Splendeurs et servitudes des Systèmes d'exploitation

Histoire, fonctionnement, enjeux

Laurent Bloch

25 août 2024

Du même auteur

Initiation à la programmation et aux algorithmes – Avec Python, Technip éd., 2020

Initiation à la programmation et aux algorithmes – Avec Scheme, Technip éd., 2020

Révolution cyberindustrielle en France, Economica éd., 2015

Sécurité informatique – Pour les DSI, RSSI et administrateurs, Eyrolles éd., 5ème édition, 2016

La pensée aux prises avec l'informatique – Systèmes d'information, Laurent Bloch éd., 2017

L'Internet, vecteur de puissance des États-Unis ? - Géopolitique du cyberespace, nouvel espace stratégique, Diploweb éd., 2017

Site Web de l'auteur
(comporte des documents complémentaires)

`https://laurentbloch.net`

Table des matières

Préface de Christian Queinnec Professeur émérite à l'Université Pierre et Marie Curie

« Pourquoi l'informatique est-elle si compliquée ? » J'entends, je lis, souvent, cette question. Elle dénote à la fois une incompréhension de ce que recouvre réellement l'informatique (manipuler une feuille de calcul dans un tableur ressortit-il à l'informatique ?), un effroi devant une révélation a priori déplaisante (l'informatique est complexe) enfin, un découragement devant l'effort supposé immense qu'il faudrait déployer pour dominer cette science.

Si l'on reprend les termes de cette question en les adaptant aux mathématiques, son inanité saute aux yeux car qui s'exclamerait « pourquoi les mathématiques sont-elles si compliquées ? ». Il est de notoriété publique que les mathématiques sont compliquées : un long apprentissage, comptant de nombreuses années d'étude et accompagné d'un discours approprié, nous en ont finalement persuadés. Tapoter une calculette, nous en sommes sûrs, ne s'apparente pas à faire des mathématiques et se tromper dans les touches n'est pas vécu comme une insuffisance en mathématiques.

Reprenons encore les termes pour les adapter à la mécanique, « pourquoi la mécanique est-elle si compliquée ? » pourrait s'exclamer un automobiliste immobilisé le long d'une autoroute. Il a encore (pour quelques années seulement) la possibilité d'ouvrir le capot de sa voiture et de contempler la belle ordonnance de fils, de tuyaux et de pièces métalliques qui autrefois fonctionnaient lorsqu'il mettait le contact. Mais, fort heureusement pour lui, existent des garagistes et des mécaniciens qui pourront remettre en état sa voiture. Notre automobiliste n'aura plus que le goût amer de l'argent dépensé sans savoir pourquoi cela ne marchait plus ni pourquoi cela re-marche. Notons que, dans ce cas, recourir à un professeur d'université en mécanique n'a que peu de chances d'être fructueux : l'écart entre la science mécanique et la technique étant par trop grand.

Conduire n'est pas « faire de la mécanique » pas plus que mettre en gras un titre n'est « faire de l'informatique ». En revanche, déceler un bruit bizarre en roulant et l'attribuer aux pneus ou au moteur, à la direction ou au freinage, aide le

diagnostic du mécanicien. Déceler si un dysfonctionnement provient de l'affichage ou du réseau, d'un disque ou de l'ordonnancement facilite, de même, le diagnostic. Encore faut-il, tout comme en mécanique, connaître les grandes fonctions et leurs relations, savoir ouvrir le capot et nommer les éléments découverts. C'est ce but que sert l'ouvrage de Laurent Bloch.

L'informatique ne se réduit pas à un ordinateur, ni même aux logiciels qui l'équipent. Un ordinateur a une structure physique (unité centrale, périphériques, etc.), il est animé par un système d'exploitation qui, lui même, est structuré (ordonnancement, système de fichiers, réseaux, etc.) et sert de support à de multiples applications. Cette structure et son histoire, la lente maturation des concepts et leur évolution sous la pression des connaissances, des désirs et de la mercatique sont excellemment narrées dans ce livre.

Ce livre est excellent et constitue une remarquable introduction à l'informatique, science de l'abstrait, par le biais d'un de ses produits les plus immatériels mais des plus répandus : les systèmes d'exploitation. J'en recommande la lecture à tout utilisateur conscient et curieux car sa future liberté, en tant que simple utilisateur de l'informatique, dépend en grande partie de sa capacité à comprendre les enjeux des batailles politiques qui, en ce moment, font rage. Et si le lecteur de cette préface se demande pourquoi la liberté intervient dans ce qui n'est qu'une matière technique, nous l'invitons derechef à se plonger dans cet ouvrage.

Avant-propos

Depuis que l'ordinateur a investi la vie quotidienne, chacun découvre son compagnon invisible, immatériel, omniprésent et tyrannique, le système d'exploitation, dont les exemples les mieux connus par leur nom sont Android, Windows ou Linux ; c'est un logiciel destiné à piloter et coordonner les différents éléments constitutifs d'un ordinateur (ou d'un téléphone, c'est la même chose) et à en faciliter l'usage par un être humain. Ce logiciel a donc deux faces : une que voit l'utilisateur, auquel il présente sur un écran des images qui sont des métaphores des objets techniques qui agissent en coulisse, et à qui il permet d'agir sur ces objets par l'intermédiaire de la souris et du clavier, tandis que l'autre face, de loin la plus complexe, mais invisible, assure la mise en œuvre cohérente et efficace des dispositifs techniques internes de la machine, ainsi que les interactions avec le réseau, les appareils de mémoire externe, les micro et haut-parleurs, etc.

S'il existe des ouvrages de vulgarisation pour expliquer le fonctionnement des ordinateurs, leur lecture est souvent frustrante parce qu'elle se limite au matériel, dont le comportement observable est en fait une représentation médiatisée par le système d'exploitation. Si l'utilisateur plus ou moins consentant d'un ordinateur veut comprendre ce qui se passe sur son écran c'est en fait le système d'exploitation qu'il faut expliquer. C'est un des objectifs de ce livre.

Pour introduire le lecteur dans l'univers des systèmes d'exploitation, le présent ouvrage emprunte un itinéraire génétique et historique qui part des problèmes qu'ont voulu résoudre les pionniers des années 1950 pour parcourir les grands domaines que doit gérer un système : les processus, la mémoire, le temps, la persistance des données, les échanges avec l'extérieur, la sécurité, enfin l'interface personne-ordinateur, la partie visible du système à quoi beaucoup croient qu'il se réduit. Cette visite historique se justifie par la constatation que, derrière une extrême diversité extérieure des réalisations proposées à l'utilisateur, les mécanismes choisis pour réaliser le cœur du système sont d'une grande similitude.

Il est difficile de parler de système d'exploitation sans parler de réseau, et il en sera question.

Alors, l'ordinateur a-t-il une âme ? Au sens religieux, certes non, mais dans l'acception étymologique de l'*animus*, oui, l'ordinateur est bien animé par le système d'exploitation sans lequel il ne serait qu'un amas de ferraille.

Complexité des systèmes d'exploitation

Le système d'exploitation est sans doute un des objets les plus complexes créé par le cerveau humain. C'est un logiciel, soit une entité abstraite constituée d'idées. Essayons d'estimer la quantité d'idées significatives contenues dans un tel logiciel : Jean-Raymond Abrial[3], co-auteur du logiciel de pilotage de la ligne de métro automatique numéro 14 du réseau parisien, a trouvé pour 100 000 lignes de code 30 000 obligations de preuve, la plupart satisfaites automatiquement par le système de spécification et de programmation[32] ; il resta 2 500 démonstrations rétives à l'automatisation, à effectuer par les ingénieurs, ce qui demanda plusieurs mois de travail[13]. Il est possible de dire que ce logiciel comporte 30 000 idées, dont 2 500 idées difficiles, ou originales, pour 100 000 lignes de texte, soit une densité d'une idée significative pour 40 lignes. Sachant que la taille du noyau du système d'exploitation Linux est de l'ordre de 20 000 000 de lignes, si nous appliquons ce ratio, il comporte 500 000 idées significatives. Évidemment nous supposons ici que les complexités algorithmiques des ces deux systèmes soient du même ordre, ce qui semble plausible, mais je n'entrerai pas dans ces considérations.

Un ingénieur spécialiste de l'intégration de grands systèmes industriels m'a donné les indicateurs suivants, pour lesquels, par exemple, le type boulon est un objet, et chaque boulon particulier de ce type une instance de l'objet boulon. Airbus A380 : 150 000 objets, 1 200 000 instances. Une voiture : 6 000 objets, 15 000 instances. N'accordons pas à ces chiffres plus d'importance qu'il ne convient, ils sont de toute façon approximatifs et contestables, mais leur comparaison n'est pas déraisonnable. Ajoutons que les avions et voitures contemporains comportent au moins quelques dizaines de systèmes d'exploitation à bord.

Une autre façon de voir les choses, c'est le volume de la documentation. Au début des années 1970 je suis entré dans l'équipe système centrale de l'Insee, à l'époque équipée d'ordinateurs IBM 360. En tant que bizuth je fus chargé de gérer la documentation du système, à l'époque entièrement sur papier et microfiches, avec comme avantage un grand bureau à cause de la taille nécessaire de la bibliothèque : une trentaine de mètres linéaires. Chaque semaine je recevais une vingtaine de mises à jour, sous la forme de feuilles destinées à remplacer dans chaque brochure les pages périmées. Les gens plus jeunes qui n'ont connu que le Web et les moteurs de recherche auront du mal à mesurer le progrès depuis cette époque, mais l'avantage de ce système manuel fastidieux était que j'avais une connaissance quasiment physique du système et de l'évolution plus ou moins rapide de ses différents composants.

Il faut comprendre quand même « comment ça marche »

Le système d'exploitation joue un rôle si crucial dans le monde contemporain, où il manifeste une telle ubiquité, qu'il est hors de question, pour tout humain responsable de lui-même et de la société, de tout ignorer de son fonctionnement. Lors de la précédente révolution industrielle, mon père et mon grand-père, pourtant de formation purement littéraire, se donnaient la peine de comprendre la machine à vapeur, la centrale hydro-électrique, le four Bessemer, le moteur à combustion interne et j'en passe, je suis d'autant plus déçu de voir qu'aujourd'hui des gens qui occupent des postes de responsabilité dans les entreprises ou dans l'administration se piquent de ne rien comprendre à l'informatique. Alors j'ai écrit ce livre pour essayer de combler ce gouffre d'ignorance, ou tout au moins y contribuer.

Aujourd'hui, il n'y a guère que trois familles de systèmes d'exploitation : z/OS pour les mainframes IBM, Windows de Microsoft (soit dit en passant héritier adultérin de VMS de Digital Equipment, par débauchage de son concepteur principal David Cutler), et Unix. On notera que la famille Unix englobe Linux, macOS, iOS, Android, FreeBSD, NetBSD, OpenBSD et quelques autres. La conception de z/OS remonte à 1964 (alors sous le nom OS/360), celle de VMS-Windows à 1977, celle d'Unix à 1969. Certes, chacun de ces systèmes a connu d'innombrables perfectionnements depuis le milieu des années 1960 : mémoire virtuelle, extension de l'espace d'adressage, interfaces graphiques, réseau, etc. Mais la philosophie générale de chacun est très stable. Il y a quarante ans que je n'ai pas touché à un système OS/360, mais lorsque j'ai écouté il y a quelques mois un exposé sur la sécurité de z/OS je me suis retrouvé dans un univers extrêmement familier.

Pourquoi le monde des systèmes d'exploitation est-il si stable ?

L'écriture de mon livre et sa mise à jour continue depuis la première édition de 2003 ont été grandement facilitées par la grande stabilité technique des systèmes d'exploitation pendant cette période, et par leur regroupement en trois familles. Cette stabilité n'avait d'égale que celle de l'architecture des ordinateurs : aujourd'hui il y a trois familles de processeurs, Intel x86, ARM et PowerPC, avec des marchés de niche, par exemple MIPS pour certains matériels réseau et parce que les industriels chinois en ont acheté la licence il y a longtemps.

Il n'en allait pas de même lors des périodes précédentes, et cette situation n'est pas forcément durable.

Si l'on pense aux décennies 1960-1970-1980, il y avait une variété considérable tant des architectures matérielles que des systèmes d'exploitation. Il suffit pour s'en convaincre de lire l'excellent ouvrage collectif *Systèmes d'exploitation*

des ordinateurs publié en 1975 par une équipe baptisée CROCUS : c'est aujourd'hui un livre d'histoire passionnant, avec aussi beaucoup d'idées encore d'actualité, et de précieuses mises au point conceptuelles. Surtout au début, cette prolifération était due au moins en partie aux tâtonnements d'une technologie débutante, mais dès les années 1970 pratiquement toutes les solutions en usage aujourd'hui existent déjà : interruptions, multiprogrammation, mémoire virtuelle, machines virtuelles, réseaux à commutation de paquets, etc. La dernière vraie révolution architecturale remonte à la toute fin des années 1970, avec les processeurs RISC (*Reduced Instruction Set Computer*, cf. p. 276), qui vont dominer les années 1980 et 1990 (ils reviennent aujourd'hui au premier plan grâce aux téléphones, tous équipés de processeurs d'architecture ARM, cf. p. 278).

Pendant toute cette période, la recherche de puissance va stimuler la création d'architectures originales, SIMD *(Single Instruction Multiple Data)* et MIMD *(Multiple Instructions Multiple Data)*, destinées à dépasser les limites de l'architecture de von Neumann en permettant à l'ordinateur de faire plusieurs choses à la fois [1]. SIMD survit dans les cartes graphiques (GPU), de plus en plus utilisées pour tout autre chose que le graphisme, notamment pour les calculs d'algèbre linéaire. Les réalisations MIMD les plus spectaculaires furent les *Connection Machines* de *Thinking Machines Corporation* (on notera la modestie de la raison sociale).

Le défaut des architectures SIMD et surtout MIMD, c'est que leur programmation est compliquée, que ce soit pour le système d'exploitation ou pour les applications, elle remet en cause les connaissances des programmeurs. Alors, pendant les cruelles années 1990, où l'essor des microprocesseurs devint irrésistible, s'il fallait deux ou trois ans pour arriver à programmer sa *Connection Machine*, pendant ce délai les progrès des microprocesseurs standard avaient permis d'obtenir les mêmes performances par les méthodes classiques enseignées dans toutes les écoles. D'où la disparition de *Thinking Machines Corporation* (il faut dire que comprendre l'organisation physique de ces machines était déjà un vrai casse-tête).

Quant aux processeurs RISC, tels que MIPS, PA-RISC, Sparc, Alpha, Power, etc., bien plus élégants et plus simples que l'odieux Intel x86 (d'architecture CISC, *Complex Instruction Set Computer*), ils furent victimes des jeux vidéo et de la bureautique : dès que l'ordinateur personnel devint un objet de grande consommation, la compatibilité avec les logiciels déjà possédés par les clients devint un impératif inévitable. Heureusement vint le téléphone informatisé, pour lequel la consommation électrique des processeurs CISC était insupportable, ce qui mena à l'adoption des processeurs RISC d'architecture ARM (cf. p. 278).

1. Il ne m'échappe pas qu'à l'échelle microscopique un processeur moderne effectue plusieurs opérations à la fois, mais à l'échelle macroscopique la sémantique d'un traitement conforme à von Neumann est respectée.

Cette stabilité est-elle durable ?

Cette concentration de l'offre de systèmes d'exploitation, qui est en partie le fruit de la concentration de l'offre d'architectures matérielles, est-elle durable ? Rien n'est moins sûr. L'ébranlement des positions acquises risque de venir de l'industrie des semi-conducteurs, aujourd'hui à la croisée des chemins. Les investissements exigés par les technologies les plus récentes sont tellement élevés (plusieurs dizaines de milliards de dollars pour une ligne de production, et des milliards pour la recherche-développement en amont) que plus aucun acteur nouveau ne peut entrer sur le marché, et que chaque nouvelle étape voit un ou plusieurs industriels abandonner le terrain.

D'autre part, l'épaisseur du diélectrique des transistors des composants les plus récents est de l'ordre d'une dizaine d'atomes, on parle même de trois atomes dans les années à venir : on approche d'une limite physique. De même, il y a déjà une dizaine d'années que les industriels ont cessé d'augmenter la fréquence d'horloge, qui reste inférieure à 4GHz, parce que la dissipation thermique et la consommation électrique, déjà considérables, deviendraient insupportables.

Si la puissance du microprocesseur unitaire cesse de croître, d'où peut venir l'amélioration des performances ? de la mise en œuvre de procédés aptes à faire coopérer de multiples processeurs, ce qui pourrait remettre au devant de la scène les technologies des années 1980, mentionnées ci-dessus, destinées à dépasser les limites de l'architecture de von Neumann (SIMD, MIMD...). Pour tirer pleinement parti de telles architectures matérielles, il faudra innover en système d'exploitation, et pour cela, peut-être, relire les articles des années 1980 pour appliquer à l'échelle microscopique les solutions parfois très audacieuses qu'ils préconisaient à l'échelle macroscopique. On trouvera d'intéressantes perspectives sur les modèles de calcul d'avenir sous la plume d'Anil Madhavapeddy et David J. Scott (*Unikernels: The Rise of the Virtual Library Operating System* [83], cf. p. 298) ou sous celle de David Chisnall (*C Is Not a Low-level Language - Your computer is not a fast PDP-11* [31]).

Il faudrait penser aussi aux objets connectés, qui ont des caractéristiques bien particulières, mais de toute façon la recherche en système d'exploitation a encore de beaux jours devant elle.

D'autres livres, d'autres auteurs

Je n'aurais garde de conclure cet avant-propos sans signaler la richesse de la bibliographie relative aux systèmes d'exploitation. Le sujet est tellement riche qu'il y a de multiples façons de l'aborder, et c'est tellement passionnant que je vous conseille de tout lire. Pour me limiter aux auteurs qui ont écrit en français, outre le CROCUS [38] déjà mentionné, je citerai Samia Bouzefrane [22], Sacha

Krakowiak [71] et Patrick Cegielski [29]. Puisse leur lecture vous procurer autant de plaisir qu'à moi.

Avertissement de l'édition 2020

Vous pouvez adresser remarques et suggestions à l'auteur à l'adresse : `lb@laurentbloch.org`.

Ce texte est disponible en ligne sur le site de l'auteur :
`https://laurentbloch.net/MySpip3/Systeme-et-reseau-histoire-et-technique`
accompagné de textes complémentaires publiés régulièrement, sur des sujets voisins.

Remerciements

L'auteur de ce livre doit ici manifester sa gratitude à quelques personnes qui ont bien voulu l'encourager et l'aider. Dominique Sabrier m'a incité à écrire cet ouvrage : je crois qu'elle l'imaginait bien différent ; en tout cas son soutien a été déterminant. Christian Queinnec a bien voulu relire ce texte, l'améliorer par ses critiques et le doter d'une préface lumineuse. Manuel Serrano, Éric Gressier et le regretté François Bayen ont relu le manuscrit et suggéré des corrections et des améliorations importantes. Michel Volle, exposé à quelques états préliminaires du texte, a été un interlocuteur toujours stimulant. Jean-Marc Ehresmann a fourni quelques aperçus mathématiques toujours éclairants. Emmanuel Lazard m'a fourni les éléments qui ont permis de mettre à jour l'annexe consacrée aux circuits électroniques.

Ce livre a été entièrement réalisé avec des logiciels libres : TEX, LATEX et XELATEX pour la composition, BibLATEX pour la bibliographie, Emacs pour la frappe, Xindy pour l'index, xfig et Inkscape pour les figures, Linux pour la coordination de l'ensemble. La communauté du logiciel libre en soit remerciée ici.

Chapitre 1 Présentation des personnages

Sommaire

1.1 Mondanité des systèmes

Aujourd'hui chacun a entendu parler de Windows ou d'Android, peut-être moins de macOS (le système du Macintosh), et chaque jour un peu plus de Linux, grâce auquel l'auteur confectionne les lignes que vous avez sous les yeux. Ces entités quotidiennes s'appellent des systèmes d'exploitation. Bon gré mal gré, une proportion de plus en plus grande des personnes actives dans toutes sortes de domaines doivent acquérir une certaine familiarité avec celui qui anime leur ordinateur afin de pouvoir faire leur travail, et cette acquisition ne va pas sans perte de temps, agacement, colère, souffrance, mais aussi quand même joie et découvertes émerveillées.

Le présent ouvrage se propose d'apporter, au gré d'un voyage dans l'histoire de l'informatique, quelques éclaircissements sur les systèmes d'exploitation à un lecteur qui ainsi devrait se sentir moins désarmé face à eux, et de ce fait plus enclin à la sérénité face à ce que, souvent, il pense être leurs caprices. Il présente quelques aspects de leur nature, de leur naissance et de leur histoire, et cherche à éclairer, sous l'angle obtenu de ce point de vue, l'évolution et le rôle dans notre société de l'informatique et des ordinateurs auxquels elle est indissolublement liée.

Ce livre est une simple introduction à la science des systèmes d'exploitation, il s'adresse aussi au lecteur curieux d'histoire des sciences, des techniques et plus généralement de la vie intellectuelle, ainsi qu'au simple utilisateur d'ordinateur désireux de comprendre un peu mieux l'origine des difficultés mais aussi, nous l'espérons, des joies que lui procure cette extraordinaire machine. En fait, ce livre pourra aussi éclairer l'informaticien dont la spécialité n'est pas le système d'exploitation, au sens suivant : un livre destiné à de futurs ingénieurs en système devra viser une couverture complète des questions abordées ; au chapitre consacré par exemple aux algorithmes d'ordonnancement de processus, il devra décrire toutes les solutions possibles en donnant les détails nécessaires à leur réalisation ; mon propos ici est autre, il s'agit seulement de faire comprendre la problématique de ces algorithmes, en en exposant un, de préférence le plus simple, et sans entrer dans les détails d'implémentation. Et si le lecteur, passionné, veut en savoir plus, il trouvera dans la bibliographie quelques références d'ouvrages sensiblement plus épais qui devraient satisfaire sa curiosité.

D'excellents ouvrages accessibles à un public de non-spécialistes ont déjà été consacrés à l'histoire de l'informatique ou des ordinateurs ainsi qu'à la sociologie et à la psychologie de leur usage, mais la question des systèmes d'exploitation y occupe une place assez étroite. Ce n'est d'ailleurs pas anormal, puisque leur apparition est assez tardive dans l'évolution de l'informatique, mais la façon dont ils en affectent aujourd'hui tous les usages avec une intensité croissante me semble justifier une approche qui les prenne comme axe. Cela dit, je n'ai pas la prétention de traiter le sujet complètement, mais plutôt de partir de quelques problèmes choisis comme exemples. Et pour délasser le lecteur fatigué par des passages un peu techniques, le développement alternera descriptions d'objets techniques et analyses de certaines attitudes sociales qu'ils suscitent.

Cette invasion de nos activités par les systèmes d'exploitation, que chacun peut vérifier en observant la prolifération des titres informatiques dans les kiosques à journaux et les sujets de controverse dans les soirées en ville, est un phénomène récent. Au début des années 1970 l'auteur de ces lignes était ingénieur système, c'est-à-dire un spécialiste de ces objets techniques. Quand un convive dans un dîner lui demandait ce qu'il faisait dans la vie, il répondait « ingénieur système », profession de foi qu'il fallait faire suivre d'une explication, parce que la teneur de cette activité n'allait pas de soi. J'y ai vite renoncé.

1.2 Quelques définitions

Un système d'exploitation est un logiciel destiné à faciliter l'utilisation d'un ordinateur (je sais que cette assertion sera pour certains lecteurs une véritable provocation). L'ordinateur constitue le *matériel* composé de fils, circuits, etc., inutilisable sans le *logiciel* constitué de programmes. Les ordinateurs actuels

sont suffisamment complexes pour qu'il soit inconcevable de les utiliser sans la médiation d'un système d'exploitation, mais ce n'a pas toujours été le cas, et d'ailleurs certains petits ordinateurs qui vivent cachés à bord des fours à micro-ondes, des ascenseurs, des voitures ou des avions en sont parfois encore dépourvus.

Nous dirons qu'un ordinateur est un automate capable d'effectuer des actions dites primitives (c'est-à-dire déterminées par ses concepteurs, et que nous nommerons désormais « primitives » tout court), de les enchaîner dans l'ordre voulu, de les répéter et surtout de choisir, en fonction du résultat des actions précédentes, la prochaine action à effectuer entre deux ou plusieurs possibilités connues à l'avance.

L'ensemble des primitives d'un ordinateur donné constitue son *jeu d'instructions*. Chaque instruction est un objet physique, en l'occurence un circuit électronique, partie du processeur qui anime l'ordinateur. Aujourd'hui les processeurs sont des microprocesseurs, c'est-à-dire qu'ils sont réalisés par un circuit unique qui peut comporter jusqu'à trois milliards de transistors, mais jusqu'au début des années 1990 les processeurs des ordinateurs les plus puissants étaient constitués de circuits multiples implantés et interconnectés sur des cartes électroniques, parce que les microprocesseurs, apparus au début des années 1970, n'étaient pas assez puissants pour les grands ordinateurs de l'époque, ils étaient cantonnés aux micro-ordinateurs. La structure et le fonctionnement des circuits logiques à base de semi-conducteurs feront l'objet de l'annexe B p. 367.

Un programme est un texte qui énumère, dans le bon ordre, les primitives de l'ordinateur dont l'exécution mènera à la production du résultat recherché. C'est le texte du programme qui pilote la séquence des actions effectuées par l'ordinateur. Un logiciel est un ensemble de programmes. Un système d'exploitation est un programme dont la fonction principale est de déclencher l'exécution d'autres programmes, à bon escient de préférence.

Nous pouvons aussi dire les choses de la façon suivante : un ordinateur est une machine qui a des états discrets. Ces états sont enregistrés dans un dispositif appelé *mémoire*. Rédiger un programme pour un ordinateur, c'est décrire la séquence de ses états successifs qui vont permettre d'obtenir le résultat voulu. Le programme enchaîne les actions primitives qui vont affecter la mémoire pour instaurer les états voulus successivement. Un ordinateur est un automate à états.

Les trois caractéristiques fondamentales d'un ordinateur sont qu'il est programmable, automatique et universel :

- programmable : la nature des opérations à effectuer peut être spécifiée complètement et exclusivement par le texte d'un programme ;
- automatique : une fois lancée l'exécution d'un programme, l'ordinateur assure cette exécution sans intervention extérieure ;

– universel : capable d'exécuter n'importe quel programme, c'est-à-dire tout enchaînement d'actions primitives décrivant une procédure effective pour obtenir le résultat voulu. Une procédure effective est l'enchaînement d'opérations élémentaires qui permettront d'exécuter les calculs nécessaires à la solution de problèmes pour lesquels existent des solutions calculables (il y a des problèmes sans solution et des solutions incalculables ; les méthodes apprises à l'école pour faire des additions ou des multiplications à la main sont des procédures effectives). Traduire ces opérations élémentaires en termes d'actions primitives d'un ordinateur, c'est programmer.

Corollaire : si le programme lancé au démarrage de l'ordinateur est un système d'exploitation, conçu comme dit ci-dessus pour déclencher l'exécution d'autres programmes, qui sont ainsi en quelque sorte des sous-programmes du système d'exploitation, les exécutions de programmes différents pourront s'enchaîner. Nous y reviendrons.

Contrairement aux logiciels d'application, tels que traitement de texte, calcul scientifique, programme financier ou de jeu, le système d'exploitation ne sert pas à une tâche particulière, mais il est dans la coulisse de toutes. Dans la coulisse, c'est-à-dire que l'utilisateur peut ignorer jusqu'à son existence, et d'ailleurs cette ignorance est sans doute à mettre à son actif. Beaucoup d'utilisateurs de Macintosh ne savent rien de macOS, et c'est la preuve que macOS remplit sa mission sans faille. Quand le système d'exploitation (on peut dire simplement système) se manifeste, souvent c'est pour signaler que quelque chose ne va pas.

1.3 La couche visible du système

Pour l'utilisateur d'un micro-ordinateur, l'aspect le plus apparent (et souvent le seul perceptible) du système d'exploitation, ce sont les différents objets graphiques qui s'exhibent sur l'écran : les fenêtres dans lesquelles s'affichent des textes ou des images, les différentes barres et poignées qui servent à déplacer les fenêtres ou à en modifier les dimensions avec la souris, les barres de défilement qui permettent de faire défiler le contenu visible dans une fenêtre, les icônes sur lesquels on peut « cliquer » pour lancer l'exécution de tel ou tel programme, les menus qui proposent un choix de fonctions à exécuter, telles qu'imprimer le contenu d'une fenêtre ou… arrêter l'ordinateur. Cet aspect visible du système d'exploitation sert essentiellement à ouvrir, présenter et gérer des fenêtres et objets graphiques analogues : nous pouvons l'appeler *gestionnaire de fenêtres* ou interface utilisateur graphique (GUI, pour *Graphical User Interface*).

Mais au-delà de cet aspect immédiat, dont beaucoup de gens croient qu'il constitue le tout du système, il y a encore beaucoup de choses qu'un système d'exploitation fait pour son utilisateur. Nous avons dit que le système d'exploitation servait d'intermédiaire, si possible facilitateur, entre l'utilisateur et l'ordina-

teur. Dans cette optique, le gestionnaire de fenêtres est la partie la plus tournée vers l'utilisateur, l'aspect le plus superficiel du système (ici le mot superficiel ne dénote pas un jugement de valeur, mais qualifie ce qui est à la surface visible, par opposition à ce qui est enfoui dans la profondeur des entrailles de la machine). Cette partie qui est en surface, directement accessible à la perception, nous pourrons l'appeler interface (entre la personne et l'ordinateur). Nous dirons que c'est une interface de haut niveau, non pas là encore par un jugement de valeur, mais parce qu'elle donne de l'ordinateur et de son fonctionnement une représentation très idéalisée, très métaphorique, en un mot très abstraite par rapport à ce que serait une interface de bas niveau, plus proche de l'ordinateur et moins parlante aux humains.

1.4 Une représentation : le modèle en couches

À l'autre extrémité du système d'exploitation, si j'ose dire, du côté de l'ordinateur, de ses circuits et de ses transistors, sont les programmes qui interagissent directement avec les éléments matériels de l'ordinateur. Ces parties du système constituent ce qui est souvent appelé le *noyau (kernel)*. Entre le noyau et l'interface utilisateur, il y a toute une série de couches intermédiaires qui distillent les messages cryptiques du matériel pour délivrer à l'utilisateur une information compréhensible. Les informaticiens utilisent communément cette représentation par une architecture en couches imaginée par le chercheur néerlandais Edsger Wybe Dijkstra (1930–2002) dans un article fameux publié en mai 1968 par les CACM *(Communications of the Association for Computer Machinery)*, « *The structure of the THE multiprogramming system* » [46].

Avant d'être le plan de construction du système concret, l'architecture en couches est un modèle destiné à se représenter intellectuellement les choses par des abstractions. Ce modèle est utile pour « penser un objet dans lequel plusieurs logiques s'articulent » (Michel Volle, `http://www.volle.com/opinion/couches.htm`), lorsqu'il faut séparer différents niveaux d'abstraction. On nommera *couches basses* les parties du système qui interagissent le plus directement avec le matériel de l'ordinateur, et *couches hautes* celles qui sont plus proches de l'utilisateur. Il n'y a là encore aucun jugement de valeur implicite dans ces expressions « couches basses » et « couches hautes ». Et la réalisation des couches basses est sans doute techniquement plus complexe, demande des compétences plus rares que celle des couches hautes, cependant que ces dernières exigent, outre des compétences techniques, des talents artistiques et une imagination digne d'un urbaniste.

Encore plus que dans le monde des systèmes d'exploitation, le modèle en couches a connu le succès dans celui des réseaux informatiques, où les couches basses décrivent la transmission de signaux sur des supports physiques tels que câble téléphonique, faisceau hertzien ou fibre optique, tandis que les couches in-

termédiaires concernent l'acheminement de messages complexes à travers des réseaux à la topologie également complexe, et que les couches hautes traitent de la présentation de ces messages à travers une interface utilisateur, de l'identification de leur destinataire et de son authentification (ces derniers problèmes sont également traités par les systèmes d'exploitation, soit dit en passant). Les ensembles de règles et de conventions qui régissent les communications entre les couches de même niveau de plusieurs systèmes communicants constituent des *protocoles de communication*. Les règles et les conventions qui régissent les échanges entre une couche donnée et la couche immédiatement inférieure d'un même système constituent une *interface*.

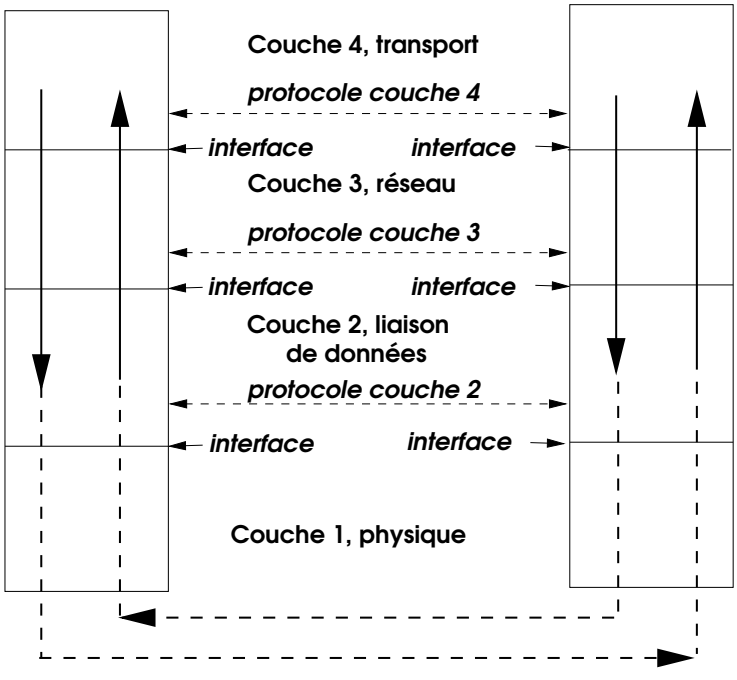


Figure 1.1 : *Exemple de modèle en couches : le réseau*

Soit un système dont les communications avec d'autres systèmes sont représentées par un modèle à n couches numérotées de 1 à n. La règle fondamentale du modèle en couche c'est que la couche de rang i ($1 < i < n$) d'un système donné ne peut communiquer concrètement qu'avec les couches $i-1$ et $i+1$ du même système. La couche 1 peut communiquer directement avec les couches 1 des autres systèmes. La couche n assure l'interface avec l'utilisateur. La couche i communique selon les règles du protocole qui lui correspond avec les couches i des autres systèmes, mais cette communication passe par la médiation des couches inférieures. Ceci est illustré par la figure 1.1 : dans un réseau

informatique, la couche 1 (physique) concerne les problèmes de câblage ou de liaison hertzienne, les éléments de la couche 2 assurent la communication entre deux stations du réseau reliées directement par un même segment de la couche 1, la couche 3 est chargée de calculer un itinéraire dans un réseau complexe, la couche 4 vérifie que le transport de bout en bout des données s'effectue sans erreur et dans l'ordre (ceci sera étudié en détail au chapitre 6 p. 131).

1.5 L'informatique est (aussi) une science

Compléter les définitions ci-dessus par un plan d'ensemble de la discipline informatique n'est peut-être pas si futile qu'il y paraît et peut aider, dans la suite, à savoir de quoi l'on parle. La figure 1.2 propose un tel plan.

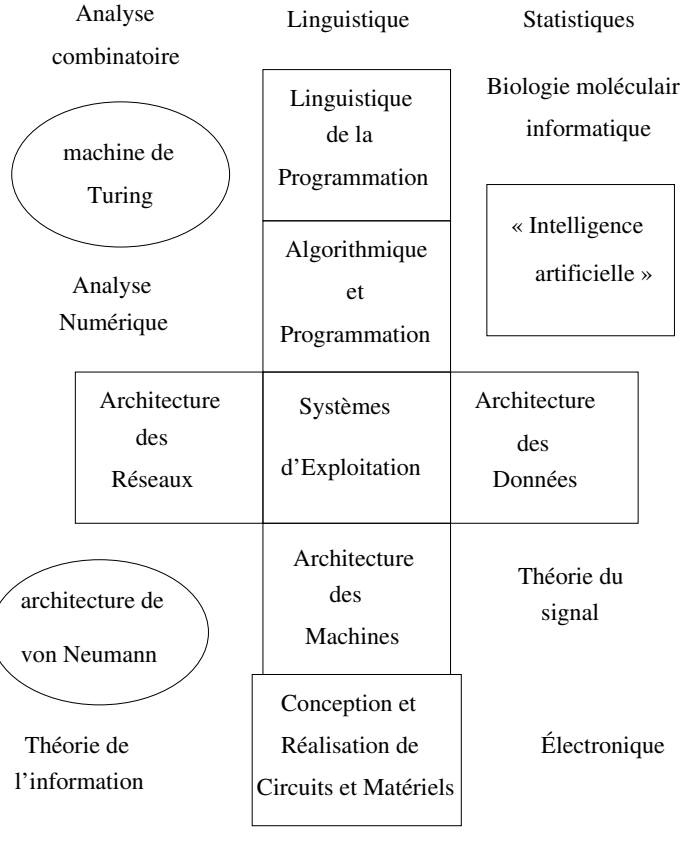


Figure 1.2 : *Les disciplines de l'informatique*

Au sein de l'informatique existent des disciplines dont la nomenclature n'est pas aussi bien établie que celles des mathématiques, certes, et la classification

proposée ici est probablement contestable. Dans la partie d'un seul tenant de mon plan en croix j'ai mentionné ce qui me semblait constituer les sept disciplines majeures, et la proximité dans le plan se veut refléter la proximité des disciplines : vers le haut, le plus abstrait, vers le bas, on se rapproche de la matière.

J'ai mis l' « intelligence artificielle » entre guillemets et dans un cadre un peu à l'écart parce que cette discipline me semble, au moins, très mal nommée. Les ellipses contiennent les noms des grands paradigmes qui donnent les clés de l'informatique. Flottant libres de cadres sont les noms de disciplines qui, pour avoir des proximités avec l'informatique, n'en font pas partie, même si parfois elles lui sont rattachées par erreur.

Le lecteur familier d'une autre science sera sans doute surpris par cette description, surtout s'il vit et travaille en France. En effet, dans ce pays où l'informatique a du mal à se faire reconnaître comme une discipline à part entière par les institutions de l'élite, chacun l'imagine selon les applications (ou les représentations) qui en sont faites dans son domaine. Ainsi le physicien a tendance à la réduire au calcul, tant cette activité est pour lui associée à l'informatique. Le biologiste s'étonnera de l'absence de l'analyse de séquences dont il croit qu'elle est partie constituante de l'informatique, tandis que le mathématicien imagine qu'elle se réduit à l'algèbre du monoïde libre et que le statisticien n'y voit que l'analyse en composantes principales. Tous ces domaines sont liés d'une façon ou d'une autre à l'informatique, mais n'en font nullement partie. Quant aux parties de l'informatique qui sont indépendantes, lorsque leur existence n'est pas purement ou simplement ignorée, elles sont considérées comme des techniques rudimentaires, alors que ce sont souvent les domaines les plus complexes.

1.6 Architectures

Les lignes qui précèdent recourent à plusieurs reprises au terme architecture. Il s'agit de plus que d'une commodité de langage, et d'autre chose que d'un fantasme d'informaticien frustré de travailler toujours dans l'immatériel et l'abstrait et envieux du matériau concret palpé par le bâtisseur.

Lorsque l'architecte conçoit un bâtiment, ou l'urbaniste un quartier, leur préoccupation première, avant de concevoir chaque pièce, ou chaque pâté de maisons, porte sur les moyens de circuler entre ces éléments : couloirs, escaliers, halls, vestibules, ou rues, allées, places, ponts, esplanades.

Il en va de même pour le concepteur d'un système informatique, qu'il soit matériel ou logiciel. Comment l'information circule-t-elle entre le disque dur et la mémoire centrale, de celle-ci au processeur ? Comment les différents programmes d'un logiciel partagent-ils des données, ou au contraire en garantissent-ils l'accès exclusif ? Voilà les grandes questions de la conception informatique, et au cœur de ces questions se trouve le grand aiguilleur, le grand régulateur : le

système d'exploitation. L'ouvrage de Hennessy et Patterson *Architecture des ordinateurs* [58] donnera au lecteur une vision approfondie de ces questions et de certaines réponses qui leur ont été apportées.

Le « plan de masse » des disciplines informatiques que nous avons dessiné à la figure 1.2 renvoie à cette analogie architecturale.

1.7 Enjeux d'une histoire

L'histoire de l'informatique peut être considérée sous différents angles : histoire de la pensée scientifique, histoire des techniques de calcul automatique, histoire des ordinateurs. Ces trois éclairages (il y en aurait d'autres) illumineraient une scène qui se joue depuis plus longtemps qu'il n'est envisagé communément, la poursuite du rêve prométhéen de construire un être mécanique pensant.

Si ce rêve, on l'imagine volontiers, a mobilisé les ingénieurs les plus inventifs, de Ctésibios [1] d'Alexandrie à Steve Jobs, on néglige souvent l'engagement dans cette voie de philosophes et savants parmi les plus éminents, tels Blaise Pascal, Gottfried Wilhelm Leibniz, John von Neumann, ou on le sous-estime en réduisant cette part de leur œuvre à un passe-temps. En fait Leibniz par exemple est le précurseur de l'informatique moderne par trois contributions majeures (et ignorées pendant plus de deux siècles) : la réalisation concrète d'une machine capable d'effectuer les quatre opérations arithmétiques, la conception d'une *caractéristique universelle* qui préfigure les systèmes formels de la logique moderne et, partant, les langages de programmation, enfin l'étude, sur des documents chinois, de l'arithmétique binaire, dont il perçoit et analyse la simplicité qu'elle peut conférer à un système de calcul automatique [2]. De même von Neumann est plus respecté pour la théorie des jeux ou ses travaux de mathématiques pures que pour son invention de l'ordinateur, la plus importante du XX^{e} siècle.

D'autres précurseurs sont quant à eux tout simplement sous-estimés comme hommes de science. Ce n'est que récemment que la véritable stature de Charles Babbage (1791–1871) est apparue : longtemps considéré comme l'ingénieur maniaque de machines à calcul qui n'ont jamais marché, il avait en fait conçu avec sa machine analytique un ordinateur complet auquel n'a manqué que la réalisation ; par ailleurs son rôle dans le progrès considérable des mathématiques anglaises au XIX^{e} siècle est désormais reconnu. La même reconnaissance échoit finalement à Alan Turing, malgré le dédain ou l'ignorance volontaire où les mathématiciens, spécialement en France, relèguent tant la logique que les recherches

1. Inventeur d'une clepsydre perfectionnée et d'un orgue à eau.

2. Cf. le texte de Leibniz, écrit en français : `https://laurentbloch.net/MySpip3/L-arithmetique-binaire-par-Leibniz-98` et un exposé des contributions leibniziennes : `https://laurentbloch.net/MySpip3/L-informatique-a-la-lumiere-de-quelques-textes-de-Leibniz`

sur les Fondements initiées par David Hilbert et Kurt Gödel. Bref, l'informatique sera bientôt vraiment reconnue comme une science, peut-être…

Il faut dire que nous revenons de loin. Si nous sommes revenus. Assez récemment, à la fin du siècle dernier, j'ai visité le très beau *Museum of Science and Industry* de Chicago. La salle consacrée à l'informatique, grande et belle, illustrait le déclin fatal infligé à la dimension spectaculaire des ordinateurs par la miniaturisation électronique. Cela dit, malgré le panneau *Computer Science* à l'entrée, qu'il s'agisse là d'une science n'était guère patent : on voyait des exploits d'ingénieurs, ce qui d'ailleurs donnait la part belle aux précurseurs français Blaise Pascal et Thomas de Colmar qui partageaient avec Wilhelm Schickard et Charles Babbage le stand des précurseurs d'avant l'électricité, des réalisations industrielles, mais nulle mention qu'il y eût aussi, dans ce domaine, un paradigme, des théories, des concepts, certes plus difficiles à mettre en vitrine que des disques durs. On pourra m'objecter que c'est aussi un musée de l'Industrie, mais bon, il y avait bien des salles de mathématiques…

Cette salle qui se disait de *Computer Science* était en fait de machines à calculer et d'ordinateurs, ce qui n'est qu'un versant de la chose. L'ordinateur apparaissait comme le fruit d'une évolution quasi darwinienne prenant son origine chez les machines mécaniques, passant aux machines mécanographiques des années 1890 - 1950, puis aux grands calculateurs électroniques pré-informatiques, pour aboutir avec l'ENIAC[3] à l'ordinateur.

Cette vision de l'histoire informatique est assez répandue, mais assez discutable et surtout assez tronquée. À la sortie de cette salle du musée une pancarte donnait un indice sur la nature du biais donné à la présentation et sur l'orientation de la vision qui avait pu l'engendrer : tout ce que j'avais vu était un cadeau de la compagnie IBM. Je ne partage pas avec certains de mes collègues universitaires[4] le dédain pour les réalisations informatiques d'IBM, qui a été à l'origine d'innovations majeures, comme le disque magnétique, le processeur RISC, la multiprogrammation, le processeur en pipeline et bien d'autres. Mais il est inévitable qu'une compagnie dont les origines sont dans la mécanographie soit sujette à y voir la naissance de l'informatique, plus que dans les recherches menées à l'IAS *(Institute for Advanced Studies)* de Princeton par des théoriciens issus d'horizons différents, en l'occurrence Church, von Neumann et Turing.

3. C'est ainsi que le présente le musée, mais l'ENIAC ne répond pas à la définition de l'ordinateur que nous avons donnée ci-dessus. Nous préciserons cela au chapitre suivant, ainsi qu'à l'annexe C.

4. Dans cette locution le mot « universitaires » n'est pas un adjectif, épithète de « collègues », mais un substantif en apposition à « collègues ». Je ne suis pas universitaire, mais les universitaires informaticiens sont mes collègues en informatique. La collégialité se déploie sur plusieurs plans.

Chapitre 2 Principe de fonctionnement de l'ordinateur

Sommaire

Introduction

Nous avons défini le système d'exploitation d'abord comme un logiciel destiné à faciliter l'utilisation d'un ordinateur, puis comme un programme dont la fonction principale est de déclencher l'exécution d'autres programmes. Nous allons le définir maintenant comme un programme qui permet à un ordinateur de faire plusieurs choses à la fois.

Pour pouvoir élaborer et nuancer cette définition, et comprendre notamment ce qu'elle comporte de paradoxe, il nous faut approfondir un peu notre vision de l'ordinateur que nous avons défini comme un automate capable d'effectuer des actions dites primitives (déterminées par ses concepteurs) selon l'énumération qu'en donne le texte d'un programme.

2.1 Modèle de l'ordinateur

C'est John von Neumann, mathématicien hongrois émigré aux États–Unis, qui dans un document tout à fait remarquable de 1945 intitulé *First Draft of a Report on the EDVAC* et désormais disponible en ligne [98] a proposé pour l'ordinateur l'architecture représentée par la figure 2.1 [1].

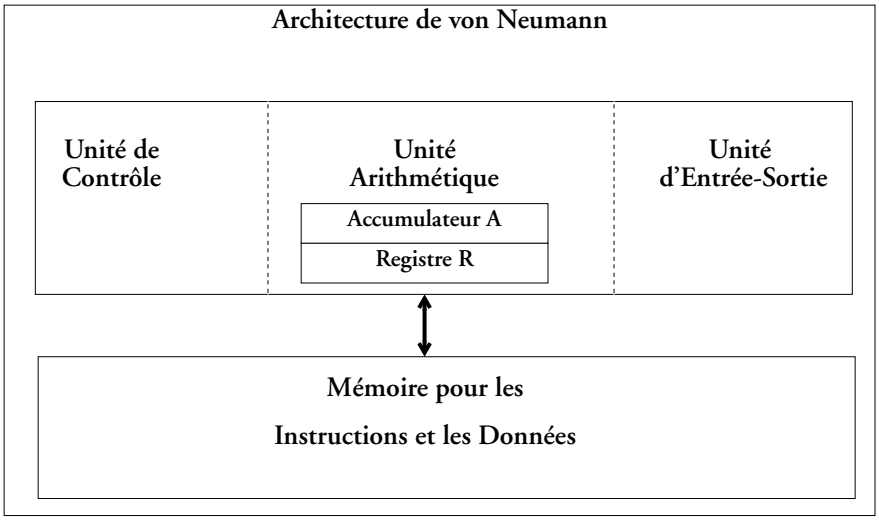


Figure 2.1 : *Structure de l'ordinateur*

Les unités de contrôle [2], arithmétique et d'entrée–sortie constituent à elles trois l'unité centrale, ou le *processeur* de l'ordinateur. Le processeur est constitué de circuits électroniques qui peuvent exécuter des actions ; de nos jours il est généralement réalisé sous la forme d'un composant électronique unique nommé *microprocesseur*. L'ensemble des actions « câblées » dans le processeur constitue le jeu d'instructions du processeur (les « actions primitives ») et détermine le langage élémentaire de son utilisation, appelé « langage machine ». À chaque instruction identifiée par son code correspond un circuit particulier.

Le rôle de l'unité de contrôle consiste à permettre le déclenchement de l'action (l'instruction) voulue au moment voulu. Cette instruction peut appartenir à l'unité arithmétique, à l'unité d'entrée-sortie ou à l'unité de contrôle elle-même. Une instruction peut en outre consulter le contenu de la mémoire (la « lire ») ou

1. La figure et les deux alinéas qui suivent sont empruntés à mon livre *Initiation à la programmation et aux algorithmes avec Scheme*, publié en 2020 par les Éditions Technip, avec l'aimable autorisation de l'éditeur.

2. Une traduction plus exacte de l'anglais *control unit* serait « unité de commande ». L'usage a entériné l'anglicisme.

modifier le contenu de la mémoire (y « écrire »). De façon générale, une action consiste soit à consulter ou à modifier l'état de la mémoire ou d'un des registres [3] A ou R (qui sont des éléments de mémoire spéciaux incorporés à l'unité centrale), soit à déclencher une opération d'entrée-sortie (communication avec le monde extérieur et notamment l'utilisateur humain), soit encore à modifier la séquence des instructions formulées par le programme en commandant de « sauter » un certain nombre d'instructions sans les exécuter, ou de « revenir en arrière » pour répéter des instructions déjà déroulées (le texte du programme n'est pas modifié, mais est modifié l'ordre dans lequel il est « lu »).

Point fondamental, un ordinateur conforme au modèle de von Neumann exécute une instruction, et une seule, à la fois (principe d'exécution séquentielle). En ce début de vingt-et-unième siècle, pratiquement tous les ordinateurs se conforment extérieurement, macroscopiquement à ce modèle. De nombreux perfectionnements techniques destinés à améliorer les performances contournent cette règle et permettent qu'à un instant donné une soixantaine d'instructions soient en cours d'exécution sur un cœur de processeur, mais cela ne modifie en principe ni le modèle d'exécution ni la sémantique du traitement de l'information (nous donnons au chapitre 9 p. 265 une description de ces techniques, qui ont fait l'objet d'un article acerbe de David Chisnall [31]).

Comment indique-t-on à l'unité de contrôle le « moment voulu » pour déclencher telle ou telle action ? C'est écrit dans le texte d'un programme. Où est le programme ? Dans la mémoire.

La mémoire est constituée d'éléments susceptibles de prendre des états. Un élément de base de la mémoire peut prendre deux états distincts et peut servir à représenter une information élémentaire, ou *bit* (*binary digit,* chiffre binaire). Cette représentation d'une information par un élément de mémoire s'appelle un code. Une mémoire avec beaucoup de bits permet le codage d'informations complexes, dans la limite de la taille de la mémoire.

Comme les constituants élémentaires de la mémoire ont deux états, il est commode d'utiliser la numération binaire pour les représenter et pour effectuer des calculs à leur sujet. À l'époque de la scolarité de l'auteur, les systèmes de numération étaient introduits en classe de cinquième, mais je me suis laissé dire que cette introduction n'était plus systématique. L'annexe A en donne les rudiments.

Si les constructeurs des premiers ordinateurs avaient imaginé des constituants élémentaires à trois états, l'informatique aurait-elle été ternaire plutôt que binaire ? En fait tout laisse supposer que l'extraordinaire développement de l'informatique doit beaucoup à la grande simplicité de la numération binaire.

3. Le registre A est souvent appelé *accumulateur*, pour son usage dans certains calculs.

Gottfried Wilhelm von Leibniz déjà l'avait conçu[4]. Les nombreuses tentatives pour développer des machines décimales ont été décevantes, sauf pour les calculettes. Et même si la technique fournissait aujourd'hui des composants ternaires économiquement intéressants, il y a fort à parier que l'informatique resterait binaire pour profiter de la simplicité, de l'uniformité et de la régularité, en un mot de l'élégance, des modèles formels qui lui donnent sa charpente.

Comme le bit est une unité d'information trop élémentaire pour la plupart des usages, on manipule ordinairement des *mots* de mémoire, constitués d'un nombre donné de bits (32 ou 64 usuellement). La taille du mot est une caractéristique importante de l'architecture d'un ordinateur. On peut se représenter ces mots comme rangés dans un grand tableau de cases numérotées. Le numéro de chaque case est l'*adresse* du mot qu'elle contient.

Le chemin par lequel unité centrale, mémoire et organes d'entrée-sortie communiquent s'appelle de façon générique un « bus ». De façon un peu formelle, un bus est un graphe connexe complet, ce qui veut dire en langage courant que tous les éléments connectés au bus peuvent communiquer entre eux.

Quel fut le premier ordinateur ?

Comme suggéré par le titre de son rapport, *First Draft of a Report on the EDVAC* [98], les principes qu'y exposait von Neumann étaient destinés à s'appliquer à la construction d'une machine nommée *EDVAC* qui aurait été la première réalisation de l'architecture dite depuis de von Neumann. Il en fut autrement.

Comment von Neumann, mathématicien réputé à la position scientifique bien établie dans plusieurs domaines, de la théorie des ensembles au calcul des probabilités, en était-il venu à s'intéresser au calcul automatique ? D'abord, il avait fréquenté Alan Turing à l'IAS *(Institute for Advanced Studies)* de Princeton de 1936 à 1938 et il connaissait ses travaux. Plus tard, Herman H. Goldstine a raconté dans son livre [54] comment, en 1943, alors qu'il était « scientifique du contingent » dans l'U.S. Navy et qu'il travaillait au projet de calculateur ENIAC destiné aux calculs balistiques des canons de marine, il avait aperçu von Neumann sur le quai de la gare d'Aberdeen (Maryland), avait osé l'aborder et lui avait parlé de son travail. Von Neumann avait été immédiatement passionné et s'était joint au projet.

Le projet ENIAC (pour *Electronic Numerical Integrator and Computer*) devait produire une grande machine à calculer et il avait été mis en route à l'Université de Pennsylvanie en 1943 sous la direction de J. Presper Eckert et de John W. Mauchly. Une fois réalisé (à la fin de 1945), l'ENIAC serait le

4. `https://laurentbloch.net/MySpip3/L-arithmetique-binaire-par-Leibniz-98`

plus grand calculateur de son temps. Mais l'ENIAC ne répondait pas à la définition que nous avons donnée de l'ordinateur : une machine programmable, automatique et universelle. La réalisation d'un calcul avec l'ENIAC demandait des interventions manuelles pour adapter la configuration de la machine, ce qui va à l'encontre de l'exigence d'être automatique et programmable. En fait la programmation était réalisée essentiellement au moyen de commutateurs et de tableaux de connexions, comme sur les machines mécanographiques. C'est en pensant aux moyens d'améliorer ce fonctionnement que von Neumann a conçu son architecture.

Plus tard, Eckert et Mauchly ont accusé von Neumann d'avoir pillé leurs idées, mais cette thèse ne résiste pas à la simple lecture du *First Draft of a Report on the EDVAC*. Il est légitime de dire qu'entre l'EDVAC et l'ENIAC il y a une différence du même ordre qu'entre la lunette de Galilée et les lunettes réalisées auparavant par un Hollandais anonyme : Galilée a certes bénéficié de l'exemple de son prédécesseur, mais, comme l'a souligné Alexandre Koyré, sa lunette est la réalisation d'une théorie scientifique, alors que l'instrument de son prédécesseur était le fruit d'une démarche empirique. Le texte de von Neumann est un des fondements (avec la machine de Turing) d'une science nouvelle. La construction de l'EDVAC prendra du retard, et la première machine de von Neumann sera britannique.

En 1995 de grandes manifestations ont été organisées aux États-Unis pour célébrer le cinquantenaire de l'ENIAC comme celui du premier ordinateur, mais c'était abusif. L'ENIAC représente sans doute l'apogée des calculateurs pré-informatiques.

Avant von Neumann, programmer, c'était brancher des fils sur des tableaux de connexion ; à partir de lui, ce fut écrire des textes (Samuel Goyet, séminaire Codes Source*).*

En fait les premiers ordinateurs véritables furent le *MARK 1* de l'Université de Manchester, réalisé sous la direction de Max Newman, opérationnel en 1948, et l'EDSAC, construit à l'Université de Cambridge sous la direction de Maurice Wilkes en 1949. Les querelles d'antériorité entre ces deux machines ne sont et ne seront sans doute pas tranchées, mais le fait que cela se joue entre elles n'est guère remis en cause. Les principes à la base de ces deux machines avaient incontestablement été élaborés par John von Neumann aux États-Unis, la théorie sous-jacente était celle du Britannique Alan Turing, mais les réalisation étaient britanniques, d'où sans doute la tentation d'une usurpation commémoratrice américaine…

Cette question de primauté ou pas de l'ENIAC est loin d'être un détail. Selon la réponse qu'on lui donne :

– l'ordinateur, au sens moderne du terme a été inventé par Eckert et Mauchly, ou par von Neumann ;

– la première réalisation est américaine, ou britannique ;
– l'informatique est née de l'évolution technique normale des machines mécanographiques, ou d'une rupture épistémologique dont la source se trouve dans la recherche fondamentale en mathématiques ;
– l'informatique est un bricolage d'ingénieur astucieux, ou une percée intellectuelle de première importance.

2.2 Traitement de l'information

Construire des ordinateurs, puis écrire des programmes : le but poursuivi par ces activités est de traiter de l'information. Traiter de l'information c'est, à partir de données que nous conviendrons de nommer $\mathcal{D}$, leur faire subir le traitement décrit par le programme P pour obtenir le résultat $\mathcal{R}$.

Ceci est très général : $\mathcal{D}$ peut être une liste de nombres dont nous voulons faire l'addition, $\mathcal{R}$ sera alors le nombre qui représente leur somme, et il faudra écrire le programme P de sorte qu'il mène au résultat. Mais $\mathcal{D}$ peut aussi être le manuscrit du texte que vous êtes en train de lire, $\mathcal{R}$ le même texte mis en page et P devra alors être un programme typographique.

Le traitement devra être réalisé par un *exécutant* dont nous n'avons pas besoin de supposer qu'il est un ordinateur, à ce stade du raisonnement. Simplement, si l'exécutant est un humain doté d'un crayon et d'un papier, les méthodes pour calculer la somme d'une liste de nombres ou pour mettre en page un manuscrit ne seront pas les mêmes que pour un ordinateur. L'être humain et l'ordinateur ne sont pas aptes aux mêmes actions primitives (nous dirons désormais primitives tout court). L'être humain, de surcroît, est capable d'inventer à tout moment de nouvelles primitives, et d'en oublier d'autres. L'ordinateur est bien plus simple.

Le traitement des données doit être *effectif*, c'est-à-dire que la méthode de passage doit pouvoir aboutir pratiquement au résultat. Prenons un exemple, soit $\mathcal{D}$ l'ensemble des numéros d'immatriculation des voitures immatriculées en France[5]. Les symboles utilisés sont des chiffres et des lettres. Les règles de formation des numéros d'immatriculation sont la syntaxe du langage. La sémantique d'un numéro est l'identité de la voiture qui le porte. Considérons $\mathcal{R}$, l'ensemble des départements français. La correspondance qui, à tout numéro d'immatriculation bien formé, fait correspondre le département d'immatriculation de la voiture qui le porte est un traitement de $\mathcal{D}$, on sait comment le réaliser.

5. Pour cet exemple j'utilise les anciens numéros, antérieurs à avril 2009. Les nouveaux numéros en vigueur depuis cette date sont dépourvus de signification, et forment de ce fait un langage moins... significatif.

En revanche la correspondance qui, à tout numéro d'immatriculation bien formé, ferait correspondre la commune d'immatriculation de la voiture associée n'est pas un traitement de $\mathcal{D}$, car on ne sait pas élaborer cette information à partir du numéro, bien qu'elle existe sûrement ; les données qui permettraient de la calculer ne font pas partie de $\mathcal{D}$. Cette correspondance ne peut pas être décrite de manière effective, c'est-à-dire par un *algorithme*.

Un algorithme est la description, pour un exécutant donné, d'une méthode de résolution d'un problème, autrement dit d'une suite d'opérations qui fournissent le résultat cherché.

La description de la méthode, c'est-à-dire l'algorithme, doit être adaptée à l'exécutant chargé d'effectuer le traitement. L'exécutant sait effectuer un nombre fini d'actions, que l'on nomme ses primitives (ce sont par exemple les instructions du jeu d'instructions du processeur décrit ci-dessus). L'algorithme doit être constitué d'une combinaison de ces primitives. Pour construire toutes les combinaisons possibles de primitives, Corrado Böhm et Giuseppe Jacopini ont démontré dans un article célèbre des CACM en mai 1966 intitulé « *Flow diagrams, Turing machines and languages with only two formation rules* » [21] qu'il suffisait de savoir réaliser l'enchaînement de deux primitives (effectuer une action à la suite d'une autre), la répétition d'une primitive donnée et le choix, pendant l'exécution d'un traitement, entre deux primitives selon le résultat d'un test. Ce qui « sait » réaliser ces combinaisons de primitives, c'est en dernière analyse l'unité de contrôle du modèle de von Neumann.

Un algorithme fournit, pour un exécutant donné, la décomposition de la tâche à réaliser en primitives de l'exécutant.

2.3 Mémoire et action, données et programme

Nous avons dit que le programme résidait dans la mémoire : ce point mérite d'être souligné. C'est l'innovation principale du modèle de von Neumann, cette invention porte le nom de « machine à programme enregistré ».

De toute évidence, lorsque l'on y réfléchit maintenant, un programme c'est de l'information. De l'information sur le traitement à appliquer à l'information, de la méta-information, si l'on veut. Mais à l'époque de von Neumann le concept de programme était beaucoup moins bien formé qu'aujourd'hui. Dans les machines à calculer telles que l'ENIAC le programme n'existait nulle part en tant que texte, il était réparti entre différents tableaux de connexions et autres commutateurs, d'où finalement le défaut de programmabilité de cette machine, et en tout cas l'impossibilité de raisonner sur son programme.

L'idée (géniale) de von Neumann consiste à dire que programme et données seront enregistrés dans la même mémoire. Chaque case de mémoire peut contenir un mot, ce mot peut représenter soit une instruction, soit un élément de donnée. Un élément de donnée peut être un nombre ou un caractère alphabétique,

par exemple, ou encore un nombre dont la valeur a une signification particulière : l'*adresse* d'une autre donnée ou d'une instruction, c'est-à-dire le numéro de la case mémoire où se trouve cette donnée ou cette instruction.

Les instructions ne se réduiront pas à d'obscures connexions de fils et de circuits, mais seront d'abord des éléments de signification (des verbes, oserai-je dire) qui diront des choses très simples (bien sûr, l'unité de contrôle pourra, au vu de chacun de ces verbes, sélectionner le circuit logique correspondant).

Dans le paragraphe précédent il y a une locution qui demande éclaircissement : *au vu*. Comment l'unité de contrôle peut-elle *voir* le verbe ? Pour l'instant nous dirons simplement que les circuits logiques de l'unité de contrôle comportent des « aiguillages », qui selon la valeur d'un ensemble de bits (le *code opération* de l'instruction) déclencheront le fonctionnement du circuit correspondant[6].

2.4 À quoi ressemble le langage machine ?

2.4.1 Premier programme

Voici un exemple de programme très simple, énoncé d'abord en langage humain et illustré par la figure 2.2 p. 27 :

1. charge dans le registre A (aussi appelé *accumulateur*) le nombre qui est dans la case mémoire numéro 20 ;
2. teste le contenu de A : s'il vaut zéro passe directement à l'instruction 6 ; sinon ne fais rien, c'est à dire continue en séquence ;
3. additionne au contenu du registre A le nombre qui est dans la case mémoire numéro 21 (le résultat effacera l'ancien contenu du registre A et prendra sa place) ;
4. copie le contenu du registre A dans la case mémoire numéro 22 (le résultat effacera l'ancien contenu de la case mémoire numéro 22 et prendra sa place) ;
5. imprime le contenu de la case mémoire numéro 22 ;
6. fin.

La figure 2.2 montre les déplacements de données en quoi consiste ce programme ; on se rappellera que les registres ne sont rien d'autre que des positions de mémoire spéciales placées dans les circuits du processeur pour que les traitements qui les affectent soient plus rapides ; les numéros des instructions concernées figurent dans des cercles à côté des flèches qui indiquent les mouvements de données.

Chaque élément de la liste ci-dessus décrit en langage humain une instruction d'ordinateur. Dans la mémoire, ces textes sont codés sous forme

6. La description du fonctionnement des circuits logiques a été reportée en annexe B.

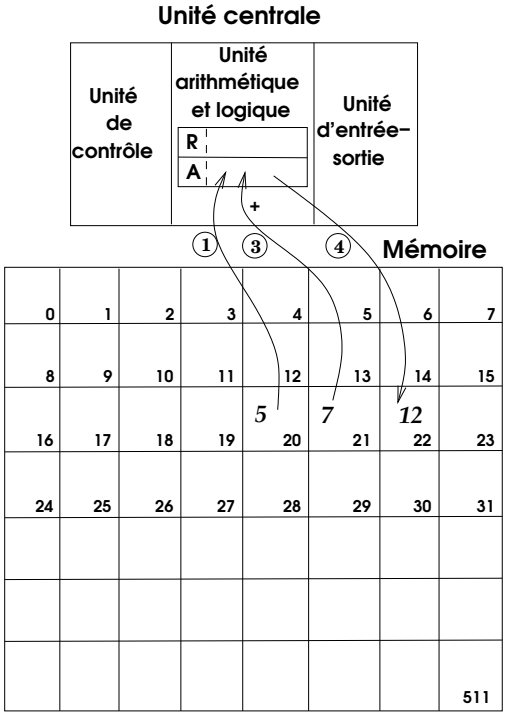


Figure 2.2 : *Exécution de programme*

d'un certain nombre de bits selon un format fixe. Par exemple, supposons que nous disposons de 16 bits pour représenter une instruction (notre ordinateur a des mots de 16 bits), nos instructions pourront avoir le format suivant :

bits 0 à 4	bit 5	bit 6	bits 7 à 15
code opération	numéro du premier registre concerné (0 pour R, 1 pour A)	numéro du second registre concerné (0 pour R, 1 pour A)	adresse mémoire

instruction	code opération	nom mnémonique
chargement mémoire → registre	00001	LOAD
copie registre → mémoire	00010	STORE
addition mémoire à registre	00011	ADD
imprimer mémoire	00100	PRINT
test registre et branchement si zéro	00101	BZ
fin	00110	END

La colonne intitulée « nom mnémonique » nous sera utile ultérieurement, pour l'instant disons qu'elle peut nous servir à désigner commodément les

instructions. Avec ce codage, et en supposant que la première instruction est chargée en mémoire à l'adresse 0 (les numérotations des informaticiens commencent souvent à 0) notre petit programme s'écrit en binaire :

code opération	1^{er} registre	2^{nd} registre	adresse (en binaire)	adresse (en décimal)
00001	1	0	0 0001 0100	20
00101	1	0	0 0000 0101	5
00011	1	0	0 0001 0101	21
00010	1	0	0 0001 0110	22
00100	1	0	0 0001 0110	22
00110	1	0	0 0001 0110	22

soit, une fois éliminées les fioritures à l'usage du lecteur humain :

```
0000  1100  0001  0100
0010  1100  0000  0101
0001  1100  0001  0101
0001  0100  0001  0110
0010  0100  0001  0110
0011  0100  0001  0110
```

L'unité de contrôle va chercher ces instructions l'une après l'autre et déclencher leur exécution. Le décodage par l'unité de contrôle du code opération (5 premiers bits) permet d'activer le circuit logique qui correspond à l'instruction désirée.

Nous donnons à la sous-section 9.2 du chapitre 9 des détails plus fins sur l'exécution des instructions, mais le modèle simple ci-dessus permet de comprendre l'essentiel.

2.4.2 Questions sur le programme

Ce programme peut susciter quelques interrogations :

- Que signifie la valeur d'adresse 5 (décimal) dans la seconde instruction ? Ceci : si la première instruction est chargée, nous le supposerons pour cet exemple, dans la case mémoire 0, la sixième sera dans la case 5. L'adresse, on l'a vu, n'est pas autre chose que le numéro de case mémoire. Et l'instruction « test registre et branchement » ordonne, si le test est positif, que le déroulement du programme soit modifié pour se continuer par l'exécution de l'instruction qui se trouve à cette adresse (pour « sauter » à cette adresse). Si le test est négatif, l'exécution se poursuit par l'instruction suivante, comme si de rien n'était.
- Pourquoi nos instructions ont-elles deux champs « registre » alors qu'elles utilisent un ou zéro registre ? Parce que le format des instructions de notre

ordinateur fictif est fixe, et qu'il est prévisible qu'il aura besoin d'instructions qui concernent deux registres, par exemple une addition registre à registre. Il nous serait évidemment loisible d'imaginer un ordinateur dont le format d'instruction soit différent, mais il faut bien faire un choix. Ce qui est sûr, c'est qu'un format d'instruction uniforme et de longueur fixe simplifie considérablement l'unité de contrôle, et que les ordinateurs réels contemporains sont ainsi conçus[7] (il y a longtemps, des processeurs avaient des instructions de longueurs diverses et même variables : il s'est avéré que les inconvénients outrepassaient largement les avantages).

– Pourquoi les instructions numéro 5 et 6, qui ne font référence à aucun des deux registres A et R, ont-elles une valeur dans chaque champ registre, en l'occurrence 1 qui désigne A et 0 qui désigne R ? Parce que le format des instructions de notre ordinateur fictif est fixe, que chacune de ses instructions a par hypothèse deux champs « registre » et qu'il faut bien qu'un bit vaille 0 ou 1.

2.5 Mot d'état de programme (PSW)

Nous avons écrit à plusieurs reprises des locutions telles que « le programme saute à telle adresse pour exécuter l'instruction qui s'y trouve » : précisons ici cette notion assez vague de saut.

À tout moment le processeur détient l'adresse de la prochaine instruction à exécuter dans le *compteur de programme* (PC, pour *Program Counter*, ou IP pour *Instruction Pointer* ; pour les processeurs Intel l'appellation est « eip » ; on trouve aussi compteur ordinal). Lors du démarrage de l'ordinateur, le PC est chargé avec l'adresse de la première instruction à exécuter, et lors de l'exécution de chaque instruction ultérieure le processeur place dans le PC l'adresse de l'instruction consécutive. Un saut, ou branchement, consiste à placer dans le PC l'adresse de l'emplacement dans le texte du programme où est située l'instruction désirée.

Pour anticiper un peu, lorsque le système d'exploitation retire le contrôle du processeur à un programme en cours d'exécution, que nous appellerons un *processus*, pour le donner à un autre, le PC du nouveau processus est chargé à la place de celui du prédécesseur, et ce PC nouvellement chargé contient l'adresse à laquelle le processus doit redémarrer (s'il avait été préalablement interrompu) ou l'adresse de démarrage du programme correspondant, s'il s'agit d'un processus « vierge ».

Le PC est dans la plupart des processeurs un des éléments d'un groupe d'informations appelé « mot d'état de programme » ou *Program Status Word* (PSW), que nous retrouverons plus loin.

7. … à l'exception notable des processeurs Intel x86.

2.6 Premier métalangage

2.6.1 Vers un langage symbolique

Au tout début des ordinateurs les programmes s'écrivaient littéralement comme celui de la section 2.4.1, avec des 0 et des 1. C'était évidemment insupportable. Aussi a-t-on rapidement inventé une notation mnémonique mieux adaptée à l'usage par les êtres humains. Cette notation consiste en ceci :

- donner un nom symbolique à chaque code opération (dans notre exemple, ceux du second tableau de la section 2.4.1) ;
- représenter les nombres en notation décimale ;
- séparer les opérandes par des virgules ;
- supprimer la mention des opérandes sans objet ;
- placer (si besoin est) devant chaque instruction une étiquette symbolique qui permet de la désigner.

Notre programme devient maintenant :

étiquette	code op.	opérandes
	LOAD	1, 20
	BZ	1, FIN
	ADD	1, 21
	STORE	1, 22
	PRINT	22
FIN	END	

Les transformations qu'il faudra appliquer à ce programme symbolique pour produire le texte du programme binaire, seul « compréhensible » par les circuits logiques de l'ordinateur, sont simples et surtout très mécaniques. La plus complexe concerne le traitement des étiquettes ; ainsi pour transformer la seconde instruction `BZ 1, FIN` en :

```
0010 1100 0000 0101
```

il faudra calculer l'adresse qui correspond à `FIN` dans la sixième instruction, ce qui revient à compter combien il y a de mots entre la première et la sixième instruction. On dit que `FIN` est un nom symbolique qui désigne l'instruction d'adresse 5. Cet usage de noms symboliques est important et nous y reviendrons, notamment à la section 4.3.2 p. 76.

De même que le symbole `FIN` correspond à une adresse, nous pouvons imaginer de désigner par des symboles les adresses des cases mémoire qui contiennent les données et le résultat du calcul. Notre programme deviendra alors :

étiquette	code op.	opérandes
	LOAD	1, VAL1
	BZ	1, FIN
	ADD	1, VAL2
	STORE	1, RESULT
	PRINT	RESULT
FIN	END	
	...	*(d'autres instructions éventuelles)*
VAL1	18	
VAL2	42	
RESULT	0	

Comment les symboles vont-ils être transformés en adresses? La réponse quelques lignes plus bas.

2.6.2 Adresses absolues, adresses relatives

Nous avons dit que les symboles `FIN`, `VAL1`, `VAL2`, `RESULT` désignaient des cases dans la mémoire, ou pour le dire autrement représentaient des adresses. Il s'agit jusqu'ici de désigner chaque case par son numéro, c'est-à-dire son adresse absolue : il pourrait être plus commode que ce soit une adresse relative, par rapport au début du texte du programme. Ceci permettrait de s'affranchir de l'hypothèse selon laquelle le programme est forcément chargé à partir de l'adresse `0`.

Supposons que le programme soit chargé à une adresse quelconque, par exemple `256` ; nous allons supposer aussi que nous disposions d'un registre pour conserver cette information ; ce registre sera appelé le *registre de base* pour le programme considéré. Chaque adresse manipulée par le programme sera en fait calculée comme la somme de l'adresse relative qui figure dans le texte du programme (sous forme numérique ou comme un symbole) et du contenu du registre de base. Ainsi le fonctionnement de notre programme sera indépendant de son emplacement en mémoire. Nous reviendrons sur ce mécanisme à la section 4.3.2.

Le programme que nous avons en remplaçant les valeurs numériques des adresses par des noms symboliques, les codes opération binaires par des noms mnémoniques et les valeurs binaires par des nombres décimaux s'appelle un programme symbolique. Ces transformations simples transforment radicalement l'acte de programmer : au lieu d'une suite illisible de 0 et de 1 le programmeur dispose d'un langage symbolique, certes rudimentaire mais infiniment plus expressif.

Le texte du programme que nous avons sous les yeux maintenant est assez différent de celui en langage machine binaire : s'il est assez évident qu'il est

plus facile à écrire pour un être humain, il va nous falloir disposer d'un outil de traduction du programme symbolique en langage machine.

2.6.3 Assembleur, table des symboles

Le programme qui réalise cette traduction du langage symbolique en texte binaire directement exécutable par la machine s'appelle un assembleur. Par extension l'habitude s'est établie de nommer *assembleur* un tel langage symbolique. Notre exemple est bien sûr extrêmement simplifié par rapport aux assembleurs réels, mais cet assembleur-jouet est un modèle raisonnablement réaliste et suffira au propos de ce livre, qui n'est pas de vous apprendre à programmer en assembleur.

Il reste un mystère à élucider : comment l'assembleur procède-t-il pour traduire les noms symboliques en adresses ? L'assembleur va construire un élément important du programme, la *table des symboles*, ce qui se fait en deux temps : dans une première passe l'assembleur dresse la liste de tous les symboles qu'il rencontre, dans une seconde passe il calcule la distance (en nombre de cases mémoire) entre le début du texte du programme et l'emplacement où le symbole est défini ; cette distance est l'adresse relative désignée par ce symbole par rapport au début su programme, et sa valeur est placée dans la table.

L'assembleur représente par rapport au langage machine un métalangage, plus loin de la réalité concrète de l'ordinateur, c'est-à-dire plus abstrait et de ce fait plus simple pour l'être humain. En fait, ceci est la première marche de l'escalier de métalangages de plus en plus abstraits que gravit l'usage contemporain des ordinateurs pour être plus accessible aux personnes. Simplement, pour qu'un langage, aussi abstrait soit-il, puisse prétendre à la programmation des ordinateurs, il faut prouver que tout programme bien formé, c'est-à-dire conforme à sa syntaxe et à sa sémantique, peut se traduire en langage machine, de façon univoque, et par un algorithme.

2.6.4 Traduction de langages

J'ai mentionné plus haut (p. 25) le résultat théorique très important de Böhm et Jacopini [21] qui donne du même coup la voie d'une telle preuve et son équivalence avec le modèle de la machine de Turing. Un langage qui satisfait à toutes ces conditions est dit *Turing-équivalent*. C'est le cas des langages de programmation généraux, tels que C, Java, Lisp ou Fortran, mais il y a des langages non-Turing-équivalents et néanmoins très utiles, comme le langage SQL d'accès aux bases de données, HTML et XML pour décrire des documents, etc.

Un programme qui traduit un langage de programmation dans un autre, généralement de plus bas niveau, s'appelle un compilateur. Un métalangage de l'assembleur est appelé un langage évolué. Fortran fut en 1954 le premier langage évolué. Un langage évolué est en principe indépendant de l'ordinateur uti-

lisé : il suffit pour exécuter un programme écrit en langage évolué sur un ordinateur quelconque de disposer d'un compilateur qui le traduise vers l'assembleur propre à cet ordinateur.

2.7 Comment cela démarre-t-il ?

Pour expliquer le fonctionnement du programme ci-dessus nous avons supposé qu'il était chargé en mémoire à l'adresse 0. Mais comment s'est-il retrouvé là, sachant que la mémoire des ordinateurs est réalisée, en ce début de vingt-et-unième siècle, par des dispositifs techniques tels que l'état en est effacé dès la coupure de l'alimentation électrique ? Le principe est le suivant :

- Pendant son exécution, le programme, sous la forme binaire que nous venons de décrire, est enregistré dans la mémoire centrale.
- Quand l'ordinateur est éteint (ou quand un autre programme est en cours d'exécution), ce même programme est stocké, sous cette même forme binaire, sur un disque dur, ou une disquette, ou un CD-ROM... bref sur[8] une mémoire permanente externe (par opposition à mémoire centrale) accessible par l'Unité d'entrée-sortie de l'architecture de von Neumann. Un programme stocké sur disque y réside sous forme d'un fichier ; nous n'avons pas encore étudié les fichiers, mais disons pour l'instant qu'un fichier est un ensemble de données identifié et délimité sur un support externe.
- Une mémoire permanente externe a les mêmes propriétés que la mémoire centrale quant à la capacité de contenir l'information, mais elle s'en distingue par la capacité à la conserver après la coupure du courant. Il ya aussi une différence considérable de temps d'accès. Pour que l'information contenue dans une mémoire externe soit traitée par l'Unité centrale, il faut au préalable la recopier dans la mémoire centrale. C'est le rôle de l'Unité d'entrée-sortie.
- Tout ordinateur moderne possède sur sa carte-mère un circuit spécial de démarrage dont le rôle est de lancer un petit programme contenu dans un élément de mémoire non-volatile physiquement distinct des autres composants de la carte mère de l'ordinateur ; depuis les années 1990 c'est en général de la mémoire Flash, analogue à celle des clés USB et des disques

8. Il y aurait à dire sur l'usage de la préposition *sur* quand il s'agit d'une mémoire externe par opposition à *dans* pour la mémoire centrale. Les mémoires externes sont souvent (étaient ? pensons aux SSD et aux clés USB) des dispositifs où effectivement l'information est enregistrée à la surface d'un médium (disque magnétique, DVD) cependant que jadis la mémoire centrale était réalisée par de belles structures tridimensionnelles de tores de ferrite traversés de fils conducteurs, dont le souvenir a survécu à l'avènement des circuits électroniques multicouches mais plats.

SSD, ce qui permet de le modifier. Ce programme spécial, le BIOS (décrit, ainsi que sa version moderne UEFI *(Unified Extensible Firmware Interface)*, au chapitre 12 p. 339), est, par construction, activé à la mise sous-tension. Son action consiste à aller chercher sur une mémoire externe préparée à cet effet un autre programme un peu moins petit, à le recopier en mémoire centrale et à en déclencher l'exécution. Ce processus est connu sous le nom de *boot-strap* ou simplement *boot*, mais il est permis de dire amorçage. Le programme que le circuit de démarrage va chercher sur la mémoire externe s'appelle programme de *boot*. Une mémoire externe sur laquelle on aura enregistré un programme de *boot* convenable s'appelle un disque *bootable*, une clé USB de *boot*, etc. Ce processus d'amorçage sera examiné plus en détail au chapitre 12 p. 339.

– Que va faire le programme de *boot* ? Charger en mémoire centrale le programme qui lui aura été désigné lors de sa conception ; dans l'exemple de la section précédente ce sera notre petit programme–jouet, mais dans le monde réel ce sera, vous l'avez deviné, le système d'exploitation, qui ensuite lancera lui-même d'autres programmes, au fur et à mesure des demandes qui lui seront adressées. Plus précisément, la partie du système qui est lancée après l'amorçage, et qui en est l'âme, est appelée *noyau (kernel)* du système.

2.8 Quel est le rôle de la mémoire ?

Nous avons introduit la notion de mémoire en disant qu'une action du processeur consistait à consulter ou à modifier l'état de la mémoire. Cette définition de l'action est très importante, elle trace la ligne de séparation entre la conception mathématique traditionnelle du calcul et la conception informatique liée à la notion de procédure effective. La mathématique ignore cette notion d'état, qui introduirait dans son univers d'abstraction un aspect physique totalement incongru.

L'informatique, et plus précisément la programmation des ordinateurs, confère au calcul une dimension concrète, effective. C'est un peu comme si le papier sur lequel le mathématicien inscrit les signes du calcul avec un crayon acquérait un statut théorique. Ce passage permanent du concret à l'abstrait est d'ailleurs l'agrément suprêmement fascinant de la programmation : dire c'est faire. J'énonce la formule d'une action, et la machine l'exécute.

La mémoire possède donc un statut théorique important : ce qui matérialise le calcul, ce sont les états successifs de la mémoire, et le moteur qui anime l'ordinateur et produit cette succession d'états est l'aptitude de l'unité centrale à affecter la mémoire, à modifier son état, à effectuer ce que l'on appelle une affectation.

Parlons de l'affectation. Un mot de mémoire peut donc être utilisé pour emmagasiner un état du calcul. On dira que c'est une *variable*, au sens de la programmation, qui est différent de l'acception mathématique usuelle. Une variable au sens de la programmation est un objet doté des qualités suivantes :

- un nom ; en langage machine, le nom d'une variable est son adresse, dans les langages évolués c'est un symbole plus commode mais équivalent en dernière analyse à une adresse ;
- une valeur ; la valeur est celle du nombre binaire contenu dans le mot, mais il est possible de le considérer comme un code auquel on confère une sémantique particulière, par exemple un caractère alphabétique, ce qui ouvre la voie au traitement de texte ;
- il est possible de « prendre » cette valeur, par exemple en langage machine pour la recopier dans un registre ou dans un autre mot de la mémoire ;
- il est possible de modifier cette valeur, par exemple en copiant dans le mot le contenu d'un registre ou d'un autre mot de la mémoire : c'est l'affectation[9].

Le modèle théorique qui rend compte de ce que nous venons de dire de la mémoire est la machine de Turing.

2.9 La machine de Turing[10]

Le but de Turing lorsqu'il a imaginé sa machine (toute abstraite et théorique, il va sans dire) était de donner une chair à la notion abstraite de procédure effective. Une procédure effective, c'est un procédé pour effectuer un calcul, par exemple la démarche à suivre pour faire une multiplication, telle qu'elle vous a été enseignée à l'école.

Le modèle formel d'une procédure effective (pour décrire un algorithme) doit posséder certaines propriétés. Premièrement, chaque procédure doit recevoir une définition finie. Deuxièmement, la procédure doit être composée d'étapes distinctes, dont chacune doit pouvoir être accomplie mécaniquement. Dans sa simplicité, la machine de Turing déterministe composée des éléments suivants répond à ce programme :

- une mémoire infinie représentée par un ruban divisé en cases. Chaque case du ruban peut recevoir un symbole de l'alphabet défini pour la machine ;

9. Nous raffinerons cette définition de la variable à la section 4.2.3 p. 73.

10. Les cinq alinéas qui suivent sont empruntés à mon livre *Initiation à la programmation et aux algorithmes avec Scheme*, publié en 2020 par les Éditions Technip, avec l'aimable autorisation de l'éditeur.

- une tête de lecture capable de parcourir le ruban dans les deux sens ;
- un ensemble fini d'états parmi lesquels on distingue un état initial et les autres états, dits accepteurs ;
- une fonction de transition qui, pour chaque état de la machine et chaque symbole figurant sous la tête de lecture, précise :
  - l'état suivant ;
  - le caractère qui sera écrit sur le ruban à la place de celui qui se trouvait sous la tête de lecture ;
  - le sens du prochain déplacement de la tête de lecture.

La configuration d'une machine de Turing peut être représentée par un triplet (q,m,u) où q est l'état de la machine, m le mot qui apparaît sur le ruban avant la position de la tête de lecture, u le mot figurant sur le ruban entre la position de la tête de lecture et le dernier caractère non blanc.

Un arc du graphe de la fonction de transition peut être représenté par un quintuplet (qi,si,sj,x,qj) où :

- q_i est l'état de départ ;
- s_i est le symbole pointé avant l'exécution du cycle ;
- s_j est le symbole qui doit remplacer s_i ;
- x est un élément de $\{G,D,w\}$ (G pour gauche, D pour droite, w pour un déplacement nul) ;
- q_j est l'état de fin de cycle.

Pour se donner une intuition de la chose, imaginons une M.T. (machine de Turing) avec un alphabet $\{0,1,$<espace>$\}$; nous conviendrons d'utiliser le système de numération unaire (celui que vous utilisez pour marquer les points au ping-pong, autrement dit « les bâtons ») et de séparer les nombres par des 0. Cette machine est représentée par la figure 2.3 p. 37. Pouvons-nous grâce à elle additionner deux nombres ?

Notre M.T. fonctionnera selon un cycle qui consiste à passer successivement par les trois phases suivantes, puis à recommencer :

Phase de lecture - La machine lit le contenu de la case courante et le transmet comme paramètre d'entrée à la fonction de transition.

Phase de calcul - La valeur de la fonction de transition est calculée en fonction de l'état courant et de la valeur contenue dans la case courante.

Phase d'action - L'action déterminée par la fonction de transition est effectuée ; elle comporte (éventuellement) une modification de la valeur contenue dans la case courante et un déplacement de la tête de lecture ; cette phase d'action est donc capable de *produire* des symboles.

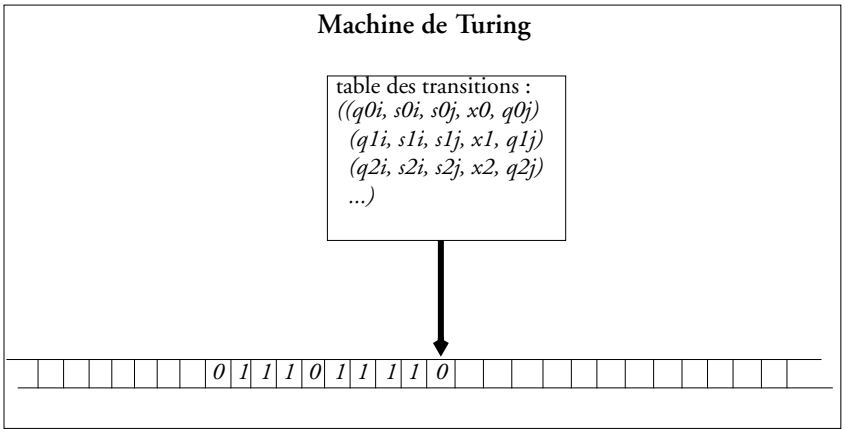


Figure 2.3 : *Un modèle théorique*

Le ruban mentionne successivement les nombres 4 et 3. Pour les additionner il suffit que la tête de lecture lise successivement les quatre chiffres unaires qui constituent le nombre 4, dont la fin sera indiquée par l'occurrence du signe zéro. Il faudra alors supprimer le zéro et récrire d'une case à droite les chiffres du nombre 3, jusqu'à la rencontre d'un signe zéro, qui subira le même traitement, pour obtenir 7. L'écriture de la table des transitions constituera pour le lecteur un exercice amusant (dont la solution est donnée ci-dessous).

État avant	symbole lu	symbole écrit	sens du déplacement	État après	Commentaires
q_0	0	0	G	q_0	Parcours vide
q_0	1	1	G	q_1	Début de m
q_1	1	1	G	q_1	Parcours de m
q_1	0	1	G	q_2	Fin de m, remplacement de 0 par 1
q_2	1	1	G	q_2	Parcours de n
q_2	0	0	D	q_3	Fin de n, recul
q_3	1	0	w	q_4	Remplacement du dernier 1 de n par 0, fin.

On peut doter sa Machine de Turing de l'alphabet fini de son choix. Son ruban peut être infini dans les deux sens ou dans un seul. Elle peut même avoir plusieurs rubans. On montre (Turing a montré) que ces diverses machines sont équivalentes. Tous les langages de programmation modernes sont équivalents à la Machine de Turing.

Chapitre 3 Du système d'exploitation au processus

Sommaire

3.1 Premiers essais

À l'époque des premiers ordinateurs, à la fin des années 1940, il n'y avait rien qui annonçât les systèmes d'exploitation. Maurice Vincent Wilkes a raconté le lancement et l'exécution du premier programme sur l'EDSAC: son texte était sous une forme tout à fait similaire au format binaire donné en exemple à la section 2.4.1. Chaque chiffre binaire était introduit manuellement à l'aide de commutateurs. Puis le programme se déroulait durant des dizaines d'heures, pour finalement afficher le résultat attendu (une table de nombres premiers en l'occurrence). On a vu comment l'invention des langages symboliques, puis d'autres niveaux de métalangages, allait simplifier les opérations de rédaction proprement dite du programme. Mais il fallait aussi faire des progrès sur la façon d'introduire programmes et données dans la machine et d'en extraire les résultats.

L'introduction manuelle des données était une perte de temps: un ingénieur qui travaillait dans l'équipe de Gene Amdahl (un des plus fameux architectes d'ordinateurs), Nathaniel Rochester, imagina au début des années 1950 de les enregistrer sur bande magnétique au préalable. Le premier ordinateur qui les utilisa, l'IBM 701, fut livré en 1953 au *Department of Defense* américain (DoD dans la suite de ce livre), ce qui illustre la tradition continue de financement des progrès de l'informatique sous l'impulsion des commandes militaires, ici celles de la guerre de Corée.

Cette utilisation des bandes magnétiques connut un nouveau développement avec l'IBM 704, livré fin 1955. Sur cette machine conçue par Gene Amdahl, l'ingénieur de General Motors Bob Patrick écrivit un programme (nommé GM-NAA I/O, pour *General Motors and North American Aviation Input/Output system*) qui enchaînait automatiquement entrée des données, calcul, impression des résultats, entrée des données, etc. L'IBM 704 fut d'ailleurs le support d'un nombre considérable d'innovations capitales [1].

1. Le parti-pris de cet ouvrage est de ne pas entrer dans le détail de l'histoire des ordinateurs, à laquelle ont déjà été consacrés des livres excellents dont le lecteur trouvera les références dans la bibliographie de celui-ci. Je me contenterai de donner la liste de ces innovations sans les décrire: seconde utilisation (derrière l'UNIVAC 1103A) de la mémoire à tores de ferrite inventée par An Wang en 1950, arithmétique à virgule flottante, premier langage de programmation évolué (Fortran), premier langage de programmation fonctionnelle (Lisp).

