] `http ://www.icann.org`
[13] `http ://www.ripe.org`

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

#### 2.3.3.3 La transmission et le routage

Il est clair que le problème majeur de la couche *réseau* concerne l'acheminement. Dans cette partie nous allons donner les éléments de base pour comprendre la solution proposée par le protocole IP. Il est nécessaire de souligner que ce qui est abordé ici n'est qu'une simple introduction des concepts de routage. Les algorithmes et protocoles ne cessent de s'affiner et de s'améliorer, mais l'analyse détaillée de tous leurs tenants et aboutissants est en dehors de la portée de ce cours.

Les données qui circulent dans un réseau IP, sont encapsulés dans un datagramme. Un datagramme contient une entête, portant les informations nécessaires à la remise à son destinataire d'une part, et les données elles-mêmes d'autre part. Le format du datagramme et le rôle des informations qu'il véhicule est détaillé dans § 2.3.3.4. Une chose fondamentale à garder à l'esprit est que le contenu véhiculé est complètement transparent pour la couche réseau et les seules informations utilisées pour l'acheminement et la remise, se trouvent dans l'entête du datagramme[14].

**La transmission** des datagrammes suit un algorithme extrêmement simple. Un nœud qui veut transmettre un datagramme analyse l'adresse de son destinataire. Si ce dernier se trouve sur le même brin de réseau que lui (*i.e.* la partie réseau de sa propre adresse et celle de son destinataire est la même) le datagramme est remis directement à son destinataire via la couche liaison, puisque, par construction même, ils partagent le même médium de liaison. Si le destinataire ne fait pas partie du même réseau que l'émetteur, l'émetteur remet le datagramme à un *routeur* qui se charge de l'acheminement.

Un routeur est un nœud du réseau qui a les propriétés suivantes :

- Il partage le même médium de liaison avec l'émetteur (la remise peut donc se faire directement par la couche liaison)
- C'est un nœud de jonction entre différents réseaux. De ce fait, il fait partie de différents réseaux locaux[15] (et dispose donc d'un nombre d'adresses IP en conséquence !)

Le routeur devient ensuite l'émetteur délégué du datagramme, et l'algorithme précédent s'applique à nouveau, jusqu'à la remise finale des données.

[14] Le parallèle avec la Poste est parfaitement exploitable dans ce contexte. Un datagramme est assimilable à une lettre. Le contenu de la lettre est inaccessible et inutile pour sa remise au destinataire.

[15] Il est important de ne pas confondre la différence entre routeur et passerelle, bien qu'il soit possible qu'une seule machine remplisse les deux fonctions. Un routeur est connecté à plusieurs réseaux. Une passerelle est connecté à différents média de liaison. Il n'est pas obligatoire qu'un routeur soit en même temps passerelle, ou *vice versa*.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

| Réseau | Masque | Routeur | Métrique | Interface |
|---|---|---|---|---|
| 127.0.0.0 | 255.0.0.0 | * | 0 | lo |
| 192.168.123.0 | 255.255.255.0 | * | 0 | eth0 |
| 152.81.0.0 | 255.255.240.0 | 192.168.123.10 | 5 | eth0 |
| default | 0.0.0.0 | 192.168.123.12 | 0 | eth0 |

TAB. 2.4 – Exemple de table de routage

**Le routage** des datagrammes se fait donc de proche en proche, en les faisant transiter de routeur en routeur, jusqu'à leur destination finale. Les routeurs sont en cela différents des nœuds ordinaires, qu'ils disposent d'informations permettant d'acheminer les datagrammes qu'ils reçoivent. En effet, ils s'appuient sur des *tables de routage* pour prendre des décisions quant au prochain routeur qui devra prendre en charge le datagramme qu'il doit transmettre. Une table de routage se présente sous la forme d'un tableau, comme, p. ex. TAB. 2.4.

Globalement il contient les informations suivantes : le réseau qu'on cherche à atteindre (ou `default`) et le *netmask* qui permet de l'extraire à partir de l'adresse IP, le routeur qui transmettra, la distance à laquelle se situe le dit réseau, et l'interface par laquelle on atteint le routeur spécifié. Le champ routeur peut valoir * auquel cas, la remise est censée être directe (*i.e.* l'émetteur et le récepteur partagent la même liaison). Pour chaque transmission de datagramme, l'adresse du destinataire est confronté à la table de routage, et transmis au premier routeur pour lequel l'application du masque de réseau à cette adresse donne l'adresse de réseau correspondant, via l'interface réseau spécifiée.

**Exercice :** Sous Linux, la commande `route` permet d'afficher l'état de la table de routage.

Le routage dans un réseau non supervisé (ou peu supervisé) comme un internet est un problème très complexe, dont nous aborderons quelques points dans la section § 2.3.2, p. 91. Indiquons seulement ici que pour acheminer un datagramme, un routeur consulte sa table de routage, en comparant l'adresse réseau du destinataire aux réseaux qui lui sont connus. Si le réseau en question figure dans la table de routage, le datagramme est transmis au router correspondant, sinon, il est transmis au router par défaut, s'il existe. S'il n'existe pas, l'émetteur peut être informé par un message ICMP de la non transmission du datagramme, et le datagramme est détruit.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

**La remise** du datagramme, par contre, n'est (volontairement) pas garantie par cette approche. Un grand nombre d'incidents peuvent faire qu'un routeur ne puisse transmettre un datagramme au suivant (ou au destinataire final) : défaillance d'un nœud, défaillance de la transmission du signal, congestionnement du réseau, dépassement de capacités des files d'attentes, mauvaise configuration des tables de routage, ... Dans certains cas, il est prévu qu'un message est renvoyé à l'émetteur (*cf.* la section § 2.3.3.7 sur le protocole ICMP), dans d'autres cas, le datagramme est tout simplement détruit, sans que l'émetteur puisse en être informé.

#### 2.3.3.4 Le format d'un datagramme IP

Afin de réaliser tous les objectifs (identification des parties communicantes, acheminement, ...) décrits dans les sections précédentes, les données transmises sont encapsulées dans une « enveloppe » électronique. Elle comporte un certain nombre d'informations supplémentaires permettant d'acheminer les données qu'elle contient, en respectant strictement le principe d'encapsulation. Pendant toute la transition dans le réseau, seules les informations contenues dans l'entête de cette enveloppe sont utilisées pour prendre les décisions nécessaires à l'acheminement.

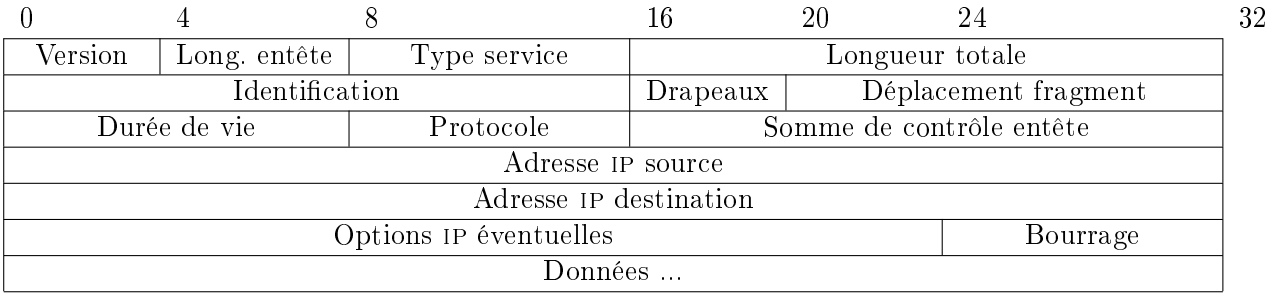


FIG. 2.16 – Format d'un datagramme IP.

Comme représenté dans FIG. 2.16, l'entête d'un datagramme IP est systématiquement aligné sur 32 bits et comporte les champs suivants :

**Version** est une valeur sur 4 bits représentant la version du protocole utilisée. La seule version vraiment opérationnelle du protocole est la version 4. La nouvelle version d'IP, IPv6, utilise un format d'entête différent.

**Longueur de l'entête** est une valeur sur 4 bits donnant la longueur totale de l'entête et mots de 32 bits. Comme l'entête peut contenir des parties optionnelles, cette valeur est nécessaire pour déterminer l'endroit ou commencent les données encapsulées.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

**Type de service** est un champ prévu par la norme, pour identifier certains degrés de qualité de service. En réalité il est quasiment systématiquement ignoré par les différents routeurs. C'est une valeur sur 8 bits, dont les trois premiers représentent la valeur de priorité (de 0 à 7) du datagramme. Les trois suivants sont appelés D, T et R, pour *Delay*, *Transfer* et *Reliability*. Ce sont des drapeaux qui, lorsqu'ils sont mis à `1`, demandent le datagramme soit transmis via un chemin qui, selon le drapeau, fournit un délai de transmission court, un taux de transfert, ou une fiabilité élevés. Les deux derniers bits ne sont pas utilisés.

Ces types de service étaient prévus pour permettre de gérer des urgences (p. ex. dans le cas de congestionnement du réseau, ou de reconfiguration brutale) ou encore de fournir des services selon les besoins d'une application si un routeur connaît plusieurs chemins (avec leurs propriétés) pour atteindre une destination. Bien que très étudiés sur le plan théorique, ces types de service n'ont jamais été réellement utilisés. Dans la version 6 d'IP, la notion de qualité de service est alors revu en conséquence.

**Longueur totale** est la valeur sur 16 bits de la taille totale du datagramme (entête + données) en nombre d'octets. Par conséquent, un datagramme IP ne peut dépasser $2^{16} - 1 = 65535$ octets.

**Identification** , comme son nom, l'indique, identifie de façon unique le datagramme qui va de `source` à `destination`. Cette valeur permet, à la fois de gérer des doublons qui pourraient être générés par le réseau, et, en combinaison avec le champ *drapeaux* et *déplacement de fragment*, de résoudre le problème de réassemblage de fragments[16]. En cas de fragmentation du datagramme, la valeur d'*Identification* est préservée.

**Drapeaux** est formé de 3 bits :

- Un *bit de non-fragmentation*, qui interdit que le datagramme soit fragmenté pour être transmis.
- Un bit indiquant des *fragments à suivre*, afin que le destinataire puisse savoir s'il a reçu toutes les parties d'un datagramme fragmenté.

**Déplacement du fragment** est une valeur sur 13 bits indiquant (en multiples de 8 octets !) la position du fragment par rapport au début du datagramme d'origine (pour plus de détails, référez-vous à la section 2.3.3.5).

**Durée de vie** indique le nombre de *hops* qu'un datagramme peut faire (*i.e.* le nombre de routeurs qu'il peut franchir) avant d'être détruit. Chaque

[16] La fragmentation et le réassemblage de fragments sera traité à part dans la section 2.3.3.5

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

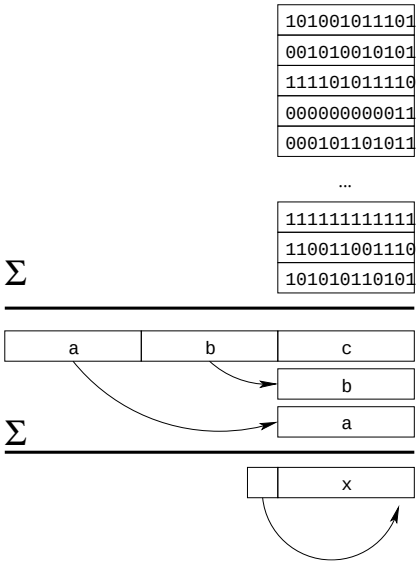


FIG. 2.17 – Principe du calcul d'une somme en complément à 1.

routeur, décrémente cette valeur au passage du datagramme, et évite ainsi que des datagrammes *fantômes* continuent indéfiniment à circuler sur le réseau. Un router recevant un datagramme dont la durée de vie vaut 0 et qui ne peut pas le remettre directement au destinataire, le détruit (et renvoie éventuellement une notification ICMPà l'émetteur).

**Protocole transporté** permet d'identifier la couche immédiatement supérieure (généralement la couche transport) à qui le contenu du datagramme doit être transmis, sans violer le principe d'encapsulation, puisque la couche IP n'a pas besoin d'aller voir le contenu à proprement dire.

**Somme de contrôle de l'entête** (*checksum*) sur 16 bits. Elle est obtenue en considérant l'entête comme une suite de mots à 16 bits, et en calculant leur somme selon l'algorithmique à complément à 1, et de prendre le complément à 1 du résultat

**Exemple 2.1 : Algorithme de calcul de** *checksum*

La somme d'un ensemble de mots en algorithme à complément à 1 est le suivant : on suppose que tous les mots sont stockés sur une même longueur (p. ex. 16 bits) le résultat devra être stocké sur un mot de cette longueur. Bien évidemment, on n'a aucune garantie que la somme réelle tienne dans ce mot. On prend donc tous les bits qui débordent, et on les rajoute en les décalant de la longueur du mot (*cf.* FIG. 2.17).

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

```
/* Hypothèse : short = 16 bits
               int = 32 bits
               long = 64 bits */
unsigned short cksum(unsigned short *buf, int taille)
{
  /* la variable somme contiendra la somme de tous les mots
     du tampon buf, on le déclare unsigned long pour être sûr
     de pouvoir tout stocker */
  unsigned long somme;

  /* Calcul de la somme */
  for( somme = 0; taille > 0; taille--)
     somme += *buf++;

  /* somme & 0xffff = les 16 bits de poids faible */
  /* somme >> 16 = somme décalé de 16 bits, équivalant
                   aux bits de poids fort restants */
  somme = (somme >> 16) + (somme & 0xffff);

  /* au cas où les bits de poids fort restants dépassent
     16 bits il faut répéter l'opération.
     Comme taille est codé sur 32 bits on sait qu'après,
     il n'y en a plus */
  somme += (somme >> 16);

  /* retour du complément à 1 */
  return ~(somme & 0xFFFF);
}
```

**Adresse IP de la source** sont les 32 bits de l'adresse de l'émetteur. Cette valeur n'est pas modifiée par le routeurs.

**Adresse IP de la destination** sont les 32 bits de l'adresse du destinataire. Cette valeur n'est pas modifiée par le routeurs.

**Options IP éventuelles** est un champ qui permet(tait) surtout aux développeurs et administrateurs de vérifier ou d'analyser le comportement d'un réseau. Les champs des options suivent une syntaxe particulière assez simple, qui n'est pas utile de développer ici. Notons, parmi les options, l'existence d'horodatage (chaque router note l'heure de passage), d'enregistrement de la route (chaque routeur intermédiaire rajoute son adresse au passage du datagramme) ou le routage définie par la source

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

(l'émetteur spécifie la route, et chaque routeur dépile l'adresse du router suivant, et lui transmet le datagramme ... cette option est connue pour être source d'ennuis de sécurité)

**Bourrage** sont des bits de remplissage, permettant d'aligner l'entête du datagramme sur 32 bits, étant donné que les options éventuelles peuvent être d'une longueur indéterminée.

#### 2.3.3.5 La fragmentation des datagrammes

La fragmentation des datagrammes est un problème lié au fait que le principe de TCP/IP est de vouloir interconnecter toutes sortes de réseaux locaux, avec des protocoles de liaison différents.

Si on considère, par exemple, Ethernet, on voit qu'une trame, dans laquelle serait encapsulée un datagramme IP, ne peut, dans sa totalité, dépasser 1518 octets (p. 88). Or, un datagramme IP n'est pas limité à cette longueur-là, mais peut aller jusqu'à 65535 octets (p. 100).

Supposons donc une communication entre un émetteur, utilisant un type de liaison permettant de générer des datagrammes de longueur optimale, et un récepteur utilisant un réseau local quelconque, et que les datagrammes transitent par une liaison de type Ethernet. Sans considération aucune des réseaux intermédiaires, l'émetteur transmettrait ses données, mais ils ne pourraient jamais être remis au destinataire, pour cause de limitation d'un chaînon intermédiaire (le réseau Ethernet qui est incapable de transmettre des datagrammes de 65535 octets). On pourrait envisager une exploration du réseau préalable à toute communication, qui calculerait la taille maximale de transfert (MTU : *Maximal Transfer Unit*). C'est lourd à mettre en œuvre, ça crée une surcharge très significative du réseau, et c'est peu fiable, puisque le réseau peut changer de chemin de routage à tout moment.

Le principe adopté par le protocole IP est donc de laisser le soin à chaque routeur de déterminer si sa couche liaison est capable de transmettre le datagramme. Si ce n'est pas le cas, il le *fragmente*. C'est-à-dire qu'il le coupe en morceaux de taille convenable, et les transmet ainsi. On laisse le soin au destinataire final de réassembler les morceaux (en aucun cas un router intermédiaire se charge de cette tâche ... il a déjà bien assez à faire).

Pour permettre au destinataire d'accomplir ce réassemblage, il lui faut un certain nombre d'informations. À chaque arrivée de datagramme, il doit être capable de déterminer si :

- c'est un fragment de datagramme sectionné. C'est le deuxième bit du champ *Drapeaux* qui le lui dit, en combinaison avec le champ *Déplacement du fragment*. Si l'un des deux est différent de 0, il s'agit d'un fragment de datagramme plus grand.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

– tous les fragments d'un même datagramme sont arrivés. C'est la combinaison des champs *Identification*, *Fragments à suivre* (bit 2 de *Drapeaux*) et l'offset (*Déplacement du fragment*) qui lui donnent cette information.

A chaque arrivée de fragment, le réceptionnaire arme une temporisation. Si à l'échéance de celle-ci le datagramme d'origine n'est pas complètement réassemblé, celui-ci est considéré comme perdu (et un message ICMP est éventuellement envoyé à l'envoyeur).

#### 2.3.3.6 ARP – Address Resolution Protocol

Dans la section précédente nous avons vu que le protocole IP permet (par l'intermédiaire du masque de réseau) de savoir si un datagramme peut être remis directement à une machine sur le réseau local, ou s'il doit être remis à un routeur qui le fera suivre. Dans tous les cas, la couche liaison reçoit de la couche IP l'ordre de remettre une quantité de données à une machine bien identifiée sur le réseau local (que celle-ci soit routeur ou non, importe peu pour elle à ce niveau).

Deux questions fondamentales se posent alors :

1. Comme l'adressage IP est purement virtuel (une adresse dénote un point d'entrée sur le réseau, mais non pas une entité physique. De plus, dans un contexte d'évolution d'un réseau, des machines peuvent changer d'adresse IP sans préavis) et qu'il n'y a, *a priori*, aucune raison qu'il y ait une correspondance entre l'adressage au niveau liaison (Ethernet, Tokenring) qui elle est intrinsèquement lié au matériel déployé, quelle est le moyen le plus réaliste pour assurer une correspondance entre ces deux adresses ? En termes plus clairs, comment faire savoir à la couche IP quelle adresse de la couche liaison elle doit utiliser pour atteindre la machine souhaitée sur son brin de réseau local.
2. Un corollaire direct de l'interrogation ci-dessus concerne la garantie de l'encapsulation stricte imposée par le modèle en couches et le souhait affiché de s'affranchir de toute contrainte de mise en œuvre de la couche liaison : Comment garantir que le protocole IP se greffe de façon transparente sur une quelconque couche liaison (Ethernet, Tokenring, ...) ?

La solution vient du protocole ARP. Dans le tableau 2.2 ce protocole est représenté comme faisant partie de la couche liaison, bien qu'en réalité il se trouve plutôt à l'écart de la hiérarchie.

Son principe de fonctionnement est simple. Lorsque la couche réseau cherche à identifier un correspondant partageant la même liaison avec lui et

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

<table>
<tr><td colspan="8">0 8 16 24 32</td></tr>
<tr><td colspan="4">Type réseau</td><td colspan="4">Type d'adresse protocole</td></tr>
<tr><td colspan="2">Long. adr. phys.</td><td colspan="2">Long. adr. prot.</td><td colspan="4">Opération</td></tr>
<tr><td colspan="8">Adr. Ethernet source (4 premiers octets)</td></tr>
<tr><td colspan="4">suite Ethernet source (2 derniers octets)</td><td colspan="4">Adr. IP source (2 premiers octets)</td></tr>
<tr><td colspan="4">suite IP source (2 derniers octets)</td><td colspan="4">Adr. Ethernet dest. (2 premiers octets)</td></tr>
<tr><td colspan="8">suite Ethernet dest. (4 derniers octets)</td></tr>
<tr><td colspan="8">Adresse IP destinataire</td></tr>
</table>

FIG. 2.18 – Format d'un datagramme ARP pour la liaison Ethernet–IP.

dont il ne connaît que l'adresse réseau `@R` (IP, par exemple). Il interroge le service ARP, qui gère une table de correspondance entre adresses liaison/réseau. Soit le correspondant est connu, et enregistré dans la table, et le service ARP répond, soit il ne l'est pas. Dans ce cas, le service construit un paquet ARP et l'envoie en *broadcast* via la couche liaison. Toutes les machines reçoivent donc une requête ARP du type « *Qui est* `@R` *?* ». Celle qui connaît la réponse, répond à l'émetteur « `@R` *a l'adresse liaison* `@L` *!* », et le service ARP transmet à la couche réseau, qui peut alors correctement construire sa trame liaison pour transmettre son datagramme.