Assez vite une constatation se fit jour : l'impression des résultats, à la cadence d'un télétype capable d'imprimer dix caractères par seconde, voire même leur écriture sur bande magnétique, pouvait prendre un temps aussi long que le calcul proprement dit, ce qui immobilisait la coûteuse unité centrale pour une tâche somme toute subalterne. Le mot « coûteuse » doit être replacé dans le contexte de l'époque : le prix du 704, avec sa mémoire de 4096 mots de 36 bits qui semblait énorme à l'époque (elle fut ensuite étendue à 32 768 mots, à comparer au milliard de mots du plus petit ordinateur en vente aujourd'hui au supermarché le plus proche de votre domicile !), se chiffrait en millions de dollars de l'époque, et il n'y eut qu'une vingtaine d'exemplaires construits pour des clients fortunés tels que les militaires ou de grands centres de recherche comme le MIT *(Massachusetts Institute of Technology)*.

Il aurait été possible de réduire la perte de temps due à l'impression des résultats en les écrivant provisoirement sur une mémoire auxiliaire électromagnétique (disque, bande, tambour…) beaucoup plus rapide qu'une imprimante, puis en les imprimant plus tard, pendant que l'unité centrale effectuerait d'autres calculs. Cela semblait possible parce que la tâche d'impression, ralentie par les opérations mécaniques de l'imprimante, n'utilisait les circuits de l'unité arithmétique et logique que fort peu, voire pas du tout si l'on avait pris soin d'enregistrer avec les résultats proprement dits des codes de commande destinés à l'électronique (rudimentaire) de l'imprimante pour indiquer les sauts de ligne, de page, etc.

La réalisation de cet objectif apparemment modeste nécessitait encore un peu de programmation : à la fin du programme de calcul il fallait qu'un programme déclenchât d'une part le démarrage du programme d'impression (destiné à vivre sa vie indépendamment), d'autre part le démarrage du programme de calcul suivant. Avant de lancer le programme d'impression il fallait aussi vérifier que l'impression des résultats précédents était bien terminée. Pendant qu'on y était, on procéderait à une optimisation analogue en autorisant le recouvrement entre le temps de calcul et le temps de lecture des données nécessaires au calcul suivant.

Il apparut vite assez logique de confier cette mission, organiser le recouvrement dans le temps de plusieurs activités, à un « méta-programme », nommé moniteur, chargé de déclencher à l'instant convenable l'exécution des programmes d'application, qui pourraient être considérés comme ses sous-programmes. Nous étions en 1955 et l'ancêtre des systèmes d'exploitation était né.

Nous allons décrire les traits marquants de l'évolution des systèmes à partir de cet ancêtre rudimentaire, mais auparavant il convient de préciser un peu notre vision de ce qu'est un programme : le chapitre 2 nous en a donné une vision statique, centrée sur le texte qui décrit l'algorithme, les sections qui suivent insistent sur l'aspect dynamique, ce qui se passe pendant l'exécution du programme.

3.2 Simultanéité et multiprogrammation

Nous avons annoncé au début du chapitre 2 que nous allions compléter notre définition du système d'exploitation par la capacité qu'il pourrait conférer à l'ordinateur par lui piloté de faire plusieurs choses à la fois. Or nous avons aussi mentionné comme une caractéristique essentielle de l'architecture de von Neumann que les ordinateurs exécutaient une et une seule instruction à la fois. S'agit-il là d'une contradiction ?

En fait, c'est le principe de von Neumann qui est exact, et le système d'exploitation ne procurera que l'illusion de la simultanéité, une *pseudo-simultanéité*[2], Devons-nous néanmoins acheter des ordinateurs aussi fallacieux ? Oui. En effet, le temps d'exécution d'une instruction câblée du processeur est très court, de l'ordre de la nano-seconde, ce qui procure plusieurs centaines de millions d'instructions par seconde, et ainsi une tranche de temps de quelques fractions de seconde, partagée entre plusieurs processus, donne à l'échelle macroscopique l'illusion de la simultanéité.

3.2.1 Chronologie d'une entrée-sortie

Voyons ce qui se passe lorsque l'utilisateur d'un ordinateur demande une opération macroscopique au logiciel qu'il utilise, par exemple « déplacer le curseur du traitement de texte à la fin de la ligne » :

- Cette opération va peut-être demander l'exécution de plusieurs milliers d'instructions élémentaires, disons dix mille en étant large.
- Le temps mis par l'utilisateur pour commander le déplacement du curseur vers la fin de la ligne sera, disons, d'un quart de seconde. Pour simplifier le modèle nous supposerons que l'opération est commandée par un raccourci au clavier qui « consomme » vingt mille instructions. (L'usage de la souris engendre des événements dont la détection et le traitement par le système sont complexes.)
- Pour qu'aucun temps d'attente ne soit perceptible à l'utilisateur, ce qui produirait de l'inconfort, il faut que l'action, une fois commandée, soit effectuée dans un délai de deux centièmes de seconde (c'est ici encore très généreux).
- Le budget des délais pour notre opération est donc le suivant : nous disposons tout d'abord de 0,25 seconde pour exécuter 20 000 instructions,

2. Ceci est vrai pour les ordinateurs qui ont un seul processeur. Il y a des ordinateurs à plusieurs processeurs capables d'exécuter plusieurs programmes en vraie simultanéité. En première approximation nous considérerons un ordinateur à n processeurs comme n ordinateurs indépendants. Un tour d'horizon des extensions et des dépassements de l'architecture de von Neumann figure au chapitre 9.

puis de 0,02 seconde pour exécuter 10 000 instructions. Or si notre ordinateur peut exécuter un milliard d'instructions par seconde, valeur banale en cette année 2018, nous voyons qu'il nous reste énormément de marge, puisqu'en 0,25 seconde il peut exécuter 250 millions d'instructions élémentaires, et en 0,02 seconde, 20 millions.

– Le système d'exploitation va donc permettre l'exécution « pseudo-simultanée » de plusieurs programmes tels que celui que nous avons décrit :

  – Nous avons dit qu'il fallait à l'utilisateur 0,25 seconde pour effectuer la commande par un raccourci au clavier : c'est plutôt vers la fin de ce délai que, les opérations manuelles terminées, le logiciel de traitement de texte (et en fait d'autres programmes liés notamment au gestionnaire de fenêtres, mais négligeons ces aspects pour l'instant) vont exécuter les 20 000 instructions nécessaires pour capter cette commande. Ces 20 000 instructions vont s'exécuter en 0,000 020 seconde, ce qui laisse pendant les opérations manuelles (ou plus exactement digitales) un temps inoccupé de 0,249 980 seconde, disponible pour l'exécution de 249 980 000 instructions appartenant à d'autres programmes.
  – De la même façon, nous voyons que les 10 000 instructions nécessaires pour envoyer le curseur en bout de ligne vont laisser pendant le délai de 0,02 seconde que nous nous sommes imposé pour le confort de notre utilisateur bien-aimé un temps libre suffisant pour exécuter 19 990 000 instructions au profit d'autres programmes, dits « programmes *concomitants* » (en anglais *concurrent*, ou si l'on veut concurrents pour l'accès au temps de processeur).

3.3 Notion de processus

De la section précédente nous pouvons déduire la notion de processus : le rôle du système d'exploitation sera de distribuer harmonieusement le temps disponible (de façon pléthorique semble-t-il d'après l'exemple ci-dessus) entre différents programmes en train de s'exécuter « pseudo-simultanément ». Lorsque l'on considère des programmes sous l'angle de leur concurrence pour l'accès au temps du processeur, nous les appellerons des *processus*. L'arbitrage de la répartition du temps entre les processus est la fonction fondamentale du système d'exploitation, c'est une fonction, bien sûr, de « bas niveau », qui relève des « couches basses ».

La capacité pour le système d'exploitation d'organiser le partage des ressources entre plusieurs processus concomitants qui s'exécutent en pseudo-simultanéité s'appelle la *multiprogrammation*.

Nous pouvons emprunter à Andrew Tanenbaum [129] la métaphore du pâtissier qui prépare deux gâteaux : le programme, c'est la recette du gâteau, c'est la même pour les deux gâteaux, elle décrit des actions qui, dans le livre de cuisine, sont à l'état abstrait. Le processus, c'est la suite d'actions effectives qui va mener à la réalisation d'un gâteau concret. Pour aboutir, le processus « gâteau numéro 1 » doit se voir attribuer un certain nombre de ressources : farine, œufs, sucre, un certain laps du temps du pâtissier, une certaine période de disponibilité du rouleau à pâtisserie, une certaine durée de cuison dans le four. Certaines contraintes doivent être respectées : le rouleau et la planche à pâtisserie doivent être affectés au processus « gâteau numéro 1 » en même temps, et avant l'affectation du four. Nous supposerons que le four ne peut contenir qu'un seul gâteau à la fois, de même que le processeur d'un ordinateur ne peut exécuter qu'une instruction à la fois. Les œufs, le sucre et la farine de gâteau 1 sont bien entendus distincts de ceux de gâteau 2.

Pour préparer ses gâteaux, le pâtissier a le choix entre deux méthodes : préparer d'abord l'un, jusqu'à la fin du processus, puis l'autre, ou bien mener de front la confection des deux gâteaux, en se consacrant alternativement à l'un, puis à l'autre, ce qui permettra par exemple de rouler la pâte de gâteau 1 pendant que celle de gâteau 2 repose. La seconde méthode permettra sans doute de servir plus rapidement le client du second gâteau, sans trop retarder la livraison du premier, mais il y faudra plus d'organisation et de coordination. Ainsi, lorsque le pâtissier passera du gâteau 1 au gâteau 2, il faudra qu'il note (ne serait-ce que dans sa mémoire) certaines informations sur l'état du processus gâteau 1 : a-t-il déjà mis du sucre, ou pas ? Ce qui revient à cocher au crayon à quel endroit du texte de la recette il s'est interrompu dans le processus gâteau 1 pour passer au processus gâteau 2.

En procédant ainsi, le pâtissier réalise deux gâteaux en « pseudo- simultanéité », ce qui permettra à ses deux clients d'être servis à temps pour le dessert.

Nous avons déjà vu à la page 29 un moyen qu'ont les ordinateurs de noter où ils en sont dans un processus : le compteur ordinal, ou compteur de programme, nommé en abrégé PC *(program counter)*[3], qui indique à chaque pas du programme l'adresse de l'instruction suivante à exécuter, et qui souvent fait partie du mot d'état de programme. Eh bien le PC sert aussi à cela : pour savoir où on en est dans le processus gâteau 1 au moment où on va l'abandonner pour s'occuper de gâteau 2, on note quelque-part la valeur du PC. Nous entrerons plus en détail dans la description de ces mécanismes, plus particulièrement aux sections 3.7 et 3.11, ce qui précisera notamment la position de ce *quelque-part* où est notée la valeur du PC.

3. Pour les processeurs Intel l'appellation est « eip » comme *extended instruction pointer*.

Nous poursuivrons l'étude de cette notion centrale qu'est le processus à la section 3.7 p. 48 ci-dessous.

3.4 Réification du calcul

Nous avons vu à la section 2.3 que les deux principes à la clé de l'architecture de von Neumann étaient l'exécution séquentielle et le partage d'une mémoire unique pour les instructions et les données du calcul. Cette réunion du programme et des données permet de considérer le programme comme de l'information, au même titre que les données, et cela a des conséquences énormes en termes de raisonnement sur les programmes, de transformations appliquées à des programmes par d'autres programmes, de traduction de programmes, pour tout dire cela permet le concept de programme.

Mais avant que la pensée en son mouvement réunisse données et programme elle avait dû apprendre à les distinguer. En effet pour le mathématicien traditionnel la question de cette distinction n'existe pas, c'est un problème qui n'a pas lieu.

L'idée de réification du processus de calcul apparaît avec Babbage, dont la machine analytique devait comporter une unité de contrôle constituée de cylindres à picots, une unité de calcul (le « moulin »), une mémoire centrale (le « magasin »), un système d'entrées-sorties de données sur carton perforé emprunté aux orgues de barbarie, et enfin un dispositif de circulation de données par tringles à crémaillère. Incidemment, nous voyons ici une autre idée fondamentale de Babbage, la distinction entre unité de contrôle et unité de calcul, la première supervisant l'exécution des opérations de la seconde, ce qui permet d'assurer un de nos postulats de départ : en fonction du résultat d'une opération de calcul, l'unité de contrôle pourra rompre la séquence d'exécution des instructions pour commander un branchement vers une instruction située plus loin dans le texte du programme, ou au contraire un retour en arrière pour répéter une séquence déjà effectuée.

Par données enregistrées sur carton perforé nous entendons aussi les programmes, et Lady Ada Lovelace, fille du poète Byron, mécène de Babbage et d'autres hommes de science anglais tels que Faraday et figure intellectuelle importante de son époque, a rédigé les premiers textes de programmes de l'histoire. C'est en son honneur que le langage de programmation Ada a été ainsi nommé [4].

4. La genèse de ces programmes met en scène un autre personnage fameux, l'ingénieur et mathématicien italien Luigi Menabrea, futur premier ministre de son pays, qui publia en 1842 (en français ; il était natif de Chambéry, mais c'est sans doute le rôle prééminent du français dans le monde mathématique qui a déterminé le choix de cette langue ; incidemment, Leibniz écrivait aussi en français) le premier article sur les travaux de Babbage. Babbage avait demandé à Ada de les traduire en anglais ; c'est au cours de ce travail de traduction et d'additions qu'Ada com-

Les logiciens de la première moitié du XX^{e} siècle abordent le problème de la réification de façon plus abstraite, sur les traces de Leibniz, par les systèmes formels. Kurt Gödel et à sa suite Alan Turing avaient dû inventer des notations pour désigner des procédures effectives, les transformer, leur faire subir des traitements. Alonzo Church réunit ces idées en un formalisme qui aujourd'hui encore satisfait théoriciens et praticiens de la programmation, le λ-calcul. En 1956 John MacCarthy élabore à partir du λ-calcul un langage de programmation, LISP, pour lequel il implémente à partir de 1958 un traducteur sur l'IBM 704.

Le λ-calcul se distingue des autres notations mathématiques en ceci que les fonctions y sont des objets comme les autres, susceptibles d'être traités comme des variables d'autres fonctions ou comme des termes d'expressions, des λ-termes dans des λ-expressions. Pour cette raison LISP est appelé un langage fonctionnel, ou encore un langage applicatif, puisqu'aussi bien le propre d'une fonction est de pouvoir être appliquée à des arguments.

3.5 Notion de sous-programme

À ce stade de l'exposé il convient d'exposer une notion d'une importance théorique et pratique cruciale, la notion de sous-programme, par quoi il est possible de diviser la difficulté de rédaction d'un programme en le découpant en plusieurs programmes plus simples.

Un programme significatif représente un texte d'une longueur respectable (par exemple 10 000 lignes), et il faut organiser ce volume d'information pour que les humains qui l'écrivent et le modifient puissent s'y retrouver. Un moyen très général et simple est d'avoir un programme principal qui joue en quelque sorte le rôle de table des matières et qui transfère le contrôle à des sous-programmes chargés de telle ou telle fonction particulière, eux-mêmes découpés en sous-programmes plus petits, etc. Ce transfert de contrôle est nommé « appel de sous-programme » (ou de fonction, ou de procédure, voire de méthode, ces mots désignent des objets similaires dans le contexte envisagé ici). Il faut garder à l'esprit qu'il s'agit essentiellement d'organiser l'information constituée par le texte du programme à l'usage du lecteur humain ; une fois traduit en langage machine il restera une suite monolithique de 0 et de 1.

Quand un programme appelle un sous-programme[5] il doit lui transmettre des informations : supposons que je dispose d'un sous-programme pour le calcul

mença à écrire des programmes destinés à résoudre différents problèmes d'analyse mathématique. Lorsque le langage machine de Babbage se révélait trop peu maniable pour un certain problème, elle en demandait la modification. Une abondante littérature est maintenant disponible sur ce sujet passionnant, y compris en édition française.

5. Incidemment tout programme est un sous-programme, le « programme principal » est appelé par le système d'exploitation auquel il rend la main en se terminant.

du cosinus d'un angle ; je vais l'utiliser chaque fois que dans mon travail j'aurai un angle dont j'ai besoin de connaître le cosinus ; il faudra que je transfère au sous-programme la valeur de l'angle, c'est l'*argument* ou le *paramètre* de mon sous-programme ; il faut aussi que le sous-programme connaisse deux autres choses, l'adresse à laquelle transférer le contrôle quand il aura fini, dite adresse de retour, et l'adresse où déposer le résultat afin que le programme appelant puisse en prendre connaissance.

Un sous-programme peut être écrit par la même personne que le programme principal, et ils seront assemblés en un tout par un programme spécial, l'éditeur de liens. Un progamme écrit et compilé par nous utilise en fait d'autres programmes fournis par le système d'exploitation, par exemple pour communiquer avec le système, ou par le compilateur. Ces programmes sont dans des bibliothèques, qui sont des fichiers qui contiennent des programmes déjà tout prêts à l'usage. Au moment de l'édition de liens, soit ils sont inclus dans le fichier exécutable (édition de liens statique), soit l'éditeur de liens place dans le fichier exécutable une référence vers leur emplacement dans une bibliothèque partageable (édition de liens dynamique) et c'est au moment du chargement que les références qui permettent la liaison entre ces différents éléments de programmes seront établies par un programme *ad hoc* nommé chargeur ou, par exemple sous Linux, interpréteur de programmes.

Observons qu'isoler une fonction particulière dans un sous-programme qui sera désigné par un nom particulier (par exemple `sinus` pour le programme de calcul du sinus d'un angle) revient à créer une méta-instruction qui enrichit notre langage de programmation. Les langages issus du λ-calcul tels que LISP et Scheme se prêtent particulièrement bien à ce processus d'enrichissement par abstraction de fonctions.

3.6 Points de vue sur les programmes

Nous commençons à avoir une idée de ce qu'est un programme : arrêtons-nous sur les différentes façons de l'envisager :

- Comme la description d'un algorithme sous une forme exécutable en machine : c'est le point de vue du programmeur, que nous avons principalement envisagé jusqu'à maintenant.
- Comme de l'information sous forme de données en mémoire : c'est le point de vue *métalinguistique* de l'auteur de compilateur, qui doit traduire le texte du programme vers un langage de plus bas niveau et, *ultima ratio*, en langage machine.
- Comme un *processus* en cours d'exécution et qui à ce titre utilise les ressources de l'ordinateur : mémoire, temps de processeur, dispositifs d'entrée-sortie ; c'est principalement le point de vue du système d'exploitation.

Inversement, le processus peut être vu comme le contexte d'exécution du programme.
– Enfin le programme a une existence matérielle sous la forme d'un fichier binaire exécutable stocké quelque part sur une mémoire auxiliaire : c'est un point de vue que nous développerons à la section 3.10 et au chapitre 5.

3.7 Vision dynamique du programme : le processus

Le programme, du point de vue du système, est une entité active qui consomme des ressources, et qui pour les obtenir entre en concurrence avec d'autres demandes. Le programme vu sous cet angle est appelé un processus (notion introduite ci-dessus à la section 3.3 p. 43), *process* en anglais (le terme tâche, *task* en anglais, est parfois employé, dans les sources du noyau Linux par exemple). Ainsi, si à un instant donné quinze personnes utilisent l'éditeur de texte *Emacs* sur le même ordinateur, il y aura quinze processus différents, même s'ils partagent la même copie du programme en mémoire (rappelons-nous la métaphore du pâtissier et des gâteaux, ci-dessus page 43, qui nous a permis d'introduire cette notion de processus).

Le système d'exploitation (*operating system*, OS) est un programme qui arbitre les demandes de ressources des différents processus et les satisfait en se conformant à une stratégie. La stratégie mise en œuvre par le système vise à satisfaire plusieurs impératifs :

– assurer le fonctionnement correct de l'ordinateur, et donc du système lui-même : une allocation incohérente de ressources cruciales comme le temps de processeur ou la mémoire peut provoquer un blocage ou un arrêt complet du système ;
– distribuer les ressources de telle sorte que tous les processus « correctement configurés » en reçoivent une allocation suffisante pour s'exécuter « normalement » ;
– corollaire des deux points précédents : empêcher qu'un processus « pathologique » n'accapare des ressources cruciales et ne réduise les autres à la « famine » ;
– assurer à chaque processus la jouissance paisible des ressources qu'il leur a allouées, et pour cela établir une protection étanche entre les domaines des différents processus, tout en leur permettant de communiquer entre eux s'ils ont été programmés à cet effet. En d'autres termes, c'est au système d'exploitation que revient d'assurer la sécurité de l'ensemble du système informatique.

Le processus réunit deux types d'attributs : certains sont de nature plutôt statique, ce sont les ressources utilisées (espace mémoire, fichiers ouverts), et

d'autre plutôt dynamiques, c'est essentiellement ce que nous pouvons appeler un « fil » d'exécution pour un programme, au sens où l'on dit « le fil de la conversation ». Une tendance récente des architectes de systèmes vise à séparer les deux types d'attributs, en considérant le processus comme un ensemble de ressources qu'utilisent un ou plusieurs fils d'exécution. L'ambiguïté du pluriel de fil en français nous conduit à conserver le terme anglais *thread* plutôt que de recourir à la solution bancale *activité* (qui subsiste sans doute ici ou là dans ce livre). Nous étudierons dans ce chapitre le processus au sens classique. L'étude des *threads* nécessite l'examen préalable des différents types de ressources à partager, elle trouvera sa place au chapitre 10 page 303.

3.8 Attributs du système d'exploitation

Quelles doivent être les caractéristiques d'un système d'exploitation, propres à mettre en œuvre la stratégie décrite ci-dessus? Avant de répondre trop hâtivement à cette question il convient de s'armer de relativisme. Le système d'exploitation des gros ordinateurs centralisés qui ont connu leur apogée pendant les années 1970 ne peut sans doute pas ressembler à celui qui habite dans votre téléphone portable. Moyennant quoi l'examen des systèmes produits du milieu des années 1960 à 2018 révèle une grande stabilité des idées qui ont guidé les réponses aux questions de la section précédente malgré une grande variété de styles de réalisation et d'interfaces personne–ordinateur. C'est sans doute qu'il n'est pas si simple d'imaginer d'autres solutions, ou bien que celles qui se sont dégagées à l'issue des premières expériences se sont révélées assez satisfaisantes dans une grande variété de contextes.

3.8.1 Mode d'exécution privilégié

De ce qui précède découle que le système d'exploitation doit pouvoir faire des choses que les programmes ordinaires ne peuvent pas faire (les programmes ordinaires ne doivent pas pouvoir faire les mêmes choses que le système). Ceci est généralement réalisé par le processeur, qui distingue deux modes d'exécution des instructions: le mode privilégié et le mode normal. Certaines opérations ne sont accessibles qu'au mode privilégié. Nous verrons que certains systèmes ont raffiné cette hiérarchie de modes d'exécution avec plusieurs niveaux de privilèges. Le mode privilégié est aussi appelé mode superviseur, ou mode noyau.

3.8.2 Contrôle des programmes

Lorsque l'on veut exécuter un programme sur un ordinateur piloté par un système d'exploitation, c'est à lui que l'on en demande le lancement. Nous décrirons ce processus plus en détail à la section 3.10 ci-dessous.

3.8.3 Contrôle de l'activité de tous les processus

À partir du moment où le système, premier programme à s'exécuter après le démarrage de l'ordinateur, s'est octroyé le mode d'exécution privilégié, et comme c'est lui qui va lancer les autres programmes, il lui est loisible de leur donner le niveau de privilèges qu'il juge nécessaire, et qui sera sauf exception le mode normal. Il peut également interrompre un programme en cours d'exécution, il contrôle les communications entre processus et empêche toute promiscuité non désirée[6].

3.8.4 Monopole d'attribution des ressources

C'est le système et lui seul qui attribue aux différents processus les ressources dont ils ont besoin, mémoire, temps de processeur, accès aux entrées-sorties. En effet sans ce monopole plusieurs entités pourraient rivaliser pour l'octroi de ressources, de quoi pourrait résulter une situation de blocage. Même avec le monopole du système les situations de blocage entre processus peuvent advenir, mais elles sont plus rares et plus souvent solubles par le système. À titre d'illustration nous allons décrire une situation classique d'interblocage, l'*étreinte fatale*.

Étreinte fatale

Un groupe de processus P_1, P_2, ... P_n est dit en situation d'étreinte fatale si chaque processus P_i est bloqué en attente d'une ressource détenue par un processus P_j différent. Comme aucun processus n'est en mesure de progresser dans son exécution, aucun ne pourra atteindre le point où il libérerait la ressource attendue par un autre, et la situation est donc fatale, sauf si une entité extérieure est en mesure d'intervenir pour interrompre un des processus en espérant débloquer tous les autres en chaîne. Le diagramme du tableau 3.1 illustre le phénomène avec deux processus seulement.

3.8.5 Contrôle de la mémoire

De toutes les ressources, la mémoire est la plus cruciale, sans mémoire aucune information ne peut exister dans l'ordinateur, et bien sûr le système a le monopole de son allocation, de sa protection et de sa libération. Rien ne serait plus grave que l'empiètement d'un processus sur une zone mémoire allouée à un autre, et c'est ce qui arriverait sans une instance unique de contrôle.

6. L'anthropomorphisme débridé de cet alinéa et d'autres à venir peut choquer : le système bien sûr ne désire ni ne juge ni ne s'octroie quoi que ce soit. Les algorithmes écrits par son concepteur et les paramètres qui leur sont fournis sont les seuls déterminants des mécanismes en jeu. Néanmoins ces façons de parler allègent l'expression des périphrases qu'il faudrait sans cesse y introduire. Nous demandons au lecteur d'imaginer leur présence.

Processus P_1	Processus P_2
...	...
Allocation de la ressource A	...
...	Allocation de la ressource B
...	...
Tentative d'allocation de la ressource B : échec, blocage	...
...	...
...	Tentative d'allocation de la ressource A : échec, blocage
...	...
Libération de la ressource A : hélas P_1 n'arrivera jamais là.	...
...	...
...	Libération de la ressource B :
...	P_2 n'y arrivera pas.

Tab. 3.1 : *Étreinte fatale (l'axe du temps est vertical de haut en bas)*

Dans le cas de la multiprogrammation (voir section 3.2) le partage de la mémoire entre les processus est une fonction essentielle du système d'exploitation.

3.8.6 Contrôle des entrées-sorties

L'accès aux dispositifs d'entrée-sortie est un type de ressource parmi d'autres, et à ce titre le système doit en posséder le contrôle exclusif, quitte à déléguer ce contrôle à un processus dans certains cas particuliers. La règle générale est qu'un processus qui veut effectuer une opération d'entrée-sortie (recevoir un caractère tapé sur le clavier, afficher un caractère à l'écran, écrire un bloc de données sur disque...) adresse une demande au système, qui réalise l'opération pour son compte. Ainsi est assuré le maintien de la cohérence entre les multiples opérations, et évitée l'occurrence d'étreintes fatales. Comment le système d'exploitation s'y prend-il pour orchestrer le fonctionnement coordonné de multiples appareils d'entrée-sortie sans conflits ni perte de temps? Nous le verrons plus loin.

Comme conséquence (ou contre-partie) de ce monopole des entrée-sorties, le système en procure aux autres processus une vue abstraite et simplifiée.

3.8.7 Contrôle du temps

Le système maintient une base de temps unique pour tous les processus et fournit des services de gestion du temps aux processus qui le demandent : estampillage, chronologie, attente, réveil...

3.8.8 Contrôle de l'arrêt et du démarrage de l'ordinateur

Nous savons déjà que le système d'exploitation est le premier programme à recevoir le contrôle lors du démarrage de la machine. Il doit aussi recevoir le contrôle de l'arrêt de l'ordinateur, du moins quand c'est possible. Lorsqu'il reçoit une commande d'arrêt, le système veille à terminer les entrées-sorties en cours et à arrêter proprement les processus encore en cours d'exécution. Quand cela n'est pas fait, par exemple lors d'une coupure de courant, certains supports de données externes peuvent rester dans un état incohérent, avec le risque de destruction de données.

3.9 Notion d'appel système

Pour mettre en œuvre les principes énumérés ci-dessus le système d'exploitation reçoit le monopole de certaines opérations, dites *opérations privilégiées* : allouer des ressources, déclencher et contrôler des opérations d'entrée-sortie, d'autres que nous verrons ultérieurement. Mais les programmes ordinaires risquent d'être singulièrement limités si par exemple ils ne peuvent pas faire d'entrées-sorties : il n'y aurait par exemple plus de logiciel de traitement de texte possible parce qu'il ne serait autorisé ni à recevoir le texte frappé au clavier par l'utilisateur, ni à l'afficher à l'écran, ni à l'imprimer. Et nous savons bien qu'il existe effectivement des logiciels de traitement de texte qui font tout cela. Comment ? Je vais vous le narrer.

Lorsqu'un processus ordinaire a besoin d'effectuer une opération privilégiée il demande au système d'exploitation de la réaliser pour son compte, et éventuellement de lui renvoyer le résultat. Cette demande de service est nommée un *appel système*. Les opérations privilégiées sont considérées comme autant de primitives du système d'exploitation, qui peuvent être invoquées par les programmes ordinaires pourvu qu'ils soient dotés des autorisations adéquates. Signalons par exemple, pour Unix :

- `fork`, pour créer un processus ;
- `kill`, pour détruire un processus ;
- `exec`, pour charger un programme en mémoire ;
- `signal`, qui permet à un processus de signaler un événement à un autre processus ;

– `read`, pour lire des données ;
– `write`, pour écrire des données ;
– `brk`, pour allouer ou libérer une zone de mémoire dynamiquement.

3.10 Lancement d'un programme

Nous prendrons l'exemple du système Unix. Unix distingue nettement les notions de *processus*, considéré comme le contexte d'exécution du programme, et le programme lui-même, constitué du texte exécutable en langage machine. Le lancement de l'exécution d'un programme comportera donc deux opérations : la création d'un processus par l'appel système `fork` et le chargement par l'appel système `exec` du programme qui va s'exécuter dans le contexte de ce processus. `exec` est une forme générale parfois spécialisée sous le nom `execve`.

Nous allons décrire les événements qui se déroulent après l'amorçage du système et le démarrage du noyau qui ont été décrits p. 33. Nous prendrons l'exemple d'Unix, qui crée des processus avec l'appel système `fork`. `fork` procède par clonage : le processus fils reçoit au départ une copie de l'environnement du processus père, et c'est `exec` qui va constituer ensuite son environnement propre.

Au commencement, le noyau lance le premier processus qui se déroule en mode utilisateur et dans l'espace mémoire réservé aux utilisateurs : il s'appelle `init`. `init` lance par l'appel système `fork` divers processus système utilitaires, puis (toujours par `fork`) une copie de lui-même associée à chaque terminal destiné aux connexions des utilisateurs. Ces clones d'`init` vont déclencher la procédure d'identification par nom identifiant et mot de passe des utilisateurs qui se connectent[7]. Une fois l'identité authentifiée, le programme `login` utilise l'appel système `execve` pour charger en mémoire un interpréteur de commandes de l'utilisateur, ce que l'on nomme un *shell*. Le *shell* va prendre en mémoire la place de `login` et permettre à l'utilisateur d'interagir avec le système d'exploitation ; ce programme mérite que l'on s'y arrête un instant.

3.10.1 *Shell*

Cette section est consacrée au programme qui est l'intermédiaire principal entre l'utilisateur et le système Unix. C'est à la fois un interpréteur de commandes qui permet le dialogue avec le système et le lancement de programmes, et un langage de programmation. On peut écrire en *shell* des programmes appelés *shell scripts* constitués de séquences de commandes agrémentées de constructions telles qu'alternative ou répétition, ce qui permet d'automatiser des tâches répétitives. Ces programmes ne sont pas compilés (traduits une fois pour toutes), mais

7. Les programmes concernés sont `getty` et `login`

interprétés, c'est-à-dire que le *shell* traduit et exécute les commandes une à une comme si l'utilisateur les tapait sur son clavier au fur et à mesure. L'utilisateur a ainsi l'illusion d'avoir affaire à une machine virtuelle dont le langage machine serait celui du *shell*.

Incidemment, alors que Unix est né en 1969, le *shell* est plus ancien : il a été inventé en 1963 lors de la réalisation du système d'exploitation CTSS au *Massachusetts Institute of Technology* par Louis Pouzin [8], ingénieur français qui s'est également illustré en 1970 à la direction du projet de réseau Cyclades, où il inventa notamment le datagramme. CTSS, dont le développement commença en 1961 sous la direction de Fernando Corbató au MIT sur IBM 709, puis 7090, fut un système d'une grande importance par le nombre de notions, de techniques et de réalisations novatrices qu'il apportait. Ce fut le premier système à temps partagé, c'est-à-dire qu'il permettait l'usage simultané de l'ordinateur par plusieurs utilisateurs qui entraient des commandes et lançaient des programmes depuis des terminaux, en utilisant la disproportion entre le temps de calcul de la machine et le temps de réaction humain telle qu'expliqué à la section 3.2.1.

Après CTSS, Corbató prit la tête du projet MAC [9] [10] qui donna naissance au système d'exploitation Multics sur General Electric GE-645, toujours au MIT. Ces systèmes ont inspiré les auteurs de Unix, tant par ce qu'ils en ont retenu que par ce qu'ils en ont rejeté d'ailleurs. Nous y reviendrons.

De CTSS le *shell* passa en 1964 à son successeur *Multics*, et de là à Unix. L'utilisateur d'Unix a d'ailleurs le choix entre plusieurs *shells* qui diffèrent par le style plus que par les fonctions. La souplesse et la programmabilité du *shell* sont pour beaucoup dans la prédilection que les développeurs professionnels ont pour Unix. Les utilisateurs moins spécialisés ont tendance à lui préférer les interfaces graphiques interactives offertes par le Macintosh ou Windows, et d'ailleurs disponibles également sous Unix grâce au système de fenêtres **X** complété plus récemment par des environnements graphiques tels que *Gnome* ou *KDE*. Mais pour un utilisateur quotidien taper des commandes au clavier est plus rapide que de cliquer avec une souris, et surtout ces commandes peuvent

8. `http://multicians.org/shell.html`

9. Au cinquième étage du bâtiment du MIT qui l'abritait, MAC signifiait *Multiple Access Computers*, au neuvième étage, *Man and Computer*. L'initiateur du projet était Robert M. Fano, professeur au MIT, sur une suggestion de J. C. R. Licklider.

10. J. C. R. Licklider est une des personnalités dont l'influence sur le développement de l'informatique a été la plus forte. Il fut directeur de la division *Information Processing Techniques Office (IPTO)* de l'ARPA *(Advanced Research Projects Agency)*, une agence du ministère américain de la défense, dans les années 1960, et fut ainsi à l'origine de projets qui débouchèrent sur les interfaces personnes–ordinateurs que nous utilisons aujourd'hui, ainsi que sur la création de l'Internet. Ses deux articles les plus célèbres portent des titres prophétiques : *Man-Computer Symbiosis* (1960) et *The Computer as a Communications Device* (1968, en collaboration avec Robert Taylor).

être enregistrées pour devenir des *shell scripts*, ce qui assure programmabilité, mémorisation et reproductibilité des actions. Avez-vous déjà essayé de vous rappeler quelle séquence de coups de souris avait produit tel résultat subtil et ardemment désiré dans Word ? ou de dicter au téléphone une telle séquence à un collègue (sauf dans le cas où il est devant son écran et peut effectuer les actions au fur et à mesure de la dictée) ? Tandis qu'une série de commandes du *shell*, cela s'envoie par courrier électronique de façon sûre.

Bref, muni du *shell*, rien n'est plus simple à l'utilisateur que de lancer un programme : il suffit de taper le nom du fichier binaire exécutable qui en contient le texte en langage machine, éventuellement suivi de quelques paramètres, puis un retour chariot et l'exécution commence... Il y aura création d'un nouveau processus pour ce programme, un processus en général fils du processus lié au *shell*. Unix procure aussi des moyens de lancer l'exécution d'un programme en l'absence d'un humain pour taper la commande, mais le principe reste le même.

3.11 Synchronisation de processus, interruption

Nous avons dit que le système d'exploitation, avec l'aide de dispositifs appropriés du matériel, pouvait répartir le temps de processeur entre plusieurs processus pseudo–simultanés. Ceci suppose qu'un processus, à un instant donné, puisse être dans l'état actif, à un instant suivant dans l'état dormant (en attente), puis encore à un autre instant redémarrer, c'est-à-dire passer de l'état dormant à l'état actif.

Nous pouvons concevoir qu'un processus actif se mette volontairement à l'état dormant. En revanche le passage de l'état dormant à l'état actif suppose l'intervention du système d'exploitation ou d'un autre processus pour réveiller le processus endormi. Comment cela peut-il se passer ? Nous allons pour l'exposer prendre un exemple particulièrement significatif et qui découle de la section 3.2 ci-dessus : le déroulement d'une opération d'entrée-sortie, lecture ou écriture sur support externe.

3.11.1 Demande d'entrée-sortie

Les opérations d'entrée-sortie, nous l'avons vu, sont des opérations privilégiées, du monopole du système d'exploitation. Lorsqu'un logiciel veut effectuer une entrée-sortie il doit effectuer un appel système. Pour réaliser cette opération d'entrée-sortie (universellement désignée par IO, comme *input-output*, E/S en français), plusieurs composants de l'ordinateur et du système entrent en jeu :

– Le programme effectue un appel système.
– Le système exécute un programme spécial, dit pilote de périphérique *(driver)*, qui transmet la demande au contrôleur de périphérique. Le contrô-

leur est un circuit de commande du périphérique physique, qui dans le cas des disques durs est un véritable petit ordinateur spécialisé doté de mémoire pour le stockage des données en transit et de capacités de multiprogrammation pour conduire plusieurs disques simultanément.

– Le périphérique physique effectue l'action réelle : lire, écrire, enregistrer, imprimer, émettre un son... Puis il prévient le contrôleur quand il a fini en émettant un signal particulier sur un fil particulier. La durée de l'action du périphérique mécanique, nous l'avons vu, est beaucoup plus longue que toutes les actions des composants électroniques : de l'ordre de 100 000 fois plus longue pour une écriture sur disque.

Comment cela finit-il ? Nous allons le voir. Le diagramme des opérations est décrit par la figure 3.1.

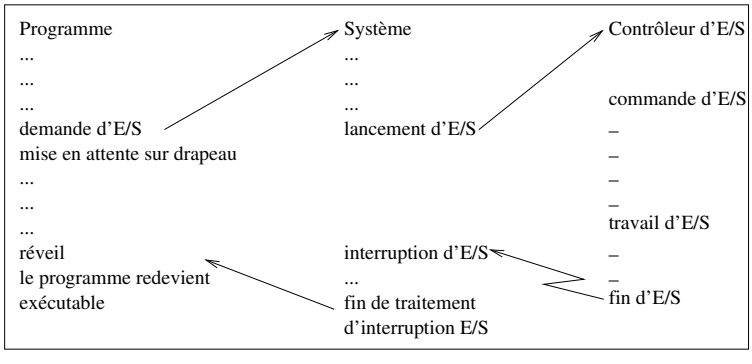


Figure 3.1 : *Diagramme d'une opération d'entrée-sortie (E/S)*

Le haut de ce diagramme correspond aux étapes initiales d'une opération d'entrée-sortie, elles sont compréhensibles avec les notions que nous possédons déjà :

– Le programme s'exécute normalement, puis il émet une demande d'entrée-sortie (un appel système).
– Immédiatement après la demande d'entrée-sortie [11] le programme se met volontairement en attente, en sommeil. La mise en sommeil se fait par un appel système qui transfère le contrôle au superviseur après avoir sauvegardé en mémoire le contexte d'exécution du processus (PSW, registres).
– Pour que cette mise en sommeil ne soit pas définitive il faut prévoir un mécanisme de réveil. En effet un programme qui s'est arrêté ne pourra pas

11. Par immédiatement on entend « avant la fin de l'exécution de l'E/S par le matériel », contrôleur et périphérique proprement dit, ce qui, nous l'avons vu à la section 3.2.1, laisse au processeur largement le temps d'exécuter quelques milliers d'instructions

se remettre en action spontanément. On pourrait imaginer un hôtel où chaque client avant de se coucher accrocherait à l'extérieur de sa chambre une petite fiche où il aurait écrit l'heure à laquelle il souhaite être réveillé et ce qu'il veut manger à son petit déjeuner. Au matin, l'employé d'étage effectue une ronde périodique dans le couloir, consulte les fiches et réveille les clients dont le temps de sommeil est expiré. Dans notre cas l'appel système (`sleep` ou `wait` selon les cas) place à un endroit connu du système les informations qui permettront, lorsque l'E/S sera terminée, de savoir quel processus réveiller et quelle information lui donner. Sinon on ne saurait jamais établir le lien entre le résultat de l'entrée-sortie et le programme qui l'avait demandée, et celui-ci ne pourrait jamais reprendre son exécution. Un tel ensemble d'informations est appelé une structure de données. Son adresse est mentionnée par une toute petite structure appelée, par exemple, bloc de contrôle d'événement (ECB, *Event Control Block*) dans les systèmes OS/360.

– La demande d'entrée-sortie est prise en charge par le système, qui s'empresse de la mettre en file d'attente. Sur un système en multiprogrammation les demandes d'entrée-sortie sont en effet multiples et il faut y mettre un ordre.
– La partie du système chargée de traiter les demandes d'entrée-sortie va, plus tard, extraire notre demande de la file et la traiter, soit en bref la transmettre au contrôleur, qui la transmettra au périphérique physique. Et là commencera le délai, incommensurablement long au regard de ce qui précède, nécessaire à l'action elle-même.

3.11.2 Interruption de fin d'entrée-sortie

Puis l'entrée-sortie suit son cours, et viendra le moment où le périphérique (disque dur, clavier, écran...) aura fini son travail. Comment le signaler au programme interrompu ? Celui-ci est dormant, il ne peut recevoir d'information. Il faut donc passer par l'intermédiaire du système d'exploitation.

À l'issue du délai considérable durant lequel le périphérique a travaillé, il envoie au contrôleur un signal pour le prévenir, quelques informations qui constituent un compte-rendu d'exécution de la tâche, et éventuellement les données qui lui étaient demandées, si par exemple il s'agissait d'une lecture de données sur un disque. Voyons la suite des opérations, pour laquelle nous allons supposer qu'il s'agit précisément d'une lecture sur disque.

– Quand le contrôleur a reçu la demande d'entrée-sortie, elle contenait un certain nombre de renseignements, notamment l'adresse de la zone de mémoire où il faut déposer les données résultant de la lecture. Il place donc les données à l'emplacement convenu.

- Comment le contrôleur place-t-il les données en mémoire ? Par où passent-elles ? Par le bus. Le bus comporte des lignes de données et des lignes de signalisation (ou de commande). Les lignes de signalisation permettent aux différents éléments de l'ordinateur de coordonner leurs actions, d'échanger des commandes ; les lignes de données leur permettent d'échanger, donc, des données. Le contrôleur sait accéder à la mémoire et sélectionner la bonne adresse.
- Une fois les données placées au bon endroit en mémoire, il faut prévenir le processeur. À cette fin le contrôleur envoie sur une ligne de signalisation particulière un signal qui va déclencher une interruption du processeur. L'interruption est un mécanisme capital pour la synchronisation des ordinateurs, nous allons en exposer le principe.
- Le signal d'interruption a deux effets :
  - il interrompt, donc, le programme en cours d'exécution ;
  - le PSW courant (cf. section 2.5 pour la définition du PSW et du PC) est sauvegardé en mémoire et remplacé par un nouveau PSW qui comporte une valeur de PC qui pointe vers une section particulière du système, le superviseur d'interruption ; c'est donc ici que va se continuer l'exécution.
- La première chose que fait le superviseur d'interruption est de déterminer la nature de l'interruption. Ici le signal lui donne la réponse : interruption d'entrée-sortie, mais il y a d'autres types (interruption volontaire par appel système, interruption déclenchée par l'horloge interne, interruption provoquée par une condition particulière de programme comme dépassement de capacité numérique...). Il se débranche donc à la section appropriée, le superviseur d'interruption d'entrée-sorties.

Le traitement des interruptions et le transfert du contrôle à la section adéquate du superviseur ou du noyau sont des éléments de l'architecture de l'ordinateur et du système d'exploitation cruciaux pour les performances et les possibilités de l'ensemble. Les éléments matériels et logiciels sont étroitement associés. Nous avons décrit une réalisation possible. Il en existe d'autres, par exemple l'architecture Intel IA-64 utilise un vecteur d'interruptions : chaque élément matériel ou logiciel du système susceptible de déclencher une interruption, et notamment chaque contrôleur de périphérique, est associé à une structure de données résidant à une adresse en mémoire fixée au démarrage du système et qui contient elle-même l'adresse de la section appropriée du superviseur. L'ensemble de ces adresses constitue le vecteur d'interruptions. L'occurrence d'une interruption provenant de tel contrôleur déclenche automatiquement le transfert du contrôle à l'adresse correspondante,

qui pointe sur la section appropriée du superviseur. Ce dispositif, dit de vectorisation des interruptions, est apparu sur les ordinateurs PDP 11 de *Digital Equipment*.

- Une fois le contrôle transféré au superviseur d'interruption d'entrée-sortie, celui-ci retrouve dans ses tables la référence du drapeau associé à la demande d'entrée-sortie concernée, par là il retrouve la structure de données qui la décrit, puis le processus dormant qui l'avait émise.
- Le superviseur fait passer le processus émetteur de l'état dormant à l'état *dispatchable* ou prêt, c'est-à-dire candidat à redevenir actif, éligible pour l'exécution.
- Le superviseur d'interruptions passe ensuite la main à une autre partie du système, l'ordonnanceur, ou programmateur (en anglais *scheduler*).

3.12 Ordonnancement de processus

Les systèmes que nous envisageons permettent la multi-programmation, c'est-à-dire que plusieurs processus sont à un moment donné en concurrence pour disposer du processeur, dont on rappelle qu'en vertu de l'architecture de von Neumann il exécute une seule instruction à la fois[12]. Pour permettre cette concurrence, aux deux sens du terme, commercial et étymologique, il faut que le système soit capable de retirer le contrôle du processeur à un processus pour le donner à un autre. La partie du noyau du système qui fait cela est l'ordonnanceur ou programmateur (en anglais *scheduler*).

L'ordonnanceur reçoit la main à un moment où tous les processus sont dans l'état d'attente. Certains sont prêts (éligibles) pour l'exécution *(dispatchable)*, c'est-à-dire qu'ils ne sont pas en attente d'un événement tel qu'une fin d'entrée-sortie ou l'expiration d'un délai, et qu'ils n'attendent que le feu vert pour recommencer à s'exécuter. D'autres sont à l'état dormant, par exemple en attente sur un drapeau. Le rôle de l'ordonnanceur est de sélectionner parmi les processus prêts celui qui va être activé. La méthode de sélection peut dépendre de plusieurs paramètres: délai d'attente déjà écoulé pour chaque processus, niveau de priorité, etc. Puis l'ordonnanceur passe la main au distributeur (en anglais *dispatcher*), qui remet effectivement en activité le processus sélectionné en restaurant son contexte d'exécution (PSW, registres). C'est sur ce processus complexe que s'appuie l'exemple de la figure 3.1.

12. Rappelons aussi que tous les systèmes utilisés réellement aujourd'hui (26 juin 2020) sont conformes macroscopiquement à l'architecture de von Neumann, c'est-à-dire que les modifications qu'ils lui apportent ne modifient pas substantiellement les conséquences que l'on peut tirer du principe d'exécution séquentielle.

Cette description du traitement d'entrée-sortie nous a amené à préciser notre vision de la gestion des processus et à comprendre le fonctionnement effectif de la multiprogrammation. Tout programme réel interagit avec le monde extérieur par des entrées-sorties ; ce faisant, nous venons de le voir, il se met en attente d'un résultat d'entrée-sortie, et par là même il donne à l'ordonnanceur l'occasion de donner la main à un autre programme. Et c'est ainsi que chaque programme s'exécute à son tour, en pseudo–simultanéité avec les autres. Nous avons vu aussi le rôle fondamental des interruptions pour le fonctionnement de la multiprogrammation.

Nous comprenons aussi qu'un programme qui ferait beaucoup de calculs et très peu d'entrées-sorties, par exemple un programme qui devrait calculer et afficher la millionième décimale de π, risquerait de bloquer tous les autres parce qu'il ne rendrait jamais la main. Pour parer une telle menace, plusieurs solutions sont envisageables.

3.12.1 Stratégies d'ordonnancement

La solution d'ordonnancement la plus simple consiste à découper le temps en tranches et à dire qu'aucun processus ne pourra avoir la main pendant deux tranches de temps consécutives. Chaque expiration d'une tranche de temps déclenche une interruption et donne la main à l'ordonnanceur, qui peut ainsi éviter la monopolisation du processeur par un programme gourmand.

Une autre solution consiste à affecter à chaque processus une priorité. L'ordonnanceur donne toujours la main au processus prêt (*dispatchable*) de plus haute priorité. Il suffit de donner une priorité basse aux processus qui font peu d'entrées-sorties et une priorité haute à ceux qui en font beaucoup, et dont on sait qu'ils vont se mettre en attente « volontairement » souvent.

Il est possible de combiner toutes ces stratégies de répartition du temps de processeur pour obtenir un système auto–régulé. Nous aurons des tranches de temps et des priorités, qui de surcroît seront variables dynamiquement. Un processus aura deux façons de s'interrompre : soit « volontairement » en faisant une demande d'entrée-sortie ou tout autre appel système suivi d'une mise en attente, soit en atteignant la fin d'une tranche de temps. L'ordonnanceur se voit attribuer une prérogative supplémentaire : les « bons » processus qui se seront interrompus « volontairement » verront leur priorité augmentée, les « mauvais » processus qui auront atteint la limite d'une tranche de temps, manifestant par là une tendance néfaste à la monopolisation, verront leur priorité diminuée, ce qui améliorera la fluidité de la multiprogrammation. L'hypothèse sous-jacente est : qui a fait des entrées-sorties, en refera ; qui a calculé, calculera. Notons néanmoins que ce dispositif n'implique pas de prédestination, et qu'il laisse grande ouverte la porte de la rédemption.

3.12.2 Interruptions et exceptions

Nous avons examiné le cas particulier de l'interruption d'entrée-sortie, qui est provoquée par un élément matériel extérieur au processeur, indépendant du cadencement des instructions. C'est une interruption *asynchrone*. Il existe par ailleurs des interruptions provoquées par le processeur lui-même, par exemple lorsqu'il détecte une condition anormale, ou simplement à la demande d'un programme. Ces interruptions sont *synchrones*, parce que le processeur ne les produit qu'après avoir terminé l'exécution d'une instruction, et elles sont aussi nommées *exceptions*.

3.12.3 Préemption

Ainsi que nous venons de le voir, le fonctionnement du processeur est cadencé par des interruptions. Une interruption peut survenir du fait de la terminaison d'une entrée-sortie, de l'expiration de la tranche de temps de processeur allouée à un processus, de l'occurrence d'une erreur du système ou du matériel, ou simplement à la demande d'un processus, comme lors d'une demande d'entrée sortie.

À chaque interruption, l'ordonnanceur prend la main. C'est pour cela que les interruptions jouent un rôle si important dans le fonctionnement du système. L'ordonnanceur examine la file d'attente des processus prêts (éligibles) pour l'exécution *(dispatchable)*, comme déjà dit. Souvent, et même presque toujours, le déclenchement d'une interruption procède d'un événement à la suite duquel cette file d'attente est modifiée : après une demande d'entrée-sortie, le processus qui l'a émise, et qui était donc actif, entre dans l'état non-prêt (dormant) ; au contraire, après la terminaison d'une entrée-sortie, le processus qui en attendait le résultat redevient prêt. Les interruptions sont les seules circonstances à l'occasion desquelles un processus peut passer d'un état (prêt, non-prêt, actif, terminé) à un autre.

Dans tous les cas, l'ordonnanceur examine la file d'attente sans préjugé et donne la main au processus prêt de plus haute priorité, sans respect pour les positions acquises. Il y a des exceptions : par exemple, si le système est paramétré pour une stratégie d'ordonnancement par tranches de temps, et si le processus le plus prioritaire vient d'épuiser la tranche précédente, la règle de répartition interdit de lui rendre la main. Mais de façon générale, tout processus de haute priorité redevenu prêt prendra la main au processus moins prioritaire qui s'exécutait jusqu'alors. On dit que le système d'exploitation qui permet ce transfert de contrôle du processeur est un système *préemptif*. Un vrai système d'exploitation doit être préemptif.

3.12.4 Synchronisation de processus et sections critiques

Nous sommes bien contents d'avoir un système préemptif, mais si nous réfléchissons un peu nous voyons apparaître quelques inconvénients. Un processus qui s'exécute peut à tout moment être interrompu, même en plein milieu d'une instruction en cas d'interruption asynchrone (et les interruptions asynchrones comportent, dans le cas des multi-processeurs, les interruptions par un autre processeur accédant à la mémoire commune), le noyau du système d'exploitation va prendre la main pour traiter l'interruption, et à l'issue de ce traitement ce sera peut-être un autre processus qui recevra la main.

Ce risque d'être interrompu à tout instant impose des précautions. Les éléments essentiels du vecteur d'état du programme doivent pouvoir être sauvegardés, afin de permettre la reprise du traitement. Ces éléments sont essentiellement la valeur du PSW, qui permet notamment de retrouver l'instruction en cours au moment de l'interruption, et le contenu des registres, qui permet de retrouver les différentes zones de mémoire utilisées. Comme nous le verrons au chapitre 4, dans un système à mémoire virtuelle moderne chaque processus dispose de son espace de mémoire virtuelle privé, ce qui simplifie les choses. Cette opération de sauvegarde du contexte d'exécution du processus interrompu et d'initialisation du contexte d'exécution du processus promu s'appelle commutation de contexte *(context switch)*. Selon les processeurs et les systèmes, ce peut être une opération figée dans le matériel et invoquée par une instruction spécifique unique, ou une séquence d'instructions répétées comme un refrain au début de chaque section de programme, comme dans l'OS 360/370. Nous avons déjà rencontré ce mécanisme aux sections 2.5 et 3.11.1.

Mais même si ces précautions ont été prises, il y a des opérations au cours desquelles une interruption, suivie ou non d'une commutation de contexte, risque d'avoir des conséquences catastrophiques. La séquence des instructions qui constitue le programme d'une telle opération est appelée *section critique*. Les sections critiques sont généralement dans le noyau du système d'exploitation. Le cas le plus courant est celui d'une allocation de ressource matérielle, ce qui suppose la mise à jour de tables : dans le cas de l'allocation d'une zone de mémoire réelle, la table des cadres de pages (ici nous anticipons sur le chapitre 4), dans le cas d'une écriture sur support externe, la table des blocs libres sur le disque et la i-liste, qui décrit la cartographie des fichiers et des blocs qui leur sont alloués sur le disque (ce sera vu au chapitre 5).

Lorsqu'un processus veut acquérir une ressource contrôlée par le noyau, il émet un appel système. Le traitement de cet appel système va résulter en l'allocation de la ressource, ce qui se traduit en mémoire par la modification des tables qui décrivent cette ressource. L'exécution de la séquence des instructions du noyau qui effectuent ce traitement est appelée chemin de contrôle du noyau. Tout ou partie du chemin de contrôle est une section critique.

La programmation d'une telle section critique impose une protection particulière pour préserver l'intégrité des données, il faut garantir une des deux conditions suivantes :

Assertion 1 : Aucune interruption ne pourra avoir lieu pendant le déroulement de la section critique.

OU BIEN :

Assertion 2 : Si une interruption survient, elle peut avoir pour effet de rendre prêt un chemin de contrôle du noyau qui était dormant, par exemple dans le cas d'une interruption de fin d'entrée-sortie. De ce fait, le chemin de contrôle du noyau qui était en train d'allouer des ressources à cet instant peut perdre la main au profit du nouveau concurrent, peut-être doté d'une priorité plus élevée. Il faut alors garantir que les tables et les autres structures de données en cours de modification ne pourront pas être modifiées par ce nouveau chemin de contrôle.

C'est compliqué, et plusieurs méthodes peuvent être employées selon les circonstances.

Nous avons introduit la notion de système préemptif, qui permet aux processus de prendre la main à des processus moins prioritaires lorsqu'ils repassent à l'état prêt. Lorsqu'un processus émet un appel système et qu'il exécute des instructions du noyau, en mode noyau (on dit aussi superviseur) donc, il n'est par construction pas préemptible par un processus en mode utilisateur (qui exécute des instructions ordinaires écrites par le commun des mortels). Mais ne risque-t-il pas la préemption par un autre processus en mode noyau ? Cela dépend du système. Les premières versions du noyau Linux n'étaient pas préemptives, un processus en mode noyau ne pouvait pas subir la préemption. Mais même un noyau non préemptif doit tenir compte des interruptions et des systèmes multi-processeurs. Les moyens d'assurer la véracité d'une de ces deux assertions sont examinées ci-dessous.

Le noyau Linux version 2.4 a vu apparaître un « patch » développé par Robert Love, destiné à le rendre préemptif. Le résultat est une amélioration assez spectaculaire des temps de réponse des processus. Le prix à payer est une complexité accrue, mais avec l'avantage associé d'un code[13] de meilleure qualité, intrinsèquement adapté aux multi-processeurs.

13. Le terme *code* est employé ici dans l'acception de « texte du programme ». On parle de code source pour désigner le texte du programme tel qu'il a été écrit par le programmeur dans le langage de développement, le code objet est le programme traduit en langage machine par le compilateur, le code exécutable est le programme sous forme binaire auquel l'édition de liens a ajouté tous les sous-programmes compilés séparément et les références aux bibliothèques partagées nécessaires à l'exécution. On notera que les détracteurs de l'informatique utilisent de façon péjorative la série de termes code, coder, codage pour dévaluer l'activité de programmation en suggérant que ce serait une activité triviale, l'application mécanique d'un code.

Atomicité des opérations

Le premier moyen qui vient à l'idée pour interdire la préemption d'un processus en section critique dans le noyau, c'est d'assurer l'ininterruptibilité de la section critique. Le moyen le plus radical, c'est que cette section critique soit réduite à une instruction unique. Même cela ne suffit pas, puisqu'une instruction qui consomme plusieurs cycles de processeurs (cas courant pour les processeurs CISC[14] tels que le Pentium) peut être interrompue par une interruption asynchrone émise par un organe périphérique ou en provenance d'un autre processeur sur un système multi-processeur. Il faut donc que la section critique soit composée d'une instruction unique sur un cycle unique. Là on est sûr que le système reste dans un état où la section critique est exécutée soit totalement soit pas du tout : cette propriété d'une opération est nommée *atomicité*, et elle garantit que :

- si le système était dans un état cohérent avant l'opération ;
- si l'opération n'introduit pas d'incohérence ;
- le système est cohérent à l'issue de l'opération.

La solution répond parfaitement à l'énoncé, le problème est qu'en un cycle de processeur on ne fait pas grand chose, et notamment on n'accède pas à la mémoire principale, on ne peut donc espérer effectuer de cette façon la mise à jour complexe de tables d'allocation de ressources. En fait, il est seulement possible, au prix d'une certaine virtuosité dans la conception des circuits logiques, de tester une position de mémoire (par exemple un registre du processeur) qui représente un drapeau logique (0 libre, 1 verrouillé, par exemple) et, dans le même cycle, si le test est favorable, d'en modifier une autre. Cette instruction est généralement nommée **TAS** *(Test and Set)*. Un programme en langage C ne peut garantir l'atomicité d'une de ses expressions, parce que cela dépend de l'usage qui sera fait du langage machine par le compilateur. Pour franchir cet obstacle, le noyau Linux procure des fonctions destinées à faire usage des possibilités « atomiques » du langage machine, nommément `atomic_dec_and_test(v)` et `atomic_inc_and_test(v)`.

Les processeurs modernes disposent d'instructions éventuellement plus complexes mais qui assurent l'atomicité de façon brutale, même dans le cas de multi-processeurs, en bloquant l'accès au bus mémoire jusqu'à la fin de l'instruction.

14. Les abbréviations CISC (pour *Complex Instruction Set Computer*), RISC (pour *Reduced Instruction Set Computer*) et VLIW (pour *Very Long Instruction Word*) désignent des classes dans la zoologie des processeurs, dont la signification est donnée à la section 9.4 p. 276.

Les opérations atomiques sont très limitées par leur concision, mais elles peuvent servir de base à des mécanismes plus complexes de protection de sections critiques, comme ceux que nous allons examiner ci-dessous.

Masquage des interruptions

Tous les processeurs courants possèdent un dispositif de masquage des interruptions. Pour la gamme IBM 360 et ses successeurs, il s'agit simplement d'un bit du PSW, pour les processeurs Intel récents un champ du registre `eflags`. Ici encore, nous disposons d'un moyen radical de protection d'une section critique : aucune interruption ne peut se manifester tant que le drapeau est positionné. Les limites de cette méthode tiennent à sa puissance même : pendant que les interruptions sont masquées, tous les échanges entre le processeur et les organes périphériques sont bloqués. De plus, il est impossible de masquer les interruptions pendant une séquence d'instructions qui risque elle-même de faire une demande d'entrée-sortie, c'est-à-dire d'entrer dans un état dormant : le système ne se réveillera jamais et restera gelé, le seul recours sera le bouton RESET...

Verrouillage de la section critique

Quand il faut créer une section critique trop longue ou trop complexe pour qu'il soit possible de la réaliser atomiquement, ou de la protéger en masquant les interruptions, il faut recourir au verrouillage par un procédé plus complexe. Il existe classiquement trois familles de procédés, dont on peut démontrer qu'elles donnent des résultats équivalents : les sémaphores inventés par Edsger Wybe Dijkstra, les moniteurs dûs à C. Antony R. Hoare[59], les bibliothèques de fonctions.

Toutes les méthodes de verrouillage reposent sur des principes communs. Un chemin de contrôle du noyau qui doit accéder, par exemple, à une table d'allocation d'une ressource système doit auparavant acquérir un verrou pour cette table. Le verrou n'est rien d'autre qu'une structure de données en mémoire : c'est purement logique, il ne faut imaginer là aucun dispositif matériel qui verrouillerait physiquement une zone de mémoire, ou un élément du processeur, ou un périphérique. C'est-à-dire qu'il n'a d'efficacité que parce que tous les chemins de contrôle susceptibles d'accéder à la même ressource utilisent la même convention de verrouillage, donc cherchent à acquérir le même verrou. En pratique, ils doivent utiliser tous la même séquence d'instructions. Inutile de dire que cette séquence est partie intégrante du noyau, et qu'il est vivement conseillé de ne pas laisser les programmes en mode utilisateur accéder aux félicités du verrouillage... ce que s'empressent de faire, bien évidemment, les applications destinées aux systèmes non préemptifs, avec les résultats que chacun peut constater.

Dans son état le plus simple, le verrou se réduit à un bit : libre ou occupé. Il peut avoir une structure plus complexe, et accéder ainsi au rang de sémaphore, qui confère les champs suivants (pour le noyau Linux) :

count valeur numérique : si elle est supérieure à 0, la ressource est libre, si elle est inférieure ou égale à 0 la ressource est occupée et la valeur absolue de **count** est égale au nombre de chemins de contrôle qui attendent pour y accéder ;

wait adresse de la file d'attente des processus [15] en attente d'accès à la ressource ;

waking une variable dont la valeur sert à sélectionner dans la file d'attente le prochain chemin de contrôle qui accédera à la ressource, selon un algorithme dont nous dirons deux mots ci-dessous.

Bref, le sémaphore est un verrou doté d'une valeur qui dit combien de clients on peut laisser passer à la fois. Traditionnellement, les sémaphores sont manipulés au moyen de deux opérations primitives mystérieusement nommées **P** et **V**, initiales de leurs noms néerlandais [16], puisque œuvre d'E.W. Dijkstra. **P** est invoquée par un processus qui cherche à accéder à la ressource, **V** par un processus qui la libère et laisse la place au suivant. Claude Kaiser, un des auteurs de Crocus [38], m'a confié un procédé mnémotechnique pour ne pas les confondre : **P** pour « Puis-je ? » et **V** pour « Vas-y ! ». Le noyau Linux les a prosaïquement baptisées **down** (qui décrémente **count**) et **up** (qui incrémente **count**).

Lorsqu'un processus invoque **P** avec en argument l'adresse d'un sémaphore, la primitive décrémente le champ **count** du sémaphore et examine son signe (les deux actions au moyen d'une unique instruction atomique vue à la section 3.12.4, sinon un autre processus pourrait tenter la même opération en même temps). Si **count** est positif ou nul, le processus acquiert le contrôle du sémaphore, et l'exécution continue normalement. Si **count** est négatif, le processus entre dans l'état dormant et est placé dans la file d'attente désignée par **wait**.

Lorsque le processus qui contrôlait le sémaphore a fini d'utiliser la ressource correspondante, il invoque la primitive **V**. Celle-ci incrémente le champ **count** et examine son signe (les deux actions au moyen d'une unique instruction atomique vue à la section 3.12.4, sinon un autre processus pourrait tenter la même opération en même temps). Si **count** est positif, aucun processus n'attendait la ressource, et l'exécution de **V** se termine. Sinon, elle incrémente le champ **waking** (opération protégée par un verrou et le masquage d'interruptions pour éviter toute concurrence) et réveille les processus de la file d'attente pointée par

15. Conformément à l'usage nous employons ici et dans les lignes qui suivent « processus » à la place du fastidieux « chemin d'accès du noyau exécuté pour le compte d'un processus » qui serait plus exact.

16. **P** est pour *passeren*, passer, et **V** pour *vrijgeven*, libérer.

wait. Chaque processus réveillé exécute alors la suite de **P**, qui contient une section critique pour tester si **waking** est positif. Le premier à exécuter cette section critique décrémente **waking** et acquiert le contrôle du sémaphore, les suivants trouvent **waking** négatif ou nul et se rendorment.

L'effet produit par l'invocation de **P** et de **V** dépend de la valeur initiale de **count** ; si elles est de 1, **P** et **V** réalisent l'exclusion mutuelle entre des processus qui essaient d'accéder à une ressource partagée ; **P** est exécuté à l'entrée d'une section critique, **V** à sa sortie, et le résultat est que seul un processus peut s'exécuter dans la section critique à la fois.

Cet algorithme donne le résultat voulu parce que tous les processus en concurrence exécutent les mêmes appels système et obéissent aux mêmes conventions pour se synchroniser. Inutile de dire que dans un système (ou prétendu tel) non préemptif qui table sur la bonne volonté coopérative des logiciels d'application pour assurer cette cohérence, il suffit d'un logiciel mal écrit pour provoquer des catastrophes, et l'expérience prouve que cela se produit.

3.13 Chronologie des premiers systèmes d'exploitation

1956 - Ainsi qu'il a été signalé au début de ce chapitre, l'ancêtre fruste (mais déjà porteur des idées principales) des systèmes d'explotation est GM-NAA I/O *(General Motors and North American Aviation Input/Output system)*, créé en 1956 par Robert L. Patrick des laboratoire de recherche de General Motors et Owen Mock de North American Aviation pour l'IBM 704. Il s'agissait en fait d'un moniteur d'enchaînement de programmes avec des possibilités de recouvrement du temps de calcul et du temps d'entrées-sorties.

1957 - Atlas Supervisor est créé à l'université de Manchester pour l'ordinateur Atlas, fruit d'une collaboration entre l'université et les industriels Ferranti et Plessey. Atlas et son système d'exploitation furent porteurs de nombreuses innovations : mémoire virtuelle (1962), multi-programmation, *Direct Memory Access* (DMA) pour les périphériques, etc. Atlas Supervisor est peut-être le premier véritable système d'exploitation.

1960 - IBSYS pour les IBM 7090 et 7094.

1960 - MCP *(Master Control Program)* de la société Burroughs. C'est le premier système écrit dans un langage de haut niveau (un dérivé d'Algol) et également le premier système gérant des multiprocesseurs.

1961 - CTSS *(Compatible Time-Sharing System)*, créé au MIT sous la direction de Fernando Corbat pour un ordinateur IBM 7094, modifié : taille de mémoire doublée, horloge programmable, dispositif de « trappe » (déroute-

ment programmé lors de l'exécution de certaines instructions). CTSS fut le premier système de temps partagé [72].

1964 - OS/360 d'IBM, premier système conçu pour toute une gamme d'ordinateurs de puissances très différentes, est annoncé. Il ne sera effectivement opérationnel qu'un peu plus tard. Plus d'un demi-siècle après sa mise en service, ce système est toujours opérationnel sous le nom de z/OS.

1965 - Multics *(Multiplexed Information and Computing Service)*, successeur de CTSS, développé par la même équipe dirigée par Fernando Corbató, écrit en langage évolué (PL/1) comme MCP, implanté sur matériel General Electric GE-645. À l'origine de nombreuses innovations : protection du système par anneaux (cf. 206), mémoire segmentée, liaison dynamique des bibliothèques de programmes, mémoire virtuelle paginée, système de gestion de fichiers, etc. Multics est l'ancêtre d'Unix.

1969 - Unix, amplement décrit et commenté dans ce livre.

Cette liste est bien sûr incomplète, il faudrait y ajouter les systèmes de Digital Equipment (RSX/11M, VMS, etc.), de Microsoft (MS/DOS, Windows), de Digital Research (CP/M), d'Apple (MacOS) et beaucoup d'autres.

Aujourd'hui ne survivent pratiquement que trois souches : z/OS pour les *mainframes* IBM, Windows de Microsoft (soit dit en passant héritier adultérin de VMS de Digital Equipment, par débauchage de son concepteur principal David Cutler), et Unix. On notera que la famille Unix englobe Linux, macOS, iOS, Android, FreeBSD, NetBSD, OpenBSD et quelques autres.

Chapitre 4 Mémoire

Sommaire

4.1 Les problèmes à résoudre

La mémoire, terme d'un anthropomorphisme abusif hérité des « cerveaux artificiels » et auquel les Anglo-Saxons ont raison de souvent préférer *storage*, est un élément essentiel de l'architecture de von Neumann. Lorsqu'un calcul est effectué par un ordinateur, ses phases successives sont représentées par différents états de la mémoire, ce qui est la réalisation technique du ruban de la machine de Turing. Dès que le calcul se complique, l'organisation de la mémoire joue un rôle important dans l'efficacité et l'intelligibilité du programme.

En outre, dès qu'un système d'exploitation permet la présence simultanée en mémoire (et l'exécution « pseudo–simultanée ») de plusieurs programmes il faut partager entre eux la mémoire disponible et veiller à éviter que l'un n'empiète sur la zone réservée à l'autre. Les systèmes modernes permettent également d'allouer dynamiquement des zones mémoires pour y représenter des objets temporaires qui seront supprimés avant la fin du programme, ce qui permettra de rendre disponible la zone mémoire correspondante.

Enfin, encore aujourd'hui la mémoire rapide est une ressource coûteuse, aussi les ordinateurs modernes sont-ils dotés non pas d'une mémoire homogène d'un seul tenant, mais d'une hiérarchie de mémoires, en commençant par une toute petite mémoire très rapide pratiquement incorporée au circuit du processeur, puis des mémoires de plus en plus grandes et de plus en plus lentes, pour finir par un espace sur disque magnétique où sont recopiées les zones mémoire provisoirement inutilisées. La petite mémoire très rapide contient des données en cours de traitement, les grandes mémoire lentes (et bon marché) contiennent des données en attente.

Toutes ces questions, regroupées sous le titre de « gestion de la mémoire », sont l'objet des soins attentifs des architectes de processeurs et des écrivains de systèmes d'exploitation, parce que leur solution plus ou moins heureuse jouera un rôle primordial pour l'efficacité plus ou moins grande du couple processeur–système, et que la conception du matériel et celle du logiciel sont ici très étroitement intriquées. Un système de mémoire raté lors de la conception initiale d'une architecture est un des rares défauts non rattrapables.

4.2 La mémoire du programme

Nous avons dit que les états successifs de la mémoire représentent les étapes successives d'un calcul. La théorie de cette représentation, telle qu'illustrée à la grande époque des systèmes formels par Kurt Gödel, Alan Turing et Alonzo

Church, conduit à des notations d'une rigueur implacable mais d'un maniement délicat, qui réserverait la programmation des ordinateurs à une élite triée sur le volet de la mathématique. L'histoire des langages de programmation est marquée par une évolution vers l'expressivité, destinée à faciliter l'écriture comme la compréhension des programmes.

4.2.1 Les mots de mémoire

Dès von Neumann la mémoire est structurée, nous l'avons vu, en mots qui regroupent un certain nombre de bits. Un détail à ne pas oublier : tous les mots ont la même taille constante. Si l'on considère chaque bit comme un chiffre binaire et un mot de taille n comme un nombre binaire de n chiffres, nous avons l'élément de base d'une arithmétique. L'annexe A donne quelques détails sur sa réalisation concrète.

Nous avons vu que nous devons aussi représenter en mémoire bien d'autres choses que des nombres. D'abord et malgré qu'en aient les physiciens et les numériciens, toutes les données ne sont pas des nombres, il y a aussi des chaînes de caractères, du texte, des images et des sons représentés de façon codée. Et nous avons vu qu'il fallait aussi stocker les instructions, qui souvent occupent chacune un mot[1].

4.2.2 Les adresses

Il faut allouer à chacun de ces objets un ou plusieurs mots de mémoire, et ensuite pouvoir les y retrouver. Ceci suppose un système d'identification et de repérage des mots individuels, un système d'*adresses*. Beaucoup de solutions ont été essayées, mais aujourd'hui tous les concepteurs se sont ralliés à la plus simple : les mots de la mémoire centrale sont numérotés du premier au dernier et chaque mot est identifié par son numéro d'ordre qui est son adresse. Enfin, à un facteur près : pour des raisons de commodité l'unité adressable de mémoire est le plus souvent aujourd'hui un *octet* de 8 bits, le mot comportera quatre ou huit octets et les adresses de mots seront des multiples de 4 ou de 8.

Ceci semble anodin et simple, et il s'agit pourtant d'un choix technique absolument crucial. L'adresse est un numéro, donc un nombre, un nombre binaire, on l'aura deviné. Les adresses, nous l'avons vu en 2.4.1, sont manipulées par les ins-

1. Les instructions des processeurs RISC contemporains (architectures ARM et MIPS par exemple) sont de longueur fixe et occupent chacune un mot (de 32 ou 64 bits), ce qui simplifie la conception de beaucoup de composants du processeur et du système. Les grands systèmes IBM (l'architecture 360 et sa postérité) ont des instructions sur un demi-mot, un mot ou un mot et demi. Sur le processeur Itanium (architecture IA-64) trois instructions de 41 bits et un masque de 5 bits se partagent un double mot. Voir le chapitre 9 pour plus de détails.

tructions, c'est-à-dire qu'elles doivent tenir dans les registres[2]. La taille d'un registre en bits est la borne supérieure du nombre de chiffres binaires d'une adresse. Si le registre a n bits, la plus grande adresse vaudra $2^n - 1$, et donc aucun ordinateur conforme à cette architecture ne pourra avoir une capacité mémoire supérieure à 2^n octets. Cette valeur de n intervient partout dans l'architecture et dans les programmes, si on l'a prévue trop petite au départ c'est irréparable.

L'architecture 360 conçue au début des années 1960 comportait des mots, et donc des registres, de 32 bits, ce qui permettait une taille mémoire maximum de 2^{32} octets, un peu plus de 4 milliards. À l'époque cette valeur semblait énorme, et les ingénieurs limitèrent la taille des adresses à 24 bits donnant accès à une mémoire de 16 millions d'octets. En effet chaque bit d'adresse supplémentaire coûte très cher : les adresses sont transmises entre le processeur, la mémoire et les périphériques par des bus spéciaux (les bus d'adresse) qui comportent un fil par bit. La largeur du bus d'adresse pèse lourd dans le budget du processeur en termes de surface, de consommation électrique, de complexité des dispositifs, sans parler de la taille accrue des circuits logiques de toutes les instructions qui doivent manipuler des adresses.

Les concepteurs du matériel mirent en garde les concepteurs de logiciels : interdiction d'utiliser les huit bits vides qui restaient dans les mots de mémoire ou les registres qui contenaient des adresses, ils seraient précieux pour une extension ultérieure de l'architecture ! Mais à une époque où chaque bit de mémoire était utilisé avec parcimonie et à bon escient c'était un supplice de Tantale. Lorsqu'à la fin des années 1970 la limite de 16 millions d'octets se révéla une contrainte insupportable, les systèmes d'exploitation développés par IBM soi-même pour l'architecture 360 utilisaient régulièrement les huit bits de poids fort des mots d'adresse pour toutes sortes d'usages, et la conversion à l'architecture XA (pour *Extended addressing*, passage aux adresses sur 32 bits) imposa un remaniement complet de milliers de modules de logiciel, au prix d'années de travail et de millions de dollars[3]. Cette imprévoyance eut de fait des conséquences bien plus lourdes même si bien moins médiatiques que le soi-disant « bug de l'an 2000 ».

2. Notre programme d'exemple avait un jeu d'instructions autorisant la présence d'adresses directement dans les instructions, ce qui n'est plus le cas des architectures récentes, où les adresses sont manipulées dans les registres.

3. En réalité XA était un adressage sur 31 bits ; pour éviter les conflits trop graves avec les anciens programmes, on conserve les adresses traditionnelles sur 24 bits et on dégage 2^{31} octets adressables nouveaux en utilisant des adresses « négatives » (voir l'annexe A pour la représentation des nombres binaires négatifs, qui explique cet artifice). Le passage aux adresses sur 64 bits aura demandé des méthodes plus radicales...

4.2.3 Noms et variables

L'adresse, que nous venons de décrire, a deux fonctions bien différentes : elle repère la position physique d'un emplacement en mémoire, et elle permet à un programme en langage machine (ou en assembleur) d'en désigner le contenu. Dans ce dernier rôle l'adresse exerce la fonction de nom : un élément de langage (un *lexème*) qui désigne est un nom.

En langage machine ou assembleur les noms sont des adresses, mais dans des langages de plus haut niveau les noms sont des lexèmes plus abstraits, en fait plus similaires à ce que nous appelons nom dans la grammaire des langages humains. De tels noms seront généralement appelés des identifiants. Mais pour être exécuté le programme en langage de haut niveau sera traduit en langage assembleur, puis en langage machine, et ses noms (ses identifiants) se résoudront en adresses qui permettront d'accéder physiquement à la donnée.

En langage machine ou assembleur même, la donnée traitée n'est pas toujours désignée directement par une adresse simple. Un mot peut contenir l'adresse d'un autre mot qui, lui, contient l'adresse de la donnée, c'est l'adressage indirect. Il peut aussi être commode de traiter de façon itérative une série de mots adjacents comme un tableau : un registre $R1$ contiendra l'adresse du premier mot, et un registre $R2$ contiendra le rang du mot courant dans la série, éventuellement multipliée par la taille du mot en octets : ainsi l'adresse réelle du mot à traiter sera $R1 + R2$. $R1$ sera appelé registre de base et $R2$ registre index du tableau. L'art de la programmation en assembleur consiste à utiliser judicieusement les possibilités d'adressage indirect et indexé.

Dans un langage évolué, en général, on a envie de conserver des résultats intermédiaires, ou des valeurs significatives d'un calcul ou d'un traitement de quelque sorte. Ces valeurs, de quelque type (nombre, texte, image), occupent un ou plusieurs mots qui constituent un objet élémentaire. Bref, on souhaite pouvoir retrouver, à la demande, la valeur d'un objet du langage. La solution retenue consiste souvent à associer à la valeur un nom (un identifiant) de telle sorte que l'évocation ultérieure de ce nom procure un accès à la valeur. L'identifiant sera le nom de l'objet.

L'objet le plus habituel des langages de programmation évolués est la *variable*. La variable, nous l'avons vu à la section 2.8, est un objet doté des propriétés suivantes (que nous enrichissons ici) :

1. posséder un nom ;
2. posséder une valeur ;
3. le langage permet, par le nom, de connaître la valeur de la variable ;
4. une variable a une durée de vie (une persistance), qui est souvent égale à la durée pendant laquelle le programme s'exécute, mais qui peut être plus courte (variable locale à un sous-programme) et que l'on peut souhaiter

plus longue (les moyens de satisfaire ce souhait feront l'objet du chapitre suivant) ;

5. une variable a une visibilité, qui peut s'étendre à l'ensemble du programme, mais qui peut être limitée par exemple à un sous-programme ;
6. il est possible par le langage de modifier la valeur de la variable. L'opération de modification de la valeur d'une variable s'appelle l'*affectation* (notons que certains langages de haut niveau tels que ML et Haskell n'autorisent pas cette opération d'affectation).

Ne perdons pas de vue qu'en dernier recours ce que nous appelons valeur sera une configuration de bits contenue dans un ou plusieurs mots de mémoire. La valeur a donc une existence physique, elle est un élément de l'état de la mémoire.

Le nom sera en dernière analyse traduit en une adresse, l'adresse du premier mot de la région de mémoire où est stockée la valeur. Le nom n'est qu'un objet du langage, un objet symbolique destiné à disparaître au cours du processus de traduction de langage évolué en langage machine. Nous avons ainsi une chaîne de noms, depuis l'identifiant du langage évolué jusqu'à l'adresse mémoire physique en passant par d'éventuelles adresses indirectes (des noms de noms) ou indexées, qui tous mènent à la même donnée. Cette chaîne parcourt en fait les différents niveaux d'abstraction qui mènent de la description formelle d'un traitement par le texte d'un programme jusqu'à son exécution par un ordinateur matériel.

Revenons un instant sur l'affectation, qui est la propriété numéro 6 de ce que nous avons appelé variable. Si nous en étions restés aux trois propriétés précédentes, ce que nous appelons variable serait *grosso modo* la même chose que ce que les mathématiciens appellent variable. Mais l'affectation opère une rupture radicale entre vision mathématique et vision informatique du calcul, elle y introduit un aspect dynamique, actif et concret qui est étranger aux mathématiciens.

Or cette rupture, si elle est radicale, n'est susceptible ni de suture ni de réduction. L'affectation est vraiment au cœur de la problématique informatique, c'est elle qui permet de modéliser un calcul par des états de mémoire successifs, bref c'est elle qui permet de réaliser des machines de Turing.

4.2.4 Protection de la mémoire

Dès que les systèmes se compliquèrent, et surtout dès qu'il y eut plusieurs programmes en mémoire, des accidents arrivèrent. Si l'on a en mémoire le petit programme en langage machine de la section 2.4.1, il est clair qu'une erreur de programmation très banale peut envoyer une donnée à une mauvaise adresse, et écraser une donnée ou une instruction qui ne devrait pas l'être, ce qui perturbera radicalement les traitements ultérieurs. Si encore le programme erroné détruit

ses propres données ou son propre texte, l'auteur n'a qu'à s'en prendre à lui-même, mais s'il s'agit de celui d'un collègue innocent c'est trop injuste. Et si ce sont les données ou le texte du système, alors la catastrophe est générale.

Les premiers systèmes de multiprogrammation abordaient ce problème par le côté du matériel. L'architecture 360 découpe (elle existe encore...) la mémoire en groupes de 2 048 octets qui constituent l'unité non partageable (le quantum) d'allocation à un processus donné. Chacun de ces groupes possède une clé physique de quatre chiffres binaires pouvant donc prendre 16 valeurs. Par ailleurs le PSW comporte un champ de quatre bits qui contient la clé attribuée à son démarrage au processus courant. Lors de tout accès mémoire, le processeur vérifie que la clé de la zone concernée est bien égale à la clé du processus courant, sinon le processeur provoque une erreur fatale et le processus concerné est interrompu. Bien sûr le système d'exploitation qui s'exécute en mode privilégié bénéficie d'une clé particulière, la clé zéro, qui lui donne accès à toute la mémoire.

Ce système est un peu rudimentaire : il n'autorise que quinze processus concomitants et le système. Le développement des systèmes à mémoire virtuelle permettra des dispositifs bien plus raffinés.

4.3 Partage de mémoire en multiprogrammation

Les sections 3.2 et 3.8 ont décrit les principes de la multiprogrammation et ses conséquences dans le domaine de la mémoire ; il faut maintenant préciser comment se fait l'attribution de la mémoire à chacun des programmes concomitants pseudo-simultanés.

4.3.1 Exemple : l'OS/360

Les premiers systèmes d'exploitation destinés à l'architecture IBM 360, au milieu des années 1960, procédaient de façon statique : une zone fixe en mémoire était attribuée à chaque programme à son démarrage et il la gardait jusqu'à la fin de son exécution. Il existait deux variantes :

– Avec l'OS/MFT *(Multiprogramming with a fixed number of tasks)*, la mémoire était découpée en partitions fixes au démarrage du système. Il appartenait à l'administrateur du système de fixer judicieusement le nombre de partitions et la taille de chacune d'entre elles. Les travaux étaient répartis en classes (en fait des files d'attente) et chaque classe était éligible pour une ou plusieurs partitions, principalement en fonction de la taille mémoire nécessaire à l'exécution des travaux de la classe. On pouvait ainsi prévoir plusieurs petites partitions pour petits travaux et une grande partition pour les gros. Le modèle de file d'attente pour l'exécution des travaux est laissé en exercice au lecteur.

L'inconvénient de ce système est facile à imaginer : les partitions sont des lits de Procuste. Leur taille est toujours plus ou moins arbitraire et tous les programmes ne peuvent pas s'y adapter exactement. De surcroît les files d'attente de certaines classes peuvent être vides alors que d'autres peuvent être embouteillées. Tout ceci entraîne un risque de mauvaise utilisation de la mémoire.

– La version plus perfectionnée, OS/MVT *(Multiprogramming with a variable number of tasks)*, abolit le découpage rigide de la mémoire. Chaque programme annonce la taille de la région de mémoire nécessaire à son exécution et dès qu'elle est disponible elle lui est attribuée, toujours une fois pour toutes et jusqu'à la fin de son exécution. Détail important : la région de mémoire ainsi allouée est contiguë.

  OS/MVT est beaucoup plus complexe mais beaucoup plus satisfaisant qu'OS/MFT. Néanmoins on voit bien que ce n'est pas vraiment satisfaisant : les programmes ont des tailles et des durées d'exécution arbitraires. Lorsqu'un programme se termine il libère une région de mémoire qui laisse un trou à une adresse arbitraire au milieu de la mémoire. Rien n'assure que parmi les programmes en file d'attente il y en aura un qui logera dans ce trou. Il peut ainsi se créer un effet « fromage de gruyère », où plusieurs trous de mémoire atteignent une taille cumulée suffisante pour les travaux en attente, mais comme chacun est trop petit, la file d'attente est bloquée.

  Ce défaut de la gestion de mémoire de l'OS/360 sera corrigé par l'introduction de la mémoire virtuelle que nous étudierons à la section suivante, mais auparavant il nous faut compléter ce qui vient d'être dit en expliquant la translation des programmes, sans quoi nous ne pourrions avoir plusieurs programmes en mémoire simultanément.

4.3.2 Translation des programmes

Oui il s'agit bien de translation, puisqu'aujourd'hui il faut préciser en employant ce mot que l'on n'est pas en train de faire une erreur de traduction (justement) de l'anglais *translation* qui signifie traduction en français[4]. Nous avons

4. Je voudrais en profiter pour réhabiliter aussi consistant, qui *n'est pas* la traduction française de *consistent,* laquelle est « cohérent ». L'inconvénient de ces confusions est la perte de signifiés. La consistance et la translation sont des concepts intéressants et utiles, menacés de disparition par extinction (usurpation ?) de leurs signifiants. Si aujourd'hui dans un milieu mathématique vous vous risquez à ne pas employer consistant où il faudrait cohérent, au mieux vous ne vous ferez pas comprendre, et vous serez soupçonné de ne pas être un habitué des conférences anglo-saxonnes. J'ai dressé un répertoire (incomplet) de ces faux-amis ici : `https://laurentbloch.net/MySpip3/Faux-amis-Deceptive-Cognates`

donc parlé plus haut de la traduction des programmes, ici c'est de leur translation qu'il s'agit.

Le lecteur vigilant, à la lecture des alinéas précédents, aura dû se demander comment il est possible de charger des programmes en mémoire à des adresses somme toute imprévisibles sans en perturber la fonctionnement? Lors de nos essais en langage machine nos programmes comportaient des adresses qui désignaient sans ambiguïté un mot de mémoire bien déterminé où devait se trouver une instruction ou une donnée indispensable au bon déroulement des opérations. *Locus regit actum*...

Si maintenant on nous dit que le programme va être chargé à une adresse qui ne sera plus l'adresse `0`, mais une position quelconque et imprévue en plein milieu de la mémoire, tout le texte va être décalé, et chaque objet désigné par le programme aura une adresse qui ne sera plus l'adresse absolue à partir de `0` que nous avions écrite, mais cette adresse additionnée de l'adresse de chargement du programme. Comment est-ce possible? Nous avons déjà abordé cette question à la section 2.6, nous allons y revenir ici.

Lorsqu'on programme en assembleur, on écrit rarement des adresses absolues. Les assembleurs allègent la tâche du programmeur en lui permettant de désigner la position d'un octet dans le texte du programme par un nom symbolique, que l'assembleur se chargera de traduire en adresse. Nous avons vu un exemple d'usage de nom symbolique dans le programme assembleur de la section 2.6, avec le traitement de l'étiquette `FIN`, qui désignait l'instruction située à l'adresse absolue `5`. Un tel symbole peut de la même façon désigner le premier octet d'une zone de mémoire qui contient une donnée.

Nous voulons maintenant que ce symbole ne désigne plus l'adresse absolue `5`, mais une adresse relative par rapport au début du programme, lequel ne serait plus nécessairement à l'adresse `0`. Pour ce faire, notre assembleur devra opérer une traduction un peu plus complexe; chaque symbole sera traduit en adresse selon le schéma suivant: l'adresse sera exprimée comme la somme de deux termes:

- un déplacement par rapport au début du programme, égal à l'adresse que nous avions écrite de façon absolue;
- une valeur dite de *base*, qui correspondra à l'adresse de chargement du programme, et sera contenue dans un registre, dit *registre de base*.

Pendant l'assemblage proprement dit, le registre de base recevra la valeur `0`. Au moment de l'exécution, il recevra l'adresse du point de chargement du programme en mémoire, et ainsi nos adresses exprimées sous la forme base + déplacement seront juste. Le mécanisme qui place dans le registre de base l'adresse de début du programme varie selon les systèmes, il est parfois à la charge du système d'exploitation, d'autres réalisations laissent cette action à

la charge du programmeur qui dispose d'instructions capables de l'effectuer (c'est le cas de l'OS/360). Enfin, n'oublions pas que lorsque que nous disons programmeur il faut la plupart du temps entendre compilateur, parce que c'est lui qui va traduire en assembleur les programmes écrits en langage évolué par des humains qui n'auront pas eu à se soucier de ces contingences techniques pourtant passionnantes.

Ce perfectionnement de l'assembleur n'est pas très spectaculaire, mais sans lui la multiprogrammation serait plus complexe à réaliser. Les programmes assemblés selon ce principe avec des adresses sous forme de base + déplacement sont appelés des programmes translatables. La translation des programmes est parfois aussi nommée réimplantation . Nous retrouverons un autre usage de cette propriété lorsque nous parlerons de l'édition de liens, qui permet de réunir plusieurs modules de programmes compilés indépendamment (c'est-à-dire éventuellement écrits dans des langages différents) pour constituer un seul programme.

4.4 Mémoire virtuelle

4.4.1 Insuffisance de la mémoire statique

Nous avons vu ci-dessus que l'allocation à un programme de la zone mémoire dont il avait besoin, de façon fixe une fois pour toutes au début de son exécution, avait un inconvénient qui était la fragmentation de la mémoire au fur et à mesure du lancement à des instants aléatoires de programmes de tailles hétérogènes. Ainsi peut être libre une zone de mémoire de taille suffisante pour lancer un programme sans que le lancement soit possible, parce que cette zone libre est constituée de fragments disjoints.

Certains systèmes des années 1960 (Univac, Control Data) palliaient cette inefficacité en réorganisant périodiquement la mémoire pour « tasser » les zones utilisées les unes contre les autres. Mais une solution bien plus radicale allait advenir : la mémoire virtuelle. La voici.

4.4.2 Organisation générale

L'organisation de mémoire virtuelle que nous allons décrire est inspirée de celle des systèmes IBM 370 et suivants, mais les autres réalisations sont assez comparables. Il y a eu des organisations différentes, mais celle-ci, qui est la plus simple, s'est aussi révélée la plus efficace, ce qui a assuré son succès général.

Il existe dans cette organisation trois états de la mémoire : virtuelle, réelle, auxiliaire. La mémoire virtuelle est celle que les programmes utilisent, mais elle n'existe pas vraiment. Un emplacement de la mémoire virtuelle peut avoir ou ne pas avoir de contenu ; s'il n'a pas de contenu il est simplement virtuel et le restera

jusqu'à ce que le programme décide d'y placer un contenu ; s'il a un contenu il faut bien que celui-ci existe quelque part, et ce quelque part sera soit la mémoire réelle, soit une zone buffer sur disque appelée mémoire auxiliaire.

La mémoire virtuelle répond à deux préoccupations. La première vise à éviter le gaspillage de fragments de mémoire en permettant à la mémoire linéaire vue par le programme d'être physiquement constituée de fragments disjoints, ce qui supprime l'inconvénient de la fragmentation de la mémoire, au prix d'un mécanisme que nous allons étudier pour rétablir la fiction de la linéarité[5].