Plus formellement, ARP construit un paquet comme, p. ex. celui représenté dans FIG. 2.18. Les deux premiers champs, *Type liaison* et *Type réseau*, contient des codes des types d'adresses dont on cherche une correspondance (p. ex. le code Ethernet correspond au type de liaison `1`, le code `0x0800` au type de réseau IP). L'*Operation* indique s'il s'agit d'une demande ou réponse, ARP ou reverse ARP (RARP). Viennent ensuite la longueur des adresses pour chaque champ, suivi des adresses liaison et réseau de l'émetteur et du récepteur respectivement. Lorsque l'émetteur fait sa demande, il remplit tous les champs sauf celui dont il veut connaître la valeur (dont il met tous les bits à `0`). La réponse aura tous les champs remplis.

**Exercice :** Sous Linux, la commande `arp` permet d'afficher l'état de la table de correspondance ARP. Notez qu'elle est périodiquement nettoyée.

**Note :** une machine qui reçoit une requête ARP à laquelle elle ne peut pas répondre, peut quand-même prendre soin de remettre à jour sa table avec les informations contenues dans la requête ; c'est-à-dire, la correspondance liaison/réseau de son émetteur (ce comportement doit toutefois être considéré avec soin, puisqu'elle peut être une faille de sécurité). De plus, si pendant

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

un certain temps aucune communication avec un élément dans la table n'est observée, celui-ci en est retiré.

#### 2.3.3.7 ICMP – Internet Control and Message Protocol

Autant ARPétait un protocole intermédiaire entre Ethernet et IP, encapsulé directement dans les trames liaison, le protocole ICMP est construit au-dessus de IP et est encapsulé dans des datagrammes IP. Le format du paquet ICMP est représenté dans FIG. 2.19.

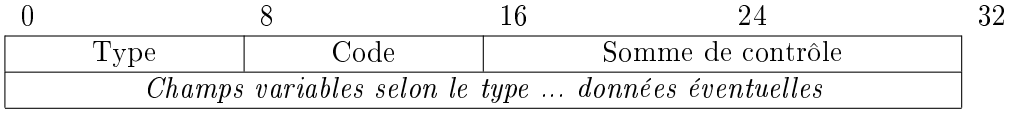


FIG. 2.19 – Format d'un datagramme ICMP.

ICMP sert principalement comme protocole d'échange d'informations sur l'état du réseau, et est utilisé par les différents routeurs pour informer l'expéditeur d'un datagramme qu'un problème est apparu en cours de route (congestionnement, routeur indisponible, adresse inconnue, ...) ou pour informer d'autres routeurs d'un changement de topologie du réseau.

Le message est identifié d'après les deux premiers champs du datagramme : `Type` et `Code`. Il peut éventuellement être complété par des données supplémentaires dans la partie dédiée aux données. Parmi quelques types de messages on a : demande et réponse d'écho (Types 0 et 8), destination inaccessible (Type 3), changement de route (Type 5), expiration de délai (Type 11), demande et réponse d'horodatage (Types 13 et 14), *etc.* Chaque type est ensuite détaillé selon la valeur du code, qui varie en fonction du type dont il dépend.

**Exemple 2.2 : La commande `ping`**

La commande Posix, `ping` envoie des paquets ICMPà un destinataire avec une demande d'écho. Une demande d'écho (Type ICMP 0) consiste à envoyer des données aléatoires au destinataire avec comme requête de les renvoyer en l'état. Ceci permettant alors de mesurer les temps de transit dans le réseau entre deux nœuds et d'avoir une idée de la qualité des transmissions entre les paquets envoyés et ceux reçus correctement.

```
Cesar <1> ping aladin
PING aladin.mines.inpl-nancy.fr (193.49.140.27): 56 data bytes
64 bytes from 193.49.140.27: icmp_seq=0 ttl=255 time=0.7 ms
64 bytes from 193.49.140.27: icmp_seq=1 ttl=255 time=0.1 ms
```

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

```
64 bytes from 193.49.140.27: icmp_seq=2 ttl=255 time=0.2 ms
64 bytes from 193.49.140.27: icmp_seq=3 ttl=255 time=0.2 ms
64 bytes from 193.49.140.27: icmp_seq=4 ttl=255 time=0.2 ms
64 bytes from 193.49.140.27: icmp_seq=5 ttl=255 time=0.2 ms

--- aladin.mines.inpl-nancy.fr ping statistics ---
6 packets transmitted, 6 packets received, 0% packet loss
round-trip min/avg/max = 0.1/0.2/0.7 ms
```

**Exercice :** La commande `traceroute` permet d'identifier les routeurs intermédiaires entre deux nœuds. Elle se base entièrement sur des mécanismes ICMP. Trouvez lesquels. Est-ce fiable ? Pourquoi, pourquoi pas ?

#### 2.3.3.8 IP – V6

La nouvelle version du protocole IP, IPv6, existe déjà, et commence à être utilisé en dehors des laboratoires de recherche. C'est une évolution naturelle de la norme (bien qu'elle change beaucoup de choses en profondeur) pour permettre de répondre au succès grandissant d'Internet et ses protocoles. Tous les principes de base, à l'origine d'IP sont conservés, mais les choix d'implémentation et de simplification en font une norme complètement remodelée.

Ses objectifs principaux sont :

1. Décongestionner l'espace d'adresses, actuellement dangereusement proche de la saturation.
2. Alléger le travail des routeurs, permettant ainsi de diminuer leurs temps de réponse.
3. Intégrer la notion de *Qualité de Service* (QoS – *Quality of Service*) pour permettre à des applications multimédia, notamment, de pouvoir réserver une bande passante garantie.

A ces fins, le contenu de l'entête s'est beaucoup simplifié (*cf.* FIG. 2.20). On y trouve le même champ de version que dans la version 4, pour permettre aux routeurs de distinguer entre les deux. Ce qui suit, en revanche est radicalement différent.

D'abord, on notera que les adresses sont maintenant codées sur 128 bits, générant ainsi un pool suffisamment confortable pour les années à venir. On notera la disparition des champs de contrôle et de fragmentation, principales

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

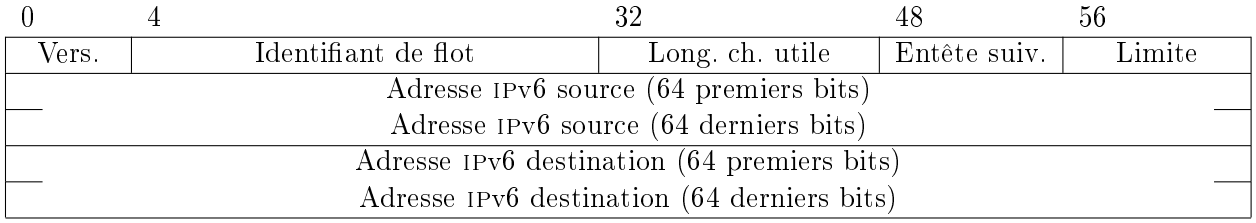


FIG. 2.20 – Format de l'entête de base IPv6 (alignement sur 64 bits).

sources de surcharge des routeurs. La validation du contenu est reléguée aux protocoles de plus haut niveau, et compte-tenu de la stabilité du réseau, la fragmentation est supposée être rendue obsolète, par des algorithmes de calcul de PMTU (Path Maximum Transfer Unit). C'est donc l'émetteur qui est censé fournir des datagrammes de longueur adéquate.

Comme, par construction même du réseau, il est toujours possible qu'un routeur tombe en panne et qu'un autre chemin (avec un autre PMTU) doit être emprunté au milieu d'une transmission, la norme permet à un routeur de créer un *tunnel* IPv6 dans lequel il encapsule les datagrammes d'origine, découpés pour satisfaire le nouveau PMTU. Il exploite pour cela la fonctionnalité d'IPv6 de rajouter des entêtes optionnels.

Comme nous l'avons déjà indiqué, l'entête principale de IPv6 est volontairement très simple. Il reste néanmoins des situations où l'on aurait besoin de plus d'informations. C'est dans ce but qu'existe le champ `Entête suivant`. En effet, le protocole prévoit qu'un datagramme puisse contenir, entre l'entête principale et les données véhiculées une liste chaînée d'entêtes supplémentaires. C'est notamment dans ces entêtes qu'un routeur peut stocker de l'information relative à la fragmentation d'un datagramme en cas d'établissement d'un tunnel. Par ailleurs, ce champ sert également d'identification du protocole encapsulé, lorsqu'il s'agit de la dernière entête de la liste chaînée d'entêtes.

Le champ `Limite` de l'entête principale est équivalent à la durée de vie du protocole IPv4, mais est maintenant exclusivement exprimé en sauts de routeur. La `Longueur de charge utile` exprime la taille des données hors entêtes. Restent donc à détailler le champ `Identifiant de flot` et le système d'adresses.

**La *Qualité de Service*** (QoS) est un concept allant largement au-delà de ce couvre le champ `Type de service` dans la version actuelle d'IP. En effet, il est remplacé par un `Identificateur de flot`, qui est lui même structuré en deux parties.

Les quatre premiers bits indiquent la catégorie de données qui est trans-

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

portée, et plus particulièrement leur sensibilité à un changement de débit. Typiquement, les transmissions comme l'envoi d'e-mails, le téléchargement de fichiers, *etc.* ne nécessitent pas un débit constant, tandis que des applications multimédia (flots de vidéo, téléphonie) ou des applications de téléguidage d'appareils ne supportent que très moyennement des diminutions de taux de transfert. Ces premiers bits quantifient cette sensibilité. Le reste est un numéro de flot associé, donné par l'émetteur, permettant aux routeurs intermédiaires, de classer les différents datagrammes d'un émetteur donné par flot, et de donner une qualité de service constante à chaque flot (voire même de privilégier certains flots, s'ils ont demandé – et payé – une qualité de service supérieure).