Localité des traitements

La seconde préoccupation répond à un phénomène appelé localité des traitements. Si nous observons, cycle par cycle, le déroulement d'un programme dont la taille mémoire est par exemple de un million d'octets, nous constaterons que pendant une tranche de temps donnée, brève par rapport au temps d'exécution total, il ne fait référence qu'à un petit nombre d'adresses proches les unes des autres. Ceci veut dire qu'à chaque instant le programme a besoin de beaucoup moins de mémoire qu'il ne lui en faut au total, et que le contenu de la mémoire inutile à un instant donné pourrait être stocké provisoirement dans un endroit moins coûteux, par exemple sur disque.

Lorsque le système d'exploitation lance l'exécution d'un programme, il lui alloue un espace de mémoire virtuelle. Comme cette mémoire est virtuelle, donc gratuite ou presque, l'espace alloué est aussi vaste que l'on veut, dans les limites de l'adressage possible avec la taille de mot disponible.

Cet espace de mémoire virtuelle est découpé en *pages* de taille fixe (souvent $2^{12} = 4\,096$ octets, pour fixer les idées) et décrit par une *table des pages*. En fait, pour éviter d'avoir une seule grande table incommode à manipuler on aura généralement une table à plusieurs niveaux, avec une table de segments au niveau le plus élevé, un segment comprenant par exemple 32 pages et chaque segment ayant une petite table de pages, mais l'idée est la même.

Les emplacements en mémoire virtuelle sont désignés par des *adresses virtuelles*, qui ressemblent comme des sœurs aux adresses réelles que nous avons vues jusqu'alors. Les adresses multiples de $4\,096$ (selon notre exemple) sont des frontières de pages, et les multiples de $32 * 4\,096 = 131\,072$ des frontières de segments, mais cela ne change pas grand-chose. La table des pages aura une entrée par page, indexée par l'adresse virtuelle de la page, qui sera le numéro d'ordre de

5. Malgré les avantages qu'elle apporte, la mémoire virtuelle n'a pas eu que des aficionados. Seymour Cray, qui fut le concepteur des ordinateurs les plus puissants du XX^{e} siècle, de 1957 à 1972 pour *Control Data Corporation* , puis jusqu'à sa mort en 1996 pour *Cray Research*, a dit : « La mémoire, c'est comme le sexe : c'est meilleur quand c'est réel. »

l'octet frontière de page dans l'espace virtuel, soit un multiple de $4\,096$, c'est-à-dire un nombre binaire se terminant par douze 0.

4.4.3 Pagination

Que dit la table des pages ? D'abord, pour chaque page virtuelle elle indique où elle se trouve réellement. Selon les trois états de mémoire évoqués ci-dessus, il y a trois situations possibles pour une page virtuelle :

1. elle peut avoir une incarnation en mémoire réelle, physique, et la table indique alors l'adresse réelle de cette incarnation ; la zone de mémoire réelle qui accueille une page de mémoire virtuelle est appelée *cadre de page (page frame) ;*
2. si elle correspond à une adresse qui n'a encore jamais été invoquée par le programme, elle reste purement virtuelle et elle n'existe physiquement nulle part, son existence est limitée à une entrée vierge dans la table des pages ;
3. si cette page a été utilisée à un moment donné mais que l'exécution du programme ne nécessitait pas son usage à cet instant et qu'il n'y avait pas assez de place en mémoire centrale, elle peut avoir été placée sur disque dans ce que l'on appellera *mémoire auxiliaire de pages,* et la table indiquera son emplacement dans cette mémoire auxiliaire.

MMU *(Memory Management Unit)*

Comment fonctionne la mémoire virtuelle ? Les programmes ne connaissent que la mémoire virtuelle, et des adresses virtuelles. Chaque fois qu'une instruction fait référence à une donnée, cette référence est une adresse virtuelle. Il faut donc traduire à la volée l'adresse virtuelle en adresse réelle : l'obtention d'une vitesse raisonnable impose un circuit logique particulier à cet effet, appelé DAT *(Dynamic Address Translation)*. Avec les autres fonctions de gestion de la mémoire virtuelle que nous allons décrire il constitue la MMU *(Memory Management Unit)*.

Le DAT fonctionne de la façon suivante, illustrée par la figure 4.1 p. 81. L'adresse (24 bits dans notre exemple) est découpée en trois parties : les 12 derniers bits (si nous poursuivons notre exemple avec des pages de taille 2^{12}), dits bits de poids faible, sont considérés comme une adresse relative par rapport à la frontière de page précédente, soit un déplacement dans la page. Les 5 bits de poids le plus fort sont un numéro de segment, index dans la table de segments du processus, qui permet de trouver la table des pages du segment. Les 7 bits de poids intermédiaire sont un numéro de page, index dans la table des pages qui permet de trouver la page concernée. La partition des 12 bits de poids fort en

numéro de segment et numéro de page n'est qu'un artifice pour hiérarchiser la table des pages, ils peuvent être considérés globalement comme le numéro de page.

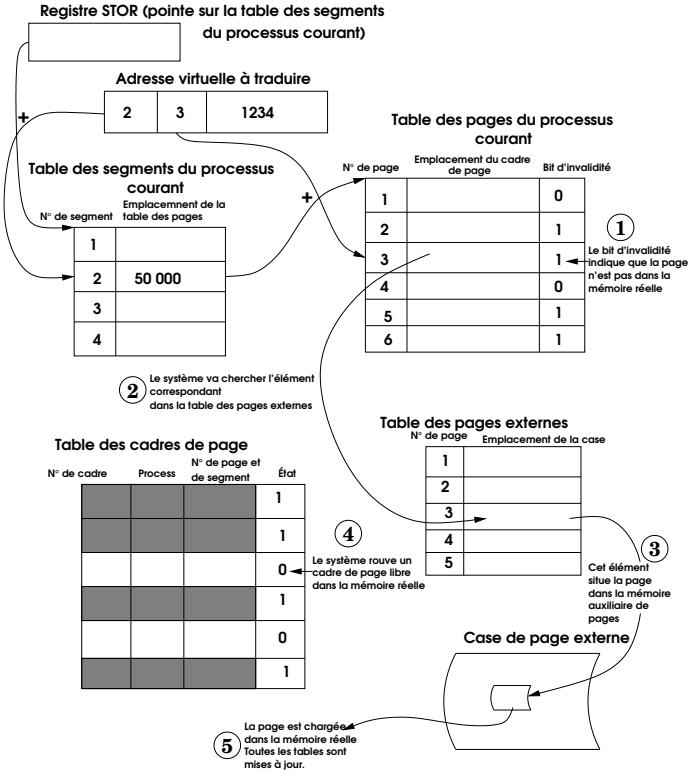


Figure 4.1 : *Pagination : cas où la page demandée n'est plus en mémoire réelle* (exemple de l'OS/370)

Le DAT consulte la table des pages pour y chercher l'entrée correspondant au numéro de page virtuelle voulu. Selon les trois cas énumérés ci-dessus trois situations peuvent se présenter :

– Dans le cas 1 de la liste ci-dessus la page existe et possède une incarnation en mémoire réelle. Le circuit de traduction d'adresse trouve dans la table des pages l'adresse du cadre de page qui contient la page virtuelle. Cette adresse est additionnée au déplacement dans la page, ce qui donne l'adresse réelle à laquelle le programme doit accéder.
– Voyons ensuite le cas 2 : l'adresse évoquée par le programme n'a encore fait l'objet d'aucune référence, non plus qu'aucune adresse de la même page. Cette situation peut se produire au lancement d'un programme, ou

après l'allocation d'une zone de mémoire dynamique vierge, par exemple. Il va falloir obtenir du système une page réelle « neuve » et placer son adresse dans la table des pages, ainsi nous serons ramenés au problème précédent, après la consommation d'un nombre non négligeable de cycles de processeur. La MMU génère une exception dite de *défaut de page* (la section 3.12.2 donne la définition des exceptions), et le gestionnaire de défaut de page va être chargé d'obtenir une page réelle neuve.
Comment obtient-on une page réelle neuve ? Le système gère une table des cadres de pages en mémoire réelle, cette table garde pour chaque cadre de page un certain nombre d'informations, et notamment s'il est libre. Deux cas peuvent se présenter :

a. Si le parcours de la table des cadres de pages révèle un cadre libre il est alloué à la page vierge, son adresse est placée dans la table des pages et le programme peut poursuivre son exécution.
b. S'il n'y a aucun cadre de page libre, il faut en libérer un. En première approximation nous dirons que le système d'exploitation procède ainsi (nous verrons plus tard quelques raffinements techniques qui améliorent les performances sans bouleverser le schéma de principe) : chaque entrée dans la table des cadres de pages comporte une estampille qui indique la date du dernier accès d'un programme à une adresse comprise dans la page qui réside dans ce cadre. L'estampille de plus faible valeur désigne le cadre de la page la moins récemment utilisée. Cette page est donc candidate à l'éviction : son contenu est recopié en mémoire auxiliaire de pages, son cadre libéré et alloué à notre page vierge. La table des pages est mise à jour et le programme continue comme dans le cas précédent.
Cet algorithme qui consiste à évincer la page la moins récemment utilisée est appelé LRU *(Least recently used)* et nous le retrouverons pour d'autres usages.

– Voyons enfin le cas 3 : la page a été utilisée, elle a un contenu, mais elle ne réside plus en mémoire réelle, elle a été déplacée sur disque en mémoire auxiliaire de page. Cette condition déclenche comme la précédente une exception de défaut de page et le transfert du contrôle au gestionnaire de défaut de page. Il faut pour pouvoir utiliser la page la ramener en mémoire réelle, et pour cela il faut qu'un cadre de page soit libre, ou en libérer un : ceci est fait selon le même mécanisme qu'à l'alinéa ci-dessus. Une fois le cadre de page obtenu, le contenu de la page virtuelle qui était en mémoire auxiliaire est recopié dans ce cadre, la table des pages est mise à jour et l'exécution du programme peut continuer. C'est le mécanisme illustré par la figure 4.1.

Une vision de la pagination

La technique de pagination a suscité une floraison de métaphores explicatives, qui aident à comprendre la question. Celle que je cite ici m'a été fournie par mon collègue Henri Leridon :

« Finalement, tout ça me semble relever du problème du garçon de plage sur la Côte d'Azur au mois d'août. Je m'explique. Le plagiste doit gérer – au mieux – un espace limité, avec des clients qui vont et viennent en exprimant des besoins (de surface au sol : on les laisse se débrouiller dans l'eau) variés. On peut compliquer un peu en supposant que les premiers arrivés d'une même famille ne savent pas à l'avance combien ils seront au total, ni à quelle heure arrivera le reste de la famille, mais qu'ils voudront être tous ensemble. On pourrait aussi admettre qu'il y a deux plagistes, situés chacun à une extrémité de la plage et gérant plus ou moins le même espace. Et pourquoi ne pas admettre que les clients seraient en droit de changer de place (pour se rapprocher du bar, par exemple), les plagistes devant alors s'évertuer à conserver leur adresse ? »

4.4.4 Espaces adresse

Les premiers systèmes à mémoire virtuelle (chez IBM par exemple OS/VS1 et VS2 en 1972) allouaient pour l'ensemble du système un espace unique de mémoire virtuelle dont la taille était fixée par la valeur maximum d'une adresse. Cet espace virtuel était partagé entre les programmes selon les mêmes principes que dans les anciens systèmes sans mémoire virtuelle, tels que nous les avons exposés au début de ce chapitre. Ceci présentait l'avantage que l'adjonction de la mémoire virtuelle modifiait assez peu le système d'exploitation.

Cependant le caractère virtuel de la mémoire allouée permet d'imaginer d'autres solutions, notamment allouer à chaque programme un espace de mémoire virtuelle entier. Pour réaliser un tel système il faut donner à chaque programme une table des pages particulière, qui décrit tout l'espace adressable. Quand le système d'exploitation donnera la main à un autre programme il changera également de table des pages. Ce sera chez IBM le système MVS *(Multiple Virtual Storage)* en 1974, chez Digital Equipment VMS en 1977, chez Data General AOS/VS... Une mémoire virtuelle à espaces adresse multiples est illustrée par la figure 4.2.

Il ne devrait pas échapper au lecteur attentif une conséquence importante de cette multiplication des espaces de mémoire virtuelle (ou espaces adresse) : une adresse virtuelle donnée est désormais traduite dans le contexte d'exécution d'un programme donné, la même adresse dans le contexte d'un autre programme est traduite à partir d'une autre table des pages, et correspond donc à une autre adresse réelle. Il est de ce fait impossible à un programme de faire référence à une adresse qui appartient à une page de mémoire virtuelle d'un autre

programme. La mémoire virtuelle à espaces adresse multiples améliore la sécurité des données.

Autre conséquence : dans les systèmes à espace adresse unique (virtuel ou non) l'adresse de chargement d'un programme dépend de l'emplacement de la zone de mémoire libre que le système a pu lui allouer, elle est donc variable, d'où les techniques de translation exposées à la section 4.3.2. Maintenant que chaque programme a son espace privé, rien n'interdit de le charger toujours à la même adresse, et d'avoir une carte de la mémoire identique d'une exécution à l'autre.

Cela dit, l'usage de cette facilité s'est rapidement heurtée à un inconvénient majeur : si on peut toujours retrouver les mêmes structures de données aux mêmes adresses dans la mémoire, cela facilite grandement le travail de ceux qui doivent mettre au point et déboguer le système, mais aussi celui des attaquants et autres pirates ! Après quelques événements dommageables, les architectes de système se résolurent à « placer de façon aléatoire les zones de données dans la mémoire virtuelle. Il s'agit en général de la position du tas, de la pile et des bibliothèques. Ce procédé permet de limiter les effets des attaques de type dépassement de zone mémoire *(buffer overflow)* par exemple [6]. » Cette technique se nomme « distribution aléatoire de l'espace d'adressage » (*address space layout randomization*, ASLR).

D'autre part, les techniques de translation conservent leur utilité pour pouvoir lier ensemble des programmes écrits séparément, elles ont même trouvé un surcroît d'utilité avec les techniques de processus légers connus sous le nom d'activités *(threads)* : la multi-activité *(multithreading)* consiste à faire exécuter plusieurs parties de programmes en pseudo–simultanéité dans le même espace adresse (voir section 10.6). L'instabilité notoire de certains programmes qui reposent sur cette technique, au premier rang desquels les navigateurs du Web, découle peut-être de cette promiscuité en mémoire, qui a par ailleurs des avantages : il est commode de pouvoir afficher plusieurs fenêtres du navigateur à l'écran, et de consulter une page dans une fenêtre cependant qu'une autre se charge dans une fenêtre différente et qu'un transfert de fichier a lieu dans une troisième. Vous ne le saviez peut-être pas mais c'est de la multi-activité.

Que chaque programme s'exécute dans son espace privé inaccessible aux autres parce que tout simplement aucun nom n'est disponible pour en désigner les emplacements, très bien, mais il y a quand même des choses à partager. Notamment le système d'exploitation, qui après tout est lui aussi un programme, avec des sous-programmes qui doivent pouvoir être appelés par les programmes ordinaires. Le système d'exploitation comporte aussi de nombreuses tables qui

6. Cf. Wikipédia, `https://fr.wikipedia.org/wiki/Address_space_layout_randomization`

contiennent des informations relatives à l'état de l'ordinateur et du système, comme par exemple le fuseau horaire de référence, des moyens d'accès aux données sur disque ou au réseau, etc. Comment y accéder ?

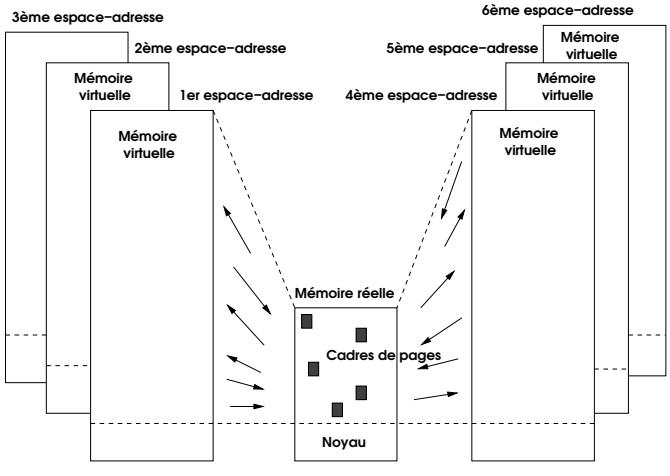


Figure 4.2 : *Mémoire virtuelle à espaces adresse multiples*

La solution imaginée par les auteurs de VMS est la suivante : les adresses sont sur 32 bits, ce qui autorise des espaces adresse de 2^{32} octets, soit plus de quatre milliards ($4\ \ 294\,967\,296$). Les 2^{31} octets qui constituent la première moitié de cet espace (avec des adresses dont le bit de poids le plus fort est 0) sont dévolus au programme et à ses données. Les 2^{31} octets suivants (avec des adresses dont le bit de poids le plus fort est 1) sont dévolus au système d'exploitation, c'est-à-dire que ces pages sont décrites par la table des pages du système. Naturellement pour tous les programmes la table des pages du système est la même, c'est-à-dire que tous les programmes voient le même système d'exploitation avec les mêmes adresses virtuelles, et n'y ont bien sûr droit qu'à un accès en lecture mais pas en modification. On dit que le système d'exploitation est *mappé* (de l'anglais *mapped*) dans l'espace adresse du programme, les adresses virtuelles des pages du système sont superposées à des adresses virtuelles de l'espace de chaque programme.

Les auteurs de Unix 4.4 BSD ont eu recours à un découpage analogue, mais ils ont été moins généreux pour le système d'exploitation (le noyau) et le programme reçoit la plus grande part de l'espace adresse. Il est vrai que 4.4 BSD était moins volumineux que VMS.

Après cet éloge des systèmes à espaces adresse multiples, il convient de signaler que les processeurs 64 bits semblent remettre à la mode l'espace adresse unique. En effet l'espace offert par une telle taille d'adresse est suffisant pour les usages actuels, et la conception du système s'en trouverait simplifiée.

L'architecture IA-64 (Itanium) prévoit le support des deux types de gestion de mémoire virtuelle, et les manuels Intel présentent l'espace adresse unique comme la voie de l'avenir… mais en 2018 l'Itanium fait figure d'espèce en voie d'extinction.

4.4.5 Registres associatifs (*Translation Lookaside Buffer*, TLB)

Tout ceci fonctionne, mais il est difficile de ne pas se poser la question suivante : si chaque accès à la mémoire entraîne une traduction d'adresse, et que celle-ci entraîne la consultation d'une table des pages, cela risque de consommer un temps considérable, même avec une table hiérarchisée et un circuit spécial. Un espace adresse de 2^{32} octets (adresses sur 32 bits) découpé en pages de $4\,096$ octets aura une table des pages avec un million d'entrées, alors ne parlons pas d'adresses sur 64 bits qui nous entraîneraient vers une table à 2^{52} entrées…

En fait, un système de mémoire tel que celui décrit jusqu'ici serait d'une lenteur exécrable. Pour en accroître la vitesse les MMU réelles ont recours à un dispositif qui ne change rien aux grands principes de fonctionnement mais qui procure une accélération spectaculaire : le *buffer*[7] *de traduction anticipée (Translation Lookaside Buffer, TLB)*.

L'idée est de tirer parti une seconde fois de la localité des traitements (cf. section 4.4.2). Puisqu'un programme à un moment donné ne fait référence qu'à un petit nombre d'adresses proches les unes des autres (c'est un fait d'observation générale), c'est qu'il utilise (pendant ce laps de temps) toujours les mêmes pages. Il serait donc judicieux de garder sous la main, c'est-à-dire dans quelques registres implantés sur le circuit du processeur, et de ce fait d'un accès beaucoup plus rapide que la mémoire, le résultat des traductions les plus récentes, soit une table de correspondance numéro de page virtuelle – numéro de cadre de page pour ces quelques pages utilisées.

Comment déterminer les pages privilégiées dont le cadre de page de résidence figurera dans le TLB ? Ce seront les pages les plus récemment utilisées. À chaque référence à la mémoire, le MMU déclenche en parallèle deux méthodes de traduction d'adresse : consulter le TLB pour voir si la page cherchée n'y figure pas, activer le DAT pour parcourir la table des pages. Si la consultation du TLB réussit, le DAT beaucoup plus lent est arrêté. Si elle échoue le DAT poursuit la traduction jusqu'à son terme et en place le résultat dans le TLB pour la prochaine fois.

Où le DAT va-t-il placer le résultat de la traduction qu'il vient d'effectuer ? À la place d'une autre entrée.

7. J'ai renoncé à traduire en français le terme anglais *buffer*, qui désigne une zone de de travail en mémoire. On propose parfois *tampon*, qui n'évoque guère que des wagons, Emmanuel Saint-James me propose *buffet*, bien trouvé mais dont l'acceptation demanderait un militantisme assidu.

Pour être efficace, le TLB n'a pas besoin d'être très grand : en fait, une taille étonnamment petite suffit, en général 64 entrées. Généralement, les systèmes à espaces adresse multiples évoqués à la section précédent 4.4.4 mettent le TLB en échec, et chaque commutation de contexte, qui entraîne un changement d'espace adresse, nécessite la remise à zéro du TLB. Le plus surprenant est que le TLB reste néanmoins efficace. Signalons que certains processeurs, tel le MIPS R4000, utilisent un TLB étiqueté *(tagged TLB)*, c'est-à-dire que chaque entrée de TLB comporte l'identifiant de l'espace adresse auquel elle appartient, ce qui évite la pénalité que nous venons d'évoquer.

Ce dispositif est si efficace et résout une si grande proportion des traductions d'adresses (plus de 99% !) que certains concepteurs se sont dit que le DAT servait très rarement et qu'il suffisait de le réaliser en logiciel, ce qui est beaucoup plus lent mais plus simple et moins coûteux, et libère des nanomètres–carrés précieux sur le circuit. Les premiers architectes à risquer cette audace furent ceux des processeurs MIPS, et comme les résultats furent excellents ceux des SPARC, des Alpha et des HP PA leur emboîtèrent le pas. Sur ces processeurs le seul matériel spécialisé pour la mémoire virtuelle est le TLB et sa logique de consultation.

Nous retrouverons d'autres dispositifs d'optimisation bâtis sur le même principe : un dispositif rapide (et donc cher) pour traiter une petite quantité de cas très fréquents, un dispositif plus lent pour traiter la grande masse des cas à faible occurrence. C'est notamment sur ce principe que reposent les dispositifs de cache, qui constituent la hiérarchie de mémoire, et que nous verrons bientôt. Ce qui est frustrant, c'est qu'aucune modélisation mathématique ne rend compte à ce jour des optimisations considérables qu'ils procurent.

4.4.6 Tables de pages inverses

Nous venons de voir qu'avec l'avènement des adresses sur 64 bits, et donc des espaces adresse de 2^{64} octets, il nous faudrait des tables de pages avec 2^{52} entrées, soit, avec huit octets par entrée, 30 millions de gibioctets[8]. C'est impossible aujourd'hui et pour encore pas mal de temps. Une solution, retenue

8. La taille de la mémoire, principale ou auxiliaire, est exprimée en multiples de $1\,024$ octets naguère nommés k, pour kilo-octet, avec des multiples comme le méga-octet, le giga-octet, etc. Cette dénomination avait l'inconvénient de créer une confusion avec les multiples de $1\,000$ utilisés dans le cadre du Système International. Une norme internationale promulguée par la Commission Internationale d'Électrotechnique (IEC) y a mis bon ordre en décembre 1998. Désormais k désigne 10^3 tandis que Ki (le « kibi ») désigne $2^{10} = 1\,024$, Mi (le mébi) désigne $2^{20} = 1\,048\,576$, Gi (le gibi) $2^{30} = 1\,073\,741\,824$. Si l'on compte en bits on aura un *kibibit* ($1\,024$ bits, 1 Kib), un *mébibit* (1 Mib) ($1\,048\,576$ bits). Ces préfixes s'appliquent aux octets, soit en anglais où octet se dit *byte* et où un *mebibyte* (1 MiB) vaut $1\,048\,576$ octets, soit en français où un *gibioctet* (1 Gio) vaut $1\,073\,741\,824$ octets. Ces nombres bizarres sont des puissances de 2. On consultera à ce sujet avec profit (mais sans trop se faire d'illusions sur l'adoption de cette norme) le site `http://physics.nist.gov/cuu/Units/binary.html`.

sur les premières versions de l'Alpha et de l'Itanium, est de réduire arbitrairement mais radicalement la taille mémoire adressable en limitant le nombre de lignes du bus d'adresses, mais ce n'est guère satisfaisant.

Une solution à ce problème, dite des tables inverses, est la suivante : au lieu d'avoir une table des pages avec une entrée par page virtuelle, on a juste une table des cadres de page qui contient pour chaque cadre la référence de la page virtuelle qu'il contient, cette référence comportant le numéro de page virtuelle et l'identifiant du programme propriétaire de l'espace adresse. L'avantage de la table des cadres de pages, c'est qu'elle est beaucoup plus petite, et par définition dans un rapport de taille avec la mémoire réelle de l'ordre de un pour mille.

L'on a ainsi une table qui donne la correspondance cadre de page physique – page de mémoire virtuelle, mais en général on cherche à faire la conversion en sens inverse, et avec une telle table cela risque d'être très laborieux : la seule solution consiste à examiner une par une en séquence les entrées de la table, en espérant trouver notre page plutôt vers le début. Cela semble désespérant, mais nous allons être sauvés. Par quoi ? par le TLB, pardi. N'avons-nous pas vu, à la section 4.4.5 il y a un instant, qu'il fournissait des traductions virtuel – réel avec une efficacité étonnante et sans table des pages ? Alors voilà…

Reste les cas résiduels des références non résolues par le TLB : leur traitement doit être traité par le logiciel (système d'exploitation), mais ces cas sont si rares que cela reste acceptable. Des méthodes telles que les tables associatives *(hash tables)* sont de nature à diminuer la pénalité qui en résulte.

4.4.7 Mémoire virtuelle segmentée

Les systèmes de mémoire virtuelle que nous avons décrits jusqu'ici utilisent des espaces adresse uniformes, c'est-à-dire découpés en pages de tailles identiques, elles-mêmes regoupées par simple raison de commodité de manipulation en segments de tailles identiques. Certaines régions de la mémoire virtuelle pourront être dévolues à un usage particulier, mais ceci ne se reflète pas dans la structure de l'espace adresse, qui reste uniforme, et plus précisément linéaire : les adresses se succèdent comme la suite des nombres entiers, avec des frontières de page tous les $4\,096$ octets et des frontières de segment tous les $32 * 4\,096 = 131\,072$ octets, par exemple.

D'autres architectures de mémoire virtuelle ont été imaginées, avec des segments de tailles variables, adaptés à des régions particulières du programme, par exemple un segment pour le code du programme, un segment pour les données initialisées au lancement du programme, un segment pour les données non initialisées, un segment de données non modifiables, un segment pour les structures de données créées par le système pour ce processus, etc. De tels systèmes conservent en général une taille de page fixe, mais le nombre de pages d'un seg-

ment est arbitraire. La gestion est plus complexe, puisque les tables de pages deviennent elles aussi de taille arbitraire.

L'archétype de ces systèmes à mémoire virtuelle segmentée est Multics (voir chapitre 8). Les Unix modernes tels que Linux recourent à la notion de segment, mais elle est alors orthogonale à la notion de page : l'espace adresse d'un processus est partagé en segments (un pour le code du noyau, un pour les données du noyau, un pour le code utilisateur, un pour les données utilisateur, un segment d'état de tâche par processus *(TSS, Task State Segment),* un segment pour la table de descripteurs de segments). Par ailleurs ces segments sont paginés ; l'impression est qu'ils ne servent pas à grand'chose.

4.4.8 Petite chronologie de la mémoire virtuelle

La première réalisation de mémoire virtuelle avec pagination date de 1961 et figure à l'actif de l'Université de Manchester, qui développait en collaboration avec le constructeur le système de l'ordinateur ATLAS de Ferranti. Le système de l'ATLAS était précurseur dans bien des domaines et surtout il fut l'objet de publications très complètes.

En 1962 Burroughs (devenu depuis sa fusion avec Univac Unisys) lançait son modèle B 5000, qui apportait les innovations suivantes :

- utilisation de piles[9] pour gérer les données locales des processus ; l'architecture de la machine comportait des instructions de gestion de pile ;
- mémoire virtuelle segmentée ;
- allocation dynamique de mémoire par demande de segment ;
- système écrit en langage évolué (Algol 60).

1961 vit les débuts du projet MAC dirigé par Fernando Corbató au MIT, et dans ce cadre la réalisation du système CTSS *(Compatible Time Sharing System)*, ancêtre de Multics, sur ordinateur IBM 709 puis 7094. CTSS comportait un système de *swap*, qui recopiait sur mémoire auxiliaire les programmes mis en attente par le système au profit de programmes réactivés et rechargés depuis la mémoire auxiliaire vers la mémoire vive. Ce système de swap, qui peut être considéré comme la mémoire virtuelle du pauvre, se retrouve sur le PDP-1 de Digital mis au point en 1962 pour BBN (Bolt, Beranek & Newman), un nom que nous retouverons au chapitre consacré aux réseaux, puis sur beaucoup de machines de la gamme PDP.

En 1963 la société française SEA dirigée par F.H. Raymond réalisait pour sa machine CAB 1500 un système qui reposait sur des principes différents, les noms généralisés.

9. La pile dont il est question ici n'est pas une pile électrique, mais une structure de données qui évoque une pile d'assiettes, et dont nous donnerons une description un peu plus loin.

En 1967 IBM réalise le modèle 360/67, qui était un 360/65 doté des dispositifs de mémoire virtuelle que nous avons décrits plus haut, avec des pages de 4 096 octets et en outre un dispositif logiciel que l'on pourrait appeler hyperviseur, CP/67, que nous évoquerons au chapitre 10, qui permettait de simuler le fonctionnement de plusieurs ordinateurs virtuels dotés chacun de son système.

En 1972 IBM généralise la mémoire virtuelle sur sa gamme 370.

4.5 Hiérarchie de mémoire

4.5.1 Position du problème

La recherche d'informations dans des espaces de mémoire très vastes nous a amenés à poser des questions de performance : parcourir séquentiellement une table pour y chercher une information quelconque demande l'exécution d'un nombre d'instructions proportionnel à la longueur de la table. Dès que cette table devient longue il faut trouver une solution plus subtile. C'est un problème très fréquent en programmation.

Un schéma très général de solution possible, inspiré d'ailleurs des méthodes exposées ci-dessus, est le suivant : si dans cette table figurent d'une part une petite minorité d'informations très souvent utilisées et de l'autre une majorité d'informations rarement utilisées, et que nous disposions d'un moyen d'identifier la petite minorité active de notre stock d'information (le « *working set* »), il serait alors possible de réaliser deux versions de la table : une petite table pour le *working set*, d'accès rapide parce que petite, et la grande table complète, peu rapide d'accès mais rarement utilisée. Lors d'une recherche on lancera simultanément la consultation dans les deux tables, et que le meilleur gagne : si l'information cherchée est dans la petite table, c'est elle qui « gagnera » (sauf si le résultat est au début de la grande table), sinon on fera la recherche longue dans la grande table. C'est l'idée de la hiérarchie de mémoires, une mémoire petite et rapide en avant–plan d'une mémoire vaste, plus lente mais complète.

Notre problème est un peu plus complexe : la composition du *working set* varie dans le temps, une information très utile à un instant va devenir inutile quelques microsecondes plus tard, cependant que d'autres informations vont sortir de l'ombre pour occuper le devant de la scène. Il nous faudra disposer d'un algorithme de remplacement des informations vieillies par de nouvelles vedettes. Remarquons que cette idée de *working set* se marie bien avec la constatation notée plus haut de la localité des traitements.

Le chapitre 2 nous a déjà procuré un tel exemple, un peu particulier, de hiérarchie de mémoire : les registres du processeur ne sont rien d'autre que des cases de mémoire incorporées à l'unité centrale pour pouvoir être atteintes et

modifiées plus rapidement. Les sections précédentes nous ont permis d'en voir deux autres exemples :

- l'organisation de la mémoire virtuelle vise à ne maintenir en mémoire réelle que les pages actives à un instant donné, (le *working set*) et à reléguer en mémoire auxiliaire les pages inactives ;
- le TLB conserve dans une toute petite table les traductions des adresses en cours d'usage, ce qui satisfait plus de 99% des demandes de traduction.

Le cas du TLB est spectaculaire parce que le rapport entre le nombre d'adresses virtuelles possibles et le nombre de celles qu'il conserve est énorme : pour un processeur 32 bits (quasiment une antiquité...) chaque espace adresse comporte 2^{20} pages de $4\ 096$ octets et le TLB en indexe $64 = 2^6$, soit si nous avons à un instant donné 20 espaces adresse actifs (valeur très modeste) une sur $20 \times 16\,384 = 327\,680$, et il résout néanmoins plus de 99% des défauts de pages.

4.6 La technique du cache

Le mot cache, curieux retour au français d'un emprunt anglais, suggère ici l'idée de cacher dans un coin (techniquement parlant, dans une zone de mémoire petite mais à accès très rapide) pour l'avoir sous la main une chose que l'on ne veut pas avoir à aller chercher à la cave ou au grenier (i.e., dans la mémoire centrale, vaste mais à accès lent par rapport à la vitesse du processeur), pour gagner du temps. À moins qu'il ne s'agisse de l'image de la chose, au premier plan d'une scène, qui cache la chose elle-même, à l'arrière-plan.

4.6.1 Cache mémoire

Les processeurs modernes utilisent la technique du cache pour les accès à la mémoire. De quoi s'agit-il ?

Supposons que l'accès du processeur à la mémoire par le bus système se fasse en un temps t. Une petite quantité de mémoire très rapide va être implantée sur le processeur proprement dit, ce sera le cache de premier niveau (L1, pour *Level 1*) qui aura un temps d'accès de $\frac{t}{40}$ (habituellement un ou deux cycles de processeur, soit de l'ordre de la nano-seconde). Puis une quantité un peu moins petite de mémoire un peu moins rapide va être connectée au bus interne du processeur, avec un temps d'accès, par exemple, de $\frac{t}{10}$. Ce sera le cache de niveau 2, L2, avec un débit d'accès de quelques milliards d'octets par seconde (les ratios de temps d'accès indiqués ici sont des ordres de grandeur vraisemblables).

Le cache L1 de l'Itanium 2 (2002), représenté par la figure 4.3, contient 32 Kio de mémoire par cœur (16 Kio pour les données, 16 Kio pour les instructions) avec un délai d'accès (temps de latence) de 2 cycles, son cache L2, qui est sur le

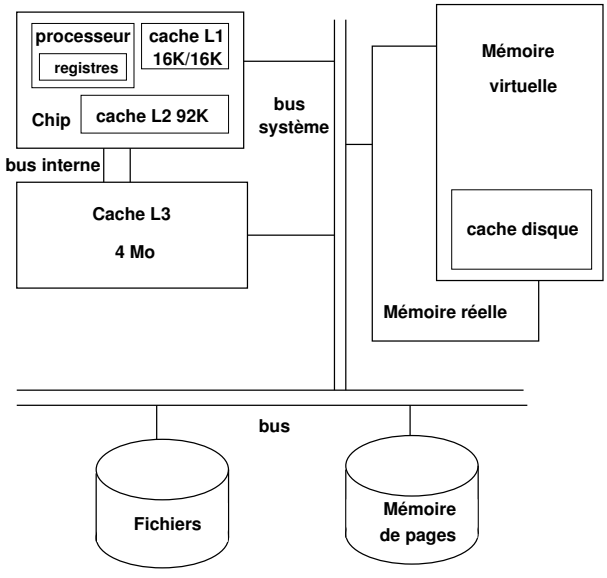


Figure 4.3 : *Hiérarchie de mémoire d'Itanium*

même circuit (la même puce) que le processeur, 256 à 1280 Kio par cœur avec un accès en 6 cycles, et il a un cache externe L3 de 1,5 à 24 mébioctets selon les configurations avec un accès en 21 cycles.

L'accès à la mémoire principale se fait à un débit de de 2,1 gibioctets[10] par seconde par un bus à 133 Mhz.

Pour prendre un type de processeur plus récent, l'Intel Xeon E3-1285 v3 avec quatre cœurs à 3,6 GHz, en géométrie 22 nm, possède 64 Kio de cache L1 et 256 Kio de cache L2 par cœur, et 8 Mio de cache L3. On voit que si le pas de la gravure (usuellement représenté par la longueur de la grille d'un transistor) et les vitesses de processeur qui en résultent ont connu une évolution spectaculaire, les données relatives aux tailles des caches ont évolué plus modérément. En effet, la localité des traitements (telle que définie à la section 4.4.2) fait qu'agrandir inconsidérément les tailles de cache ne procure que des gains de performance rapidement décroissants.

4.6.2 Hiérarchie de mémoire : données numériques

J'emprunte à Jeffrey Dean[40], architecte système chez Google, le tableau des temps d'accès comparés (en nanosecondes) aux différents niveaux de mémoire qu'il a présenté en 2009 à la conférence *Large Scale Distributed Systems and*

10. Cf. note 8 p. 87.

Middleware (LADIS) et qui ont été actualisés[11] en 2018 (on notera que depuis au moins 2010 les vitesses de processeurs n'augmentent plus, à la suite d'une sorte d'accord d'armistice entre les industriels, parce que la concurrence sur ce terrain menait à des évolution néfastes, en termes de dissipation thermique essentiellement):

Numbers Everyone Should Know (2018)	
Register	0.25 ns
L1 cache reference	0.5 ns
Branch mispredict	3 ns
L2 cache reference	4 ns
Mutex lock/unlock	17 ns
Main memory reference	62 ns
Send 2K bytes over local network	88 ns
Compress 1K bytes with Zippy	2 000 ns
Read 1 MB sequentially from memory	5 000 ns
SSD random read	16 000 ns
Read 1 MB sequentially from SSD	78 000 ns
Round trip within same datacenter	500 000 ns
SAS/SATA Disk seek	3 000 000 ns
Read 1 MB sequentially from disk	1 000 000 ns
Send packet CA->Netherlands->CA	150 000 000 ns

4.6.3 Mise en œuvre du cache

Comment seront utilisés ces caches? Leur utilité repose sur la localité des traitements (cf. 4.4.2 p. 79): on a observé qu'à une échelle de temps petite pour l'observateur mais grande par rapport à la vitesse du processeur, disons pendant l'exécution d'un sous-programme moyen, le processeur accède toujours à peu près aux mêmes zones de la mémoire. Donc si on charge ces zones dans le cache on va accélérer les traitements.

L'algorithme de gestion du cache consiste à y charger toute zone de mémoire demandée par le processeur à la place de la zone la moins récemment utilisée de celles qui étaient déjà là, en spéculant sur le fait que si cette zone est demandée maintenant elle va l'être souvent (des milliers de fois) dans les instants qui suivent (quelques milli-secondes). La partie plus compliquée de l'algorithme consiste à maintenir la cohérence entre les caches et la mémoire principale par la réécriture des zones modifiées dans le cache. Le lecteur trouvera dans le livre de Hennessy et Patterson [58] toutes informations souhaitables sur

11. `https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html`

ce mécanisme de hiérarchisation de la mémoire. S'il est soucieux de se tenir au courant des derniers développements techniques il devra s'abonner à l'excellente revue *Microprocessor Report* [90]. Deux choses importantes doivent être retenues à propos de la technique du cache : elle procure des augmentations de performances très spectaculaires, et aucun modèle général satisfaisant de son fonctionnement n'a pu être proposé à ce jour.

4.7 Langage et mémoire

Le système d'exploitation alloue à chaque programme l'espace de mémoire nécessaire à son exécution. Comment se présente cet espace, du point de vue du programme ? Cela dépend du programme, et singulièrement du langage de programmation utilisé, ou plus exactement du traducteur. Mais auparavant il n'aura sans doute pas été inutile de se remémorer les développements page 46 sur la notion de sous-programme, qui permet de découper un grand programme en entités plus maniables.

4.7.1 Langages à mémoire statique

Le cas le plus simple est celui des langages à gestion de mémoire statique comme Fortran (nous parlons du Fortran IV traditionnel ou de son successeur Fortran 77, pas du langage au baroquisme ébouriffant qui porte aujourd'hui ce nom), pour les programmes desquels la mémoire est allouée une fois pour toutes au lancement du programme. Le texte du programme en langage machine et les variables sont à des emplacements fixes et ne sont pas extensibles, c'est-à-dire que tous les noms présents dans le programme sont associés (liés) à des emplacements de mémoire lors de la compilation. Le compilateur construit l'image binaire du programme traduit et de ses zones de données, l'éditeur de liens assemble les différents sous-programmes sans oublier les sous-programmes de bibliothèque en effectuant les translations appropriées (voir à la page 76), et le résultat est un fichier binaire exécutable prêt à être chargé en mémoire pour s'exécuter.

De cette politique d'allocation de mémoire il résulte que les variables locales d'un sous-programme sont au même endroit à chaque activation du sous-programme.

Cette technique d'allocation a trois inconvénients :

1. la taille de chaque structure de donnée doit être connue une fois pour toutes à la compilation et elle ne pourra plus varier pendant l'exécution ;
2. les sous-programmes ne peuvent pas s'appeler eux-mêmes (ce que l'on appelle récursion) parce que les objets locaux pour chaque activation partageraient les mêmes emplacements de mémoire ;

3. il n'est pas possible de créer des objets dynamiquement à l'exécution.

et elle a deux avantages :

1. les programmes écrits dans des langages statiques peuvent être rapides, parce qu'ils ne nécessitent pas de création de structures de données pendant l'exécution, et parce que l'emplacement fixe des objets permet l'usage de l'adressage direct, plus efficace que l'adressage indirect ;
2. un programme à mémoire statique est par construction à l'abri de l'échec par pénurie de mémoire en cours d'exécution.

4.7.2 Vecteur d'état d'un programme

Tout sous-programme s'exécute dans un contexte [12] qui comporte au moins les informations que nous avons mentionnées ci-dessus à la page 46 : l'adresse de la liste (éventuellement vide) des arguments que lui aura transmis le programme appelant, l'adresse de retour à laquelle il devra effectuer un branchement quand il se sera terminé, l'adresse où déposer le résultat de son exécution. Il faut y ajouter les variables internes (locales) au sous-programme. Ces éléments de contexte constituent ce que nous appellerons vecteur d'état du (sous-)programme, et il s'agit donc d'une (petite) collection de mots.

4.7.3 Langages à mémoire dynamique

Tous les langages ne sont pas, tel Fortran IV, confinés à une mémoire statique. Les systèmes d'exploitation proposent depuis des lustres des mécanismes et des appels système qui permettent à un processus d'acquérir une zone de mémoire supplémentaire, et les langages en font usage pour surmonter les limitations mentionnées à la section 4.7.1.

Il y a deux grandes catégories de méthodes pour allouer de la mémoire supplémentaire dynamiquement à un programme lors de son exécution : sur la pile et dans le tas. La plupart des langages modernes utilisent les deux :

– on utilise généralement la pile pour les données qui tiennent dans un mot (nombre, adresse, pointeur...) ou peu de mots, doivent être traitées

12. Le contexte que nous envisageons ici est celui du programme vu comme la description d'un traitement. Il est distinct du contexte du programme vu comme *processus* que nous avons évoqué aux section 3.11.1 et 3.11.2. Le contexte du programme est créé et utilisé selon la logique du déroulement du traitement, dans l'espace adresse de l'utilisateur, cependant que le processus peut être interrompu à tout moment par un événement extérieur, asynchrone et sans respect pour le traitement en cours. La restauration du contexte du processus, qui comporte notamment les contenus du PSW et des registres, permet évidemment de retrouver le contexte du programme qui réside dans son espace adresse.

rapidement (liste d'arguments d'un sous-programme), et qui en outre ne doivent pas avoir une durée de vie qui excède la terminaison de l'exécution courante du sous-programme ;
– le tas est utilisé pour les données de taille quelconque et éventuellement variable (tableau, chaîne de caractères, liste...), ainsi que pour les données qui doivent survivre au sous-programme courant.

Nous allons examiner ces deux techniques. Mais ne perdons pas de vue qu'il s'agit toujours d'allouer de la mémoire disponible dans l'espace adresse du processus : quand cet espace-adresse sera saturé, les tentatives d'allocation déclencheront des erreurs.

Allocation de mémoire sur la pile

Dans un langage à gestion de mémoire dynamique, le système, pour chaque appel de sous-programme, crée un vecteur d'état (*activation record*, ou *activation frame* en anglais ; il contient, rappelons-le, les arguments passés par le programme appelant, l'adresse de retour, l'adresse du résultat éventuel, et les variables locales), et le détruit lorsque la procédure se termine.

Pour réaliser cela le compilateur utilise une structure de données appelée *pile* (en anglais *stack*) : les vecteurs d'état successifs, au fur et à mesure de leur création, vont être empilés comme des assiettes, puis dépilés au fur et à mesure de la terminaison des sous-programmes correspondants. À chaque instant un pointeur permet de connaître le sommet de la pile, qui correspond au sous-programme actif à cet instant.

Pour empiler un vecteur c'est simple : on place les mots de mémoire qui le constituent dans la zone qui commence à l'adresse immédiatement supérieure à la valeur courante du pointeur de pile, puis on additionne à celui-ci la taille du nouveau contexte afin qu'il pointe bien sur le sommet de la pile.

Pour dépiler un vecteur c'est encore plus simple, il suffit de soustraire du pointeur de pile la taille du vecteur à supprimer. En général, le résultat renvoyé par un sous-programme qui se termine aura été placé dans la pile, à un endroit convenu du vecteur d'état du programme appelant, c'est-à-dire « en dessous » du vecteur courant.

Pourquoi se compliquer la vie à faire tout cela, qui prend du temps, diront les adeptes de Fortran, les physiciens ? Pour avoir une organisation plus fine et plus sûre de l'information, pour faciliter le travail du programmeur et lui éviter des risques d'erreur, notamment par les traits suivants :

1. Différentes activations d'un sous-programme peuvent coexister, avec chacune son vecteur d'état distinct, ce qui permet notamment à un sous-programme d'être *récursif*, c'est-à-dire de s'appeler lui-même.

2. La taille d'une structure de données locale peut dépendre des arguments passés au sous-programme.
3. Les valeurs, associées aux noms locaux, contenues dans le vecteur d'état stocké sur la pile, sont détruites à la fin de l'activation, ce qui élimine une cause d'erreur de programmation.
4. Le vecteur d'état d'un sous-programme appelé ne peut plus exister après la terminaison de l'appelant.

Allocation de mémoire sur le tas

La plupart des systèmes offrent un appel système pour obtenir une allocation de mémoire, d'une taille quelconque déterminée par le programme à l'exécution, prise parmi les pages disponibles de l'espace adresse du processus. Pour l'OS 360 il s'agit de `GETMAIN`, pour UNIX de `brk()`, plus connu sous son habillage grand public `malloc()`. La mémoire demandée est prise dans une zone de l'espace adresse du processus appelée le *tas(heap)*, par opposition à la pile et le système renvoie au programme un pointeur sur la zone allouée, qui, lui, réside généralement dans la pile.

L'assembleur et les langages de bas niveau comme C et C++ utilisent des fonctions explicites comme `malloc()` pour obtenir de la mémoire dans le tas et lui faire correspondre les structures de données que le programmeur veut y placer, cependant que des langages dotés d'un plus haut niveau d'abstraction font ce travail en coulisse à l'insu du programmeur. Les cadres de pages ne sont effectivement affectés au processus que lorsque celui-ci génère une exception en essayant d'accéder à l'une de leurs adresses virtuelles.

Si l'allocation est explicite, la libération doit l'être aussi : imaginons un sous-programme appelé dans une boucle et qui à chacune de ses exécutions demande une allocation de mémoire dans le tas ; le pointeur qui permet d'y accéder est sur la pile et il sera donc libéré à chaque terminaison, mais il n'en va pas de même pour la mémoire obtenue sur le tas, dont rien ne permet de justifier la libération si le programme ne fait pas appel explicitement à la fonction `free()`, qui remet le pointeur de début de la mémoire libre à sa valeur antérieure. Et il est parfois difficile de savoir s'il est légitime de le faire.

En effet dans un programme complexe plusieurs sous-programmes peuvent faire référence à la même zone de mémoire allouée. Imaginons un sous-programme qui obtient une zone de mémoire, y place des données et renvoie comme résultat à l'appelant, en se terminant, la valeur du pointeur sur cette zone : le pointeur sur la zone obtenu initialement était sur la pile et disparaît avec le sous-programme, mais l'appelant en a récupéré la valeur, qu'il peut alors passer en argument à de nombreux autres sous-programmes, ce qui fait qu'à un instant donné un nombre indéterminé de sous-programmes possèdent dans leur vecteur d'état un pointeur sur cette zone dynamique.

Quand pourra-t-on la libérer? Quand plus aucun pointeur actif n'y fera référence. Comment le saura-t-on? C'est un problème difficile, dépourvu de solution simple, pour lequel existent plusieurs algorithmes heuristiques.

Gestion de la mémoire dynamique

Les langages évolués contemporains peuvent au bout du compte être classés en deux catégories: les langages à gestion de mémoire explicite comme C, C++, Pascal, où le programmeur est responsable de l'allocation et de la libération des zones de mémoire obtenues dynamiquement sur le tas, et les langages à gestion de mémoire automatique, comme Lisp, Smalltalk, Scheme, Caml ou Java, où la mémoire est allouée implicitement quand la création d'un objet le nécessite. Les langages de la seconde catégorie mettent en œuvre un algorithme heuristique de libération des zones de mémoire allouées et devenues inutiles; ces algorithmes sont connus sous le nom de ramasse-miettes, ou glaneur de cellules *(garbage collector)*, en abrégé GC; ils s'exécutent périodiquement, de manière asynchrone par rapport au programme lui-même. Il existe une réalisation de Scheme pour Macintosh, MacGambit, qui offre au programmeur une présentation visuelle très suggestive du déclenchement et de l'exécution du GC.

Signalons le cas intéressant du langage Ada, dont toute la logique interne suggère la présence d'un GC, prévu d'ailleurs par les concepteurs, mais le langage a trouvé son public dans la communauté du temps réel, c'est-à-dire des ingénieurs qui écrivent des systèmes pour piloter Ariane V; les auteurs de tels systèmes sont très rétifs à l'idée du déclenchement asynchrone du GC, par exemple au moment de la mise à feu du second étage de la fusée lanceuse. Cette réticence a longtemps retenu les auteurs et réalisateurs d'Ada, qui n'avait pas de GC mais à la place la procédure générique `UNCHECKED_DEALLOCATION`, ce qui veut tout dire. Les versions récentes de la norme Ada (2005, 2012) prévoient un GC, qui peut être désactivé. La norme autorise la publication de compilateurs sans GC.

Les langages à gestion de mémoire explicite (C, C++) sont des langages de bas niveau, qui donnent au programmeur le contrôle d'objets très proches du matériel, en l'occurrence des pointeurs qui sont en fait des adresses mémoire. Ce contrôle est indispensable pour écrire des systèmes d'exploitation, des pilotes de périphérique, des programmes très dépendants du matériel. Mais pour écrire des programmes de résolution de systèmes d'équations, de comparaison de séquences d'ADN ou de construction d'arbres généalogiques, cela ne se situe pas au niveau d'abstraction approprié et oblige le programmeur (qui est peut-être un mathématicien, un biologiste ou un généalogiste) à acquérir des compétences dont un langage mieux adapté devrait le dispenser, d'autant plus que l'expérience tend à montrer que ces compétences sont acquises incomplètement et que la maîtrise des programmes qui en résulte est approximative.

Les langages à GC comme Scheme ou Java sont des langages plus abstraits (par rapport au réel, l'ordinateur) et de ce fait plus expressifs et plus utilisables pour des tâches moins tournées vers le système d'exploitation. L'argument de la performance est régulièrement avancé en faveur des langages à gestion de mémoire explicite : il est vrai qu'il est plus facile, avec ces langages, d'avoir des temps d'exécution assez stables, bien que les fluctuations induites par les différents niveaux de cache affectent tous les programmes.

4.7.4 Mémoire d'un programme en cours d'exécution

La plupart des compilateurs modernes construisent des programmes exécutables structurés en plusieurs sections, selon l'usage des données qui y seront placées :

section de code (instructions) : les instructions du programme ne doivent pas pouvoir être modifiées, ce qui induirait une faille de sécurité béante ; la section de code sera donc verrouillée de sorte qu'elle ne soit accessible qu'en lecture et interdite à l'écriture ;

section de données statiques (BSS) : cette section contient les données déclarées de taille fixe et non initialisées ;

section de la pile *(stack)* : cette section contient la pile, zone de mémoire organisée et gérée selon les principes exposés ci-dessus p. 96 ;

section du tas *(heap)* : cette section contient le tas, décrit p. 97.

Un programme peut avoir un plus grand nombre de sections, par exemple par sous-programme.

Chapitre 5 Persistance

Sommaire

Introduction

Les chapitres précédents nous ont montré l'activité frénétique des processus qui se bousculent pour obtenir le contrôle d'un processeur désespérément séquentiel et de l'espace mémoire qui pour être virtuel n'en est pas moins irrémédiablement fini. De cette activité résulte la construction de structures qui peuvent être grandioses, mais qui resteraient à jamais imperceptibles aux humains si l'ordinateur ne comportait des dispositifs destinés à communiquer avec

le monde extérieur: écrans, claviers, haut-parleurs, *joysticks*, imprimantes etc. Nous ne nous attarderons pas sur le fonctionnement de ces appareils et sur la façon dont le système d'exploitation les actionne, non que ce sujet soit dédaignable, mais parce qu'il est tout compte fait assez peu différent (en plus simple) du fonctionnement des mémoires auxiliaires sur disque magnétique que nous allons examiner maintenant.

5.1 Mémoire auxiliaire

Ces structures magnifiques construites dans l'espace immense de la mémoire virtuelle par des processus qui accomplissent un milliard d'actions par seconde, qu'en reste-t-il lorsque l'on interrompt l'alimentation électrique de l'ordinateur, ou même simplement lorsque le processus se termine? Rien. La technique employée actuellement pour réaliser la mémoire centrale repose sur des bascules à semi-conducteurs dont l'état dépend de la tension électrique aux bornes, et disparaît sans recours en l'absence de celle-ci. Il n'en a pas toujours été ainsi: jusque dans les années 1970 la technologie de mémoire dominante reposait sur une invention d'An Wang de 1950 et utilisait des tores de ferrite magnétisés. Un courant de commande créait une magnétisation dont l'orientation déterminait la valeur du bit, 0 ou 1. Le champ magnétique subsistait à l'interruption de l'alimentation électrique, ce qui permettait théoriquement de retrouver l'état de la mémoire au redémarrage. Mais cette possibilité était rarement utilisée: la mémoire était petite, les champs magnétiques étaient susceptibles aux perturbations créées par les courants de rupture lors de l'arrêt de la machine, le redémarrage du système d'exploitation comportait des opérations nombreuses qui utilisaient elles-mêmes la mémoire... Bref, si l'on voulait conserver les données et les résultats élaborés par le programme pendant son exécution, et cette conservation apparaissait bien nécessaire, il fallait d'autres dispositifs de mémoire dite auxiliaire, par opposition à la mémoire centrale.

Les types les plus répandus de mémoire auxiliaire sont les disques magnétiques et les SSD (pour *Solid-State Drive*). Les premiers sont en voie d'être supplantés par les seconds, plus rapides, mais le disque magnétique possède encore quelques avantages par rapport à son jeune concurrent.

5.1.1 Structure physique du disque magnétique

Nous n'entrerons pas dans la description détaillée du disque magnétique. Disons seulement ceci. Il comporte généralement plusieurs plateaux circulaires fixés en leur centre sur un axe, comme on peut le voir sur la figure 5.1 p. 103. Chaque face d'un plateau est recouverte d'une substance magnétisable assez semblable à celle qui revêt la bande d'une cassette de magnétophone. Chaque

surface est survolée par une tête de lecture-écriture constituée d'un électro-aimant. Pour écrire, le dispositif de commande envoie dans le bobinage de l'électro-aimant un courant de sens et d'intensité propre à créer un champ magnétique qui va inscrire sur la surface du disque un bit à 0 ou à 1. Pour lire, l'électro-aimant va se laisser traverser par le champ magnétique du bit inscrit sur la surface, ce qui va créer dans son bobinage un courant induit que le dispositif de commande va interpréter comme un 0 ou un 1.

Les bits sont situés sur des pistes concentriques, c'est-à-dire que pendant une opération de lecture ou d'écriture la tête est fixe au-dessus de la surface du disque, jusqu'à la fin du traitement de la piste, puis elle se déplace radialement, par exemple jusqu'à survoler la piste voisine. C'est différent des platines de lecture des disques vinyle : la tête ne touche pas le disque. Et contrairement aux CD-ROMs et aux disques vinyl, il n'y a pas une piste en spirale mais plusieurs, concentriques.

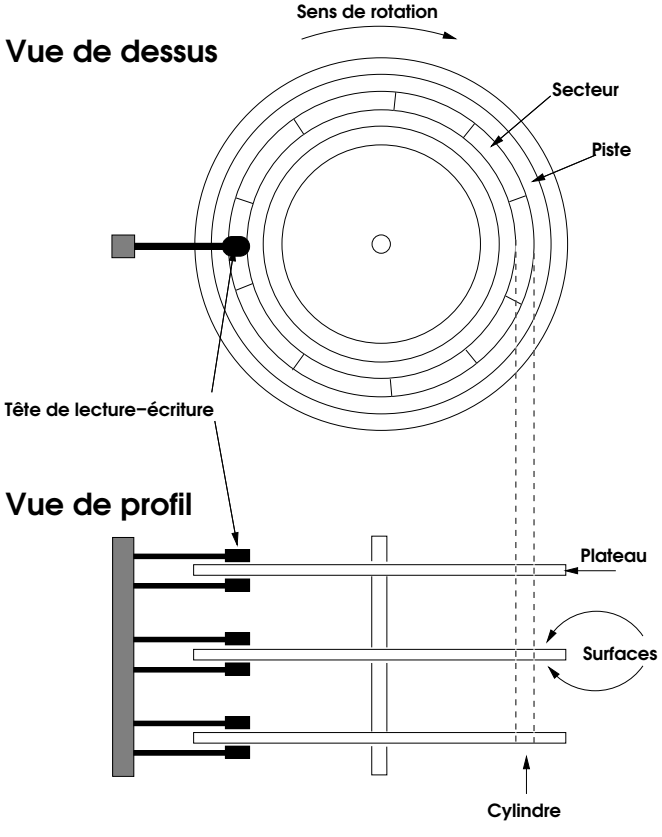


Figure 5.1 : *Organisation physique d'un disque*

Chaque piste est découpée en secteurs. Une donnée est repérée par son numéro de piste depuis la plus extérieure (appelé numéro de cylindre, parce que l'ensemble des pistes de même rang, situées à la verticale l'une de l'autre, constitue un cylindre), son numéro de plateau (ou de tête de lecture, ce qui revient au même), le numéro du secteur qui la contient et le rang de l'octet dans le secteur. Chaque opération d'accès physique au disque transfère un ou plusieurs secteurs d'un coup, c'est le logiciel (du contrôleur ou du système d'exploitation) qui découpe ou assemble les données à l'intérieur des secteurs.

Les plateaux et les têtes de lecture-écriture sont enfermés dans une enceinte scellée, le HDA *(Head-Disk Assembly)*, mais jusque dans les années 1970 les piles de disques étaient amovibles, ce qui diminuait beaucoup la fiabilité.

Le disque dur est un support de données à accès direct, par opposition à une bande magnétique par exemple qui est un support à accès séquentiel, c'est-à-dire que les blocs de données enregistrés sur une bande sont accessibles les uns après les autres dans l'ordre selon lequel ils ont été écrits; tandis que sur un disque chaque secteur est accessible individuellement. Le logiciel (en l'occurrence le système) donne au contrôleur un numéro de secteur ou de bloc, le contrôleur place les têtes magnétiques au-dessus du bon cylindre, sélectionne la tête correspondant au bon plateau et attend que le bon secteur passe dessous. Les données sont lues et envoyées au système.

Actuellement tous les contrôleurs présentent au système un adressage linéaire uniforme des secteurs, c'est-à-dire que tous les secteurs sont numérotés en séquence. Le contrôleur fait son affaire de traduire ce numéro d'ordre en numéro de cylindre, numéro de tête de lecture, numéro de secteur. Le contrôleur dispose aussi d'une mémoire tampon *(buffer)* assez vaste pour regrouper et optimiser les accès physiques.

Un disque de gros ordinateur à la fin des années 1960 pouvait contenir 14 millions de caractères, avec un diamètre de 14 pouces (36 cm) et une dizaine de plateaux, soit à peu près 15 cm de haut. En 2020 les disques récents pour serveur peuvent avoir une capacité de 4000 milliards de caractères avec un diamètre de 3,5 pouces (9cm) et trois plateaux (2,5 cm d'épaisseur totale). Les temps d'accès moyens ont beaucoup moins progressé, et sont de l'ordre de 4 millisecondes pour un accès aléatoire comportant un déplacement des têtes de lecture et une demi-rotation. Les vitesses de rotation, longtemps de 3 600 tours/minute, atteignent aujourd'hui 15 000 tours par minute. Les débits de transfert atteignent les cent millions de caractères par seconde. Dans ce domaine des performances il convient de noter que les temps d'accès peuvent varier considérablement selon le contexte d'utilisation, et que ces variations sont mises à profit par les publicités de certains vendeurs pour annoncer des chiffres à étudier avec précaution. Ainsi des temps d'accès moyens de 2 ms ne sont possibles que lors d'accès séquentiels où les secteurs d'une même piste sont lus successivement avec des déplacements

de tête rares, à la fin de chaque piste, ce qui ne correspond pas à un accès aléatoire.

5.1.2 Stockage SSD

De plus en plus se généralisent les disques SSD (pour *Solid-State Drive*), qui ne sont en fait pas des disques, mais de la mémoire « Flash » analogue à celle des clés USB, c'est-à-dire sans pièce mécanique mobile, ce qui assure d'excellentes performances, en contrepartie d'une capacité plus faible à cause du prix plus élevé de ce type de support (d'un facteur 6 à 8 en 2018). Le temps d'accès typique est de l'ordre de 0,1 ms, et il est constant du fait de l'absence de déplacement de pièces mécaniques. Les débits de transfert atteignent les six cent millions de caractères par seconde (six fois ceux des meilleurs disques classiques). La capacité peut atteindre 4 000 milliards de caractères (4 téraoctets), mais le coût est élevé (0,33 dollar par gigaoctet, contre 0,10 dollar en technologie classique).

Les premiers disques SSD, par souci de compatibilité, étaient dotés d'interfaces héritées des disques durs traditionnels (SATA, SCSI, cf. ci-dessous section 5.3 p. 120), mais la tendance est à l'adoption des interfaces NVMe *(Non-Volatile Memory express)*, directement raccordées au bus PCI Express, lancées en 2011 par un consortium qui regroupe tous les principaux industriels du stockage, et qui s'affranchit des contraintes héritées des caractéristiques mécaniques des disques traditionnels. En effet, avec une mémoire à base de composants électroniques sans pièces mécaniques mobiles, nul besoin de tenir compte des déplacements de têtes de lecture pour ordonnancer les accès, il est possible d'accéder simultanément à plusieurs emplacements physiques éloignés l'un de l'autre, par contre il faut veiller à minimiser le nombre d'écritures pour éviter l'usure des composants, qui ne peuvent supporter qu'un nombre limité de cycles d'écriture-effacement.

Pour un exposé plus détaillé des caractéristiques des mémoires SSD et des méthodes de gestion mises en œuvre par le système d'exploitation et par le micrologiciel embarqué qui les pilote, on pourra se reporter au chapitre que leur ont consacré Youngbin Jin et Ben Lee dans un numéro d'*Advances in Computers* [66]. La tendance actuelle (2020) s'oriente vers la possibilité offerte au système d'exploitation et aux logiciels d'application (on pense aux Systèmes de Gestion de Bases de Données, ou SGBD) d'interagir avec le micrologiciel du SSD et de lui déléguer certaines fonctions, par exemple le contrôle des autorisations d'accès aux données ou certaines opérations d'indexation et d'extraction des SGBD. On peut notamment citer le projet Willow [121].

5.1.3 Visions de la mémoire auxiliaire

Dès l'origine il y a eu deux visions de la mémoire auxiliaire. La première la considère comme une extension moins rapide et de plus grande capacité de la mémoire centrale, avec une propriété supplémentaire : la persistance. Ce serait un peu comme si, après la terminaison d'un processus, le contenu de son espace de mémoire virtuelle était conservé en mémoire auxiliaire de pages pour un usage ultérieur par le même programme ou par un autre. Le processus en se terminant aurait soin de laisser dans cet espace de mémoire des données pertinentes et dont la conservation soit utile.

La seconde vision considère les données persistantes comme foncièrement hétérogènes au contenu de la mémoire centrale. La notion de « fichier », métaphore qui sous-tend cette vision, est celle d'un employé administratif, des services fiscaux par exemple, qui calculerait les impôts d'une population de contribuables : il disposerait en entrée du fichier des personnes imposables, avec une fiche par personne, et par ses calculs il constituerait en sortie un fichier d'avis d'imposition, avec un avis par contribuable. Ce fichier en sortie serait transmis en entrée au service du courrier, lequel après mise sous pli et affranchissement le transmettrait à la poste, etc. On voit que le contenu des fichiers n'a pas grand-chose de commun avec le contenu de la mémoire de l'employé, qui incidemment pour faire son travail a recours à une autre fichier, le volume du Code général des Impôts.

Cette seconde vision introduit donc un type d'objet supplémentaire, le fichier. La naissance de la notion de fichier s'explique historiquement pour deux raisons. Incarner dans un ordinateur réel la vision de la mémoire auxiliaire comme extension de la mémoire centrale butait jusqu'à tout récemment sur des obstacles techniques importants dûs à la difficulté de réaliser des mémoires et des disques de capacité suffisante et de sûreté de fonctionnement assez stable pour y conserver de grands volumes de données. Ensuite, lorsque dans les années 1960 l'informatique a commencé à être utilisée pour la gestion des entreprises et des administrations, elle a rencontré des habitudes de travail où les données étaient justement organisées sous forme de fichiers de cartes perforées traitées par des machines mécanographiques [1], et c'est assez naturellement que la notion de fichier s'est transposée à l'informatique.

D'expérience la notion de fichier n'est pas d'une intuition facile. Les difficultés qui l'accompagnent ne sont pas uniquement pédagogiques, en témoignent le soin que mettent les informaticiens théoriciens à éviter d'en parler et la grande hétérogénéité des errances qui caractérisent ses réalisations techniques.

1. Une thèse historique discutable situe l'origine de l'informatique dans la mécanographie. Cette thèse a pris naissance dans l'entourage des grandes entreprises mécanographiques qui se sont converties à l'informatique de gestion, comme IBM et Bull.

Considérer la mémoire auxiliaire comme un prolongement persistant de la mémoire centrale, conformément à la première vision évoquée ci-dessus, est beaucoup plus simple à tout point de vue. Qui n'a eu l'occasion, en portant secours à un utilisateur néophyte, de l'entendre parler de son document « en mémoire » sans pouvoir distinguer la mémoire centrale du disque dur ? Dès les années 1960 les auteurs du système Multics avaient choisi de doter leur système d'une mémoire virtuelle de grande taille, dont une partie était constituée de données persistantes sur disque magnétique, chargées en mémoire centrale en tant que de besoin. C'était très élégant et simple, mais la technologie électronique de l'époque condamnait un tel système à la lenteur et à une relative inefficacité, et de surcroît cette solution contrariait les habitudes déjà prises par la corporation des informaticiens.

Au cours des années 1970 sont apparus les systèmes de gestion de bases de données (SGBD), destinés à perfectionner la notion de fichier en dotant une collection de fichiers d'un ensemble de logiciels et d'index qui en permettent une vision et une gestion logiques, globales et cohérentes. L'idée générale est simple : imaginons le système de gestion d'une bibliothèque, autre monde de fichiers. Nous aurons un fichier des auteurs, un fichier des ouvrages, un fichier des éditeurs, un fichier des sujets, etc. Un ouvrage peut avoir plusieurs auteurs, plusieurs éditeurs, plusieurs sujets, mais chaque information n'est enregistrée qu'une fois et les logiciels et les index permettent de retrouver, en balayant les fichiers, tous les livres de tel auteur ou sur tel sujet, sans avoir à écrire un programme spécial à cet effet. L'unicité d'enregistrement de chaque donnée (la non-redondance) concourt à garantir la cohérence de la base.

La commodité apportée par les bases de données dans la gestion des informations persistantes a donné l'idée de créer des systèmes d'exploitation entièrement construits autour d'un SGBD. Ce furent essentiellement le système Pick et la Série 3 d'IBM, dont la postérité a franchi le changement de millénaire sous le nom d'AS/400. Pick, l'œuvre de Richard Pick, était un système excellent pour la gestion de toutes sortes d'objets, mais la corporation des informaticiens a eu sa peau. Il faut dire que Pick apportait une telle simplification à la gestion que sa généralisation aurait obligé une quantité de programmeurs occupés à des tâches banales et répétitives, désormais réalisables par des non-informaticiens, à trouver un travail plus exigeant.

Aujourd'hui les recherches dans le domaine des systèmes d'exploitation basés sur une mémoire persistante se poursuivent, et nous pensons que leur succès serait la source de progrès notables dans la facilité d'usage des ordinateurs par les humains. Nous y reviendrons.

5.2 Système de fichiers

Puisque tous les systèmes d'exploitation disponibles pratiquement aujourd'hui utilisent des fichiers, nous allons décrire leur organisation. Nous prendrons comme exemple le système de fichiers des systèmes Unix ou Linux, qui à défaut d'une efficacité foudroyante a l'avantage de la simplicité et de l'élégance.

Pour Unix un fichier est une suite de caractères, un point c'est tout. Le programme qui accède au fichier reçoit les caractères les uns après les autres comme un flux, et c'est au programmeur d'avoir prévu les actions nécessaires pour reconnaître dans ce flux des structures de données plus complexes qu'il pourra organiser et traiter.

Les caractères qui constituent ce flux séquentiel qu'est le fichier résident sur disque dur, où ils occupent un certain nombre de pistes elles-mêmes découpées en secteurs (voir la figure 5.1). Un secteur contient généralement 512 caractères. Les secteurs qui constituent un fichier peuvent être physiquement consécutifs sur le disque, mais ce n'est pas forcément le cas. Pour retrouver un fichier, il faut un système de répertoire : chaque fichier possède un nom et le répertoire permet de faire correspondre au nom un emplacement sur le disque (cylindre-piste-secteur), ou plus exactement une collection d'emplacements. Pour éviter un trop grand morcellement les secteurs de 512 octets peuvent être alloués à un fichier par *blocs* de taille supérieure, en général pas plus de 16 secteurs par bloc, soit 8 192 octets.

5.2.1 Structure du système de fichiers Unix

Notion de système de fichiers

Pour utiliser un disque avec Unix, avant de pouvoir y écrire des fichiers, il faut y installer un ou plusieurs *systèmes de fichiers*. Ce terme, système de fichiers, désigne à la fois le principe d'organisation des fichiers, les éléments de logiciel qui implémentent (réalisent) ce principe, et un ensemble de fichiers organisés selon ce principe. Dans le cas où plusieurs systèmes de fichiers (au sens : ensemble de fichiers) résident sur le même disque, celui-ci sera partagé en partitions, chacune constituée de cylindres contigus et vue par le système comme si elle était un disque physique. Chaque partition reçoit un système de fichiers. Il existe aujourd'hui pour Unix des systèmes de fichiers (au sens : principe d'organisation et logiciels pour l'incarner) qui permettent à une partition de s'étendre sur plusieurs disques et de changer de taille dynamiquement, mais nous nous en tiendrons ici au système de fichiers classique de Unix, tel **ext4** pour Linux[2].

2. Les systèmes de fichiers tels que **ext4** sont « classiques » au sens où ils diffèrent de l'archaïque **UFS** ; **ext4** (ainsi que son prédécesseur **ext3**) permet la journalisation des opérations (cf. ci-

Il faut aussi mentionner le fait que les contrôleurs de disques modernes dissimulent de plus en plus au système la structure physique du disque : ils font leur affaire de la gestion des pistes et des cylindres, qu'ils optimisent, et présentent le disque au système comme une séquence de blocs logiques simplement repérés par leur numéro d'ordre (pour les PCs cette façon de voir les disques se nomme *LBA*, comme *Linear Block Addressing*). Pour compléter cet assaut d'abstraction en marche, les Unix modernes comme Linux permettent l'utilisation simultanée de systèmes de fichiers différents, comme par exemple les systèmes VFAT de Windows 98 ou NTFS de Windows 2000, dont les particularités sont cachées par un système de fichiers virtuel (VFS) qui présente aux autres éléments du système une interface uniforme, manipulée par des commandes identiques que VFS exécute au moyen d'opérations adaptées à chaque système de fichiers particulier, dont le détail est dissimulé à l'utilisateur. On dit que le traitement de ces systèmes de fichiers conformément à leur organisation particulière est pour l'utilisateur rendu « transparent », terme dont on observera qu'en informatique il est synonyme d'opaque.

La *i-liste*

L'origine de toute information sur le contenu d'un système de fichiers est le *super-bloc*, qui comporte notamment les données suivantes : taille du système de fichiers, nombre de blocs libres, début de la liste des blocs libres. Comme le superbloc contient des informations vitales pour la validité du système de fichiers, il est reproduit en plusieurs exemplaires à des emplacements convenus. La structure d'un système de fichiers **ext4** est représentée par la figure 5.2.

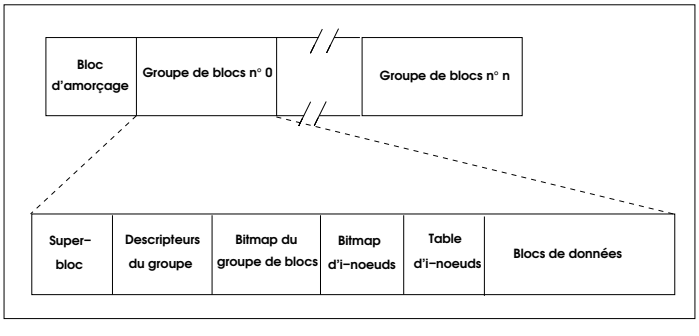


Figure 5.2 : *Structure d'un système de fichiers Ext4 et d'un groupe de blocs*

dessous section 5.4.1) et l'extension dynamique de partitions étendues sur plusieurs disques physiques. L'ancêtre commun des systèmes classiques est **FFS** *(Fast File System)*, créé pour Unix BSD par Marshall Kirk McKusick .

Le super-bloc pointe sur une autre structure de données cruciale, la *i-liste* (*i* pour index), qui est en quelque sorte une carte du système de fichiers, permettant d'y retrouver les fichiers. Pour les utilisateurs de tel ou tel autre système d'exploitation, la i-liste correspond à la *FAT-table* de Windows 98 ou à la MFT de NTFS, le système de fichiers de Windows 2000, ou encore à la *VTOC (Volume table of contents)* des grands systèmes IBM. Les éléments de la i-liste sont appelés *i-nœuds*. La i-liste doit refléter à chaque instant la structure logique et le contenu du système de fichiers, et comme celui-ci change à tout moment, au fur et à mesure que des fichiers sont créés, détruits, agrandis ou rétrécis, la i-liste doit posséder une structure très flexible.

Chaque fichier est décrit par un i-nœud, qui doit permettre d'en retrouver tous les fragments, puisque, comme nous l'avons dit plus haut, les blocs qui constituent un fichier ne sont pas forcément contigus, et comment en serait-il autrement d'ailleurs, puisque lorsqu'on agrandit un fichier les blocs qui viennent à la suite ont pu être utilisés entre-temps. Un i-nœud comporte :

- douze *pointeurs directs*, qui donnent l'emplacement sur le disque de douze blocs de données (cas de Linux, pour d'autres Unix ce peut être dix) ;
- un *pointeur indirect*, qui pointe sur un bloc de pointeurs directs, qui eux-mêmes pointent sur des blocs de données ;
- un pointeur *double indirect*, qui pointe sur un bloc de pointeurs indirects, qui eux-mêmes pointent sur des blocs de pointeurs directs, qui eux-mêmes pointent sur des blocs de données ;
- un pointeur *triple indirect*, qui pointe sur un bloc de pointeurs doubles indirects, qui eux-mêmes pointent sur des blocs de pointeurs indirects, qui eux-mêmes pointent sur des blocs de pointeurs directs, qui eux-mêmes pointent sur des blocs de données.

Si le fichier tient dans moins de douze blocs (et c'est le cas de la majorité des fichiers), il sera décrit par les pointeurs directs du i-nœud, et les pointeurs indirects ne seront pas utilisés, mais on voit que cette structure géométrique permet de décrire un très grand fichier (voir figure 5.3).

Avec des blocs de 4 kibioctets[3] ($4\,096$ octets), chaque pointeur tient sur quatre octets, et nous pouvons avoir :

- direct : 12x4Ki = 48 kibioctets ;
- indirect : 1024x4Ki = 4 mébioctets ;
- double indirect : 1024x1024x4Ki = 4 gibioctets ;
- triple indirect : 1024x1024x1024x4Ki = 4 tébioctets.

3. Voir note 8 p. 87.

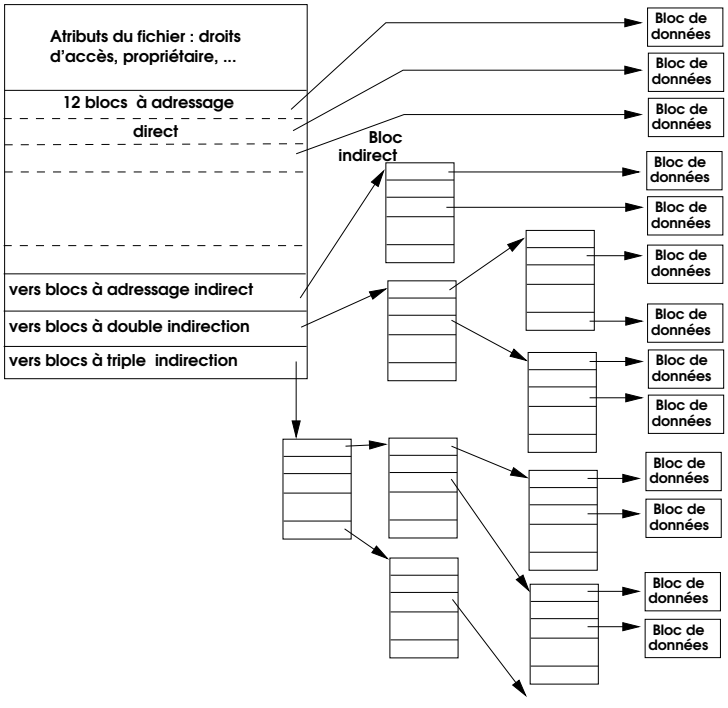


Figure 5.3 : *Structure d'i-nœud*

Outre ces pointeurs qui permettent, à partir du i-nœud, de retrouver les blocs de données, le i-nœud contient aussi d'autres informations capitales, qui sont les attributs du fichier :

- les droits d'accès au fichier ;
- l'identifiant numérique du propriétaire du fichier ;
- sa taille ;
- la date du dernier accès au fichier ;
- sa date de dernière modification ;
- et d'autres...

Le contenu d'un i-nœud peut être consulté par la commande Unix **ls**.

Le grand avantage de la structure de la i-liste sur d'autres méthodes de gestion, c'est qu'un i-nœud n'a besoin d'être chargé en mémoire que si le fichier qu'il décrit est en cours d'utilisation. Une table linéaire des blocs de disque devrait, *a contrario*, résider en mémoire de façon permanente, et avec les disques actuels ce serait vraiment encombrant.

Répertoires de fichiers

La i-liste permet de retrouver les fichiers sans ambiguïté par leur numéro de i-nœud, dont l'unicité est garantie, mais ce n'est pas un procédé réellement commode. Les êtres humains préfèrent souvent désigner un fichier (ou un autre objet) par un nom propre qui leur rappelle la nature des données, comme `Photo_Tante_Léonie` ou `fichier_clients_2013`. Pour répondre à cette attente, la plupart des systèmes d'exploitation proposent des *répertoires* (en anglais *directories*), appelés parfois aussi dossiers *(folders)* ou catalogues.

De même qu'un répertoire téléphonique permet, pour une personne dont on connaît le nom, de retrouver son numéro de téléphone, un répertoire de fichiers permet pour un fichier de nom connu de retrouver son numéro de i-nœud, qui lui-même permettra d'accéder au fichier. Un répertoire n'est qu'un fichier un peu particulier, dont le contenu est en fait une liste de fichiers, dont certains peuvent d'ailleurs être eux-mêmes des répertoires, appelés pour l'occasion sous-répertoires. Dans la liste figure, pour chaque fichier, son nom et son numéro d'i-nœud, ce qui permet de retrouver commodément le fichier par son nom.

La figure 5.4 donne le format des entrées de répertoire pour un système de fichiers sous Unix (en l'occurrence ici **ext2**, **ext3** ou **ext4** pour Linux). Les entrées sont de taille variable, ce qui offre l'avantage de permettre des noms de fichiers longs sans pour autant gaspiller trop d'espace disque[4].

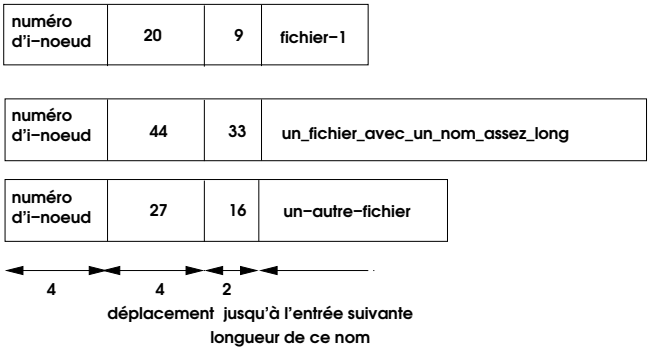


Figure 5.4 : *Entrées de répertoire d'un système de fichiers Unix (format classique)*

4. L'examen des sources du noyau et la lecture des bons auteurs m'a révélé la surprenante diversité des formats de répertoire parmi même différentes versions d'une même variété d'Unix, Linux en l'occurrence. Je donne donc ici un format générique, non pas une référence à utiliser les yeux fermés. Je ne parle ici que des systèmes de fichiers « classiques », parce que les systèmes novateurs comme *Reiserfs*, XFS et JFS ont abandonné les répertoires à structure de liste linéaire au profit de structures en arbre, à consultation plus rapide.

Du point de vue de l'utilisateur, un système de fichiers se présente donc avec une structure d'arbre.

Un arbre est une structure de données définie de la façon (récursive) suivante :

- un arbre est soit l'arbre vide soit un nœud ;
- un nœud a des fils qui sont des arbres ;
- si tous les fils d'un nœud sont l'arbre vide on dit que ce nœud est une feuille ;
- outre des fils, chaque nœud comporte une valeur.

Un arbre peut en outre avoir une racine, qui est un nœud situé en haut quand on le dessine, contrairement aux arbres des forêts. Les nœuds qui ne sont pas des feuilles sont parfois appelés « nœuds intérieurs ».

La racine de l'arbre « système de fichiers » est un répertoire tel que tous les fichiers du système de fichiers considérés figurent soit dans ce répertoire racine, soit dans un sous-répertoire du répertoire racine. Les sous-répertoires sont les nœuds intérieurs, et les fichiers ordinaires les feuilles de cet arbre.

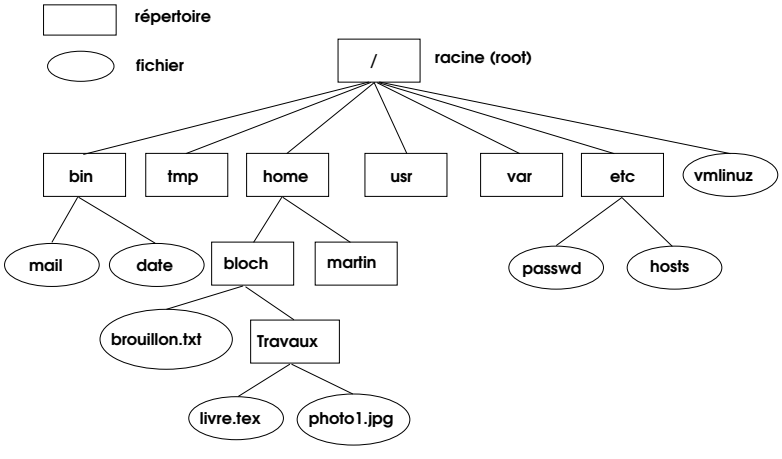


Figure 5.5 : *Arborescence des répertoires et fichiers Unix*

Les figures 5.5 et 5.6 représentent une partie d'un système de fichiers Unix. Un système Unix comporte au moins un système de fichiers, décrit par un répertoire appelé racine *(root)*. Il comporte traditionnellement les sous-répertoires suivants :

- **etc** contient les fichiers de configuration du système et des principaux logiciels ;
- **bin** contient les programmes exécutables fondamentaux ;

– **local** contient les données et programmes généraux propres au système local ;
– **home** contient les répertoires des utilisateurs ;
– et d'autres...

Ainsi, le fichier `photo_Tante_Leonie.jpg` qui appartient à l'utilisateur Marcel P., dont le nom d'utilisateur est `marcel`, sera répertorié dans le répertoire `marcel`, lui-même sous-répertoire de `home`, lui-même sous-répertoire de la racine, à laquelle on ne donne pas de nom. Par convention, le caractère « **/** » sert à marquer les échelons descendus dans l'arborescence du répertoire : ainsi le fichier `photo_Tante_Leonie.jpg` a-t-il comme nom complet depuis la racine `/home/marcel/photo_Tante_Leonie.jpg`. Comme nous n'avons pas donné de nom à la racine, tous les *shells* s'accordent à la nommer par convention « **/** ». Le nom complet d'un fichier, qui comporte les noms de tous les répertoires qu'il faut parcourir pour parvenir à lui depuis la racine, est aussi nommé *chemin (path)*.

Outre les sous-répertoires déjà indiqués, le répertoire racine répertorie aussi des fichiers ordinaires (par opposition aux répertoires) tels que **/vmunix**, le fichier exécutable du noyau, que l'auteur de Linux a baptisé quant à lui **/vmlinuz**.

Le terme de dossier employé parfois pour désigner les répertoires ne me semble pas très heureux parce qu'il suggère un contenant dans lequel seraient contenus les fichiers, ce qui n'est pas le cas. Un répertoire n'a d'étendue que celle nécessaire à loger ses propres informations sur les fichiers, mais pas les fichiers eux-mêmes, et encore moins les i-nœuds. Il est vrai que la tentation de cette métaphore est omniprésente : lorsque l'on enregistre un répertoire dans un autre répertoire, dont il devient de ce fait un sous-répertoire, tous les fichiers qu'il répertorie apparaissent désormais comme des feuilles de l'arbre de ce répertoire père. Comme, d'expérience commune, la navigation dans une telle arborescence est difficile à faire comprendre aux utilisateurs, l'espoir que la métaphore du dossier rende la chose plus accessible était légitime, même si peu confirmé.

Création d'un système de fichiers

On choisit généralement de fractionner l'ensemble de cette arborescence en plusieurs systèmes de fichiers, installés chacun dans une partition du disque. Ainsi les fichiers des utilisateurs sont-ils généralement répertoriés à partir d'un répertoire **/home**, qui répertoriera les répertoires personnels des utilisateurs, qui eux à leur tour répertorieront les sous-répertoires et les fichiers personnels. Il est considéré comme de bonne gestion de placer le répertoire **/home** et tous ses sous-répertoires dans un système de fichiers à part. Ainsi, notamment, lorsque l'on installera dans « **/** » une nouvelle version du système d'exploitation en effaçant tout l'ancien contenu, les fichiers des utilisateurs ne seront pas affectés. Pour réaliser ceci à partir d'un ordinateur vierge de tout système, on va partir par exemple d'un CD-ROM ou d'une clé USB qui contiendra d'une part les éléments

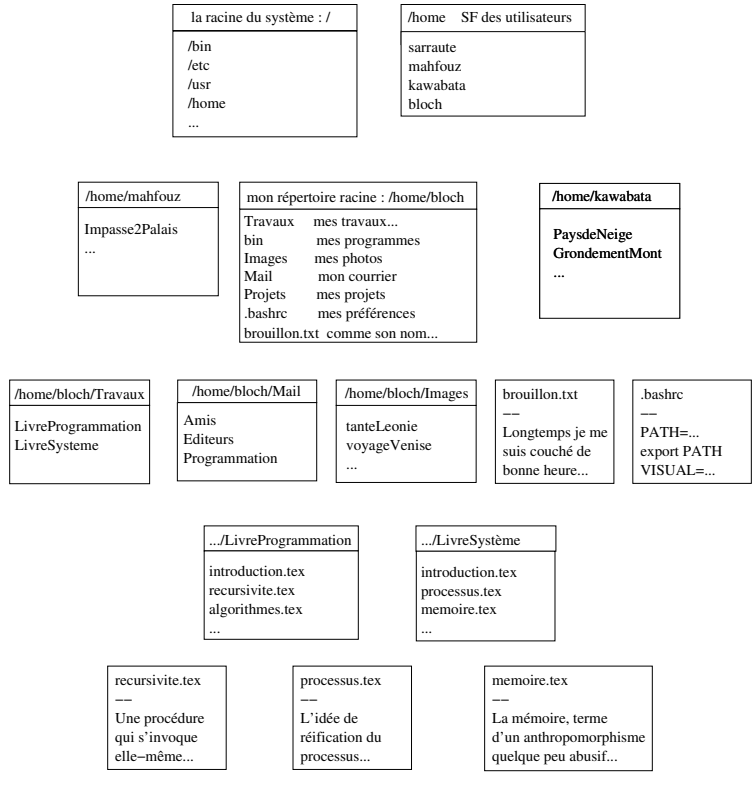


Figure 5.6 : *Répertoires et fichiers*

du futur système d'exploitation à installer sur disque dur, d'autre part une version rudimentaire du système pour pouvoir réaliser ce travail. Les distributions récentes du système GNU/Linux automatisent cette procédure, notamment au moyen du logiciel de partition **gparted**, ce qui fait que la plupart du temps l'utilisateur n'aura pas à faire tout cela « à la main », mais lorsque l'on utilise un système automatique il est bon de savoir ce qu'il fait à votre insu.

– On crée avec un programme utilitaire spécial (par exemple **fdisk**[5]) sur le ou les disques dont on dispose des partitions dont le nombre et la taille sont choisis après lecture de la documentation du système et selon les projets que l'on forme quant à l'utilisation de l'ordinateur. La création d'une partition installe sur le disque un certain nombre de structures de données. Nous prendrons l'exemple d'un disque configuré pour le système de

5. **fdisk** est un programme de bas niveau, que **gparted** invoque en dissimulant les détails ci-dessous à l'utilisateur.

fichiers **ext4** sous Linux sur un ordinateur de type PC, avec un BIOS traditionnel et de ce fait une table de partitions limitée à quatre partitions installée dans le premier secteur du disque, nommé *Master Boot Record* (MBR). Nous reviendrons sur ce sujet au chapitre 12 p. 339, où nous décrirons les systèmes plus récents, où le BIOS est remplacé par UEFI *(Unified Extensible Firmware Interface)* et la table du MBR par la GPT, pour *GUID Partition Table (Globally Unique IDentifier Partition Table)*, qui autorise 128 partitions sur un même disque, avec une capacité possible de 9,4 ZB ($9{,}4 \times 10^{21}$ octets).

  - le MBR, donc, contient la table des partitions du disque, repérées par les numéros de leurs secteurs de début et de fin ; une ou plusieurs de ces partitions peuvent être *bootables*, c'est-à-dire contenir un système d'exploitation susceptible d'être chargé en mémoire ; le MBR contient aussi un petit programme qui permet à l'utilisateur de choisir au démarrage la partition de *boot* (d'amorçage) ; ce petit programme était naguère **LILO**, il est aujourd'hui plus souvent **GRUB** ;
  - chaque partition a pour premier bloc un bloc d'amorçage, afin de pouvoir être *bootable* ;
  - le reste de la partition est divisé en groupes de blocs ; tous les groupes de blocs ont la même longueur et la même structure ; ils contiennent chacun une copie de la structure de données appelée « super-bloc » et une copie des descripteurs de groupes de blocs, qui décrivent la structure de la partition et notamment de la i-liste ; ces copies redondantes permettent de reconstituer la cohérence du système de fichiers contenu par la partition après un dommage physique provoqué, par exemple, par une coupure de courant intempestive ; le programme de réparation de système de fichiers sous Unix est **fsck**, homologue de *SOS Disk* bien connu des vieux utilisateurs de Macintosh ;
  - **fdisk** crée aussi la i-liste, destinée à décrire le contenu de la partition.

- On affectera une partition à la racine du système « / » et une autre partition à `/home` ; les partitions peuvent indifféremment être sur le même disque ou sur des disques différents ; il pourra en outre y avoir d'autres partitions dont nous ne parlerons pas ici, destinées à d'autres systèmes de fichiers tels que `/usr`, `/local`, `/tmp`, `/var`... En revanche `/bin` et `/etc` restent dans la partition racine parce qu'ils sont indispensables au démarrage du système, y compris pendant la phase où il n'a pas encore accès aux systèmes de fichiers « subordonnés ».
- On construit dans chacune de ces partitions un système de fichiers vide avec un programme utilitaire (en général `mkfs`) ; la création d'un système

de fichiers comporte notamment la création de son répertoire racine, qui est une racine « relative » par rapport à la racine « absolue », « / ».

– On crée dans « / » un répertoire vide nommé `home`. Ce répertoire vide sera le point par lequel à partir de « / » on accèdera au système de fichiers `home`, il est appelé *point de montage* du système de fichiers `home`. Cette opération de montage est décrite à l'alinéa suivant.
– La procédure de construction du nouveau système d'exploitation va comporter la création d'une table des systèmes de fichiers, le fichier `/etc/fstab`, qui indiquera que le répertoire à la racine du système de fichiers réservé pour les fichiers des utilisateurs sera raccordé à « / » au point `/home` : cette opération de raccordement s'appelle le *montage* d'un système de fichiers. On observera que les notions de système de fichiers et de répertoire sont distinctes bien qu'elles interagissent sans cesse. Le système de fichiers est un objet physique, installé sur une partition. La i-liste décrit physiquement les fichiers situés dans la partition considérée. Le répertoire est une structure logique qui sert à en décrire le contenu du point de vue de l'utilisateur.

Sous Linux, l'ensemble des opération décrites ci-dessus (partitionnement du disque, création des systèmes de fichiers, montage des partitions par rapport à la racine du système) peut être facilité et en partie automatisé par le logiciel **parted** ou, plus facile encore, par sa version avec interface graphique **gparted**, qui est d'ailleurs généralement invoqué par la procédure d'installation du système. Il restera toujours de la responsabilité de l'utilisateur de choisir sur combien de partitions répartir son système. Tout mettre dans un seule grande partition (comme le suggère la procédure d'installation de Windows) est une mauvaise idée, parce que s'il faut un jour réinstaller le système, ce qui arrive quand même de temps en temps, il faudra réinstaller aussi les données, à condition d'avoir une copie de sauvegarde à jour. Si on a pris soin d'avoir un **home** distinct de la racine, les données des utilisateurs seront préservées.

5.2.2 Traitement de fichier

Nous avons vu qu'un fichier était une suite de caractères, stockés dans un certain nombre de blocs sur disque ; un i-nœud permet de repérer ces blocs et de les identifier comme parties du fichier ; une entrée de répertoire permet d'associer à ce i-nœud un nom ainsi qu'un chemin d'accès depuis le répertoire racine du système « / ». Par exemple, si je veux conserver des collections de séquences d'ADN, je pourrai donner à mes fichiers des noms qui évoquent l'organisme dont proviennent les séquences qu'il contient : `listeria.monocytogenes`, `arabidopsis.thaliana`, `xenopus.laevis`. Ces noms ne sont-ils pas jolis ? Ils désignent respectivement la bactérie de la listériose, une petite plante commune mais élégante, une grenouille africaine.

Le programmeur qui écrit un programme pour lire ou écrire dans le fichier, de son côté, associe un nom symbolique au flux de caractères en entrée ou en sortie, par exemple `sequence.entree` pour le flux de caractères associé à la lecture d'une séquence à traiter.

On peut se représenter le fichier comme une file de caractères, lors de son ouverture un curseur est placé sur le premier caractère, chaque ordre de lecture renvoie le caractère placé sous le curseur et déplace le curseur jusqu'au caractère suivant.

Ouverture de fichier

Comment établir le lien entre un fichier physique, nommé par exemple `arabidopsis.thaliana`, et le nom symbolique d'un flux dans un programme, par exemple `sequence.entree` ? Cette connexion s'appelle assez universellement *l'ouverture (open)* du fichier. Elle consiste à construire en mémoire une structure de données qui contiendra toutes les informations relatives à l'une et l'autre entité, fichier et flux, ce qui permettra la réalisation effective des entrées-sorties. C'est vite dit, mais l'opération d'ouverture de fichier est assez complexe du fait du grand nombre de types de périphériques et de types de fichiers possibles, ainsi que de la grande variété de méthodes de traitement qui s'y appliquent. Sous Unix, une fois la structure de données construite, elle est ajoutée à une collection de ses semblables, la liste des fichiers ouverts pour le processus courant, liste dans laquelle son numéro d'ordre est appelé descripteur de fichiers.

Le fait de désigner le flux par un nom symbolique qui ne sera associé au fichier physique que lors de l'ouverture permet d'utiliser le même programme pour traiter les fichiers `listeria.monocytogenes`, `arabidopsis.thaliana`, `xenopus.laevis` et bien d'autres.

Une fois le fichier ouvert, il est possible d'exécuter des opérations de lecture ou d'écriture dans le flux correspondant, comme nous l'avons décrit aux sections 3.11.1 et 3.11.2.

5.2.3 Fichiers, programmes, mémoire virtuelle

Parmi les fichiers il en est qui jouent un rôle un peu particulier : ceux qui contiennent des programmes exécutables sous forme binaire. En fait ce sont des fichiers comme les autres, simplement au lieu d'être créés et lus par des programmes ordinaires, ils sont créés par des compilateurs[6], qui sont en fait des programmes comme les autres, et lus par le système d'exploitation lors du lancement du programme correspondant (voir le chapitre 2, et notamment la

6. et des programmes cousins appelés éditeurs de liens, destinés à réunir plusieurs programmes simples pour en construire un plus complexe.

section 2.7, ainsi que la section 3.10). Plus précisément, nous avons vu que c'était l'appel système **execve** qui devait charger le programme en mémoire et lui fournir des pointeurs sur les éléments de l'environnement établi pour le processus dans le contexte duquel il allait devoir s'exécuter.

Lorsque la mémoire virtuelle a été introduite, par exemple dans les systèmes IBM 370, un programme dont l'exécution démarrait était chargé depuis le fichier où il résidait vers la mémoire virtuelle ; par la suite les pages affectées à ce programme étaient éventuellement évacuées de la mémoire réelle vers la mémoire auxiliaire de pagination. Il y avait là quelque chose d'illogique, qui a été résolu dans les systèmes modernes : le fichier qui correspond au programme possède un format adapté à la pagination, et lorsque le programme est chargé en mémoire virtuelle ce fichier d'origine sert de fichier de pagination aux pages qui lui sont allouées.

5.2.4 Cache de disque

De même que le cache mémoire permet de garder près du processeur les mots de mémoire les plus probablement utilisés dans les instants qui suivent, le système réserve une partie de la mémoire pour y conserver les blocs disque les plus probablement utilisés. Ceci vient s'ajouter au fait que les disques modernes sont dotés de contrôleurs qui comportent également de la mémoire vive qui sert de cache.

Avec un système Unix, la taille du cache de disque s'ajuste dynamiquement à la taille de la zone mémoire disponible. Il n'est pas rare que le cache de disque occupe la moitié de la mémoire réelle à un instant donné. La stratégie est toujours la même : essayer d'avoir en mémoire la page de fichier ou le bloc de disque qui a une forte probabilité d'être bientôt lu ou écrit. Il faut donc trouver de bons prédicteurs des lectures ou écritures prochaines, et cela selon les types d'accès.

Pour le fonctionnement en lecture, le principe « ce qui vient d'être lu sera lu » peut être appliqué. Mais comme souvent les fichiers sont lus séquentiellement, il peut aussi être de bonne politique de charger en mémoire cache (de disque) les blocs de fichier qui suivent celui qui vient d'être lu.

Pour le fonctionnement en écriture, la question est : à quel moment les données contenues dans le cache vont-elles être écrites réellement sur le disque ? Deux politiques sont possibles : lancer simultanément l'écriture dans la mémoire de cache et sur le disque *(write-through)*, ou attendre un moment opportun ultérieur pour recopier le contenu du cache sur le disque *(write-back)*, par exemple lorsque le volume de données à écrire sera jugé optimum, ou lorsque le bus sera libre. La seconde méthode donne de meilleures performances, mais le laps de temps durant lequel les données ne sont qu'en mémoire volatile donne des frissons dans le dos des âmes pusillanimes.

Pour expliquer la différence entre les deux politiques de gestion de cache, risquons une comparaison. Je reçois chaque jour une masse de courrier qui s'empile sur mon bureau : ce sont des résultats d'opérations d'entrée-sortie. La plupart de ces courriers ne me concernent pas directement, mais je dois les conserver dans mes dossiers, pour un éventuel usage futur. J'ai le choix entre deux méthodes :

- chaque jour, ouvrir le courrier, repérer pour chaque message de quoi il s'agit afin de déterminer le dossier qui l'accueillera dans mes placards, et ranger chaque document à sa place ainsi choisie : c'est la méthode *write-through* ;
- laisser pendant des mois le courrier non ouvert s'empiler sur mon bureau ; quand l'invasion rend la situation intenable, trois ou quatre fois par an, ranger : c'est la méthode *write-back*.

Il va sans dire que j'ai recours à la seconde méthode, *write-back*, bien plus efficace : ainsi quand j'ouvre le courrier, une proportion importante des lettres, notes et autres convocations concerne des événements révolus depuis longtemps, et peut donc aller directement à la corbeille à papiers, sans passer par l'étape fastidieuse et coûteuse du rangement en dossiers. Les adeptes du *write-through* sont condamnés à devenir de purs bureaucrates à force de prendre au sérieux des messages dont l'expérience prouve que les ignorer purement et simplement ne cause aucun dommage. Et, *a priori*, j'aurai choisi comme date de rangement un jour tranquille où d'autres obligations plus urgentes ne seront pas différées à cause de cette activité subalterne.

Le rôle de la mémoire persistante est tenu par les dossiers dans les placards ; le plateau de mon bureau joue le rôle de cache. Un long entrainement me permet de savoir assez bien ce que contiennent les piles apparemment anarchiques qui encombrent mon bureau, et je suis capable d'accéder assez vite à un document si le besoin s'en fait sentir, en tout cas beaucoup plus vite que si je dois fouiller dans mes beaux dossiers bien rangés mais dont j'ai oublié depuis longtemps ce que j'y ai mis.

5.3 Systèmes de fichiers en réseau : NFS, SANs et NAS

Pour un centre de calcul d'une certaine importance, il y a trois modèles de solutions possibles pour stocker les bases de données :

1. disques connectés directement aux serveurs par des attachements IDE, SATA, SCSI *(Small Computer Systems Interface)*, SAS, etc. (ou NVMe *(Non-Volatile Memory express)* pour les disques SSD) ;
2. disques organisés selon la technologie SAN *(Storage Area Network)* ;
3. disques organisés selon la technologie NAS *(Network Attached Storage)*.

Nous allons examiner successivement ces trois solutions. Pour en savoir plus sur SCSI, SAN et NAS on consultera avec profit le livre de W. Curtis Preston[104].

5.3.1 Disques connectés directement aux serveurs

Cette solution est identique à celle qui existe pour les ordinateurs personnels de bureau. Il existe deux techniques de connexion : IDE et SCSI. IDE est limité à quatre disques, y compris d'éventuels lecteurs ou graveurs de CD-ROM, ce qui ne convient manifestement pas aux configurations envisagées ici, aussi nous limiterons-nous aux interfaces SCSI, non sans avoir signalé quand même qu'IDE a connu des évolutions (ATA comme *AT Attachment* et SATA comme *Serial ATA*) qui pourraient un jour en faire des concurrents sérieux de SCSI. Il existe déjà des armoires de disques SATA pour faire des NAS de second niveau bon marché.

La connexion d'un disque SCSI à un ordinateur suppose l'installation dans le fond de panier de l'ordinateur d'une carte contrôleur SCSI qui comporte l'interface adéquate, appelée *bus*. Il faudra aussi configurer le système d'exploitation pour y inclure les pilotes SCSI adéquats. À un bus SCSI on pourra raccorder, selon les versions, huit ou seize appareils SCSI, disques ou autres (le contrôleur compte pour un), qui seront connectés en chaîne les uns derrière les autres. Retenons que toute raccordement d'une chaîne SCSI à un ordinateur nécessite une intervention avec ouverture du boîtier de la machine. Il s'agit d'une connexion de bas niveau, étroitement couplée à un ordinateur donné.

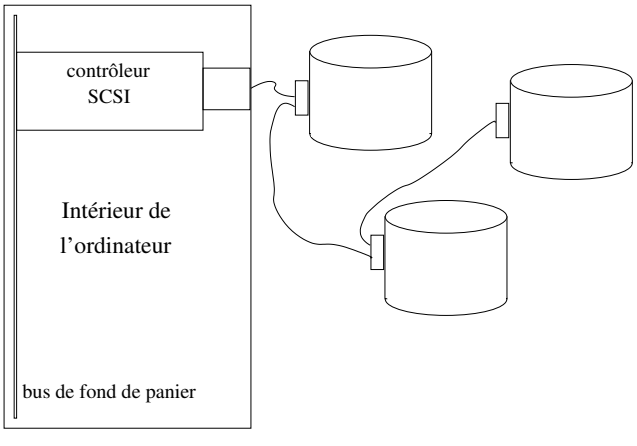


Figure 5.7 : *Chaîne de trois disques SCSI connectée à un ordinateur*

5.3.2 Systèmes de fichiers en réseau

Lorsque plusieurs ordinateurs cohabitent sur le même réseau local il est tentant de leur permettre de partager des fichiers. Cette tentation a donné naissance aux systèmes de fichiers en réseau, dont les principaux représentants sont NFS *(Network File System)* pour les systèmes Unix, SMB *(Server Message Block)*, aussi appelé CIFS *(Common Internet File System)*, pour les systèmes Windows. Citons également le système *AppleShare* pour les ordinateurs Apple.

Le principe de ces systèmes est toujours le même : le système de fichiers distant est présenté à l'utilisateur comme s'il était local, et celui-ci émet des appels système habituels pour y effectuer des opérations d'entrée-sortie. Le noyau du système intercepte ces appels système et les encapsule dans un message qui va être envoyé au système distant. Le message contient la description de l'opération d'entrée-sortie à effectuer. Il s'agit donc d'un appel de procédure à distance, ou *Remote Procedure Call*, RPC en abrégé. Le résultat de l'opération est retourné à l'expéditeur par le même procédé.

On conçevra aisément que ce processus, qui consiste à commander l'exécution d'un programme sur un autre ordinateur, est une faille béante de sécurité. Il est donc recommandé de limiter l'usage des systèmes de fichier en réseau à des environnements soigneusement contrôlés. Ces systèmes sont généralement des protocoles sans état, ce qui fait qu'ils ne comportent pas de système de verouillage pour garantir la cohérence des fichiers distants (voir à ce sujet la section 6.6.2). Il est souvent prudent de permettre l'accès à des systèmes de fichiers distants soit à plusieurs utilisateurs, mais uniquement en lecture, soit en lecture et en écriture, mais à un seul utilisateur.

Pour partager des données réparties de façon plus complexe sans encourir les risques que nous venons d'évoquer, il convient d'utiliser pour gérer les données accessibles par le réseau un Système de Gestion de Bases de Données, qui comportera alors les dispositifs désirables de contrôle d'accès.

5.3.3 Architecture SAN

L'architecture SAN est fondamentalement une extension de la technologie SCSI, dont elle reprend les principes tout en en améliorant la réalisation sur les points suivants :

1. le protocole SCSI a été étendu pour donner deux protocoles plus puissants, *Fibre Channel* et *iSCSI*, qui permettent des débits et des longueurs de câbles supérieurs ;
2. le maximum théorique d'appareils que l'on peu connecter à un SAN en *Fibre Channel* est de 16 millions ;
3. plusieurs ordinateurs connectés à un même SAN peuvent accéder concurrement à tous les disques du SAN.

On voit bien le progrès que la technologie SAN apporte en extension et en souplesse de configuration pour de vastes ensembles de serveurs et de support de stockage.

La figure 5.8 montre la topologie d'un SAN de type *fabric* en *Fibre Channel* ; il y a deux types de topologies possibles : *fabric* et *arbitrated loop* ; la seconde était justifiée par le prix prohibitif des commutateurs de type *fabric*, mais comme ceux-ci sont devenus plus abordables, la topologie *arbitrated loop*, moins efficace, ne se justifie plus vraiment et je ne la décrirai pas ici. Lorsque le protocole *iSCSI* sera sorti de son état actuel de quasi-prototype, il sera possible de construire des SANs à base de commutateurs Ethernet Gigabit, beaucoup plus économiques. En effet, un commutateur 16 ports *Fibre Channel* de marque Brocade coûte de l'ordre de 20 000 Euros, contre 6 000 Euros pour un commutateur Gigabit Ethernet chez Cisco.

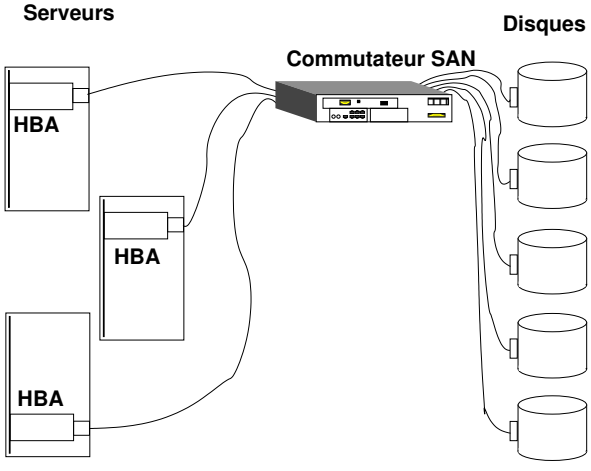


Figure 5.8 : *Topologie d'un SAN en* Fibre Channel

Les serveurs comportent des HBA *(Host Bus Adapters)*, qui ne sont pas autre chose que des contrôleurs SCSI adaptés au support *Fibre Channel*.

Contrairement à ce que pourrait suggérer la figure, le commutateur n'affranchit pas les serveurs de la gestion de bas niveau du protocole *Fibre Channel* (c'est-à-dire en fait SCSI), ils doivent notamment être dotés du matériel et des pilotes adéquats. Le commutateur ne fait qu'aiguiller des flots d'octets vers la bonne destination.

Comme les données sur les disques sont traitées au niveau physique, cela veut dire que les serveurs doivent être dotés de logiciels absolument compatibles, il n'y a aucune abstraction des données.

5.3.4 Architecture NAS

Comme l'indiquent les noms des protocoles d'accès aux données généralement proposés, un NAS *(Network Attached Storage)* est un serveur de **fichiers** (le terme est important) connecté au réseau. Les autres serveurs peuvent accéder au fichiers servis par le NAS au moyen des protocoles de partage de fichiers habituels : NFS *(Network File System)* pour les systèmes Unix, SMB *(Server Message Block)*, aussi appelé CIFS *(Common Internet File System)*, pour les systèmes Windows.

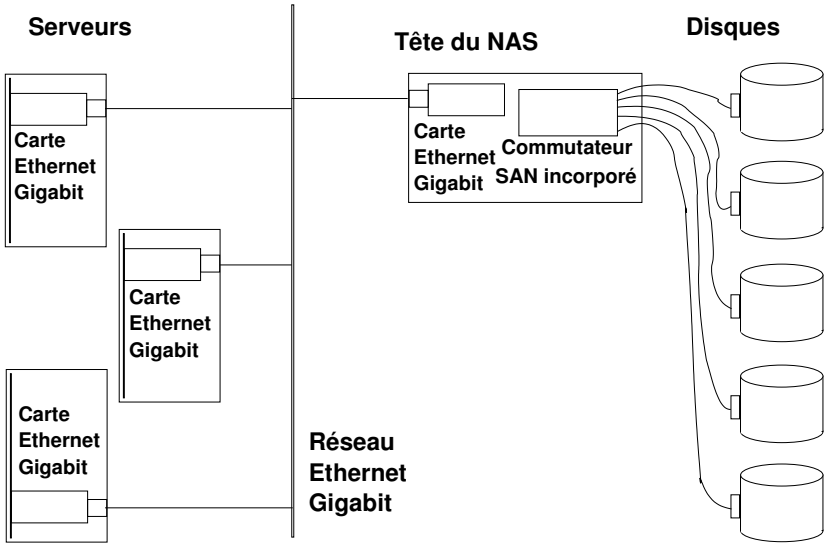


Figure 5.9 : *Topologie d'un NAS*

La figure 5.9 représente l'architecture d'un NAS. L'objet appelé « tête du NAS » est en fait un ordinateur équipé d'un système d'exploitation spécialisé qui ne fait que du service de fichiers. En général c'est un système Unix dépouillé de toutes les fonctions inutiles pour le système de fichiers, et spécialement optimisé pour ne faire que cela. Le plus souvent, la façon dont le NAS effectue la gestion de bas niveau de ses disques est... un SAN en *Fibre Channel*.

Quelle est la différence entre un NAS et un serveur de fichiers ordinaire ? Fonctionnellement, on pourrait répondre : aucune. En fait, tout est dans le système d'exploitation spécialisé. Les protocoles de partage de fichiers mis en œuvre sur des systèmes ordinaires ont la réputation de performances médiocres et de robustesse problématique. Les NAS de fournisseurs sérieux ont résolu ces difficultés et offrent des performances excellentes.

Quel est l'avantage du NAS par rapport au SAN ? Les serveurs de calcul sont totalement découplés des serveurs de données, il n'est plus nécessaire de les équiper du matériel et des pilotes nécessaires à l'accès physique aux disques, une

carte Ethernet Gigabit (quelques dizaines d'Euros) et la pile TCP/IP standard font l'affaire et on ne s'occupe plus de rien. Il est prudent de prévoir un réseau réservé à l'usage du NAS et de ses clients.

Quelles sont les limites de l'architecture NAS ? C'est du service de fichiers, donc les accès de bas niveau ne sont pas disponibles (c'est d'ailleurs le but). Dans l'antiquité, les SGBD utilisaient souvent un mode d'accès aux disques qui court-cicuitait le système de fichiers *(raw device)*, pour des raisons de performances. Aujourd'hui l'amélioration des caractéristiques du matériel a fait que cette approche n'est plus guère utilisée. Les bases de données Oracle de l'Inserm sont installées en mode « système de fichiers ». Les base de données Oracle chez *Oracle, Inc.* sont installées sur des NAS de la maison *Network Appliance*.

Quels sont les autres services disponibles avec un NAS ?

- *mirroring* de systèmes de fichiers à distance, par exemple duplication des données sur un site distant ;
- gestion du cycle de vie des données : on peut prévoir un stockage de second niveau pour les données inactives mais qu'il faut garder, sur des disques plus lents et moins chers ;
- sauvegarde des données autonome, sans intervention des serveurs de calcul : la tête de NAS est un ordinateur, capable de piloter une sauvegarde selon le protocole NDMP *(Network Data Management Protocol)* supporté par le logiciel de sauvegarde *Time Navigator* utilisé par l'Inserm.

Dernier avantage : les solutions NAS sont moins onéreuses que les solutions SAN, parce qu'au lieu de mettre de la quincaillerie partout, le matériel destiné au stockage de données est concentré dans une seule armoire (un seul rack pour les NAS moyens, un boîtier pour les petits, à usage personnel).

5.4 Critique des fichiers ; systèmes persistants

Si nous nous remémorons la description du système de mémoire virtuelle donnée au chapitre 4 et que nous la comparions à la description du système de fichiers qui vient d'être donnée, nous ne pouvons manquer d'être frappés par leur redondance mutuelle. L'un et l'autre systèmes ont pour fonction d'assurer la persistance de données qui étaient dans la mémoire centrale pour y subir un traitement, qui cessent d'y résider pour une raison ou une autre, et que l'on souhaite néanmoins conserver pour un usage ultérieur. La différence entre les deux réside finalement dans les circonstances qui dans l'un et l'autre cas amènent les données à cesser de résider en mémoire, et c'est cette différence qui est à l'origine de réalisations techniques dont la ressemblance ne saute pas aux yeux. Mais au fond, la mémoire virtuelle et le système de fichiers font la même chose, avec des différences d'interface plus que de fonctionnement, et l'on peut

dire que si la mémoire virtuelle était venue plus tôt les fichiers n'auraient sans doute pas vu le jour.

D'ailleurs, des précurseurs ont choisi de s'en passer. Dans Multics, déjà évoqué à la section 3.10.1 p. 53, la mémoire virtuelle est découpée en segments de taille variable. C'est une structure de mémoire virtuelle plus complexe que celle que nous avons décrite, mais elle remplit les mêmes fonctions. Eh bien, pour Multics on dit simplement que certains segments sont persistants et résident de façon permanente sur disque. C'est un attribut d'un segment : la persistance ! Et pour savoir quels segments sont présents à tel ou tel emplacement de la mémoire persistante, on utilise une commande de liste des segments, en abrégé **ls**, que les Unixiens reconnaîtront.

Nous avons déjà mentionné le système Pick et celui de l'IBM AS400, construits autour d'une base de données. Le choix est surtout clair avec Pick : chaque donnée individuelle porte un nom utilisable dans l'ensemble du système, et ce système de nommage est unifié. Par exemple, si le système est utilisé pour une application de paie dans un organisme public, il y a une variable et une seule pour donner la valeur du point d'indice des fonctionnaires. Cette variable a un propriétaire (sans doute le Secrétaire d'État à la Fonction Publique), qui dispose seul du droit d'en modifier la valeur. Tous les programmes qui doivent connaître cette valeur peuvent y accéder. Lors de la parution au Journal Officiel d'une modification de la valeur du point d'indice, une seule opération garantit la justesse de tous les calculs.

Incidemment, cette architecture de données procure aux utilisateurs profanes (et aux autres !) une vision considérablement simplifiée de l'ensemble du processus de traitement et de stockage de l'information. Une grande partie de la complexité de ces choses pour le néophyte tient à la difficulté d'avoir une vision d'ensemble d'un système qui réunit et coordonne des objets qui à l'état actif sont en mémoire et qui au repos sont dispersés dans une foule de fichiers aux statuts variés. Avoir un concept unique de mémoire pour tous les objets, persistants ou non, et une désignation unique pour tout objet quels que soient son état et son activité, ce sont des améliorations intellectuelles considérables. Le corporatisme informatique en a eu raison, provisoirement souhaitons-le.

La recherche sur les systèmes persistants continue, même si elle reste assez confidentielle. L'augmentation des performances des processeurs et des mémoire devrait l'encourager en abolissant les obstacles de cet ordre qui ont eu raison des précurseurs.

Le principe de persistance orthogonale est apparu à la fin des années 1970, dans le domaine des langages de programmation. Il proclame que toute donnée doit être habilitée à persister pendant un délai aussi long qu'il est utile, et que la méthode d'accès à une donnée doit être indépendante de la nature de sa persistance. Dans les systèmes classiques il en va tout autrement : les données volatiles en mémoire centrale sont invoquées par leur nom, les données persistantes sur

mémoire externe sont accueillies en mémoire centrale comme le résultat de l'invocation d'une commande d'entrée-sortie. Un système persistant reléguera ces différences techniques dans les couches basses du système et présentera une interface uniforme d'accès aux données.

Les tentatives pour implanter la persistance orthogonale dans les langages ou les bases de données utilisées dans le contexte de systèmes d'exploitation classique comme Unix n'ont pas donné de très bons résultats, parce que le problème de l'accès aux données est trop fondamental pour être résolu de façon totalement différente par un programme d'application d'une part, par son environnement d'autre part. L'auteur de système persistant est amené à gérer la mémoire de manière parfois subtile et complexe, or dans un système classique tel Unix c'est le noyau qui décide des pages de mémoire virtuelle à garder en mémoire volatile ou à reléguer en mémoire auxiliaire, ce qui aboutit à des contradictions.

L'exemple d'un tel échec est fourni par les bases de données à objets qui avaient soulevé un grand intérêt dans les années 1990 avec leur programme très séduisant, qui consistait à stocker les données sous la même forme en mémoire centrale pendant leur vie « active » et en mémoire auxiliaire pendant leur vie « latente ». Le revers de la médaille était que le format des données dans les bases dépendait alors du langage de programmation et du matériel utilisés : une base de données créée par un programme C++ n'était pas accessible à un programme Java, et si elle avait été créée sur une machine Sun à processeur 32 bits elle n'était pas accessible à un programme exécuté par une machine à processeur Alpha 64 bits, sauf à passer par des programmes de conversion de données qui font perdre tout l'avantage attendu. De surcroît la mémoire persistante était réalisée à base de systèmes de fichiers classiques, totalement inadaptés à une telle fonction. Et enfin il résultait de tout ceci une programmation laborieuse et des performances le plus souvent médiocres. Si je puis mentionner mes modestes expériences personnelles de combat avec un Système de Gestion de Données Objet (SGDO) pourtant réputé très industriel (par opposition aux logiciels libres développés par des chercheurs dans les universités), je ne puis me déprendre d'une impression de bricolage : la trace du système révélait que le SGDO passait le plus clair de son temps à balayer en long, en large et en travers des arborescences de répertoire pour y chercher des données qui étaient tout à fait ailleurs, et à recopier un nombre incalculable de fois la même (grande) portion de fichier en mémoire (au moyen de l'appel système **mmap** pour les connaisseurs[7]). Il n'y avait bien sûr pour un utilisateur naïf aucun moyen simple d'extraire des données de la base : la seule façon était d'écrire du code C++ *bare metal*, exercice

7. **mmap** est un appel système qui réalise la projection d'un fichier en mémoire, c'est-à-dire son chargement intégral, ce qui permet ensuite d'y accéder au prix d'un accès en mémoire. C'est judicieux si l'on doit effectuer de façon intensive des accès à tout le fichier, mais si l'on veut simplement accéder à une donnée c'est un marteau-pilon pour écraser une mouche.

particulièrement punitif ou pervers. Alors que même si l'on peut reprocher aux Systèmes de Gestion de Bases de Données Relationnelles (SGBDR) tels que PostgreSQL, Oracle ou Sybase une certaine rigidité, ils offrent au moins une méthode d'accès aux données relativement normalisée et relativement simple avec le langage SQL.

La leçon à en tirer semble être qu'il vaut mieux implanter les fonctions de persistance au niveau du système d'exploitation, quitte à ce que ce soit une couche d'interface ajoutée à un système classique sous-jacent. L'implantation de la persistance dans le système garantit une uniformité de vision pour tous les programmes d'application, évite la redondance de fonctions qui alourdissait tellement les bases de données à objets, bref elle assure la cohérence de la sémantique d'accès aux données. C'était ce que faisait à sa façon Pick, et que fait encore en 2018 le *System i* d'IBM (précédemment AS400).

Les principaux projets de recherche en persistance au début de ce millénaire sont les suivants :

- Le projet **MONADS** a démarré en 1976 à l'Université Monash (Australie). L'abstraction de base est le segment, à la *Multics* : persistance à gros grain.
- ***Clouds*** vient du Georgia Institute of Technology (1988). Les abstractions sont l'objet et l'activité *(thread)*. Repose sur le micro-noyau *Ra*.
- ***Eumel*** et ses successeurs **L3** et **L4** ont leur origine en 1977 à l'Université de Bielefeld, puis au GMD (*Gesellschaft für Mathematik und Datenverarbeitung*, équivalent allemand d'Inria), et sont principalement l'œuvre du regretté Jochen Liedtke. Eumel fut le premier système à persistance orthogonale. Il s'agit aussi d'un système à micro-noyau (voir chapitre 10).
- ***Grasshoper*** est un système à persistance orthogonale développé à l'Université de Saint Andrews, Écosse. Les entités persistantes sont des *containers*, des *loci* et des *capabilities*. Dans les systèmes classiques la notion d'espace–adresse est inextricablement mêlée à celle de processus. Les *containers* et les *loci* sont des entités analogues mais mieux distinguées (pardon : orthogonales), qui ont vocation à persister. Le noyau du système lui-même est persistant (il y a quand même toujours une partie non persistante, ne serait-ce que pour le boot).
- ***Charm*** est le successeur de *Grasshoper*. Le noyau de *Charm* n'exporte aucune abstraction pour le support de la persistance, mais uniquement des *domaines de protection* capable de communiquer avec le noyau.
  Charm appartient à une nouvelle tendance parmi les systèmes d'exploitation : au lieu de cacher le matériel derrière des abstractions, il l'expose afin que les stratégies de gestion des ressources soient implémentées en « mode utilisateur ». Cela suppose des dispositions favorables de l'architecture matérielle.

Le motif est de séparer :

- les règles de gestion des ressources de bas niveau ;
- des mécanismes qui les implémentent.

L'auteur du système de haut niveau est libre d'implémenter règles et mécanismes par une bibliothèque.

Incidemment, on peut signaler qu'il a existé un système d'exploitation universellement répandu et qui possèdait beaucoup de caractéristiques « persistantes » : *PalmOS*, qui anime les ordinateurs de poche *PalmPilot* et *Handspring Visor*. Chaque programme restait en mémoire dans l'état où l'utilisateur l'abandonnait, et où il pouvait le retrouver lorsqu'il rallumait l'engin. Il n'y avait pas de vrai système de fichiers. C'était assez surprenant quand on était habitué aux ordinateurs classiques.

5.4.1 Reprise sur point de contrôle

La notion de *reprise sur point de contrôle (checkpoint-restart)* est bien sûr au cœur de tous ces systèmes. Le problème à résoudre est le suivant : si un programme est interrompu inopinément en cours de traitement, comment reprendre son exécution sans avoir à la recommencer depuis le début ? Idéalement, il faudrait reprendre au point où l'on s'était arrêté, mais il est raisonnablement acceptable de repartir d'un point en amont pas trop éloigné. La difficulté principale réside bien sûr dans la restauration de ce que nous avons appelé le vecteur d'état du programme : valeur des variables, et contenu des fichiers.

Ce problème n'est pas propre aux systèmes persistants, et il a déjà été abordé et résolu. Dès les années 60 l'OS 360 offrait une possibilité de *checkpoint-restart* pour les programmes utilisateur. Cette possibilité consistait à enregistrer périodiquement sur disque une copie de la mémoire et un relevé de l'état des fichiers ouverts. En cas d'incident le programme repartait automatiquement du dernier point de contrôle. Ce mécanisme entraînait une consommation d'espace disque considérable pour l'époque et surtout une organisation rigoureuse du lancement et de l'exécution des chaînes de programmes, ce qui en restreignait l'usage.

Aujourd'hui une fonction analogue existe pour les ordinateurs portables : une combinaison de touches de commande permet de sauvegarder le contenu de la mémoire sur disque et de mettre l'ordinateur en veille, puis de le réactiver plus tard, ce qui permet par exemple de changer de batterie sans passer par la procédure d'arrêt du système et de redémarrage, elle-même grosse consommatrice de temps et d'électricité.

Les SGBD convenables offrent la possibilité de déclarer des transactions, c'est-à-dire de déclarer un ensemble d'opérations, par exemple une mise à jour complexe de la base, comme une méta-opération atomique. Conformément

à l'étymologie une méta-opération atomique est insécable : soit toutes les opérations de la transaction sont accomplies correctement, soit elles sont toutes annulées. Ceci est destiné à éviter de laisser la base dans un état incohérent, voire inconnu. Les transactions sont généralement associées à un dispositif de *roll in-roll out* qui permet d'entériner une transaction *(roll in)* ou de ramener la base de données à l'état antérieur à la transaction *(roll out)*.

Depuis les années 1980 sont apparus sur les systèmes en production, issus de la recherche, les systèmes de fichiers journalisés tels *Andrew File System* de l'Université Carnegie-Mellon, *ADVFS* dérivé du précédent dans les laboratoires *Digital Equipment (DEC)*, *JFS* développé par IBM, *XFS* réalisé par *Silicon Graphics (SGI)*, et plus récemment sous Linux *Ext3fs* et *Reiserfs*. Le principe de la journalisation est le suivant : avant toute opération de modification du système de fichiers par une opération d'entrée-sortie, on enregistre dans un fichier spécial (journal, en anglais *log*) la description de l'opération, et après l'opération on enregistre qu'elle s'est bien passée. Ainsi, après un incident qui aurait corrompu une partie du système de fichiers, il suffit de « rejouer » les opérations enregistrées dans le journal pour restituer un état cohérent, ce qui est infiniment plus sûr et plus rapide que de recourir à un logiciel de contrôle et de restauration de la cohérence interne du système de fichiers tels **fsck** sous Unix.

Avec les systèmes persistants, le problème est simplifié autant que généralisé. Un système persistant digne de ce nom n'a ni fichiers ni *a fortiori* système de fichiers. Il est de ce fait indispensable de garantir à tout prix le maintien de la cohérence du contenu de la mémoire virtuelle, seul lieu de conservation des données, et ce quel que soit l'incident, y compris une coupure de l'alimentation électrique. Les systèmes persistants disposent donc d'un système d'enregistrement de points de contrôle, qui permet de sauvegarder périodiquement le contenu de la mémoire, et de reprise sur incident à partir du dernier point de contrôle. Ce mécanisme n'est pas une option comme dans les environnements classiques, mais un fondement du système. Une conséquence amusante de ce type de fonctionnement, qui surprit les auteurs des premiers systèmes persistants, c'est qu'un programme d'application ne peut pas être informé d'un arrêt du système, puisqu'il est reparti automatiquement comme si de rien n'était.

Chapitre 6 Réseaux

Sommaire

Introduction

La question des réseaux informatiques et celle des systèmes d'exploitation sont en principe distinctes, et elles forment dans les cursus universitaires des disciplines particulières, mais les ordinateurs contemporains sont pratiquement toujours connectés à des réseaux de quelque sorte, et les techniques qui leur permettent d'y accéder sont tellement intimement enfouies au cœur du système qu'il n'est guère possible de parler de celui-ci sans aborder ceux-là. De surcroît, des systèmes sont apparus qui mettent à profit le réseau pour regrouper plusieurs ordinateurs et les considérer comme un seul multi-ordinateur, ou *système distribué*. Bref, s'il est de bonne méthode de distinguer l'architecture des ordinateurs, qui traite de l'organisation des éléments matériels des machines, l'architecture des systèmes d'exploitation, qui envisage la couche logiciel d'interface entre le matériel et les programmes de l'utilisateur, et l'architecture des réseaux, consacrée aux moyens de communication entre ordinateurs distants, il est clair qu'aucun de ces domaines ne pourra être traité sérieusement sans qu'il soit fait appel aux deux autres.

6.1 Transmettre de l'information à distance

Le problème de base à résoudre pour concevoir et réaliser un réseau d'ordinateurs consiste à établir un échange de données entre deux ordinateurs distants. Ce problème se divise en deux parties : pour que les données circulent correctement elles doivent être représentées selon un codage approprié commun aux deux extrémités, et il y faut un support physique également approprié.

La position de ce problème remonte au moins à Aristote, qui a envisagé la communication d'information entre deux personnes en termes de message et de code. Incidemment, ce modèle est beaucoup mieux adapté à la communication entre ordinateurs qu'à la communication entre êtres humains, qui est extraordinairement compliquée par tout un contexte (culturel, social, sensoriel) et par des éléments non verbaux (expressions du visage, intonations de la voix) qui rendent ce modèle, dans ce cas, assez inapproprié. « Le langage travestit la pensée. Et notamment de telle sorte que d'après la forme extérieure du vêtement l'on ne peut conclure à la forme de la pensée travestie; pour la raison que la forme extérieure du vêtement vise à tout autre chose qu'à permettre de reconnaître la forme du corps [1] ». Bref, pour les ordinateurs le modèle aristotélicien convient bien.

L'invention du téléphone a conduit à le formaliser sous le nom de « communication sur un canal bruité ». En effet il y a du bruit, c'est-à-dire qu'aucun canal de communication n'est parfait, certains éléments du message sont altérés ou perdus. Dans le cas du téléphone c'est tolérable jusqu'à un certain point, il en résulte quelques grésillements et bourdonnements ; Henrik Nyquist, dès les années 1920, et Claude Shannon[122] en 1948 ont posé les bases théoriques précises de ce que veut dire ici « jusqu'à un certain point », et ces bases constituent la théorie dite de l'information [2]. Il va sans dire que pour transmettre de l'information codée sous forme numérique, les altérations des messages sont beaucoup moins tolérables. Nous allons dire quelques mots très sommaires de la théorie de l'information.

6.1.1 Théorie de l'information [3]

Le transfert d'information dans un système de communication s'effectue par *messages*. Un message est une suite de signes (de symboles) tirés d'un *alphabet*.

1. Cf. Wittgenstein, *Tractatus logico-philosophicus*, [141, aphorisme 4.002].

2. Michel Volle me fait remarquer que cette expression consacrée par l'usage est malheureuse, il serait plus approprié de dire « théorie de la communication », conformément d'ailleurs au titre de l'article de Shannon, *A mathematical theory of communication*, (1948) [122], ou « théorie des données ». L'ordinateur, ou le réseau de communication, contiennent et transmettent des données, qui ne deviennent de l'information que dans l'esprit d'un être humain, capable (et lui seul en est capable) de leur donner un *sens*. Voir à ce sujet le texte de Michel Volle [135].

3. Les quatre alinéas qui suivent sont empruntés à mon livre *Initiation à la programmation avec Scheme*, publié en 2011 par les Éditions Technip, avec l'aimable autorisation de l'éditeur.

L'ensemble $S = \{m_1 ... m_i ... m_n\}$ des messages que l'on peut former à l'aide d'un alphabet donné constitue une *source (discrète) de messages* : un *texte* est une suite de messages.

La notion d'information est liée à l'ignorance du destinataire quant aux messages émis depuis S : il n'y a apport d'information que si le destinataire ignore le contenu du message qu'il va recevoir. L'incertitude, quant à la teneur du message, est d'autant plus grande que le message a une faible probabilité d'être émis ; inversement, la réception de ce message contribue à lever une incertitude d'autant plus grande, et apporte donc une plus grande quantité d'information ; ainsi apparaît une relation entre la quantité d'information d'un message m_i et sa probabilité d'émission, p_i, relation représentée par la fonction logarithmique suivante[4] :

$$I(m_i) = \log_\alpha(1/p_i) = -\log_\alpha p_i$$

I étant l'incertitude, ou l'information, α le nombre de symboles de l'alphabet utilisé. On en déduit immédiatement que la valeur de l'unité d'information est celle d'un message de probabilité $\frac{1}{\alpha}$. On prend généralement $\alpha = 2$, l'unité correspondante étant le bit.

Pour donner un contenu plus facile à retenir à la formule ci-dessus, on peut utiliser les logarithmes décimaux, et regarder la quantité d'information transmise par un message émis avec une certaine probabilité :

probabilité d'émission	quantité d'information
$p_1 = 1$	$I(m_1) = \log_{10}(1/p_1) = -\log_{10} 1 = 0$
$p_2 = 0{,}1$	$I(m_2) = \log_{10}(1/p_2) = -\log_{10} 10^{-1} = +1$
$p_3 = 0{,}01$	$I(m_3) = \log_{10}(1/p_3) = -\log_{10} 10^{-2} = +2$
...	...

Cette définition probabiliste de la quantité d'information d'un message montre qu'elle dépend avant tout de la source de messages utilisée ; cette dernière peut être caractérisée par une quantité d'information (ou incertitude) moyenne, d'après l'expression :

$$H(S) = -\sum_{i=1}^{n} p_i \times \log_\alpha p_i$$

4. On rappelle que par définition le logarithme de base α de x, noté $\log_\alpha x$, est le nombre m tel que $\alpha^m = x$. On vérifie que $\log_\alpha \alpha = 1$ et que, indifféremment à la valeur de α, $\log 1 = 0$ et $\log(a \times b) = \log a + \log b$. Cette dernière propriété, très intéressante puisqu'elle permet, avec une table de logarithmes, de faire des multiplications même si on ne connaît que la table d'addition, est (était ?) le principe de base de la règle à calcul, qui n'est pas autre chose qu'une table de logs en plastique.

qui permet d'évaluer a priori la quantité moyenne d'information que peut fournir un message ; sa valeur est maximale pour des messages équiprobables. Cette grandeur a la même forme que l'entropie thermodynamique et on l'appelle entropie de S.

L'entropie permet d'évaluer la richesse informationnelle d'un texte ; Shannon a montré que si l'information moyenne d'un alphabet de 27 signes équiprobables était : $\log_2 27 = 4{,}75$ bits/lettre, le contenu d'un texte anglais ordinaire n'était que de 1 bit/lettre, soit une redondance de : $1 - \frac{1}{4{,}75}$, ou 80% de signes inutiles.

6.1.2 Premières réalisations

La transmission de données entre calculateurs a précédé l'invention de l'ordinateur proprement dit. En 1940 George Robert Stibitz travaillait depuis déjà quelques années à la conception et à la réalisation de calculateurs à relais électromécaniques pour les *Bell Telephone Laboratories*. Il avait organisé une démonstration de sa machine au Dartmouth College, dans le New Hampshire, où se tenait le congrès de la Société américaine de mathématiques, alors que le calculateur était à New York, à 330 km de là. Le matériel était très encombrant et le déménager compliqué et aléatoire : Stibitz décida de réaliser la démonstration à partir d'un télétype relié par une ligne téléphonique à la machine, ce qui fut fait le 11 septembre 1940 avec un total succès. Dès lors le « problème de base » pouvait être considéré comme résolu. On savait faire communiquer à distance deux machines à traiter de l'information.

Dans ce cas précis le support matériel de la communication était une ligne de téléphone en fils de cuivre, comme on en utilise encore aujourd'hui, mais les réseaux informatiques peuvent utiliser toutes sortes de supports matériel sans que cela modifie la nature du problème à résoudre : fibre optique, faisceau hertzien, canal satellite, câble coaxial, rayons infrarouge, signal laser etc. Il suffit que les équipements à chaque extrémité soient configurés correctement, de façon cohérente, ce qui d'ailleurs n'est pas une mince affaire.

6.1.3 Un modèle pour les réseaux

Considérons comme acquise la solution du problème de base, faire communiquer à distance deux machines à traiter de l'information. Avons-nous pour autant un réseau de telles machines ? Sans doute non. Le problème de l'acheminement d'un message dans un réseau complexe se compose de plusieurs sous-problèmes. Un groupe d'experts de l'ISO (Organisation Internationale de Normalisation) réuni de 1977 à 1986 sous la conduite d'Hubert Zimmerman a classé les sous-problèmes selon une échelle allant du plus concret (support physique de la communication) au plus immatériel (le logiciel de communication avec l'utilisateur). Il en est résulté un modèle en couches

nommé OSI (pour *Open Systems Interconnexion*) conforme au principe exposé à la section 1.4, où la couche la plus basse correspond aux questions liées au support physique, et la plus haute au logiciel de contact avec l'utilisateur final ; nous allons examiner dans les sections suivantes les sous-problèmes selon la couche du modèle qui leur correspond.

Avant d'entamer la description du contenu et des fonctions de chaque couche du modèle OSI de l'ISO, il convient de préciser le vocabulaire employé pour décrire les réseaux. Un réseau sert communément à relier un certain nombre de points. De façon générale, un certain nombre de points reliés par des lignes constituent un graphe. Les points reliés sont appelés sommets (*vertex* en anglais) du graphe, et la ligne qui relie deux points est appelée arc (*edge* en anglais). Si l'arc qui relie un sommet **A** à un autre, **B** par exemple, relie également **B** à **A**, on dit que le graphe n'est pas orienté ; dans les graphes orientés les arcs ont un sens ; les graphes dont nous parlerons sont non orientés. La figure 6.1 représente un graphe non orienté $\mathcal{G}$ à sept sommets.

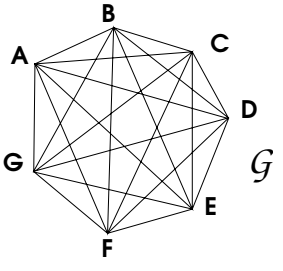


Figure 6.1 : *Graphe connexe complet*

Le graphe $\mathcal{G}$, ou **ABCDEFG**, est tel qu'entre deux quelconques de ses sommets il existe au moins un chemin constitué d'arcs du graphe : un tel graphe est dit *connexe*. De surcroît, chaque sommet est relié par un arc à chacun des autres : c'est un graphe connexe *complet*. Chaque sommet est relié par un arc à chacun des $n-1$ autres sommets, et chaque arc joue le même rôle pour les deux sommets qu'il relie, $\mathcal{G}$ possède donc $\frac{n\times(n-1)}{2}$ arcs.

La figure 6.2 représente un graphe $\mathcal{H}$ à sept sommets simplement connexe. Lorsque l'on parle de réseaux informatiques, il peut être commode de les représenter par des graphes. Les sommets sont alors généralement appelés nœuds du réseau, et les arcs, lignes ou liaisons.

6.2 Couche 1, physique

Étant donné un support physique approprié reliant deux matériels de traitement d'information, dits *équipements terminaux,* le premier sous-problème

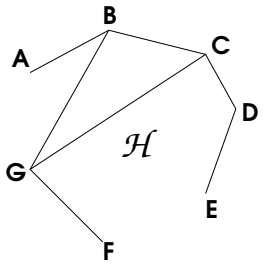


Figure 6.2 : *Graphe simplement connexe*

consiste à établir les conditions pour que chacun de ces deux équipements puisse émettre et recevoir des signaux en provenance de et destinés à l'autre. Il s'agit du « problème de base » vu plus haut, que G. Stibitz a été le premier à résoudre, et dont Nyquist et Shannon ont posé les bases théoriques, notamment en donnant le débit maximum d'information que peut assurer un canal donné.

En restant schématique, l'envoi d'informations se fait en modulant une onde hertzienne, ou en modifiant la tension électrique appliquée aux bornes d'un conducteur. La modification de la phase de l'onde ou de la tension est un signal élémentaire, en d'autres termes un bit, une unité d'information. Une telle modification ne peut intervenir qu'un certain nombre de fois par seconde, selon les caractéristiques du matériel. Cette fréquence maximum de variation définit la largeur de bande du canal, qui limite la quantité d'information qu'il peut acheminer.

La solution du sous-problème de la couche 1 fait appel à la physique, à l'électronique, au traitement du signal plus qu'elle ne constitue une question d'informatique à proprement parler, et nous ne nous y attarderons guère. Signalons simplement quels sont les types de supports les plus couramment utilisés pour édifier des réseaux informatiques :

- Pour construire un réseau privé dans un immeuble ou sur un campus (LAN, pour *Local Area Network*), la solution classique consiste à poser de la fibre optique dès que les distances sont un peu grandes, de la paire torsadée téléphonique pour le câblage dit capillaire qui alimente les prises terminales. Les systèmes sans fil (hertzien ou infrarouge) sont aussi utilisés. Les débits peuvent atteindre les 100 milliards de bits par seconde (100 Gigabit/s).
- Les réseaux à longue distance ou WAN (pour *Wide Area Network*), comme ceux des fournisseurs d'accès à l'Internet (FAI), ont recours à plusieurs types de solutions : location de lignes aux opérateurs de télécommunications ou aux propriétaires de réseaux d'autre nature (compagnies de chemin de fer, de distribution d'électricité, de métro...), construction

d'infrastructures propres. Physiquement, la liaison est presque toujours de la fibre optique. Les débits sont de l'ordre du milliard de bits par seconde. Le satellite est peu utilisable à cause du délai de propagation, qui est d'un quart de seconde aller et retour (pour atteindre les antipodes il faut deux rebonds sur des satellites, soit une demi-seconde).

– Pour l'accès des particuliers à leur FAI, l'usage de modems et du réseau téléphonique commuté ne disparaîtra pas tout de suite. Les accès dits à haut débit (câble TV ou ADSL sur réseau téléphonique, de plus en plus souvent fibre optique) procurent un débit de l'ordre de un à plusieurs millions de bits par seconde (Mégabit/s).

Tous ces débits sont donnés à titre indicatif et très provisoire. Signalons aussi que dans certains cas la couche physique est capable d'acheminer des signaux simultanément dans les deux sens.

6.3 Notion de protocole

Pour acheminer un flux de données sur un canal de transmission, les systèmes matériels et logiciels qui composent le réseau utilisent un ensemble de règles, de conventions et de mises en forme des données qui constituent un *protocole* de communication. Une des fonctions des protocoles de communication est la détection et la correction des erreurs de transmission, qui constituent un des problèmes classiques des réseaux informatiques.

Les canaux de communication tels que les lignes téléphoniques sont soumis à des erreurs aléatoires, qui constituent le bruit. Pour une conversation cela provoque juste un parasite sonore désagréable, mais dans le cas de la transmission de données ce sera une altération de nature à rendre le message inutilisable. Il faut donc instaurer des moyens de vérifier l'intégrité des données et, quand c'est possible, de la restaurer quand elle a été altérée. Ceci devra être fait pour chaque couche du protocole de transmission, mais c'est spécialement important pour la couche 2, dont le rôle est de garantir aux couches supérieures un canal de transmission sans erreur d'un flux de bits. Le traitement de ce problème peut différer selon qu'il s'agit de liaisons point à point ou de diffusion.

Le protocole définit également, pour chaque couche, d'autres caractéristiques de la transmission : les messages sont généralement découpés en unités de taille homogène (appelés *trames* pour la couche 2 et *paquets* pour la couche 3), les nœuds reçoivent lorsque c'est nécessaire des *adresses* qui permettent, comme celles des personnes, de les trouver et de les identifier dans le réseau.

Nous pouvons nous représenter un protocole de communication comme un protocole au sens habituel ; quand deux entités du réseau entament une communication, l'échange se passe un peu comme ceci :

« Attention, je vais t'envoyer un message de plusieurs pages, es-tu prêt à le recevoir ?

« — Oui, je suis prêt.

« — Tu es sûr ?

« — Oui, je suis prêt.

« — Je vais envoyer.

« — Vas-y.

« — (Envoi de la première page, attachée à une flèche tirée à l'arc).

« — Bien reçue.

« — (Envoi de la seconde page, enroulée autour d'un caillou lancé avec une fronde).

« — Bien reçue.

« — (Envoi de la troisième page, confiée à un pigeon voyageur).

« — Bien reçue.

« — Normalement tu as dû recevoir trois pages numérotées de 1 à 3. Vérifie que tous les numéros sont là et dans le bon ordre.

« — Oui, j'ai bien reçu toutes les pages dans le bon ordre. »

Comme signalé à la section 1.4, dans un modèle en couches les protocoles définissent les règles du dialogue entre couches de même niveau sur des systèmes différents, cependant que les interfaces spécifient les services qu'une couche inférieure fournit à la couche qui lui est immédiatement supérieure au sein du même système.

6.4 Couche 2, liaison de données

La couche 2 du modèle de transmission de données, appelée couche de *liaison de données*, assure la transmission fiable d'un flux de bits entre deux nœuds adjacents du réseau sur un support physique procuré par la couche 1. Quand on dit adjacents, il faut entendre que les deux équipements terminaux sont connectés par un canal de transmission qui peut être vu comme un fil, c'est à dire que les bits émis à une extrémité sont délivrés exactement dans le même ordre à l'autre extrémité. Le travail de la couche 2 consiste à faire en sorte qu'ils soient également transmis sans omission ni déformation et remis à la couche 3. La figure 6.3 représente cette coopération des fonctions fournies par chaque couche[5].

5. Sans trop entrer dans les détails, signalons que la couche de liaison de données comporte deux sous-couches, dont l'une est la couche MAC (*Medium Access Control*, ou commande de l'accès au support physique). Dans le cas des réseaux locaux *(Local Area Networks, LAN's)*, la couche 2 se réduit pratiquement à la sous-couche MAC, et de ce fait on rencontre les expressions couche MAC, adresse MAC etc.

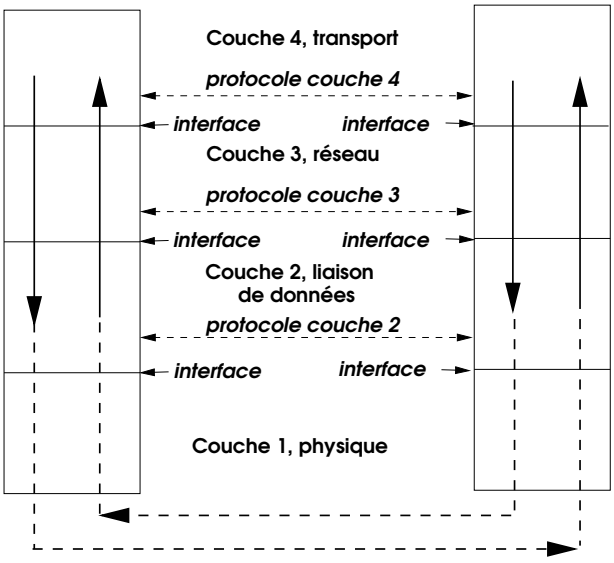


Figure 6.3 : *Les trois couches basses*

La liaison peut être établie de deux façons. La plus simple est la liaison point à point, c'est celle que vous utilisez quand vous vous connectez de votre domicile à l'Internet avec un modem. Le chemin entre votre modem et le modem de votre fournisseur d'accès est une liaison point à point, d'ailleurs gérée par le protocole PPP (*Point to Point Protocol*, comme son nom l'indique), c'est à dire que vous pouvez considérer ce lien logiquement comme un fil unique, même si le réseau téléphonique traversé est, du point de vue téléphonique, complexe. Derrière le modem du fournisseur d'accès (FAI), il y a bien sûr un réseau complexe qui n'est sans doute plus point à point. La liaison point à point est généralement le procédé utilisé pour les liaisons à longue distance qui constituent un réseau étendu, ou WAN (pour *Wide Area Network*), par opposition à un réseau local (LAN, pour *Local Area Network*), c'est-à-dire cantonné à un immeuble ou à un campus. Ces notions de longue distance et de localité sont toutes relatives, par exemple les réseaux d'accès à l'Internet des distributeurs de télévision par câble sont des réseaux locaux à l'échelle d'une ville comme Paris (on parle alors de réseau métropolitain).

Un réseau local est constitué différemment : un assez grand nombre de nœuds partagent une même infrastructure de câblage[6], par exemple dans un bâtiment Cette infrastructure constitue un unique réseau d'acheminement, qui offre notamment un service de couche 2. Le réseau d'acheminement réunira

6. ... ou de transmission sans fil.

un ou plusieurs réseaux de couche 2 et accédera à un réseau plus vaste de couche 3, par exemple l'Internet, par l'intermédiaire d'un ordinateur spécialisé appelée *passerelle de couche 3*, ou plus souvent *routeur*. Mais examinons pour l'instant les réseaux de couche 2. Nous pouvons les considérer comme des graphes connexes complets conformes à la figure 6.1, c'est-à-dire que chaque nœud peut « parler » directement à tous les autres nœuds du même réseau (de couche 2). Ces réseaux utilisent généralement la diffusion *(broadcast)*, c'est-à-dire que l'émetteur envoie les signaux à tous les nœuds du réseau, à charge pour le destinataire de reconnaître ceux qui lui sont destinés par un procédé d'adressage décrit à la section suivante. C'est le cas des réseaux locaux de type Ethernet.

6.4.1 Notion d'adresse réseau

Dans le cas d'une liaison point à point, identifier le destinataire d'un message n'est pas difficile puisqu'il n'y a que deux nœuds en cause: l'émetteur à une extrémité, le récepteur à l'autre. Dans le cas de la diffusion, l'émetteur d'un message doit désigner le destinataire. Et au niveau de la couche 3, le réseau est complexe et contient généralement de nombreux nœuds qu'il faut identifier et atteindre. À cette fin chaque nœud se voit attribuer une *adresse réseau*. Aux couches 2 et 3 correspondent des visions différentes du réseau, et de ce fait elles possèdent des systèmes d'adressage distincts.

Adresse de couche 2

L'architecture des réseaux Ethernet, dont notamment leur système d'adressage, a été conçue par Xerox et réalisée par Xerox, Intel et Digital Equipment. Plus tard, l'IEEE *(Institute of Electrical and Electronics Engineers)* a promulgué la norme 802.3 censée décrire Ethernet, mais qui introduit quelques divergences qui obligent les industriels à s'adapter aux deux variantes, ce qui n'est pas trop compliqué.

Pour qu'un ordinateur puisse être connecté à un réseau Ethernet, il faut qu'il possède une interface matérielle qui se présente généralement sous la forme d'une carte dite carte réseau, ou NIC *(Network Interface Card)*. Un ordinateur peut posséder plusieurs cartes réseau, pour être relié plusieurs fois au même réseau, ou à plusieurs réseaux de couche 2 différents, ce qui lui permet éventuellement de jouer le rôle de routeur, c'est-à-dire de faire passer les données d'un réseau à un autre.

Chaque interface réseau reçoit à sa fabrication une adresse de couche 2, dite « adresse MAC » (MAC pour *Medium Access Control*) ou adresse Ethernet, ou encore adresse physique (parce qu'associée à un dispositif matériel), de 48 bits de longueur. L'unicité à l'échelle mondiale des adresses MAC est assurée

par l'IEEE, qui attribue à chaque industriel producteur un préfixe, charge à lui de gérer l'unicité des chiffres suivants. Ainsi l'adresse MAC de l'ordinateur que j'utilise en ce moment est `00:48:54:C0:C9:04`, où chaque groupe de deux caractères isolé par « : » représente un octet codé en hexadécimal, soit une valeur décimale comprise entre 0 et 255 (voir annexe A section A.4.1 pour les détails de cette représentation). Ceci assure en principe qu'il n'y aura pas sur le réseau de dysfonctionnements provoqués par deux interfaces ayant la même adresse, mais en fait il est possible de modifier une adresse MAC dynamiquement par logiciel. Cela dit les logiciels raisonnables (et non piratés) ne font pas cela.

6.4.2 Détection et correction d'erreur pour la couche 2

Découpage en trames *(framing)*

La transmission de la voix par un procédé analogique se fait selon un flux continu de signaux, à l'image de la parole. La nécessité du contrôle d'intégrité des données impose pour la transmission de données numériques de procéder autrement. Pour les locuteurs de langues non écrites, la parole se manifeste comme un flux continu, et la conscience de son découpage en mots n'est pas de la même nature que pour les locuteurs de langues écrites qui ont dû apprendre à procéder mentalement à ce découpage (voir [73] pour de plus amples éclaircissements). De façon analogue, le flot de bits, que la couche 1 est prête à fournir à la demande, est découpé par les protocoles de couche 2 en entités discrètes de longueur limitée appelées *trames* (en anglais *frame*).

Le découpage du flot de données en trames est en fait plutôt une image : il nous faudra un moyen de reconnaître le début et la fin d'une trame. Les solutions raisonnables sont :

- séparer les trames par des intervalles de « silence » ; cette solution est employée pour les réseaux locaux parce que le débit est élevé et la bande passante disponible à profusion ;
- utiliser des trames de longueur soit fixe, soit variable mais connue en cours de transmission, et compter les bits ; cette solution et la suivante sont employées notamment pour les liaisons à longue distance ;
- utiliser des configurations de bits particulières et inutilisées par le codage des données pour marquer le début et la fin de trames, et guetter leur occurrence, ce qui est la solution la plus répandue, éventuellement combinée au comptage de bits.

Détection de trames endommagées

Les trames seront les entités dont les procédures de détection d'erreur vérifieront l'intégrité. Nous considérons ici, dans un premier temps, le traitement des

trames qui arrivent à destination mais dont le contenu a été altéré par un parasite quelconque.

Le principe de base de la détection de ce type d'erreur est la redondance : avant d'émettre une trame, la station émettrice ajoute au message à transmettre (ici le contenu de la trame) une information supplémentaire calculée à partir des bits du message selon un algorithme dit de hachage *(hash)*. À la réception, la station réceptrice effectue le même calcul ; si elle ne trouve pas le même résultat c'est qu'il y a eu une erreur. Cette information supplémentaire calculée à partir de l'information utile s'appelle *somme de contrôle* (en anglais *checksum*). L'algorithme de calcul de la somme de contrôle doit bien sûr être le même aux deux extrémités : cette convention fait partie du protocole. Une méthode très répandue est le code de redondance cyclique (CRC), dont nous ne donnerons pas le calcul ici.

Si le calcul prévu par la procédure donne le résultat attendu, il n'y a pas d'erreur et alors, dans le cas des réseaux longue distance (WAN), le protocole de couche 2 (côté récepteur) envoie un acquittement (conventionnellement ACK) à l'émetteur ; sinon il envoie par exemple un acquittement négatif (NAK) qui demande à l'émetteur de retransmettre la trame considérée. Pour savoir quelle trame est acquittée, le protocole prévoit aussi que chaque trame comporte un numéro de séquence permettant de la distinguer des précédentes et des suivantes. Ethernet procède sans échange d'acquittements : les détections d'erreur sont signalées par un signal de couche 1.

Il existe aussi des codes auto-correcteurs, dont le plus célèbre est le code de Hamming : au prix de plus de redondance, ces codes permettent de connaître précisément les positions des bits erronés, s'il n'y en a pas trop, et partant de les corriger. Cela semble séduisant mais n'est pas tellement utilisé, parce qu'en pratique on observe que les trames sont le plus souvent soit intactes, soit trop fortement endommagées pour être corrigées. De plus les trames endommagées sont rares sur la plupart des réseaux modernes, où les taux d'erreur sont de l'ordre de 10^{-6}, voire moins[7]. Il est plus efficace de retransmettre les données erronées.

Contrôle de flux

Les contrôles d'erreur évoqués à la section précédente faisaient l'hypothèse que les délais de transmission d'une trame et de l'acquittement en sens inverse étaient négligeables, ce qui est vrai pour un réseau local mais beaucoup moins pour une liaison à longue distance. Un protocole de couche 2 doit également se

7. Ceci est vrai des réseaux locaux et des réseaux téléphoniques des pays riches, maintenant presque toujours numériques. Ce ne l'est pas dans les pays dotés de réseaux téléphoniques et électriques de mauvaise qualité, qui de ce fait ont parfois intérêt à continuer à utiliser des protocoles moins rapides mais plus robustes.

prémunir contre un autre risque : si un émetteur a une interface réseau beaucoup plus rapide que celle du récepteur, le récepteur ne pourra pas absorber toutes les trames qui lui sont envoyées et des données vont se perdre. Il faut donc un algorithme pour réguler les transmissions, et notamment s'assurer que les trames envoyées sont bien reçues.

La solution évoquée à la section précédente, qui consiste, avant d'envoyer une nouvelle trame, à attendre d'avoir reçu un acquittement positif pour la précédente, répond à la question, mais impose un ralentissement insupportable et une utilisation extrêmement inefficace de la bande passante. Cette inefficacité est particulièrement grande pour les liaisons par satellite géostationnaire, qui peuvent avoir un débit élevé mais un temps de transit incompressible de l'ordre d'un quart de seconde : s'il fallait attendre l'acquittement, on aurait un débit de deux trames par seconde, ce qui ne serait évidemment pas acceptable.

Pour améliorer cette situation, les protocoles adaptés aux liaisons à délai de transit important utilisent un algorithme basé sur le principe suivant : l'émetteur n'attend l'acquittement de la trame numéro n qu'après l'émission de la trame $n + p$, avec $p > 1$. Le délai nécessaire à la transmission de p trames est appelé *délai de garde (timeout interval)*. Cette méthode est appelée *pipeline*, parce que l'on enfourne les trames dans le « tuyau » sans attendre que les précédentes soient sorties. Comme à la section précédente, chaque trame est dotée d'un numéro de séquence qui permet de savoir, notamment, quelle trame acquitte tel ACK (dans le cas des protocoles WAN).

Mais alors, que va-t-il se passer si une trame, au milieu d'un long message, est corrompue ou n'arrive pas ? Rappelons que la couche 2 a pour mission de délivrer les trames **dans l'ordre** à la couche 3. Un tel cas est illustré par la figure 6.4, où nous supposerons $p = 3$. Soit le cas d'école suivant : la trame 2 est émise, mais perdue ou détériorée en route. Le récepteur détecte le problème sans coup férir : si la trame est détériorée, par une des méthodes de détection d'erreur indiquées à la section précédente 6.4.2 ; si elle est perdue en route, le contrôle des numéros de séquence montre que la trame 1 devrait être suivie de la trame 2, or il reçoit à la place la trame 3 (ou une autre…), qui ne satisfait pas aux règles du protocole.

Dès que le récepteur constate la défaillance de la trame 2, il rejette imperturbablement toutes les trames que l'émetteur continue à lui envoyer. L'émetteur va-t-il s'en apercevoir ? Oui : après p émissions, soit après la trame $n+p = 2+3 = 5$, il s'attend à recevoir l'acquittement de la trame 2, attente déçue. Que va-t-il faire alors ? Il va simplement réémettre la trame 2, puis toutes ses suivantes. Le récepteur va enfin recevoir la trame 2 attendue, il va l'accepter et l'acquitter, ainsi que les suivantes.

Combien de trames ont été perdues ou rejetées ? $p + 1 = 4$. Pour que l'algorithme soit correct, il faut que l'émetteur garde en mémoire au moins les $p+1$ dernières trames émises, afin de pouvoir les réémettre. Plus p sera grand, plus le protocole sera rapide, mais plus il faudra de mémoire dans les équipements

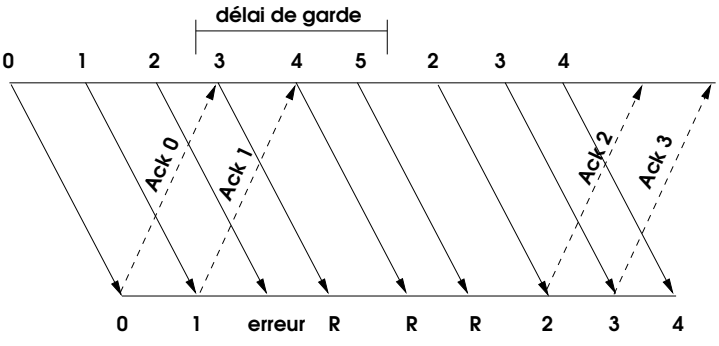


Figure 6.4 : *Fenêtre glissante*

de transmission, ce qui est un compromis constant pour les performances des ordinateurs et autres machines à traiter de l'information.

Du côté du récepteur, notre algorithme rejette toutes les trames consécutives à la trame détériorée : on pourrait imaginer de conserver les trames correctes qui arrivent après la trame détériorée. Ainsi lorsqu'à l'expiration du délai de garde l'émetteur constaterait qu'une trame n'a pas été acquittée, il n'aurait que celle-là à retransmettre. Bien sûr, tant que le récepteur n'a pas reçu la trame détériorée, il ne peut pas remettre les suivantes à la couche réseau, il doit donc les garder dans sa mémoire de travail (une telle zone de mémoire est appelée communément *buffer*). Le nombre de trames que le récepteur peut ainsi conserver définit la largeur d'une *fenêtre de réception*. Notre exemple initial, où toutes les trames consécutives à l'erreur étaient rejetées, correspond à une fenêtre de largeur 1. Le nombre de trames que l'émetteur s'impose de garder en mémoire en attendant leur acquittement s'appelle la fenêtre d'émission. Cet algorithme est nommé *protocole de la fenêtre glissante*.

La largeur des fenêtres, en émission et en réception, peut varier. Cet algorithme est dit de *contrôle de flux* : si l'émetteur constate que sa fenêtre d'émission est toujours vide, il peut augmenter son débit si c'est possible. Si le récepteur ne peut pas épuiser la fenêtre de réception, la fenêtre d'émission va se remplir et l'émetteur sera obligé de s'interrompre.

Ce protocole de fenêtre glissante, que nous venons de décrire pour la couche 2, est également utilisé pour la couche 4 de transport (TCP).

6.4.3 Un exemple de liaison de données : Ethernet

Si vous utilisez un réseau local à votre domicile ou sur votre lieu de travail, il y a de fortes chances que ce soit un réseau Ethernet. Cette technologie a supplanté la plupart des concurrentes, et de nos jours une carte réseau de ce type vaut une quinzaine d'Euros. L'auteur se rappelle la première « carte » Ethernet qu'il

a achetée (pour le compte de son employeur) : elle avait coûté l'équivalent de 100 000 Euros et il avait fallu abattre une cloison pour loger l'armoire d'extension de 50 cm de large, 120 cm de hauteur et 70 cm de profondeur nécessaire à son installation. C'était un matériel DEC connecté à un VAX 11/750 sous VMS, en 1984.

Ethernet a été inventé au PARC *(Palo Alto Research Center)* de Xerox en 1973 par Robert Metcalfe et David Boggs. Le lecteur trouvera dans le livre de réseau de Tanenbaum [130] une description historique et technique détaillée. La première version commercialisée a été introduite en 1980 par Xerox, Intel et DEC. Le protocole a été normalisé par l'IEEE en 1983 sous la référence 802.3, toujours en vigueur.

Le nom Ethernet est un hommage à un ancêtre de ce protocole, ALOHA, inventé à l'Université d'Hawaï en 1970 par Norman Abramson. Comme Abramson voulait relier les campus de l'université, situés sur des îles différentes, ALOHA était un protocole de réseau par radio, les communications circulaient dans l'éther.

Selon ALOHA, toutes les stations émettent et reçoivent sur la même bande de fréquence. Les messages sont découpés en trames, identifiées par un numéro d'ordre et l'adresse de la station destinataire. C'est une conversation à plusieurs : toutes les stations reçoivent toutes les trames, identifient celles qui leur sont destinées, jettent les autres.

La communication par radio entre sites distants interdit toute idée de contrôle centralisé : ALOHA doit prévoir le cas où deux stations (ou plus) voudraient commencer à émettre en même temps. Cette circonstance est nommée *collision*, et résulte en trames brouillées, incompréhensibles, en un mot perdues. Il va falloir réémettre ces trames perdues, et si possible en évitant de provoquer une nouvelle collision, sinon le protocole n'aboutira jamais.

Pour diminuer le risque de nouvelle collision, ALOHA utilise un algorithme probabiliste : chacune des stations qui a émis une trame perdue à cause de la collision calcule un nombre aléatoire, en déduit un intervalle de temps et réémet. La probabilité que deux stations aient calculé le même délai est très faible ; si néanmoins c'est le cas une nouvelle collision a lieu, et il faut réitérer le calcul. La probabilité que deux stations calculent trois fois de suite le même délai est tellement faible que cet algorithme, en pratique, fonctionne très bien. Il peut être encore amélioré au moyen d'une horloge centrale qui émet un signal dont la période est égale au délai de transmission d'une trame à travers le réseau : les trames ne peuvent être émises qu'au « top » d'horloge donné par ce signal. Cette discrétisation des émissions améliore l'efficacité du réseau en diminuant l'intervalle de temps minimum avant réémission en cas de collision.

Ethernet utilise le même principe qu'ALOHA : le support physique du réseau est accessible par toutes les stations simultanément, qu'il s'agisse d'un câble coaxial comme dans les années 1980, de liaisons en paires torsadées dans les années 1990, de fibre optique, d'un réseau hertzien, comme la mode (fort pratique

mais avec quelques problèmes de sécurité induits) s'en répand en ce début de millénaire, de liaisons infra-rouges, etc. Il y a donc des collisions. Ethernet apporte deux perfectionnements par rapport à ALOHA :

- les stations écoutent le trafic sur le câble avant d'émettre et, si le câble est occupé, observent un délai avant de réessayer ;
- si une collision se produit, les émetteurs la détectent immédiatement et cessent d'émettre.

Ces améliorations confèrent au protocole Ethernet le qualificatif CSMA-CD (*Carrier Sense Multiple Access with Collision Detection*, ou accès multiple avec écoute de la porteuse et détection de collision).

Les aspects non déterministes, probabilistes, du protocole Ethernet ont pu inquiéter : et s'il y a sans arrêt des collisions, alors le réseau sera bloqué ? Les concurrents ont bien sûr mis à profit ces angoisses pour faire la promotion de leurs protocoles déterministes, beaucoup plus lourds et moins efficaces, tel *Token ring* d'IBM, aujourd'hui bien oublié. Le succès d'Ethernet, qui a été total, est un argument de plus pour croire aux résultats du calcul des probabilités.

La convergence des algorithmes d'Ethernet impose un certain nombre de contraintes sur la topologie et l'organisation physique des réseaux. Il faut notamment qu'une trame puisse traverser le réseau aller et retour selon son diamètre (c'est-à-dire entre les deux nœuds les plus « éloignés » l'un de l'autre) dans un délai aller et retour de 51,2 μs. Pour la première version d'Ethernet sur coaxial à 10 Mbps (mébibits par seconde), dite 10Base5, cette condition imposait qu'il n'y ait pas plus de 2,5 km de câble et pas plus de 4 répéteurs entre les nœuds. Un répéteur est un matériel de couche 1 qui possède plusieurs interfaces connectées chacune à un segment de réseau et qui se contente de recevoir, amplifier et retransmettre les signaux sur toutes ses interfaces ; il peut ainsi étendre le réseau à plusieurs segments de câble ; les répéteurs sont aujourd'hui le plus souvent abandonnés au profit des commutateurs. Typiquement, les réseaux Ethernet sur paires torsadées sont organisées selon une topologie en étoile autour d'un commutateur *(switch)* . La leçon pratique à retenir est que s'il est commode d'étendre son réseau en multipliant des commutateurs en cascade (comme des prises multiples pour l'électricité), il vient un moment où l'on crée des boucles, et alors le réseau se met à adopter des comportements erratiques et non souhaités : ce n'est pas forcément une panne franche, par moments cela marche, par moments non, c'est-à-dire que c'est une panne difficile à diagnostiquer.

Un commutateur, au lieu de propager le signal sur toutes les interfaces, ne l'envoie que sur celle qui correspond à l'adresse de destination de la trame reçue. Comment le commutateur connaît-il l'adresse de destination de la trame ? Par apprentissage : lorsqu'il reçoit une trame pour une station qui ne s'est jamais manifestée, il agit comme un répéteur et la diffuse à toutes ses interfaces. Dès qu'une

station s'est manifestée il conserve son adresse dans une table et n'envoie qu'à elle les trames qui lui sont destinées. Cette nouvelle organisation permet accessoirement de diminuer considérablement le nombre de collisions, puisqu'ainsi seules les stations concernées écoutent le trafic qui leur est destiné. Cela améliore aussi la sécurité. Pourquoi alors ne pas l'avoir fait plus tôt ? Parce qu'il a fallu du temps pour pouvoir fabriquer à des coûts raisonnables une électronique suffisamment complexe et rapide : échantillonner un signal à 10 MHz (et maintenant 100 MHz pour le FastEthernet, 1 GHz pour le GigabitEhernet, demain 10 GHz) et analyser à la volée l'adresse MAC de destination semble banal, mais ne l'était pas au début des années 1980.

Signalons aussi que la fibre optique, support très rapide, permet d'outrepasser la règle des quatre répéteurs, ce qui en fait un support de choix pour les épines dorsales *(backbones)* de réseaux de campus, voire de métropole (*Metropolitan Area Networks*, ou MAN) comme le Réseau Académique Parisien, qui reliait naguère plusieurs universités et centres de recherches parisiens au moyen de fibres optiques posées dans les tunnels du métro.

Saisissons l'occasion de cette section consacrée à Ethernet pour signaler le nombre considérable d'innovations capitales dont l'origine se trouve au PARC créé en 1970 par Xerox : Ethernet pour commencer. Le langage de description de page PostScript : les fondateurs d'Adobe ont quitté Xerox, frustrés de voir leur employeur ne rien faire de leur invention. L'impression laser vient aussi du PARC (1971). Les interfaces à fenêtres popularisées par le Macintosh ont été inventées par Alan Kay à la fin des années 1960 alors qu'il travaillait au langage Smalltalk, d'abord à l'Université d'Utah, puis à l'Université Stanford, enfin au PARC de 1972 à 1983, où Steve Jobs le rencontra et conçut le projet d'un ordinateur Apple sur la base de ces idées. Incidemment, si la programmation par objets avait déjà été inventée en 1965 par l'équipe de Simula, Alan Kay et le PARC ont beaucoup contribué à son succès avec *Smalltalk*. Bref, que serions-nous sans eux. Une des explications de cette fécondité extraordinaire dont Xerox a finalement tiré peu de gains financiers est que le PARC avait été créé essentiellement pour dépenser à fonds perdus une partie des bénéfices énormes engendrés par les brevets de la photocopie à sec, afin d'éviter qu'ils n'engendrent des impôts également énormes. La rentabilité était un non-objectif.

6.5 Couche 3, réseau

Dès que notre réseau comportera un nombre n d'équipements terminaux supérieur à 2 ou 3, il ne sera plus raisonnable de les relier deux à deux par des supports physiques dont le nombre serait égal, nous l'avons vu, à $\frac{n\times(n-1)}{2}$, même s'il s'agit de faisceaux hertziens. Il faut donc acheminer les messages par un trajet complexe qui passe par plusieurs segments de liaison de données.

Ce sous-problème suppose résolu le précédent (en d'autres termes, la couche 3 fonctionne en utilisant les services fournis par la couche 2) et constitue le problème du réseau proprement dit.

6.5.1 Commutation de circuits

Une première solution consiste à établir un circuit entre deux équipements en sélectionnant un certain nombre de segments de câble (par exemple) qui, mis bout à bout, constituent un itinéraire de l'un à l'autre. Aux jonctions des segments devront se trouver des *équipements de transmission de données* capables de jouer le rôle de poste d'aiguillage pour les signaux échangés. Nous aurons ainsi constitué un circuit physique, dit *circuit commuté*, qui une fois établi peut être considéré comme l'équivalent d'une liaison point à point. C'est ainsi que fonctionnent, par exemple, les réseaux de téléphones, fussent-ils mobiles : pendant une communication, un circuit physique est établi entre les deux correspondants, circuit constitué de plusieurs segments aux jonctions desquels se trouvent des équipements de commutation capable d'associer une destination à un numéro.

Le réseau téléphonique automatique (dit « commuté ») est constitué de lignes physiques qui résolvent le problème de base et relient entre eux des équipements terminaux (des téléphones fixes ou portables, des modems, des fax...) et des équipements de transmission de données, qui sont essentiellement des auto-commutateurs. Quand vous composez un numéro de téléphone vous utilisez une ligne (de cuivre pour un téléphone fixe, hertzienne pour un portable) qui vous met en relation avec l'auto-commutateur du secteur ; celui-ci peut identifier la destination de votre appel d'après le numéro de votre correspondant et sélectionner une ligne qui va soit atteindre directement votre correspondant dans le cas d'un appel local, soit atteindre un autre auto-commutateur qui va lui-même sélectionner une ligne propre à acheminer votre communication vers votre correspondant, qui après quelques étapes entendra sonner son téléphone. Pendant toute la durée de votre conversation les différents segments de ligne qui auront été sélectionnés par les auto-commutateurs successifs seront réservés à votre usage exclusif. Ils constitueront un *circuit commuté*, et pour cette raison le réseau téléphonique est qualifié de réseau à commutation de circuits, ou réseau commuté tout court.

Si un des auto-commutateurs du circuit n'a plus de ligne libre au moment de votre appel, celui-ci ne pourra pas être établi et une voix enregistrée vous suggérera de renouveler votre appel ultérieurement. Cette nécessité de réserver physiquement un canal de communication de bout en bout pour une communication individuelle, même quand vous et votre interlocuteur marquez une pause dans la conversation, est une limite du réseau téléphonique.

6.5.2 Commutation de paquets

Beaucoup de réseaux informatiques ont fonctionné en se contentant de la commutation de circuits, ce qui, soit dit en passant, revient à se ramener à des liaisons de couche 2 : une fois le circuit établi, on a l'illusion d'une liaison unique. Depuis la fin des années 1960 une autre idée s'est imposée. Monopoliser un circuit physique pour une seule communication semblait logique pour acheminer la voix lors d'un échange téléphonique : cela correspondait au modèle de taxation des compagnies de téléphone, la voix exige pour être transmise sans déformation le respect de la cadence d'émission des impulsions, ce que l'on appelle l'isochronie, à l'époque la transmission était analogique, et le débit prévisible et à peu près constant. Mais pour la transmission de données l'impératif d'isochronie est moins fort et l'on peut imaginer d'autres solution.

Pour comprendre les réseaux d'ordinateurs il peut être utile de les comparer à d'autres types de réseaux : téléphonique, ferroviaire, électrique, routier, de distribution d'eau ou de courrier postal. Tous ces réseaux ont des caractéristiques communes ; dès qu'ils atteignent une certaine taille, l'acheminement d'un message (ou respectivement d'une communication téléphonique, d'un wagon de marchandises, d'une quantité d'énergie électrique, d'une lettre...) pose des problèmes d'itinéraire et de vérification de bon acheminement. Il faut aussi optimiser l'usage du réseau : chaque wagon n'a pas sa propre locomotive du départ à l'arrivée, mais il peut être accroché successivement à différents trains qui le rapprocheront de sa destination, ce que nous appellerons du multiplexage (de messages ou de wagons).

Tentons une comparaison avec le réseau téléphonique : lorsque nous avons résolu le problème de base, faire communiquer deux équipements, nous avons l'équivalent d'une paire de talkie-walkie. C'est un bon début, mais si nous pensons à la façon dont nous utilisons le téléphone, pour appeler par une procédure automatique un abonné au Japon ou un portable en rase campagne, nous concevons qu'il faut bien des équipements et des techniques pour passer du stade « talky-walky » à un vrai réseau complexe, ou de la couche 1 à la couche 2 en l'occurrence.

Circuits virtuels

Pensons maintenant à la circulation de wagons de marchandises dans un réseau ferré : ils sont accroché à des locomotives pour former des trains. Soit par exemple un wagon à acheminer de Lille à Nice. Il va d'abord être accroché à un train Lille–Paris. À Paris, le train va être démembré dans une gare de triage et notre wagon accroché à un nouveau train Paris-Marseille, en compagnie de nouveaux compagnons wagons. À Marseille un nouveau triage aura lieu à l'issue duquel un train venant par exemple de Perpignan mènera notre wagon

jusqu'à Nice. Ces trois trains successifs auront roulé à des vitesses différentes en comportant des wagons différents, mais nous pouvons dire, du point de vue de l'entreprise lilloise qui envoyait le contenu d'un wagon de Lille à Nice, qu'ils ont constitué un unique train virtuel Lille–Nice pour le wagon qui nous intéresse. Sur aucun des trois segments de la ligne virtuelle Lille–Nice, le wagon n'a eu besoin d'un conducteur informé de la destination et de l'itinéraire détaillé pour y parvenir : le conducteur de la locomotive et les opérateurs des postes d'aiguillage ont assuré son acheminement. Il fallait juste une étiquette (sans doute à code barre ou électronique) « Nice » sur le wagon pour qu'il soit correctement aiguillé lors des opération de triage.

Les réseaux informatiques conformes à la norme X25, dont l'exemple en France est le réseau Transpac, popularisé en son temps par le Minitel dont il était le support, fonctionnent selon un principe conforme à la métaphore du train de marchandises. Le flux d'informations est découpé en paquets de taille plus ou moins fixe, quelques centaines à quelques milliers de caractères, et ces paquets vont être acheminés par un circuit virtuel. Lorsque l'on veut établir une communication entre deux nœuds du réseau, on commence par déterminer un circuit virtuel qui passe par différents équipements intermédiaires que nous appellerons concentrateurs. Un concentrateur est un ordinateur spécialisé qui dispose d'un certain nombre de lignes de communications, d'une mémoire et d'un logiciel. Le logiciel du concentrateur de Paris est capable de déterminer que pour contribuer à l'établissement du circuit virtuel Lille–Nice il faut envoyer les paquets en provenance de Lille et destinés à Nice sur la ligne qui se dirige vers le concentrateur de Marseille, qui fera suivre. Le concentrateur gardera en mémoire une table (dite *table de routage*) qui associera un circuit virtuel Lille–Nice à un numéro d'identification et à une sortie vers le concentrateur de Marseille. Chaque paquet expédié sur ce circuit virtuel comportera comme une étiquette le numéro d'identification de circuit virtuel qui permettra au concentrateur de l'expédier dans la bonne direction.

L'ensemble de ces conventions — format et contenu des tables de routage et des paquets, format des adresses (analogues à des numéros de téléphone) qui identifient de façon unique chaque nœud et chaque extrémité du réseau, procédure d'établissement du circuit virtuel, structure de son numéro d'identification, règles d'acheminement des paquets, procédure d'accusé de réception par laquelle l'extrémité destination avertit l'extrémité origine de la bonne fin de l'échange — constituent un protocole de communication, ici en l'occurrence le protocole X25, protocole de couche 3 pour les réseaux à commutation de paquets par circuits virtuels.

Le progrès par rapport à la commutation de circuits est considérable : plusieurs circuits virtuels peuvent partager, pour une partie de leurs trajets respectifs, la même infrastructure physique. Les tronçons très fréquentés peuvent être équipés de lignes à plus haut débit que ceux qui le sont moins. Les concentra-

teurs réalisent l'adaptation de débit entre les liaisons de caractéristiques différentes. Ce modèle a beaucoup plu aux opérateurs téléphoniques traditionnels parce qu'il permettait un mode de tarification conforme à leurs habitudes : une fois le circuit virtuel établi, tous les paquets empruntent le même itinéraire et il suffit de les compter dans un concentrateur quelconque pour pouvoir facturer au bit près.

Nous allons voir mieux. Le modèle que nous venons de décrire a des limites : l'établissement d'une communication exige que soit constitué dès auparavant un circuit virtuel de bout en bout. Cela ne pose pas de problème particulier au sein d'un réseau doté d'une administration unique et centralisée, par exemple Transpac. Il est à la rigueur concevable d'organiser une coordination entre quelques grands opérateurs nationaux, conformément encore une fois au modèle familier à France Télécom et à ses homologues dans d'autres pays, quoique l'expérience ait prouvé le caractère laborieux (et onéreux) d'une telle coordination. Mais nous allons voir un principe plus souple, celui de l'Internet, qui permet l'acheminement sûr des communications parmi un univers de réseaux multiples au foisonnement anarchique.

Commutation de paquets « pure »

Passons de la métaphore ferroviaire à la métaphore routière. Soit une noce : la famille et les différents groupes d'invités doivent se rendre au village où ont lieu la cérémonie et le banquet. Ils y vont en voiture. Plusieurs groupes partent de la même ville, mais ils ne tiennent pas tous dans la même voiture, alors ils voyagent indépendamment. Tous ne prennent pas le même itinéraire, et les premiers partis ne sont pas forcément les premiers arrivés. Certains ont étudié la carte et déterminé un trajet jusqu'au village, mais d'autres, plus insouciants, se fient aux panneaux indicateurs qu'ils observent au bord de la route ou aux carrefours au fur et à mesure de leur progression vers l'objectif, ce qui ne devrait pas les empêcher d'arriver à bon port. Si la noce est très nombreuse elle peut saturer l'autoroute, auquel cas les panneaux lumineux de type « bouchon à 5 km » viendront avertir les attardés qu'il vaut mieux prendre l'itinéraire de délestage, ce qui leur permettra éventuellement d'arriver avant les premiers partis.

À l'arrivée au village il a été convenu de former un cortège, ce qui suppose bien sûr un ordre protocolaire : d'abord la voiture de la mariée, puis celle du marié, suivi de la belle-mère, etc. Évidemment les voitures n'arrivent pas dans le bon ordre, et pour rester fidèle à la tradition la mariée arrivera la dernière, aussi faudra-t-il une manœuvre supplémentaire pour constituer le cortège dans le bon ordre, ce qu'aurait évité un voyage par train spécial.

Le tableau que nous venons de dresser du départ et du regroupement final de la noce figure assez fidèlement l'acheminement d'un message de bonne taille par l'Internet conformément au protocole IP *(Internet Protocol)*. Chaque voiture re-

présente un paquet, appelé aussi *datagramme* IP, ce qui insiste sur son caractère autonome, par opposition au paquet X25, sagement rangé en file sur un circuit virtuel. Le cortège nuptial des voitures remises dans le bon ordre représente le message tel qu'il doit parvenir à destination. Comme chaque paquet est acheminé indépendamment des autres, par un itinéraire éventuellement différent, il ne suffit pas pour savoir comment l'acheminer d'un numéro de circuit virtuel, donc chaque paquet doit comporter son adresse d'origine et son adresse de destination. Le protocole IP définit un format d'adresse, et les organismes de coordination de l'Internet en assurent l'unicité à l'échelle mondiale.

Aux nœuds du réseau se trouvent non plus des concentrateurs X25, mais des ordinateurs spécialisés qui remplissent un rôle similaire, même si sensiblement plus complexe, appelés routeurs. Fondamentalement, un routeur reçoit un paquet par une de ses lignes de communication, analyse son adresse de destination, consulte ses tables de routage, et en déduit sur quelle ligne il doit réexpédier le paquet, ou s'il doit le rejeter.

À première vue, tout ceci semble moins rationnel et moins efficace que les circuits virtuels de X25. Mais c'est aussi considérablement plus souple, en réalité plus efficace et plus simple, et finalement ce modèle l'a emporté, pour des raisons que nous développerons à la section 6.5.9 p. 184.