**Le système d'adresses** n'a plus rien à voir avec ce qui existait dans le protocole précédent. À l'instar des classes de réseau, utilisés en IPv4, les 3 à 8 premiers bits d'une adresse IPv6 véhiculent une information sur le type d'adresse (bien qu'il ne s'agit plus de classes de réseaux). On identifie ainsi 22 groupes d'adresses, dont la majorité sont réservés pour utilisation et définition ultérieure, et qui ont les préfixes suivants :

| Préfixe (en binaire) | Type |
|---|---|
| 0000 0000 | Encapsulation des adresses IPv4 existantes. |
| 010 | Adresses unicast affectés par des fournisseurs. |
| 100 | Adresses géographiques |
| 1111 1110 | Adresses non-routables (pour utilisation locale) |
| 1111 1111 | Adresses multicast |

Ce qui donne, globalement, 27% du pool d'adresses total, qui sont disponibles qui peuvent être utilisés dans le futur proche. Par ailleurs, des règles d'encodage hiérarchique des adresses (codes par fournisseur, puis sous-fournisseurs délégués, puis administrateurs locaux) permettront d'alléger la tâche des routeurs, de diminuer la taille des tables de routage, et d'améliorer le service de transmission.

### 2.3.4 Les couches Transport et Session

Jusqu'à présent nous avons abordé le problème de la remise de données à travers un réseau internet. La couche réseau, le protocole IP, répondent à cet objectif. Reste maintenant son utilisation par des applications communicant à travers ce réseau et leurs besoins éventuels qui ne sont pas couverts par IP. En voici quelques uns en vrac :

– Le protocole ICMP peut signaler d'éventuels problèmes de remise, mais

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

dans de nombreux cas, des datagrammes peuvent ne pas arriver à destination, sans que l'émetteur en soit informé.
- Par construction du protocole IP, le chemin de parcours n'est pas garanti, et donc par conséquent, ni le temps de transfert de bout en bout. Il en résulte que l'ordre d'arrivée des données n'est pas garantie (c'est surtout le cas lorsqu'on veut envoyer des données d'une taille supérieure à la taille maximale d'un datagramme IP, qui est, rappelons-le de 65536 octets, *cf.* p. 100).
- Dans le cas où plusieurs applications sur deux machines veulent communiquer entre eux, comment distinguer le destinataire (*i.e.* l'application, ou le processus) au-delà de la simple adresse IP de son hôte ?
- Comment vérifier si une application est prête à recevoir des données, et quelle a la capacité de traitement pour gérer les données qui lui sont envoyées.
- Comment optimiser la vitesse d'émission (trop élevée, elle peut causer la perte de données, si les intermédiaires ou le destinataire sont sous-dimensionnés, trop basse, elle sous-exploite les capacités de traitement de chacun)

Tous ces points doivent être transparents pour les applications qui souhaitent communiquer, et doivent donc être gérés par le protocole de la couche Transport (voire Session) à laquelle elles s'adressent. La famille de protocoles TCP/IP offre deux approches : UDP, la communication en mode déconnecté, et TCP, en mode connecté. Avant de les aborder en détail, nous évoquerons le moyen qu'ont les applications pour s'identifier au-delà de l'adresse IP de leur hôte.

#### 2.3.4.1 Les ports de communication

La solution est assez simple. Une interface IP est enrichie de $2^{16}$ *ports*, de 0 à 65535. Chaque processus qui souhaite communiquer se voit allouer (dynamiquement ou statiquement) par le système d'exploitation un numéro de port qu'il est le seul à utiliser, le temps de la communication. C'est le couple adresse IP + numéro de port, qui identifie de façon unique l'interlocuteur. Par convention, les ports de 0 à 1024 sont réservés au système d'exploitation. Les autres sont libres d'utilisation.

**Exercice :** Sous Unix ou Linux, les ports réservés au système, et les services associés sont répertoriés dans `/etc/services`.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

#### 2.3.4.2 Encapsulation dans IP

Bien évidemment, toute cette information est encapsulée dans la partie `Données` du datagramme IP. Selon le protocole de transport retenu (UDP ou TCP), on renseigne le champ `Protocole` du datagramme. La partie `Données` contient ensuite l'entête propre du protocole encapsulé (lequel, comme on le verra, contient le port de l'émetteur et celui du destinataire) plus les données effectivement échangées entre les applications. Lors de la réception d'un datagramme IP, le module de gestion réseau du système d'exploitation extrait l'entête et les données encapsulées, et les transmet au module de traitement du protocole de transfert concerné. C'est lui qui identifie ensuite le port, et l'application à laquelle les vraies données doivent être transmises.

#### 2.3.4.3 UDP – User Datagram Protocol

Le protocole UDP est le plus simple des deux protocoles de transmission. Il permet une communication non garantie sans connexion. En d'autres termes, il permet d'envoyer un datagramme d'un port donné à un autre port donné, sans plus. De ce fait, son format est des plus simples (*cf.* FIG. 2.21).

| 0 | 16 | 32 |
|---|---|---|
| Port source | Port destination | |
| Longueur du datagramme | Somme de contrôle UDP | |
| Données ... | | |

FIG. 2.21 – Format d'un datagramme UDP.

Le protocole UDP est donc majoritairement utilisé dans le cas ou l'on souhaite transmettre des données dont la réception ne doit pas être certifiée (soit parce que c'est inutile, soit parce que l'application elle même en assure le rôle), et dont l'ordre d'arrivée des données n'a pas d'importance. Il convient donc particulièrement dans un environnement à échange de messages. Dans des environnements où l'on requiert un échange de flots (continus) de données, le protocole TCP est plus adapté.

#### 2.3.4.4 TCP – Transmission Control Protocol

Le protocole TCP est plus complexe qu'UDP. Il garantit une communication connectée avec garantie de remise et contrôle de flux. De ce fait, son format est beaucoup plus élaboré, et contient une plus grande quantité de meta-données (*cf.* FIG. 2.22).

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

**La communication en mode connecté** nécessite une négociation préalable entre l'émetteur et le destinataire : l'émetteur contacte le destinataire avec une demande d'établissement de connexion. Seulement lorsque le destinataire accepte la connexion, la communication s'établit.

**La garantie de remise** est obtenue par la combinaison un système de numérotation des données et d'acquittements. Lorsque le destinataire reçoit une donnée, il en informe l'émetteur. Comme, par ailleurs, les données sont numérotées, il peut également s'apercevoir que des données manquent.

**La contrôle de flux** permet aux deux parties de s'accorder sur les vitesses de transmission, afin d'éviter qu'un émetteur trop puissant inonde de données un destinataire de capacités inférieures et que ce dernier perde des données ou n'arrive plus à accomplir sa tâche. Il permet aussi de ralentir temporairement la cadence (par exemple lorsque le destinataire est sollicité par ailleurs), pour l'accélérer dès lors que le destinataire est à nouveau en mesure de traiter les données au rythme initial.

L'analyse des différents champs de l'entête d'un segment TCP permettra de mieux comprendre les différents mécanismes en jeu.

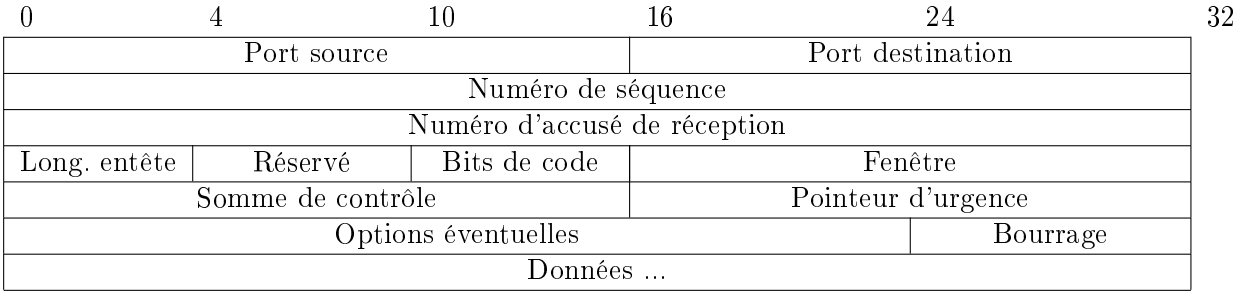


FIG. 2.22 – Format d'un segment TCP.

**Port source – Port destination** ne requièrent pas beaucoup d'attention. Ils identifient les ports de communication utilisés.

**Numéro de séquence** contient le numéro du segment dans le flot actuel. Lorsqu'un flot est transmis, il est découpé en segments qui sont numérotés séquentiellement. Le premier segment d'une séquence a une valeur aléatoire. Les suivants sont des incréments successifs.

Le fait de prendre un numéro aléatoire a son importance. Nous l'aborderons lorsque nous présenterons en détail le déroulement d'une session TCP, 114.

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

**Numéro d'accusé de réception** fait référence au dernier segment reçu et dont on accuse réception, au détail près que l'on accuse réception d'un segment $n$ en envoyant $n+1$ comme accusé de réception.

En d'autres termes, au lieu de signaler « *j'ai bien reçu n* », on signale « *je suis prêt à recevoir* $n+1$ ». C'est fondamentalement différent, puisque cela permet d'acquitter des blocs de segments en acquittant uniquement le dernier arrivé, plutôt que d'encombrer le réseau avec des envois d'acquittements individuels.

**Longueur entête** exprimé en multiples de 32 bits. Cette valeur permet de déterminer où commencent les données, puisqu'il peut y avoir des options de taille variable.

**Réservé** (comme son nom l'indique ...)

**Bits de code** au nombre de 6. Ils indiquent que certains champs de l'entête sont valides, ou que des options particulières sont actives :

| Dénomination | Rôle |
|---|---|
| URG | Le champ `pointeur de données urgentes` est valide (*urgent*) |
| ACK | Le champ `accusé de réception` est valide (*acknowledgment*) |
| PSH | Les données dans ce segment ne doivent pas être bufferisés (*push*) |
| RST | La communication doit être coupée immédiatement (*reset*) |
| SYN | Demande de synchronisation des numéros de séquence (*synchronize*), demandée en début de communication. |
| FIN | On signale la fin du flot à transmettre, et sa disponibilité à terminer la connexion. |

**Fenêtre** est un entier non signé indiquant la taille du tampon de réception de l'émetteur. Elle indique au destinataire combien de segments il peut envoyer en rafale avant de recevoir un accusé de réception. Ceci permet d'augmenter l'efficacité de la transmission en n'attendant pas l'arrivée d'un accusé de réception d'un segment avant d'envoyer le suivant.