6.5.3 Le protocole IP et l'Internet

Le protocole IP correspond à la couche 3 du modèle OSI, la couche réseau. La « pile » TCP/IP (comme une pile de couches... empilées) n'obéit pas strictement à la nomenclature du modèle OSI : elle comporte une couche liaison de données qui englobe les couches 1 et 2 de l'OSI, la couche IP (réseau) correspond à la couche 3 de l'OSI, la couche TCP[8] (transport) correspond à la couche 4 de l'OSI. La couche « applications » englobe tout ce qui relève des couches hautes de l'OSI.

L'architecture de TCP/IP peut être vue sous l'angle suivant. À partir d'un message émis par un utilisateur, chaque couche en partant de la plus haute lui ajoute des en-têtes qui contiennent les informations nécessaires à son fonctionnement, ce que montre la figure 6.5.

Ainsi, un message électronique sera d'abord doté par votre logiciel de courrier des en-têtes applicatives, en l'occurrence telles que décrites par le RFC 822 (ce sont les lignes `From:`, `To:`, `Subject:`, etc. que vous lisez en haut des messages). Puis ce message conforme au RFC 822 se verra découpé en segments TCP, chacun doté de l'en-tête convenable (décrite plus bas). Chaque segment TCP sera empaqueté dans un datagramme IP qui possède lui aussi une en-tête. Et chaque datagramme sera expédié sur la couche liaison de données qui correspond au support physique, Ethernet par exemple.

8. ... ou UDP, l'autre couche transport disponible au-dessus d'IP.

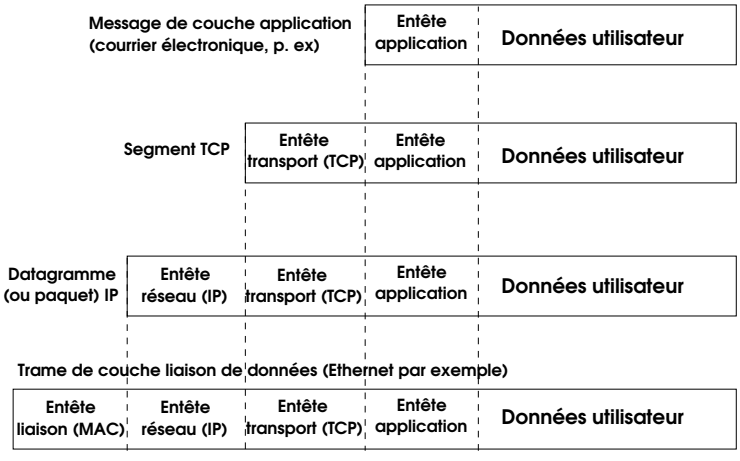


Figure 6.5 : *En-têtes des quatre couches de TCP/IP*

Le protocole réseau IP fournit à la couche transport un service non fiable non connecté de datagrammes. Le terme datagramme signifie que le flux de bits remis par la couche transport (TCP) est découpé en paquets acheminés indépendamment les uns des autres. Par non fiable nous entendons que la couche IP ne fournit aucune garantie de remise des datagrammes ni aucun contrôle d'erreur, et par non connecté nous entendons que la couche IP ne maintient aucune information d'état sur une transmission de données en cours, et notamment qu'elle ne garantit pas la remise des datagrammes dans l'ordre dans lequel ils ont été émis.

Ces caractéristiques sont de nature à inquiéter les néophytes, et semblent curieuses, d'autant plus que la couche de liaison de données fournit à la couche réseau, pour chaque segment physique d'un chemin de données utilisé par un datagramme, un service fiable de flux de bits remis dans le bon ordre.

En fait, la couche IP ne fournit pas de contrôle d'erreur parce que de toute façon la couche TCP devra en effectuer, ainsi que la vérification du bon ordre de remise des datagrammes au terminus de la transmission, et que de tels contrôles au niveau de la couche 3 seraient redondants. Son ascétisme et sa désinvolture confèrent à la couche IP la simplicité, la légèreté et la souplesse qui font son efficacité. Mais avant d'en décrire plus précisément les principes techniques, il nous faut donner quelques informations sur l'organisation du plus grand réseau IP : l'Internet.

Organisation administrative de l'Internet

Une chose que les néophytes ont souvent du mal à comprendre, c'est que l'Internet ne soit la propriété de personne ni d'aucune institution. Saisissons cette occasion de démentir la légende répétée *ad nauseam* qui voudrait que l'Internet ait été inventé à la demande des militaires américains selon un cahier des charges rédigé en vue de préserver une capacité de communication après une frappe nucléaire. Il n'y a jamais rien eu qui ressemble de près ou de loin à un « cahier des charges de l'Internet ». Cette thèse télescope plusieurs événements liés mais distincts. Paul Baran, de la firme RAND, contractant du DoD *(Department of Defense)*, avait posé les principes d'un tel système de communications dépourvu de point de centralisation unique afin de maintenir certaines liaisons même si certains nœuds venaient à être détruits. Les travaux de Baran furent publiés entre 1960 et 1964. Le même DoD, plusieurs années plus tard, en 1969, a impulsé par son agence l'ARPA *(Advanced Research Projects Agency)* la création du réseau ARPANET, qui n'était pas destiné aux communications militaires mais à faciliter la collaboration entre centres de recherches universitaires et industriels sous contrat avec l'ARPA. BBN (Bolt, Beranek & Newman) a été très impliqué dans le développement d'ARPANET, où se retrouvent certaines idées de Paul Baran ; ce réseau fut un des premiers (avec Cyclades en France, sous la direction de Louis Pouzin) à utiliser la technique de commutation de paquets. En 1976, la clairvoyance de Vint Cerf et de Robert Kahn, le financement de BBN et les contrats de l'ARPA devenue entre temps la DARPA donnèrent naissance au protocole réseau de l'Internet, TCP/IP. Un peu plus tard, en 1979, le groupe de recherche en système *(Computer Systems Research Group, CSRG)* de l'Université de Californie à Berkeley allait incorporer TCP/IP à une nouvelle version de Unix, dite BSD *(Berkeley Software Distribution)*. Tous ces éléments marinaient dans une soupe originelle qui a donné naissance à l'Internet à la fin des années 1970, quand les centres de recherche connectés à ARPANET ont voulu communiquer avec les universités de leur choix et que la NSF *(National Science Foundation)* a entrepris de leur en donner les moyens.

Le fonctionnement de l'Internet, à l'image de sa construction, repose sur la coopération volontaire. Les décisions organisationnelles et techniques sont prises par des instances aux séances desquelles tout un chacun peut assister et participer.

- L'*Internet Architecture Board (IAB)* (IAB) est responsable des grandes orientations et de la coordination.
- L'*Internet Engineering Task Force (IETF)* se charge de la normalisation à court terme et émet les *Requests for Comments (RFC)*, qui sont les documents de référence pour le fonctionnement du réseau. Citons ici le nom de Jon Postel, éditeur des RFC depuis la première en 1969 jusqu'à sa mort en 1998, et auteur ou coauteur de 204 d'entre elles, ce qui lui a conféré une

influence considérable sur la physionomie du réseau. Toutes les RFC sont accessibles par l'URL *(Universal Resource Locator)* `http://www.ietf.org/rfc/` ou sur de nombreux sites miroirs. Nous ne saurions trop en conseiller la lecture, même si leur qualité littéraire est inégale elles fournissent sur l'Internet une information de première main, souvent exposée très clairement, et dont la citation dans les dîners en ville vous assurera une réputation de guru du réseau.

– L'*Internet Steering Group (IESG)* coordonne l'IETF, dont l'effectif est devenu très important.
– L'*Internet Assigned Numbers Authority (IANA)* centralise et contrôle les conventions relatives à l'identification des objets du réseau, et notamment veille à l'unicité des adresses.
– L'*Internet Corporation for Assigned Names and Numbers* (ICANN) supervise l'attribution des noms de domaines et des adresses.

Cette organisation coopérative ne signifie pas l'absence de rapports de force marchands ou politiques, mais elle exclut (au moins à court terme) la prise de contrôle par une entreprise unique.

Organisation topographique de l'Internet

La figure 6.6 donne une représentation schématique de la topographie de l'Internet. Cette représentation est simplifiée, notamment elle est purement arborescente, alors que rien n'empêche une entreprise d'avoir plusieurs FAI, ni un FAI d'être connecté à plusieurs centres d'interconnexion de niveau supérieur, ce qui complique le schéma et le transforme d'un arbre en un graphe connexe quelconque. L'essentiel dans l'établissement d'une communication, c'est, à chaque nœud, de savoir quel est le prochain nœud sur l'itinéraire, et par quelle liaison l'atteindre. Les nœuds terminaux, origines ou destinations des communications, sont au sein des réseaux locaux de campus ou d'entreprise. Les routeurs sont des nœuds particuliers, dotés de plusieurs adresses réseau (une par interface raccordée à une ligne de communication) et possédant des tables de routage.

La figure 6.7 représente un gros plan sur un réseau local simple, raccordé à l'Internet par un routeur, et constitué d'un unique segment de couche 2, en l'occurrence un réseau Ethernet.

La double ligne horizontale symbolise le bus, ou graphe connexe complet, qui stipule qu'une trame émise par une des stations atteindra toutes les stations du réseau. Les réseaux Ethernet contemporains ont une topologie physique assez différente de ce schéma, surtout s'ils utilisent les transmissions sans fil, mais ce schéma correspond toujours bien à la structure logique de la communication.

Les stations ordinaires ont une seule interface réseau, et donc une seule adresse de couche 2 et une seule adresse de couche 3 (dans les deux couches

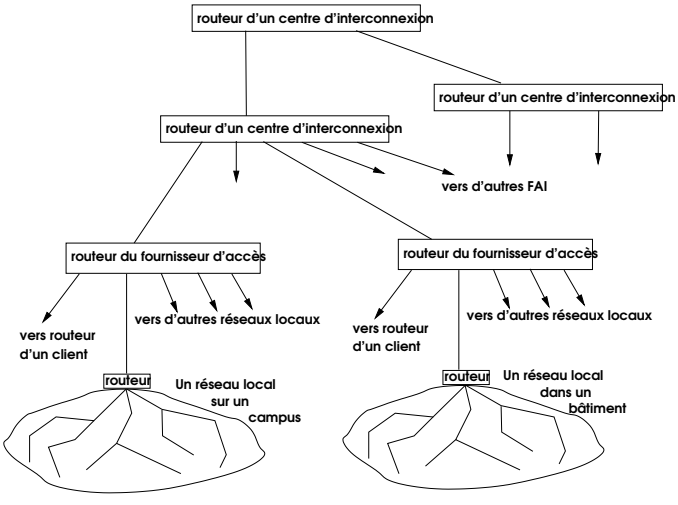


Figure 6.6 : *Topographie de principe de l'Internet*

les adresses sont associées aux interfaces). Dans notre exemple les adresses (de couche 3) des stations vont de `192.168.2.101` à `192.168.2.105`. Le routeur, par définition, possède au moins deux interfaces et donc deux adresses, ici vers le réseau local et vers le FAI et l'Internet. Dans notre exemple l'adresse intérieure est `192.168.2.1` et l'adresse extérieure `171.64.68.22`.

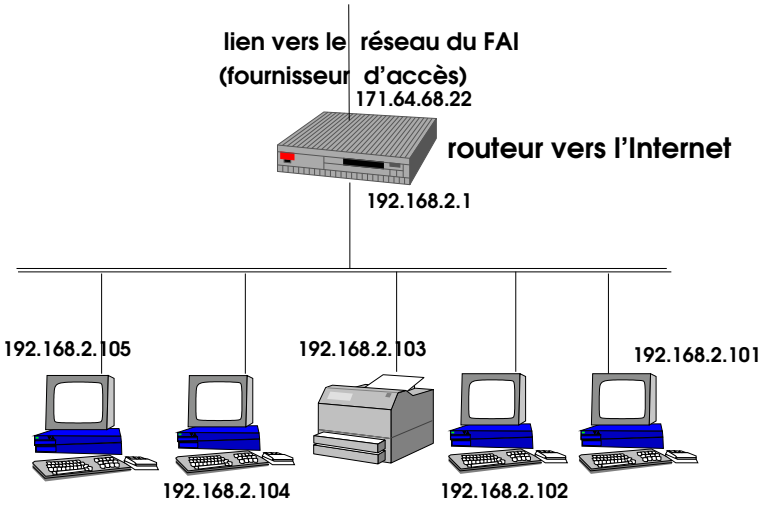


Figure 6.7 : *Un réseau local simple*

L'adresse et le datagramme IP

Comme nous l'avons vu plus haut, la couche réseau a sa propre vision de la topologie du réseau, et partant son propre système d'adressage.

Il faut tout d'abord rappeler que les adresses de couche 3, comme celles de couche 2, sont attribuées aux interfaces et non aux machines. Une machine qui a trois cartes réseau aura au moins trois adresses ; à chaque interface peuvent correspondre plusieurs adresses.

Comme nous allons le voir, l'adresse a deux fonctions : l'identification d'un nœud et sa localisation ; elle est en cela analogue au numéro de téléphone. On aurait pu imaginer qu'il en soit autrement : notre numéro de sécurité sociale nous identifie sans nous localiser, ce qui nous évite d'avoir à en changer à chaque déménagement. Mais c'est ainsi et la nouvelle version du protocole IP, IPv6, reste en gros fidèle à ce principe ; la dissociation de ces deux rôles de l'adresse aurait des avantages indéniables pour les stations mobiles, de plus en plus nombreuses, et des recherches se poursuivent dans cette direction.

Chaque adresse IP est unique[9] dans tout l'Internet, ce qui lui permet d'assurer sa fonction d'identifiant. Quant à la fonction de localisation elle est assurée par les mécanismes complexes du routage.

La façon dont une station (terme qui désigne un nœud vu du point de vue de celui qui s'en sert) reçoit son adresse IP est variable. L'ICANN distribue des tranches d'adresses à des organismes régionaux (pour l'Europe, c'est RIPE, pour Réseaux IP Européens), qui les redistribuent à des organismes qui peuvent être nationaux (pour la France c'est l'association AFNIC), qui eux-mêmes les attribuent aux fournisseurs d'accès à l'Internet (FAI). Lorsqu'une entreprise ou un particulier veut obtenir un accès à l'Internet, il s'adresse à un FAI, qui lui attribue, selon l'importance de son réseau et de son abonnement, une ou plusieurs adresses IP. En général, les particuliers n'ont pas besoin d'une adresse fixe permanente : lorsqu'ils allument leur ordinateur et leur modem, le routeur du FAI le détecte et leur envoie une adresse temporaire, affectée dynamiquement à partir d'un pool d'adresses, qui servira le temps de la session de travail. Cette adresse peut d'ailleurs être privée (voir plus haut).

9. Il y a une exception à l'unicité des adresses IP : sur un réseau connecté à l'Internet, on peut décider que certaines machines ne seront pas « visibles » de l'extérieur, c'est-à-dire qu'elles ne pourront pas être directement atteintes par une communication en provenance de l'Internet. Les communications qu'elles entameront se feront par l'intermédiaire d'un routeur qui, lui, sera « visible ». Ces machines non visibles pourront recevoir des adresses dites privées, selon le RFC 1918, pour lesquelles la contrainte d'unicité sera limitée au réseau local considéré. Le moyen par lequel elles communiquent avec l'Internet, que nous décrirons plus en détail un peu plus loin (cf. section 6.5.4 p. 161), repose sur une *traduction d'adresse* (NAT, pour *Network Address Translation*) effectuée par le routeur.

Depuis une dizaine d'années, IP est en pleine transition : les adresses couramment utilisées sont celles d'IP version 4 (spécifié par le RFC 791 de septembre 1981), qui comportent 32 chiffres binaires (bits), ce qui autoriserait théoriquement un maximum de 4 milliards de nœuds, en fait moins, à cause de la structure de l'adresse. Cette valeur qui semblait astronomique dans les années 1970 est en passe d'être atteinte par le développement de l'Internet. Les organismes qui coordonnent l'Internet ont défini une nouvelle version du protocole, IP version 6, et sont en train de planifier son déploiement, ce qui ne sera pas une mince affaire. L'adresse IPv6 comporte 128 bits. Cette nouvelle version du protocole apporte d'autres innovations, dans le domaine de la sécurité et dans celui de l'autoconfiguration des nœuds qui rejoignent le réseau notamment.

L'adresse IPv4 est structurée en deux parties : les chiffres les plus significatifs désignent le numéro de réseau, ou adresse de réseau, les moins significatifs l'adresse locale, qui identifie une interface d'un nœud dans le réseau (une machine peut avoir plusieurs interfaces, et donc plusieurs adresses). Il existait naguère trois classes de réseaux, distinguées par le nombre de bits réservés à l'adresse de réseau et à l'adresse locale respectivement. Ce partage de l'espace adresse en classes ne s'est pas avéré très judicieux avec la montée de la pénurie mondiale d'adresses et il est partiellement abandonné aujourd'hui au profit d'une notation plus souple, qui permet de spécifier *ad libitum* le nombre de bits destinés à identifier le réseau, et ceux qui restent disponibles pour identifier les nœuds. IPv6 offre plus de souplesse dans la gestion des adresses, en tenant compte de phénomènes inconnus lors de la spécification d'IPv4, comme le rôle des fournisseurs d'accès à l'Internet (FAI), la mobilité et le nomadisme.

Écrire un nombre binaire de 32 chiffres est malcommode ; pour représenter une adresse IP on utilise la convention suivante : elle est découpée en quatre nombres de huit bits (octets), et on écrit ces nombres en notation décimale, séparés les uns des autres par un point. Voici une adresse représentée de la sorte : `171.64.68.20`. Pour préciser la forme des adresses IPv4 donnée à l'alinéa précédent, le RFC 791 distinguait des adresses de classe A, avec le premier bit à 0, 7 bits pour l'adresse de réseau (le premier octet est un nombre inférieur à 128) et 24 bits pour l'adresse locale (interface d'un nœud) ; des adresses de classe B avec le premier bit à 1, le deuxième bit à zéro (le premier octet est un nombre compris entre 128 et 191 inclus), 14 bits pour l'adresse de réseau et 16 bits pour l'adresse locale ; les adresses de classe C avec les deux premiers bits à 1 et le troisième bit à 0 (le premier octet est un nombre supérieur à 191 et inférieur à 224), 21 bits pour l'adresse de réseau et 8 bits pour l'adresse locale.

Actuellement ce système assez rigide est contourné et les classes sont de fait abolies par le système CIDR *(Classless Interdomain Routing)*, qui instaure une situation hiérarchique plus simple où, à une feuille de l'arborescence, l'administrateur d'un réseau local fixe les quelques bits les plus à droite de l'adresse d'une interface, puis le FAI régional fixe quelques bits un peu plus à gauche, puis un

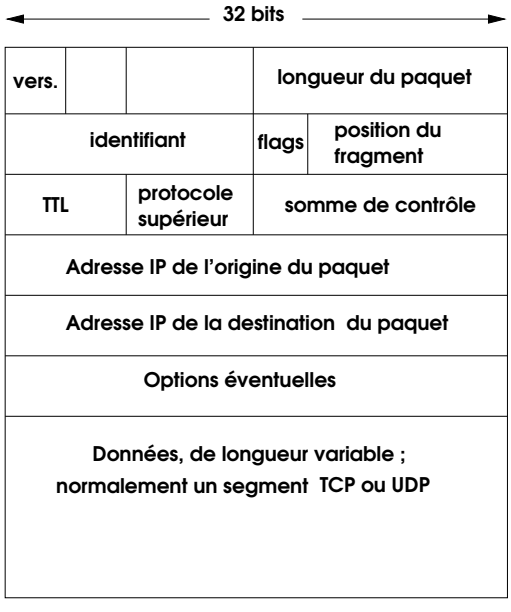


Figure 6.8 : *Datagramme (ou paquet) IPv4*

organisme national encore quelques bits supplémentaires, et ainsi de suite jusqu'à l'ICANN qui distribue les bits de gauche à toute la planète, tout ceci en essayant de s'organiser pour que des machines topologiquement proches les unes des autres aient le plus de bits de gauche en commun afin de simplifier et d'abréger les tables de routage. Mais comme le RFC 791 est toujours en vigueur et que la plupart des adresses que vous rencontrerez au cours des prochaines années lui obéiront, autant comprendre cette syntaxe curieuse.

La partie « adresse de réseau » de l'adresse joue le rôle de préfixe : ainsi, dans un réseau local de campus ou d'immeuble tel que représenté par la figure 6.7, tous les nœuds du réseau ont le même préfixe, qui caractérise le réseau. Dans notre cas, l'adresse de réseau est `192.168.2`, ce qui incidemment est un préfixe d'adresse réservé aux réseaux privés, non visibles de l'Internet. À l'intérieur du réseau, les nœuds sont distingués par le suffixe de leur adresse. De l'extérieur du réseau, pour envoyer un message à un de ses nœuds, il suffit de l'envoyer à l'adresse extérieure du routeur d'entrée (dans notre exemple `172.64.68.22`), ce routeur détiendra l'information propre à acheminer le message à la bonne adresse. Quand une machine du réseau souhaite expédier un datagramme à une machine extérieure, elle l'envoie à l'adresse intérieure du routeur (`192.168.2.1`), qui joue le rôle de passerelle de sortie et qui sait (nous allons bientôt le savoir nous aussi) comment le faire parvenir à destination.

Pour donner une meilleure vision du datagramme (ou paquet) IPv4, la figure 6.8 en donne le diagramme complet. Le champ « protocole supérieur » permet de connaître la nature des données, en général soit un segment TCP, soit un segment UDP. Les informations de la seconde ligne (identifiant, flags, position du fragment) servent dans le cas suivant : tous les segments d'un itinéraire dans un réseau n'ont pas forcément les mêmes caractéristiques techniques, et notamment certains peuvent n'accepter qu'une taille maximum de paquet inférieure à la taille fournie par le nœud précédent, ce qui oblige le routeur à fragmenter le paquet, qui devra bien sûr être réassemblé à un moment ou à un autre, grâce à ces informations. Cette possibilité de fragmentation est un dispositif souvent utilisé par les pirates pour leurrer les logiciels de sécurité qui analysent les paquets de données, parce que les algorithmes de fragmentation et de réassemblage sont compliqués et donc souvent mal réalisés par certains fournisseurs.

Le paquet IPv4 comporte beaucoup d'informations en fait inutiles et parfois gênantes, en particulier la somme de contrôle qui doit être recalculée chaque fois que l'en-tête du paquet est modifié (elle ne contrôle l'intégrité que de l'en-tête). Le paquet IPv6 est beaucoup plus simple, comme en témoigne la figure 6.9.

L'adresse IPv6, de 128 bits donc, est notée de la façon suivante : elle est découpée en tranches de 16 bits, représentée chacune par quatre chiffres hexadécimaux, les tranches séparées les unes des autres par le signe « : », ainsi :

```
0123:4567:89ab:cdef:0123:4567:89ab:cdef
```

Il y a des règles assez subtiles pour abréger cette représentation lorsqu'elle comporte de longues suites de bits à zéro, dont l'exposé ne nous semble pas indispensable, sachant que le lecteur curieux en trouvera le détail dans le RFC 2373 (`http://www.ietf.org/rfc/rfc2373.txt`).

IPv6 introduit d'autres modifications dans le traitement des adresses : si une adresse est toujours attribuée à une interface (plutôt qu'à un nœud), cette attribution est temporaire et les adresses sont réputées changer. Les chiffres les moins significatifs de l'adresse IPv6 sont calculés à partir de l'adresse de couche 2 (MAC) lorsqu'il y en a une.

6.5.4 Exception à l'unicité des adresses : traduction d'adresses (NAT)

Le principe du standard téléphonique d'hôtel

Le système de traduction d'adresses [10] NAT *(Network Address Translation)* est apparu en 1994 dans le RFC 1631 (remplacé maintenant par le 3022), initialement

10. Incidemment, l'anglais *translation* se traduit ici en français par *traduction*, translation d'adresse ne veut rien dire et celui qui employe cette locution prouve simplement qu'il connaît mal l'anglais et le français et qu'en outre il ne connaît guère le processus qu'il décrit.

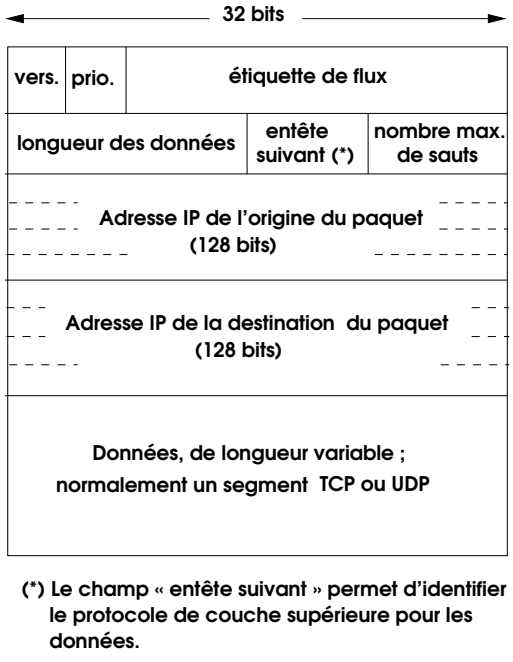


Figure 6.9 : *Datagramme (ou paquet) IPv6*

pour permettre la communication entre l'Internet et des réseaux privés contenant des adresses IP non conformes au plan d'adressage de l'Internet, et il a été ensuite très largement utilisé pour pallier le déficit d'adresses IP engendré par l'étroitesse de la plage d'adresses de la version 4 du protocole. Il est devenu de ce fait à la fois une solution et un problème de sécurité des réseaux.

Le principe en est le suivant : chaque nœud de l'Internet doit posséder une adresse IP pour mettre en œuvre le protocole TCP/IP, et cette adresse doit être unique, comme pour les numéros de téléphone, sinon il serait impossible d'acheminer correctement les communications.

Mais, pour poursuivre la comparaison avec le téléphone, dans un hôtel par exemple, seul le standard a un numéro de téléphone unique, et le poste de chaque chambre a un numéro local, à usage strictement interne, et qui peut très bien être le même que celui d'une chambre dans un autre hôtel : cela n'a aucune conséquence fâcheuse parce que le numéro de la chambre n'est pas visible de l'extérieur ; ceci permet parfaitement à l'occupant de la chambre d'appeler l'extérieur en donnant un code particulier (« composer le 0 pour avoir l'extérieur »), et de recevoir des communications en passant par le standard qui effectue la commutation vers la ligne de la chambre.

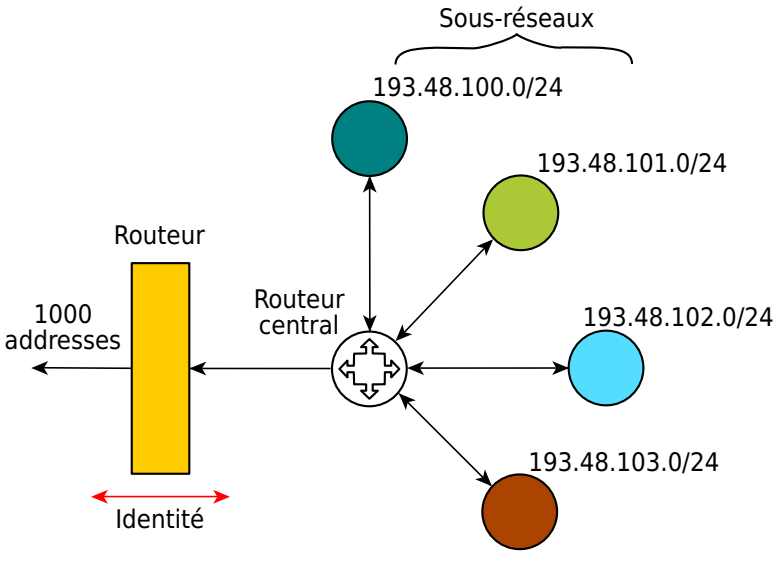


Figure 6.10 : *Réseau sans NAT : les adresses des hôtes sont des adresses uniques et routées sur Internet.*

Adresses non routables

Le système NAT repose sur un principe analogue : dans un réseau local, seuls les serveurs qui ont vocation à abriter des serveurs vus de tout l'Internet, comme le serveur WWW de l'entreprise ou sa passerelle de messagerie, doivent recevoir des adresses reconnues universellement, et donc uniques et conformes au plan d'adressage de l'Internet. Les postes de travail ordinaires peuvent recevoir des adresses purement locales, qui ne sont pas routables, c'est-à-dire qu'un paquet à destination d'une telle adresse peut circuler sur le réseau local et atteindre sa destination, mais ne peut pas franchir un routeur, parce que ces classes d'adresses sont explicitement désignées pour que les routeurs les oublient. Sont dites *non routables* toutes les adresses appartenant aux blocs d'adresses définis à cet effet par le RFC 1918 : 192.168.0.0 à 192.168.255.255 (préfixe 192.168/16), 172.16.0.0 à 172.31.255.255 (préfixe 172.16/12) et 10.0.0.0 à 10.255.255.255(préfixe 10/8).

Accéder à l'Internet sans adresse routable

Si la gestion des adresses non routables s'arrêtait là, ces malheureux ordinateurs dotés d'adresses de seconde zone ne pourraient jamais naviguer sur l'Internet : en effet, une communication aussi simple que l'accès à un serveur Web demande que les paquets comportent une adresse source et une adresse destination valides, ne serait-ce que pour que le serveur puisse renvoyer au client

le contenu de la page qu'il a voulu consulter. D'ailleurs dans un réseau fermé sans connexion à l'Internet les possibilités de communication sont limitées au réseau local, et c'est pour de tels réseaux qu'avaient été créées à l'origine les classes d'adresses non routables, que NAT a ensuite astucieusement détournées de leur destination, si j'ose dire.

Sur un réseau connecté à l'Internet qui ne contient que des postes de travail dotés d'adresses non routables, il y a au moins un nœud qui possède une adresse routable, c'est le routeur d'entrée du réseau, puisque justement il est connecté. Alors il y a au moins un moyen de faire communiquer un poste du réseau local avec l'extérieur : il faut pour cela que le routeur soit doté de la capacité de traduction d'adresses, justement ; ainsi il pourra jouer vis-à-vis des nœuds du réseau local le même rôle que le standard de l'hôtel vis-à-vis des postes téléphoniques des chambres, en « passant les communications ». Le principe de NAT est de remplacer une adresse interne non routable par une adresse routable.

Réalisations

La façon la plus simple de réaliser la traduction d'adresse est la méthode *statique* : à chaque adresse interne non routable on fait correspondre, bijectivement, une adresse routable qui la remplace. Le routeur contient la table de correspondance et fait la traduction, sans autre forme de procès.

La traduction d'adresse statique est simple, mais dans l'univers de la fin des années 1990 la pénurie des adresses IP (la version 4 du protocole IP comporte des adresses sur 32 chiffres binaires, ce qui autorise un maximum de 4 294 967 295 adresses uniques, mais en fait un peu moins compte tenu des contraintes sur la structure de ces adresses) a conduit vers d'autres réalisations, notamment la traduction d'adresses dite *dynamique*, et plus particulièrement vers une de ces méthodes dynamiques, dite *IP masquerading* (masquage d'adresse IP), aujourd'hui prédominante et que nous allons décrire bièvement (pour plus de détails et de références, cf. Wikipédia [138]). Avec NAT et le masquage d'adresse IP, seul le routeur possède une adresse routable, toutes les communications des nœuds internes sont vues de l'extérieur comme issues de cette adresse ou destinées à elle, et le tri est fait par le routeur au moyen d'une manipulation des numéros de port, de façon tout à fait analogue au travail du standardiste de l'hôtel que nous évoquions ci-dessus.

En anticipant sur la section suivante, disons qu'une connexion TCP est identifiée par la quadruplet {adresse IP de destination, numéro de port de destination, adresse IP d'origine, numéro de port d'origine}[11]. En général, dans le paquet qui

11. En précisant qu'un *port* (en français *sabord*, orifice destiné à laisser passer un flux) dans la terminologie TCP/IP est un numéro conventionnel qui, associé à une adresse IP, identifie une extrémité de connexion, ou en termes plus techniques une *socket*, que l'on pourrait traduire par

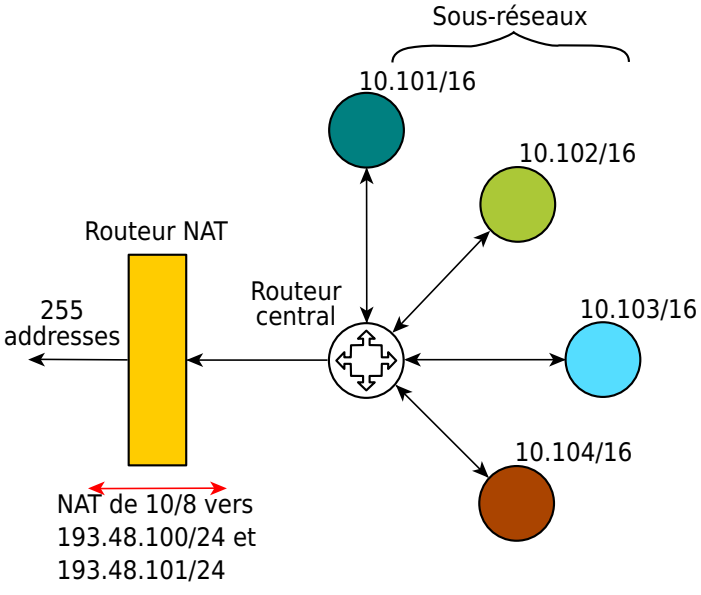


Figure 6.11 : *Réseau avec NAT : les adresses des hôtes sont des adresses réutilisables. Le routeur d'entrée fait la traduction d'adresse. On notera que la modification du plan d'adressage alloue désormais un réseau /16 par sous-réseau, s'affranchissant de la limite des 254 adresses possibles avec un /24.*

initie la connexion, le numéro de port de destination obéit à une convention (par exemple 80 pour l'accès à un serveur Web), et le numéro de port d'origine est quelconque, supérieur à 1024, et choisi de façon à former un couple unique avec l'adresse d'origine. Lorsque le routeur recevra un tel paquet, où l'adresse d'origine sera une adresse NAT non routable, il remplacera cette adresse par sa propre

prise. Par convention certains numéros de ports sont réservés aux serveurs de certains protocoles ; ainsi le port 80 est réservé au protocole HTTP (WWW), le port 25 à SMTP (courrier électronique), les ports n° 137, 138 et 139 au protocole de partage de fichiers *Netbios*, c'est-à-dire qu'un *serveur* Netbios sera en écoute sur le réseau et attendra des tentatives de connexion sur ces numéros de port, cependant que les *clients* Netbios essaieront de s'y connecter. À l'extrémité côté client, le numéro de port est quelconque, en général supérieur à 1024, et choisi de façon à former un couple unique avec l'adresse d'origine. Incidemment, il est possible, lors de l'initiation d'une connexion réseau, de déterminer un tel couple {*adresse*,*port*}, nommé *socket*, doté de la propriété d'unicité, parce que ce n'est pas le logiciel client qui établit la connexion, mais le *noyau du système d'exploitation*, du moins dans les systèmes sérieux. La connexion est ainsi identifiée de façon unique par le quadruplet {adresse IP d'origine, port d'origine, adresse IP de destination, port de destination}. Cette abstraction permet à un nœud unique du réseau d'être simultanément serveur pour plusieurs protocoles, et également d'être à la fois serveur et client. Les pirates recherchent activement sur l'Internet les machines accessibles qui laissent ouverts des ports de protocoles réputés vulnérables pour essayer de compromettre le serveur à l'écoute.

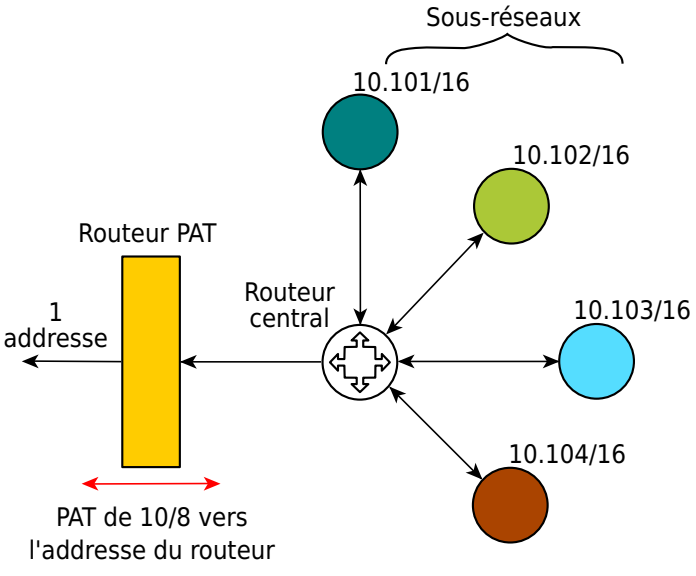


Figure 6.12 : *Réseau avec NAT et masquage d'adresse IP : seule l'adresse de l'interface externe du routeur est utilisée ; le multiplexage/démultiplexage des adresses IP internes se fait grâce aux numéros de ports (modifiés par le routeur).*

adresse, éventuellement il remplacera le numéro de port d'origine par un autre, s'il a déjà utilisé ce couple { adresse, numéro de port} pour une autre traduction, et il conservera dans une table la correspondance entre ce couple {adresse, port} envoyé sur l'Internet et celui du poste émetteur, ce qui permettra, au prix donc d'une traduction, d'acheminer les paquets dans les deux sens.

6.5.5 Une solution, quelques problèmes

À première vue, NAT est une *solution* de sécurité : avec un tel procédé et le masquage d'adresse IP, les adresses des nœuds du réseau interne, qui sont en général les postes de travail des utilisateurs, ne sont pas visibles de l'extérieur, ils sont donc hors d'atteinte de connexions établies par des malfaisants, et de fait il n'y a en général aucune raison valable pour qu'une connexion soit établie depuis l'extérieur vers un poste de travail individuel ; si tel devait être le cas cela devrait être fait selon une méthode de traduction explicite, par exemple pour permettre la prise de contrôle à distance dans un contexte d'assistance technique ou d'administration du système (mise à jour d'anti-virus, etc.).

Cette protection du réseau privé par NAT est réelle et ne doit pas être sous-estimée. Il convient cependant d'avoir conscience du fait qu'avec la version 6 du protocole TCP/IP NAT va probablement disparaître, au moins sous sa forme

actuelle, et avec lui les politiques de sécurité qui reposeraient trop exclusivement sur ses caractéristiques contingentes.

NAT pose des problèmes aux protocoles qui transportent des adresses IP et des numéros de port dans la partie « données » de leurs paquets. De tels protocoles sont dits « sales », parce qu'ils ne respectent pas le modèle d'abstraction en couches, et qu'ils transportent de l'information de niveau protocolaire (adresses) sous forme de données quelconques. Le type même du protocole sale est H323, utilisé pour la téléphonie sur IP et la visio-conférence.

NAT pose aussi des problèmes à IPSec, parce que NAT modifie les adresses et les numéros de ports, donc modifie les paquets, ce qui, du moins en IPv4, oblige à recalculer la somme de contrôle qui y figure (IPv6 supprime cette contrainte).

Dans un réseau qui met en œuvre NAT, le masquage d'adresse IP et les adresses non routables du RFC 1918 (cf. ci-dessus 6.5.4), ce qui est très répandu, notamment avec les petits routeurs ADSL que chacun installe maintenant à son domicile, les adresses sont généralement affectées de façon dynamique par un protocole conçu à cet effet, DHCP *(Dynamic Host Configuration Protocol)*. Ce protocole n'est pas exempt de critiques du point de vue de la sécurité, notamment parce qu'il émet des diffusions générales à la cantonade sans que ce soit toujours nécessaire, et aussi parce qu'il n'est pas protégé contre les usurpations d'identité : je monte un serveur DHCP pirate, j'alloue aux clients naïfs des adresses que je contrôle, je fais croire au service de noms local que les communications à destination de ces adresses doivent m'être envoyées, et ainsi j'intercepte des communications qui ne me sont pas destinées.

6.5.6 Traduction de noms en adresses : le DNS

Depuis quelques paragraphes nous parlons d'expédier des datagrammes à des adresses ici ou là, mais nous, utilisateurs de l'Internet, comment connaissons-nous les adresses des machines avec lesquelles nous voulons communiquer ? Ce que nous voulons faire, le plus souvent, c'est envoyer un courrier électronique à `France.Gall@freesurf.fr`, ou consulter le serveur `http://www.sncf.fr` pour connaître l'horaire du train pour la rejoindre. `www.sncf.fr` n'est pas une adresse, mais un nom qui désigne la machine qui abrite le serveur désiré. `freesurf.fr` n'est pas une adresse mais un nom qui désigne un domaine au sein duquel se trouvera une machine nommée par exemple `mail.freesurf.fr`, qui abritera le serveur de courrier électronique qui détient la boîte aux lettres électronique de France Gall (n'essayez pas, cette adresse est fictive). Mais la couche réseau IP n'a que faire des noms, elle ne connaît que des adresses.

Incidemment, avant même de résoudre cette embarrassante affaire de noms et d'adresses, ce que nous voulons envoyer ce ne sont pas des datagrammes IP, mais des courriers électroniques ou des interrogations au serveur. Mais là, la réponse est aisée. Notre logiciel de courrier électronique donnera à notre mes-

sage la mise en forme convenable (définie par un RFC fameux entre tous, le RFC 822), puis le transmettra à un logiciel serveur de messagerie (couche application) conforme au protocole de transport de courrier électronique SMTP *(Simple Mail Transfer Protocol)* tel que *Sendmail* ou *Postfix*, qui s'occupera de déterminer comment atteindre le destinataire, et transférera toutes les informations et données nécessaires au protocole de transport TCP (couche 4 de l'OSI), qui lui-même entamera les manœuvres nécessaires en envoyant des flux de bits à la couche IP (couche 3 de l'OSI), qui découpera tout cela en datagrammes avec les bonnes données et les bonnes adresses, et les enverra à la couche liaison de données (couche 2 de l'OSI).

Revenons maintenant à la question initiale, nous connaissons le nom d'un serveur (de courrier électronique, WWW, etc.) et ce qu'il faut à la couche IP c'est son adresse. Dans la vie courante, pour répondre à ce genre de question il y a des annuaires, eh bien sur l'Internet c'est la même chose. L'annuaire qui permet, si l'on connaît le nom d'un serveur, de trouver son adresse, et vice-versa, s'appelle le DNS *(Domain Name System)*. C'est une immense base de données distribuée sur l'ensemble de la planète, peut-être la plus grande qui existe. Ce processus de résolution de noms en adresses est complété par un autre service, qui publie les noms des serveurs de courrier électronique qui desservent un domaine. Mais qu'est-ce qu'un domaine ?

L'espace des noms de l'Internet (il est important de garder à l'esprit que les schémas qui vont suivre décrivent un espace abstrait de noms de serveurs et de domaines, et **pas** la topologie du réseau physique qui les relie) est organisé de façon hiérarchique, selon un schéma calqué sur l'organisation du système de fichiers Unix que nous avons vu à la section 5.2.1 illustré par la figure 5.6. Ce système génial, dont le fonctionnement n'est pas très intuitif, a été gravé dans le marbre des RFC 1034 et 1035 par Paul Mockapetris, actuel président de l'IAB.

La figure 6.13 montre l'organisation hiérarchique de l'espace de noms de l'Internet. Chaque nœud de l'arbre, représenté par un cercle, comprend un label, qui peut avoir jusqu'à 63 caractères de long, et pour lequel les lettres minuscules et majuscules ne sont pas distinguées. Le **nom de domaine** d'un nœud s'obtient en construisant la séquence de tous les labels des nœuds compris entre le nœud considéré et la racine inclus, séparés par des points, comme par exemple `vera.sophia.inria.fr`.

Sous une racine sans nom se trouvent un certain nombre de domaines de premier niveau (*TLD*, pour *Top Level Domains*). Chaque entreprise, association, université ou autre entité désireuse d'accéder à l'Internet appartiendra à un de ces domaines. Ceux qui ont des noms à trois lettres sont dits *domaines génériques* : `com`, `edu`, `net`, `gov`, respectivement pour les activités commerciales, éducatives, liées au réseau ou rattachées au gouvernement américain. Les TLD à deux lettres sont des domaines géographiques : `fr`, `ca`, `be`, `de`, `dz` respectivement pour la France, le Canada, la Belgique, l'Allemagne et l'Algérie. Le domaine `arpa`

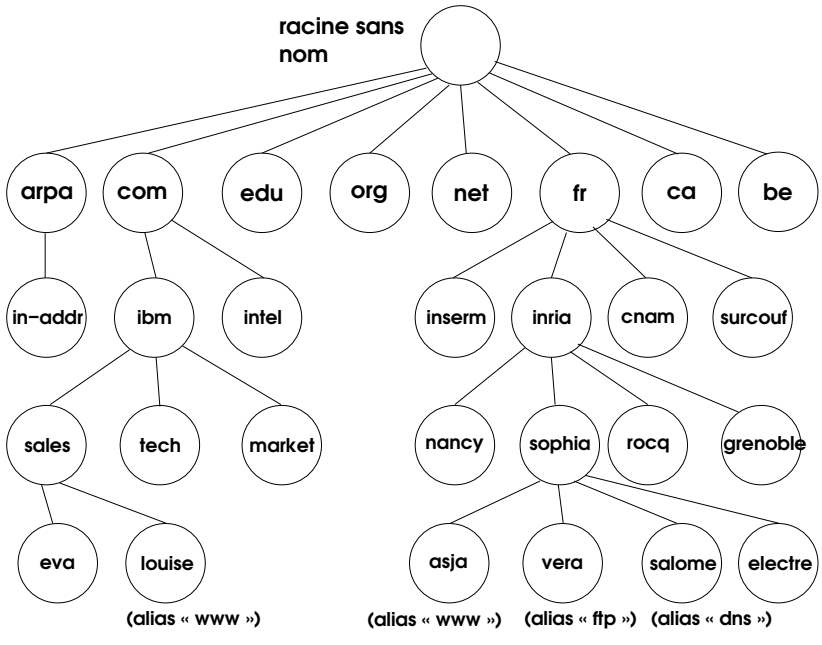


Figure 6.13 : *Organisation en arbre des noms de domaine*

a un rôle particulier, il sert à la résolution inverse, c'est-à-dire à la traduction des adresses en noms.

Au niveau inférieur, au sein du TLD, on trouve généralement les domaines qui correspondent aux universités, entreprises, etc. qui se sont connectées à l'Internet. Elles ont choisi elles-mêmes leur nom de domaine, avec la contrainte que le nom complet doit être unique : il ne peut y avoir qu'un domaine `inria.fr`, mais il peut y avoir `ibm.com`, `ibm.fr`, `ibm.be`, etc. Ces domaines peuvent être eux-mêmes subdivisés : ainsi Inria (Institut National de la Recherche en Informatique et en Automatique) aura sans doute un domaine pour chacune de ses unités de recherche, Rocquencourt, Sophia-Antipolis, Nancy, Grenoble, etc., qui s'appelleront peut-être `sophia.inria.fr`, `nancy.inria.fr`, `grenoble.inria.fr`, etc.

Cette subdivision peut atteindre des niveaux encore plus fins, mais nous allons supposer qu'Inria en est resté là, et qu'au sein du domaine `sophia.inria.fr` au niveau juste inférieur se trouvent les noms des nœuds du réseau, qui sont des stations ou des serveurs auxquels leurs utilisateurs ont donné les noms `asja`, `vera`, `salome`, `electre`. Leurs noms complets, uniques pour tout l'Internet et auxquels le DNS aura pour mission de faire correspondre leur adresse IP, seront `asja.sophia.inria.fr`, `vera.sophia.inria.fr`, `salome.sophia.inria.fr`, `electre.sophia.inria.fr`.

Une station sur le réseau peut avoir, outre son nom propre tel que nous venons de le voir, un ou plusieurs alias. Ainsi il est de coutume que le serveur Web d'un organisme soit connu sous le nom `www.quelquechose.fr`. Alors si le serveur Web d'Inria Sophia est hébergé sur la machine `asja`, celle-ci recevra un alias, `www.sophia.inria.fr`. Les deux noms désigneront la même machine, ou plus exactement la même interface sur le réseau.

Il serait possible d'imaginer une administration centralisée de l'arbre des domaines, mais une fraction de seconde de réflexion révélerait l'immensité des difficultés qui en résulteraient. Aussi cet arbre est-il découpé en sous-arbres appelés *zones*, administrées séparément. Ainsi en France l'association AFNIC administre-t-elle tous les domaines dont le nom se termine par `.fr`: on dit que l'AFNIC a reçu *délégation* pour la zone `fr`. De même l'AFNIC déléguera l'administration de la zone `inria.fr` à Inria, qui lui-même déléguera à une équipe de son unité de Sophia-Antipolis l'administration de *sophia.inria.fr*.

Dès lors qu'un organisme a reçu délégation de l'administration d'une zone, il a le devoir de mettre en service des **serveurs de noms** pour cette zone, au moins deux, un primaire et un secondaire [12]. Un serveur de noms est un logiciel que l'on peut interroger: si on lui fournit le nom d'une machine il renvoie son adresse, et vice-versa. Dès qu'un nouvel ordinateur est mis en service dans une zone, l'administrateur du DNS de cette zone doit lui affecter un nom et une adresse et les ajouter à la base de données du serveur de noms primaire local. On dit que ce serveur de noms a l'**autorité** sur la zone.

Un serveur primaire obtient les informations relatives à sa zone en accédant directement les bases de données locales. Un serveur secondaire (il peut y en avoir plusieurs, et il est recommandé qu'ils soient physiquement distincts et redondants) obtient ces mêmes informations en les demandant au serveur primaire. L'opération par laquelle un serveur secondaire reçoit du serveur primaire l'information qui décrit la zone est nommée **transfert de zone**. La pratique courante est de demander à un collègue sur un autre site d'être secondaire pour votre zone, à charge de revanche.

Donc tout système installé dans la zone, lorsqu'il voudra traduire un nom en adresse, posera la question au serveur de la zone. Plus précisément, le logiciel d'application qui a besoin de l'adresse (par exemple votre navigateur WWW ou le logiciel de transfert de courrier électronique) fait appel à un *résolveur*, qui va dialoguer avec le serveur de noms qui lui aura été désigné.

Si le nom à traduire désigne une machine locale, le serveur interrogera directement sa base. Sinon, il doit interroger un autre serveur de noms, qui, lui, connaîtra la réponse. Comment trouver un tel serveur de noms, en possession de la ré-

12. Selon la règle plus récente il faudrait dire « maître » plutôt que primaire et « esclave » plutôt que secondaire, mais cette terminologie m'écorche la bouche.

ponse à la question posée? Chaque serveur connaît (il en possède les adresses dans sa base de données) la liste des **serveurs de noms racines**, à ce jour au nombre de treize, dispersés à la surface de la planète, surtout aux États-Unis qui en abritent dix, et en fait recopiés à des centaines d'exemplaires jusqu'au fond des steppes d'Asie centrale [13]. Ces serveurs racines détiennent la liste des serveurs de noms qui détiennent l'autorité pour tous les domaines de second niveau (dans notre schéma de la figure 6.13, la ligne `ibm.com`, `inria.fr`, etc.).

Notre serveur va donc interroger un serveur racine. Celui-ci répondra en donnant l'adresse du serveur qui détient l'information autorisée relative au domaine de second niveau dont relève le nom de domaine que nous cherchons à résoudre ; ce troisième serveur, interrogé, donnera soit la réponse, soit l'adresse d'un quatrième serveur plus proche du domaine concerné, etc. Le serveur interrogé initialement peut soit transmettre la première réponse au résolveur, charge à ce dernier d'interroger le serveur de noms suivant, et ainsi de suite : une telle interrogation est dite *itérative*. Le résolveur peut au contraire demander au serveur de faire son affaire des interrogations des autres serveurs de noms impliqués, une telle interrogation sera dite *récursive*.

Toute cette subtile conversation entre serveurs sera bien sûr ignorée de l'utilisateur. Les logiciels de courrier électronique ou de navigation sur le WWW savent faire appel au résolveur. Lorsqu'un abonné individuel à l'Internet allume son modem, la plupart du temps le routeur de son FAI lui envoie, grâce au protocole DHCP *(Dynamic Host Configuration Protocol)*, en même temps que son adresse IP dynamique, l'adresse du ou des serveurs de noms auxquels le résolveur pourra s'adresser. Mais ainsi vous saurez à quoi correspond la case la plus perturbante du menu de configuration de votre accès au réseau : celle où on vous demande l'adresse de votre serveur DNS. Heureusement vous n'aurez presque plus jamais à la remplir.

6.5.7 Mécanisme de la couche IP

Nous allons maintenant étudier la façon dont des datagrammes IP qui contribuent à l'acheminement d'un message quelconque atteignent leur destination. Ce transfert de données a pour origine un ordinateur raccordé au réseau, évidemment. Le principe de la couche réseau IP consiste, à chaque nœud du réseau traversé, à déterminer le prochain nœud à atteindre, et plus concrètement sur quelle interface réseau émettre le datagramme. S'il n'y a qu'une interface réseau active, la décision est très simple, comme on peut l'imaginer. Par convention, la couche IP considère aussi une interface dite « locale », qui n'a pas d'existence physique et qui permet à un nœud de s'atteindre lui-même, cas trivial mais qu'il ne faut pas oublier de traiter.

13. Cf. `http://www.root-servers.org/`

Sur le nœud émetteur, le rôle de la couche IP est le suivant :

- recevoir un flux de bits de la couche TCP (transport) ; TCP découpe ce flux en morceaux de taille raisonnable appelés *segments* ;
- mettre chaque segment dans un datagramme (ou exceptionnellement plusieurs) ;
- déterminer l'interface réseau appropriée pour atteindre l'adresse de destination ;
- munir le datagramme d'une en-tête qui comporte les informations nécessaires à son acheminement, c'est-à-dire essentiellement, en ce qui relève de nos préoccupations du moment, l'adresse IP de l'interface émettrice et l'adresse IP du destinataire ;
- remettre le datagramme à la couche liaison de données attachée à la bonne interface, avec si nous sommes sur un réseau local la bonne adresse de couche 2 (MAC).

Effectuer ces opérations, et notamment les trois dernières, c'est déterminer l'itinéraire (en anglais *route*) que doit emprunter le datagramme, soit par extension de langage effectuer le *routage* de ce datagramme.

Chaque station connectée à un réseau IP possède une table de routage, qui contient deux types d'informations : des adresses de réseaux et le moyen de les atteindre. Pour une simple station « feuille » du réseau, le nombre de cas possibles est limité :

- dans le cas d'un réseau auquel la station est directement connectée, le moyen de l'atteindre est le nom de l'interface qui assure cette connexion ;
- pour tout autre réseau, le moyen de l'atteindre est l'adresse du routeur qui y mène.

Fonctionnellement, la différence entre une station ordinaire et un routeur, c'est que le routeur est programmé pour recevoir des paquets sur une interface et les réémettre sur une autre, alors que la station sait juste émettre et recevoir. Lorsqu'un routeur réémet un paquet, il ne modifie pas l'adresse de l'émetteur, qui reste celle de l'émetteur original.

Une fois obtenue l'adresse IP de destination au moyen du DNS (voir section 6.5.6), l'algorithme d'émission d'un datagramme est le suivant :

- Extraire de l'adresse de destination la partie adresse de réseau.
- Chercher dans la table de routage cette adresse de réseau. Quatre cas sont possibles :
  1. l'adresse du réseau de destination figure dans la table de routage et y correspond à un réseau directement connecté : il y a remise directe du

datagramme sur ce réseau par l'interface désignée et le routage est fait (il faudra encore traduire l'adresse IP en adresse MAC par le protocole ARP *(Address Resolution Protocol)*[14] ;
2. le réseau de destination figure dans la table de routage et le moyen de l'atteindre qui y est mentionné est l'adresse d'un routeur : le datagramme est transmis à ce routeur selon le procédé vu pour le cas précédent ;
3. le réseau de destination ne figure pas dans la table de routage, mais la table mentionne un routeur par défaut : le datagramme est transmis à ce routeur ;
4. tout autre cas déclenche une erreur de routage (le trop célèbre message `Network is unreachable`).

Le cas le plus fréquent est bien sûr le cas 3 ! Si nous imaginons le réseau comme un océan qui reçoit des fleuves eux-mêmes dotés d'affluents, le réseau local de notre campus ou de notre université est un peu comme le bassin d'une rivière et de ses affluents : l'itinéraire par défaut pour nos bouteilles à la mer, c'est d'aller vers l'aval et de tomber dans le fleuve qui figure le réseau de notre FAI. Mais une fois dans l'océan, il va bien falloir trouver l'embouchure du fleuve qu'il faudra remonter pour arriver finalement au ruisseau sur la berge duquel repose le serveur WWW que nous voulons atteindre. Y a-t-il un gigantesque routeur central qui posséderait dans ses tables de routage les adresses de tous les réseaux de l'univers ? Ce serait une solution théorique, mais ce que nous avons déjà dit de l'Internet et ce que nous savons de sa vitesse d'évolution nous suggère que cela ne fonctionne pas ainsi : la solution est à la section suivante.

Algorithmes de routage

La solution hypothétique évoquée ci-dessus, d'un routeur central de l'Internet distribuant les datagrammes à tous les réseaux, et que l'on peut raffiner en découpant l'Internet en plaques organisées chacune autour d'un routeur possédant toutes les adresses réseau de la plaque et communiquant avec un routeur central moins monstrueux, possédant les adresses des plaques et le moyen d'attribuer un réseau à une plaque, cela s'appellerait le *routage statique*. C'était la solution retenue par les réseaux X25 du bon vieux temps du minitel et du monopole des réseaux, et c'est une solution utilisable à une échelle pas trop grande,

14. Le principe du protocole ARP est le suivant : la station qui possède une adresse IP et veut connaître l'adresse MAC correspondante (cela ne marche qu'au sein d'un même réseau local, de type Ethernet par exemple) envoie en diffusion générale à toutes les stations du réseau un message qui comporte l'adresse IP en question. La station qui possède cette adresse IP se reconnaît et répond « C'est moi, et voici mon adresse MAC ».

pour un réseau d'entreprise par exemple. Mais pour un réseau de grande taille et dépourvu d'administration centralisée, comme l'Internet, ce ne serait guère réaliste. Ce qui a fait la force de l'Internet, c'est sa capacité à acheminer les paquets à destination dans un réseau en évolution permanente et sans administration centrale, bref le routage dynamique dont nous allons étudier maintenant les principes.

Pour poursuivre notre métaphore fluviale et maritime, les réseaux locaux et de FAI correspondent aux ruisseaux, rivières et fleuves et possèdent tous un routage par défaut simple : si la destination n'est pas locale, elle sera, le plus souvent vers l'Internet, c'est-à-dire au-delà des mers, donc vers l'embouchure (respectivement, vers le routeur de sortie du réseau local).

À leur embouchure sur l'océan de l'Internet, nous pouvons nous imaginer que chaque réseau de FAI possède un routeur, ou plusieurs dans le cas de deltas, chargés de trouver les bons itinéraires pour les paquets qui sortent du réseau ou qui veulent y entrer. Il pourra y avoir aussi des routeurs en pleine mer, chargés d'orienter de grands flux maritimes, vers le Cap Horn ou le détroit de Gibraltar... Les routeurs du delta du Gange ignorent bien sûr le détail des réseaux du bassin de la Méditerranée, détail qui devra en revanche être connu du routeur du détroit de Gibraltar. L'idée du routage dans l'Internet, c'est que tous ces routeurs sont capables d'échanger des informations, et que plus précisément ils informent leurs voisins des réseaux auxquels ils sont directement connectés. Ainsi, une fois notre datagramme [15] tombé dans l'océan, il va aller de routeur en routeur, aucun ne connaissant sa destination finale, mais chacun étant capable de trouver un itinéraire qui l'en rapproche. C'est ce que l'on appelle le *routage dynamique*.

Le routage dynamique, pour être efficace dans un réseau aussi vaste que l'Internet, met en œuvre des protocoles complexes. En fait l'Internet est une confédération de réseaux IP, mais il existe pour l'organisation du routage un niveau d'agrégation intermédiaire, le *Système Autonome (Autonomous System, AS)*, qui est un regroupement de réseaux qui peuvent être vus de l'extérieur comme une entité pourvue d'une autorité administrative unique. Ainsi, sauf pour ceux qui n'ont pas bien compris le fonctionnement de l'Internet, chaque FAI et ses clients apparaîtront comme un seul AS. Des tables de routage globales seront échangées entre AS. Au sein d'un AS seront utilisés des protocoles plus simples et des tables de routage plus petites, étant donné qu'un client désireux d'envoyer un paquet à une adresse extérieure à l'AS la remettra à un routeur de son FAI, qui, lui, possédera les tables de routages globales. Après tout, lorsque nous mettons une carte postale à la boîte, nous nous attendons à ce que la Poste

15. Rappelons qu'un datagramme IP c'est un paquet, à la fragmentation près : si un datagramme est fragmenté, c'est chaque fragment qui est un paquet. La fragmentation tombe en désuétude.

française sache comment la faire parvenir à la Poste vénézuélienne, qui elle saura trouver notre correspondante à Caracas.

Le protocole global de communication de tables de routages d'AS à AS est BGP *(Border Gateway Protocol)*. Il y a plusieurs protocoles de routage dynamique au sein d'un AS ou d'un réseau, celui qui tend aujourd'hui à être le plus utilisé est OSPF *(Open Shortest Path First)*, qui repose sur un algorithme de recherche de parcours dans un graphe dû à Dijkstra en 1959 (encore lui! inutile de dire qu'à cette époque il ne soupçonnait pas l'usage qui serait fait de son algorithme). OSPF a supplanté d'autres protocoles parce qu'il donne de meilleurs résultats, mais cette supériorité est au prix d'une complexité élevée. Pour ceux de nos lecteurs qui ne se destinent pas à la profession d'ingénieur réseau, nous exposerons un protocole plus simple et encore souvent utilisé dans de petits réseaux, RIP (pour *Routing Information Protocol*), qui repose sur l'algorithme de Bellman-Ford, imaginé en 1957 par Richard Bellman et doté d'une version distribuée en 1962 par Lestor R. Ford Jr et D. R. Fulkerson [16]. Comme beaucoup d'algorithmes utilisés dans le monde des réseaux, il est issu du domaine de la recherche opérationnelle, et appartient à la famille des algorithmes de calcul de chemin le plus court dans un graphe par une méthode du type « vecteur de distance », par opposition à OSPF qui appartient à la famille des méthodes de calcul « par l'état des liaisons ».

Les méthodes de calcul de chemin le plus court dans un graphe « par l'état des liaisons » comme OSPF imposent que chaque routeur possède dans ses tables la topographie et la description de l'ensemble du domaine de routage, et que toute cette information soit retransmise à travers tout le domaine chaque fois qu'elle subit une modification. En revanche avec les méthodes de calcul « par vecteur de distance » comme RIP chaque routeur ne maintient que l'information qui le concerne lui-même et celle relative à ses voisins immédiats. On conçoit qu'OSPF ait attendu pour se généraliser une époque de débits élevés et de mémoire bon marché, et que RIP ait eu plus de succès dans la période précédente.

Le but d'un algorithme de routage est de trouver le chemin le plus court entre deux points d'un graphe (respectivement d'un réseau). En termes de réseaux informatiques, « court » ne désigne pas vraiment une distance en termes de longueur, mais plutôt en termes de débit de liaison et de nombre de nœuds traversés ; une distance faible désignera une liaison rapide avec peu de routeurs, une distance élevée une liaison lente ou qui traverse de nombreux routeurs.

Le principe de fonctionnement de RIP est le suivant : chaque routeur du réseau propage sur toutes ses interfaces des vecteurs de distance, qui constituent en fait le résumé de sa table de routage. Initialement (c'est-à-dire à la mise sous tension) un routeur ne connaît qu'un itinéraire, celui qui mène à lui-même, avec

16. BGP utilise également l'algorithme de Bellman-Ford.

une distance nulle. Mais en propageant cette table de routage élémentaire il va permettre à ses voisins d'apprendre son existence; lui-même apprendra de la même façon l'existence de ses voisins, et des voisins de ses voisins; ainsi au fur et à mesure les tables de routage des uns et des autres s'enrichiront. Ce que nous démontrent MM. Bellman, Ford et Fulkerson, c'est que cet algorithme converge, c'est à dire qu'à l'issue d'un certain nombre d'échanges de tables de routage le système constitué par ce réseau de routeurs atteindra un état stable, où l'envoi de nouvelles informations de routage ne provoquera plus aucune modification des tables.

Un routeur est capable de tester ses interfaces, et notamment de détecter la présence ou l'absence d'une interface qui répond à l'autre extrémité. En cas de coupure inopinée d'une liaison, les routeurs concernés la détectent, recalculent leurs tables de routage en affectant une distance infinie à la destination précédemment atteinte par la liaison coupée, et l'algorithme de propagation est exécuté à nouveau, jusqu'à l'obtention d'un nouvel état stable.

Le lecteur curieux de ces questions consultera avec profit le livre de Christian Huitema *Routing in the Internet* [64].

Calcul des tables de routage

Nous raisonnerons sur un réseau très simple à cinq nœuds (cinq routeurs) tel que représenté par la figure 6.14. Notre exercice de routage doit beaucoup au livre de Christian Huitema *Routing in the Internet* [64] (avec son aimable autorisation), qui donne une description complète de ces problèmes et de leurs solutions.

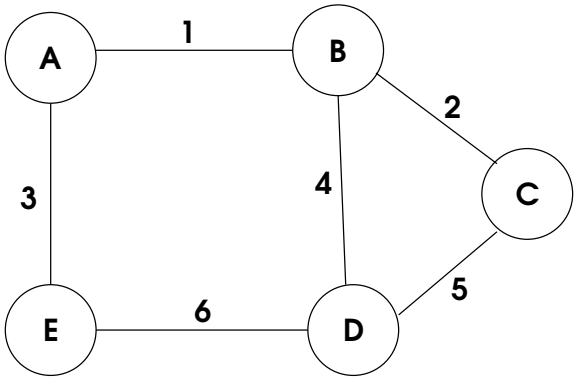


Figure 6.14 : *Un réseau à cinq routeurs.*

Chaque arc du graphe est flanqué d'un numéro d'identification de liaison. Nous supposons le graphe non orienté, et la distance de **A** à **B** égale à la distance

de **B** à **A**. Chaque arc correspond à une liaison dont la distance entre extrémités vaut 1. Examinons maintenant les tables de routage de chaque routeur à l'initialisation du système, par exemple lors de la mise sous tension des cinq routeurs. Soit par exemple la table de routage de **A** :

de **A** vers	liaison	distance
A	locale	0

Lors de son initialisation **A** ne possède dans sa table de routage qu'un itinéraire vers lui-même par l'interface locale, avec une distance nulle. On suppose qu'un routeur connaît sa propre adresse et ses interfaces actives, mais au démarrage il ignore ce qu'il y a derrière les interfaces. Le vecteur de distance de **A**, à cet instant, est très simple :

A=0

Peu complexé par l'indigence de cette information, **A** va l'émettre sur toutes ses interfaces à l'usage de ses voisins immédiats, en l'occurrence **B** et **E**. Ainsi **B** reçoit ce vecteur par la liaison 1, et ajoute à toutes les distances reçues le coût de la liaison 1, que nous avons supposé égal à 1, ce qui transforme le message en **A=1**. Puis **B** examine sa table pour voir s'il dispose déjà d'une information de liaison vers **A** ; comme ce n'est pas le cas il va juste introduire cette nouvelle donnée. La table de routage de **B**, qui avant de recevoir le message de **A** était ceci :

de **B** vers	liaison	distance
B	locale	0

devient cela :

de **B** vers	liaison	distance
B	locale	0
A	1	1

Comme **B** vient d'enrichir sa table de routage, il va émettre son propre vecteur de distance à destination de ses voisins **A**, **C** et **D** par les liaisons 1, 2 et 4 :

B=0, A=1

Pendant ce temps **E** aura reçu le message de **A**, aura effectué les mêmes opérations de mise à jour de sa table de routage *(mutatis mutandis)* et transmettra son vecteur de distance à **A** et **D** sur les liaisons 3 et 6 :

E=0, A=1

Supposons que **A** reçoive le message de **B** avant celui de **E**; il ajoute 1 à toutes les distances, qui deviennent donc **B=1, A=2** et les compare à celles qui figurent dans sa table. Comme la distance **A=2** est supérieure à celle qui figure déjà dans la table, il n'insère que l'information relative à **B**, puis il reçoit le message de **E**, effectue les mêmes calculs et sa table devient :

de **A** vers	liaison	distance
A	locale	0
B	1	1
E	3	1

C reçoit par la liaison 2 le message **B=0, A=1** ; toujours selon le même algorithme il ajoute 1 à toutes les distances et met à jour sa table qui devient :

de **C** vers	liaison	distance
C	locale	0
B	2	1
A	2	2

D aura reçu de **B** le même message que **C** par la liaison 4, puis par la liaison 6 un message de **E**. Ces messages comportent chacun une destination vers **A**, avec des distances équivalentes ; nous supposerons que la première information est retenue et la table de **D** devient :

de **D** vers	liaison	distance
D	locale	0
B	4	1
A	4	2
E	6	1

A, **C** et **D** ont donc de nouvelles tables de routage, ils vont en déduire de nouveaux vecteurs de distance qu'ils vont diffuser à leurs voisins. De ce fait **B**, **D** et **E** vont mettre à jour leurs tables de routage et diffuser de nouveaux vecteurs qui vont provoquer des mises à jour des tables de **A**, **C** et **E**. Nous invitons le lecteur à effectuer ces calculs à titre d'exercice délassant, et à constater qu'à l'issue de ces opérations le système a atteint un état stable, c'est-à-dire que l'envoi des nouveaux vecteurs de distance résultant de la dernière mise à jour ne provoquera aucune modification des tables de routage : on dit que l'algorithme a convergé.

Reconfiguration en cas de coupure de liaison

Un des avantages attendus du routage dynamique, c'est que le réseau soit capable de se reconfigurer automatiquement en cas de modification inopinée de sa topologie. Supposons donc que la pelleteuse traditionnelle et canonique

coupe soudain la liaison 1, avec pour résultat la topologie représentée par la figure 6.15.

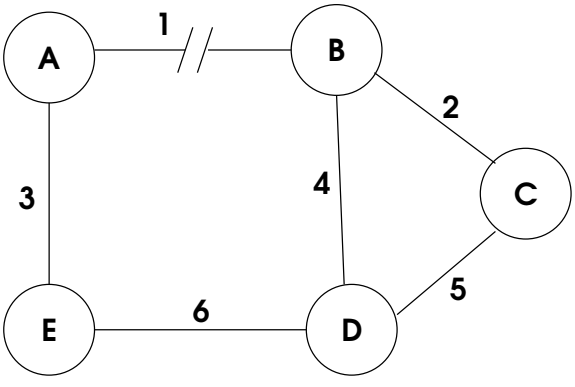


Figure 6.15 : *Rupture de liaison !*

Les interfaces correspondantes des nœuds concernés, **A** et **B**, vont détecter l'incident ; **A** et **B** vont affecter à la liaison 1 un coût infini (∞) et mettre à jour leurs tables de routage en notant une distance infinie pour toutes les destinations atteintes par la liaison 1 :

de **A** vers	liaison	distance
A	locale	0
B	1	∞
C	1	∞
D	1	∞
E	3	1

de **B** vers	liaison	distance
A	1	∞
B	locale	0
C	2	1
D	4	1
E	1	∞

A va calculer et émettre un nouveau vecteur de distance sur la liaison 3 :

A=0, B=∞, C=∞, D=∞, E=1

et **B** sur les liaisons 2 et 4 :

A=∞, B=0, C=1, D=1, E=∞

E va recevoir le message de **A**, **C** et **D** celui de **B**, et ils vont mettre leurs tables à jour en fonction des nouvelles distances :

de **E** vers	liaison	distance
A	3	1
B	3	∞
C	6	2
D	6	1
E	locale	0

de **C** vers	liaison	distance
A	2	∞
B	2	1
C	locale	0
D	5	1
E	5	2

de **D** vers	liaison	distance
A	4	∞
B	4	1
C	5	1
D	locale	0
E	6	1

À l'issue de ces mises à jour **C**, **D** et **E** vont émettre des vecteurs de distance :

C émet : **C=0, B=1, A=∞, D=1, E=2** sur les liaisons 2 et 5 ; **D** émet : **D=0, B=1, A=∞, E=1, C=1** sur les liaisons 4, 5 et 6 ; **E** émet : **E=0, A=1, B=∞, D=1, C=2** sur les liaisons 3 et 6.

qui vont à leur tour déclencher les mises à jour des tables de **A**, **B**, **D** et **E** :

de **A** vers	liaison	distance
A	locale	0
B	1	∞
C	3	3
D	3	2
E	3	1

de **B** vers	liaison	distance
B	locale	0
A	1	∞
C	2	1
D	4	1
E	4	2

de **D** vers	liaison	distance
D	locale	0
A	6	2
B	4	1
C	5	1
E	6	1

de **E** vers	liaison	distance
E	locale	0
A	3	1
B	6	2
C	6	2
D	6	1

qui vont à nouveau émettre des vecteurs de distance :

A émet : **A=0, B=∞, C=3, D=2, E=1** sur la liaison 3 ; **B** émet : **A=∞, B=0, C=1, D=1, E=2** sur les liaisons 2 et 4 ; **D** émet : **D=0, B=1, A=∞, E=1, C=1** sur les liaisons 4, 5 et 6 ; **E** émet : **E=0, A=1, B=∞, D=1, C=2** sur les liaisons 3 et 6.
ce qui va déclencher la mise à jour des tables de **A**, **B** et **C** :

de **A** vers	liaison	distance
A	locale	0
B	3	3
C	3	3
D	3	2
E	3	1

de **B** vers	liaison	distance
A	4	3
B	locale	0
C	2	1
D	4	1
E	4	2

de **C** vers	liaison	distance
A	5	3
B	2	1
C	locale	0
D	5	1
E	5	2

Ces nœuds vont à nouveau émettre des vecteurs de distance, mais qui n'apporteront aux destinataires aucune distance vers un nœud plus courte que celles qu'ils possèdent déjà, et qui donc ne déclencheront aucune modification des tables. Cet exercice inspiré de Christian Huitema conduit à un nouvel état stable

qui assure la connectivité générale, ce que nous promettaient MM. Bellman, Ford et Fulkerson.

Problèmes de routage

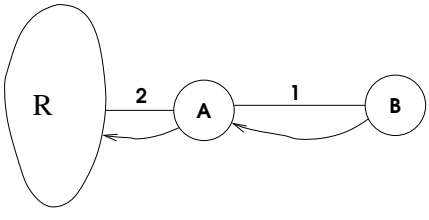


Figure 6.16 : *Avant le rebond*

Dans de nombreux cas, RIP réussit brillamment à reconfigurer automatiquement un réseau endommagé et à lui rendre sa connectivité. RIP doit aussi éviter des pièges, comme celui tendu par la situation illustrée par la figure 6.16, où l'on voit, à l'état initial, un nœud B qui accède au réseau **R** par l'intermédiaire du nœud A.

Supposons maintenant que la liaison 2 entre A et **R** soit coupée inopinément (figure 6.17) : A connaissait un itinéraire vers **R** par cette liaison 2, il ne peut plus désormais essayer de faire passer ses paquets que par la liaison 1 vers B, ce qui est clairement sans espoir. Avec un algorithme trop naïf, A peut envoyer ses paquets à B, qui dirait « oui, je connais un excellent itinéraire vers **R**, via A », et acheminerait les paquets vers A, qui les renverrait vers B, et ainsi de suite... En fait, la convergence de l'algorithme dépend de qui envoie son vecteur de distance le premier :

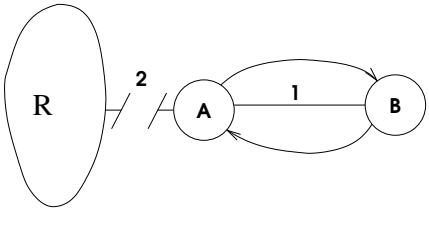


Figure 6.17 : *Rebond !*

– si A émet son vecteur de distance avant que B ait pu le faire, l'information de la coupure de la liaison 2 sera effectivement correctement propagée ;

– si B émet son vecteur de distance entre l'instant où A détecte la coupure de liaison et celui où A aurait effectivement dû émettre le sien, par exemple parce qu'il a reçu une mise à jour en provenance d'un autre point du réseau, A va accepter la vision de B et nous aurons une situation de boucle ou rebond.

Une telle situation ne peut être évitée que par une convention. À chaque échange de vecteur de distance, la distance de A à **R** croît de 2 par le mécanisme suivant :

– lorsque A constate la rupture de liaison, il mentionne dans sa table de routage une distance infinie vers **R** ;
– si à cet instant A reçoit de B un vecteur qui indique pour **R** une distance 2, il constate que cette valeur est inférieure à celle contenue dans sa table, et la met à jour avec pour **R** la liaison 1 et la distance 2+1=3, puis diffuse son vecteur de distance ;
– B reçoit le vecteur de A et apprend que son itinéraire vers **R** a maintenant une distance de 3 : il lui ajoute 1 et diffuse son vecteur avec la valeur 3+1=4…

Pendant tout ce temps, les paquets envoyés de A ou de B vers **R** seront routés de A à B puis de B à A et vice versa…

Pour éviter ce processus de boucle infinie, on fixera une valeur élevée qui sera considérée, par convention, égale à l'infini, et quand la distance atteindra cette valeur la destination sera réputée impossible à atteindre. Dans ce contexte, l'infini pourra être fixé à 32, par exemple. Les paquets seront aussi dotés d'un paramètre « durée de vie » *(TTL, Time to live)*, à savoir un nombre maximum de nœuds qu'ils auront le droit de traverser avant d'être purement et simplement abandonnés.

6.5.8 Nouvelles tendances IP

Nous l'avons dit, le protocole IP est entré dans une période de transition de la version 4 à la version 6 (la version 5 n'a jamais été déployée). Compte tenu du nombre considérable d'installations concernées, cette transition sera longue et les deux versions sont appelées à cohabiter pendant des années. Les routeurs du cœur de l'Internet (les *core routers*) seront bien sûr appelés les premiers à pouvoir traiter simultanément les deux versions, ils le font déjà d'ailleurs, cependant que la migration des stations de travail de base n'est pas vraiment urgente. Il existe un RFC qui précise comment encapsuler une adresse v4 au format v6.

IPv6, outre le nouveau format d'adresses qui en est l'aspect le plus spectaculaire, comporte d'autres nouveautés qui vont donner des solutions techniquement correctes à des problèmes que la version 4 résout par des artifices fragiles. Parmi les artifices employés par IPv4 pour faire face à la pénurie d'adresses, nous

avons cité CIDR *(Classless Interdomain Routing)* et NAT *(Network Address Translation)*. Le nouvel espace d'adressage offert par les 128 bits de l'adresse IPv6 mettra un terme à ces acrobaties, et de toute façon l'ancienne notion de classe disparaît.

IPv6 introduit également de nouveaux protocoles de sécurité désignés collectivement par le terme IPSec. En fait IPSec intervient au niveau de la couche transport (TCP), mais IPv6 le rend obligatoire, cependant que des adaptations permettent de l'introduire avec IPv4. IPSec définit en gros deux classes de services, une destinée à l'authentification des parties, l'autre au chiffrement.

Le support des nœuds mobiles est un autre problème apparu après la conception d'IPv4, et qui avait dû être résolu par des adjonctions plus ou moins satisfaisantes. La question intervient au stade du routage : la station mobile établit une communication avec une station fixe, qui n'est par définition pas toujours la même. L'émission depuis la station mobile vers une adresse IP quelconque ne pose aucun problème particulier, ce qui est inédit c'est la façon d'atteindre la station mobile depuis un point quelconque de l'Internet. La solution utilisée sous IPv4, nommée *mobile IP*, assez expérimentale, est incorporée à IPv6 et généralisée.

La configuration d'une station sous IPv4 était souvent une opération manuelle laborieuse par manque de possibilités d'auto-configuration. Quel utilisateur n'a pas blêmi devant un écran où un logiciel lui demandait son adresse IP et, pire, celle de son serveur de noms ? Le déploiement de DHCP *(Dynamic Host Configuration Protocol)* a contribué à résorber ce problème, et cette solution sera généralisée avec IPv6.

6.5.9 En quoi IP est-il supérieur à X25 ?

Nous avons déjà vu quelques éléments de réponse à cette question, dont nous allons faire ici la synthèse.

Invention de la transmission par paquets

La transmission de données par paquets sur un réseau a été inventée indépendamment par Paul Baran de la *Rand Corporation* (un organisme de recherche à but non lucratif sous contrat avec le Département de la Défense américain), par Leonard Kleinrock de l'université Stanford et par Donald Davies du *National Physical Laboratory* britannique, au début des années 1960. On se reportera au livre de Janet Abbate [1] pour une discussion des questions d'antériorité.

Lawrence Roberts, à l'origine (à la suite de Robert Taylor) du réseau ARPANET, créé en 1969 par la firme Bolt, Beranek & Newman (BBN) pour l'*Advanced Research Projects Agency* (ARPA) du Département de la Défense américain, avait eu connaissance des techniques de transmission par paquets en rencontrant Roger Scantlebury, un collaborateur de Donald Davies, lors d'une conférence sur les réseaux à Gatlinburg,

Tennessee, en octobre 1967. Convaincu par l'idée, il allait la mettre en œuvre dans ARPANET. Si ARPANET était bien une émanation du Département de la Défense, son usage n'était pas militaire, mais destiné au partage de moyens de calculs entre laboratoires de recherche, le plus souvent universitaires, sous contrat avec l'ARPA. Janet Abbate décrit de façon nuancée les équilibres délicats entre administrateurs militaires et chercheurs d'esprit plutôt anarchiste, ce en pleine période de guerre du Viêt-Nam.

Commutation de circuits

Dans les réseaux de télécommunications anciens, l'établissement d'une communication consistait à établir un circuit physique continu (en cuivre) entre les deux extrémités. Le circuit ainsi établi était physiquement réservé à la communication en cours. Puis furent inventées des techniques de multiplexage destinées à permettre le partage de liaisons physiques entre plusieurs communications : le multiplexage de fréquence partageait le spectre disponible en plages de fréquences attribuées chacune à une communication, le multiplexage temporel leur attribuait des intervalles de temps successifs.

Comme mentionné plus haut, la transmission par paquets permettait un multiplexage plus souple et plus efficace, grâce à la présence dans chaque paquet de l'adresse de destination, qui permettait de mélanger des paquets de diverses communications sans contraintes particulières, dans la limite du débit maximum de chaque tronçon de réseau emprunté.

Les premiers réseaux de paquets fonctionnaient selon un principe de *circuits virtuels* : pour une transmission donnée, les ordinateurs de commande du réseau, que l'on n'appelait pas encore des routeurs, calculaient un itinéraire, et tous les paquets de ce flux empruntaient cet itinéraire. La technique des circuits virtuels plaçait les opérations de calcul d'itinéraire et de contrôle d'acheminement au cœur du réseau, dans ses systèmes de supervision qui effectuaient l'aiguillage (la *commutation*) des flux de données.

Le réseau ARPANET, le réseau du *National Physical Laboratory* britannique, ainsi que les réseaux X25 comme le réseau français Transpac, étaient des réseaux à commutation de circuits virtuels.

L'année charnière : 1972

1972 est une année charnière dans le monde des réseaux. Vinton Cerf et Robert Kahn étaient les chefs de file du développement d'ARPANET, qui reliait 29 nœuds. L'IRIA français, ancêtre d'Inria, avait lancé le réseau Cyclades sous la direction de Louis Pouzin. Le *National Physical Laboratory* britannique avait un réseau conçu par Donald Davies. Les PTT de plusieurs pays européens (dont la France) avaient aussi des projets de réseaux.

En octobre 1972 tous ces groupes se réunirent à Washington pour la première Conférence internationale sur les communications informatiques, avec l'idée, entre autres, d'interconnecter leurs réseaux. Cette réunion fut le lieu d'intenses échanges d'idées et d'expériences, à l'origine de la conception par Cerf et Kahn du protocole TCP[17] en 1974 (la couche réseau n'apparaîtra comme un protocole distinct de TCP sous le nom d'IP[18] qu'en 1978), et aussi du modèle OSI [139] en sept couches publié par un article fameux [142] d'Hubert Zimmerman (1941-2012).

Le modèle OSI a connu fort peu de réalisations (à ma connaissance la plus complète fut DECNET de Digital Equipment) et encore moins de déploiements significatifs, mais par contre il a fourni un cadre de référence conceptuel inégalé et toujours d'actualité. Ce n'est pas moi qui le dis, c'est Robert Kahn : les auteurs du modèle OSI « ont donné aux gens un moyen de se représenter les protocoles sous forme de couches. Nous avions sûrement déjà cela à l'esprit, mais nous ne l'avions jamais formulé de cette façon, et ils l'ont fait. » [67]. Un des enjeux débattus était la possibilité d'interconnecter des réseaux différents, ce qui allait mener à l'idée d'Internet.

Je traduis le passage de Janet Abbate qui commente la contribution de l'équipe Cyclades à cette conférence (p. 171 de l'édition électronique) : « Cyclades était un projet de réseau expérimental lancé en 1972 avec un financement du gouvernement français. Ses architectes, Louis Pouzin et Hubert Zimmermann, avaient des idées très tranchées sur l'interconnexion de réseaux [*(internetworking)*]. En fait, Cyclades, à la différence d'ARPANET, avait été conçu explicitement pour permettre l'interconnexion ; il pouvait, par exemple, traiter des adresses de formats différents et des niveaux de service variés [103, p. 416].

« Cyclades était basé sur un système très simple de commutation de paquets. Plutôt que de confier au réseau la tâche de maintenir une connexion stable entre deux extrémités [*hosts*, hôtes], à la façon d'ARPANET, Cyclades émettait simplement des paquets individuels (sous le nom de "datagrammes"). L'argument de Pouzin et Zimmermann était que plus les fonctions du réseau resteraient simples, plus la construction d'un internet serait facile. Selon Pouzin [103, p. 429], "plus un réseau sera perfectionné, moins il sera facile de l'interfacer correctement avec un autre. En particulier, toute fonction autre que l'émission de paquets sera selon toute probabilité trop spécifique pour communiquer correctement avec un voi-

17. TCP, pour *Transport Control Protocol*, est conformément à son nom le protocole qui spécifie comment un message, éventuellement constitué de nombreux paquets, est acheminé de bout en bout (de l'émetteur au récepteur), complet et sans erreur.

18. IP, pour *Internet Protocol*, est le protocole de réseau, qui spécifie comment est calculé, pour chaque paquet de données, l'itinéraire selon lequel il sera acheminé à destination. La séparation de TCP et d'IP permet de laisser à TCP le soin du contrôle d'intégrité des données, cependant qu'IP ne s'occupe que du calcul d'itinéraire.

sin." Afin de cantonner les fonctions du réseau au strict minimum, les chercheurs français soutenaient qu'il était nécessaire d'attribuer la responsabilité de maintenir des connexions stables au protocole d'extrémité [hôte]. Cela allait à l'encontre à la fois de la façon dont BBN avait conçu l'ARPANET et de celle dont les opérateurs de télécommunications en France et ailleurs prévoyaient de construire leurs réseaux publics de données. Sans doute en prévision d'une opposition à leur approche non conventionnelle, les membres du groupe Cyclades défendaient leur philosophie d'interconnexion avec une extrême vigueur. Pouzin et Zimmermann étaient actifs au sein de l'INWG [*(International Packet Network Working Group)*]. Un autre membre de l'équipe Cyclades, Gérard Le Lann, travaillait dans le labo de Cerf à Stanford, où il avait la possibilité de participer directement à la conception du système internet de l'ARPA. Selon Cerf [30], le groupe Cyclades "contribuait beaucoup aux premières discussions pour savoir à quoi ressemblerait le [protocole d'extrémité]." »

L'émission de datagrammes indépendants les uns des autres, simplement munis de leurs adresses d'émission et de destination, laisse à chaque routeur la décision de la prochaine étape dans le réseau. Les différents datagrammes d'une même transmission peuvent emprunter des itinéraires différents et arriver à destination dans le désordre. On laisse à la charge du protocole d'extrémité (par exemple TCP) le soin de vérifier qu'ils sont tous arrivés, sans erreur, et de les mettre dans le bon ordre. C'est une différence essentielle avec les protocoles dits « connectés », tels que X25. Ou en d'autres termes, l'équipe Cyclades proposait un protocole de datagrammes plutôt qu'un protocole de messages.

Commutation de paquets

L'Internet est de toute évidence une œuvre collective, il n'y a pas *un* unique inventeur de l'Internet, mais il n'y en a pas non plus une multitude. Ce qui ressort de la lecture d'historiennes spécialistes du domaine comme Janet Abbate et Valérie Schafer ou d'acteurs de ces événements comme Vinton Cerf, Robert Kahn et Alexander McKenzie [84], c'est que si Cerf et Kahn furent bien les maîtres d'œuvre de la transition d'ARPANET à l'Internet que nous connaissons et de la réalisation de TCP puis d'IP, l'invention du datagramme, qui allait à contre-courant de tous les projets et de toutes les réalisations de l'époque, et qui a joué un rôle déterminant dans la constitution du réseau tel que nous le connaissons, est bien le fait de l'équipe Cyclades dirigée par Louis Pouzin.

Louis Pouzin avait eu une autre idée, malheureusement non retenue à l'époque. L'adresse IP d'un nœud du réseau sert à la fois à l'identifier et à le localiser. Pouzin avait suggéré de séparer ces deux fonctions, mais Cerf et Kahn avaient trop avancé dans la voie qu'ils s'étaient fixée et n'ont pas voulu revenir en arrière. Cette confusion des fonctions d'identification et de localisation a de nombreux inconvénients, notamment pour la sécurité du réseau, surtout depuis la généralisation des mo-

biles, et pour la contourner les opérateurs du réseau sont contraints à d'inélégantes gymnastiques. Plusieurs projets existent pour corriger cette situation, mais modifier de façon significative le fonctionnement de l'Internet en son cœur a de quoi faire fléchir les âmes les mieux trempées.

Le datagramme allait permettre de s'affranchir d'une gestion lourde du réseau, au profit d'un réseau plus simple et du report de la complexité dans les protocoles d'extrémité (par exemple TCP). Cette simplification a permis l'extraordinaire croissance de l'Internet, sans avoir à remettre fondamentalement en cause l'architecture de la couche réseau (IP). D'autre part, c'est le fait de réserver aux protocoles d'extrémité la plus grande partie de la complexité qui a permis le développement sur le réseau de multiples protocoles spécialisés, et ce, encore une fois, sans remise en cause de l'architecture générale. C'est en cela que le datagramme est une invention extraordinaire, qui a permis le succès non moins extraordinaire de l'Internet, et dont nous sommes redevables à l'équipe Cyclades dirigée par Louis Pouzin.

6.6 Couche 4, transport

La couche transport a le numéro 4 dans le modèle OSI ; elle est la troisième couche de TCP/IP. Nous nous intéresserons essentiellement à TCP/IP, qui en fait propose le choix entre deux modèles de transport : UDP *(User Datagram Protocol)*, protocole simple non fiable sans état, et TCP *(Transmission Control Protocol)*.

6.6.1 TCP *(Transmission Control Protocol)*

Le protocole de transport TCP (couche 4 du modèle OSI) fournit aux protocoles de niveau application (HTTP comme *HyperText Transfer Protocol* pour le WWW, SMTP comme *Simple Mail Transfer Protocol* pour le courrier électronique, H323 pour la visioconférence, etc.) un flux de bits fiable de bout en bout au-dessus de la couche IP.

Pour obtenir ce résultat, TCP est un protocole en mode connecté, par opposition à IP qui est un protocole sans connexion, c'est-à-dire que tous les segments TCP qui correspondent au même échange de données sont identifiés comme appartenant à une même connexion.

Connexion

TCP découpe les messages qui lui sont remis par les protocoles de la couche application en segments. Chaque segment sera ensuite remis à la couche réseau (IP) pour être inclus dans un datagramme IP. La taille d'un segment est donc calculée de façon à ce qu'il puisse tenir dans un datagramme, c'est-à-dire que

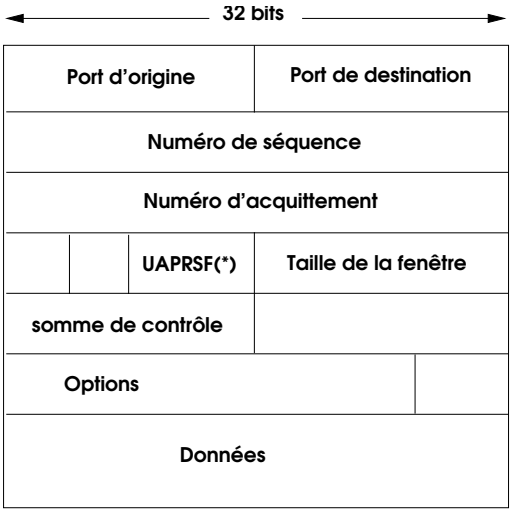


Figure 6.18 : *Segment TCP*

la taille maximum d'un segment est égale à la taille maximum d'un datagramme diminuée de la longueur des en-têtes de datagramme.

Une connexion est totalement identifiée par ses adresses d'origine et de destination (inscrites dans les en-têtes de ses datagrammes IP) et par ses numéros de ports [19] d'origine et de destination, qui sont inscrits dans les en-têtes de ses segments TCP. L'association d'une adresse IP et d'un numéro de port sur le même nœud constitue une extrémité de connexion.

En fait le mécanisme des sockets réseau repose sur le mécanisme des sockets du noyau, qui est un mécanisme de communication entre processus. Mais il n'est pas illogique qu'un mécanisme de communication en réseau se traduise au bout du compte par une communication entre processus.

19. Le terme port ne doit pas suggérer une métaphore portuaire : il s'agit en fait d'un numéro conventionnel, qui identifie éventuellement un protocole de niveau application, ainsi port 25 pour le protocole SMTP de courrier électronique, port 80 pour le Web, port 22 pour le protocole de communication chiffrée SSH. En anglais *port* signifie sabord, lumière dans le piston d'un moteur deux-temps, bref un orifice par lequel peut s'écouler un flux. Comme le flux des données d'une communication selon un protocole donné. Un port est choisi lors de l'ouverture d'une *socket* (douille, ou prise), ce qui complète la métaphore de l'établissement d'un tuyau entre deux systèmes communicants, comme le camion citerne et la cuve à mazout. Voir aussi la note 11 p. 164.

Modèle client-serveur et numéros de port

Les numéros de port sont des identifiants conventionnels. Selon le modèle client-serveur implicite dans ce type d'accès au réseau, un client actionné par un utilisateur (navigateur WWW, logiciel de courrier électronique) demande un service à un serveur distant (serveur WWW, serveur de courrier, serveur de fichiers...). Du côté du serveur, le service demandé est identifié par un numéro de port conventionnel, connu et habituel, en un mot *réservé* : 80 pour un serveur WWW, 25 pour un serveur de courrier électronique, 53 pour un serveur de noms). Du côté du client, TCP attribue à la demande de connexion un numéro de port arbitraire, non encore utilisé. Ainsi, il est possible d'établir entre deux nœuds plusieurs connexions vers un même service sans qu'elles se confondent : elles auront même adresse de serveur, même adresse de client, même port de serveur, mais des ports de client différents.

Poignée de main en trois étapes *(three-way handshake)*

Avant tout transfert de données, TCP ouvre donc une connexion, ce qui se passe selon la procédure suivante appelée poignée de main en trois étapes *(three-way handshake)* (voir figure 6.19) :

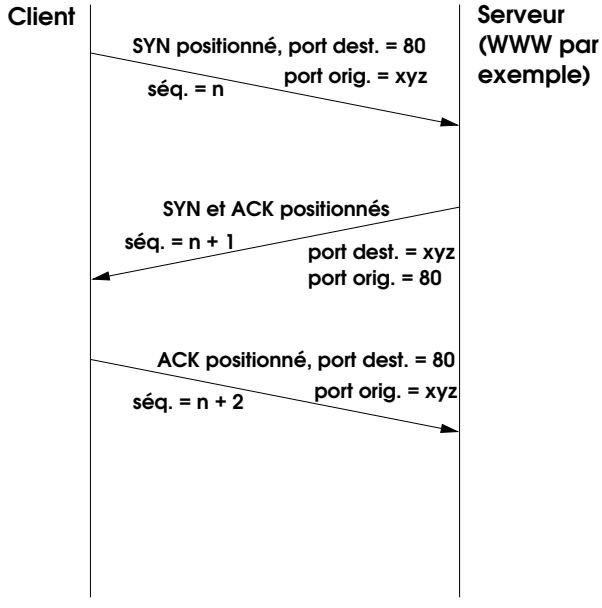


Figure 6.19 : *Poignée de mains en trois temps*

1. Le nœud à l'origine de la demande de communication (appelé communément le **client**) émet un segment TCP avec le bit **SYN** positionné, le numéro de port du **serveur** avec lequel le client veut communiquer dans le champ **port de destination** de l'en-tête de segment, un numéro de port arbitraire dans le champ **port d'origine**, les adresses d'origine et de destination convenables dans l'en-tête de datagramme. Le numéro de séquence est également initialisé dans l'en-tête de segment.
2. Le serveur répond en acquittant ce message au moyen d'un segment dont le bit **SYN** est lui aussi positionné, le bit **ACK** positionné également, les numéros de ports et adresses d'origine et de destination logiquement inversés par rapport au segment du client.
3. Le client acquitte lui-même ce message en renvoyant un segment avec le bit **ACK** positionné. À l'issue de cet échange en trois temps, client et serveur sont réputés s'être mis d'accord sur les numéros de ports nécessaires à l'établissement de la connexion.

Contrôle de flux et évitement de congestion

TCP est un protocole fiable de bout en bout au-dessus d'une couche réseau non fiable, c'est-à-dire qu'il assure entre les deux stations qui communiquent en dernière analyse le même type de sûreté que le protocole de liaison de données assure entre deux nœuds adjacents.

Pour garantir l'absence de pertes et le bon ordre de remise des segments, TCP utilise un algorithme de fenêtre glissante tout à fait similaire à celui que nous avons décrit pour la couche liaison de données à la section 6.4.2, que nous vous invitons de ce fait à relire.

Nous avions dit en décrivant cet algorithme de fenêtre glissante pour la couche 2 qu'il permettait un contrôle de flux, c'est-à-dire l'adaptation mutuelle du débit d'émission de l'émetteur et du débit de réception du récepteur par l'accroissement ou la diminution de la largeur de la fenêtre d'émission. Cette propriété de l'algorithme est bien sûr conservée si on l'applique à la couche transport, mais au lieu d'agir sur une liaison entre deux nœuds adjacents, le contrôle agit désormais à travers tout l'Internet. D'ailleurs, comme TCP est un protocole bi-directionnel, chaque extrémité de la connexion possède deux fenêtres, une en émission et une en réception.

L'application du contrôle de flux à grande distance produit des effets puissants mais à manier avec précautions : les implémentations modernes de TCP utilisent la technique dite du « démarrage lent » pour éviter de saturer un routeur surchargé à un point quelconque de l'itinéraire. La connexion démarre avec une fenêtre dont la largeur correspond à un seul segment. Si tout se passe bien (les acquittements **ACK** arrivent), la fenêtre d'émission est élargie progressivement. Si

une fois la connexion établie en régime permanent des pertes de segments sont constatées, ce qui indique une congestion quelque part sur le trajet, la fenêtre sera à nouveau rétrécie.

Cette politique de démarrage lent et d'évitement de la congestion joue un rôle capital dans le fonctionnement général de l'Internet, qui sinon se serait effondré depuis longtemps. L'inventeur de cette innovation de grande valeur est Van Jacobson.

Nous invitons le lecteur à quelques secondes de réflexion admirative devant un dispositif technique conçu à l'origine pour une centaine de nœuds et qui a pu résister à une croissance de six ou sept ordres de grandeur, d'autant plus que cette conception n'était le fait ni d'un grand groupe industriel ou financier, ni d'un gouvernement, ni d'un conglomérat de telles puissances. Mais peut-être était-ce là le secret ?

6.6.2 UDP *(User Datagram Protocol)*

Nous ne dirons que peu de mots d'UDP, qui est un protocole beaucoup plus simple que TCP. Il fournit au-dessus d'IP un protocole de transport non fiable, sans connexion et sans état. UDP se contente de construire un datagramme doté, comme les segments TCP, d'un numéro de port d'origine et d'un numéro de port de destination, et de l'encapsuler dans un datagramme IP. UDP convient bien à l'envoi de messages brefs et isolés ; cela dit, il est généralement considéré comme un protocole dangereux, à n'utiliser qu'en toute connaissance de cause. Notamment, dans un mode sans connexion, il n'est pas possible de vérifier l'appartenance d'un paquet à une connexion légitime.

De façon générale, les protocoles sans état sont des trous de sécurité. C'est spécialement le cas des protocoles de niveau application destinés au partage de fichiers en réseau, que ce soit NFS pour Unix, Netbios pour Windows-xx, Appleshare pour MacOS. Il est relativement facile d'introduire des paquets parasites dans un échange de messages établi avec ces protocoles, et ils sont particulièrement dangereux parce qu'ils exécutent des programmes à distance par le protocole RPC *(Remote Procedure Call)*, créent, modifient et détruisent des fichiers, bref ils donnent accès à tous les outils dont peut rêver un pirate sur une machine distante qu'il se propose d'attaquer.

6.7 Les téléphonistes contre-attaquent : ATM

La controverse entre les réseaux à commutation de circuits virtuels comme X25 et les réseaux à commutation de paquets « pure » comme TCP/IP a animé la décennie 1980. Les tenants de la première technique, ainsi qu'il a été signalé ci-dessus, étaient les opérateurs téléphoniques traditionnels, qui y voyaient un moyen de préserver leur méthode de facturation du transport de données au

volume, cependant que les constructeurs de réseaux informatiques en ressentaient les lourdeurs qui se répercutaient en rigidités insupportables. L'Internet et TCP/IP ont fini par l'emporter grâce à la facilité de déploiement que leur conféraient la commutation de paquets associée aux protocoles de routage dynamique.

Les années 1990 ont vu une contre-attaque de grande envergure des téléphonistes sous les espèces d'ATM *(Asychronous Transfer Mode)*. ATM ressuscite la commutation de circuits selon un protocole conçu par un centre de recherche de France Télécom, normalisé par l'UIT (Union Internationale des Télécommunications) et implémenté avec des moyens considérables.

Comme X25, l'architecture ATM comporte des aspects qui relèvent de couches différentes du modèle OSI :

- couche 2 : format des données transmises au support physique, dont les différentes variétés sont spécifiées dans le protocole ;
- couche 3 : établissement d'un circuit virtuel, qui fixe le ou les itinéraires possibles de bout en bout ;
- couche 4 : contrôle de flux de bout en bout.

Cette confusion des couches nuit à l'adoption d'un protocole. Une des clés du succès de TCP/IP, c'est que les protocoles de couche 3 et 4 (IP, TCP, UDP...) sont totalement indépendants du support physique et de la couche de liaison de données sous-jacents. Avec X25 ou ATM, vous ne choisissez pas seulement un protocole, mais aussi une technologie de réseau, et en fin de compte un fournisseur de services réseau, c'était d'ailleurs le but poursuivi, il aurait peut-être été atteint en situation de monopole fort des téléphonistes, mais aujourd'hui ATM est globalement un échec même si certains opérateurs télécom l'utilisent encore dans leurs réseaux internes.

ATM fournit un service non fiable et connecté de transmission de datagrammes sur un circuit virtuel. Le terme datagramme signifie que le flux de bits transmis par le protocole est découpé en paquets acheminés indépendamment les uns des autres. Par non fiable nous entendons que le protocole ne fournit aucune garantie de remise des datagrammes ni aucun contrôle d'erreur. Le caractère connecté d'ATM est en fait discutable : certes il y a établissement de circuit virtuel, mais le protocole ne maintient aucune information d'état sur une transmission de données en cours.

Les datagrammes d'ATM s'appellent des cellules et ont une taille fixe de 48 octets de données (oui, c'est minuscule) auxquels s'ajoutent 5 octets d'information de contrôle, dont un numéro de chemin virtuel sur 12 bits et un numéro de circuit virtuel sur 16 bits, soit 53 octets au total.

La conjonction du mode non fiable et du circuit virtuel peut sembler paradoxale : X25 était fiable et connecté. En fait le circuit virtuel a deux rôles : éviter de placer dans chaque cellule une information complète d'origine et de destination

(il ne resterait plus guère de place pour les données utiles), et maintenir des paramètres de qualité de service, essentiellement débit et isochronie. Préférer le débit à la fiabilité des données, c'est une attitude de téléphoniste ou de télévidéaste : une cellule corrompue dans une conversation téléphonique ou une image, ce n'est pas grave. Mais une cellule corrompue dans une transmission de données, c'est tout le message à retransmettre, et comme les cellules ne contiennent aucune information qui permette de les identifier, il faut retransmettre toute la transaction, éventuellement un grand nombre de cellules. Le numéro de circuit virtuel est fixé lors d'une procédure d'appel qui recourt à des cellules au format particulier, dites de signalisation, comme avec X25.

Beaucoup des idées élaborées pour permettre à ATM de gérer des communications avec des qualités de services diverses, appropriées à des applications telles que le transport de la voix et de l'image, ou le contrôle en temps réel de dispositifs matériels (machines, robots, dispositifs de sécurité), ont trouvé leur chemin dans les recherches actuelles sur le développement de TCP/IP.

6.8 Promiscuité sur un réseau local

Lorsqu'il est question de sécurité du réseau, on pense le plus souvent à la protection contre les attaques en provenance de l'Internet. Or, négliger les attaques en provenance de l'intérieur par le réseau local (*Local Area Network*, LAN) serait s'exposer à des menaces qui deviennent de jour en jour plus réelles avec le développement du nomadisme et des réseaux sans fil, et qui d'ailleurs existaient de tout temps. De ce point de vue nous pouvons dire que les réseaux sans fil ne produisent aucune menace qui n'ait déjà existé, ils ne font que susciter la prise de conscience des risques qui en résultent, et bien sûr en accroître l'intensité.

Nous allons le voir, il règne sur un réseau local une véritable *promiscuité*, au sens où un utilisateur mal intentionné dispose de moyens d'accès aux communications qui ne lui sont pas destinées.

6.8.1 Rappel sur les réseaux locaux

On nomme habituellement réseau local une infrastructure de couche 2 (liaison de données) qui dessert un bâtiment ou un campus. Une telle infrastructure comporte en général, outre le câblage, des répéteurs, des commutateurs et des bornes d'accès pour réseaux sans fil, mais pas de routeur, hormis celui qui relie le réseau local à l'Internet. C'est le schéma classique de l'équipement d'un site d'entreprise.

Nous n'évoquerons ici que les réseaux locaux définis par la norme IEEE 802.3, plus communément nommés *Ethernet*, puisque les autres types de réseaux définis par ce groupe de normes ne sont plus guère utilisés. Disons tout de suite

que les réseaux sans fil 802.11 (dits *Wi-Fi*) reposent par bien des points sur les mêmes principes techniques que 802.3, notamment pour leurs caractéristiques significatives du point de vue de la sécurité.

La norme 802.3 décrit des réseaux où toutes les stations partagent un support physique unique ; à l'origine il s'agissait d'un câble coaxial sur lequel toutes les stations étaient branchées en émission comme en écoute (câblage dit 10Base5), puis apparurent des répéteurs *(hubs)* auxquels les stations étaient reliées par des paires téléphoniques torsadées (câblage dit 10BaseT), mais toutes les données circulant sur le réseau atteignaient toutes les stations. Aujourd'hui la plupart des réseaux utilisent un câblage 100BaseT ou 1000BaseT en étoile autour de commutateurs *(switches)*, qui sont des répéteurs « intelligents » capables d'« apprendre » sur quelle branche de l'étoile se trouve telle station, ce qui leur permet d'établir des liaisons point à point et améliore ainsi considérablement la sécurité des communications. On peut dire que les réseaux 802.11 font revivre la première époque d'Ethernet 802.3, où toutes les stations accédaient au même support physique et pouvaient, de ce fait, recevoir *toutes* les données échangées sur ce support.

Une autre conséquence du partage du support physique par toutes les stations, c'est que deux stations peuvent essayer d'émettre simultanément, avec pour résultat ce que l'on appelle une *collision*, qui provoquera le brouillage temporaire des communications. Pour résoudre ce problème, les stations d'un réseau 802.3 mettent en œuvre le protocole dit *Carrier Sense Multiple Access with Collision Detection* (CSMA-CD), ou accès multiple par écoute de la porteuse, avec détection de collision. De leur côté, les réseaux 802.11 ont recours au protocole *Carrier Sense Multiple Access with Collision Avoidance* (CSMA-CA), analogue à CSMA-CD, mais avec évitement des collisions, parce que sur un réseau sans fil les collisions ne peuvent pas toujours être détectées, du fait que chaque station ne « voit » pas forcément toutes les autres.

Vocabulaire : La porteuse

Les systèmes de transmission électro-magnétiques, avec ou sans fil, procèdent souvent par l'émission d'une onde sur une fréquence constante, le signal étant réalisé par une modification de cette fréquence, ou sa *modulation*. L'onde de fréquence constante est appelée la *porteuse* (*carrier* en anglais).

6.8.2 Réseaux locaux virtuels (VLAN)

Les réseaux locaux virtuels (*Virtual LAN*, VLAN) sont apparus en 1995, avec les commutateurs 802.3. Il s'agit donc d'un dispositif de couche 2 (liaison de don-

nées), en pratique Ethernet. L'idée est la suivante : il peut être tentant, notamment pour des raisons de sécurité, de regrouper les stations d'un groupe de personnes qui travaillent dans la même équipe sur un réseau local qui leur sera réservé, séparé des réseaux des autres équipes. Mais si les membres des différentes équipes sont dispersés dans différents bâtiments et mélangés avec les autres groupes, adapter le câblage physique à l'organisation peut se révéler coûteux et malcommode, d'autant plus que la répartition géographique des membres de chaque équipe peut changer. On a donc recherché des moyens de créer, sur une infrastructure parfois complexe, des réseaux locaux virtuels, qui isoleraient logiquement les communications propres à un groupe de stations, lequel partagerait tout ou partie d'un même support physique avec d'autres groupes. En somme il s'agit de faire au niveau de la couche 2 (liaison de données) ce que les VPN (voir page 223) font au niveau de la couche 3 (réseau).

Après quelques errements, les VLAN ont été normalisés en 1998 par la norme 802.1Q, qui a nécessité une modification du format de la trame Ethernet afin de lui ajouter 4 octets, dont 12 bits constituent une étiquette *(tag)* destinée à identifier les trames qui appartiennent à tel ou tel réseau local virtuel. Les commutateurs modernes sont programmés pour tenir compte de ces étiquettes, et pour n'acheminer les trames que vers des destinations qui appartiennent au VLAN désigné par leur étiquette.

Ce sont les commutateurs qui jouent le rôle principal dans la gestion des VLAN : le premier commutateur que rencontre une trame lui affecte une étiquette, qui déterminera son VLAN, et, partant, son destin.

Un lien physique partagé par plusieurs VLAN est nommé *trunk* dans le jargon des VLAN, ou parfois *channel* dans la terminologie du constructeur *Cisco*.

Il est de bonne politique que le routeur de sortie du réseau vers l'Internet appartienne à tous les VLAN, ou du moins à tous ceux dont les stations doivent pouvoir atteindre l'Internet. Exclure ce routeur d'un VLAN est un bon moyen d'interdire aux utilisateurs de ce VLAN de naviguer sur l'Internet.

Les VLAN peuvent être utiles en termes de sécurité, par exemple en limitant la promiscuité sur un réseau local. Une application assez répandue et commode de ce procédé consiste, sur un campus ou au sein d'une entreprise, à créer pour accueillir les ordinateurs portables des visiteurs extérieurs un VLAN où ils seront confinés, ce qui évitera qu'ils puissent accéder aux serveurs internes, ou qu'ils répandent dans l'entreprise les virus dont ils pourraient être infectés, tout en ayant la possibilité d'accéder à l'Internet ou à toute autre ressource qui leur aura été autorisée.

6.8.3 Sécurité du réseau de campus : VLAN ou VPN ?

Nous venons de voir que les VLAN permettaient d'améliorer la sécurité d'un réseau local en cloisonnant le trafic réseau par la réservation à chaque équipe ou

entité fonctionnelle d'un réseau privé virtuel, et en limitant ainsi la promiscuité des données.

Une autre façon de segmenter le réseau est de recourir à des routeurs. Nous avons vu ci-dessus (page 223) que les réseaux privés virtuels (VPN) permettaient d'établir des tunnels chiffrés entre deux stations quelconques sur l'Internet.

Comment choisir entre ces deux types de solution ?

Les VLAN, par définition, ne peuvent être déployés qu'au sein d'un même réseau local. Dans ce rôle ils sont très commodes : une fois les commutateurs configurés, tout est automatique. Les commutateurs sont plus faciles à configurer que les routeurs, ils sont aussi moins chers et plus rapides. Les réseaux commutés demandent moins de compétences humaines et moins d'investissements matériels que les réseaux routés. Leur inconvénient principal, malgré la promulgation de la norme 802.1Q, est de reposer le plus souvent sur des recettes de configuration propres à chaque constructeur, qui violent plus ou moins ouvertement le principe de l'indépendance protocolaire : les VLAN mélangent des fonctions qui relèvent de la couche 2 avec des fonctions de couche 3. Cette confusion n'a pas que des inconvénients théoriques, elle peut conduire à l'édification d'un réseau à la topologie confuse dont l'évolution ultérieure et la maintenance seront difficiles.

Le routage est une technique qui repose sur des bases théoriques et conceptuelles solides et acceptées par tous. En fait, il est la pierre angulaire de l'Internet. Les protocoles privés créés naguère par certains constructeurs cèdent de plus en plus souvent la place aux protocoles normalisés et documentés, tel OSPF *(Open Shortest Path First)*[20]. Un réseau privé virtuel peut s'étendre, virtuellement donc, à l'ensemble de la planète, mais il est aussi tout à fait possible de construire pour un coût marginal un minuscule VPN entre mon ordinateur au bureau, celui de mon domicile et mon ordinateur portable connecté à un point d'accès sans fil.

En pratique

Nous pensons qu'il est très intéressant, sur un campus, de créer un VLAN pour accueillir les ordinateurs portables des visiteurs auxquels on ne veut pas accorder de droits, mais qui doivent quand même travailler et accéder à l'Internet, ne serait-ce que pour communiquer avec leurs bases. Pour tout autre usage, avant de créer un VLAN il faut se demander si le routage ne serait pas une solution plus satisfaisante. On pourra aussi regarder le protocole VXLAN, mentionné à la section 10.3.6 p. 298, cf. aussi la note 6 p. 226.

20. *Open Shortest Path First* (OSPF) est un protocole de routage basé sur un algorithme de recherche de parcours dans un graphe dû à Dijkstra.

6.9 Client–serveur ou pair à pair *(peer to peer)* ?

Tous les usages du réseau que nous avons évoqués jusqu'ici reposent plus ou moins explicitement sur le modèle client-serveur : un navigateur WWW (le client) demande une page à un serveur WWW, un logiciel de messagerie (le client) relève sa boîte à lettres sur un serveur par le protocole POP *(Post Office Protocol)*, etc.

Ce modèle a de nombreux usages fort utiles, mais il ne saurait prétendre à l'universalité et il donne une vision restreinte des possibilités du réseau. En concentrant une grande partie du trafic sur des nœuds spécialisés, les serveurs, il crée une absence de fluidité et un risque de congestion. On peut imaginer un autre modèle où chaque nœud serait connecté, plus ou moins virtuellement, à tous les autres. Si un tel modèle était inimaginable il y a vingt ans pour des raisons techniques, la croissance continue des débits des réseaux, des tailles mémoire et des puissances de calcul disponibles permet d'imaginer que chaque ordinateur personnel au domicile de chacun devienne à la fois serveur et client. Je peux ainsi héberger mon propre site WWW sur ma machine, avec mes photos de vacances et mes articles. Mon fournisseur d'accès à l'Internet ne me donne pas d'adresse IP fixe ? Qu'à cela ne tienne, des bienfaiteurs de l'humanité comme `dyndns.org` fournissent (et gratuitement de surcroît) un service de noms de domaines dynamiques qui ajuste périodiquement l'adresse IP qui correspond à mon nom de domaine à moi, ce qui permet à mes « clients » d'atteindre mon site à tout moment.

Ce qui est décrit ci-dessus est encore trop artisanal : il faut publier des services afin que chacun puisse y accéder sans avoir à connaître leur localisation *a priori*. C'est le pas franchi par des protocoles comme Napster ou Gnutella, auxquels chacun peut s'abonner, et devenir serveur même à son insu (ce qui n'a pas forcément que des avantages, par exemple lorsque le serveur en question abrite des données que la loi interdit de divulguer parce qu'elles sont protégées par le droit d'auteur ou tout simplement interdites). L'information sur les services disponibles circule de proche en proche.

Une autre direction importante où les protocoles *peer to peer* joueront un rôle est celui du « calcul en grille » *(grid computing)* : un calcul important, tel que ceux effectués en dynamique des fluides ou en analyse de génomes, est partagé entre de nombreux ordinateurs en réseau. Si le problème et ses données sont faciles à subdiviser, les communications entre les nœuds de calcul ne seront pas intenses, mais si ce n'est pas le cas le réseau sera soumis à une forte charge et la coordination par un serveur central constituerait un goulot d'étranglement insupportable. L'utilisation d'algorithmes distribués et de communications bilatérales indépendantes entre les nœuds s'imposera. Ces questions sont encore du domaine de la recherche, même si des réalisations opérationnelles existent déjà.

Signalons que ce type de service a eu un précurseur très précoce, le protocole NNTP de diffusion des News du réseau (des forums, en fait), qui est doté de

primitives dont les noms disent tout : `ihave`, `post`, `takethis`, `check`. Les articles des News se propagent de site en site, sans arbre hiérarchique prédéfini.

6.10 Versatilité des protocoles pair à pair

6.10.1 Définition et usage du pair à pair

Un grand coup de hache sur le modèle client-serveur est venu des protocoles *peer to peer* (souvent abrégés en P2P), ce que Wikipedia propose de traduire en français pas *pair à pair* et décrit ainsi :

« P2P désigne un modèle de réseau informatique dont les éléments (les nœuds) ne jouent pas exclusivement les rôles de client ou de serveur mais fonctionnent des deux façons, en étant à la fois clients et serveurs des autres nœuds de ces réseaux, contrairement aux systèmes de type client-serveur, au sens habituel du terme.

« ...

« Les réseaux P2P permettent de communiquer et de partager facilement de l'information - des fichiers le plus souvent, mais également des calculs, du contenu multimédia en continu *(streaming)*, etc. sur Internet. Les technologies P2P se sont d'ailleurs montrées si efficaces que le P2P est considéré par certains comme l'étape ultime "de la liberté et de la démocratie" sur Internet. Sans aller jusque là, on considère souvent que le P2P porte (et est porté par) une philosophie de partage et un profond esprit communautaire. »

Pour une présentation des évolutions récentes on pourra consulter la communication de Franck Cappello aux journées JRES 2005 [27].

Ces protocoles pair à pair sont utilisés massivement par les internautes équipés d'une connexion à heut débit pour échanger des fichiers aux contenus musicaux ou cinématographiques, au titre de ce que le droit français nomme la *copie privée*, et le droit américain *fair use*.

Les industries du disque et du cinéma n'étaient pas préparées à cette extension de la copie privée, à laquelle elles ont réagi principalement par le recours à la loi. Les premiers protocoles P2P, tel Napster, comportaient un serveur central qui recueillait et distribuait les adresses des participants, ce qui a permis aux industriels d'engager contre le propriétaire de ce serveur des actions en justice et d'obtenir sa fermeture.

Instruit par cette expérience, les protocoles pair à pair contemporains, tels KaZaA, Skype ou eMule, ne comportent pas de serveur central, ce qui oblige les entreprises qui souhaiteraient poursuivre leurs utilisateurs à les identifier un par un.

6.10.2 Problèmes à résoudre par le pair à pair

Les nœuds des systèmes pair à pair, quasiment par définition, sont des ordinateurs situés à la périphérie de l'Internet, et qui sont le plus souvent soit des machines personnelles dans un domicile privé, soit des postes de travail individuels au sein d'une entreprise qui n'a pas vraiment prévu qu'ils soient utilisés pour du pair à pair, voire qui essaye de l'interdire. Les conséquences techniques de cette situation sont les suivantes :

- les ordinateurs concernés sont souvent éteints ;
- ils n'ont souvent pas d'adresse IP permanente ;
- voire pas d'adresse routable (adresses dites « NAT *(Network Address Translation)* »).

Il faudra malgré ce contexte d'amateurisme que tous les nœuds puissent être à la fois clients et serveurs, qu'ils puissent communiquer directement deux à deux, et que chacun en fonction de ses capacités contribue au fonctionnement général de l'infrastructure. Il faut qu'un nœud qui rejoint le réseau puisse *découvrir* ceux qui offrent les ressources qui l'intéressent, selon le schéma de la figure 6.20.

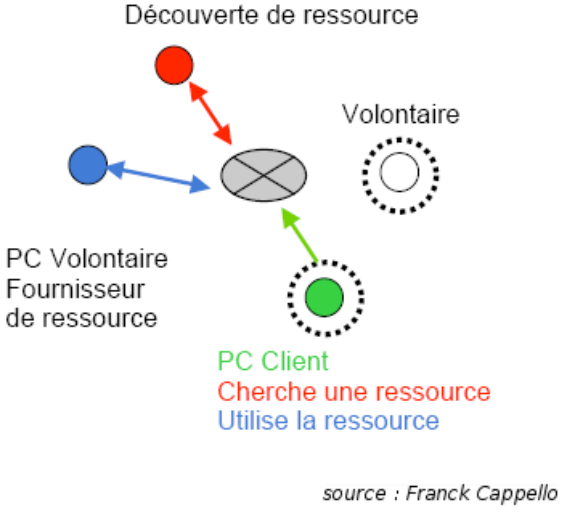


Figure 6.20 : *Un poste client tente de rejoindre une communauté de pairs.*

Pour surmonter les difficultés énumérées plus haut et atteindre ces objectifs, un système pair à pair comporte quatre composants fondamentaux:

1. une passerelle, qui publie l'adresse IP d'autres nœuds et permet à l'utilisateur de choisir une communauté au sein de laquelle il va échanger des données, comme représenté par la figure 6.21;

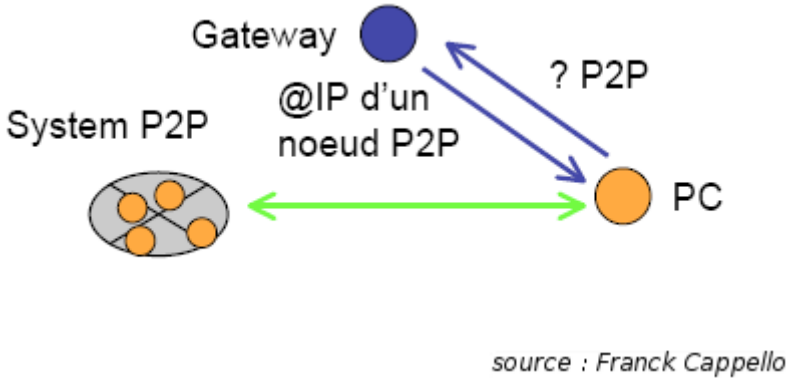
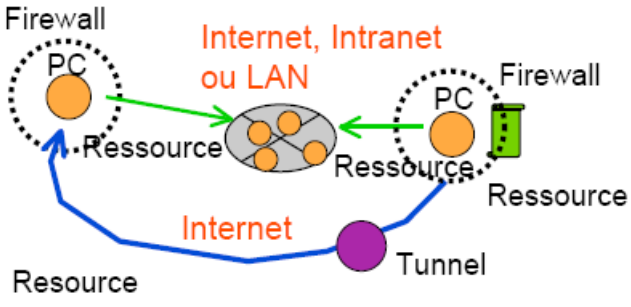


Figure 6.21 : *Une passerelle* (gateway) *va permettre au nouvel arrivant de découvrir l'adresse IP d'un membre déjà connecté.*

2. un protocole réseau pour l'établissement des connexions et l'exécution des opérations de transport de données; un élément crucial de ce protocole sera bien sûr son aptitude au franchissement de coup-feu, comme indiqué par la figure 6.22; en effet la communication pair à pair serait impossible dans le respect des règles de filtrage qu'imposent la plupart des réseaux, notamment en entreprise;
3. un système de publication de services et d'annonces de ressources disponibles, qui permet à chacun de contribuer à l'œuvre commune;
4. un système, symétrique du précédent, de recherche de ressources, qui permet de trouver ce que l'on cherche, tel morceau de musique ou tel film, ou le chemin d'accès à tel téléphone réseau.



source : Franck Cappello

Figure 6.22 : *Ici deux nœuds confinés par des coupe-feux* (firewalls) *essaient néanmoins de construire une voie de communication entre eux, mais le procédé retenu est rudimentaire et peu efficace.*

Chapitre 7 Protection et sécurité

Sommaire

7.1 Protection

Un système d'exploitation digne de ce nom doit comporter des dispositifs et des procédures de protection des objets qu'il permet de manipuler. Les objets à protéger appartiennent à deux grandes catégories : les objets persistants tels que les fichiers, et les objets éphémères créés en mémoire pendant l'exécution d'un processus et destinés à disparaître avec lui. Les objets matériels, tels que périphériques physiques, interfaces réseau, etc., sont assimilés à des objets persistants. La protection consiste à empêcher qu'un utilisateur puisse altérer un fichier qui ne lui appartient pas et dont le propriétaire ne lui en a pas donné l'autorisation, ou encore à empêcher qu'un processus en cours d'exécution ne modifie une zone mémoire attribuée à un autre processus sans l'autorisation de celui-ci, par exemple.

De façon très générale la question de la protection d'un objet informatique se pose dans les termes suivants, inspirés des concepts mis en œuvre par le système Multics :

- Un objet a un propriétaire identifié, généralement l'utilisateur qui l'a créé. Un objet est, sous réserve d'inventaire, soit un fichier, soit un processus, soit une structure de données éphémère créée en mémoire par un processus, mais nous avons vu à la section 5.4 que pour Multics tous ces objets sont en fin de compte des segments ou sont contenus dans des segments de mémoire virtuelle.
- Le propriétaire d'un objet peut avoir conféré à lui-même et à d'autres utilisateurs des droits d'accès à cet objet. Les types de droits possibles sont en général les suivants (on peut en imaginer d'autres) :
  - droit d'accès en consultation (lecture) ;
  - droit d'accès en modification (écriture, destruction, création) ;
  - droit d'accès en exécution ; pour un programme exécutable la signification de ce droit est évidente ; pour un répertoire de fichiers ce droit confère à ceux qui le possèdent la faculté d'exécuter une commande ou un programme qui consulte ce répertoire ;
  - droit de blocage, par exemple pour un processus en cours d'exécution ou éligible pour l'exécution.
- À chaque objet est donc associée une liste de contrôle d'accès *(access control list)* qui énumère les utilisateurs autorisés et leurs droits.
- Avant toute tentative d'accès à un objet par un utilisateur, l'identité de cet utilisateur doit être authentifiée.
- Pour qu'un utilisateur ait le droit d'exécuter une action sur un objet, et dans un système informatique cette action est perpétrée par l'entremise d'un processus, il faut en outre que le processus en question possède le *pouvoir*

voulu. Le pouvoir est un attribut d'un processus, il peut prendre des valeurs qui confèrent à ce processus des *privilèges* plus ou moins étendus. Jusqu'à présent nous n'avons rencontré que deux valeurs possibles de pouvoir : le mode superviseur et le mode utilisateur, mais nous allons voir que certains systèmes ont raffiné la hiérarchie des valeurs de pouvoir.

– La valeur du pouvoir d'un processus peut changer au cours de son exécution. Ainsi un processus qui se déroule dans un mode utilisateur peut faire une demande d'entrée-sortie, ce qui nécessite le mode superviseur. Ceci sera résolu, sous Unix par exemple, par le mécanisme de l'appel système, qui transfère le contrôle, pour le compte du processus utilisateur, à une procédure du noyau qui va travailler en mode superviseur.
– Nous définirons la notion de *domaine de protection* dans lequel s'exécute un processus comme l'ensemble des objets auxquels ce processus a accès et des opérations qu'il a le droit d'effectuer sur ces objets. Lorsqu'un processus change de valeur de pouvoir, il change par là même de domaine de protection.

Les dispositifs et procédures de protection du système d'exploitation vont consister à faire respecter les règles qui découlent des droits et pouvoirs énumérés ci-dessus et à empêcher leur violation. La protection au sens où nous allons l'étudier dans ce chapitre ne consiste pas à empêcher les erreurs humaines, les défaillances techniques ou les actes de malveillance qui pourraient faire subir à un objet un sort non désiré, mais seulement à empêcher leur incidence sur les objets en question. Il faut protéger les données et les processus d'un utilisateur contre les processus des autres utilisateurs, protéger le fonctionnement du système contre les processus des utilisateurs et vice-versa, enfin protéger l'un de l'autre les processus d'un même utilisateur.

La qualité des dispositifs et procédures de protection fait la *sûreté* d'un système d'exploitation. On conçoit notamment aisément que le contrôle des droits et des pouvoirs doive être à l'abri des manipulations d'utilisateurs désireux sans légitimité d'accroître leurs privilèges, ce qui signifie que les procédures de contrôle doivent s'exécuter avec le mode de pouvoir le plus grand et les droits les plus étendus, inaccessibles aux simples utilisateurs. Cette réflexion de simple bon sens suffit à refuser le qualificatif « sûr » à tel système d'exploitation qui comporte un système perfectionné de listes d'accès réalisé... en mode utilisateur, et pour lequel de surcroît l'identification des utilisateurs est facultative.

En effet, il va sans dire, mais disons-le : il ne sert à rien de contrôler les droits et les pouvoirs du propriétaire d'un processus si déjà son identité n'est pas raisonnablement certaine. Les procédures d'identification et d'authentification des utilisateurs sont un préalable à toute stratégie de protection.

7.1.1 Un parangon de protection : Multics

Dans le domaine de la protection, l'approche mise en œuvre par le système Multics dès les années 1960 fait encore aujourd'hui figure de référence exemplaire. Nous allons la décrire.

De même que les auteurs de Multics avaient accompli une percée conceptuelle considérable et qui reste aujourd'hui à poursuivre en réunissant les objets de mémoire et les fichiers dans un concept unique de segment, ils ont aussi imaginé pour la protection une approche et des concepts originaux et puissants que les systèmes d'aujourd'hui redécouvrent lentement.

Les dispositifs de protection de Multics

Nous décrirons les dispositifs et procédures mis en œuvre dans Multics pour assurer la protection des objets parce que, bien qu'anciens, ils restent à ce jour de l'an 2018 une réalisation de référence. Cette description doit beaucoup à celles de l'ouvrage collectif de Crocus [38], *Systèmes d'exploitation des ordinateurs* et du livre de Silberschatz et ses collègues [123] *Principes appliqués des systèmes d'exploitation*.

La protection sous Multics repose sur une structure dite « en anneaux ». Chaque processus s'exécute dans un anneau, chaque anneau correspond à un niveau de privilèges. Multics offre huit anneaux numérotés de 0 à 7, l'anneau 0 procure les privilèges les plus élevés, l'anneau 7 les moins élevés. L'anneau du processus courant figure dans le mot d'état de programme (PSW, cf. section 2.5 p. 29).

Chaque segment (de mémoire volatile ou persistante), pour chaque type d'accès (lecture, écriture, exécution si le segment contient un programme ou un répertoire), appartient à un anneau. Si un processus s'exécute dans un anneau de valeur inférieure ou égale à l'anneau d'exécution d'un segment, par exemple, il peut exécuter le programme contenu dans ce segment, sinon non.

À tout moment un processus peut changer d'anneau, sous le contrôle du système d'exploitation évidemment, et ainsi acquérir de façon temporaire ou définitive des privilèges supérieurs qui lui ouvriront l'accès à de nouveaux segments.

Finalement il apparaît que les plus fidèles disciples de l'équipe Multics furent les ingénieurs d'Intel. Depuis le modèle 80286 jusqu'à l'actuel Itanium les processeurs de la ligne principale d'Intel disposent d'une gestion de mémoire virtuelle à adressage segmenté. Aucun système d'exploitation implanté sur ces processeurs, que ce soient ceux de Microsoft ou les Unix libres FreeBSD, NetBSD ou Linux, ne tire parti de ce dispositif pour unifier les gestions de la mémoire virtuelle et de la mémoire persistante (le système de fichiers) ; les premiers sont contraints à la compatibilité avec leurs ancêtres... et les Unix aussi.

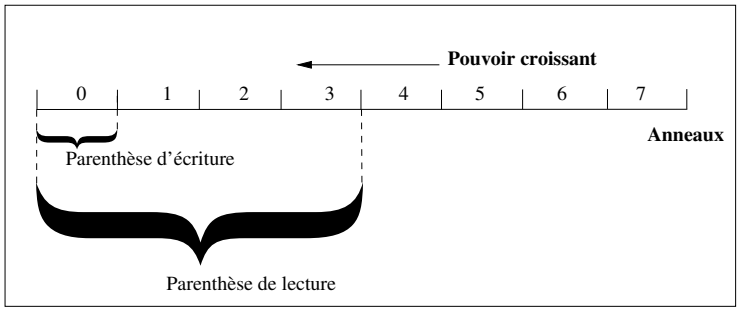


Figure 7.1 : *Protection en anneaux sous* Multics

Les processeurs Intel disposent d'un système de protection à quatre anneaux, typiquement destinés respectivement au noyau du système pour l'anneau 0, aux fonctions auxiliaires du système pour les anneaux 1 et 2, et aux programmes en mode « utilisateur » pour l'anneau 3. Le VAX disposait aussi d'un tel système à quatre anneaux. Sur les Intel ces possibilités ne sont guère utilisées par les systèmes d'exploitation. Les systèmes conventionnels comme Unix possèdent un système d'anneaux dégradé à seulement deux anneaux (le mode superviseur et le mode utilisateur) et un système de listes d'accès dégradé avec pour chaque fichier des droits d'accès en lecture, en écriture et en exécution pour trois ensembles d'utilisateurs : le propriétaire du fichier, les membres de son groupe, tous les autres utilisateurs. Linux utilise l'anneau 0 comme mode noyau et l'anneau 3 comme mode utilisateur et c'est tout. Ces systèmes plus rudimentaires ont (avaient ?) l'avantage d'être moins lourds.

7.2 Sécurité

7.2.1 Menaces, risques, vulnérabilités

La sécurité des systèmes informatiques (et de façon plus large celle des systèmes d'information) est un vaste problème dont les aspects techniques ne sont qu'une partie. Les aspects juridiques, sociaux, ergonomiques, psychologiques et organisationnels sont aussi importants, mais nous ne les aborderons pas ici. Nous laisserons également de côté les aspects immobiliers de la sécurité, qui ne doivent bien sûr pas être oubliés mais sont loin de notre propos.

Les problèmes techniques actuels de sécurité informatique découlent directement ou indirectement de l'essor des réseaux, qui multiplie la quantité et la gravité des menaces potentielles. Ces menaces entrent dans une des catégories suivantes : atteinte à la disponibilité des systèmes et des données, destruction de données, corruption ou falsification de données, vol ou espionnage de données,

usage illicite d'un système ou d'un réseau, usage d'un système compromis pour attaquer d'autres cibles.

Les menaces engendrent des risques : perte de confidentialité de données sensibles, indisponibilité des infrastructures et des données, dommages pour le patrimoine scientifique et la notoriété, coûts humains et financiers. Les risques peuvent se réaliser si les systèmes menacés présentent des vulnérabilités.

7.2.2 Principes de sécurité

La résorption des vulnérabilités repose sur un certain nombre de principes et de méthodes que nous allons énumérer dans la présente section avant de les décrire plus en détail.

Inutile de se préoccuper de sécurité sans avoir défini ce qui était à protéger : en d'autres termes une organisation quelconque désireuse de protéger ses systèmes et ses réseaux doit déterminer son périmètre de sécurité, qui délimite l'intérieur et l'extérieur. Une fois fixé ce périmètre, il faut aussi élaborer une politique de sécurité, en d'autres termes décider ce qui est autorisé et ce qui est interdit. À cette politique viennent bien sûr s'ajouter les lois et les règlements en vigueur, qui s'imposent à tous. Et il faut bien sûr prendre en considération la généralisation des ordinateurs portables, smartphones et autres tablettes, qui rendent le périmètre de sécurité très poreux.

Ceci fait, il sera néanmoins possible de mettre en place les solutions techniques appropriées à la défense du périmètre selon la politique choisie. Mais déjà il est patent que les dispositifs techniques ne pourront pas résoudre tous les problèmes de sécurité.

Les systèmes et les réseaux comportent des données et des programmes que nous considérerons comme des *ressources*. Certaines ressources sont d'accès public, comme par exemple un serveur WWW, d'autres sont privées pour une personne, comme une boîte à lettres électronique, d'autres sont privées pour un groupe de personnes, comme l'annuaire téléphonique interne d'une entreprise. Ce caractère plus ou moins public d'une ressource doit être traduit dans le système sous forme de droits d'accès, comme nous l'avons vu au début de ce chapitre.

Les personnes qui accèdent à une ressource non publique doivent être *identifiées* ; leur identité doit être *authentifiée* ; leurs droits d'accès doivent être *vérifiés* : à ces trois actions correspond un premier domaine des techniques de sécurité, les méthodes d'**authentification**, de signature, de vérification de l'**intégrité** des données objet et d'attribution de droits.

La sécurité des accès par le réseau à une ressource protégée n'est pas suffisamment garantie par la seule identification de leurs auteurs. Sur un réseau local de type Ethernet où la couche de liaison de données fonctionne en diffusion il est possible à un tiers de capter la transmission de données. Si la transmission a lieu

à travers l'Internet, les données circulent de façon analogue à une carte postale, c'est-à-dire qu'au moins le facteur et la concierge y ont accès. Dès lors que les données doivent être protégée, il faut faire appel aux techniques d'un second domaine de la sécurité informatique : le **chiffrement**.

Authentification et chiffrement sont indissociables : chiffrer sans authentifier ne protège pas des usurpations d'identité (l'attaque dite de type *man in the middle*), authentifier sans chiffrer laisse la porte ouverte au vol de données. Mais ces deux méthodes de sécurité ne suffisent pas, il faut en outre se prémunir contre les intrusions destinées à détruire ou corrompre les données, ou à en rendre l'accès impossible. Les techniques classiques contre ce risque sont l'usage de **coupe-feux** *(firewalls)* et le **filtrage** des communications réseaux, qui permettent de protéger la partie privée d'un réseau dont les stations pourront communiquer avec l'Internet sans en être visibles. Entre le réseau privé et l'Internet les machines publiques seront placées dans une zone démilitarisée (**DMZ**), où elles hébergeront par exemple le serveur WWW et le relais de messagerie de l'entreprise. Ces machines exposées au feu de l'Internet seront appelées bastions.

Certains auteurs considèrent que ces techniques de sécurité par remparts, ponts-levis et échauguettes sont dignes du Moyen-âge et leur préfèrent les systèmes de détection d'intrusion (IDS), plus subtils. Cela dit, dans un paysage informatique où les micro-ordinateurs prolifèrent sans qu'il soit réaliste de prétendre vérifier la configuration de chacun, le filtrage et le coupe-feu sont encore irremplaçables.

Nous allons examiner un peu plus en détail chacune de ces collections de techniques, en commençant par la cryptographie parce que les techniques de l'authentification en sont dérivées.

7.3 Chiffrement

Nous ne saurions tracer ici une histoire complète des codes secrets, pour laquelle le lecteur pourra se reporter au livre de Simon Singh [124] par exemple. Tout ce qui est antérieur à 1970 a un intérêt essentiellement historique, bien que passionnant et riche d'enseignements, comme par exemple le rôle récemment mis en lumière d'Alan Turing dans le déroulement de la Seconde Guerre mondiale, évoqué dans la biographie d'Andrew Hodges [61].

7.3.1 Chiffrement symétrique à clé secrète

De l'époque de Jules César à la fin des années 1970, un grand nombre de systèmes de chiffrement ont été inventés, qui consistaient à faire subir à un texte clair une transformation plus ou moins complexe pour en déduire un texte inintelligible, dit chiffré. La transformation repose sur deux éléments, une fonction mathématique (au sens large) et une clé secrète. Seule une personne

connaissant la fonction et possédant la clé peut effectuer la transformation inverse, qui transforme le texte chiffré en texte clair. C'est la même clé qui sert au chiffrement et au déchiffrement, et pour cette raison elle doit rester secrète : nous décrirons plus loin des systèmes de *chiffrement asymétrique,* qui utilisent des clés différentes pour le chiffrement et le déchiffrement, ce qui permet de rendre publique la clé de chiffrement, puisque'elle ne permet pas le déchiffrement.

La science de l'invention des codes secrets s'appelle la cryptographie. La science, adverse, du déchiffrement de ces codes est la cryptanalyse. Si le cryptanalyste ignore tout de la fonction de chiffrement et de la clé il aura le plus grand mal à déchiffrer, mais un bon code doit résister à la découverte de sa fonction de chiffrement tant que la clé reste secrète.

Une bonne fonction de chiffrement doit éviter de prêter le flanc à la cryptanalyse. Ainsi le code de César, qui reposait sur une simple transposition circulaire des lettres de l'alphabet, est très facile à décoder par l'analyse des fréquences des lettres dès lors que l'on sait dans quelle langue a été écrit le message. Un bon code doit aussi chiffrer de façons différentes deux occurrences successives d'un même texte dans le corps du message pour éviter que la détection d'une répétition ne fournisse des indices au cryptanalyste. La connaissance simultanée d'un texte clair et de sa version chiffrée, comme dans le cas de Champollion et de la pierre de Rosette, est bien sûr une aubaine pour le décodeur, comme l'occurrence de noms propres etc.

7.3.2 Naissance de la cryptographie informatique : Alan Turing

L'invention de l'ordinateur a bien sûr donné un essor considérable à la cryptographie et à la cryptanalyse. Ce n'est d'ailleurs pas un hasard si le créateur du modèle théorique de l'ordinateur, Turing, a été aussi pendant la guerre un formidable concepteur de machines à déchiffrer les codes allemands chiffrés par les automates *Enigma*. Les machines de Turing, appelées « Bombes », étaient fondées sur une réalisation originale du logicien polonais Marian Rejewski. La courbe qui trace le succès des attaques de sous-marins allemands contre les convois transatlantiques qui acheminaient les fournitures américaines à la Grande-Bretagne subit des fluctuations importantes qui correspondent au délai à l'issue duquel l'équipe d'Alan Turing à Bletchley Park parvenait à déchiffrer plus ou moins parfaitement le code allemand après un changement de combinaison des *Enigma*. Lorsque l'on sait l'importance militaire qu'ont eue ces fournitures, on ne saurait sous-estimer la contribution de Turing à la victoire alliée.

7.3.3 *Data Encryption Standard (DES)*

Le premier système de chiffrement informatique normalisé fut créé par un Allemand émigré aux États-Unis en 1934, Horst Feistel. Sa nationalité et son mé-

tier de cryptographe lui valurent quelques difficultés avec la National Security Agency (NSA), désireuse avant tout de garder la maîtrise des moyens de chiffrement et de pouvoir percer les codes utilisés par des personnes privées. Finalement il mit ses compétences au service d'IBM, pour qui il développa au début des années 1970 le cryptosystème *Lucifer*, base du futur *Data Encryption Standard (DES)*.

Le DES repose sur les principes suivants : le texte clair est codé en numération binaire et découpé en blocs de 64 bits. Chaque bloc est découpé en demi-blocs dont les bits subissent des permutations complexes, puis les demi-blocs sont additionnés et soumis à d'autres transformations. L'opération est recommencée seize fois. La fonction de transformation comporte des variations en fonction de la clé, qui est un nombre arbitraire choisi par les utilisateurs du code. Le nombre de valeurs possibles pour la clé détermine le nombre de façons différentes dont un message peut être chiffré. L'émetteur du message secret le chiffre selon l'algorithme DES au moyen de la clé, le destinataire applique la fonction inverse avec la même clé pour le déchiffrer.

La NSA a obtenu que la normalisation du DES en 1976 comporte une limitation de la taille de la clé à 56 bits, ce qui correspond à 10^{17} valeurs possibles. Aujourd'hui cette valeur est notoirement trop faible, et l'on utilise le triple DES, avec une longueur de clé de 112 bits.

La nouvelle norme AES utilise des clés de 128, 192 et 256 bits. La mise en concurrence pour AES a été lancée le 2 janvier 1997 et le choix de la solution a eu lieu le 3 octobre 2000. C'est l'algorithme *Rijndael* développé par Joan Daemen et Vincent Rijmen de l'Université catholique de Leuven qui a été retenu.

La postérité actuelle du DES procure un chiffrement qui peut être considéré comme robuste, à condition que soit résolu le problème crucial de tous les systèmes qui reposent sur une clé secrète utilisée aussi bien pour le chiffrement que pour le déchiffrement : les participants doivent s'échanger des clés de façon secrète, ce qui n'est pas simple.

7.3.4 Diffie, Hellman et l'échange de clés

Si Alex veut entretenir une correspondance secrète avec Bérénice, ils peuvent convenir de chiffrer leurs messages avec un protocole tel que le triple DES, que nous venons de présenter. Ce protocole présente toutes les garanties de robustesse, mais il faudra que Bérénice et Alex conviennent d'une clé secrète : pour ce faire, ils devront se rencontrer, ce qui peut être impossible, ou se communiquer la clé par la poste : dans les deux cas, l'instant de l'échange est celui dont un espion peut profiter pour dérober leur secret et ainsi réduire à néant la sûreté de leurs communications. C'est le problème de l'échange de clés.

Le problème de l'échange de clés

Depuis des siècles le problème de l'échange des clés était considéré comme un inconvénient naturel du chiffrement. Les ambassades et les états-majors y consacraient des efforts importants, que les espions s'efforçaient de déjouer.

Avec l'utilisation de l'ordinateur et des télétransmissions, et la dématérialisation de l'information qu'ils autorisent, le problème se pose différemment. Dans les années 1970 un chercheur indépendant et excentrique, Whitfield Diffie, réfléchissait au moyen pour deux utilisateurs du réseau ARPANET d'échanger des courriers électroniques chiffrés sans se rencontrer physiquement. En 1974 il donna une conférence sur le sujet au centre de recherche Thomas J. Watson d'IBM à Yorktown Heights (déjà le lieu de travail de Horst Feistel), et là il apprit que Martin Hellman, professeur à l'Université Stanford à Palo Alto, avait déjà donné une conférence sur le même sujet. Aussitôt il prit sa voiture et traversa le continent pour rencontrer Hellman.

Diffie et Hellman cherchaient une méthode pour convenir d'un secret partagé sans le faire circuler entre les participants ; en d'autres termes une fonction mathématique telle que les participants puissent échanger des informations dont eux seuls puissent déduire le secret. Les caractéristiques souhaitées d'une telle fonction sont la relative facilité de calcul dans le sens direct, et la quasi-impossibilité de calculer la fonction réciproque. Ainsi, si s est le secret en clair, F la fonction de chiffrement, c le secret chiffré, D la fonction de déchiffrement, il faut que $c = F(s)$ soit facile à calculer, mais $s = D(c)$ pratiquement impossible à calculer pour tout autre que les participants, au prix de quel stratagème, c'est ce que nous allons voir.

Fondements mathématiques de l'algorithme Diffie-Hellman

La solution repose sur un chapitre de l'arithmétique très utilisé par les informaticiens, l'*arithmétique modulaire*, ou l'arithmétique basée sur les classes d'équivalence *modulo* n.

Considérons l'ensemble des entiers relatifs $\mathbb{Z}$ muni de l'addition et de la multiplication. La division entière de a par b que nous avons apprise à l'école primaire y est définie ainsi :

$$a \div b \rightarrow a = b \times q + r$$

où q est le quotient et r le reste de la division. Ainsi :

$$13 \div 3 \rightarrow 13 = 3 \times 4 + 1$$

Intéressons-nous maintenant à tous les nombres qui, divisés par un nombre donné n, par exemple 3, donnent le même reste r. Nous avons déjà trouvé un nombre, 13, pour lequel $r = 1$, donnons-en quelques autres :

$$\begin{aligned} 1 \div 3 &\rightarrow 3 \times 0 + 1 \\ 4 \div 3 &\rightarrow 3 \times 1 + 1 \\ 7 \div 3 &\rightarrow 3 \times 2 + 1 \\ 10 \div 3 &\rightarrow 3 \times 3 + 1 \\ 13 \div 3 &\rightarrow 3 \times 4 + 1 \\ 16 \div 3 &\rightarrow 3 \times 5 + 1 \end{aligned}$$

On dit que ces nombres constituent une classe d'équivalence, et qu'ils sont tous équivalents à $1 \mod 3$ (prononcer « un modulo trois »):

$$\begin{aligned} 4 &\equiv 1 \mod 3 \\ 7 &\equiv 1 \mod 3 \\ &\dots \end{aligned}$$

On construit de la même façon une classe des nombres équivalents à $0 \mod 3$, qui contient $-6, -3,0,3,6,9,12, \dots$, et une classe des nombres équivalents à $2 \mod 3$, avec $-7, -4, -1,2,5,8,11, \dots$.

On peut définir une addition modulaire, par exemple ici l'addition $\mod 3$:

$$\begin{aligned} 4 + 7 \pmod 3 &= (4 + 7) \mod 3 \\ &= 11 \mod 3 \\ &= 2 \mod 3 \end{aligned}$$

On démontre (exercice laissé au lecteur) que l'ensemble des classes d'équivalence *modulo* n muni de cette relation d'équivalence (réflexive, transitive) et de cette addition qui possède les bonnes propriétés (associative, commutative, existence d'un élément neutre $0 \mod n$ et d'un symétrique pour chaque élément) possède une structure de groupe appelé le groupe additif $\mathbb{Z}_n$ (prononcé « Z modulo n »).

On peut aussi faire des multiplications:

$$\begin{aligned} 4 \times 7 \pmod 3 &= (4 \times 7) \mod 3 \\ &= 28 \mod 3 \\ &= 1 \mod 3 \end{aligned}$$

Nous pouvons montrer là aussi que la multiplication modulo 3 possède toutes les bonnes propriétés qui font de notre ensemble de classes d'équivalence un groupe pour la multiplication, mais cela n'est vrai que parce que 3 est premier.

En effet si nous essayons avec les classes d'équivalence modulo 12, nous aurons des diviseurs de zéro, ce qui détruit la structure de groupe :

$$\begin{aligned} 4 \times 7 \pmod{12} &= (4 \times 7) \bmod 12 \\ &= 28 \bmod 12 \\ &= 4 \bmod 12 \end{aligned}$$

$$\begin{aligned} 4 \times 6 \pmod{12} &= (4 \times 6) \bmod 12 \\ &= 24 \bmod 12 \\ &= 0 \bmod 12 \end{aligned}$$

Dans la seconde expression, le produit de 4 et 6 est nul, ce qui est très regrettable. Aussi pourrons-nous bien définir un groupe multiplicatif $\mathbb{Z}_n^*$, qui si n est premier aura les mêmes éléments que le groupe additif $\mathbb{Z}_n$ à l'exclusion de 0, mais si n n'est pas premier il faudra en retrancher les classes correspondant aux diviseurs de n et à leurs multiples :

$$\begin{aligned} \mathbb{Z}_3^* &= \{1,2\} \\ \mathbb{Z}_{12}^* &= \{1,5,7,11\} \\ \mathbb{Z}_{15}^* &= \{1,2,4,7,8,11,13,14\} \end{aligned}$$

Dans ce groupe multiplicatif chaque élément a un inverse (sinon ce ne serait pas un groupe) :

$$\begin{aligned} 5 \times 5 \bmod 12 &= 25 \bmod 12 \\ &= 1 \bmod 12 \\ 7 \times 7 \bmod 12 &= 49 \bmod 12 \\ &= 1 \bmod 12 \\ 11 \times 11 \bmod 12 &= 121 \bmod 12 \\ &= 1 \bmod 12 \end{aligned}$$

$$\begin{aligned} 7 \times 13 \bmod 15 &= 91 \bmod 15 \\ &= 1 \bmod 15 \end{aligned}$$

On note que les calculs sont faciles mais les résultats un peu imprévisibles : justement, c'est le but que poursuivent nos deux cryptographes. La fonction $y = ax$ n'est pas monotone. L'exponentielle est définie :

$$\begin{aligned} 5^3 \bmod 11 &= 125 \bmod 11 \\ &= 4 \end{aligned}$$

et si n est premier elle a les mêmes propriétés que dans $\mathbb{Z}$:

$$(a^x)^y = (a^y)^x = a^{x.y}$$

Mise en œuvre de l'algorithme Diffie-Hellman

Voici maintenant le protocole d'échange de clés de Diffie-Hellman, illustré par un exemple avec de petits nombres pour pouvoir faire les calculs à la main. Martin Hellman en a eu l'inspiration une nuit, mais il est le résultat de leur travail commun, auquel d'ailleurs il faut adjoindre Ralph Merkle[1]. Le protocole repose sur une fonction de la forme $K = W^X \bmod P$, avec P premier et $W < P$. Une telle fonction est très facile à calculer, mais la connaissance de K ne permet pas d'en déduire facilement X. Cette fonction est publique, ainsi que les valeurs de W et P. Prenons $W = 7$ et $P = 11$[2].

1. Anne choisit un nombre qui restera son secret, disons $A = 3$.
2. Bernard choisit un nombre qui restera son secret, disons $B = 6$.
3. Anne et Bernard veulent échanger la clé secrète, qui est en fait $S = W^{B.A} \bmod P$, mais ils ne la connaissent pas encore, puisque chacun ne connaît que A ou que B, mais pas les deux.
4. Anne applique à A la fonction à sens unique, soit α le résultat:

$$\begin{aligned} \alpha &= W^A \bmod P \\ &= 7^3 \bmod 11 \\ &= 343 \bmod 11 \\ &= 2 \end{aligned}$$

1. Martin Hellman et Whitfield Diffie ont reçu le Prix Turing 2015 pour cette invention. Lors de leurs discours de réception ils ont abondamment rendu hommage à Ralph Merkle, oublié par le jury, mais c'était trop tard.

2. Le lecteur attentif remarquera que beaucoup d'auteurs utilisent cet exemple numérique. S'il se donne la peine de quelques essais personnels il constatera qu'il y a une bonne raison à cela: les autres valeurs numériques suffisamment petites donnent des résultats corrects mais peu pédagogiques du fait de coïncidences fâcheuses.

5. Bernard applique à B la fonction à sens unique, soit β le résultat :

$$\begin{aligned} \beta &= W^B \bmod P \\ &= 7^6 \bmod 11 \\ &= 117\,649 \bmod 11 \\ &= 4 \end{aligned}$$

6. Anne envoie α à Bernard, et Bernard lui envoie β, comme représenté par la figure 7.2. α et β ne sont pas la clé, ils peuvent être connus de la terre entière sans que le secret d'Anne et de Bernard soit divulgué.

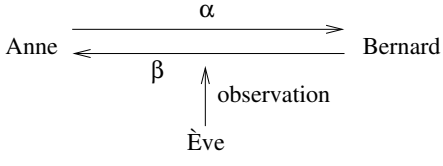


Figure 7.2 : *Échange de clés selon Diffie et Hellman*

7. Anne a reçu β et calcule $\beta^A \bmod P$ (qui est, soit dit en passant, $(W^B)^A \bmod P$, soit $7^{B.A} \bmod 11$, mais elle ne connaît pas B) :

$$\begin{aligned} \beta^A \bmod P &= 4^3 \bmod 11 \\ &= 64 \bmod 11 \\ &= 9 \end{aligned}$$

8. Bernard a reçu α et calcule $\alpha^B \bmod P$ (qui est, soit dit en passant, $(W^A)^B \bmod P$, soit $7^{A.B} \bmod 11$, mais il ne connaît pas A) :

$$\begin{aligned} \alpha^B \bmod P &= 2^6 \bmod 11 \\ &= 64 \bmod 11 \\ &= 9 \end{aligned}$$

Anne et Bernard obtiennent à la fin de leurs calculs respectifs le même nombre 9 qui n'a jamais été exposé à la vue des indiscrets : c'est la clé S ! N'est-ce pas miraculeux ? Ils ont juste échangé l'information nécessaire pour calculer la clé, sans divulguer celle-ci.

Supposons qu'Ève veuille épier les conversations d'Anne avec Bernard : elle pourra intercepter l'échange des messages non chiffrés α et β, à partir desquels elle veut calculer $S = \alpha^B \bmod P$. Elle ignore S et B. L'équation à résoudre pour calculer B consiste à calculer la fonction réciproque de la fonction à sens unique :

$$W^B = \beta \bmod P$$

Si nous étions dans le monde des nombres réels la solution serait triviale :

$$B = \frac{\log \beta}{\log W}$$

Mais dans le monde des classes d'équivalence modulo n ce problème dit du *logarithme discret* n'a pas de solution simple. C'est un sujet de recherche. Le « front » est aujourd'hui à des valeurs de P qui sont des nombres de 450 chiffres binaires. L'algorithme est sûr si P a 512 chiffres binaires.

L'algorithme de Diffie-Hellman est sans doute une découverte majeure, totalement contraire à l'intuition. Il procure à deux acteurs d'un cryptosystème le moyen d'échanger une clé sans la faire circuler sur le réseau. Mais il restait à faire une découverte encore plus stupéfiante, inspirée d'ailleurs par celle que nous venons de décrire : un cryptosystème fondé sur des clés publiées dans des annuaires publics !

7.3.5 Le chiffrement asymétrique à clé publique

La méthode de Diffie et Hellman permet l'échange de clés, mais elle impose une concertation préalable entre les acteurs. Parfois ce n'est pas pratique : si Anne veut envoyer à Bernard un courrier électronique chiffré pendant qu'il est en vacances, elle sera obligée d'attendre son retour pour établir la clé avec lui.

Whitfield Diffie avait eu une autre idée, pour laquelle il n'avait pas trouvé de solution mathématique appropriée : un système où l'on utiliserait une clé pour chiffrer et une autre pour déchiffrer. Ainsi, Bernard proposerait à Anne une clé de chiffrement, avec laquelle elle coderait le message, et Bernard le décoderait avec une clé différente, la clé de déchiffrement. La clé de chiffrement ne permet que de chiffrer, même Anne serait incapable de déchiffrer son propre message avec cette clé, seul Bernard le peut avec sa clé de déchiffrement. Comme la clé de chiffrement ne fonctionne que dans un sens, elle permet de créer des secrets mais pas d'en dévoiler, et elle peut donc être publique, inscrite dans un annuaire ou sur un site Web. Quiconque veut envoyer un message chiffré à Bernard peut la prendre et l'utiliser.

Il faut juste pouvoir être sûr que personne ne pourra calculer la clé de déchiffrement à partir de la clé de chiffrement. Et là il faut une intuition mathématique décisive.

Si l'idée du chiffrement asymétrique à clés publiques revient à Diffie et Hellman (sans oublier les précurseurs britanniques tenus au secret), la réalisation de cette idée revient à Rivest, Shamir et Adleman, qui ont trouvé une solution mathématique permettant la mise en œuvre de l'idée et donné son nom à cette solution : RSA, leurs initiales.

Une personne désireuse de communiquer selon cette méthode doit procéder ainsi :

1. Prendre deux nombres premiers p et q. En cryptographie réelle on choisira de très grands nombres, de 150 chiffres décimaux chacun. Nous allons donner un exemple avec $p = 3$ et $q = 11$.
2. Calculer $n = pq$, soit dans notre exemple $n = 33$.
3. Calculer $z = (p-1)(q-1)$. (Ce nombre est la valeur de la fonction $\phi(n)$, dite fonction phi d'Euler, et incidemment elle donne la taille du groupe multiplicatif modulo n, $\mathbb{Z}_n^*$). Dans notre exemple $z = 20$.
4. Prendre un petit entier e, impair et premier avec z, soit $e = 7$. Dans la pratique e sera toujours petit devant n.
5. Calculer l'inverse de $e \pmod z$, c'est-à-dire d tel que $e.d = 1 \bmod z$. Les théorèmes de l'arithmétique modulaire nous assurent que dans notre cas d existe et est unique. Dans notre exemple $d = 3$.
6. La paire $P = (e,n)$ est la clé publique.
7. La paire $S = (d,n)$ est la clé privée.

Voyons ce que donne notre exemple. La clé publique de Bernard est donc $(7, 33)$. Anne veut lui envoyer un message M, disons le nombre 19. Elle se procure la clé publique de Bernard sur son site WWW et elle procède au chiffrement de son message M pour obtenir le chiffré C comme ceci :

$$C = P(M) = M^e \bmod n$$
$$C = P(19) = 19^7 \bmod 33 = 13$$

Pour obtenir le texte clair T Bernard décode avec sa clé secrète ainsi :

$$T = S(C) = C^d \bmod n$$
$$T = S(13) = 13^3 \bmod 33 = 19$$

Miraculeux, non ? En fait c'est très logique :

$$\begin{aligned} S(C) &= C^d \bmod n \\ &= (M^e)^d \bmod n \\ &= M^{e.d} \bmod n \\ &= M \bmod n \end{aligned}$$

Le dernier résultat, $M^{e.d} = M \pmod n$ découle du fait que e et d sont inverses modulo n, il se démontre grâce au petit théorème de Fermat.

Le petit théorème de Fermat

Si p est un nombre premier, pour tout entier a non divisible par p on a: $a^{p-1} - 1 \equiv 0 (\mathrm{mod} p)$

***Démonstration**:*

Considérons les $p-1$ premiers multiples de a, a, $2a$, ..., $(p-1) \times a$. Par exemple, pour $p = 23$ et $a = 5$, nous aurons la liste L:

5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,100,105,110

Calculons les restes de la division de ces nombres par $p = 23$:

```
(define (reste-23 n) (remainder n 23))

(map reste-23 L)
(5 10 15 20 2 7 12 17 22 4 9 14 19 1 6 11 16 21 3 8 13 18)
```

La liste de ces restes est, dans le désordre, celle des valeurs 1,2,3,...$p-1$. En effet aucun de ces restes ne peut être nul, parce ni a ni aucun de ses multiples de rang inférieur à p n'est divisible par p (il faudrait pour cela que soit a, soit un entier inférieur à p soit divisible par p, ce qui est impossible par hypothèse pour a et évidemment pour un entier inférieur à p).

Montrons en outre que ces restes sont tous différents. Soient n et m, différents ($n > m$) et inférieurs à p, supposons que:

$$(n \times a) \div p \rightarrow (n \times a) = q_1 \times p + r_1$$

$$(m \times a) \div p \rightarrow (m \times a) = q_2 \times p + r_2$$

Si maintenant nous supposons les restes égaux: $r_1 = r_2 = r$, nous pouvons écrire:

$$(n \times a) = q_1 \times p + r$$

$$(m \times a) = q_2 \times p + r$$

La soustraction membre à membre de ces égalités nous donne:

$$(n - m) \times a = (q_1 - q_2) \times p$$

soit un multiple de a par un nombre inférieur à p divisible par p, ce qui est contraire à l'hypothèse. Donc $n = m$. Ne pouvant être nuls, et étant forcément différents les uns des autres, les restes de la liste L sont donc bien toujours les valeurs 1,2,3,...$p-1$.

Si maintenant on fait le produit de ces multiples a,$2a$,$3a$,...$(p-1)\times a$, le résultat est $(p-1)!\times a^{p-1}$

Le reste modulo p de ce produit, compte tenu du résultat précédent sur les restes modulo p des multiples, est donc:

$$a\times 2a\times ...\times (p-1)a\equiv 1\times 2\times ...\times (p-1)(\mathrm{mod}p)$$

soit

$$a^{p-1}\times (p-1)!\equiv (p-1)!(\mathrm{mod}p)$$

ce que l'on peut réécrire sous la forme:

$$a^{p-1}\times (p-1)!-(p-1)!\equiv 0(\mathrm{mod}p)$$

$$(a^{p-1}-1)\times (p-1)!\equiv 0(\mathrm{mod}p)$$

ce qui signifie que l'expression à gauche du symbole d'équivalence est divisible par p. D'après le lemme de Gauss (si un nombre entier a divise le produit de deux autres nombres entiers b et c, et si a est premier avec b, alors a divise c), c'est soit l'un soit l'autre des deux facteurs de cette expression qui est divisible par p. Comme $(p-1)!$ n'est pas divisible par p, c'est $a^{p-1}-1$, donc:

$$a^{p-1}-1\equiv 0(\mathrm{mod}p)$$

□

Cette démonstration est inspirée de celle donnée par Guillaume Saupin dans le numéro 214 (avril 2018) de GNU/Linux Magazine [117].

À quel type d'attaque est exposé RSA? Un espion (Ève par exemple) pourra obtenir la clé publique de Bernard $P=(e,n)$, qui a servi à chiffrer M, ainsi que le message chiffré, C. Pour trouver M l'équation à résoudre est:

$$C=M^e \bmod n$$

n, C et e étant connus. Encore une fois dans le monde des réels la solution est triviale: $M=\sqrt[e]{C}$. Mais dans le monde modulaire la solution est $M=\sqrt[e]{C} \bmod n$, et il n'y a pas de méthode connue pour la calculer, même pour de petites valeurs de e. Ainsi, trouver la racine cubique modulo n d'un nombre y n'est pas un problème résolu aujourd'hui, et d'ailleurs beaucoup de cryptosystèmes industriels utilisent effectivement $e=3$.

En fait la seule attaque possible (outre la recherche de failles de réalisation du logiciel) consisterait à trouver p et q par recherche des facteurs de n, ce que

l'on appelle la factorisation du nombre n. La factorisation permettrait de calculer $z = \phi(n) = (p-1)(q-1)$. Le nombre secret d est tel que $e.d \equiv 1 \bmod z$. d est un nombre du même ordre de grandeur que z, soit un nombre de mille chiffres binaires. Ce calcul serait réalisable, mais le problème est que la factorisation n'est pas un problème résolu, et qu'il est donc impossible en général de calculer p, q et z.

Les réalisations industrielles ont longtemps utilisé, et utilisent parfois encore $e = 3$. De nos jours $e = 2^{16} + 1 = 65\,537$ est populaire. Avec un tel choix d est du même ordre de grandeur que n, soit $d \approx 2^{1024}$. L'élévation à une puissance de cet ordre peut être réalisée efficacement par des algorithmes de type « élévation au carré et multiplication » *(square and multiply)*, qui prennent moins d'une seconde dans une carte à puce[3].

Le lecteur trouvera des explications mathématiques supplémentaires dans l'ouvrage de Cormen, Leiserson et Rivest (le R de RSA) [35] ou dans celui de Menezes, van Oorschot et Vanstone [87], ou encore, de façon plus abordable, dans ceux de Gilles Dubertret[48] d'Albert Ducrocq et André Warusfel [49]. Au demeurant, il est stupéfiant de constater que les découvertes prodigieuses de Diffie, Hellman, Merkle, Rivest, Shamir et Adleman reposent sur des bases mathématiques déjà entièrement établies par Leonhard Euler (1707–1783), sinon par Pierre de Fermat (1601–1665), et que personne n'y avait pensé avant.

7.3.6 *Pretty Good Privacy (PGP)* et signature

Le système PGP défraya la chronique judiciaire en 1993 lorsque son auteur Philip Zimmerman fut soumis à une enquête sévère du FBI pour le motif d'avoir exporté illégalement des armes de guerre, en l'occurrence pour avoir placé son logiciel en accès libre sur l'Internet. Les autorités policières américaines (et françaises) ont tendance à penser que le chiffrement robuste est un obstacle à leurs investigations parce qu'il leur interdirait de déchiffrer les messages échangés par des criminels ou des ennemis. Aujourd'hui tous les dispositifs cryptographiques les plus puissants sont accessibles facilement par l'Internet et ainsi disponibles pour lesdits criminels, espions, etc. Une législation restrictive ne s'appliquerait par définition qu'aux honnêtes citoyens soucieux de respecter la loi parce que c'est la loi, pas parce que c'est difficile de faire autrement. Une telle législation n'aurait donc pour effet que de mettre les honnêtes gens à la merci des criminels, ce qui ne semble pas l'effet recherché, en principe du moins.

Sachant que de telles législations sont en déclin, même en France, pays qui a fermement tenu l'arrière-garde jusqu'en 1998, voyons le contenu de PGP. En

3. Je remercie le regretté François Bayen pour ses suggestions qui ont notablement amélioré les exposés cryptographiques de ce chapitre.

fait, PGP n'apporte aucune révolution, il est plutôt un assemblage ingénieux et pratique des techniques évoquées ci-dessus.

L'idée du chiffrement asymétrique avec un couple clé publique–clé privée semble tellement puissante qu'on ne voit pas de raison pour qu'elle ne supplante pas toutes les autres techniques. En fait un algorithme aussi puissant soit-il ne résout pas tous les problèmes. D'abord les algorithmes de chiffrement asymétriques tel RSA sont très lourds en temps de calcul, et échanger de nombreux messages chiffrés ainsi devient vite un fardeau.

L'attaque par le milieu *(Man in the middle)*

Ensuite, le meilleur chiffrement du monde ne peut pas empêcher qu'un agent malintentionné, disons Charles, se soit fait passer pour Bernard, ait intercepté les communications d'Anne, et lui ait présenté sa clé publique comme étant celle de Bernard : ainsi Charles pourra facilement déchiffrer les messages d'Anne avec sa propre clé privée, les lire, puis les re-chiffrer avec la vraie clé publique de Bernard et les faire parvenir à ce dernier. Ce type d'attaque, appelé *Man in the middle* (par le milieu), est difficile à déjouer une fois que Charles a réussi à s'introduire dans le circuit de communication ; elle peut être tentée contre RSA et aussi contre Diffie-Hellman.

En fait nous sommes ramenés au sempiternel problème dont nous nous croyions débarrassés : comment établir une relation de confiance entre Anne et Bernard, comment échanger des clés dignes de foi. Mais nous avons quand même accompli un progrès : cet échange de clés doit être certifié, mais il peut se faire au grand jour puisque les clés sont désormais publiques. Les clés publiques doivent être signées par une autorité supérieure, ce qui donne naissance à le notion d'infrastructure de gestion de clés, ou IGC (PKI en anglais), voir plus loin section 7.4.

Pour pallier la lenteur des calculs d'algorithmes à la RSA, Zimmerman eut l'idée de recourir au bon vieux chiffrement à clé partagée ; comme le point faible de ce dernier est l'envoi de la clé, on utilisera RSA pour communiquer une clé de session pour un algorithme à clés symétriques, clé qui servira à chiffrer la suite des communications avec cet algorithme classique. En l'occurrence Zimmerman choisira IDEA, un cousin de DES à clés de 128 bits, créé à Zurich par James L. Massey et Xuejia Lai, et réputé très robuste. Incidemment les systèmes de communication chiffrés tels que SSL *(Secure Socket Layer)* utilisés pour les transactions par le WWW, la relève de courrier électronique et la connexion conversationnelle à distance par SSH *(Secure Shell)* fonctionnent de cette façon.

Cette utilisation combinée des méthodes de chiffrement symétrique (DES en l'occurrence) et asymétrique sera la vraie révolution pratique, qui suscitera la colère de la NSA et de ses homologues dans d'autres pays dont la France. Avant que cette possibilité existe, les utilisateurs de cryptosystèmes se mettaient

laborieusement d'accord sur une clé, puis ils l'utilisaient pendant longtemps. La NSA disposait sans doute des moyens de casser le chiffrement DES, ce qui lui ouvrait des mois de lecture paisible de messages réputés secrets. Avec la combinaison de DES et RSA, les utilisateurs changent de clé à chaque échange de messages, ce qui complique beaucoup la tâche des « services ».

PGP sera la cible principale de l'ire des services gouvernementaux, non parce qu'il serait un cryptosystème révolutionnaire, mais parce qu'il constitue une trousse à outils facile d'emploi pour l'usage quotidien, avec les outils de chiffrement symétrique et asymétrique, la gestion de « trousseaux de clés » publiques et privées, l'incorporation automatique de ces outils au logiciel de courrier électronique de l'utilisateur, sans oublier les accessoires de signature électronique. Bref, on installe PGP (ou maintenant sa version libre GnuPG) sur son ordinateur personnel et ensuite tous les messages sont chiffrés et déchiffrés sans que l'on ait à s'en préoccuper. Les services semblaient mal supporter cette situation.

Signature

Outre sa fonction de chiffrement, RSA est aussi utilisable de façon très simple pour signer de façon sûre et non répudiable un document : il suffit que l'émetteur (le signataire en l'occurrence) chiffre le document à authentifier avec sa clé *privée* : le destinataire déchiffrera avec la clé publique de l'émetteur, et si le déchiffrement réussit ce sera une authentification sûre.

En fait, plutôt que de signer en chiffrant de cette façon l'ensemble du message, on en extrait un résumé numérique par un algorithme de condensation, tel MD5 créé par Ron Rivest, ou SHA (*Secure Hash Standard* FIPS 180-1), que l'on chiffre. Outre une signature non répudiable, ce procédé garantit en pratique l'intégrité du message. Le principe d'une fonction de condensation (ou de hachage) est le suivant : soient M et M' deux messages, et H la fonction :

1. si $M \neq M'$, la probabilité que $H(M) = H(M')$ est très voisine de 0 ;
2. quel que soit M, il est difficile de trouver un $M' \neq M$ tel que $H(M') = H(M)$.

7.3.7 Usages du chiffrement : VPN

À la section 7.3 p. 209, nous avons décrit l'usage de techniques cryptographiques pour le chiffrement de messages individuels, mais ce n'est en aucun cas le seul usage de cette technique. On peut imaginer, et c'est de plus en plus ce qui sera réalisé, le chiffrement systématique de toutes les communications en réseau.

Si l'on procède ainsi, chiffrer message par message serait très inefficace : on choisira plutôt de chiffrer le flux de l'ensemble du trafic sur un ou plusieurs

itinéraires donnés, cela constituera un *réseau privé virtuel*, (VPN, *Virtual Private Network*). Il s'agira par exemple d'établir un canal chiffré entre deux nœuds quelconques de l'Internet, ces nœuds pouvant eux-mêmes être des routeurs d'entrée de réseaux. On aura ainsi établi une sorte de tunnel qui, à travers l'Internet, reliera deux parties éloignées l'une de l'autre du réseau d'une même entreprise pour donner l'illusion de leur contiguïté. Mais le chiffrement permet aussi d'établir un VPN personnel pour un utilisateur, par exemple entre son ordinateur portable et le réseau local de l'entreprise.

Encapsulation, tunnel, inspection en profondeur

Les premières pages de ce chapitre nous ont familiarisés avec les notions de protocole, de paquet, de segment et d'en-tête. Munis de ce savoir, posons-nous le problème suivant : il faut acheminer les échanges relatifs à un protocole A au travers d'un réseau R qui n'autorise pas ce protocole. C'est possible en utilisant un protocole autorisé, appelons-le B ; à l'entrée du réseau R qui refuse le protocole A, je place les paquets complets de A, munis de leurs en-têtes, comme des données de paquets B, munis des en-têtes B ; à la sortie de ce réseau j'extraierai les paquets A, qui pourront continuer leur chemin. Pour que cela marche, il aura bien sûr fallu que je mette en place quelques conventions qui me permettent de retrouver mes paquets A à la sortie. Cette opération s'appelle *encapsulation* de A dans B. En encapsulant le protocole A dans le protocole B, nous avons réalisé un *tunnel* pour le protocole A au travers du réseau R inhospitalier.

Maintenant le travail des ingénieurs de sécurité du réseau R, dont la mission est d'empêcher le transit du protocole A à travers R, est plus difficile. Avant le tunnel, il leur suffisait de bloquer le numéro de port caractéristique du protocole A. Désormais, il leur faut configurer un logiciel de détection d'intrusion qui va examiner le contenu de tous les paquets, y compris ceux de protocoles autorisés comme B, pour voir si les données ne sont pas des paquets A encapsulés. Ce travail d'examen des données contenues dans les paquets se nomme *inspection en profondeur*.

L'encapsulation de protocole utilise fréquemment des protocoles universellement autorisés, comme HTTP (Web, port 80) ou le DNS (port 53).

On remarquera que le fonctionnement normal du réseau comporte déjà l'encapsulation de chaque protocole en partant du haut de la pile dans le protocole de la couche immédiatement inférieure, par exemple de TCP dans IP. Mais rien n'empêche d'encapsuler un flux IP dans TCP. Naguère, du

temps du réseau Transpac, on encapsulait IP dans X25, protocole étranger à l'Internet.

Principes du réseau privé virtuel

Le chiffrement est généralement utilisé pour les VPN de la façon suivante : l'algorithme de Diffie-Helmann est utilisé pour procéder au choix d'un secret partagé, qui constituera une clé de session pour chiffrer le trafic, et qui sera renouvelé à intervalles réguliers.

Il y a en revanche une assez grande variété de solutions pour introduire le VPN dans l'architecture du réseau :

- **Couche 3 :** introduire le VPN au niveau de la couche réseau (n° 3 du modèle ISO) semble la solution la plus logique : il s'agit bien de créer un *réseau virtuel*, après tout. C'est la solution retenue par la pile de protocoles désignés collectivement par l'acronyme IPsec, que nous décrirons à la section suivante. Les protocoles IPsec sont implantés dans le noyau du système d'exploitation, ce qui assure une plus grande sûreté de fonctionnement (face aux attaques notamment) et de meilleures performances (un protocole implanté en espace utilisateur passe son temps à recopier des tampons de mémoire entre l'espace noyau et l'espace utilisateur).
- **Couche 4 :** La disponibilité de bibliothèques SSL/TLS (pour *Secure Socket Layer/Transport Layer Security*) à la mise en œuvre facile a encouragé le développement de VPN de couche 4 (transport), comme *OpenVPN* ou les tunnels SSL[4]. *OpenVPN*, par exemple, établit un tunnel entre deux stations, et par ce tunnel de transport il établit un lien réseau, chaque extrémité recevant une adresse IP.
- **Couche 7 :** Le logiciel SSH *(Secure Shell)*, qui comme son nom l'indique est un client de connexion à distance chiffrée, donc de couche 7, permet de créer un tunnel réseau.
- **Couche 2 :** Mentionnons ici, pour mémoire, les réseaux locaux virtuels (VLAN) ; il ne s'agit pas à proprement parler de VPN, mais ils ont souvent un même usage : regrouper les stations d'un groupe de personnes qui travaillent dans la même équipe sur un réseau qui leur soit réservé, séparé des réseaux des autres équipes. Et on peut même encapsuler un VLAN

4. `http://openvpn.net/`

(couche 2) dans des paquets UDP de couche 4, c'est le protocole *Virtual Extensible LAN*[5] (VXLAN), indispensable avec l'informatique en nuage[6].
L2TP *(Layer Two Tunneling Protocol)*, comme son nom l'indique, encapsule une liaison de couche 2 (liaison de données) sur un lien réseau (couche 3).

IPsec

IPsec désigne un ensemble de RFC destinées à incorporer les techniques de chiffrement (et d'autres, relatives aussi à la sécurité) au protocole IP lui-même, plutôt que d'avoir recours à des solutions externes. IPv6 a été conçu pour *pouvoir* comporter d'emblée toutes les spécifications IPsec, qui sont aussi disponibles pour IPv4.

IPsec comporte essentiellement deux protocoles :

– le protocole AH *(Authentication Header)* assure l'authenticité et l'intégrité des données acheminées ; c'est un protocole réseau, de couche 3 donc, que l'on peut voir comme une option d'IP ;
– le protocole de transport ESP (couche 4) *(Encapsulating Security Payload)* assure la confidentialité et l'intégrité des données, leur authenticité étant assurée de façon optionnelle.

Avec l'un ou l'autre de ces protocoles, IPsec peut fonctionner en *mode transport* ou en *mode tunnel* :

– en mode tunnel chaque paquet IP est encapsulé dans un paquet IPsec lui-même précédé d'un nouvel en-tête IP ;
– en mode transport un en-tête IPsec est intercalé entre l'en-tête IP d'origine et les données du paquet IP.

La figure 7.3 illustre ces différentes possibilités.

Les protocoles AH et ESP sont complétés par le protocole d'échange de clés IKE *(Internet Key Exchange)*, défini dans la RFC 2409, et par le protocole de gestion de clés ISAKMP *(Internet Security Association and Key Management Protocol)*, défini dans la RFC 2408. Selon certains experts du réseau, ISAKMP serait un protocole irrémédiablement mal conçu.

5. Cf. `http://vincent.bernat.im/fr/blog/2012-multicast-vxlan.html`

6. En effet la possibilité, offerte par l'informatique en nuage *(cloud computing)*, que la machine virtuelle chargée d'effectuer votre travail se déplace subitement à l'autre bout de la planète crée un dilemme : soit elle se déplace dans la couche 3, et elle change alors de réseau, donc d'adresse IP, et si elle abrite votre serveur web vous perdez vos visiteurs, ou alors elle se déplace dans la couche 2, mais cela impose une infrastructure de niveau 2 transcontinentale, ce qui est impraticable à cause de la taille excessive de la table d'adresses MAC (de couche 2) et des problèmes de sécurité entraînés par la non-segmentation d'un réseau tellement vaste. VXLAN est une des solutions possibles du dilemme. NVGRE *(Network Virtualization using Generic Routing Encapsulation)* est une proposition alternative.

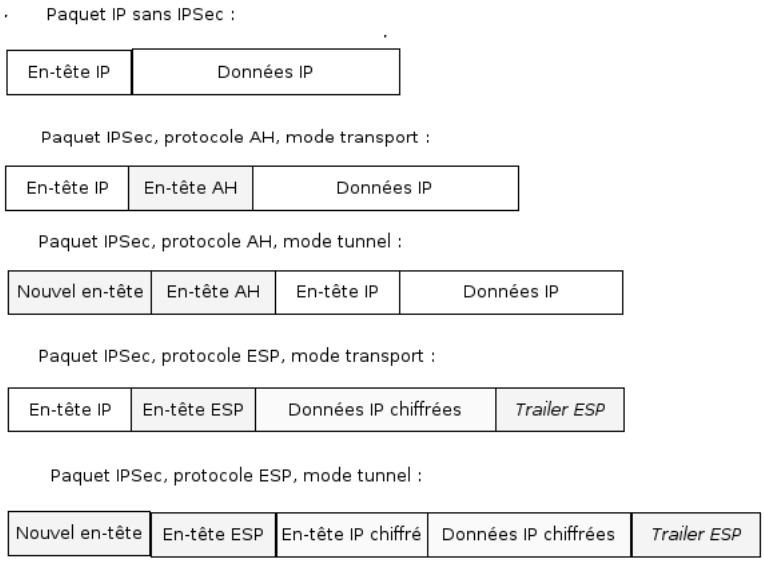


Figure 7.3 : *Protocoles et modes IPsec*

ISAKMP est censé faciliter un grand déploiement mais n'est pas nécessaire à IPsec.

Autres réseaux privés virtuels

À côté d'IPsec, il existe d'autres procédés pour créer des réseaux privés virtuels, intégrés à TCP/IP de façon peut-être moins satisfaisante, mais plus pratique.

1. L2TP (*Layer Two Tunneling Protocol*, RFC 2661), comme son nom l'indique, encapsule une liaison de couche 2 (liaison de données) sur un lien réseau (couche 3), ce qui permet à un PC distant d'avoir accès au réseau de son entreprise comme s'il était connecté au réseau local, et ainsi d'avoir accès aux serveurs de fichiers, aux imprimantes, etc.
2. MPLS (*Multi-Protocol Label Switching*, RFC 2547) est un protocole de niveau 3 (réseau) qui permet d'établir un tunnel privé au sein d'un réseau public ; il est surtout utilisé par les fournisseurs d'accès à l'Internet pour proposer à leurs clients un moyen de créer un réseau privé entre plusieurs sites d'une même entreprise.
3. Mentionnons également des procédés pour créer des tunnels dits « IP dans IP » (couche 3, réseau) par divers procédés, ou encore les réseaux virtuels créés au moyen de TLS *(Transport Layer Security)*, qui, comme son nom

l'indique, est une version de la couche 4 renforcée du point de vue de la sécurité.

Aujourd'hui, la plupart des VPN effectivement en fonction appartiennent à la dernière catégorie de la liste ci-dessus. Il existe des boîtiers qui contiennent des systèmes tout configurés, qu'il suffit de placer derrière les routeurs d'entrée de réseau pour disposer d'un VPN entre deux sites.

Il convient, avant de clore cette section, de signaler que si la technique des réseaux locaux virtuels *(Virtual Local Area Network, VLAN)* vise un objectif en principe assez différent de celui des VPN, elle peut dans certains cas être envisagée comme une solution de substitution.

7.4 Annuaire électronique et gestion de clés

À ce stade de l'exposé, nous disposons de deux types de cryptosystèmes, l'un symétrique à secret partagé, l'autre asymétrique avec clés publiques et clés privées, le second permettant l'échange du secret partagé nécessaire au premier. Nous avons aussi, sans coût supplémentaire, un système de signature sûre et non répudiable qui garantit en outre l'intégrité des messages reçus. Ce qu'un système technique ne peut fournir à lui seul, c'est l'établissement du circuit de la confiance : comment être sûr que telle clé publique ne m'a pas été fournie par un usurpateur ? PGP fournit à ce problème une solution à l'échelle d'une personne et de son cercle de relations : trousseau de clés publiques et privées conservé sur le disque dur d'un ordinateur personnel. Mais il est patent que PGP ne répond pas, du moins à lui tout seul, à ce problème à l'échelle d'une entreprise, ni *a fortiori* à celle de l'Internet. Pour ce faire il faut recourir à un système d'annuaire électronique complété par une infrastructure de gestion de clés (IGC, en anglais *Public Key Infrastructure, PKI*).