Il est important que ce soit le récepteur qui spécifie la taille de la fenêtre. Comme des segments peuvent arriver dans le désordre, il doit être capable de stocker temporairement tous les segments d'une fenêtre en attendant qu'un segment manquant arrive. Elle permet aussi au récepteur de moduler la vitesse d'émission de la source en fonction de sa charge, tout simplement en réduisant sa fenêtre de réception.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

**Somme de contrôle** calculé selon l'arithmétique à complément à un sur 16 bits. Il est à noter que cette somme se fait sur l'entête TCP, plus les données, ainsi que sur une pseudo-entête IP incluant les adresses IP source et destination, et la longueur de segment TCP.

**Pointeur d'urgence** contient une valeur uniquement lorsque le drapeau `urg` est mis dans les bits de code. Le pointeur contient le décalage (à partir du segment courant) à partir duquel les segments contiendront à nouveau des données non urgentes. En d'autres termes, il définit l'intervalle $[a, b]$ de numéros de séquence des segments comportant des données urgents, $a$ étant le numéro de séquence actuel, $b$ étant la somme de $a$ et du pointeur d'urgence.

**Options** éventuelles permettant le réglage des délais d'attente, de la taille maximale des segments *etc.*.

**Bourrage**

**La mise en œuvre et le déroulement d'une session** TCP passe par plusieurs phases.

0. Le serveur démarre et se met en attente de connexions (ce qui constitue plutôt un préalable que réellement une partie intégrante de la session).
1. Le client fait une demande d'établissement de connexion aurpès du serveur. Ce dernier peut accepter ou refuser la connexion.
2. En cas d'acceptation, la communication s'établit et l'échange de données a lieu
3. Le serveur ou le client informe son homologue de sa volonté de mettre fin à la communication ; puis la communication se termine.

Ces différentes phases se caractérisent par des états et le contenu de certains des champs des segments échangés. Nous les détaillons ici :

**Ouverture de connexion** Pour obtenir une ouverture de connexion, le client émet un segment dont le drapeau SYN est mis à `1` (SYN = *synchronisation des numéros de séquence*) et dont le numéro de séquence est initialisé à un nombre aléatoire $n$. S'il accepte la connexion, le serveur répond en transmettant un segment acquittant la demande (drapeau ACK à `1`, et le champ *accusé de réception* valant $n+1$), et demandant la sychronisation avec son propre numéro de séquence aléatoire $m$, selon le même schéma. Le client répond avec un segment d'acquittement (drapeau ACK à `1`, et le champ *accusé de réception* valant $m+1$) avec un numéro de séquence incrémenté de 1. S'il refuse la connexion, le serveur répond en mettant le drapeau RST au lieu de SYN, tout en

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

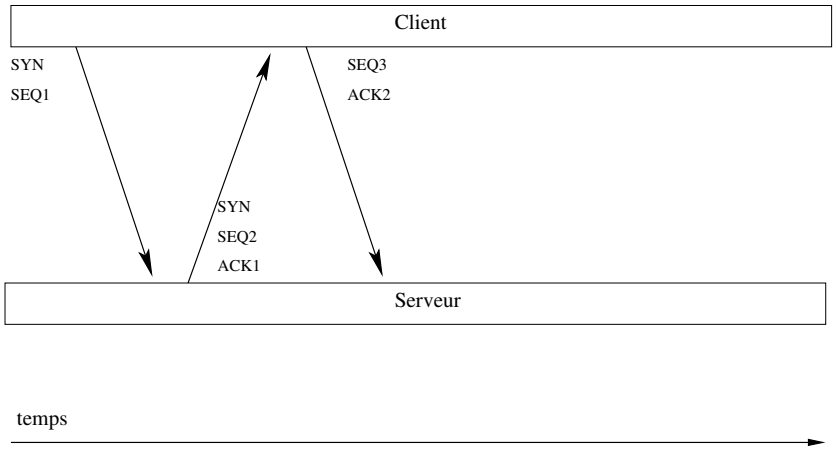


FIG. 2.23 – Ouverture de session TCP .

acquittant le segment $n$. Par ailleurs, si le client ne reçoit pas de réponse avant un temps déterminé, il retente l'opération en réémettant le segment initial, jusqu'à un nombre maximal d'essais. Le processus d'échange de drapeaux SYN est appelé *la poignée de mains en trois temps* (*triple handshake* en anglais).

**Note :** on peut expliquer ici le rôle du nombre aléatoire, plutôt de de numéroter systématiquement les séquences à partir de 0. En effet, imaginons le scénario suivant, et représenté dans FIG. 2.24 :

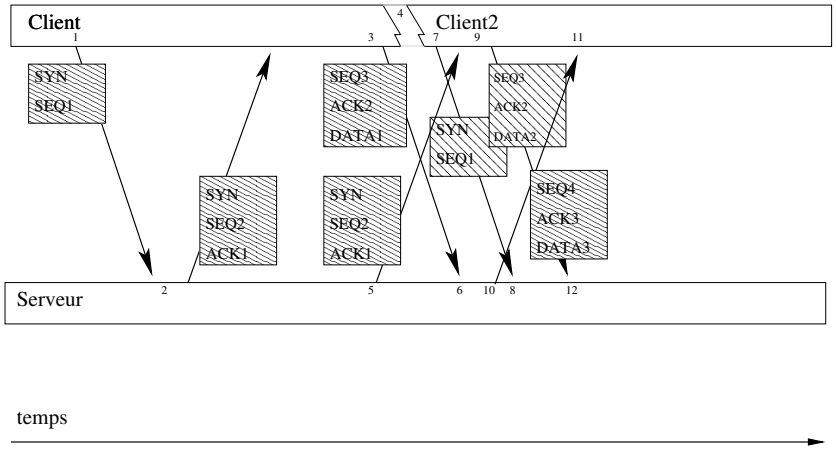


FIG. 2.24 – Utilité d'un numéro de séquence aléatoire en TCP.

1. Le client demande une connexion (émission de SYN – SEQ 0)
2. Le serveur reçoit la demande et répond (émission de SYN – ACK 1 – SEQ 0)

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

3. Le client reçoit le segment SYN, rencontre des anomalies internes et finit par envoyer en retard l'acquittement de celui-ci ainsi que les premières données de la communication (émission de ACK 1 – SEQ 1 – DATA).
4. Immédiatement après cette transmission, le client tombe en panne.
5. Entre-temps, le serveur, n'ayant pas encore reçu l'acquittement de son segment SYN le retransmet (ré-émission de SYN – ACK 1 – SEQ 0).
6. Il reçoit l'acquittement du client qui avait était envoyé juste avant que ce dernier tombe en panne, et répond (émission de ACK 2 – SEQ 1 – DATA')
7. Un nouveau client demande une connexion au server (émission de SYN – SEQ 0)
8. Le serveur reçoit cette demande de connexion et la rejette, car il peut légitimement supposer qu'il s'agit d'une réémission superflue ou d'un doublon dans le réseau.
9. Entre-temps, le client reçoit le segment SYN – ACK 1 réémis à l'étape 5, et suppose légitimement que le serveur a accepté la communication. Il envoie donc son acquittement et les premières données de sa communication à lui (émission de ACK 1 – SEQ 1 – DATA$^\star$).
10. Par ailleurs, le serveur reçoit le dernier segment du client interrompu envoyé à l'étape 6 et répond normalement.
11. Le nouveau client reçoit ce segment, et l'accepte sans problème comme réponse à ses données DATA$^\star$ tandis qu'en réalité il s'agit de la réponse aux données DATA.
12. Lorsque le serveur reçoit le segment DATA$^\star$ il le considère comme un doublon de DATA (puisque même origine, même numéro de séquence) et le jette.

C'est bien le fait de démarrer systématiquement avec un même numéro de séquence qui est à l'origine de cette confusion. Si nouveau client avait pris un autre numéro de séquence, il n'aurait pas interprêté les segments qui ne lui étaient pas destinés comme étant valables et les communications se seraient interrompues naturellement.

**Échange de données** L'un des aspects les plus importants de TCP est la remise fiable des données. Nous en avons déjà aperçu un exemple dans le scénario précédent, dans lequel les deux interlocuteurs *acquittent* les

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

segments qu'ils reçoient, et *réémettent* des segments, lorsque l'acquittement n'arrive pas au bout d'un certain temps. Le protocole permet tout une autre série de contrôles sur l'échange des données : les délais de réémission, la taille des segments, la taille des fenêtres d'acquittemet, *etc.*. Il est important que le lecteur soit conscient de ces possibilités. En revanche, nous laissons le soin aux intéressés de les étudier dans les ouvrages spécialisés. Bien que le contrôle de flux du protocole fonctionne globalement de façon satisfaisante, il existe un grand nombre cas théoriques et pratiques où il est mis en défaut et où l'on prouve que les mécanismes eux-mêmes peuvent être génératrices de perturbations. Ces points font l'objet de recherches très actives.

Nous nous attarderons dans ce paragraphe uniquement sur les principes de l'acquittement et des fenêtres glissantes. Le principe de base de TCP est que chaque segment émis doit être acquitté par le récepteur. Si l'émetteur ne reçoit pas d'acquittement au bout d'un temps déterminé, il réémet le segment. Bien évidemment, on conçoit facilement que cela peut être très inefficace si on exige que l'émetteur ne transmette le segment suivant que lorsqu'il a reçu l'acquittement du précédent ... surtout si on a un réseau à haut débit, mais de forte latence. Le protocole permet déjà de transmettre des données en même temps que de les acquitter. Ceci peut suffire sur des réseaux a faible latence et lorsque les deux interlocuteurs s'échangent des flux équivalents et constants. En revanche, s'il y a un déséquilibre dans les échanges ou si le réseau est à forte latence, le temps d'attente des acquittements individuels devient trop pénalisant. Par ailleurs elle peut créer une surcharge non négligeable sur le réseau. C'est pour cette raison que l'on a introduit le concept de *fenêtre glissante* dans le protocole. Selon ce principe, l'émetteur a le droit d'émettre un nombre de segments convenu entre les deux communiquants, avant d'attendre un acquittement du récepteur. C'est ce qu'on appelle la *taille de la fenêtre*. Par ailleurs, le récepteur a le droit d'acquitter une série de segments, plutôt que chaque segment individuellement. Lorsque l'émetteur reçoit un acquittement pour $n$ segments, il avance la fenêtre de envoye les $n$ segments suivants. Bien évidemment (comme le montre FIG. 2.25) $n$ n'est pas nécessairement égal à la taille de la fenêtre.