L'annuaire électronique est une base de données au format un peu particulier qui rend les services habituels d'un annuaire : répertorier des personnes ou des serveurs selon un schéma hiérarchique, de l'entité la plus englobante (pays) à la plus petite (personne) en passant par plusieurs niveaux (entreprise, département, service...). L'annuaire électronique contient aussi, idéalement, des certificats, qui comprennent notamment les clés publiques des entités enregistrées. Pour attester la véracité de ces certificats, ils sont, toujours idéalement, signés par une ou plusieurs autorités de certification, et éventuellement par le détenteur de la clé lui-même.

Il existe une norme assez généralement acceptée pour la structure hiérarchique de désignation des objets de l'annuaire, héritée de la norme d'annuaire X500 de l'ISO et adaptée de façon simplifiée par l'IETF pour les protocoles de l'Internet, sous le nom LDAP *(Lightweight Directory Access Protocol)*. La syntaxe ne fera peut-être pas l'unanimité, mais elle permet de traiter à peu près tous les

cas possibles. Voici le DN *(Distinguished Name)* de l'objet « Jacques Martin », c'est-à-dire son nom absolu, constitué de RDNs *(Relative Distinguished Names)* successifs, un peu comme les noms relatifs dans une arborescence de fichiers Unix constituent le chemin absolu d'un fichier ; CN signifie *Common Name*, OU *Organizational Unit*, O *Organization* :

```
cn=Jacques Martin, ou=Groupe Système,
ou=Division Informatique, o= Compagnie Dupont
```

La forme des certificats découle également de la normeX500, et elle obéit à la norme X509.

Qui certifie la signature des autorités de certification ? En bref, qui me garantit que le contenu de l'annuaire n'est pas un artefact créé par un escroc ? La procédure de création de l'IGC et d'enregistrement des entités comportera nécessairement une intervention humaine qui à chaque étape constate l'identité de la personne (physique, morale ou technique) à laquelle est délivré le certificat. Un certificat émis par l'IGC décrit une entité et contient sa clé publique, ainsi que les signatures des autorités qui garantissent le certificat.

Dans un monde idéal (idéal du point de vue de la sécurité informatique, qui n'est certes pas le seul possible), une hiérarchie d'IGC propage une certaine confiance. Chaque personne qui accède à un système d'information est identifiée par l'annuaire et authentifiée par son certificat et sa clé publique, dont le pendant est la clé privée. Chaque serveur est également identifié et authentifié. Les communications entre les deux peuvent être chiffrées. Ceci aurait pour avantage, outre de faire obstacle plus efficacement à la fraude informatique, de permettre aux personnes de posséder un système d'identification électronique unique *(single sign on)* au lieu d'avoir à connaître des dizaines de mots de passe et autres codes secrets pour leur carte bancaire, leur téléphone mobile, leurs courriers électronique privé et professionnel, la consultation en ligne de leurs comptes bancaires, les différents systèmes informatiques auxquels elles accèdent pour leur vie professionnelle et privée.

7.5 Sécurité d'un site en réseau

7.5.1 Découpage et filtrage

Assurer la sécurité de systèmes informatiques abrités sur un site connecté à l'Internet et de ce fait accessible du monde entier est une autre branche de ce domaine dont, en 2018 encore, beaucoup d'organisations ne semblent pas avoir pris l'exacte mesure.

Comme nous l'avons déjà mentionné, il appartient tout d'abord aux responsables du site de déterminer le périmètre qu'ils veulent protéger, ainsi que ce qu'ils veulent permettre et autoriser.

Cet examen aboutit généralement à identifier un certain nombre de services qui doivent être accessibles de l'extérieur par nature, comme un serveur WWW, un relais de courrier électronique, un serveur DNS. Les ordinateurs qui abritent ces services devront être visibles de l'Internet, c'est-à-dire que le DNS doit publier leurs adresses.

Les autres ordinateurs, que ce soient des serveurs internes ou les stations de travail des personnes qui travaillent sur le site, ne doivent pas être visibles, mais il faut néanmoins qu'ils puissent accéder à l'Internet. En d'autres termes, une session TCP initiée de l'intérieur du site depuis un de ces ordinateurs est autorisée, mais une session initiée de l'extérieur vers le même ordinateur est interdite parce que réputée erronée ou hostile (pour un expert en sécurité informatique les deux termes sont synonymes).

De quels moyens et méthodes disposons-nous en 2018 pour mettre en œuvre une telle politique de sécurité? Il nous faut pour cela nous rappeler le chapitre 6 consacré aux réseaux, et notamment ce que nous avons dit des informations contenues dans les en-têtes de datagrammes IP et de segments TCP, ainsi que du routage.

Chaque paquet IP qui se présente à un routeur est doté d'une fiche signalétique constituée de ses en-têtes. Les informations principales par rapport au sujet qui nous occupe sont les adresses IP d'origine et de destination et le protocole de transport (TCP ou UDP), figurant dans l'en-tête de datagramme IP, et les numéros de ports [7] d'origine et de destination, figurant dans l'en-tête de segment TCP ou de datagramme UDP. La mention dans l'en-tête IP du protocole de transport permet de connaître le format de l'en-tête de transport (TCP ou UDP), et ainsi d'y retouver le numéro de port. Nous avons vu que l'association d'une adresse et d'un port constituait une *socket* (voir section 6.6.1). Une paire de *sockets* identifie de façon unique une connexion dans le cas de TCP. Le routeur maintient une table des connexions TCP établies qui lui permet de déterminer si ce paquet appartient à une communication déjà en cours (parce que entre mêmes adresses IP et avec mêmes numéros de ports, par exemple) ou à une nouvelle communication.

Le routage est un grand instrument de sécurité. Il permet de découper un grand réseau en autant de sous-réseaux qu'on le souhaite, et de contrôler le trafic entre ces sous-réseaux. Les sous-réseaux peuvent d'ailleurs être virtuels, pour s'affranchir des contraintes de localisation, ce qui sera de plus en plus le cas avec le développement de l'informatique mobile. Ceci exige des compétences et du travail, parce que ce que nous avons dit du routage montre que c'est tout sauf

7. Rappelons qu'un port dans la terminologie TCP/IP est un numéro conventionnel qui, associé à une adresse IP, identifie une extrémité de connexion.

simple. Mais un tel investissement est indispensable à qui veut disposer d'un réseau sûr.

Forts de cette possibilité, nous pourrons découper le réseau de notre site en un sous-réseau public, qui abritera les serveurs visibles de l'extérieur, et un sous-réseau privé, éventuellement divisé lui-même en sous-réseaux consacrés à tel groupe ou à telle fonction. Chacun de ces sous-réseaux verra son accès et son trafic régis par des règles spéciales.

Les règles d'accès et de trafic appliquées aux réseaux consistent à établir quels sont les type de paquets (en termes de protocole et de numéro de port, en l'état actuel de la technique) autorisés en entrée ou en sortie depuis ou vers tel réseau ou telle adresse particulière. Ainsi un serveur de messagerie pourra recevoir et émettre du trafic SMTP (port 25) mais n'aura aucune raison de recevoir du trafic NNTP *(Network News Transfer Protocol)*. Appliquer ce genre de règles s'appelle du *filtrage par port*.

Le sous-réseau public (souvent appelé « zone démilitarisée » ou DMZ) devra faire l'objet de mesures de sécurité particulièrement strictes, parce que de par sa fonction il sera exposé à toutes les attaques en provenance de l'Internet. Le principe de base est : tout ce qui n'est pas autorisé est interdit, c'est-à-dire que tout paquet qui n'a pas de justification liée aux fonctions du serveur de destination doit être rejeté.

Il est prudent que les serveurs en zone publique contiennent aussi peu de données que possible, et même idéalement pas du tout, pour éviter qu'elles soient la cible d'attaques. Ceci semble contradictoire avec le rôle même d'un accès à l'Internet, mais cette contradiction peut être résolue en divisant les fonctions. Ainsi pour un serveur de messagerie il est possible d'installer un relais en zone publique qui effectuera toutes les transactions avec le monde extérieur mais transmettra les messages proprement dit à un serveur en zone privée, inaccessible de l'extérieur, ce qui évitera que les messages soient stockés en zone publique en attendant que les destinataires en prennent connaissance. De même un serveur WWW pourra servir de façade pour un serveur de bases de données en zone privée. Ces serveurs en zone publique qui ne servent que de relais sont souvent nommés serveurs mandataires (*proxy servers* en anglais).

La figure 7.4 représente un tel dispositif, avec un routeur d'entrée qui donne accès à la DMZ et un coupe-feu (*firewall*), qui est en fait un routeur un peu particulier dont nous détaillerons le rôle ci-dessous, qui donne accès à un réseau privé.

Le relayage entre zone publique et zone privée fonctionne aussi dans l'autre sens : l'utilisateur en zone privée remet son courrier électronique au serveur privé, qui l'envoie au relais en zone publique, qui l'enverra au destinataire. Pour consulter une page sur le WWW, l'utilisateur s'adresse au serveur relais qui émettra la vraie requête vers le monde extérieur. Ici le relayage peut procurer un autre avantage, celui de garder en mémoire cache les pages obtenues pendant

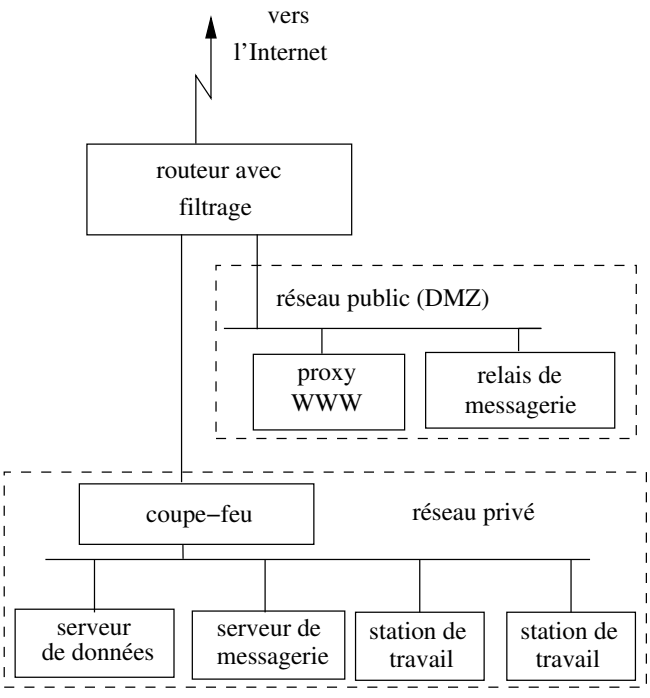


Figure 7.4 : *Réseau avec DMZ et coupe-feu*

un certain temps, pour les fournir au même utilisateur ou à un autre lorsqu'il les redemandera sans avoir à accéder à la source distante (l'analyse statistique des requêtes révèle que dans un contexte donné les gens accèdent souvent aux mêmes pages).

Le filtrage par port permettra la communication entre le proxy et le serveur en zone privée de façon contrôlée. Les routeurs disposent de fonctions de filtrage assez élaborées, permettant de distinguer quels protocoles et quels numéros de ports sont autorisés selon l'origine et la destination, et si telle communication a été initiée depuis un nœud à l'intérieur ou à l'extérieur du réseau.

Une pratique courante consiste à placer à l'entrée du réseau privé un coupe-feu (*firewall*). Un coupe-feu est un ordinateur qui filtre les communications un peu comme un routeur, d'ailleurs il est possible de configurer un routeur pour lui faire jouer le rôle d'un coupe-feu simple. Le coupe-feu a généralement des possibilités plus étendues, notamment en termes de suivi de connexion et d'analyse des paquets. Plus précisément, un routeur doit décider au coup par coup du sort de chaque paquet individuel, avec très peu de possibilité d'analyse historique, un coupe-feu peut disposer d'informations plus complètes, peut garder des datagrammes en file d'attente jusqu'à reconstituer une séquence plus

longue de la communication et en faire une analyse plus complète. Bien sûr ceci a un coût en termes de débit...

Routeur filtrant et coupe-feu, configurés pour repousser certaines attaques en rejetant des paquets appartenant à des connexions suspectes, produisent des fichiers de comptes rendus *(logs)* qui relatent les incidents. Il est bien sûr indispensable, pour bénéficier de la protection qu'ils sont censés procurer, d'analyser le contenu de ces fichiers et de les confronter avec les avis publiés par les CERT *(Computer Emergency Response Teams)* (sur les CERT voir la section suivante 7.6). Ceci suppose des ingénieurs compétents pour ce faire, ce qu'oublient certaines entreprises qui se croient protégées en achetant (fort cher) un coupe-feu clés en mains, configuré avec des filtres pertinents à un instant donné mais périmés quinze jours plus tard, et qui se retrouvent ainsi dotés d'une magnifique ligne Maginot[8].

7.6 Les CERT *(Computer Emergency Response Teams)*

Organisation des CERT

Les CERT *(Computer Emergency Response Teams)*, comme leur nom l'indique, centralisent, vérifient et publient les alertes relatives à la sécurité des ordinateurs, et notamment les annonces de vulnérabilités récemment découvertes. Les alertes peuvent émaner des auteurs du logiciel, ou d'utilisateurs qui ont détecté le problème. Détecter une vulnérabilité ne veut pas dire qu'elle soit exploitée, ni même exploitable, mais le risque existe.

Les vulnérabilités publiées par les CERT sont relatives à toutes sortes de systèmes; leur publication constitue une incitation forte pour que les industriels concernés (les producteurs du système ou du logiciel le plus souvent) les corrigent. Certains tentent aussi de ralentir le travail des CERT, dont ils aimeraient bien qu'ils ne dévoilent pas leurs faiblesses...

Le premier CERT a vu le jour à l'Université Carnegie-Mellon de Pittsburgh à la fin des années 1980. En 2018 la France dispose de trois CERT principaux (il y a des CERT privés ou d'intérêt local): le CERTA pour les besoins des administrations et services publics, le CERT Renater s'adresse aux Universités et centres de recherche, le CERT-IST s'adresse au monde industriel. En fait la coopération entre les CERT est assez étroite.

La publication des avis des CERT est une contribution majeure et vitale à la sécurité des systèmes d'information. Simplement le volume est tel que leur dépouillement représente un travail considérable par des personnes compétentes.

8. Le lecteur non français peut ignorer le nom de cette ligne de fortifications infranchissable établie par l'armée française dans les années 1930 pour se prémunir d'une invasion allemande, et qui n'avait que le défaut de ne pas être placée à l'endroit où cette invasion s'est produite en 1940.

Faut-il publier les failles de sécurité ?

Un débat s'est engagé sur le bien fondé de certains avis, et sur la relation qu'il pourrait y avoir entre le nombre d'avis concernant un logiciel ou un système donné et sa qualité intrinsèque. Les détracteurs des logiciels libres ont par exemple mis en exergue le volume très important d'avis des CERT qui concernaient ceux-ci (par exemple Linux, le serveur WWW *Apache*, *Sendmail*, etc.) pour en inférer leur fragilité. Leurs défenseurs ont riposté en expliquant que les avis des CERT concernaient par définition des failles de sécurité découvertes et donc virtuellement corrigées, alors que l'absence d'avis relatifs à tel système commercial pouvait simplement signifier que son auteur passait sous silence ses défauts de sécurité en profitant de son opacité. Or l'expérience prouve que tout dispositif de sécurité a des failles ; les vrais attaquants ne perdent pas leur temps à faire de la recherche fondamentale sur la factorisation des grands nombres entiers, ils essaient de repérer les failles d'implémentation et ils les exploitent. Face à ce risque la meilleure protection est une capacité de riposte rapide, qui consiste le plus souvent à commencer par désactiver le composant pris en défaut en attendant la correction. La communauté du logiciel libre excelle dans cet exercice, mais avec les logiciels commerciaux les utilisateurs n'ont souvent aucun moyen d'agir sauf à attendre le bon vouloir de leur fournisseur. Dans ce contexte, la publication d'avis des CERT relatifs à des logiciels commerciaux est très bénéfique parce qu'elle incite les fournisseurs à corriger plus rapidement un défaut dont la notoriété risque de nuire à leur réputation. Mais certains fournisseurs cherchent à obtenir le silence des CERT en arguant du fait que leurs avis risquent de donner aux pirates des indications précieuses... ce qui est fallacieux parce que les sites WWW des pirates sont de toute façon très bien informés et mis à jour, eux, selon les principes du logiciel libre, ce qui indique où est l'efficacité maximum. L'expérience tend à prouver qu'une faille de sécurité est d'autant plus vite comblée qu'elle est publiée vite et largement. L'accès au code source du logiciel en défaut est bien sûr un atout supplémentaire.

Chapitre 8 De Multics à Unix et au logiciel libre

Sommaire

8.1 Un échec plein d'avenir

Comme nous l'avons déjà mentionné à la section 3.10.1, Multics est né en 1964 au MIT (*Massachusetts Institute of Technology*) dans le cadre d'un projet de recherche nommé MAC, sous la direction de Fernando Corbató. La même équipe avait déjà créé le système CTSS. Multics était destiné aux ordinateurs General Electric de la famille GE 635, pour lesquels le constructeur fournissait de son côté un système d'exploitation plus conventionnel. Le projet associait le MIT, General Electric et les *Bell Telephone Laboratories* (une filiale d'*AT&T, American Telegraph*

and Telephone, qui détenait le monopole des télécommunications pour les États-Unis).

L'objectif du projet était la réalisation d'un grand système informatique capable de fournir des services interactifs en temps partagé à un millier d'utilisateurs simultanés. Multics comportait beaucoup d'innovations de grande portée : le langage de commande pour piloter le fonctionnement de la machine était un langage de programmation, le même que celui dont disposait l'utilisateur pour interagir avec le système, le *shell* évoqué à la section 3.10.1. Le système d'exploitation était écrit en langage évolué (en l'occurrence PL/1), voie ouverte par les systèmes Burroughs écrits en Algol, mais encore peu fréquentée. Les concepts de mémoire centrale pour les données volatiles et de fichiers pour les données persistantes étaient fondus en un concept unique de mémoire virtuelle segmentée, certains segments étant dotés de la qualité de persistance, comme décrit à la section 5.4. Les segments persistants étaient catalogués dans des répertoires à l'organisation arborescente tels que ceux décrits à la section 5.2.1. Moins spectaculaire en apparence mais peut-être aussi importante était l'existence de toute une collection d'outils programmables et combinables entre eux destinés à faciliter le travail d'un type d'utilisateur : le programmeur, et plus spécialement celui qui écrivait un système d'exploitation.

Malgré ou à cause de toutes ces qualités uniques et promises à un grand avenir, Multics fut en gros un échec, au moins du point de vue de sa diffusion. Ses mérites furent reconnus tardivement, et le plus souvent tacitement, par ceux qui en reprirent les idées à leur compte.

Cette méconnaissance longue des mérites de Multics a ses raisons : la technologie des ordinateurs disponible à l'époque de sa diffusion, essentiellement pendant les années 1970, ne permettait d'implémenter les méthodes et les concepts élaborés de ce système qu'au prix d'une grande lourdeur. Les ordinateurs exploités sous Multics étaient gros, chers et lents. L'institut de statistique qui employait à l'époque l'auteur de ces lignes en avait acquis un, fruit de la fusion CII-Honeywell-Bull, et ses experts en informatique démographique avaient essayé d'imaginer les moyens de traiter avec cet engin les recensements de la population, mais ils avaient fini par regarder ce drôle de système un peu de l'œil de la poule qui a couvé un œuf de cane.

Leur perplexité n'avait d'ailleurs d'égale que celle des ingénieurs commerciaux qui essayaient de vendre la chose, ou celle des ingénieurs spécialistes de Multics auxquels nous posions des questions incongrues comme « Pourrions-nous traiter des fichiers sur bande magnétique en traitement par lots ? », méthode béotienne mais indispensable à l'époque du fait de la faible capacité des disques.

Il résulta de cette expérience peu de résultats nouveaux dans le travail statistique courant, mais un sursaut intellectuel parmi ceux des informaticiens qui

n'avaient pas définitivement sombré dans la léthargie coboliste [1] dont on n'émergerait que pour sombrer dans Windows. Si changer de langage ou de système ne rend pas plus intelligent, il y a des systèmes et des langages qui incitent à la réflexion, et d'autres moins. Cela dépend pour une grande part du type d'initiative laissé à l'utilisateur et du niveau d'intelligibilité que le système exhibe. Il y a des systèmes dont les paramètres de fonctionnement sont enfouis dans des fichiers binaires inaccessibles à l'utilisateur, qui n'a qu'un seul choix : cliquer sur des menus en espérant que cela finira par produire un résultat désiré ; c'est le cas de Windows. Unix au contraire contient tous ses paramètres dans des fichiers texte lisibles et réunis dans un endroit connu (le répertoire **/etc**). La première méthode n'est justifiable que si la réalisation du système est sans faille, et autorise l'utilisateur à ne jamais se préoccuper de ces paramètres, ce qu'ont réussi les concepteurs de MacOS, mais pas ceux de Windows ; elle suppose aussi que lesdits utilisateurs ne sont que cela, utilisateurs, qu'ils ne nourrissent aucun intérêt pour le système et n'envisagent pas une seconde de se mettre à le modifier, ou du moins à en modifier le comportement en jouant sur les paramètres.

Multics, sous cet angle comme sous certains autres, était un précurseur d'Unix, système qui considère ses utilisateurs comme des personnes intelligentes, mais aussi suffisamment intéressées par le système lui-même pour lui consacrer un temps non négligeable dont les employeurs ne comprennent pas toujours la fécondité. C'est ainsi que des industriels ont inventé pour Unix une interface utilisateur nommée CDE *(Common Desktop Environment)* dont le principe est de plaquer sur le système une sorte de super-Windows dont le paramétrage, réservé à des administrateurs, est ensuite propagé à la masse des utilisateurs. Cette vision centralisée et hyper-organisée aurait sans doute bien réussi dans les années 1960, mais elle risque de ne pas résister aux sables mouvants de la sociologie réelle des organisations des années 2000.

8.2 Où l'on commence à rêver à Unix

Les créateurs d'Unix, Ken Thompson et Dennis M. Ritchie, avaient une forte expérience de Multics, et ils savaient aussi bien ce qu'ils voulaient en retenir que ce qu'ils en rejetaient. Ils en retenaient notamment les aspects suivants :

– Le système est écrit non pas en assembleur, mais dans un langage de haut niveau (PL/1 pour Multics, C pour Unix). Seuls quelques fragments du code du noyau intimement liés à un matériel particulier sont en assembleur.

1. Adjectif formé sur le nom COBOL, langage de programmation particulièrement aride et punitif.

Ceci facilite le *portage*[2] du système sur un nouveau modèle d'ordinateur. On a pu dire que le langage C était un assembleur portable.

– Le système de commandes qui permet de piloter le système est le même interpréteur qui permet à l'utilisateur d'exécuter des programmes et des commandes, et il donne accès à un langage de programmation. C'est le *shell*.
– Le système de fichiers d'Unix est très inspiré de celui de Multics, d'où vient aussi l'idée d'exécuter chaque commande comme un processus distinct.
– Mais surtout, comme Dennis Ritchie l'a expliqué dans son article de 1979, ce que lui et ses collègues des *Bell Laboratories* voulaient retrouver de Multics en créant Unix, c'était un système qui engendrait pour ainsi dire spontanément la communication et l'échange d'expériences entre ses adeptes.

Cette qualité, partagée par Multics et Unix, d'être propice à la création d'une communauté ouverte et communicative, mérite que l'on s'y arrête. Lorsque Multics a été introduit dans l'Institut statistique évoqué ci-dessus, il y a immédiatement cristallisé la formation d'une petite communauté intellectuelle, que la Direction n'a d'ailleurs eu de cesse de résorber parce qu'elle n'en comprenait pas la fécondité et qu'elle percevait son activité comme un gaspillage. D'innombrables expériences similaires ont eu lieu autour de Multics et d'Unix, sans qu'une cause unique puisse leur être attribuée. Le fait que ces systèmes aient été créés par des chercheurs, habitués à l'idée que la connaissance soit objet de partage gratuit et de communication désintéressée mais assortie de plaisir, est un élément. L'existence de logiciels commodes pour la création de textes, le courrier électronique et les forums en ligne a aussi joué, mais cette existence était-elle une cause ou une conséquence ? La nature programmable du *shell*, l'accès possible pour tous aux paramètres du système, inscrits dans des fichiers de texte ordinaires, encourageaient un usage intelligent du système, et l'intelligence va de pair avec l'échange.

Si l'on compare Multics et Unix aux systèmes industriels disponibles à l'époque, comme l'OS 360, GCOS 8 ou plus tard VMS de Digital Equipment, il apparaît que ces derniers ne sont pas conçus dans le même esprit : l'utilisateur dispose d'un mode d'emploi du système, réputé contenir les solutions pour tout

2. *Porter* un logiciel d'un ordinateur à un autre ou d'un système à un autre signifie l'adapter aux caractéristiques techniques particulières de ce nouveau contexte. L'opération s'appelle un *portage*. Un logiciel pour le portage duquel le travail à faire est nul ou négligeable est dit *portable* ; cette qualité, très recherchée, s'appelle *portabilité*. Unix est le système d'exploitation le plus portable parce qu'il est écrit pour l'essentiel dans un langage évolué (C) plutôt que dans l'assembleur d'un processeur particulier, et grâce à la bonne abstraction de ses primitives, grâce aussi à la simplicité et à l'élégance de son architecture générale.

problème qu'il pourrait se poser. Lorsque tout ceci est bien fait, comme par exemple dans VMS, le système à mémoire virtuelle conçu pour l'ordinateur VAX, cela suscite un usage efficace et commode mais passif du système.

L'auteur de ces lignes a été un utilisateur longtemps réticent et sceptique d'Unix, dérouté par l'aspect « caisse à outils » du système. VMS, que je pratiquais simultanément, avait été conçu par une équipe de (très bons) ingénieurs, soucieux de livrer un produit homogène et cohérent, et ils avaient parfaitement réussi. La meilleure preuve de cette réussite était la documentation du système, souvent un aspect un peu négligé : celle de VMS était une merveille de clarté et d'exhaustivité, au prix certes d'un nombre impressionnant de mètres linéaires de rayonnage. Mais quel que soit le problème à résoudre, on était sûr que la réponse était dans la « doc ». Lorsque Digital Equipment a produit sa propre version d'Unix, ils ont publié un petit manuel résumé des commandes Unix, baptisé « *The little grey book* » (la couleur canonique de la documentation Digital venait de virer de l'orange au gris). Par opposition, la documentation VMS s'est trouvée baptisée « *The big grey wall* ».

Habitué donc à l'univers confortable et hyper-balisé de VMS, je découvrais avec Unix un système de prime abord beaucoup moins homogène, même si je devais découvrir plus tard que son homogénéité résidait ailleurs. Comme chaque commande Unix s'exécute sous le contrôle d'un processus distinct, elle peut être assez découplée du noyau du système. Cette modularité, qui est un avantage, avait permis de confier l'écriture de beaucoup de commandes à des étudiants en stage, quand elles n'étaient pas tout simplement des contributions spontanées, et alors leur qualité et celle de leur documentation pouvaient être assez inégales.

De Multics les créateurs d'Unix rejetaient la lourdeur. La tentation fatale, pour des auteurs de systèmes informatiques en général, et de systèmes d'exploitation ou d'architectures de processeurs tout particulièrement, consiste à céder au perfectionnisme et à réaliser des dispositifs qui ajouteront au système global une complexité considérable pour résoudre des problèmes qui ne surgiront que très rarement. Les problèmes rares peuvent se contenter de solutions inefficaces mais simples. Force est de constater que les auteurs de Multics n'ont pas évité cette ornière. VMS non plus d'ailleurs, qui succédait aux merveilleux RSX-11M et autres IAS.

Frederick P. Brooks, le concepteur de l'OS/360, dans son livre justement célèbre *The Mythical Man-Month* [24], décrit ce qu'il appelle le *syndrome du second système*, et qui s'applique à Multics comme à l'OS/360 : une équipe constituée en grande partie des mêmes hommes autour de Fernando Corbató avait développé avec succès CTSS ; en s'attaquant à Multics ils ont voulu y introduire tous les perfectionnements coûteux qu'ils avaient, avec sagesse mais frustration, écartés de leur œuvre précédente. En informatique comme ailleurs, point de salut sans une part de renoncement.

En fait, à la fin des années 1960, l'échec de Multics aux *Bell Labs* était patent. L'équipe qui allait y concevoir Unix, autour de Ken Thompson et Dennis Ritchie, comprit que Multics ne serait pas utilisable pour un travail réel dans un délai raisonnable. De son côté le propriétaire de Multics, la compagnie *General Electric*, se sentait assez peu concerné par ce système développé par des chercheurs universitaires et préférait commercialiser ses ordinateurs avec son système conventionnel, GECOS. Lorsque Multics deviendrait utilisable, à la toute fin des années 1970, les ordinateurs qu'il pouvait piloter étaient définitivement périmés et sans espoir de succession.

Dans un article de 1979 [110] Dennis Ritchie a décrit cette période où les *Bell Labs* se retiraient du projet Multics. Ce processus s'accompagnait d'un autre facteur d'incertitude : une réorganisation visait à séparer les équipes de recherche en informatique des équipes de l'informatique opérationnelle ; ce genre de séparation, conforme aux vues des managers financiers efficaces et à jugeote courte, a pour conséquence habituelle de diminuer les moyens disponibles pour la recherche et de réduire la qualité de l'informatique opérationnelle, privée de stimulation intellectuelle. Le groupe de D. Ritchie, K. Thompson, M. D. McIlroy et Joseph F. Ossanna souhaitait conserver l'environnement de travail luxueux que Multics leur procurait à un coût d'autant plus exorbitant qu'ils en étaient les derniers utilisateurs. Pour ce faire ils allaient développer leur propre système sur un petit ordinateur bon marché et un peu inutilisé récupéré dans un couloir, un PDP 7 de Digital Equipment. Unix était sinon né, du moins conçu.

8.3 Les hommes d'Unix[3]

Le lecteur, à la fin de l'alinéa précédent, se sera peut-être fait la réflexion que pour que des employés d'une grande entreprises puissent développer un système d'exploitation, même ascétique, pendant leur temps libre, il fallait que leur encadrement ne soit pas trop rigide. Parce que ce même lecteur devrait maintenant être convaincu que le système d'exploitation est l'objet technique le plus complexe que l'homme ait conçu et réalisé au cours du XX^{e} siècle. Quelle était au fait la mission théorique de ces jeunes gens ? Qui contrôlait la réalisation de leurs objectifs ?

Peter H. Salus a écrit un livre (*A Quarter Century of UNIX,* [115]) qui met en scène les principaux acteurs de la naissance d'Unix. De prime abord, on est frappé en lisant ces aventures de découvrir que cette création, qui a eu des répercussions considérables dans les domaines technique autant qu'industriel

3. Effectivement, peu de femmes dans cette histoire. Raison de plus pour mentionner la regrettée Evi Nemeth, et, peut-être pas tout à fait dans le même domaine ni à la même époque, Radia Perlman, spécialiste des protocoles de réseau, et Elizabeth Zwicky.

et économique, n'a vraiment été décidée ni par un groupe industriel, ni par un gouvernement, ni par aucun organisme doté de pouvoir et de moyens financiers importants. On peut d'ailleurs en dire autant de l'Internet (pour des détails, voir la section 6.5.3), une autre création aux répercussions considérables, d'ailleurs très liée à Unix et issue du même milieu social.

À la lecture du livre de Salus, quiconque a un peu fréquenté les milieux scientifiques d'une part, les milieux industriels de l'autre, ne peut manquer d'être frappé par le caractère décalé, pour ne pas dire franchement marginal, de la plupart des acteurs de la genèse unixienne.

Le monde de la science a sa hiérarchie, où les disciplines spéculatives et abstraites ont le pas sur les recherches appliquées et les disciplines descriptives, et où bien sûr les chercheurs sont patriciens et les ingénieurs et techniciens ilotes, entourés d'une population au statut incertain, les étudiants en thèse ou en post-doc, dont une minorité d'élus accédera au patriciat mais dont la majorité ne deviendra même pas ilote, contrainte à descendre aux enfers, c'est-à-dire le monde réel des entreprises industrielles et commerciales.

Dans cet univers social, l'informatique, discipline récente et mal identifiée, perçue (au mépris de toute vraisemblance, mais qu'importe au sociologue) comme un vague sous-produit de la branche la moins noble des mathématiques (l'analyse numérique), se situe plutôt vers le bas. Au sein de la discipline, le haut du pavé est tenu par les domaines où il y a une théorie possible, de préférence mathématique ou à la rigueur physique : linguistique de la programmation, algorithmique (surtout numérique ou logique), traitement de l'image ou du signal en général. Les systèmes d'exploitation disposent de tout un arsenal de concepts, mais pas d'une théorie, c'est ainsi ; de surcroît ils sont bien près du matériel, cette chose qui a des relents de cambouis et de sueur. Donc ils sont en bas. Alors des ingénieurs qui s'occupent de systèmes d'exploitation…

Le monde industriel (nous nous plaçons à l'époque de la naissance d'Unix, avant la prise de pouvoir par les financiers à costume noir et cortex de calmar) a un système de valeurs symétrique de celui de la science : on y respecte celui qui *fait* des choses, de vraies choses. C'est un univers dominé par les ingénieurs, qui sont censés se coltiner avec la matière. On sait bien qu'une industrie dynamique doit avoir des centres de recherche, et que dans ces endroits travaillent des types bizarres appelés chercheurs, mais même si on ne les méprise pas vraiment, ils sont considérés avec une certaine distance.

Or que nous apprend Salus ? Thompson et Ritchie étaient chercheurs dans une entreprise industrielle. Au fur et à mesure de leur apparition, les noms de ceux qui font fait Unix, parmi eux Kirk McKusick, Bill Joy, Eric Allman, Keith Bostic, sont toujours accompagnés d'un commentaire de la même veine : ils étaient étudiants *undergraduates* ou en cours de PhD, et soudain ils ont découvert qu'Unix était bien plus passionnant que leurs études. Bref, les auteurs d'Unix n'ont jamais emprunté ni la voie qui mène les ingénieurs perspicaces vers

les fauteuils de Directeurs Généraux, ni celle que prennent les bons étudiants vers la *tenure track*, les chaires prestigieuses, voire le Nobel[4].

8.4 Introduction à la démarche unixienne

Comme le note Christian Queinnec aux premiers mots de son livre « ABC d'Unix » [106], « UNIX est un système de production de programme ». Et, conformément à l'esprit d'Unix, cette assertion est à prendre à la fois de façon extensive : ce système comporte tout ce dont peut rêver un auteur de programme, et, aussi, de façon restrictive : malgré quelques concessions récentes, il ne comporte fondamentalement *rien d'autre*. Les Unix modernes tels que Linux sont certes dotés de logiciels utilisables par le commun des mortels, avec des interfaces graphiques, mais les vrais unixiens n'en abusent pas.

La documentation canonique d'Unix (les *man pages*) constitue une excellente entrée en matière : aucun effort pédagogique, aucune de ces redondances qui facilitent l'acquisition d'une notion. Les mots en sont comptés, aucun ne manque mais pas un n'est de trop. La lecture attentive (très attentive) de ces pages délivre une information nécessaire et suffisante à l'usage du système, d'où une locution proverbiale souvent proférée par les Unixiens expérimentés en réponse à un néophyte qui demande de l'aide : « RTFM! » *(Read the f... manual!)*. On le voit, Unix est à l'opposé de ces logiciels à interface graphique dont les vendeurs laissent croire qu'ils peuvent être utilisés sans lire aucune documentation[5].

À qui était, comme l'auteur, habitué aux systèmes des grands ordinateurs IBM des années 1970, ou au système VMS que Digital Equipment Corporation (DEC) avait développé pour ses ordinateurs VAX, la transition était rude. Les systèmes d'IBM et de DEC étaient conçus dans le but d'élargir l'audience de l'informatique à des utilisateurs moins professionnels, à une époque où les micro-ordinateurs n'existaient pas. Pour ce faire, la syntaxe de leur langage de commandes cherchait

4. On sait qu'Alfred Nobel, lorsqu'il créa ses Prix, ne voulut pas en attribuer aux Mathématiques. La légende dit que cette exclusion serait due à une trop grande sympathie qu'aurait éprouvée Madame Nobel pour un certain mathématicien. Pour se consoler les mathématiciens créèrent la médaille Fields, décernée tous les quatre ans. Les informaticiens ont encore plus besoin de consolation, puisqu'ils n'ont même pas réussi à séduire Madame Nobel. Ils ont créé le *Turing Award*, qui a notamment été décerné, parmi nos personnages, à Maurice V. Wilkes, E.W. Dijkstra, Donald E. Knuth, C. Antony R. Hoare, Frederick P. Brooks, Fernando Corbató, Ken Thompson, Dennis M. Ritchie, Leslie Lamport, Whitfield Diffie et Martin Hellman. Voir `http://www.acm.org/awards/taward.html` pour plus de détails.

5. Cette prétention flatte la paresse naturelle de l'utilisateur, mais elle est fallacieuse. Il est certes possible, avec tel logiciel de traitement de texte dont le nom signifie « mot » dans une langue germanique et insulaire, de créer facilement un document laid et peu lisible, mais dès lors que l'on veut un résultat présentable il faut se plonger dans la documentation et découvrir que ce logiciel est complexe, tout simplement parce que la typographie est complexe.

à s'adoucir en utilisant des lexèmes plus proches du langage humain, en tolérant des abréviations ou au contraire des tournures plus bavardes mais plus faciles à mémoriser. La réponse du système à une commande était aussi édulcorée : présentation aérée, commentaires explicatifs.

Pour un Unixien, toutes ces variations pédagogiques ne sont que concessions coupables à l'ignorance informatique des secrétaires et des comptables. L'initiation informatique des ces professions respectables est peut-être un objectif louable, mais dont il ne veut rien savoir, et Unix non plus, parce que pour un développeur[6] ces aides pédagogiques sont autant d'obstacles à son travail.

La syntaxe des commandes Unix est sèche comme un coup de trique, d'une part parce qu'elles sont destinées à des professionnels qui les connaissent par cœur à force de les utiliser à longueur de journée, d'autre part parce qu'elles constituent un langage de programmation (le *shell*) qui permettra d'automatiser des opérations répétitives, et que pour un langage toute souplesse syntaxique se paye en espace et en temps (il faut bien écrire les instructions qui vont interpréter les commandes, et autant de variations possibles autant de dizaines de lignes de code en plus).

La réponse du système à l'utilisateur qui lui soumet une commande est tout aussi austère, le plus souvent d'ailleurs il n'y a pas de réponse. Ainsi, si vous voulez modifier le nom d'un fichier, et que le nouveau nom que vous souhaitez lui donner est déjà pris par un autre fichier, si le renommage est effectué le fichier homonyme sera détruit. Les systèmes à l'usage des secrétaires, des comptables ou des présidents d'université, face à un tel cas, posent la question à l'utilisateur : « veux-tu vraiment détruire l'autre fichier ? », ou renomment le fichier menacé. Pour Unix rien de tel : le fichier est froidement détruit sans même une notification

6. C'est avec Unix que « développeur » a supplanté « programmeur ». Ces deux termes ont rigoureusement le même sens, « personne dont le métier est la création de programmes informatiques », mais « programmeur »avait été victime d'une dévalorisation injuste. Les premiers managers de l'informatique ont pensé y reproduire un schéma qui était déjà en déclin dans l'industrie : l'ingénieur conçoit, l'ouvrier fait. En informatique l'analyste concevrait ce que le programmeur ferait. Erreur. L'ingénieur disposait, pour transmettre à l'ouvrier la description de ce qu'il devait faire, d'un outil précis et rigoureux, le dessin industriel. Rien de tel en informatique, malgré des efforts qui durent encore pour créer des systèmes de spécification infaillibles dont UML est le dernier avatar. Et à cela il y a des raisons : un ordinateur doté de son système d'exploitation et de ses langages de programmation n'est pas seulement infiniment plus versatile et plus souple qu'un étau-limeur ou qu'une fraiseuse, il est surtout d'une tout autre nature. Il traite de l'information, et comme nous l'avons vu l'information n'est que parce qu'imprévisible. Et l'information qui décrit un programme à réaliser est de la méta-information, puisque le programme est lui-même de l'information. De ce fait la vraie difficulté réside bien toujours dans l'écriture du programme, qui est un exercice incroyablement délicat et laborieux. E.W. Dijkstra se définit lui-même comme programmeur. Mais rien n'y fait, le programmeur sent le cambouis et la sueur, alors que son remplaçant le développeur arbore (jusqu'à la retraite et au-delà) l'uniforme seyant et l'éthos décontracté des étudiants californiens.

post mortem. C'est un système pour les vrais hommes, qui savent ce qu'ils font, assument leurs erreurs et savent les réparer.

Mais à cet ascétisme il y a une raison encore plus dirimante, et qui n'est pas de l'ordre du sado-masochisme. L'invention sans doute la plus géniale d'Unix est la possibilité, par la simple syntaxe du *shell*, de réaliser des opérations de composition de processus, au sens algébrique du terme.

Les opérations les plus simples consistent à rediriger les résultats de sortie d'une commande vers un fichier, et à donner en entrée à une commande des données stockées dans un fichier, ce qui n'est pas encore de la composition de processus. Mais pour réaliser ceci il fallait déjà standardiser les formats d'entrée et de sortie des commandes, et découpler les mécanismes d'entrée et de sortie, pour introduire les notions d'entrée standard et de sortie standard, ce qui ouvrait la voie à des réalisations plus ambitieuses.

L'opérateur de composition de processus en séquence est « `;` ». On remarque la concision. « `a ; b` »se lit : exécuter la commande `a`, puis la commande `b`. La plupart des commandes Unix s'exécutent comme un processus indépendant. Le lancement d'un programme créé par un utilisateur obéit à la même syntaxe et aux mêmes règles, ce qui encourage les développeurs à utiliser les conventions des commandes Unix, et ainsi à contribuer à l'enrichissement du système.

L'opérateur de composition de processus parallèles asynchrones est « `&` ». « `a & b` »se lit : lancer l'exécution de `a`, puis sans attendre qu'elle se termine lancer aussitôt `b`. Les deux processus seront concomitants (et concurrents pour l'acquisition du contrôle du processeur).

L'opérateur de composition de processus parallèles synchrones est « `|` ». « `a | b` »se lit : lancer l'exécution de `a`, puis lancer `b` qui va aussitôt attendre la première ligne de résulat issue de `a`, la traiter puis se bloquer en attente de la suivante, etc.

Prenons un exemple simple : je veux la liste de tous les processus en cours d'exécution qui exécutent le serveur WWW *Apache*, avec leur numéro de processus.

La commande qui affiche la liste des processus s'appelle « `ps` », qui doit, pour afficher non seulement les processus de l'utilisateur mais tous les autres, être agrémentée des paramètres `a` et `x`, ce qui s'écrit donc « `ps ax` ». Cette commande va produire la liste de tous les processus, avec leur numéro et le nom du programme exécuté, à raison d'une ligne par processus. Je veux filtrer cette liste pour n'en retenir que les lignes où le nom de programme qui apparaît est `apache`.

Parmi les plus belles commandes d'Unix il faut citer `grep` (pour *Global (search for) Regular Expression and Print*, analyseur général d'expressions régulières). Cette commande peut faire des choses très savantes, mais nous allons l'utiliser de façon très simple, pour retrouver une chaîne de caractères dans le texte soumis à son entrée standard. « `grep apache` » signifie : si la ligne

de l'entrée standard contient le texte « `apache` », afficher le texte à l'écran, sinon passer à la suivante.

Nous allons composer les deux commandes « `ps ax` » et « `grep apache` » par l'opérateur de composition parallèle synchrone « `|` »:

```
ps ax | grep apache
```

Chaque ligne issue de la première commande sera soumise à la seconde pour analyse, ce qui réalisera le filtre souhaité:

```
> ps ax | grep apache
  284 ?        S      0:00 /usr/sbin/apache
  295 ?        S      0:00 /usr/sbin/apache
  296 ?        S      0:00 /usr/sbin/apache
  297 ?        S      0:00 /usr/sbin/apache
  298 ?        S      0:00 /usr/sbin/apache
  299 ?        S      0:00 /usr/sbin/apache
  434 pts/0    S      0:00 grep apache
```

Je reçois ainsi la liste de tous les processus *Apache* avec leur numéro, et en prime le processus d'analyse, puisque sa ligne de commande comporte elle aussi le texte `apache`.

Cette métaphore du filtre est promue au rang de paradigme par UNIX: les programmes vraiment unixiens doivent être écrits comme des filtres, c'est à dire recevoir un flux de caractères sur leur entrée standard et émettre un flux de caractères (convenablement modifié) sur leur sortie standard, ce qui permet de les combiner *ad libitum*.

C'est le livre de Jean-Louis Nebut [97] qui me semble-t-il explicite le mieux la syntaxe du *shell* en termes de filtres et de composition de processus. Il est aisé de concevoir que le fonctionnement d'un tel système suppose une discipline assez ascétique dans la syntaxe des commandes et leur mode d'interaction. Notamment, puisqu'elles doivent être les syntagmes d'un langage de programmation, dont les programmes sont usuellement appelés *shell scripts*, il n'est pas souhaitable que les commandes engagent un dialogue avec l'utilisateur, qui dans ce cas ne sera pas un humain mais le système d'exploitation.

8.5 Dissémination d'Unix

8.5.1 Un système exigeant

Nous venons de dire qu'Unix était un système de production de programme, conçu pour satisfaire les programmeurs professionnels et à peu près personne d'autre. Comment expliquer alors qu'il se soit répandu dans beaucoup d'autres

domaines, souvent au prix de beaucoup de crises de nerfs de la part d'utilisateurs furieux? Parce qu'il faut bien dire qu'Unix n'est pas un système confortable pour qui n'est pas disposé à y consacrer la moitié de son temps de travail.

L'auteur de ces lignes, il y a de nombreuses années, a compris que s'il voulait espérer garder l'estime de certains collègues il lui fallait savoir se servir assez couramment d'Unix et surtout de l'éditeur de texte *Emacs* avec lequel d'ailleurs il compose le présent texte. Cette prise de conscience a entraîné de nombreuses et lourdes conséquences. Il en va d'Emacs comme d'Unix: aucun espoir d'acquérir un minimum de maîtrise de cet éditeur (génial) sans plusieurs heures de pratique quotidienne, qui au bout de quelques mois permettront de savoir raisonnablement utiliser quelques dizaines parmi ses 14 000 et quelques fonctions, sans parler du langage de programmation qui lui est incorporé. Bien, il fallait s'y mettre, et pour cela une seule solution: utiliser Unix et Emacs pour tout, rédaction de documents, courrier électronique, lecture des *News*.

De ce type de situation on serait tenté d'inférer une conception un peu paradoxale du métier d'informaticien: le travail consisterait essentiellement à rester en symbiose avec le système et les outils de base comme Emacs, à se tenir au courant de leurs évolutions en fréquentant des collègues, par des rencontres, des colloques, les *News*, à essayer les nouveaux logiciels dès leur sortie, et, logiquement, à en produire soi-même. Il va de soi que dans cette perspective les demandes trop précises d'un employeur qui n'aurait pas compris ce processus seraient perçues comme autant d'obstacles mesquins. Le trait ici est bien sûr forcé, mais l'employeur avisé n'oubliera pas que ses ingénieurs, pour rester compétents, ont besoin de temps pour les activités énoncées ci-dessus.

La conclusion qui semblerait logique après la lecture des lignes précédentes serait qu'Unix aurait dû disparaître sous les coups furieux des DRH et des chefs de projet, ou du moins aurait dû rester cantonné à un petit monde de chercheurs, de développeurs et de hobbyistes. Il n'en a rien été pour au moins deux raisons exposées à la section 8.5.2.

8.5.2 Naissance d'une communauté

Lorsqu'après quelques années de travail assez confidentiel il n'a plus été possible à *AT&T (American Telegraph and Telephone)* d'ignorer le travail de Thompson et Ritchie, celui-ci avait acquis une certaine ampleur et une certaine notoriété, notamment dans le monde universitaire. *AT&T* décida de vendre Unix assez cher aux entreprises[7], et se laissa convaincre d'en concéder (en 1974)

7. En fait à cette époque AT&T possédait le monopole des télécommunications aux États-Unis, en contrepartie de quoi il lui était interdit d'avoir une véritable action commerciale dans d'autres domaines comme l'informatique. Lorsque ce monopole fut aboli et AT&T démantelé, en 1982, Unix

l'usage gratuit aux Universités. Cette décision fut à l'origine de la popularité d'Unix dans le monde universitaire.

C'est en 1974 que Vinton Cerf (de l'Université Stanford) et Robert Kahn (de BBN) publièrent le premier article sur TCP/IP. Le travail sur les protocoles fut intense. En 1977 TCP/IP atteignit une certaine maturité, et c'est de ce moment que l'on peut dater la naissance de l'Internet expérimental.

En 1979 la DARPA lançait des appels d'offres assez généreux auprès des centres de recherche prêts à contribuer au développement de TCP/IP, elle souhaitait notamment l'adapter au VAX 11/780, l'ordinateur à mots de 32 bits que DEC venait de lancer (les PDP-11 étaient des machines à mots de 16 bits). Bill Joy, du *Computer Systems Research Group (CSRG)* de l'Université de Californie à Berkeley et futur fondateur de *Sun Microsystems*, convainquit la DARPA qu'Unix serait une meilleure plateforme que VMS pour ce projet parce qu'Unix avait déjà été porté sur plusieurs types d'ordinateurs.

De fait, dès 1977 le CSRG avait créé une version expérimentale d'Unix (la « 1BSD », pour *Berkeley System Distribution*) dérivée de la *Sixth Edition* des *Bell Labs*. La *Seventh Edition* de 1978 tournait sur DEC PDP-11 et avait été portée sur un ordinateur à mots de 32 bits, l'*Interdata 8/32* : elle fut portée sur VAX sous le nom de 32V, et le CSRG (nommément Bill Joy et Ozalp Babaoğlu) réunit ces diverses souches pour créer la 3BSD en 1979.

Le financement de la DARPA devait stimuler les deux projets : le développement de cette nouvelle version d'Unix nommée « Unix BSD », résolument tournée vers le monde des réseaux, et celui de ce que l'on connaît aujourd'hui sous le nom de TCP/IP. De ce jour, les développements respectifs de TCP/IP, de l'Internet et d'Unix furent indissociables. La souche *Bell Labs* continua son évolution indépendamment pour engendrer en 1983 la version *System V*. La suite de cette histoire se trouve ci-dessous à la section 8.5.3.

La disponibilité pratiquement gratuite pour les Universités, les subventions généreuses de la DARPA, c'était deux contributions de poids au succès d'Unix. Un troisième ingrédient s'y ajouta, sans lequel les deux autres n'eussent sans doute pas suffi : ce monde du réseau des centres de recherche était par ses traditions prédisposé aux échanges intellectuels, et justement la construction du réseau lui fournissait le moyen de s'y adonner de façon décuplée. Dans le monde scientifique d'antan, les contacts avec l'extérieur un peu lointain étaient l'apanage des patrons de laboratoire, qui allaient aux conférences où ils accédaient à des informations qu'ils pouvaient ensuite distiller pendant des séminaires suivis religieusement par les disciples. Avec le réseau, de simples étudiants ou de vulgaires ingénieurs pouvaient entrer en contact directement avec des collègues

put devenir une marque commerciale, et AT&T se lança même dans la fabrication d'ordinateurs sous Unix, sans grand succès.

prestigieux. En outre, ces échanges étaient indispensables, parce qu'Unix était gratuit, mais sans maintenance, les utilisateurs étaient contraints de s'entr'aider pour faire fonctionner leurs systèmes. Je me rappelle encore en 1981 les collègues de l'IRCAM qui administraient un des premiers VAX sous Unix d'Europe, toute leur maintenance logiciel était en direct avec Berkeley. Une communauté (scientifique ? technique ? dans les marges de la science et de la technique ?) allait se créer. L'association *USENIX* en serait une des instances visibles, mais elle resterait largement informelle.

Il est à noter que cette communauté serait assez vite internationale : pour les managers d'AT&T qui s'étaient laissé convaincre de concéder Unix gratuitement aux chercheurs, comme pour la DARPA, le monde informatisable se limitait aux États-Unis et à leur extension canadienne, ils n'avaient pas un instant envisagé l'existence d'Universités en Europe ou en Australie, et encore moins qu'elles puissent s'intéresser à Unix. Pourtant, Salus énumère les institutions inscrites à la première liste de diffusion électronique, telles que citées par le numéro 1 de *UNIX NEWS* de juillet 1975, et on y trouve déjà l'Université catholique de Louvain et l'Université hébraïque de Jérusalem. N'oublions pas que le premier article consacré à Unix était paru dans les *Communications of the Association for Computing Machinery (CACM)*, l'organe de la principale société savante informatique, en juillet 1974, sous la plume de Thompson et Ritchie, seulement un an auparavant donc.

Accéder au réseau, pour les non-Américains, posait quand même un problème de taille : financer une liaison téléphonique privée transatlantique n'était, et n'est toujours pas, à la portée d'un budget de laboratoire. Ce n'est pas avant la décennie 1980 que les subventions conjuguées de la *National Science Foundation (NSF)* américaine et, par exemple, du ministère français de la Recherche permettront un accès convenable pour l'époque à ce qui était devenu entre-temps l'Internet. Mais dès les années 1970 des groupes Unix quasi militants apparaissaient dans quelques pays : Australie en 1975, Grande-Bretagne en 1976, Pays-Bas en 1978, France en 1981. Unix se propage sur bande magnétique, son usage est recommandé de bouche à oreille, c'est assez analogue au phénomène des radioamateurs dans les années 1960 : tout le plaisir est de réussir à établir une communication avec le Japon ou le Kenya, peu importe après tout ce que l'on se dit, mais le sentiment d'appartenance à la même société d'initiés est d'autant plus fort que les gens sérieux et raisonnables ne comprennent pas.

Ce milieu social d'étudiants en rupture de PhD et d'ingénieurs de centres de calcul dont les responsables ont renoncé à comprendre la teneur exacte de l'activité va assurer le développement d'Unix et de l'Internet, tant les deux sont indissociables. Ce faisant ils vont engendrer une nouvelle entité technique et économique, le logiciel libre. Tout cela sans maîtrise d'ouvrage, sans cahier des charges, sans *business plan*, sans marketing, sans conduite du changement ni plan qualité, ni tout un tas d'autres choses soi-disant indispensables.

Avant d'aborder la question du logiciel libre, il faut s'interroger sur un phénomène quand même surprenant : nous avons dit qu'Unix était très inconfortable pour tout autre qu'un développeur utilisant ses diverses fonctions à longueur de journée. Comment expliquer alors qu'en une dizaine d'années il se soit vu reconnaître une position hégémonique dans tout le monde de la recherche ? Parce que même dans les départements d'informatique des universités et des centres de recherche, la majorité des gens ne passent pas leur temps à programmer, il s'en faut même de beaucoup, alors ne parlons pas des biologistes ou des mathématiciens.

La réponse n'est pas univoque. Mon hypothèse est que si cette population d'étudiants et d'ingénieurs, pauvre en capital social et en légitimité scientifique, a pu se hisser à cette position hégémonique, c'est que la place était à prendre. Pendant les années 1960 et 1970, on assiste aux tentatives des autorités académiques légitimes de l'informatique, dont les porte-drapeaux ont nom Dijkstra, Hoare, Knuth, ou en France Arsac, Ichbiah, Meyer, pour imposer leur discipline comme une science à part entière, égale de la Physique ou de la Mathématique. Pour ce faire ils élaborent des formalisations, des théories, des concepts souvents brillants. Peine perdue, ils échouent, malgré le succès technique et économique retentissant de l'informatique, ou peut-être même victimes de ce succès. Le public, fût-il universitaire, ne discerne pas l'existence d'une science derrière les objets informatiques qui deviennent de plus en plus ses outils de travail quotidiens. Les raisons de cet état de fait restent en grande partie à élucider, sur les traces de chercheurs en histoire de l'informatique tel en France un Pierre-Éric Mounier-Kuhn. Ce désarroi identitaire de l'informatique universitaire snobée par ses collègues laissait le champ libre à des non-mandarins d'autant plus dépourvus de complexes qu'ils n'avaient aucune position à défendre et que le contexte économique d'Unix lui permettait de se développer dans les marges du système, sans gros budgets hormis le coup de pouce initial de la DARPA. Les financements absorbés par Unix et TCP/IP sont assez ridicules si on les compare à ceux de l'intelligence artificielle, sans doute la branche la plus dispendieuse et la plus improductive de la discipline[8], ou même à ceux du langage Ada, projet sur lequel se sont penchées toutes les bonnes fées de la DARPA et du monde académique, et qui finalement n'a jamais percé en dehors des industries militaires et aérospatiales (ce n'est déjà pas si mal, mais les espoirs étaient plus grands).

Finalement, les outsiders unixiens l'ont emporté par leur séduction juvénile et leur occupation du terrain pratique, qui leur a permis de proposer à telle ou telle discipline les outils qui lui manquaient au moment crucial : le système

8. Pour être juste, il faut dire que si l'intelligence artificielle au sens fort du terme a été une telle source de déceptions depuis l'article fondateur de McCulloch et Pitts en 1943 qu'il est difficile de résister à la tentation de parler d'imposture, ses financements somptueux ont permis la réalisation de produits dérivés du plus grand intérêt, comme les langages LISP, les interfaces graphiques à fenêtres, le système d'édition de texte Emacs et bien d'autres, souvent d'ailleurs des logiciels libres.

de composition de documents TEX pour les mathématiciens, qui seul répondait à leurs exigences typographiques, et pour les informaticiens toutes sortes de langages et surtout d'outils pour créer des langages. J'ai vu dans le monde de la biologie Unix supplanter VMS : il faut bien dire que les tarifs pratiqués par Digital Equipment et la rigidité de sa politique de produits lui ont coûté la domination d'un secteur qui n'avait pas beaucoup de raisons de lui être infidèle. Un collègue m'a confié un jour « Le principal argument en faveur d'Unix, c'est que c'est un milieu sympathique ». Cet argument m'avait paru révoltant, mais je crois qu'il avait raison, si l'on prend soin de préciser que par « sympathique » on entend « propice aux libres échanges intellectuels ».

8.5.3 Le schisme

Une si belle unanimité ne pouvait pas durer (et aurait été de mauvais augure). La souche BSD manifestait une certaine indépendance par rapport à l'orthodoxie AT&T. À la section ci-dessus 8.5.2 nous avons laissé d'une part les versions issues de la souche *Bell Labs*, regroupées à partir de 1983 sous l'appellation *System V*, de l'autre celles issues de la souche BSD, qui en 1983 en sont à la version 4.2BSD. De cette époque on peut vraiment dater la séparation de deux écoles rivales. On pourra se reporter au livre de McKusick et ses coauteurs [85] qui donne dans ses pages 5 et 6 un arbre phylogénétique complet du genre Unix.

Quelles étaient les différences entre *System V* et BSD? En fait la seule différence vraiment profonde, perceptible dans l'architecture du noyau, était le code destiné à la communication entre processus (et de ce fait au fonctionnement du réseau), organisé dans les systèmes BSD selon le modèle de *socket* que nous avons évoqué à la section 6.6.1, tandis que les System V utilisaient un autre modèle, moins versatile, baptisé STREAMS. BSD fut aussi en avance pour adopter un système de mémoire virtuelle à demande de pages et un système de fichiers amélioré *(Fast File System)*. Autrement certaines commandes du shell étaient différentes, ainsi que le système d'impression et l'organisation des fichiers de paramètres du système (répertoire `/etc`), etc. La différence était surtout perceptible pour les ingénieurs système et pour les développeurs de logiciels proches du noyau.

Au fur et à mesure qu'Unix se répandait, certains industriels en percevaient l'intérêt commercial et lançaient des gammes de matériels sous Unix. Bill Joy et certains de ses collègues de Berkeley créaient en 1982 Sun Microsystems dont les ordinateurs à base de processeurs Motorola 68000 mettaient en œuvre une version dérivée de BSD, *SunOS*. Chez DEC c'était *Ultrix*. HP-UX de Hewlett-Packard et AIX d'IBM étaient de sensibilité System V. Dès 1980 Microsoft avait lancé Xenix; à ce sujet il convient d'ailleurs de noter qu'à cette époque Bill Gates considérait Unix comme le système d'exploitation de l'avenir pour les micro-ordinateurs! AT&T lançait ses propres microprocesseurs et ses

propres ordinateurs (la gamme 3B) sous Unix, qui n'eurent aucun succès: le démantèlement du monopole d'AT&T sur les télécommunications aux États-Unis au début des années 1980 lui donnait l'autorisation de commercialiser des ordinateurs, mais cela ne suffit pas à assurer le succès de cette gamme de machines...

En fait les différences idéologiques étaient plus tranchées que les différences techniques. Le monde de la recherche et de l'université, ouvert sur les réseaux, penchait pour BSD, issu du même univers, cependant que le monde de l'entreprise avait tendance à se méfier de l'Internet (ou à lui être indifférent) et à se tourner vers la version de la maison-mère, System V.

Il y eut des trahisons sanglantes et impardonnées: en 1992, Sun, porte-drapeau du monde BSD avec SunOS 4.1.3, à l'époque le meilleur Unix de l'avis de la communauté des développeurs, conclut avec AT&T des accords d'ailleurs sans lendemain aux termes desquels il passait à System V sous le nom de *Solaris*, honni par les puristes BSD.

Ce qui est certain, c'est que ces luttes de clans et ce culte de la petite différence ont beaucoup nui à la diffusion d'Unix et beaucoup contribué au succès des systèmes Windows de Microsoft. La communauté des développeurs a également déployé tous ses efforts pour combattre toute tentative de développer au-dessus d'Unix et du système de fenêtrage X une couche d'interface graphique « à la Macintosh », qui aurait rendu le système utilisable par des non-professionnels. De tels systèmes apparaissent aujourd'hui (Gnome, KDE), alors que la bataille a été gagnée (au moins provisoirement) par Windows.

On peut aujourd'hui considérer que le schisme BSD-System V est résorbé dans l'œcuménisme: tous les System V ont des *sockets* (pour faire du TCP/IP il faut bien) et tous les BSD ont le système de communication inter-processus (IPC) STREAMS de System V, notamment. Mais l'idéologie BSD reste toujours vivace.

8.6 Aux sources du logiciel libre

8.6.1 Principes

Le logiciel libre mobilise beaucoup les esprits en ce début de millénaire. La définition même de la chose suscite de nombreuses controverses dues en partie au fait que le mot anglais *free* signifie à la fois libre et gratuit. Si l'on s'en tient à une acception restrictive, l'expression logiciel libre désigne un modèle économique et un mouvement associatif créés en 1984 par un informaticien de génie, Richard M. Stallman, auteur depuis 1975 d'un logiciel extraordinaire, Emacs. En 1983, Stallman, qui travaillait au laboratoire d'intelligence artificielle du MIT, excédé par les restrictions au développement d'Unix induites par les

droits de propriété industrielle d'AT&T et de quelques autres industriels [9], fonde le projet *GNU ("GNU is Not Unix")* destiné à créer un système d'exploitation libre de droits et dont le texte source serait librement et irrévocablement disponible à tout un chacun pour l'utiliser ou le modifier.

L'année suivante, Stallman crée la *Free Software Foundation (FSF)* pour « promouvoir le droit de l'utilisateur d'ordinateur à utiliser, étudier, copier, modifier et redistribuer les programmes d'ordinateur », c'est à dire étendre le modèle du projet *GNU* à d'autres logiciels. Un corollaire indissociable de ce principe est que le texte source du logiciel libre doit être accessible à l'utilisateur, ainsi que la documentation qui permet de l'utiliser. Cette clause confère à tout un chacun le moyen de modifier le logiciel ou d'en extraire tout ou partie pour une création nouvelle. Pour assurer la pérennité du principe, tout logiciel libre conforme aux idées de la FSF est soumis aux termes d'une licence, la *GNU GPL (GNU General Public License)*, qui impose les mêmes termes à tout logiciel dérivé. Ainsi il n'est pas possible de détourner du logiciel libre pour créer du logiciel non libre sans violer la licence.

8.6.2 Préhistoire

Avant d'examiner plus avant les tenants et les aboutissants de ces principes et de ce modèle économique, il convient de signaler que jusqu'en 1972 le logiciel, s'il n'était pas libre au sens de la GPL, était pratiquement toujours disponible gratuitement et très souvent sous forme de texte source, et ce parce que jusqu'alors la conscience du coût et de la valeur propre du logiciel était dans les limbes. IBM, qui avait fini par accaparer 90% du marché mondial de l'informatique, distribuait systèmes d'exploitation et logiciels d'usage général à titre de « fournitures annexes » avec les ordinateurs.

Peu après l'annonce de la gamme IBM 360 en 1964, RCA annonça les ordinateurs *Spectra 70* dont l'architecture était conçue afin de pouvoir accueillir le logiciel développé pour les IBM 360, y compris le système d'exploitation. Cette ambition ne se réalisa pas, notamment parce que les ingénieurs de RCA n'avaient pu se retenir d'ajouter à leur système des « améliorations » qui en détruisaient la compatibilité, mais IBM avait perçu la menace et commença à élaborer une parade juridique qui consistait à séparer la facturation du logiciel de celle du matériel : ce fut la politique d'*unbundling*, annoncée en 1969, mais dont l'application à ce moment fut entamée assez mollement.

9. En fait l'ire fondatrice qui décida de la vocation prophétique du Maître était dirigée contre Xerox et son pilote d'imprimante... *Felix culpa.* Le lecteur remarquera que malgré leurs péchés AT&T et Xerox jouent un rôle capital dans l'histoire de l'informatique en y apportant plus d'innovations que bien des industriels du secteur.

Au début des années 1970, quelques industriels (notamment Telex, Memorex et Calcomp) commencèrent à vouloir profiter de la manne et pour cela vendre des matériels compatibles avec ceux d'IBM, c'est à dire tels que les clients pourraient les acheter et les utiliser en lieu et place des matériels IBM originaux. IBM riposta à ce qu'il considérait comme une concurrence déloyale en cessant de divulguer le code source des parties de système d'exploitation qui permettaient la conception et le fonctionnement des systèmes concurrents. Il en résulta des procès acharnés, et en 1972 un arrêt de la Cour suprême des États-Unis, au nom de la législation anti-trust créée pour limiter l'emprise de Rockfeller, statua dans le procès Telex–IBM et imposa à IBM de facturer à part logiciel et matériel. Ceci précipita l'entrée en vigueur de la politique d'*unbundling*. Les observateurs de l'époque déclarèrent que cela n'aurait aucun effet, mais deux industries étaient nées : celle du matériel compatible IBM, qui serait florissante une vingtaine d'années, et celle du logiciel, dont Microsoft est le plus beau fleuron.

8.6.3 Précurseurs

Si l'Église chrétienne a reconnu à Jérémie, Isaïe, Daniel et Ézéchiel la qualité de précurseurs de la vraie foi et de la venue du Sauveur, Richard Stallman est plus intansigeant, mais n'en a pas moins lui aussi des précurseurs. Ils se situent dans les limbes, à l'époque où *ARPANET* engendrait TCP/IP, qui était bien évidemment du logiciel, et qui était distribué aux membres du réseau, c'est à dire, nous l'avons vu, potentiellement à toutes les universités et tous les centres de recherche. Comme tout cela se passait à Berkeley, il en résulta que le système de prédilection de TCP/IP et, partant, de l'Internet fut Unix, et que traditionnellement les principaux logiciels de réseau sous Unix furent distribués gratuitement, du moins aux centres de recherche, sous les termes d'une licence dite « BSD » qui garantissait les droits de propriété intellectuelle des « Régents de l'Université de Californie ». Les éléments de base du protocole TCP/IP proprement dit font partie du noyau Unix BSD, d'où ils ont assez vite été adaptés aux noyaux des autres Unix, cependant que les logiciels d'application qui en permettaient l'usage, tels que *Sendmail* pour envoyer du courrier électronique, *Ftp* pour transférer des fichiers, *Telnet* pour se connecter à un système distant, etc., étaient distribués indépendamment. Plus tard *Bind* pour la gestion du service de noms de domaines, *INN* pour le service de *News* et beaucoup d'autres logiciels s'ajouteront à cette liste, toujours gratuitement.

Par ailleurs, Unix devenait populaire dans le monde de la recherche et les logiciels développés par des scientifiques étaient aussi rendus disponibles dans des conditions analogues : TEX pour la composition typographique, *Blast* pour la comparaison de séquences biologiques, *Phylip* pour l'analyse phylogénétique, pour ne citer que trois exemples parmi une foule, sont disponibles selon les termes de licences assez variées (ou d'absence de licence…), mais toujours sans

versement de redevances. Bref, le monde de la recherche fait et utilise du logiciel libre depuis longtemps sans forcément le dire ni même en avoir conscience.

8.6.4 Économie du logiciel

Concurrence monopolistique

L'économie du logiciel, étudiée notamment avec brio par Michel Volle dans son livre *iconomie* [136], exprime un paradoxe dont la conscience ne s'est manifestée que récemment. Citons Michel Volle dans la présentation de son livre : « Le "système technique contemporain" est fondé sur la synergie entre microélectronique, informatique et automatisation. On peut styliser sa fonction de production en supposant qu'elle est "à coûts fixes" : le coût de production, indépendant du volume produit, est payé dès l'investissement initial. Développons cette hypothèse : les usines étant des automates, l'emploi réside dans la conception et la distribution. La distribution des revenus n'est pas reliée au salariat. Les entreprises différencient leur production pour construire des niches de monopole. Elles organisent leurs processus autour du système d'information. Le commerce passe par des médiations empruntant la communication électronique.

« L'investissement est risqué, la concurrence est mondiale et violente. On retrouve dans cette présentation stylisée des aspects tendanciels de notre économie. Elle éclaire la description des secteurs de l'informatique, de l'audiovisuel, des réseaux (télécommunications, transport aérien, etc.), du commerce, ainsi que les aspects stratégiques et tactiques des jeux de concurrence dans ces secteurs et dans ceux qui les utilisent. On voit alors que cette économie hautement efficace pourrait aller au désastre si elle était traitée sur le mode du "laissez faire", sans considérer les exigences de l'éthique et de la cohésion sociale. » (texte disponible sur le site `http://www.volle.com` selon les termes de la *GNU Free Documentation License*).

Michel Volle nous explique que ces rapports de production créent une situation de *concurrence monopolistique* telle que l'entreprise qui parvient à dépasser ses concurrents en tire un tel avantage qu'elle les anéantit : puisque les rendements sont croissants, le plus gros est le plus rentable. Microsoft pour les systèmes d'exploitation, Oracle pour les bases de données, Intel pour les processeurs, Google pour les moteurs de recherche en sont des exemples.

La seule façon pour une entreprise de se développer à côté d'un de ces géants est de créer un nouveau marché avec des produits différents qui attirent une autre clientèle, ce qu'a su faire avec succès Apple, en anéantissant au passage Nokia en quelques mois par le lancement de l'iPhone en 2007.

Une autre issue est d'adopter un modèle économique complètement différent, comme celui du logiciel libre dont GNU/Linux est un bon exemple.

Économie du système d'exploitation

Voyons ce qu'il en est pour le système d'exploitation. Un système d'exploitation commercial moderne tel que Windows comporte quelques 40 millions de lignes de programme. Estimer le nombre de lignes de Linux est assez difficile parce qu'il faut ajouter à la taille du noyau celle des multiples utilitaires et logiciels généraux qui le complètent. La version 4.15 (architecture Intel x86-64) utilisée pour rédiger le présent ouvrage, en comptant les pilotes de périphériques et les modules, compte 20,3 millions de lignes, auxquelles il faudrait en ajouter presqu'autant pour le système de fenêtrage X, sans oublier les bibliothèques de commandes et de fonctions diverses issues des projets GNU et BSD pour la plupart, mais dont il conviendrait de retirer quelques dizaines de milliers de lignes pour les parties redondantes liées par exemple à des architectures différentes (Intel x86, ARM, PowerPC, etc.).

Le rendement moyen d'un développeur est hautement sujet à controverses, mais si l'on compte les temps consacrés à la rédaction de la documentation et à l'acquisition de connaissances, 50 lignes par jour de travail semble un maximum, soit 10 000 par an et par personne. Mais comme le signale Brooks dans son livre *Le mythe de l'homme-mois* [24], une telle valeur est éminemment trompeuse. La création d'un logiciel est une tâche très complexe, et la division du travail la complique encore en multipliant les fonctions de direction, de coordination et d'échanges d'information, qui peuvent aboutir à ralentir le processus plus qu'à l'accélérer. Manuel Serrano en a tiré les conséquences dans sa thèse d'habilitation [120] en plaidant pour le « logiciel moyen » : les grands logiciels ne seraient devenus encombrants, dans bien des cas, que par la prolifération incontrôlée d'un processus de développement devenu bureaucratique. Une réflexion plus intense d'un groupe plus petit et plus conscient des objectifs à atteindre permettrait d'obtenir un logiciel plus petit et de meilleure qualité.

La création d'un logiciel important tel qu'un système d'exploitation est une tâche colossale qui peut mobiliser des centaines de personnes pendant une décennie, le coût marginal du produit final livré dans un grand magasin est pratiquement nul, le prix payé par le client est essentiellement celui de la boîte en carton, des frais de transport et de gestion et de l'effort commercial. Il en résulte, dans l'industrie du logiciel, une tendance irréversible au monopole : dans une industrie à coût marginal de production nul, dès que vous vendez plus que les autres, vous êtes très vite beaucoup plus riche, avec les conséquences qui en découlent. Dès lors qu'un marché prend forme et s'organise, il émerge un fournisseur unique : c'est la situation de concurrence monopolistique que nous avons mentionnée ci-dessus.

Comme nous l'avons signalé ci-dessus, l'espoir de diversité, dans un tel contexte industriel, ne peut venir que de l'apparition de nouveaux segments de marchés, desservis par de nouvelles technologies, rôle joué dans le passé

par les mini-ordinateurs, puis les micro-ordinateurs à base de microprocesseur, innovations technologiques qui réduisaient de plusieurs ordres de grandeur les coûts de production. Ou du recours à un autre modèle économique.

8.6.5 Modèle du logiciel libre

Le logiciel libre, face à cette situation, représente un potentiel très dynamique, parce qu'il obéit à un modèle économique tout autre. Microsoft ne peut utiliser contre lui aucune des armes classiques de la concurrence industrielle, telles que la guerre des prix, la publicité, les fournitures associées, l'effet de gamme, etc., parce que le logiciel libre n'est sur aucun de ces terrains.

Les caractères économiques du logiciel libre ont été étudiés, entre autres, par Marie Coris dans son travail de thèse de doctorat à l'Université Montesquieu de Bordeaux IV (voir sa communication au congrès JRES 2001 : [34]). Elle énumère d'abord les caractères du logiciel en général :

- un bien d'information, aspect amplement développé ici dont découle l'importance des économies d'échelle ;
- un bien *en réseau* : son utilité croît en raison du nombre de ses utilisateurs ;
- un bien à cheval entre public et privé :
  - le coût de production pratiquement engagé en totalité dès le premier exemplaire, l'usage non destructif (il peut être utilisé par un nombre infini d'utilisateurs), l'usage non exclusif (il est difficile d'empêcher quelqu'un d'autre de l'utiliser) sont des caractéristiques d'un bien public,
  - le recours à la protection du droit d'auteur ou du brevet permet d'annuler les aspects « publics », par exemple en limitant la reproductibilité, et de faire du logiciel un bien privé.

L'alternative se situe entre le logiciel comme bien privé, idée des entreprises telles que Microsoft, Oracle, etc., et le logiciel comme bien public, idée du logiciel libre.

Volle, Coris et d'autres ont montré que le marché d'un bien d'information ne peut prendre que deux formes :

- si les instances de ce bien sont suffisamment différenciées, plusieurs fournisseurs peuvent coexister dans des niches du marché ;
- dès qu'un fournisseur réussit à prendre sur ses concurrents un avantage significatif en déniant leur différence, il obtient une situation de monopole du fait des économies d'échelle considérables procurées par le volume des ventes.

Le logiciel libre échappe à cette alternative parce qu'il se situe hors de la logique marchande, et que la rétribution de ses auteurs relève des domaines symbolique et moral. Michel Volle a fait remarquer qu'un auteur de logiciel libre aurait aussi un accès facilité au capital–risque le jour où il voudrait créer une entreprise du fait de la reconnaissance acquise dans le domaine non marchand.

La *GNU GPL* définit parfaitement les « quatre libertés » caractéristiques du logiciel libre : liberté d'utilisation, liberté de copie, liberté de modification et liberté de redistribution. Elle autorise donc la modification et la redistribution, mais en imposant que le logiciel reste sous GPL, et ce également dans le cas de l'incorporation d'un logiciel libre à un nouveau logiciel : le caractère « libre » est héréditaire et contagieux. Dans ce dispositif, le statut du code source détermine la nature publique du bien, plus qu'il ne sert vraiment à la maintenance par l'utilisateur. La publicité du code interdit l'appropriation privée. Mais plonger dans les sources pour y introduire des modifications est une entreprise à n'envisager qu'avec circonspection ; cela risque de coûter fort cher.

Reste à se poser une question : le logiciel libre, comme le logiciel non libre, est écrit par des hommes et des femmes qui y consacrent tout ou partie de leur vie professionnelle et qui ne vivent pas de l'air du temps. Qui finance la production de logiciels libres, et comment, puisque, quoique ses apôtres s'en défendent, sa caractéristique principale est bien qu'il est possible de l'utiliser sans bourse délier ?

Bertrand Meyer, dans un article assez polémique de critique du libre [88], dresse une nomenclature des sources de financement possibles, qui énerve les adeptes mais à laquelle il est difficile de dénier toute véracité :

1. une donation : le développeur vit de sa fortune personnelle ou développe pendant ses nuits et ses jours de congé ;
2. le financement public : le logiciel a été créé par un centre de recherche, une université ou une autre entreprise publique ;
3. le financement privé : une entreprise décide de distribuer un logiciel développé à ses frais selon le modèle libre ;
4. la subvention (publique ou privée) : le développeur crée un logiciel en utilisant son temps de travail et les ressources de son employeur, public ou privé, sans que celui-ci lui ait confié cette tâche.

Le cas 4 est celui qui provoque parfois quelque agacement, et on ne peut exclure qu'il soit assez répandu. Cela dit, un examen de ce cas informé par les tendances les plus récentes de la sociologie du travail montre que cette situation n'est pas forcément scandaleuse, et que l'initiative prise par le développeur peut comporter des avantages pour son employeur même s'il n'en avait pas initialement conscience. La création d'Unix en est le plus bel exemple, et si l'on regarde en arrière, on peut se dire qu'AT&T en aurait sans doute tiré encore plus

d'avantages en en faisant un logiciel libre ; Unix ne lui a pas vraiment rapporté beaucoup d'argent, et sa facturation aux entreprises a considérablement restreint sa diffusion. Le succès actuel de Linux apporte *ex post* des arguments à l'appui de cette hypothèse. Toutefois, Bertrand Meyer a raison d'écrire que sans le couple Intel–Microsoft il n'y aurait jamais eu de PC à 300 Euros, et partant jamais de succès pour Linux.

Un exemple typique et très instructif du cas 3 fut celui du logiciel Ghostscript, produit par la société Aladdin Enterprises (aujourd'hui disparue), qui est un interpréteur du langage PostScript. PostScript est un langage de description de pages utilisé comme format de sortie par de nombreux logiciels et comme format d'entrée par beaucoup d'imprimantes. Ghostscript est utile pour afficher à l'écran le contenu d'un fichier PostScript, et pour l'imprimer sur une imprimante dépourvue d'interpréteur PostScript incorporé, ce qui est le cas notamment de beaucoup de petites imprimantes à jet d'encre. Ce logiciel a deux groupes bien distincts d'utilisateurs : des millions de propriétaires de petites imprimantes bon marché qui veulent afficher et imprimer du PostScript, et une dizaine d'industriels fabricants d'imprimantes, qui incorporent Ghostscript par paquets de cent mille exemplaires à leurs productions.

Dans sa grande sagesse, la société Aladdin Enterprises avait décidé de ne pas se lancer dans la commercialisation à grande échelle d'un logiciel qui vaudrait quelques dizaines d'Euros, et de le distribuer aux particuliers sous les termes d'une licence dite *Aladdin Ghostscript Public License*, qui protégeait la propriété intellectuelle d'Aladdin et permettait un usage gratuit. Depuis 2000, Ghostscript est un logiciel libre disponible sous les termes de la *GNU GPL*. Aladdin Enterprises tirait plutôt ses revenus de la clientèle des fabricants d'imprimantes.

Le cas 2 est sans doute le plus fréquent. La justification initiale de ce modèle me semble remonter au principe constant de l'administration américaine : ce qui a été payé une fois par le contribuable ne doit pas l'être une seconde fois. Les logiciels dont le développement a été financé par des contrats de recherche de la DARPA doivent être mis gratuitement à la disposition du public, au même titre que les photos de l'espace prises par la NASA.

Ce principe ne semble pas scandaleux : ce qui a été financé par l'argent public [10] (au sens large) revient au public. Les résultats de la recherche publique sont disponibles publiquement. Il s'agit d'un système de redistribution : tous les

10. Ramenons ici à de justes proportions la vision que l'on a souvent en France du système universitaire américain, dont les universités seraient autant d'entreprises privées gérées comme des sociétés commerciales et dont les étudiants seraient purement et simplement des clients. Cette vision est simpliste : ainsi l'Université de Californie, organisée autour de huit campus répartis sur le territoire de l'État, reçoit 80% de ses financements de l'État de Californie, de l'État fédéral, des municipalités et d'agences fédérales. Les étudiants californiens y sont admis gratuitement sur des critères de dossier scolaire. Du moins en était-il ainsi encore récemment, les dernières nouvelles

contribuables financent les logiciels produits de cette façon, et les bénéficiaires en sont les utilisateurs. Il est légitime de s'interroger sur l'équité de ce processus de répartition, mais il en est sans doute de pires.

Qui pourrait s'estimer lésé? Essentiellement les entreprises dont la prospérité repose sur le logiciel commercial, et qui pourraient arguer d'une concurrence déloyale, puisque le plus souvent alimentée par des financements publics. Curieusement, on observe peu de protestations de cette nature, et encore moins de procès.

Il convient aussi de remarquer que le modèle du logiciel libre, s'il n'apporte apparemment que des avantages à l'utilisateur, peut comporter des inconvénients pour l'auteur, qui s'interdit en fait tout contrôle exclusif sur la divulgation de son travail. Certes, les clauses de la *GNU GPL* permettent la commercialisation de logiciel libre, et il est parfaitement possible de recourir au système de la double licence, par exemple *GNU GPL* pour le monde académique et licence commerciale pour le monde industriel. Mais il est clair qu'un logiciel novateur dont l'auteur peut espérer des revenus importants sera mal protégé des contrefaçons si son code source est divulgué. En fait, dans le monde académique la pression idéologique pour la *GNU GPL* est très forte, et les auteurs de logiciels qui souhaitent vivre des fruits de leur activité de développeur plutôt que d'un emploi universitaire (ou qui, faute d'un tel emploi, n'ont pas le choix) sont assez marginalisés par ce système. Le caractère contagieux et contraignant de la *GNU GPL* est très pénalisant pour l'auteur qui ne souhaiterait pas vivre dans l'abnégation (c'est le terme exact : le logiciel qu'il a écrit ne lui appartient plus), ou qui, faute d'avoir obtenu un poste dans un organisme public, ne le pourrait pas. Il y a des exemples d'auteurs qui pour avoir refusé les servitudes de la *GNU GPL* se sont vu mettre au ban de la communauté, leurs travaux passés sous silence et leurs logiciels exclus des serveurs publics.

En fait, la réalité usuelle du développeur de logiciel libre est qu'il gagne sa vie autrement, et que la rétribution qu'il attend pour son œuvre est la reconnaissance de ses pairs. Quiconque bénéficie du logiciel libre ressent le désir d'y contribuer et ainsi d'adhérer à une communauté perçue comme éthique. Il est souhaitable que la *GNU GPL* ne reste pas hégémonique et que d'autres licences aux termes moins idéologiques et plus équilibrés apparaissent dans le monde du logiciel libre.

---

laissent craindre une dégradation de ce système généreux. D'autres universités ont sans doute des modes de fonctionnement plus éloignés du service public français, bref, les situations sont variées et il n'est pas certain que les formations universitaires soient plus réservées aux catégories privilégiées là-bas qu'ici, même si les barrières ne sont pas disposées de la même façon.

8.6.6 Une autre façon de faire du logiciel

Le modèle du logiciel libre n'est pas sans influence sur la nature même du logiciel produit. En effet, dans ce contexte, un auteur peut facilement utiliser d'autres logiciels s'ils sont libres, que ce soit pour recourir à des bibliothèques de fonctions générales ou pour s'inspirer d'un logiciel aux fonctions analogues mais dans un environnement différent.

Des systèmes de développement coopératif se mettent en place par le réseau, qui seraient impensables pour du logiciel non-libre : les programmes sous forme source sont accessibles sur un site public, et chacun peut soumettre sa contribution. L'archétype de ce mode de développement est celui du noyau Linux proprement dit, coordonné par Linus Torvalds personnellement.

Pour qu'un tel procédé donne des résultats utilisables, il faut que le logiciel présente une architecture qui s'y prête, notamment une grande modularité, afin que chaque contributeur puisse travailler relativement indépendamment sur telle ou telle partie. Par exemple, dans le noyau Linux, tout ce qui permet le fonctionnement de machines multi-processeurs et la préemption des processus en mode noyau (voir section 3.12.4) demande une synchronisation beaucoup plus fine des fils d'exécution : les adaptations nécessaires ont été réalisées par Robert Love, ce qui a été possible parce qu'il n'était pas trop difficile d'isoler les parties du code concernées. À l'inverse, lorsque Netscape a voulu donner un statut Open Source à une partie du code de son navigateur connue sous le nom Mozilla, l'opération a été rendue difficile parce que le code initial n'avait pas été réalisé selon un plan suffisamment modulaire.

Finalement, la réutilisation de composants logiciels, dont plusieurs industriels parlent beaucoup depuis des années sans grand résultat, sera sans doute réalisée plutôt par les adeptes de l'Open Source. En effet, l'achat d'un tel composant est un investissement problématique, tandis que le récupérer sur le réseau, l'essayer, le jeter s'il ne convient pas, l'adopter s'il semble prometteur, c'est la démarche quotidienne du développeur libre. On pourra lire à ce sujet l'article de Josh Lerner et Jean Tirole, *The Simple Economics of Open Source* [77].

L'analyse détaillée des conséquences d'un tel mode de construction de logiciel reste à faire, mais en tout cas il ne fait aucun doute que le résultat sera très différent. Rappelons les étapes classiques de la construction d'un système informatique pour un client selon le mode projet :

- expression des besoins ;
- cadrage, opportunité, faisabilité ;
- spécification ;
- réalisation ;
- recette...

Oublions tout cela dans le monde du libre. Le logiciel commence à prendre forme autour d'un noyau, noyau de code et noyau humain, généralement une seule personne ou un tout petit groupe. L'impulsion initiale procède le plus souvent du désir personnel des auteurs de disposer du logiciel en question, soit qu'il n'existe pas, soit que les logiciels existants ne leur conviennent pas, que ce soit à cause de leur prix ou de leur environnement d'exécution. Puis d'autres contributions viennent s'ajouter, presque par accrétion. Un coordonnateur émerge, souvent l'auteur initial, il assure la cohérence de l'ensemble. Quand des divergences de vue surgissent, il peut y avoir une scission : ainsi deux versions sont disponibles pour Emacs, Gnu Emacs et Xemacs, toutes deux libres.

Le catalogue du logiciel libre est assez vaste. Tout d'abord le logiciel libre avant la lettre :

- TEX, LATEX, respectivement de Donald Knuth et de Leslie Lamport, avec lesquels est réalisé le présent document ;
- les logiciels réseau :
  - *Sendmail*, pour envoyer du courrier ;
  - *Bind*, pour résoudre les noms de domaine,
  - beaucoup d'autres, ce sont eux qui « font marcher » l'Internet ;
- *X Window System*, créé par un consortium qui unissait IBM, DEC et le MIT ;
- des quantités de programmes scientifiques : l'essentiel de la biologie moléculaire se fait avec du logiciel libre.

Et pour le logiciel libre canonique, de stricte obédience :

- *Gnu Emacs*, un éditeur de texte, *GCC*, un compilateur C, *The Gimp*, un concurrent libre de Photoshop, *GNAT*, un environnement de programmation ADA ;
- *Linux, FreeBSD, OpenBSD, NetBSD*, des systèmes d'exploita-tion Unix ;
- *PostgreSQL*, *MariaDB* et *MySQL*, des systèmes de gestion de bases de données relationnelles ;
- *Apache*, un serveur Web.

Chacun dans son domaine, ces logiciels sont de toute première qualité, et il n'y a pas à hésiter : ce sont eux qu'il faut utiliser ! Il manque bien sûr des choses, comme un logiciel OCR comparable à *Omnipage* de *Caere*...

8.6.7 Linux

Linux est indubitablement un système emblématique du logiciel libre. S'il y a d'autres systèmes d'exploitation libres, y compris dans la famille Unix, et si

les logiciels libres ne sont pas tous destinés à Unix, l'association avec Linux est inévitable.

Le début de ce chapitre montre la filiation entre le mouvement autour d'Unix et le développement du logiciel libre, mais le paradoxe était qu'Unix lui-même n'était pas libre, AT&T en détenait les droits. Pourtant la communauté Unix était très attachée à l'idée de l'accès au code source du système, puisque le développement de nouveaux logiciels, certains très proches du système, ne pouvait être entrepris qu'avec un tel accès. En outre les droits d'AT&T semblaient contestables, parce qu'Unix avait recueilli beaucoup de contributions extérieures souvent non rémunérées. Plusieurs moyens ont été envisagés pour briser ou contourner ce paradoxe.

Le premier moyen consistait à obtenir d'AT&T une licence source. Si pour des chercheurs ou des universitaires c'était théoriquement possible, pratiquement des obstacles surgissaient. Si l'on était loin des États-Unis et que l'on n'avait pas de relations dans ce milieu, nouer les contacts nécessaires pouvait demander des mois. Une fois la licence source obtenue, il fallait obtenir du fournisseur de son ordinateur les sources de la version de système précise installée sur la machine, ce à quoi il ne se prêtait pas toujours avec bonne volonté. Bref, le seul moyen de pouvoir accéder réellement aux sources d'un système opérationnel était d'appartenir à un des groupes en vue du monde Unix. Rappelons qu'au début des années 1980 une machine capable de « tourner » Unix et le réseau coûtait au minimum l'équivalent de 100 000 Euros et que l'Internet n'atteignait pas l'Europe.

Ces difficultés étaient particulièrement ressenties par les enseignants qui voulaient initier leurs étudiants à Unix, ce qui était bien sûr impossible sans accès au code source. Ils ont très tôt imaginé des moyens de contourner les droits d'AT&T. Le précurseur quasi légendaire de cette démarche fut un professeur australien, John Lions, dont le livre de 1977 *A Commentary on the Unix System, V6* [81] comportait le code du noyau. AT&T n'avait autorisé qu'un tirage limité, mais comme en URSS du temps de Brejnev apparut une véritable activité clandestine de *Samizdat*. Ce livre fut réédité en 1996, et il mérite toujours d'être lu. John Lions aura vu la publication légale de son œuvre avant sa mort en décembre 1998.

Andrew Tanenbaum, professeur à l'Université Libre d'Amsterdam *(Vrije Universiteit Amsterdam)*, rencontra le même problème. Pour les besoins de son enseignement il créa de toutes pièces un système miniature inspiré d'Unix, adapté aux micro-ordinateurs à processeur Intel et baptisé Minix. Le cours donna naissance à un livre [129] dont les premières versions (1987) comportaient en annexe le code de Minix. Il était également possible d'acheter les disquettes pour installer Minix sur son ordinateur, mais ce n'était ni très bon marché ni très commode.

Pendant ce temps le groupe BSD travaillait à expurger son code de toute instruction rédigée par AT&T, de façon à l'affranchir de tout droit restrictif. Le résul-

tat fut 4.3BSD Net1 en 1989, puis 4.3BSD Net2 en 1991. L'objectif était de produire 4.4BSD-Lite en 1993, mais USL (Unix System Laboratories, une branche d'AT&T qui était à l'époque propriétaire des droits Unix) attaqua ce projet en justice, ce qui en différa la réalisation d'un an. Du groupe BSD émanèrent aussi un système Unix pour processeurs Intel en 1992, 386BSD, et une société destinée à le commercialiser, BSD Inc. Mais tous ces efforts furent retardés par des questions juridiques. Ils débouchèrent, sensiblement plus tard, sur les Unix libres FreeBSD, OpenBSD et NetBSD.

Rappelons que depuis 1983 le projet GNU visait lui aussi à produire un système similaire à Unix mais libre de droits et disponible sous forme source. Au début des années 1990 ce groupe avait réalisé un certain nombre de logiciels libres utilisables sur les divers Unix du marché, notamment des utilitaires, mais toujours pas de noyau. En fait ce n'est qu'en 2000 que le système *Hurd*[11] destiné à remplacer le noyau Unix et basé sur le micro-noyau *Mach* créé à l'Université Carnegie-Mellon commença à ressembler à un vrai système.