**Clôture de la connexion** Il existe deux façons de mettre un terme à une session. L'une brutale, en cas d'urgence, par l'envoi d'un segment contenant le drapeau RST. Elle met immédiatement fin à la session. L'autre, la plus usuelle, se fait en (presque) trois temps, comme l'établissement de session. En effet, l'initiateur de la terminaison signale son souhait en

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

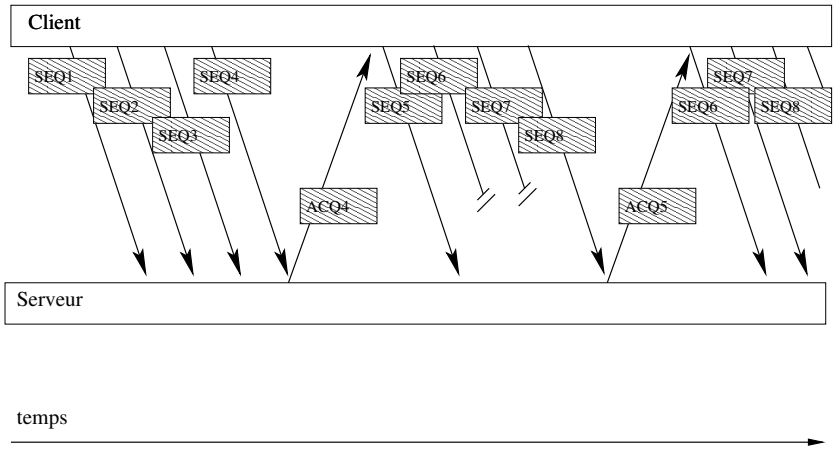


FIG. 2.25 – Exemple d'échange avec une fenêtre glissante de 4.

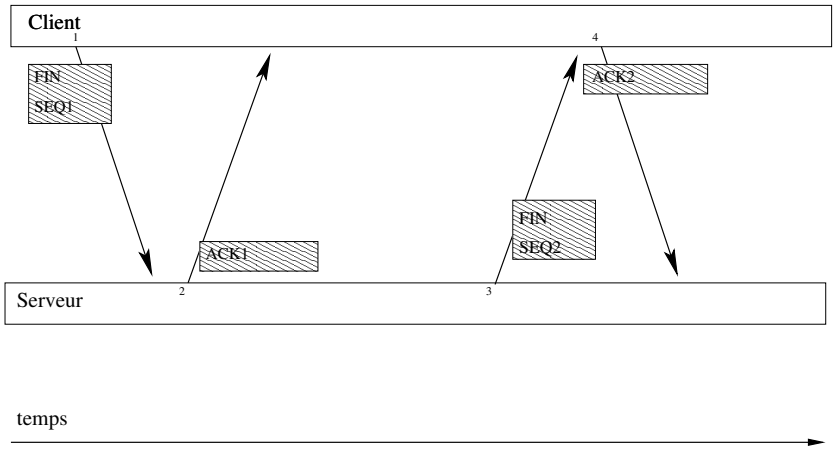


FIG. 2.26 – Fermeture de session TCP .

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

mettant le drapeau FIN (1). L'autre acquitte (2), puis fait le nécessaire pour terminer proprement la session. Il envoie ensuite, à son tour, un drapeau de FIN (3), puis l'arrivée de l'acquittement final termine la session (4). Cet échange est représenté dans FIG. ??.

## 2.4 Applications réparties et communicantes

Dans cette partie nous abordons la relation entre les réseaux et les systèmes d'exploitation, en situant cette relation à trois niveaux :

1. Au niveau le plus bas, le système d'exploitation doit être capable d'identifier des *interfaces réseau*, ou, en d'autres termes, déterminer lesquels de ses périphériques lui permettent d'utiliser un réseau, de quel(s) réseau(x) il s'agit, quels protocoles ils utilisent. Il s'agit là d'une tâche de configuration ou d'administration. Une fois l'interface correctement identifée, elle est généralement accessible comme n'importe quelle autre périphérique via les appels système d'écriture et de lecture standards.
2. Au niveau intermédiaire, plutôt que de donner un accès brut au périphérique, le système d'exploitation gère, à l'instar des descripteurs de fichiers, l'accès concurrentiel à l'interface réseau par différents processus. Un des outils standards étant la *socket*, nous aborderons son utilisation dans les sections qui suivent.
3. Enfin, connecté au réseau, le système peut s'intégrer dans un environnement de services (mail, login, serveurs de fichiers, ...), qui chacun s'appuye sur un protocole particulier. Nous énumérerons un nombre d'entre eux.

### 2.4.1 Les sockets et IPC

L'outil le plus proche du système d'exploitation est la *socket*[17]. Conceptuellement, c'est un point d'entrée (et/ou de sortie) d'un canal de communication entre deux processus communicant entre eux. D'un point de vue plus technique, la socket s'apparente au descripteur de fichier : c'est une structure de données fournie par le système d'exploitation qui permet l'accès à une ressource de communication, quelque soit le type de communication sous-jacent. À cet égard, les sockets sont hérités par des processus fils, et plusieurs processus peuvent donc avoir des sockets représentant la même liaison de communication. À l'inverse des fichiers, en revanche, les données que l'on

[17] Le terme anglais *socket* signifiant *prise* ou *emboîtement*

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

lit ou écrit dans une socket, sont volatiles. Dès lors qu'une donnée est lue, elle cesse d'exister. On ne peut donc pas partager des données envoyées par un processus parmi plusieurs récepteurs éventuels. C'est seulement le premier qui lit l'information qui la reçoit. De ce fait, une bonne règle consiste à s'efforcer à maintenir la règle qui prévoit qu'à chaque extrémité d'une communication on ne trouve qu'un seul processus.

Historiquement, les sockets datent de la version 4.2 de la version BSD d'Unix, et font partie de la batterie d'outils de communication inter-processus (IPC – *Inter Process Communication*) comme les signaux *cf.* p. 42. Depuis Windows2000, elles sont aussi parties intégrantes des plateformes Windows–NT/XP.

#### 2.4.1.1 Principes de base

Il est important de remarquer qu'a priori, il n'y a de correspondance stricte entre une communication établie sur un réseau et la vue que peuvent en avoir différents processus. De la même façon qu'un fichier sur disque peut être ouvert en même temps par différents processus, une même communication peut être observée par différents aceurs du système.

**La définition d'une communication** entre deux hôtes du réseau est donnée par le quadruplet `(@IP, port) émetteur`, `(@IP, port) destinataire`. Tout segment ou datagramme arrivant sur un port, provenant d'un hôte particulier avec un port identifié sera automatiquement remis par le système aux mandataires de la communication. Il convient donc, pour tout processus voulant échanger des données sur le réseau de s'enregistrer auprès du système d'exploitation pour que les informations lui soient remises.

C'est plus précisément le rôle des *sockets*. Les appels système permettant de créer ou d'activer des sockets ont pour effet d'enregistrer le processus appelant auprès du système comme mandataire d'une communication particulière. Les opérations de lecture ou d'écriture dans la socket ne concerneront qu'uniquement la communication spécifiée. Il est toutefois possible que plusieurs processus ouvrent des sockets (distinctes) sur une même communication (à condition de respecter un certain nombre de contraintes de droits d'accès *etc.* [18]).

Il est toutefois une bonne idée de réserver seulement une communication particulière à une socket dans un seul processus.

[18] Par exemple, sous Posix, lors de l'invocation de la commande `fork()`, toutes les sockets ouvertes par le processus père sont hérités par les processus fils.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

#### 2.4.1.2 Techniques de base

Lors de la programmation d'applications réseaux, la première étape consiste à créer et déclarer la socket, de lui affecter certains attributs (type, port local, hôte et port distant, *etc.*), et de l'activer enuite. Une fois activée, une socket peut avoir trois états, selon le type de communication qu'elle représente :

**ouverte, non connectée** est l'état correspondant à une communication type UDP. La connexion est ouverte (la socket est connectée à un port UDP) prête à recevoir des datagrammes ou d'en remettre, sans contrainte particulière sur leur origine ou leur destination. A chaque utilisation, il est nécessaire de fournir l'adresse du destinataire. Il appartient aussi au processus en question de faire le tri des données reçues afin d'identifier leur traitement approprié.

**ouverte, connectée** est l'état correspondant à une communication de type TCPétablie ou UDP contrainte à un hôte particulier. La socket est liée à un quadruplet `(@IP, port) émetteur`, `(@IP, port) destinataire`, et son utilisation ne nécessite plus d'attention particulière du processus propriétaire : en écriture, le destinatiare est parfaitement identifié, ainsi que l'origine pour la lecture.

**à l'écoute d'une connexion** est l'état d'une socket appelée serveur, et est particulier aux communications de type TCP. C'est une socket qui ne gère pas de données, mais qui est en attente d'une demande de connexion provenant de n'importe quel hôte. C'est elle qui va gérer la poignée de mainui établira la connexion entre les deux hôtes. En général, cette procédure est bloquante tant qu'aucune demande de connexion n'arrive. Ensuite, lors d'une nouvele connexion, elle crée une nouvelle socket de type *ouverte connectée* qui gèrera ensuite les échanges réels de données.

**Note importante :** les sockets ne sont pas réservées uniquement à des communications `TCP/IP`. Elles supportent toutes sortes de protocoles, et servent même, dans le monde Unix – dont elles sont issues – comme support aux communications inter-processus sur une même machine. Dans ce document, nous les illustrerons toutefois uniquement sur `TCP/IP`.

#### 2.4.1.3 Cas concrets : des serveurs et des clients

Le choix du type de socket et du type de communication (mode connecté ou déconnecté) utilisés dans une application dépendent majoritairement des besoins et du type d'échanges qui devront être supportés. Nous esquissons ici quelques cas d'école.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

**Vocabulaire :** on utilise couramment les termes de *client* et de *serveur* dans le cas d'applications interconnectées.

- La notion de *serveur* est lié à la notion de *service*. C'est un processus qui détient des informations, ou est capable de faire des choses, qu'il met à disposition d'autres processus. Il a donc la particularité « d'exister », même si personne, à un temps $t$ n'a besoin des dits services. Il est généralement en attente de communications provenant d'une origine non identifiée d'avance.
- La notion de *client* se rapporte à la consommation d'un service. Le client va contacter un serveur identifié, afin d'obtenir l'information ou le service dont il a besoin. C'est donc lui qui va initier la communication, lorsqu'il en a l'utilité.

Il est important de comprendre que ces notions n'ont qu'une valeur de conceptualisation et de hiérarchisation d'un échange de données à un instant donné. Une application complexe peut bien évidemment être à la fois client et serveur, selon les scénarios d'exécution qu'elle gère.

Il y a toutefois dans la littérature professionnelle une conotation fondamentalement différente lorsque l'on parle d'*architectures client-serveur*. Elle s'oppose dans ces cas-là aux architectures *mainframe*, *n-tiers* ou *peer-to-peer*. Cette différence se situe plus au niveau de l'architecture logicielle, lié au concept de *serveur* (applicatif) central et unique et des *clients* conçus spécifiquement pour la communication avec le serveur en question. Elle est de ce fait opposée aux architectures *n-tiers* dans lesquels le serveur est éclaté en chouches interopérantes, et les clients deviennent « légers » et non dédiés, ou aux architectures *peer-to-peer* où la notion de serveur est diluée dans les clients, et où chacun est à la fois serveur et client. Il reste à remarquer que ces architectures se basent toujours sur des échanges *serveurs* et *clients* au sens où nous l'entendons ici, et que nous continuerons à appeler l'initiateur de ces échanges le *client* et celui qui s'était préparé à en recevoir le *serveur*.

Dans ce qui suit nous développons deux cas particuliers d'échanges client-serveur. Il en existe d'autres. Il n'est notamment pas obligatoire qu'in serveur connecté soit nécessairement parallèle ou qu'un échange non connecté s'appuie sur un serveur séquentiel. Dans le premier cas, nous développons un échange UDP de façon séquentielle. Le serveur est en attente de communications, et répond au fur et à mesure de leur réception. Comme le montre l'exemple 2.3, où les clients envoient une rafale de datagrames et ne reçoivent les réponses que lorsque le server a eu le temps de traiter ceux qui étaient arrivés avant.

**Exemple 2.3 : Le serveur séquentiel non connecté – en `C`**

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

Dans cet exemple, nous présentons un service d'horodatage : un serveur fournit l'heure à qui en fait la demande. Le programme fourni, écrit en `C`, se contente de lancer deux threads (*cf.* Exemple 1.2). Le premier démarrant un serveur UDP, écoutant sur le port `port_serveur`, et renvoyant l'heure à chaque client qui lui envoie un datagramme. Le second démarrant un client, qui envoie un petit message UDP au serveur, et récupère l'horodatage.

Le serveur est bien séquentiel, dans le sens où il récupère un datagramme, le traite, puis passe au suivant. Tant que le traîtement d'un datagramme n'est pas terminé, les autres restent en attente. (dans ce cas précis, cela n'a que peu d'incidence, puisque le traitement est très rapide).

Nous ne détaillons pas, ici, tous les types et toutes les fonctions système utilisés. Ils sont tous suffisament détaillés dans les pages man de n'importe quel système Unix ou Linux (*e.g.* `man 2 bind`). Attention toutefois à des homonymes dans des chapitres différents.

Serveur séquentiel non connecté – `C` horodatage.c

```
#include <sys/types.h>
#include <sys/time.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

#define s_buf 100

int port_serveur = 2020;
int true = 1;

void serveur();
void client(void *);

int main(int argc, char* argv[]) {

  pthread_t thread1, thread2, thread3;

  pthread_create( &thread1, NULL, (void*)&serveur, 0);
```

main

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

```
  sleep(1); /* Pause pour être sûr que le serveur soit prêt */

  pthread_create( &thread2, NULL, (void*)&client, "client 1");
  pthread_create( &thread3, NULL, (void*)&client, "client 2");

  /* On attend que les threads se terminent ... */
  pthread_join( thread2, NULL);
  pthread_join( thread3, NULL);

  exit(0);
}

void serveur() {                                                    serveur
  /* création d'une socket dans le domaine
     AF_INET (protoles Internet)
     de type SOCK_DGRAM (mode non connecté)
     de protocole 0 (il n'y en a qu'un : UDP) */
  int sck_serv = socket(AF_INET, SOCK_DGRAM, 0);

  /* structure de renseignement de la socket */
  struct sockaddr_in nom;
  nom.sin_family = AF_INET; /* famille de protocoles */
  nom.sin_port = htons(port_serveur); /* le port local */
  /* adresse de rattachement (utile si on a plusieurs interfaces réseau) */
  nom.sin_addr.s_addr = INADDR_ANY;

  /* Enregistrement de la socket auprès du système */
  if( bind(sck_serv, (struct sockaddr*) &nom, sizeof(nom)) ) {
    perror(strerror(errno));
    exit(errno);
  }

  /* Boucle de gestion des évènements */
  while( true ) {
    /* Création d'un tampon pour les données échangées */
    unsigned char buf[s_buf];

    /* Création d'un structure d'adresse pour recueillir
       les infos sur le client qui se connecte */
    struct sockaddr_in emetteur;
    socklen_t taille = sizeof(emetteur);
```

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

```
    /* Réception des données                                    */
    /*  - remplissage du tampon α hauteur de « reception » octets
        - renseignement de la structure « emetteur » */
    int reception = recvfrom(sck_serv, buf, s_buf, 0,                    70
                             (struct sockaddr*) &emetteur, &taille);

    /* Affichage des informations recueillies
       inet_ntoa() transforme le numéro d'adresse binaire en format
       IP classique xxx.xxx.xxx.xxx */
    printf("Arrivée d'une connexion depuis %s sur le port %d\n",
           inet_ntoa(emetteur.sin_addr), ntohs(emetteur.sin_port));

    /* Parade contre d'éventuels débordements de tampon */
    buf[(reception<s_buf)?reception:s_buf-1] = '\0';                     80
    printf("Données = %s\n", buf);

    sleep(1); /* pour avoir un horodatage différent */

    /* Envoi de la réponse                                      */
    /* récupération de l'heure, et copie dans le tampon */
    time_t temps = time(0);
    strncpy(buf, ctime(&temps), s_buf);

    /* envoi du contenu du tampon */                                     90
    sendto(sck_serv, buf, s_buf, 0, (struct sockaddr*) &emetteur, taille);
  }
}

void client(void* arg) {                                          client
  /* déclaration et création d'une socket UDP */
  int sck = socket(AF_INET, SOCK_DGRAM, 0);

  /* structure de renseignement de la socket */
  struct sockaddr_in nom;                                            100
  nom.sin_family = AF_INET;
  nom.sin_port = htons(0);
  nom.sin_addr.s_addr = INADDR_ANY;

  /* récupération des données IP du serveur (ici « localhost ») */
  struct hostent *hp;
```

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

```
  hp = gethostbyname("localhost");

  /* renseignement de la structure sockaddr pour la communication */
  struct sockaddr_in dest;                                          110
  dest.sin_family = AF_INET;
  dest.sin_port = htons(port_serveur);
  memcpy(&dest.sin_addr, hp->h_addr, hp->h_length);

  /* Enregistrement de la socket auprès du système */
  if( bind(sck, (struct sockaddr*) &nom, sizeof(nom)) ) {
    perror(strerror(errno));
    exit(errno);
  }
                                                                    120
  /* Envoi de 3 demandes d'horodatage */
  unsigned int i;
  for(i=0; i<3; ++i)
    sendto(sck, arg, strlen(arg), 0, (struct sockaddr*) &dest, sizeof(dest));

  printf("%s : fin d'envoi des messages\n", arg);

  /* Préparation d'un tampon pour la réception */
  char buf[s_buf];
                                                                    130
  /* Réception des réponses */
  for(i=0; i<3; ++i) {
    /* (Attention, contrairement au serveur, on a choisi de ne pas
       se préoccuper de la provenance de la réponse!!) */
    int reception = recv(sck, buf, s_buf, 0);
    printf("%s : données reçues = %s", arg, buf);
  }

  close(sck);
}                                                                   140
```

Dans le second exemple, en mode connecté cette fois, le serveur se contente d'être à l'écoute de nouvelles connexions. Dés qu'une demande de communication arrive, la fonction d'écoute `listen()` renvoie une socket de communication, qui est aussitôt passée à un gestionnaire (*handler* en anglais) qui

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

s'exécute dans un thread à part. Cette opération permet alors au serveur d'être immédiatement disponible pour recevoir d'autres demandes.

**Exemple 2.4 : Le serveur parallèle connecté – en JAVA**

Un serveur parallèle, comme dans cet exemple, délègue la gestion de chaque communication à un thread indépendant, se rendant ainsi tout de suite disponible pour prendre la communication suivante. En quelque sorte, des connexions simultanées sont traitées en parallèle, d'où la dénomination.

Serveur parallèle connecté – JAVA JavaSocketExemple.java

```
import java.net.*;
import java.io.*;

public class JavaSocketExemple {

    public static int ServerPort = 45000;

    // Le programme principal :
    // Un serveur se met en attente de connexions (ici 5)
    // Chaque client lui envoie son nom, et le serveur lui répond
    // avec « Bonjour + nom ».
    public static void main(String[] args) {

        // Déclaration, puis création d'une socket de type ServerSocket
        // laSocket se mettra en attente de demandes de connexion,
        // provenant des clients éventuels sur le port 45000.
        ServerSocket laSocket = null;

        try {
            laSocket = new ServerSocket(ServerPort);
        } catch (IOException e) {}

        // Création des threads Client
        int nbClient = 5;
        for(int i=0; i<nbClient; ++i)
            (new Client("client "+i)).start();

        // Traitement des connexions provenant des clients crées.
        // Note : habituellement le nombre de connexions n'est pas
        // connue ni limitée.
```

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.

```
        int nbConnexions = 0;
        while(nbConnexions++ < nbClient) {
            try { // ServerSocket.accept() est bloqant
                (new ConnexionHandler(laSocket.accept())).start();
            } catch(IOException e) { }
        }
    }
}

// La classe ConnexionHandler est un thread qui gère une connexion.
// A sa construction on lui passe une socket active, et elle gère
// toutes les communications avec celle-ci. Ensuite elle la ferme et
// le thread se termine.
class ConnexionHandler extends Thread {

    private Socket connexionSocket;

    public ConnexionHandler(Socket s) {                    ConnexionHandler
        connexionSocket = s;
    }

    public void run()                                      run
    {
        // Connection les flots d'entrée et de sortie α la socket
        try {
            PrintStream sortie =
                new PrintStream(connexionSocket.getOutputStream());
            BufferedReader entree = new BufferedReader(new
                InputStreamReader(connexionSocket.getInputStream()));

            // Lecture et écriture dans la socket
            String nomClient = entree.readLine();
            sortie.println("Bonjour " + nomClient);

            System.out.println(entree.readLine());
            sortie.println("Au revoir " + nomClient);

            // Fermeture de la connexion
            connexionSocket.close();
        }
        catch (IOException e) {}
```

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

```
    }
}

// Le thread client. Une fois démarré, le thread se connecte au port 45000,
// envoie son nom, et récupère la réponse du serveur.
class Client extends Thread {

    // Constructeur on ne peut plus classique
    public Client(String nom) {                                    80 Client
        super(nom);
    }

    // La méthode principale du thread
    public void run()                                               run
    {
        // Attente pour laisser le serveur de se mettre en route.
        try {
            sleep(1000);
        } catch(InterruptedException e) {}                          90

        Socket laSocket = null;

        // On crée une socket
        try {
            // Ici on se contente de se connecter à localhost.
            try {
                laSocket = new Socket("localhost",
                                      JavaSocketExemple.ServerPort);
            } catch(UnknownHostException e) {                       100
                System.out.println("Hôte inconnu !");
            }

            // Connection les flots d'entrée et de sortie à la socket
            PrintStream sortie =
                new PrintStream(laSocket.getOutputStream());
            BufferedReader entrée = new BufferedReader(new
                InputStreamReader(laSocket.getInputStream()));

            // Petit dialogue avec le serveur                       110
            sortie.println(getName());
```

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

```
            String réponse = entrée.readLine();
            System.out.println(réponse);

            if(réponse.equals("Bonjour "+getName()))
                sortie.println("Bien le bonjour α vous aussi.");
            else
                sortie.println("Non, il y a méprise !");
                                                                    120
            System.out.println(entrée.readLine());
        }
        catch(IOException e) {}
    }
}
```

**Note :** dans les deux exemples cités ci-dessus, les clients et serveurs sont issus d'un même programme principal. Bien évidemment ceci est fait pour des raisons de facilité du discours. Dès lors qu'un client connaît l'adresse et le port d'un serveur, il peut tenter de s'y connecter, sans pour autant être issu d'un même processus parent.

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

# Annexe A

# L'utilisation de la pile et des registres d'état

Nous reproduisons ici la version complète, en assembleur, du programme donné dans FIG. 1.3, p. 8. L'assembleur n'est pas forcément celui qui est utilisé en standard sur les plateformes Intel, mais est celui généré par `gcc`, qui utilise parfois d'autres conventions que celles rencontrées dans la littérature (notamment pour l'ordre des opérandes).

Le source `C` était celui-ci :

Utilisation des registres d'état - `C` assembleur.c

```
// Déclaration d'une fonction prenant 2 paramètres et retournant un entier
int additionne(int a, int b)                                          additionne
{
  int resultat ;          // Création d'une variable locale
  resultat = a + b ;      // Calcul + affectation α la variable locale
  return resultat ;       // Retour du contenu de la variable locale
}

// Déclaration de variables globales initialisées
int arg1 = 5 ;                                                            10
int arg2 = 1 ;

// Déclaration de la fonction principale
int main()                                                              main
{
  arg1 = additionne(arg1, arg2) ; // appel de fonction, passage de paramètres et
```

```
                                    // récupération + affectation du résultat
}
```

La commande (Linux) `gcc -S test.c` permet d'obtenir le code assembleur suivant :

Utilisation des registres d'état - Assembleur assembleur.s

```
        .file   "test.c"
        .text
.globl additionne
        .type   additionne, @function
additionne :
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax                                   10
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        leave
        ret
        .size   additionne, .-additionne
.globl arg1
        .data
        .align 4
        .type   arg1, @object
        .size   arg1, 4                                         20
arg1 :
        .long   5
.globl arg2
        .align 4
        .type   arg2, @object
        .size   arg2, 4
arg2 :
        .long   1
        .text
.globl main                                                     30
        .type   main, @function
main :
```

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

```
pushl   %ebp
movl    %esp, %ebp
subl    $8, %esp
andl    $-16, %esp
movl    $0, %eax
subl    %eax, %esp
subl    $8, %esp
pushl   arg2                                                    40
pushl   arg1
call    additionne
addl    $16, %esp
movl    %eax, arg1
leave
ret
.size   main, .-main
.section        .note.GNU-stack,"",@progbits
.ident  "GCC : (GNU) 3.3.2 (Mandrakelinux 10.0 3.3.2-7mdk)"
```

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

# Annexe B

# Les sémaphores en POSIX

Nous reproduisons ici la version compilable et exécutable de l'exemple simplifiée de l'utilisation des sémaphores qui figure p. ??.

Sémaphores POSIX – C — semaphoresreels.c

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#include <stdio.h>
#include <sys/types.h>

#include <sys/time.h>
#include <unistd.h>

#include <signal.h>

#include <sys/sem.h>
#include <sys/ipc.h>

#define KEYFILE "/dev/null"

#define S_IRUSR 0x100 /* Le neuvième bit vaut 1 */
#define S_IWUSR 0x080 /* Le huitième bit vaut 1 */

#define BAD_ADDR (char *) -1
```

```
int WRAPPER_NbSemaphors = 1;
int WRAPPER_globalSemId = 0;

int WRAPPER_MemSize = sizeof(int);
int WRAPPER_MemId = 0;
char* WRAPPER_MemAddr = 0;                                          30

key_t WRAPPER_CreateUniqueKey()                                     WRAPPER_CreateUn
{
  key_t key;

  /* get the key */
  if ((key = ftok(KEYFILE,'q')) == -1) {
    perror("Cannot create a key");
    exit(1);
  }                                                                 40

  return key;
}

int WRAPPER_InitSemaphors(key_t key, int number)                    WRAPPER_InitSema
{
  WRAPPER_NbSemaphors = number;

  /* Essai de création des sémaphores associés α 'key' */
  if ((WRAPPER_globalSemId = semget(key, WRAPPER_NbSemaphors,       50
                    IPC_CREAT | S_IRUSR | S_IWUSR)) != -1)
    return WRAPPER_globalSemId;

  /* Si échec pour cause d'existence des sémaphores, alors
     récupération des sémaphores sans création */
  if(errno == EEXIST &&
     (WRAPPER_globalSemId = semget(key, WRAPPER_NbSemaphors,
                                   S_IRUSR | S_IWUSR)) != -1)
    return WRAPPER_globalSemId;
                                                                    60
  /* Echec de création : sortie de programme */
  perror("Impossible de créer les sémaphores");
  exit(1);
}
```

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

```
void WRAPPER_SemSet(unsigned int semnum, int val)                    WRAPPER_SemSet
{
  if ((semctl(WRAPPER_globalSemId, semnum, SETVAL, val)) == -1) {
    perror("Impossible d'attribuer une valeur au sémaphore");
    exit(1);                                                                  70
  }
}

void WRAPPER_down(int num)                                           WRAPPER_down
{
  struct sembuf psem;
  fprintf(stderr,"Opération down (P) sur le sémaphore %d\n", num);
  psem.sem_op= -1;
  psem.sem_flg=SEM_UNDO;
  psem.sem_num=num;                                                           80
  semop(WRAPPER_globalSemId,&psem,1);
}

void WRAPPER_up(int num)                                             WRAPPER_up
{
  struct sembuf vsem;
  fprintf(stderr,"Opération up (V) sur le sémaphore %d\n", num);
  vsem.sem_op= 1;
  vsem.sem_flg=SEM_UNDO;
  vsem.sem_num=num;                                                           90
  semop(WRAPPER_globalSemId,&vsem,1);
}

void WRAPPER_DestroySemaphors()                                      WRAPPER_DestroyS
{
  semctl(WRAPPER_globalSemId, 0, IPC_RMID);
}
```

© 2003-2004, Bart Lamiroy. École Nationale Supérieure des Mines de Nancy.

# Références bibliographiques

[1] *« Architecture de l'ordinateur »* troisième édition, A. Tanenbaum éditions Dunod, 2000.
Livre complet, entrant dans des détails très bas niveau de la structure d'un ordinateur. Il donne également un très bon apperçu des principes de plus haut niveau de la couche matérielle qui ont plus de rapport avec la première partie du cours. Cette référence est donnée à titre indicative pour les lecteurs qui veulent aller plus loin que les objectifs premiers de ce cours.
Ce livre n'est pas à lire pour quelqu'un qui s'intéresse seulement aux aspects algorithmiques et haut niveau du fonctionnement d'un système d'exploitation. Il s'adresse au lecteur qui veut en savoir plus sur les parties d'un OS qui doivent s'adresser au matériel par l'intermédiaire de routines écrites en assembleur (pilotes de périphériques, changements de contexte, gestion des interruptions, *etc.*)

[2] *« `TCP/IP` Illustré »*, W. Richard Stevens, Eyrolles, 2002.
Le livre de référence le plus complet sur le protocole `TCP/IP`. Très technique et absolument complet.

[3] *« tcp/ip, Architecture, protocoles, applications »* troisième édition, D. Comer, Masson InterÉditions, 1996.
Une *bible* très pédagogique qui aborde tous les points de la pile de protocoles tcp/ip. Ce livre, qui se lit très facilement, est incontournable pour tous ceux qui veulent avoir une idée précise de l'ensemble des protocoles traités en cours, et plus.

[4] *« Windows–nt/xp Shell Scripting »*, T. Hill, Macmillan Technical Publishing, 1998.
Windows–nt/xp n'est pas une interface graphique « click-and-play ». C'est un système d'exploitation complet. À l'instar d'Unix il permet de faire toutes les opérations sans interface graphique et de façon automatisée. Ce livre montre comment écrire des scripts dans le langage de script de Windows–nt/xp.

[5] *« Inside Microsoft Windows 2000 »*, D. A. Salomon et M. Russi-

NOVICH, Mircosoft Press, 2000.
[6] *« Short History of the Internet »*, B. STERLING, http ://www.forthnet.gr/forthnet/isoc/short.history.of.internet
[7] *« Internetworking Technology Handbook »*, Cisco Systems, http ://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/index.htm
[8] *« Measured Capacity of an Ethernet : Myths and Reality »*, David R. BOGGS, Jeffrey C. MOGUL, Christopher A. KENT, HP/Compaq, http ://www.research.compaq.com/wrl/publications/abstracts/88.4.html
[9]

© 2003-2004, Bart LAMIROY. École Nationale Supérieure des Mines de Nancy.