Le salut allait venir d'ailleurs. En 1991 un étudiant de l'Université d'Helsinki, Linus Torvalds, achète un ordinateur doté d'un processeur Intel 386 et cherche un moyen d'explorer son fonctionnement. Minix lui semble une voie prometteuse, dans laquelle il s'engage, mais il y rencontre quelques obstacles et commence à réécrire certaines parties du noyau. En août 1991 Linux est né, en septembre la version 0.01 est publiée sur le réseau. Elle ne peut fonctionner que sous Minix. La version 0.02 publiée le 5 octobre 1991 permet l'usage du shell `bash` et du compilateur C `gcc`, deux logiciels GNU. La première version réputée stable sera la 1.0 de mars 1994.

Pour résumer la nature de Linux, nous pourrions dire que c'est un noyau, issu à l'origine de celui de Minix, et dont le développement est toujours assuré sous la supervision personnelle de Linus Torvalds, entouré des outils GNU, le tout sous licence GPL. Ce que Linus Torvalds a fait, d'autres auraient peut-être pu le faire eux-mêmes, et ils regrettent sans doute d'en avoir laissé l'occasion, à un Européen de surcroît, mais l'histoire est ainsi.

Linux est au départ plutôt un Unix System V, mais doté de toutes les extensions BSD souhaitables, ainsi que des dispositifs nécessaires à la conformité POSIX[12]. Sa principale originalité tient sans doute à son processus de développement : alors que tous les autres systèmes d'exploitation cités dans

11. *Hurd* est fondé sur un micro-noyau, ce n'est donc pas un noyau. Voir chapitre 10.

12. POSIX *(Portable Operating System Interface)* est une norme de l'IEEE qui définit une API *(Application Program Interface)* et une interface de commande pour les systèmes d'exploitation, assez inspirés d'Unix. La première version de POSIX a été rédigée en 1986 par un groupe de travail auquel appartenait notamment Richard M. Stallman. Cet effort de normalisation fut au départ assez mal reçu par une communauté Unix plutôt libertaire : je me rappelle une conférence Usenix à la fin des années 1980 où certains participants arboraient un badge « *Aspirin, Condom, POSIX* ». Mais il est certain que POSIX fut salutaire.

ce chapitre ont été développés par des équipes organisées, le développement du noyau Linux s'est fait depuis le début par « appel au peuple » sur l'Internet. Quiconque s'en sent la vocation peut participer aux forums, lire le code et proposer ses modifications (appelées *patches*). Elles seront examinées par la communauté, et après ce débat Linus Torvalds tranchera et décidera de l'incorporation éventuelle au noyau officiel. Il est difficile d'estimer les effectifs d'une telle communauté informelle, mais le nombre de contributeurs actifs au noyau Linux est sans doute inférieur à 200 (nombre de développeurs recensés pour la version 2.0). Le succès de Linux résulte sans doute en partie de facteurs impondérables : pourquoi l'appel initial de Torvalds a-t-il séduit d'emblée ? La réponse à cette question est sûrement complexe, mais en tout cas le succès est incontestable. Ce qui est sûr, c'est que l'appel à contribution d'août 1991 répondait à une attente et à une frustration largement répandues.

Comme nous pouvons nous y attendre, Linux connaît aussi la division en chapelles. Ici elles s'appellent « distributions ». Au début, la diffusion des éléments de Linux s'effectuait par le réseau, mais assez vite cela devint volumineux et compliqué. Il fallait transférer le noyau lui-même (en code source), ainsi que les logiciels (surtout GNU) sans lequel il aurait été inutilisable, en veillant à la synchronisation des versions compatibles. Au bout de quelques années apparurent des éditeurs qui distribuaient pour un prix modique des CD-ROMs comportant tout ce qu'il fallait pour démarrer.

Par exemple en 1996 on pouvait acheter pour l'équivalent de moins de 30 Euros (somme compatible avec la notion de logiciel libre, puisqu'elle ne rémunérait que la copie, l'emballage et la distribution, pas le logiciel lui-même) le jeu de CD Infomagic, qui comportait notamment la distribution *Slackware*. La Slackware était une distribution assez ascétique : les différents logiciels étaient simplement fournis sous forme d'archives compressées qu'il fallait compiler, en bénéficiant quand même du logiciel de gestion de configuration *make*.

D'autres distributions proposent des paquetages : il s'agit de programmes tout compilés. Rassurez-vous, les principes du logiciel libre sont respectés, le paquetage source est fourni à côté. Les distributions RedHat et Debian ont chacune leur format de paquetage, leur logiciel d'installation qui en outre contrôle les dépendances entre paquetages, l'installation et la mise à jour par le réseau, etc. Il faut bien reconnaître que c'est assez pratique. Mais pour le débutant qui se lance dans l'installation de Linux il est néanmoins conseillé d'avoir à proximité un ami qui est déjà passé par là !

Chapitre 9 Au-delà du modèle de von Neumann

Sommaire

Introduction

La recherche de performances a inspiré des tentatives pour contourner la loi d'airain de l'architecture de von Neumann : une seule instruction à la fois. Ces tentatives se classent en deux catégories : radicales et pragmatiques. Le présent chapitre en fait un bref tour d'horizon. Les architectures qui remettent fortement en cause le principe de von Neumann n'existent aujourd'hui que dans quelques laboratoires (même si elles sont sans doute promises à un avenir meilleur et si certaines d'entre elles ont un passé respectable), et celles qui sont des extensions pratiques de ce principe ne remettent pas en cause la sémantique du modèle de calcul, même si elles en compliquent considérablement la mise en œuvre technique pour le bénéfice d'une amélioration non moins considérable des performances.

9.1 Architectures révolutionnaires

Les tentatives radicales visent à créer de nouveaux modèles de calcul en imaginant des ordinateurs capables d'exécuter un grand nombre d'instructions simultanément. Ces modèles révolutionnaires se classent selon diverses catégories :

9.1.1 SIMD *(Single Instruction Multiple Data)*

Comme le nom le suggère, il s'agit d'ordinateurs capables d'appliquer à un moment donné la même opération à un ensemble de données, ce qui postule une certaine similarité de celles-ci. Ce postulat est rarement satisfait dans le cas général, mais il s'applique bien à un cas particulier important, celui du calcul vectoriel ou matriciel, qui a de nombreuses applications pratiques en météorologie, en aérodynamique, en calcul de structures, en océanographie, en sismologie, bref dans tous les domaines pour lesquels la physique ne donne pas de solution analytique simple mais où un modèle heuristique étalonné sur des données recueillies pour des problèmes à solution connue et appliqué à un ensemble de données empiriques aussi vaste que possible permet d'atteindre une solution approchée satisfaisante. Le modèle SIMD a connu sous le nom d'ordinateur vectoriel un certain succès, qui se perpétue sous d'autres formes (proceseur graphique, dit aussi GPU, par exemple).

En réalité un processeur vectoriel ne dispose que d'un seule unité de contrôle, qui pilote de multiples unités d'exécution. La réalisation d'un tel ordinateur pose des problèmes théoriques raisonnablement solubles ; il en va de même pour la programmation.

Les super-ordinateurs Cray, Fujitsu, NEC, les mini-super Convex, les processeurs spécialisés *Floating Point Systems* ont connu de réels succès jusque dans les années 1980 et 1990 en employant à des degrés divers la technologie vectorielle.

Si la baisse de prix des processeurs ordinaires les a peu à peu écartées, ces architectures pourraient revenir en scène lorsque les progrès de la technologie classique rencontreront des obstacles sérieux, et s'appliquer assez bien à tous les problèmes de traitement et de création d'images. D'ailleurs le développement extraordinaire des jeux vidéo, qui sont aujourd'hui (en 2018, comme en 2002 pour la première édition de ce texte) l'un des aiguillons les plus aiguisés de la recherche micro-électronique, a nécessité la conception de processeurs graphiques époustouflants qui recyclent pas mal de technologie vectorielle.

9.1.2 Architectures cellulaires et systoliques

Le modèle SIMD poussé à son extrême donne le modèle cellulaire, conçu pour exploiter un parallélisme de données massif : on a toujours un processeur par donnée, les processeurs sont très simples mais ils se comptent par dizaines de milliers. Il y eut surtout des prototypes de laboratoire, excepté le premier modèle de la *Connection Machine* de l'entreprise modestement nommée *Thinking Machines Corporation*. Cet ordinateur, doté de 65 536 processeurs, chacun flanqué d'une mémoire locale de 65 536 bits, avait été conçu par Daniel Hillis, un étudiant en intelligence artificielle au MIT et réalisé grâce à une commande du DoD (ministère de la Défense américain). Il s'en vendit une vingtaine d'exemplaires (dont deux en France) pour un coût unitaire équivalent à une dizaine de millions de dollars.

Une variante du modèle cellulaire est l'ordinateur systolique, qui exploite en outre un parallélisme de flux de données : un vecteur de données passe par des processeurs successifs qui correspondent à des étapes successives de calcul et qui effectuent chacun un type d'opération.

Il convient également de mentionner la conception et la réalisation de processeurs systoliques spécialisés pour certaines applications, notamment la comparaison de séquences biologiques. La plupart de ces processeurs, par exemple le réseau systolique linéaire SAMBA de 256 processeurs pour la comparaison de séquences biologiques développé par Dominique Lavenier à l'IRISA de Rennes, implémentent l'algorithme classique de Smith et Waterman (un algorithme dit de « programmation dynamique »).

La comparaison de séquences biologiques est une comparaison inexacte : il s'agit de savoir si deux chaînes de caractères représentant deux séquences d'ADN (l'alphabet est ATGC) ou deux séquences de protéines (l'alphabet est alors plus vaste pour représenter les vingt acides aminés et quelques autres informations) sont suffisamment semblables, au prix de quelques disparités de caractères (de possibles mutations), de quelques décalages (insertions ou délétions).

Le système est constitué de plusieurs processeurs capables surtout de comparer deux caractères entre eux. Les processeurs sont disposés en série ; s'il y en a cent on peut placer dans la mémoire locale de chacun d'eux un caractère d'une séquence de cent caractères que l'on veut étudier en la comparant à une collection de séquences connues par ailleurs. On fait défiler les séquences de la collection (généralement nommée banque) dans les processeurs qui détiennent la séquence étudiée (la cible). Il n'est pas possible d'exposer ici le détail de l'algorithme, mais cette architecture est très bien adaptée à cet algorithme. Malheureusement elle est limitée à un seul algorithme.

9.1.3 MIMD *(Multiple Instructions Multiple Data)*

Les machines MIMD sont des ordinateurs capables d'exécuter simultanément de nombreuses instructions quelconques appliquées à des données également quelconques. En fait il y a de multiples processeurs qui se partagent une mémoire unique, parfois ils ont chacun en plus une mémoire locale, quelquefois ils n'ont qu'une mémoire locale. L'interconnexion de tous ces processeurs, leur synchronisation et le maintien de la cohérence de la mémoire partagée posent des problèmes absolument passionnants de conception du matériel et du système d'exploitation : une fois ceux-ci résolus il reste à programmer tout ça, et c'est là qu'achoppe l'ambition de ces tentatives. La complexité de la programmation est telle que le temps qu'elle absorbe suffit aux processeurs ordinaires pour combler l'écart de performance que l'architecture MIMD était censée creuser. Ces architectures connaîtront une nouvelle faveur lorsque la courbe inexorable du progrès des processeurs de von Neumann s'infléchira.

Parmi les réalisations MIMD les plus achevées on retiendra les dernières *Connection Machines* de *Thinking Machines Corporation*, entreprise disparue durant les cruelles années 1990. Citons aussi les *Transputers* de la société britannique *Inmos*, destinés à former les éléments de base d'architectures plus complexes, inspirés des travaux de C.A.R. Hoare[1] sur les processus séquentiels communicants. Le Transputer a connu un certain succès, notamment dans le

1. Le Britannique C. Antony R. Hoare est à l'origine de quelques contributions de premier plan. Des études universitaires de tonalité plutôt littéraire lui ont donné l'occasion de partir en stage en 1960 dans le laboratoire de Kolmogorov à l'Université de Moscou pour travailler sur un projet de traduction automatique. Pour réaliser un dictionnaire électronique nécessaire à ce projet (par ailleurs vite abandonné) il inventa l'algorithme de tri « *Quicksort* » que tout étudiant en informatique se doit d'avoir étudié en détail et programmé. Ce stage lui avait donné le goût de la programmation, mais, comme il le raconte lui-même avec humour, il eut la chance que sa formation initiale lui fermât les portes du *UK National Physical Laboratory*, et il entra dans l'industrie. C'est là qu'il eut l'occasion de participer au développement de systèmes d'exploitation, domaine pour lequel il élabora la méthode des moniteurs de Hoare afin de contrôler les accès concurrents et exclusifs à des ressources partagées, méthode reprise pour les *threads* du langage

domaine industriel.

Ce rapide tour d'horizon des architectures non von Neumann illustre leur échec général dû à la difficulté de leur programmation. À leur apparition elles ont souvent énormément séduit, mais la nécessité de recourir à des méthodes et à des langages de programmation peu répandus et maîtrisés par peu d'ingénieurs a vite découragé industriels et clients. D'où le succès, en revanche, de contournements du principe de von Neumann par des voies moins radicales, qui recherchaient le même résultat tout en conservant la sémantique classique et que nous allons examiner maintenant.

9.2 Architectures réformistes

Parmi les différentes réformes apportées à l'architecture de von Neumann pour en améliorer les performances tout en conservant un comportement extérieur apparent identique, la plus importante par ses conséquences est peut-être la substitution à la mémoire centrale unique d'une hiérarchie de mémoires de temps d'accès décroissants, depuis le cache de niveau 1 incorporé au processeur jusqu'à la mémoire auxiliaire de pages sur disque : mais cette notion de hiérarchie de mémoire dépasse le cadre du présent chapitre et elle est abordée dans plusieurs chapitres du corps de cet ouvrage. Nous examinerons ici les architectures en « pipe-line » et super-scalaires. Mais auparavant nous devons donner un peu plus de détails sur le déroulement des instructions que nous ne l'avions fait à la sous-section 2.4.1.

Séquence d'exécution d'une instruction

Dans l'exposé du principe de l'ordinateur à la sous-section 2.4.1 nous avons considéré chaque instruction machine comme une opération indivisible dans le temps, *atomique*, et il en est bien ainsi selon von Neumann. En fait la plupart des instructions sont effectuées selon une séquence temporelle identique et régulière dont nous allons exposer un exemple.

L'exécution des instructions successives de notre architecture a lieu selon la séquence suivante. Notons que cet exemple, pour une raison que nous allons voir, postule des instructions à format fixe.

1. **Étape de lecture `FETCH` :** l'unité de contrôle va chercher en mémoire la prochaine instruction à exécuter. Comment sait-on où aller la chercher ?

Java, et la théorie des processus séquentiels communicants, encore aujourd'hui le seul modèle complet et cohérent qui excède réellement le modèle de von Neumann.

L'unité de contrôle maintient cette information dans un **compteur de programme** (**PC**) qui à chaque instant contient l'adresse en mémoire de l'instruction suivante. Dans le cas le plus simple (pas de débranchement) c'est l'adresse du mot qui suit l'instruction en cours. Dans le cas d'un débranchement : eh bien l'instruction de débranchement consiste précisément à placer dans le PC l'adresse de l'instruction à laquelle le programme doit se poursuivre (doit « sauter » : les débranchements sont aussi appelés sauts ; on distingue les branchements conditionnels et les sauts simples, inconditionnels).

Le PC peut résider dans un registre de l'unité centrale. Dans notre exemple simple il serait possible de lui réserver R qui n'est pas utilisé à autre chose. Dans beaucoup de processeurs le PC est une partie du **PSW** *(Program Status Word)* qui conserve différents indicateurs fondamentaux du processeur.

2. **Étape de décodage** `DEC:` l'unité de contrôle analyse le code opération (5 premiers bits dans notre exemple de la section 2.4.1) et détermine ainsi le circuit logique qui correspond à l'instruction désirée. Simultanément au décodage, l'unité de contrôle effectue la lecture des registres impliqués dans l'instruction. Cette simultanéité impose des instructions à format fixe, où le nombre et l'emplacement des bits qui désignent les registres soient toujours les mêmes. Le contenu des registres est recopié dans des registres de travail temporaires.
3. **Étape d'exécution** `EXEC:` l'instruction déterminée à l'étape de décodage est exécutée ; s'il doit y avoir un accès mémoire l'adresse effective est calculée ; s'il s'agit d'un branchement conditionnel le contenu du registre lu à l'étape précédente permet de déterminer si le branchement doit être « pris » et si oui l'adresse de branchement est calculée.
4. **Étape d'accès mémoire** `MEM:` cette étape a lieu pour les opérations de chargement et de rangement en mémoire et pour les branchements. Les chargement ou rangements ont lieu, ou la valeur convenable est placée dans le PC. Ces opérations ont lieu dans le cache, ce qui explique que cette étape ait une durée comparable aux autres.
5. **Étape d'écriture du résultat** `RES` dans les registres affectés.

9.3 Le pipe-line

9.3.1 Principe du pipe-line

Pour l'exemple de processeur que nous venons de donner, l'exécution d'une instruction se décompose en cinq étapes plus élémentaires. À part l'exécution proprement dite (étape `EXEC`), ces étapes sont les mêmes quelle que soit

l'instruction particulière qui doit être exécutée. L'idée est venue de confier chaque étape à une unité de circuits logiques spécifique : unité de lecture, unité de décodage, unité d'exécution, unité d'accès mémoire, unité de livraison du résultat. Ainsi le processeur peut commencer à traiter une instruction avant que la précédente soit terminée, en utilisant les unités libérées par la terminaison des premières étapes. Soient i_1,i_2,i_3,i_4,i_5 cinq instructions consécutives, elles seront traitées ainsi :

i_1	FETCH	DEC	EXEC	MEM	RES				
i_2		FETCH	DEC	EXEC	MEM	RES			
i_3			FETCH	DEC	EXEC	MEM	RES		
i_4				FETCH	DEC	EXEC	MEM	RES	
i_5					FETCH	DEC	EXEC	MEM	RES

Cette structure s'appelle un *pipe-line*, parce qu'elle évoque un tuyau dans lequel les instructions s'engouffrent les unes derrière les autres sans attendre que la précédente soit sortie. Nos instructions sont découpées en cinq étapes, ce qui fait que notre pipe-line à un moment donné contient cinq instructions en cours d'exécution : on dit que c'est un pipe-line à cinq étages. Certains processeurs ont des pipe-lines avec sept ou huit étages, voire plus : le Pentium III a douze étages de pipe-line, le Pentium IV en a vingt.

9.3.2 Histoire et avenir du pipe-line

L'idée du pipe-line est apparue assez tôt ; l'IBM 7030 *Stretch* (1960), une machine à mots de 64 bits, est considéré comme la première machine à pipe-line. Le Stretch succédait au 704 et devait être 100 fois plus puissant. Il n'en a été construit que seize exemplaires[2] mais cette machine a représenté une étape importante avec beaucoup d'innovations.

Le Control Data 6600, une autre machine novatrice apparue en 1964, considérée comme le premier super-ordinateur et dont l'architecte principal était Seymour Cray, comportait aussi une structure de pipe-line, et ses concepteurs avaient compris que pour qu'elle fonctionne efficacement il fallait des instructions simples, de longueur fixe, toutes à peu près de la même durée d'exécution, si possible en un cycle. Seymour Cray poussera ces principes à l'extrême dans les super-ordinateurs vectoriels qu'il construira sous son nom pour Cray Research, et qui furent les plus puissants de leur époque. Ce sont ces idées que nous retrouverons au cœur de l'architecture RISC *(Reduced Instruction Set Computer)*, cf. ci-dessous.

2. La console de l'un d'eux, acquis par le CEA, figure dans les réserves du Musée National des Techniques au Conservatoire National des Arts et Métiers, d'où elle est parfois extraite pour des expositions temporaires.

Cette technique du pipe-line et l'évolution consécutive des techniques de compilation se sont beaucoup développées avec l'apparition au milieu des années 1980 des processeurs à architecture RISC, par opposition à la mode précédente rétrospectivement baptisée CISC *(Complex Instruction Set Computer)*.

9.3.3 Cycle de processeur, fréquence d'horloge

Qu'est-ce qu'un *cycle* de processeur ? Toutes les opérations dans un processeur sont synchronisées par une *horloge*, ce qui permet de savoir à quel moment le résultat d'une micro–opération va être disponible. J'appelle micro–opération une opération moins complexe qu'une instruction, par exemple une étape d'instruction telle que décrite à la section 9.2. Il est nécessaire de savoir à quel instant précis telle modification de tel registre sera disponible et stable, pour pouvoir enchaîner la micro–opération suivante : c'est le rôle de l'horloge.

Comment fonctionne l'horloge du processeur ? Elle repose sur un dispositif à quartz, qui régule un circuit oscillant selon une fréquence extrêmement précise. À chaque fin de période le circuit oscillant émet un signal. On rappelle que la période est l'inverse de la fréquence : si un circuit a une période de 1/50 de seconde, on dit que sa fréquence est de 50 Herz (Hz). La sortie du circuit logique correspondant à une micro–opération est couplée à une entrée d'un circuit ET dont l'autre entrée est couplée à l'horloge. À chaque *top* d'horloge la porte ET délivre un résultat. Ainsi toutes les opérations sont synchronisées. Ce que l'on appelle la fréquence d'un processeur est la fréquence de son horloge.

9.3.4 Processeurs asynchrones

Ce cadencement des opérations simplifie grandement la conception des circuits, mais on peut imaginer qu'il fait perdre du temps à certains endroits : toutes les micro-opérations ne durent pas le même temps et certains résultats obtenus ne sont pas disponibles parce qu'on attend le top d'horloge. Aussi existe-t-il un domaine de recherche prometteur, les processeurs asynchrones. La synchronisation est remplacée par la signalisation : un élément de circuit qui obtient un résultat prévient le consommateur de ce résultat par l'émission d'un signal, ce qui nous place dans une logique assez semblable à celle des protocoles de réseau, où l'on ignore toujours ce qui se passe « à l'autre bout ». Parmi les avantages collatéraux fournis par l'asynchronisme, signalons une consommation électrique et des interférences électromagnétiques réduites.

Comme beaucoup de techniques avancées, les processeurs asynchrones sont aujourd'hui un domaine assez confidentiel, mais qui pourrait connaître une grande expansion au fur et à mesure que les progrès des processeurs classiques deviendront plus laborieux.

9.3.5 Apport de performances par le pipe-line

Le pipe-line est un facteur considérable d'accélération des processeurs, et notamment d'augmentation de la fréquence nominale.

Supposons une architecture classique où le temps total d'exécution d'une instruction simple telle qu'un chargement de registre ou une addition registre à registre correspond à un ou deux cycles (même si des instructions complexes peuvent prendre dix ou vingt cycles).

Prenons maintenant une architecture avec un pipe-line à cinq étages, comme dans notre exemple, et avec des éléments micro–électroniques aussi rapides que notre architecture classique hypothétique. Notre instruction va prendre le même temps pour s'exécuter, mais elle est maintenant cadencée par le passage dans les cinq étages du pipe-line. Chaque étage correspond à un cycle. L'instruction qui s'exécutait en deux cycles s'exécute maintenant en cinq cycles, toujours dans le même temps : c'est que chaque cycle est plus court. La fréquence a été multipliée par 2,5. C'est ainsi que le pipe-line est l'arme secrète qui permet les fréquences élevées des processeurs contemporains.

Certes, pour une instruction donnée chaque cycle « fait » moins de choses, mais ce n'est pas une escroquerie : à chaque fin de cycle il y a bien livraison du résultat d'une instruction. Enfin presque. Parce qu'il y a quand même des difficultés qui s'opposent parfois au fonctionnement continu du pipe-line.

9.3.6 Limite du pipe-line : les branchements

La première difficulté inhérente au pipe-line est de bon sens : on commence à exécuter plusieurs instructions consécutives, mais s'il y a un débranchement au milieu l'ordre d'exécution sera modifié. Notre processeur aura commencé à exécuter des instructions dont la suite des événements montre que ce n'était pas les bonnes.

Quand sait-on si un branchement conditionnel a effectivement lieu ? Lors de l'étape `EXEC`. Quand est-il exécuté ? Lors de l'étape `MEM`.

Si par exemple i_3 est un branchement conditionnel et que le chemin choisi soit effectivement le débranchement, cette condition est connue lors de l'étape `EXEC` de i_3. À ce stade, i_4 est déjà passée par les étapes `FETCH` et `DEC`, i_5 par l'étape `FETCH`. Rien d'irrémédiable n'a été accompli, ni accès mémoire ni écriture de résultat, mais on a perdu du temps, en exécutant des étapes pour des instructions inutiles et il faut « vider » la suite du pipe-line pour le recharger avec d'autres instructions.

Il est assez clair que si les débranchements sont fréquents le gain de performances procuré par le pipe-line va fortement diminuer. Plus le pipe-line a d'étages plus la pénalité sera forte, et dans certains cas il sera nécessaire de restituer à certains registres leur valeur antérieure, modifiée par une exécution

trop hardiment anticipatrice. Le processeur devra comporter des circuits logiques destinés à traiter ces situations de retour en arrière. On le voit, tout gain de performance a un coût qui en atténue l'avantage.

Les architectes de processeurs déploient des techniques très subtiles pour éviter l'occurrence trop fréquente de ces situations de débranchement (de saut) où le pipe-line « cale » *(to stall)*. La prédiction de branchement consiste à essayer de savoir à l'avance, par l'examen anticipé du texte du programme, si un branchement va avoir lieu, et dans ce cas modifier l'ordre des instructions pour annuler le branchement. Dans le cas d'une section de programme répétitive, l'historique des exécutions permet de savoir quels sont les branchements les plus probables et de réorganiser le texte du programme pour que le cas le plus fréquent se déroule sans branchements. En désespoir de cause on peut recourir à la technique du « branchement retardé » qui consiste à insérer une instruction nulle derrière le branchement, ce qui évite d'avoir à commencer des instructions qu'il faudra annuler : ceci n'est jamais nécessaire dans notre exemple de pipe-line à cinq étages, mais peut l'être avec un plus grand nombre d'étages ; dans ce cas la détection de branchement pourrait intervenir après que les instructions suivantes auront modifié des registres, qu'il faudrait alors restaurer à leur valeur précédente, ce qui serait extrêmement complexe.

Les auteurs de compilateurs sont aussi mis à contribution dans la lutte contre les branchements. Ils sont invités à produire du code machine aussi exempt de branchements que possibles, à anticiper les cas les plus probables pour leur réserver l'exécution la plus linéaire possible. Évidemment ces efforts ne réussissent pas toujours et il reste des branchements à détecter dynamiquement, c'est-à-dire lors de l'exécution (par opposition à une détection statique, par l'examen du texte du programme).

9.3.7 Limite du pipe-line : les interruptions

La seconde difficulté de réalisation d'un pipe-line efficace est d'une nature voisine de celle que nous venons d'examiner, mais encore plus redoutable. Nous avons bien des difficultés avec les débranchements, mais au moins ce sont des événements internes à un programme, à un processus et à un processeur. Mais qu'en est-il lorsque survient un événement asynchrone, dont le type par excellence est l'interruption ?

Dans une architecture traditionnelle, von-Neumannienne de stricte obédience, le processeur à un instant donné exécute au plus une instruction. La valeur du compteur ordinal (PC, **eip**...) désigne exactement la limite entre ce qui est déjà exécuté et ce qui reste à exécuter. C'est en général après la terminaison de chaque instruction que le processeur scrute les registres de contrôle des interruptions pour savoir s'il y en a une en attente, auquel cas il décide de la traiter, ou de ne pas la traiter d'ailleurs si les interruptions sont masquées

à ce moment. Le cas des interruptions pour faute de page est un exemple de situation un peu différente, où le PC après l'interruption doit pointer sur l'instruction qui a causé la faute afin qu'elle puisse être exécutée à nouveau, la page étant désormais en mémoire. Le problème est assez bien circonscrit.

Avec une architecture à pipe-line, l'interruption survient dans un contexte où plusieurs instructions sont en cours d'exécution à des stades variés. La valeur du compteur ordinal ne rend pas forcément fidèlement compte de la limite entre ce qui est exécuté et ce qui ne l'est pas. Sans doute, elle donne l'adresse de la prochaine instruction à introduire dans le pipe-line, plus probablement que celle de la dernière instruction exécutée. Bref, après avoir traité l'interruption le processeur devra déterminer où il s'était arrêté dans le pipe-line et trouver un état bien déterminé pour redémarrer, ce qui ne sera pas simple.

Nous n'entrerons pas dans les détails de ce problème, qui relève de l'architecture matérielle plus que du système d'exploitation, mais sachons que pour mieux le cerner les processeurs modernes ont souvent deux types d'interruptions : les *interruptions précises* qui laissent le processeur dans un état bien défini (PC sauvegardé en un endroit connu, instructions antérieures à celle pointée par le PC totalement exécutées, instructions ultérieures non entamées, état de l'instruction courante connu), et les *interruptions imprécises* qui laissent tout en chantier, charge au programmeur de système de reconstituer une situation de redémarrage cohérente à partir des informations d'état que le processeur aura obligeamment déversées sur la pile. Il y a des cas où une interruption imprécise fait très bien l'affaire, par exemple si elle correspond à une erreur fatale l'état ultérieur du programme importe peu.

Pour prendre un exemple, l'architecture *ARC 700* lancée en 2004 comporte un modèle d'interruptions précises qui permet, au choix, de forcer la terminaison de toutes les instructions qui précèdent celle qui a provoqué l'exception ; d'annuler l'instruction fautive avant de valider son résultat ; d'avertir la routine de traitement des exceptions de la cause de l'exception ; et de relancer le pipe-line après le traitement de l'exception dans l'état où il se trouvait auparavant.

Les interruptions précises augmentent considérablement la complexité des circuits matériels dévolus au contrôle des interruptions, et de ce fait les architectes essayent de les éviter. Les interruptions imprécises sont de plus en plus populaires parmi les architectes de processeurs, et comme d'habitude la complexité est renvoyée aux concepteurs de systèmes d'exploitation, ce qui est après tout de bonne politique : il est plus facile de modifier un paragraphe de programme qu'un morceau de silicium (déjà installé dans le salon du client de surcroît).

Lorsque nous aborderons le modèle d'exécution super-scalaire quelques pages plus bas, nous verrons que les difficultés causées par un contexte mal déterminé lors d'une interruption sont encore plus graves.

9.4 RISC, CISC et pipe-line

9.4.1 Architecture des ordinateurs, avant les microprocesseurs

Jusque dans les années 1970 les concepteurs d'unités centrales (qui n'étaient pas encore des microprocesseurs) pensaient que les progrès dans cet art viendraient de jeux d'instructions machine de plus en plus riches, qui rapprocheraient le langage machine des langages dits *évolués*, c'est-à-dire, par rapport au langage machine, plus proches du langage humain. Il en résulta l'architecture dite *(ex-post)* CISC, avec des instructions de plus en plus élaborées, qui effectuaient des opérations complexes.

Entre les langages évolués et le langage machine il y a le langage assembleur, dont les instructions sont, une pour une, celles du langage machine, mais écrites selon une graphie plus confortables. L'assembleur procure aussi quelques aides au programmeur, notamment des symboles pour représenter les adresses, le calcul automatique du déplacement entre deux adresses, etc. Chaque instruction machine occupe en mémoire un nombre de mots déterminé, rigide, un programme assembleur n'est pas un texte dont la composition serait laissée à la discrétion du programmeur, la disposition du texte correspond à sa disposition dans la mémoire de l'ordinateur.

9.4.2 Apogée des architectures CISC

Les machines CISC qui ont connu leur apogée au début des années 1980 (VAX de Digital Equipment Corporation, 68000 de Motorola) avaient un jeu d'instructions très vaste (plus de 300 pour le VAX) et très complexe, avec des instructions de longueurs différentes, et même parfois de longueur variable selon les opérandes. La richesse du jeu d'instructions était censée faciliter la tâche des programmeurs qui utilisaient le langage assembleur et, surtout, des auteurs de compilateurs pour langages évolué, en leur fournissant des instructions machine qui ressemblaient déjà à du langage évolué, plus facile et maniable pour le programmeur. D'ailleurs le langage C, langage évolué de bas niveau (on a pu le qualifier d'assembleur portable, la notion de portabilité désignant l'aptitude à fonctionner sur des ordinateurs d'architectures différentes), est assez largement inspiré de l'assembleur des ordinateurs PDP, ancêtres des VAX.

Cette richesse et cette complexité du jeu d'instructions avaient bien sûr un coût en termes de complexité et de lenteur du processeur. Le risque était notamment que les instructions simples (chargement de registre depuis la mémoire, copie de registre vers la mémoire, addition registre à registre) soient condamnées à s'exécuter aussi lentement que les opérations complexes (copie de zone mémoire à zone mémoire de longueur variable, opérations complexes en mémoire). Le VAX notamment n'esquivait guère ce risque.

9.4.3 Naissance de l'idée RISC

Dans la seconde moitié des années 1970 des chercheurs ont fait des statistiques sur la composition en instructions des programmes en langage machine, soit qu'ils aient été directement écrits en assembleur, soit qu'ils aient été produits par un compilateur. Citons notamment les travaux de D.A. Fairclough, (*A unique microprocessor instruction set*, IEEE Micro, mai 1982 [52]). Ils constatèrent d'abord que 43% des instructions étaient des déplacements de données d'un endroit à un autre, que le quart des instructions étaient des branchements, ensuite que pour chaque programme le nombre d'instructions utilisées était très réduit, enfin que seules les instructions les plus simples étaient largement utilisées. Une autre constatation était que les opérations de loin les plus coûteuses étaient l'appel de sous-programme (un programme lance l'exécution d'un autre programme en lui communiquant des paramètres) et le retour d'un sous-programme au programme principal.

Sur la base de ces constatations ils préconisèrent de concevoir des processeurs avec un jeu réduit d'instructions plus simples. La notion de processeur RISC était née, et le premier processeur de ce type, l'IBM 801, fut réalisé par John Cocke en 1979. La société MIPS, fondée par John Hennessy, pionnier du RISC à l'Université Stanford, fut créée en 1985. Hewlett-Packard fut le premier grand constructeur d'ordinateurs à réaliser toute sa gamme en architecture RISC en 1986. Sun et Digital Equipment suivirent.

9.4.4 Avènement des microprocesseurs RISC

Le livre emblématique de la révolution RISC, *Computer architecture: a quantitative approach* [58], a été écrit par John L. Hennessy, l'architecte des processeurs MIPS, et David A. Patterson, originaire du campus de Berkeley, l'architecte des processeurs SPARC de Sun. Les processeurs MIPS ont été les premiers à défricher la voie, et les plus hardiment innovateurs : nous l'avons vu au chapitre 4 à propos de l'utilisation du TLB. Il faudrait ajouter à ce répertoire Richard L. Sites, l'architecte des processeurs Alpha de Digital.

Les traits les plus frappants initialement dans les architectures RISC étaient le petit nombre d'instructions, avec l'absence d'instructions mémoire–mémoire : il n'y avait que des chargements de mémoire dans un registre, des copies de registre vers la mémoire et des opérations dans les registres.

D'autres traits se révélèrent bientôt aussi importants : longueur et format d'instruction fixes, usage intensif du pipe-line. Pour tirer parti correctement d'une architecture aussi ascétique une grande part de la complexité était reportée sur les compilateurs, qui devaient produire un code capable d'utiliser efficacement les registres et le pipe-line.

La nouvelle architecture se révéla bientôt extrêmement rapide, et doublement rapide: en effet, pour tirer parti d'un progrès de la micro-électronique, il ne suffit pas de concevoir un processeur rapide, il faut aussi que le délai nécessaire à sa conception ne soit pas trop long. La simplicité du RISC était aussi un atout de ce côté. Actuellement il faut à peu près trois ans à une équipe de 100 à 200 ingénieurs pour concevoir un nouveau processeur. Une usine entièrement équipée pour le construire coûtera de l'ordre de quatre milliards de dollars. La conception d'une architecture novatrice demande une douzaine d'années.

9.4.5 Résistance des architectures CISC

La technologie CISC parut condamnée, ce qui fut effectivement le cas pour les gammes VAX et Motorola 68000. Tout le monde attendait la chute de l'architecture x86 d'Intel, sur laquelle reposent les dizaines de millions de PC vendus chaque année. C'était compter sans les efforts qu'Intel pouvait mobiliser grâce à la rente du PC. Les Pentium actuels, depuis le Pentium Pro de 1995 (architecture P6 conçue par Bob Colwell), sont en fait des processeurs constitués d'un noyau RISC autour duquel des circuits supplémentaires et du micro-code (notion introduite ci-dessous à la section 9.4.7) simulent l'ancienne architecture CISC afin de préserver la compatibilité avec les systèmes et les programmes existants. On se reportera pour plus de détails à un excellent article de Samuel « Doc TB » Demeulemeester pour Canard PC Hardware [44], qui retrace toute l'histoire des processeurs Intel.

Quant à la gamme des grands systèmes IBM, l'immense stock de programmes existants dont la conversion exigerait des dépenses phénoménales semble la vouer à une immortalité dont l'érosion ne se fait qu'au gré du changement lent des applications.

L'évolution du début des années 2000 a suggéré un demi-échec de l'architecture RISC, qu'Intel et HP croyaient pouvoir remplacer par l'architecture VLIW *(Very Long Instruction Word)* avec IA-64, qui s'est révélée un échec. Le mouvement RISC a profondément révolutionné la conception des processeurs et aussi, de façon moins spectaculaire mais aussi profonde, celle des techniques de compilation. En fait, hommage du vice à la vertu, tous les processeurs modernes comportent un cœur RISC entouré de circuits qui procurent un décor différent, Pentium par exemple (cf. p. 333).

Même les grands systèmes IBM sont maintenant animés par des microprocesseurs qui implémentent leur jeu d'instructions traditionnel en RISC.

9.4.6 L'avenir appartient-il au RISC ?

Si le marché des ordinateurs, serveurs comme appareils personnels, est encore monopolisé par les architectures CISC d'Intel (et accessoirement d'AMD,

qui produit des processeurs compatibles x86), il faut mesurer les conséquences de l'extraordinaire prolifération des téléphones mobiles et des tablettes, qui sont des ordinateurs Turing-complets propulsés par des processeurs RISC de conception ARM. Ces processeurs sont de loin les plus répandus dans le monde, et si il y a quinze ans les accès au Web émanaient pour plus de 95 % d'ordinateurs x86 sous Windows, aujourd'hui pour plus de 50 % ils viennent d'appareils sous Android ou, sous iOS dotés de processeurs ARM.

Si les concepteurs de ces appareils portables ont choisi la plate-forme ARM, c'est pour de bonnes raisons : à puissance de calcul comparable, le poids et la consommation électrique sont bien moindres, d'au moins un ordre de grandeur. Il est tout à fait possible que les processeurs ARM, qui ont commencé leur carrière de façon quasi-artisanale, soient au cœur des architectures de demain.

Si les processeurs ARM sont les plus répandus, il convient de noter que la maison ARM (achetée en 2016 par le fonds japonais SoftBank) ne fabrique ni ne vend aucun microprocesseur, pas un seul, et d'ailleurs en termes de chiffre d'affaires c'est un nain au regard d'Intel : 1,17 milliard d'euros en 2017 (ce fut une entreprise britannique, basée à Cambridge, dont Apple fut actionnaire fondateur). D'où viennent alors ces milliards de processeurs, et où vont-ils ? La seconde question est celle qui appelle la réponse la plus simple : les processeurs ARM sont dans votre Game Boy, votre iPhone, votre iPad, votre appareil photo Canon, votre téléphone Samsung ou LG, etc. Et sans doute aussi dans votre voiture, votre téléviseur, la box qui vous relie à l'Internet, votre GPS, votre cadre pour photos numériques, etc.

Naguère, l'électronique de ces types d'appareils était assez rudimentaire, mais aujourd'hui les processeurs ARM qui les équipent en font des ordinateurs universels complets, Turing-complets comme disent les informaticiens.

Si ARM ne fabrique pas de processeurs, d'où viennent-ils ? Les 6 250 employés d'ARM (en 2018) conçoivent l'architecture des circuits et en réalisent les plans numériques. ARM vend ces plans à des entreprises, qui éventuellement les intègrent à des ensembles plus vastes, et qui les fabriquent ou les font fabriquer par des *fonderies de silicium*, dont les plus importantes sont Samsung, le taïwanais TSMC, le franco-italien STMicro, ou encore Global Foundries.

9.4.7 Micro-code : le retour

Le micro-code était un élément architectural intermédiaire entre le logiciel et le matériel, caractéristique des processeurs CISC, à peu près disparu sur les processeurs RISC. En fait il s'agissait d'une couche de logiciel de très bas niveau qui simulait certains éléments de matériel pour les présenter au logiciel. Ainsi l'architecture 360 comportait 16 registres généraux, mais les modèles les plus petits comme le 360/20 n'en avaient que deux (un accumulateur et un registre). La couche de micro-code, stockée dans une mémoire spéciale, présentait au

système d'exploitation une machine virtuelle à 16 registres, implantée sur la machine réelle à deux registres. Une disquette permettait de charger en mémoire une nouvelle version du micro-code. Le BIOS des PCs Intel joue un peu le même rôle, il sert notamment à présenter les périphériques au système sous un aspect différent de leur réalité physique, ce qui permet par exemple de voir tous les disques de la même façon, avec un adressage linéaire des blocs de données, quelle que soit leur structure réelle. Le micro-code était un facteur de complexité et de lenteur, ce pourquoi la révolution RISC en a fait table rase. Mais ne survit-il pas en fait à l'intérieur du processeur ? Si bien sûr : les processeurs modernes sont si complexes et embarquent tellement de mémoire sur le *chip* (la puce proprement dite) que leur réalisation est elle-même micro-programmée. Le processeur Intel Pentium, de la famille CISC, est construit autour d'un cœur RISC entouré de micro-code qui simule l'architecture officielle (cf. p. 333). Le micro-code est aussi présent dans les contrôleurs de disques et autres périphériques. À l'heure où il a officiellement disparu, il n'a jamais autant existé.

9.5 Super-scalaire

La décomposition des instructions en étapes plus élémentaires que nous avons examinée à la section 9.2 permet le pipe-line, mais aussi une autre forme de simultanéité que l'on appelle « super-scalaire », et qui consiste à avoir plusieurs unités d'exécution actives simultanément, selon le modèle de la figure 9.1. Ce modèle d'exécution se combine avec le pipe-line dans les architectures modernes.

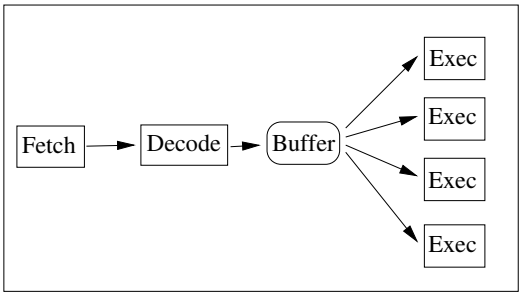


Figure 9.1 : *Modèle de l'exécution super-scalaire.*

Les instructions sont décodées à l'avance et emmagasinées dans le *buffer*, qui est en fait un jeu de registres. Dès qu'une unité d'exécution se libère, une instruction est extraite du buffer pour être exécutée. L'architecture super-scalaire de la figure 9.1 peut exécuter quatre instructions par cycle. Si de plus le pipe-line a divisé le temps de cycle par trois, la combinaison de ces deux techniques a multiplié la vitesse de notre processeur par 12 à technologie micro-électronique constante. Évidemment, ceci est vrai en l'absence de dépendances entre les

données manipulées par les instructions successives, parce que si le pipe-line introduisait des problèmes en cas de branchement, le traitement super-scalaire introduit des risques de collisions de données. Supposons une machine à seize registres, deux unités d'exécution et le programme suivant :

N°	Buffer	Unité d'exécution 1	Unité d'exécution 2
1 2 3 4	R2 + R3 → R4 R5 - R6 → R7 LOAD R8, COUNT R8 * R2 → R3		
	conflit de données → potentiel évité ici	R2 + R3 → R4 LOAD R8, COUNT R8 * R2 → R3	R5 - R6 → R7

Le conflit de données est le suivant : l'instruction 3 charge le registre R8 avec la valeur contenue à l'adresse mémoire COUNT. L'instruction 4 effectue la multiplication des contenus des registres R8 et R2 et écrit le résultat dans R3. L'écriture rigoureusement fidèle à von Neumann de notre programme stipule une exécution séquentielle, l'instruction 4 suit l'instruction 3 et utilise dans R8 la valeur que cette dernière y aura déposée.

Si nous lançons l'instruction 3 dans l'unité d'exécution 1 et, simultanément, l'instruction 4 dans l'unité d'exécution 2, quelle sera la valeur du contenu de R8 utilisée par l'instruction 4 ? Celle qui résulte de l'exécution de l'instruction 3, ou celle qu'il contenait auparavant ? La réponse à cette question est indéterminée, et pour maintenir la sémantique d'une machine de von Neumann le processeur doit contenir une logique capable de détecter ce problème et de décider de ne pas exécuter ces deux instructions simultanément, ce qu'illustre notre figure. Cette détection doit avoir lieu dynamiquement, à la volée, pour chaque exécution de cette séquence d'instructions.

L'évitement des conflits de données complique le circuit et diminue le bénéfice attendu de l'architecture super-scalaire. Ce bénéfice reste néanmoins suffisant pour que tous les processeurs modernes utilisent ces techniques, même au prix d'une complexité proprement diabolique. La section suivante nous montrera comment les architectes de processeurs se sont débarrassés de cette complexité et ont passé la patate chaude aux auteurs de compilateurs.

Bien évidemment, les difficultés que nous avons signalées à la section 9.3.7 au sujet du pipe-line et qui procèdent de la survenue d'événements asynchrones (les interruptions) dans un contexte où plusieurs instructions sont en cours d'exécution à des stades variés d'achèvement, ces difficultés se retrouvent encore aggravées par l'exécution super-scalaire, puisque ce modèle respecte encore moins l'ordre d'exécution que le pipe-line.

9.6 Architecture VLIW *(Very Long Instruction Word)*

Les modèles d'exécution en pipe-line et super-scalaire ont l'avantage considérable sur les architectures SIMD ou MIMD de préserver la sémantique de l'architecture de von Neumann, ce qui permet de continuer à utiliser les langages de programmation traditionnels avec les méthodes et les compétences qui existent et qui sont éprouvées. Le prix à payer pour cet avantage, c'est d'avoir à faire de la prédiction de branchement, de la prévention de conflits de données et du traitement d'interruptions imprécises, et nous avons vu que ce n'était pas précisément simple.

L'espoir forcément est né de bénéficier des avantages sans avoir à supporter les inconvénients. La différence entre le monde de la technique et celui des relations humaines, c'est que dans le premier une telle idée est vertueuse.

Pour faire fonctionner pipe-line et multiples unités d'exécution sans arrêts brutaux, il suffirait que le programme (ou plutôt le compilateur qui traduit le programme en langage machine) « dise » au processeur quelles sont les instructions susceptibles d'être exécutées simultanément sans risque de conflit de branchement ou de données, et que le processeur soit équipé d'un format d'instruction capable de recevoir ces informations.

Les ingénieurs de Hewlett-Packard et d'Intel ont réuni leurs efforts pour concevoir une telle architecture, connue sous le nom propre **IA-64**, qui est une architecture **VLIW** *(Very Long Instruction Word)*. La conception de IA-64 a pris une douzaine d'années avant de déboucher en 2001 sur la livraison d'un premier processeur, **Itanium**.

9.6.1 Parallélisme explicite

La méthode mise en œuvre par IA-64 s'appelle EPIC *(Explicitly Parallel Instruction Computing)*. Le processeur reçoit du compilateur une liasse *(bundle)* de 128 bits. Chaque liasse comporte trois instructions de 41 bits et un masque *(template)* de 5 bits. Chaque instruction comporte trois numéros de registre de sept bits chacun (ce qui autorise $2^7 = 128$ registres), six bits de registre de prédicat *(predicate register)* et un code opération de 13 bits.

Liasse *(bundle)* IA-64 :

Instruction (41 bits)	Instruction (41 bits)	Instruction (41 bits)	Masque (5 bits) *(template)*

Instruction IA-64 :

code opération	n° registre prédicat *(predicate register)*	n° registre 1	n° reg. 2	n° reg. 3

Les cinq bits de masque indiquent quelles sont les instructions qui peuvent s'exécuter en parallèle, ainsi que l'éventualité du chaînage de cette liasse avec une autre liasse qui en contiendrait la suite. Les compilateurs peuvent placer dans ce masque des valeurs qui indiquent au processeur les instructions à lancer en parallèle[3].

Confier au compilateur tout ce travail auparavant réalisé par le processeur présente plusieurs avantages. Le processeur ne possède aucune information *a priori* sur le programme qu'il exécute, il ne peut que les déduire du texte binaire au fur et à mesure qu'il en décode les instructions, tandis que l'auteur du compilateur peut se conformer à une stratégie systématique d'organisation des instructions.

De plus, le compilateur peut passer beaucoup de temps à optimiser le code, en principe le programme n'est compilé que par son auteur, ou par celui qui l'installe sur un ordinateur, alors qu'il sera exécuté par de nombreux utilisateurs. Il est efficace de consacrer du temps à la compilation pour en gagner à l'exécution.

3. Pour ce faire, le processeur Itanium, premier de l'architecture IA-64, possède deux unités d'exécution d'instructions arithmétiques entières, deux unités d'exécution d'instructions arithmétiques à virgule flottante, trois unités de chargement - déchargement de registre, un pipe-line à dix étages.

9.6.2 Élimination de branchements

Imaginons le fragment de programme suivant en pseudo-langage évolué :

```
Si I égale 0
   Alors
      instruction 1
   Sinon
      instruction 2
```

que le compilateur traduirait en pseudo–assembleur pour un processeur RISC classique (non–EPIC) :

```
       COMPARE I à 0
       SAUTE   à l'étiquette SINON si non-égal
ALORS: instruction 1
       SAUTE   à l'étiquette SUITE
SINON: instruction 2
SUITE:
```

Voici comment procédera un processeur EPIC :

```
COMPARE I à 0
commence à décoder instruction 1,
     le prédicat positionné pour pointer sur le
     registre de prédiction P1
commence à décoder instruction 2,
     le prédicat positionné pour pointer sur le
     registre de prédiction P2
si I égale 0, positionner le registre P1 à vrai (1),
              positionner le registre P2 à faux (0)
calculer et délivrer les résultats de toutes les
     instructions dont le prédicat pointe sur le
     registre dont la valeur est vrai (1), en
     l'occurrence P1.
```

Le processeur n'exécute aucun saut (débranchement), il commence à exécuter simultanément les branches `ALORS` et `SINON` comme s'il s'agissait d'instructions en séquence.

IA-64 prévoit 64 registres de prédicat susceptibles de recevoir les valeurs vrai ou faux (0 ou 1). Le champ registre de prédicat de chaque instruction pointe sur un registre de prédicat. Quand le processeur a fini de comparer `I` à `0`, seuls les résultats des instructions qui pointent sur un registre de prédicat qui vaut vrai (1) sont délivrés.

Considérons ce qui se passe : d'abord, il n'y a aucun risque à commencer l'exécution simultanée des branches `ALORS` et `SINON` de ce programme, il n'y a par construction aucune dépendance entre elles puisqu'elles sont exclusives l'une de l'autre. Ensuite nous pouvons observer que notre pseudo–code EPIC ne comporte plus de branchement : ce n'est pas un tour de passe-passe, cette suppression de la plupart des branchements est un fondement du modèle EPIC.

Mais, pourra objecter le lecteur vigilant, en quoi cette suppression des branchements peut-elle être bénéfique pour les performances ? Après tout le processeur EPIC, en commençant à exécuter les deux branches du programme, effectue deux fois plus de travail qu'un processeur classique qui essaie de prévoir quelle branche va être exécutée réellement (et il faut savoir que les processeurs modernes sont capables de faire une prédiction juste dans 90% des cas), et la moitié de ce travail est inutile, puisque finalement seule une des branches sera retenue.

Premier temps de la réponse à l'objection : d'abord exécuter les deux branches en parallèle ne fait pas perdre de temps, puisqu'elles sont indépendantes l'une de l'autre et que notre processeur dispose de tous les bons dispositifs super-scalaires, notamment des unités d'exécution multiples, pour que le parallélisme soit effectif. Ensuite, pour les 10% de cas dans lesquels le processeur classique prédit la mauvaise branche le gain est considérable, puisqu'alors le processeur classique doit tout reprendre au départ. Enfin puisque le texte du programme ne contient plus de branchements il n'est plus divisé en petits blocs `ALORS/SINON`, ce qui autorise les instructions des branches à être placées dans des liasses ou des chaînes de liasses, associées aux instructions précédentes ou suivantes. Associées signifie ici éligibles pour le parallélisme explicite.

9.6.3 Optimisation des accès mémoire : chargement anticipé

Dans le modèle VLIW, comme dans le modèle RISC, les opérandes d'une instruction doivent être chargés dans des registres avant d'être traités. L'exécution efficace de l'instruction `LOAD` est donc un élément crucial de l'architecture. Pour en comprendre les ressorts nous invitons le lecteur à se remettre tout d'abord en mémoire ce qui a été écrit sur la technique du cache à la section 4.6.1.

Les développements sur le cache ont montré l'extrême importance de sa bonne gestion. Si par malheur au moment où le processeur essayait d'exécuter une instruction `LOAD` la zone mémoire recherchée n'était pas dans le cache il en résulterait une pénalité de l'ordre de 20 cycles au moins, autant dire que le fruit de tous nos efforts de parallélisme et de pipe-lining serait anéanti, et au-delà. Il est donc crucial d'éviter de telles situations. Pour ce faire, dès les années 1995, les processeurs les plus rapides (EPIC comme l'Itanium, mais aussi RISC comme l'Alpha ou CISC comme l'AMD Athlon) avaient recours au chargement anticipé dans le cache L1 des zones mémoires nécessaires aux instructions

appelées à s'exécuter dans les quelques dizaines de cycles qui suivaient. C'était le chargement spéculatif, dont l'usage s'est depuis généralisé.

9.6.4 De la programmation VLIW

Arrivé ici, le lecteur sera en droit de penser que les notions qu'il pouvait avoir de la programmation des ordinateurs viennent de recevoir un sacré surcroît de complexité, et il n'aura pas tort. Il pourra alors se dire que cette activité qui lui semblait banale prend enfin des couleurs attrayantes, ou au contraire que c'est trop effrayant et qu'il renonce à son projet de s'y engager.

En fait, ce qui se complique, c'est la programmation en langage machine ou en assembleur. Aujourd'hui pratiquement personne ne programme plus en langage machine, et quand on le fait ce sont des programmes très brefs, comme une séquence d'amorçage *(boot)*, avec une exception : la connaissance du langage machine est indispensable à la lutte contre les virus informatiques et autres logiciels malfaisants, parce que c'est en langage machine que ces logiciels peuvent être repérés, observés, analysés. Quant à la programmation en assembleur, elle est réservée aux auteurs de systèmes d'exploitation et aux auteurs de compilateurs. L'immense majorité des programmes sont écrits de nos jours en langage évolué, et sont traduits en assembleur, puis en langage machine, par un compilateur. Beaucoup de programmes embarqués (à bord d'avions, de voitures, de fours à micro-ondes, de téléphones portables) étaient encore il y a peu écrits en assembleur pour des raisons de performance et d'encombrement : les progrès de la micro-électronique permettent aujourd'hui de les écrire dans des langages hyper-évolués tels que Java ou Ada. Certains compilateurs ne sont pas écrits en assembleur et ne produisent pas directement de l'assembleur : il traduisent le langage source auquel ils sont destinés vers un langage cible de plus bas niveau, qui dispose lui-même d'un compilateur vers l'assembleur, appelé compilateur en *mode natif*.

Pour nous résumer, le lecteur qui se sent irrésistiblement attiré par les subtilités de la programmation en assembleur pour une architecture VLIW doit se tourner vers l'écriture de compilateurs ou de systèmes d'exploitation. Il n'aura pas à le regretter, ce sont des domaines passionnants, en évolution rapide et où le chômage ne menace pas. Le lecteur que la complexité de ces techniques rebuterait peut être rassuré : l'auteur de programmes en langage évolué n'a pas à se préoccuper de détails techniques d'aussi bas niveau, les langages modernes procurent tous à leurs programmeurs un niveau d'abstraction qui leur permet de se consacrer au problème à programmer plutôt qu'aux détails techniques de l'ordinateur.

Chapitre 10 Machines virtuelles et micro-noyaux

Sommaire

10.1 Notion de machine virtuelle

Les chapitres qui précèdent ont montré à plusieurs reprises que la conception d'une architecture informatique consistait le plus souvent en grande partie à organiser des niveaux d'abstraction différents afin de donner une intelligibilité supérieure à un mécanisme, ou au contraire pour le dissimuler aux yeux de l'utilisateur. Nous pouvons subsumer certains de ces artefacts sous la notion de *machine virtuelle*.

L'exemple le plus simple de machine virtuelle est donné par certains langages de programmation interprétés, qui fonctionnent finalement comme des calculettes : l'utilisateur entre au clavier une phrase du langage, l'interpréteur évalue la phrase et donne la réponse. L'interpréteur se présente à son utilisateur comme un ordinateur dont le langage machine serait le langage qu'en fait il traduit à la volée. Il est une machine virtuelle du langage considéré. Basic, Scheme, Python, le shell Unix sont des langages interprétés, par opposition aux langages compilés comme C, Scheme (oui, il peut être les deux), Ada, pour lesquels il faut d'abord écrire le texte du programme jusqu'au bout, puis le soumettre à un traducteur appelé compilateur qui le traduit en langage machine.

Le système d'exploitation présente une métaphore de l'ordinateur qu'il anime à l'utilisateur : celui-ci, en soumettant des phrases du langage de commande (ou en agissant sur les objets de l'interface graphique), agit symboliquement sur des abstractions, et déclenche ainsi des actions réelles d'objets matériels. On peut dire que le système d'exploitation exhibe une machine virtuelle qui représente de façon stylisée et elliptique la machine réelle, dont les aspects les plus sordides sont ainsi dissimulés.

10.1.1 Émulation et machines virtuelles

Assez tôt l'idée s'est fait jour qu'une telle possibilité pourrait être mieux exploitée : puisqu'un système d'exploitation est capable de donner une métaphore de la machine sous-jacente, pourquoi ne pourrait-il pas représenter une autre machine, qu'il simulerait ? Un tel programme est appelé un émulateur, dont on dit qu'il émule la machine simulée. Ainsi aux temps préhistoriques IBM fournissait un émulateur 7090 pour le système 360 qui permettait d'exploiter sur

ce dernier les programmes écrits pour le premier. Des émulateurs d'Unix sur VAX sous VMS ont existé, ainsi que des émulateurs de VMS sous Unix. Apple et d'autres sociétés ont produit des logiciels qui émulent un PC sous Windows sur un Macintosh. L'inverse existe aussi d'ailleurs. Bref, il est ainsi possible d'utiliser le système et les logiciels destinés à un ordinateur que l'on ne possède pas, c'est pratique mais souvent les performances sont assez médiocres parce qu'il faut exécuter en fait le code de trois systèmes : le vrai système de la vraie machine, l'émulateur et le système émulé. Mais au fil des ans les techniques d'émulation ont atteint une perfection diabolique, et il est même possible d'exécuter sous Linux ou MacOS des logiciels de jeux destinés à Windows, exercice particulièrement délicat puisqu'il faut actionner des interfaces graphiques complexes, des manettes de pilotage, *joysticks*, manches à balai [1] etc.

La société *VMware Inc.* de Palo Alto en Californie a développé une technologie très élaborée d'émulation généralisée : son logiciel de machine virtuelle, par exemple, peut fonctionner sur un PC à processeur Intel sous Linux, et accueillir un système hébergé, par exemple Windows. Tous les accès de Windows au matériel sont interceptés de telle sorte que le système hébergé ne puisse pas faire la distinction entre un périphérique réel et l'abstraction présentée par la machine virtuelle. L'émulateur peut d'ailleurs accueillir simultanément plusieurs machines virtuelles tournant sous des systèmes différents. C'est particulièrement utile pour un développeur qui veut tester ses programmes sur plusieurs types de plateformes.

10.1.2 De CP/67 à VM/CMS

La technologie mise en œuvre par VMware remonte en fait à 1967, date de lancement d'une machine dont nous avons déjà parlé à la section 4.4.8, l'IBM 360/67, doté de l'« hyperviseur » CP/67 capable de supporter plusieurs machines virtuelles 360/65 tournant sous des versions de système différentes. On voit bien que ce type de méta-système a été inventé par les développeurs de systèmes (en l'occurrence ceux du *Cambridge Research Lab.* d'IBM) pour se faciliter la tâche en disposant de systèmes de test sans avoir à réclamer à leur management des machines supplémentaires, dont le prix à l'époque se comptait en millions de dollars. CP/67 a engendré VM/CMS, qu'IBM a commercialisé pendant les années 1970-1980. L'idée de CP/67 et de VM/CMS était un micro-système très dépouillé, fournissant des fonctions très élémentaires d'accès au matériel, et laissant la

1. Il faut savoir qu'aujourd'hui ce sont les logiciels de jeux électroniques qui sont le moteur de l'industrie du microprocesseur. Les processeurs actuels sont en effet très suffisants pour satisfaire tout usage professionnel raisonnable en dehors de certains domaines scientifiques assez spécialisés tels que la mécanique des fluides, la sismographie, la météorologie, l'analyse génomique, etc. Mais l'avidité des jeux est sans limite, et le marché des jeux est plus vaste que celui de l'océanographie ou des souffleries numériques.

complexité au système des machines virtuelles accueillies. Ce dépouillement avait d'ailleurs des avantages, à tel point que certains clients utilisaient VM/CMS tout seul, sans machine virtuelle: ils disposaient ainsi d'une machine temps partagé simple et bon marché. C'est peut-être pour cette raison qu'IBM a laissé mourir VM/CMS, qui révélait que la complexité et la lourdeur de ses autres systèmes était peut-être inutile...

10.2 Machines virtuelles langage : l'exemple Java

La notion de machine virtuelle allait connaître un nouvel avatar avec le lancement par Sun Microsystems du langage Java en 1995, dont le concepteur principal était James Gosling. Rarement langage aura suscité un tel engouement, justifié non pas par sa syntaxe, assez banalement héritée de C débarassé de ses traits de bas niveau tels que les pointeurs et augmentée d'une couche simple de gestion des objets (ce qu'aurait dû être C++), ni par ses performances en termes de vitesse de calcul, assez faibles pour les premières versions (cela s'est bien amélioré, avec l'aide des progrès des microprocesseurs), mais justement par sa machine virtuelle. Signalons trois autres traits novateurs de Java:

- Il comporte un système de gestion des *threads*[2] dans le langage; les *threads* font l'objet de la section suivante, mais disons qu'il s'agit d'un moyen de faire de la multi-programmation à l'intérieur d'un processus utilisateur. Les *threads* de Java sont une réalisation des moniteurs de Hoare[59] (cf. p. 305).
- Le jeu de caractères standard de Java est Unicode, ce qui autorise des identifiants en toutes sortes d'écritures, par exemple l'écriture coréenne Hangul.
- Java comporte un glaneur de cellules (*garbage collector*, GC) qui assure la gestion automatique de la mémoire, c'est-à-dire que le GC implémente un algorithme heuristique d'allocation de pages mémoire lorsque l'exécution du programme l'exige, et de libération de ces pages lorsqu'elles ne sont plus utilisées (cf. p. 98 au chapitre qui traite de la mémoire et de sa gestion). Ainsi le programmeur n'a plus à se préoccuper d'effectuer « manuellement » des `malloc` et des `free` de ses zones mémoire, sources d'erreurs de programmation particulièrement vicieuses.

L'exécution d'un programme Java obéit à un enchaînement d'opérations inhabituel. Le texte du programme source est d'abord soumis à un compilateur,

2. Désolé, mais aucune des traductions proposées pour *thread* ne me semble satisfaisante: activité, fil d'exécution, processus léger...

qui produit un résultat dans un langage intermédiaire appelé *bytecode*. Ce langage intermédiaire est le langage d'une machine virtuelle Java (JVM), et il suffit de soumettre le *bytecode* à l'interpréteur de la JVM pour que le programme s'exécute[3].

Tout l'intérêt de la démarche réside dans le fait que la JVM est normalisée, c'est-à-dire qu'un programme Java compilé peut être exécuté tel quel sur n'importe quelle plate-forme *(compile once, run anywhere)*, à la différence d'un programme en langage classique qui doit subir une compilation spécifique pour chaque architecture de machine cible et chaque système. La JVM est conçue pour être peu encombrante et sécurisée. Votre navigateur Web comporte une JVM embarquée. Ainsi, un serveur Web peut envoyer un petit programme Java (en anglais *applet*, soit en français appliquette, ou aplète) à votre navigateur Web et celui-ci saura l'exécuter sur votre machine. Cela s'appelle du code mobile et en 1995 c'était quand même assez révolutionnaire. Signalons qu'en 2018 Java est en désuétude sur la plupart des navigateurs, parce que Html 5 et JavaScript procurent des fonctions équivalentes de façon plus simple et plus standardisée.

Cette possibilité pour un serveur d'exécuter du code sur une machine distante pose certes des problèmes de sécurité. Pour éviter les malveillances les plus évidentes la JVM exécute les aplètes dans un « bac à sable » *(sandbox)* qui les isole de l'environnement local. En fait Java a causé moins d'incidents de sécurité que de plantages de navigateurs ou de systèmes un peu fragiles ou que de chargements interminables de pages HTML passionnantes. La technologie est certes perfectible. En fait la principale cause du désenchantement relatif que subit depuis quelque temps Java réside dans la politique assez opaque du propriétaire de la technologie (aujourd'hui Oracle, qui a racheté Sun), qui modifie arbitrairement les règles du langage, de son implémentation et du copyright afférent à ses API, comme en témoigne un procès contre Google, dont le système Android est écrit en Java. Ces péripéties détournent les développeurs. Et le premier charme épuisé on redécouvre cette chose déplorablement banale : réaliser les calculs sur un serveur central correctement administré a quand même beaucoup d'avantages.

Quoi qu'il en soit, Java et sa JVM connaissent un grand succès parce que ce système est assez facile à installer dans toutes sortes de petits processeurs qui peuplent les téléphones portables, les machines à laver, les routeurs d'appartement, les voitures, les caméscopes, les cartes à puce, etc., et cela renouvelle entièrement la programmation de ce genre d'objets, qui était auparavant de la magie noire. Les applications destinées au système mobile Android reposent pour la plupart sur un système Java doté d'une machine virtuelle particulière, naguère

3. Cette technique de production d'un langage intermédiaire interprété n'est pas inédite, elle a été utilisée par Pascal UCSD, LeLisp, Emacs Lisp, CAML...

Dalvik, aujourd'hui remplacé par ART (abréviation de *Android Runtime*). Les *threads* permettent même la réalisation de mini-systèmes d'exploitation en Java. Signalons aussi l'apparition de compilateurs pour d'autres langages que Java vers le *bytecode*, ce qui diversifie les moyens de créer des aplètes. Ainsi le compilateur Bigloo permet d'écrire ses aplètes en Scheme. Mentionnons également Scala, langage qui produit du *bytecode* JVM à partir d'une syntaxe proche de Java mais plus concise et plus élégante.

10.3 Machines virtuelles système

La présente section est consacrée aux machines virtuelles qui permettent d'émuler un système informatique complet (par opposition aux machines virtuelles langage, comme celle de Java, décrite ci-dessus).

10.3.1 Que veut-on virtualiser ?

Ainsi que nous l'avons vu lors des premiers chapitres de ce livre, on peut résumer le rôle du système d'exploitation en disant qu'il est de présenter à l'utilisateur d'un calculateur (utilisateur qui peut d'ailleurs être un autre programme) une vue simplifiée et stylisée du calculateur sous-jacent. En effet, le système d'exploitation d'un ordinateur physique doit effectuer des opérations sur des disques durs, écrans, imprimantes et autres dispositifs matériels, dits périphériques, dont il existe une grande variété. Il est indispensable d'interposer une couche d'abstraction entre ces matériels et l'utilisateur : je peux recopier le texte de mon programme sur mon disque dur sans savoir combien celui-ci possède de pistes et combien il peut stocker de caractères par piste. Le système d'exploitation me cache ces détails, qui n'ont aucun intérêt pour moi, mais que l'ingénieur qui écrit le sous-programme du système chargé d'écrire sur le disque doit connaître (ce sous-programme se nomme un pilote d'appareil périphérique, en anglais *driver*). Incidemment, la prise réseau est un périphérique comme un autre.

De même, le système d'exploitation me cache les méthodes complexes grâce auxquelles je peux exécuter simultanément sur mon ordinateur plusieurs logiciels : naviguer sur le Web, y copier des données pour les recopier dans la fenêtre du traitement de texte, imprimer un autre texte, etc.

En fait, tous les autres programmes sont des sous-programmes du système d'exploitation.

10.3.2 Pratique des machines virtuelles

Développons les idées de la section 10.1.1 ci-dessus pour décrire les usages modernes des machines virtuelles système.

Dans le cas simple, sans recours à la virtualisation, il y a un ordinateur, sur cet ordinateur est installé un système d'exploitation, et ainsi on peut exécuter des programmes sous le contrôle de ce système d'exploitation, qui s'interpose entre le programme et le matériel de l'ordinateur. Le matériel, ce sont des appareils qui sont capables d'émettre et de recevoir des signaux électroniques qui commandent leur fonctionnement.

Maintenant, instruit par les chapitres précédents, je peux écrire un logiciel qui se comporte en tout point comme un autre ordinateur (qui le simule, ou qui l'émule, selon le jargon en usage). Et ce simulacre d'ordinateur peut accueillir un système d'exploitation, qui lui même permettra l'exécution de logiciels. Ce logiciel qui fait semblant d'être un ordinateur, on l'appelle une machine virtuelle. Sur un ordinateur physique, je peux ainsi avoir plusieurs machines virtuelles qui émulent plusieurs autres ordinateurs physiques et/ou plusieurs autres systèmes d'exploitation. C'est très pratique pour les usages suivants :

- Tester un nouveau système sans mobiliser un ordinateur à cet effet.
- Avoir plusieurs systèmes actifs simultanément sur le même ordinateur : Linux, OpenBSD, Windows…
- Il est même possible d'avoir une machine virtuelle qui simule un ordinateur physique d'un modèle différent de la machine d'accueil, avec un jeu d'instructions (une architecture matérielle) différent(e).
- Comme les machines virtuelles ont des système d'exploitation différents, deux logiciels qui fonctionnent sur deux machines virtuelles différentes sont mieux isolés l'un de l'autre que s'ils coexistaient sous le contrôle du même système, ce qui a des avantages en termes de sécurité.
- Une machine virtuelle est constituée de logiciel, et aussi des données nécessaires à son fonctionnement, telles que paramètres du système et données des utilisateurs, enregistrées sur support persistant. L'ensemble constitué du texte du logiciel de la machine virtuelle et des données persistantes associées constitue l'image physique de la machine virtuelle, recopiable, transférable par le réseau.
- Une machine virtuelle, c'est du logiciel et des données, donc il est possible de la recopier comme un document ordinaire ; ainsi, avec la virtualisation, l'administration des systèmes devient plus facile : déplacer un serveur, c'est déplacer un fichier, le sauvegarder, c'est copier un fichier sur une clé USB, doubler un serveur, c'est recopier un fichier, cela prend quelques secondes et quelques clics de souris, sans se déplacer en salle machine (on dit maintenant centre de données) ; c'est sur ce principe que repose l'informatique dans les nuages *(cloud computing)*, qui consiste à créer à la demande de nouvelles machines virtuelles et à les déplacer par le réseau sur les machines physiques les moins chargées, par exemple en Nouvelle-Zélande pour profiter du décalage horaire.

– Bien sûr, pour tout ce qui est enseignement, travaux pratiques, expériences, c'est très commode.

10.3.3 Différents niveaux de virtualisation

Comme la virtualisation repose sur du logiciel qui simule du matériel, l'imitation peut se faire de diverses façons, selon le niveau d'indépendance souhaité pour nos machines virtuelles :

– S'il faut simplement des systèmes isolés les uns des autres sur la même machine physique, il existe des systèmes de cloisonnement qui procurent à chaque logiciel serveur un environnement qui donne l'illusion de disposer d'une machine privée : *containers* Linux, *jail* FreeBSD. Le système *Docker* est une réalisation assez réussie de *containers* Linux, qui peuvent même fonctionner sous Windows, et qui peuvent être multipliés et déployés dans les nuages grâce au système d'orchestration Kubernetes, par exemple. Il y a en fait sur chaque machine physique un seul système d'exploitation en service, mais chaque serveur fonctionne comme s'il était seul, ce qui lui confère une sécurité accrue (sauf défaillance du logiciel).

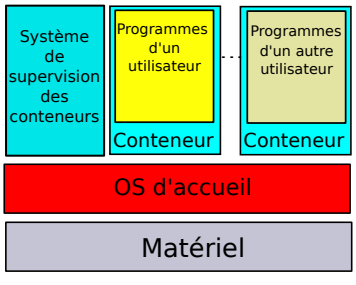


Figure 10.1 : *Système de cloisonnement (virtualisation simplifiée).*

– S'il faut vraiment des machines virtuelles distinctes, mais toutes sur le même modèle de processeur (même architecture matérielle), il faudra interposer entre le matériel et les différentes copies du système d'exploitation un logiciel de simulation, mais les opérations élémentaires seront néanmoins effectuées par le matériel sous-jacent, ce qui évitera la grande diminution des performances qui serait entraînée par la simulation en logiciel des dites opérations. VMware, Xen, KVM, Citrix, Microsoft Hyper-V Server sont de tels systèmes, nommés *hyperviseurs*. Les hyperviseurs sont en fait des systèmes d'exploitation allégés de beaucoup des fonctions qui seront dévolues aux OS hébergés. Les processeurs modernes sont équipés

de dispositifs matériels qui facilitent leur exécution (Intel VT et AMD Pacifica).

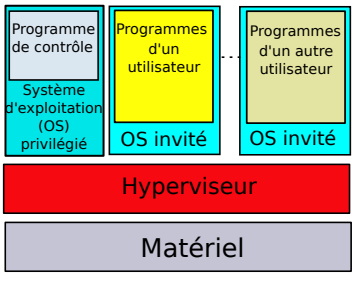


Figure 10.2 : *Système hyperviseur (virtualisation sur architecture identique).*

– Ce n'est que si l'on veut simuler sur un ordinateur physique d'architecture A une machine virtuelle d'architecture B qu'il faudra simuler sur A, par du logiciel, le jeu d'opérations élémentaires de B, ce qui aura un coût élevé en termes de performances. QEMU est un logiciel libre d'émulation de processeur, qui offre ce type de possibilité, par l'intermédiaire d'un hyperviseur tel que Xen ou KVM.

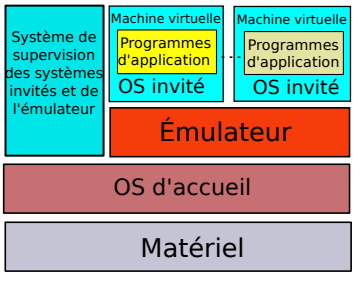


Figure 10.3 : *Système de virtualisation complète.*

– Toutes ces machines virtuelles peuvent bien sûr communiquer entre elles et avec le vaste monde par un réseau... virtuel évidemment ! mais qui doit néanmoins établir des passerelles, voire des ponts, avec le réseau réel, par l'intermédiaire de routeurs et de commutateurs virtuels : tous les systèmes de virtualisation modernes fournissent ce type d'accessoire, dès lors que l'on sait virtualiser, on peut tout virtualiser.

Comme une machine physique, une machine virtuelle peut être « démarrée » et « arrêtée » ; dans ce cas il s'agira en réalité du lancement d'un programme et de son arrêt.

10.3.4 Administration d'un système informatique

Pour maintenir en état de marche correct un ordinateur (réel ou virtuel), un certain nombre de tâches doivent être effectuées régulièrement, notamment :

- mise à jour du système d'exploitation et des logiciels à partir des nouvelles versions fournies par les éditeurs ;
- mise à jour des bases de données des systèmes anti-virus et anti-intrusion ;
- mise à jour de la base de données des utilisateurs autorisés pour tenir compte des arrivées et des départs ;
- consultation quotidienne des journaux d'incidents ;
- sauvegarde des données ;
- vérification de la disponibilité d'un espace de stockage de données suffisant ;
- application des corrections de sécurité pour supprimer les vulnérabilités connues.

Ces opérations (la liste n'est pas complète), que nous désignerons du terme d' « administration système », peuvent être en partie automatisées, elles sont bien sûr moins absorbantes pour un poste de travail personnel que pour un serveur avec des dizaines d'utilisateurs directs, ou que pour un serveur Web ouvert à tous les publics, mais elles constituent une part importante du travail des ingénieurs système, que les serveurs soient virtuels ou des machines physiques.

10.3.5 Administration d'un système virtuel éteint

Les opérations d'administration du système d'une machine physique supposent que la machine soit en marche : en effet, la consultation des paramètres du système, leur modification, les copies de fichiers supposent que soient actifs un éditeur de texte, et quelques commandes du système, et que l'on ait accès aux données persistantes sur les disques durs, à tout le moins. Machine éteinte, rien ne serait possible.

Il en va tout autrement pour une machine virtuelle : nous avons vu qu'en fait elle était constituée de logiciels et de données hébergées sur un ordinateur physique. De ce fait, machine virtuelle arrêtée - et à condition bien sûr que la machine physique sous-jacente, elle, ne soit pas arrêtée - il est possible, grâce au système d'exploitation et aux logiciels de la machine physique d'accueil, d'accéder aux paramètres de son système et à ses données. Si l'on en connaît le format et l'organisation, on pourra effectuer les opérations d'administration. Enfin, c'est plus vite dit que fait.

Trois ingénieurs de *CA Technologies* à Hyderabad en Inde et à Datchet en Angleterre, Nishant Thorat, Arvind Raghavendran et Nigel Groves, ont

mis en œuvre une telle solution d'administration, et ils ont écrit dans les *Communications of the ACM*[4], un article qui la décrit. Ils ont tiré parti du fait que les principaux éditeurs de systèmes de virtualisation se sont entendus sur des formats de données publiés. La *Distributed Management Task Force* (DMTF)[5] a publié l'*Open Virtualization Format*, ou OVF, une spécification adoptée par les principaux éditeurs (tels que Citrix, Microsoft et VMware) et acceptée comme norme en août 2010 par l'*American National Standards Institute* (ANSI)[6].

Les avantages d'une telle solution sont patents : à l'heure de l'informatique en nuage, où les machines virtuelles se propagent aux quatre coins de l'Internet et s'y reproduisent à qui mieux mieux, leur appliquer « au vol » un plan de maintenance est bien plus difficile que pour un parc de machines physiques sagement rangées dans les armoires d'un centre de données. Il est plus facile de travailler sur l'image physique qui a servi à engendrer toutes ces machines virtuelles, et qui est généralement stockée dans un endroit centralisé bien identifié.

Nos auteurs ne manquent pas de souligner que cette solution présente aussi des inconvénients (mineurs) : certaines opérations qui nécessitent l'observation de la machine en marche ne sont pas possibles, et il reste du travail à faire pour que l'on puisse utiliser les mêmes procédures d'administration sur les machines arrêtées et sur les machines en marche.

10.3.6 Déplacer une machine virtuelle dans le réseau

Créer des machines virtuelles et les déplacer est facile et utile, aussi les grands centres d'hébergement administrent-ils des dizaines (voire des centaines) de milliers de machines virtuelles sur les milliers de serveurs physiques, et au gré des besoins il faut les déplacer d'un groupe de serveurs à un autre.

Pour organiser le réseau au sein du centre d'hébergement, on a le choix entre la couche 2 (Ethernet) ou la couche 3 (IP).

Sur un réseau local Ethernet

Organiser le réseau du centre d'hébergement au niveau de la couche 2 (Ethernet) est le plus facile à administrer… jusqu'à la catastrophe.

- explosion du domaine de diffusion (broadcast) ;
- calcul laborieux de l'algorithme de l'arbre recouvrant (spanning tree), qui permet de déterminer une topologie sans boucle dans le réseau ;

4. *Communications of the ACM*, Vol. 56 No. 4, Pages 75-81
5. `http://dmtf.org/`
6. `http://ansi.org/`

– explosion du nombre de Réseaux virtuels (VLAN) ;
– ou promiscuité dangereuse de niveau 2.

Dans un réseau IP avec du routage

Pour éviter les inconvénients de la couche 2, on peut organiser le réseau du centre d'hébergement au niveau de la couche 3 (IP). Là, plus de problème de promiscuité, on a un vrai réseau avec du routage, ce qui, il faut le noter, demande de vraies compétences réseau. Avec un réseau de couche 3, il est même possible d'étendre l'espace de répartition des machines virtuelles à d'autres sites, géographiquement distants.

Cette solution de couche 3 a néanmoins des inconvénients :

– lorsqu'une machine virtuelle se déplace d'un sous-réseau à un autre, elle change d'adresse IP, ce qui est gênant pour de nombreuses applications ;
– un réseau routé demande un niveau de compétence plus élevé pour les administrateurs ;
– le routage consomme des ressources de calcul et d'infrastructure.

Combiner la couche 2 et la couche 3 ?

Le protocole VXLAN *(Virtual eXtensible Local Area Network)*[7] permet, comme son nom l'indique, de transporter de l'Ethernet (couche 2) dans de l'IP (couche 3), ce qui vise à résoudre les difficultés évoquées aux deux sections précédentes, notamment lorsqu'il s'agit de déplacer une machine virtuelle dans le réseau sans casser toutes ses connexions réseau.

Les trames Ethernet sont encapsulées dans des datagrammes IP. On leur ajoute une entête VNI *(Virtual Network Identifier)*, ce qui simule un « câble virtuel ». Le protocole de résolution d'adresses (ARP) fonctionne sur un réseau multicast. VXLAN est à l'état de *draft* à l'IETF.

10.4 Machines virtuelles applicatives

10.4.1 Un logiciel, une VM, un OS sur mesure, compilés ensemble, en langage fonctionnel

Peut-on aller plus loin ? C'est ce que nous expliquent, dans un article des *Communications of the ACM* [83] intitulé *Unikernels: The Rise of the Virtual Library Operating System*, Anil Madhavapeddy et David J. Scott, qui travaillent sur le sujet depuis une dizaine d'années, du côté de Cambridge (UK) et chez Citrix. Si

7. Cf. note 6 p. 226 et `http://en.wikipedia.org/wiki/Virtual_Extensible_LAN`

on n'est pas abonné aux CACM, on pourra lire en ligne une version préliminaire de cet article[8].

Puisque l'on peut multiplier les machines virtuelles à volonté, sans coût important, sans manipulation physique et sans délai, on peut en profiter pour avoir une VM pour chaque logiciel utilisé, pour chaque application métier par exemple.

Puisque chaque OS invité n'exécute qu'une seule application, est-il nécessaire qu'il soit muni de tous les dispositifs ultra-complexes destinés à garantir l'étanchéité des espaces de mémoire et de données de chaque application, tout en leur permettant de communiquer entre elles lorsqu'il le faut ? Ces fonctions d'isolation et de communication seront assurées, bien plus efficacement et bien plus simplement, par l'hyperviseur. Il est donc possible de les retirer du système.

Puisqu'en outre cet OS n'invité n'aura pas à piloter toute une variété de dispositifs physiques complexes et changeants, mais seulement quelques périphériques simulés ultra-simplifiés et stables, il sera allégé des fonctions correspondantes.

Et puisque seront éliminés la plupart des risques liés aux accès directs au matériel et à la cohabitation de logiciels entre lesquels il faut éviter les interférences, il ne sera plus utile d'avoir une distinction entre le mode superviseur et le mode utilisateur, ni entre la mémoire du noyau et l'espace mémoire des utilisateurs.

Après toutes ces simplifications, les OS invités pourront se présenter sous forme de simples bibliothèques de fonctions, qui seront compilées et liées avec les logiciels d'application.

Et tant qu'à faire, on écrira toutes ces bibliothèques dans un langage fonctionnel de haut niveau, ce qui facilitera considérablement le développement, et réduira le risque d'apparition de vulnérabilités telles que les débordements de buffer, inévitables en programmation de bas niveau, et toujours en tête du hit-parade des CERT.

En l'occurrence, le langage choisi par nos auteurs est OCaml[9], un logiciel libre dont, soit dit en passant, l'équipe de conception et de développement est née en France autour de Xavier Leroy.

10.4.2 Unikernel : avantages et inconvénients

La technologie qui consiste à compiler une application avec les morceaux de système d'exploitation dont elle a besoin, et uniquement ceux-là, se nomme *Unikernel* ou *library operating system* (libOS). Nos auteurs citent les premières avancées sur ce terrain à la fin des années 1990, Exokernel [51] et Nemesis [78].

8. `http://anil.recoil.org/papers/2013-asplos-mirage.pdf`
9. Cf. `http://fr.wikipedia.org/wiki/Ocaml`

Un des avantages majeurs du fait d'avoir l'application et les fonctions système dans le même espace mémoire, sans séparation des privilèges, consiste à éviter d'avoir à copier sans cesse des données de l'espace utilisateur à l'espace noyau et en sens inverse. De plus, les applications ont accès directement aux dispositifs matériels, sans l'intermédiaire du noyau, ce qui améliore les performances.

En contrepartie, expliquent nos auteurs, Nemesis et Exokernel ont rencontré deux obstacles majeures : la difficulté d'isoler chaque application de ses voisines, et la nécessité d'écrire des pilotes pour chaque dispositif matériel nouveau. Or, justement, ces deux difficultés sont résolues par le recours aux machines virtuelles, puisque c'est l'hyperviseur qui fournira les pilotes et qui assurera le cloisonnement des VM. C'est ce dont ont tiré parti nos auteurs, en profitant des progrès des techniques de virtualisation, et aussi de ceux des processeurs, qui permettent aujourd'hui d'utliser ces techniques avec des performances décentes.

10.4.3 MirageOS

Anil Madhavapeddy et David J. Scott ont baptisé leur système *MirageOS*, un nom qui convient bien à un OS destiné à animer des machines virtuelles dans les nuages.

La construction d'une application avec MirageOS commence avec la création d'un graphe de dépendances pour identifier les ressources nécessaires. En effet, une VM classique embarque un système d'exploitation complet, sans oublier un serveur Web, un système de gestion de bases de données et un système de gestion de fenêtres, qui doivent chacun lire leurs fichiers de configuration au démarrage pour s'initialiser, alors que seule une petite partie de leurs fonctions seront utilisés par l'application utilisée. Le but d'un Unikernel tel que MirageOS est d'élaguer ces processus pour ne charger et configurer que les fonctions utiles. C'est d'ailleurs pourquoi les fichiers de configuration sont inclus d'emblée dans le graphe de dépendances, et chargés lors de la compilation de l'application. Il en va de même des parties utiles du noyau, disponibles sous formes de bibliothèques dans les entrepôts de code source OCaml. Signalons par exemple que la bibliothèque d'exécution d'OCaml comporte une pile TCP/IP complète, ce qui signifie que la machine virtuelle MirageOS n'aura à demander les services réseau de l'hyperviseur qu'au niveau de la couche 2 (Ethernet).

À la date de rédaction de l'article, MirageOS comporte une centaine de bibliothèques (en *open source*) qui réalisent un large éventail de fonctions attendues du noyau d'un système d'exploitation. Son portage dans des systèmes commerciaux, tels que XenServer de Citrix, est en cours.

Il est difficile de prévoir l'avenir de MirageOS : l'histoire de l'informatique est pavée d'excellentes idées (et d'excellentes réalisations) supplantées par des systèmes bien moins bons. Une vision aussi révolutionnaire aura à surmonter le conservatisme des entreprises et, surtout, de leurs ingénieurs. Mais il ne fait

aucun doute que les évolutions récentes de l'informatique, notamment en nuage, sont un appel à ce type de solutions.

10.5 Informatique en nuage *(Cloud Computing)*

10.5.1 Une véritable innovation technique

L'informatique en nuage (en anglais *Cloud Computing*, traduit *infonuagique* par les Canadiens francophones) est un service d'hébergement informatique en réseau dont la première apparition fut le lancement par Amazon de son offre *Amazon Web Services* (AWS) en 2006. Il s'agissait alors pour Amazon de commercialiser la puissance de calcul inutilisée des serveurs déployés de par le monde pour son propre usage, et qui n'étaient utilisés qu'à 10% de leur capacité, afin de pouvoir faire face aux points saisonnières, notamment lors des fêtes de fin d'année..

L'idée d'une offre de services informatiques détachée, grâce au réseau, des caractéristiques techniques de son implémentation avait été formulée quelques années plus tôt, par exemple par des chercheurs tels que Michel Volle [10].

L'originalité de l'informatique en nuage par rapport aux offres traditionnelles d'hébergement de données, de sites Web ou de serveurs de calcul repose sur les cinq caractéristiques suivantes :

- déploiement et arrêt des services à la demande, en self-service, généralement par une interface Web, quasi instantanément ;
- accès par réseau à haut débit ;
- mutualisation de ressources non localisées : infrastructures, réseau, logiciel, stockage ;
- allocation et désallocation rapide des ressources (« élasticité ») ;
- facturation à la consommation, typiquement heure par heure.

Cette souplesse est permise par la disponibilité de quatre technologies déjà bien connues, mais dont les performances ont accompli récemment des progrès considérables : l'informatique distribuée, un réseau à haut débit omniprésent, le système de noms de domaines (DNS), et des plates-formes efficaces pour machines virtuelles :

- la nécessité d'un réseau rapide et omniprésent est évidente ;

10. `http://www.volle.com/ouvrages/e-conomie/table.htm`, `http://www.volle.com/ouvrages/informatique/informatique1.pdf`

– la disponibilité de systèmes efficaces de virtualisation, dont une analyse détaillée sera donnée ci-dessous, elle permet de déployer facilement, et même dans certains cas automatiquement, de nouveaux serveurs à la demande, alors que s'il s'agissait de machines physiques il y faudrait touteune logistique de transport, de distribution d'énergie et d'infrastructure réseau ;
– l'usage de techniques perfectionnées de gestion du DNS confère à cette répartition dans l'espace (physique et topologique) la souplesse nécessaire ;
– une fois que l'on a déployé de nombreuses machines virtuelles, les principes de l'informatique distribuée sont indispensables pour les faire coopérer de façon cohérente [11].

10.5.2 Trois formes pour l'informatique en nuage

– IaaS *(Infrastructure as a service)* : le client se voit livrer une machine (virtuelle) nue, c'est-à-dire sans système d'exploitation installé, mais avec de l'espace disque et une ou plusieurs interfaces réseau (virtuelles) ; il installe sur cette machine le système et les logiciels de son choix, et fait son affaire des mises à jour, de sécurité notamment ;
– Paas *(Platform as a service)* : le client reçoit une machine virtuelle dotée du système d'exploitation qu'il aura choisi sur le catalogue du fournisseur, ainsi que de quelques programmes utilitaires (base de données, serveur Web par exemple) ; c'est le fournisseur qui assurera les mises à jour des logiciels qu'il aura installés, cependant que le client sera responsable de la gestion des données et des logiciels d'application qu'il aura installés lui-même ;
– Saas *(Software as a service)* : le client reçoit les droits d'accès à un système entièrement configuré avec les logiciels choisis sur le catalogue du fournisseur (par exemple paie, messagerie, blog, wiki ou gestion financière), il n'a plus qu'à les utiliser avec ses propres de données.

Grâce à la virtualisation des serveurs et du réseau, l'utilisateur de services en nuage ne sait où se trouvent ni ses données, ni l'ordinateur qui les exploite, et d'ailleurs leur emplacement physique peut changer à tout instant, même en cours de travail.

La plupart des services en réseau destinés au grand public ou aux entreprises, tels que les Google Apps, Facebook, Dropbox, etc., fonctionnent en nuage : on ne sait où sont ni les données, ni les ordinateurs qui les créent et qui les transforment.

11. Le texte de référence inégalé sur les principes de l'informatique distribuée reste le livre de Sir Charles Antony Richard Hoare [60].

10.5.3 Répartir les services en nuage grâce au DNS

Le système de noms de domaines (DNS) est une technologie complexe, dont le principe est très simple(cf. p. 167), c'est celui de l'annuaire téléphonique : on cherche un nom, le DNS répond avec le « numéro IP » qui correspond au nom. Ainsi, au nom `www.laurentbloch.net` correspond le numéro (on dit plutôt l'adresse) IP 194.15.166.220.

Il est possible au propriétaire du nom `www.laurentbloch.net` de configurer le serveur DNS qui fait autorité pour son domaine `laurentbloch.net` de sorte qu'au lieu de répondre par une adresse unique, il réponde par une collection d'adresses, qui correspondent chacune à un serveur (physique ou virtuel) distinct. Ainsi il sera possible de répartir les demandes d'accès au service entre les différents serveurs, ce qui lui procurera une sécurité accrue et une répartition de la charge de travail.

10.6 Les *threads*

10.6.1 Séparer le fil d'exécution des ressources allouées

Le chapitre 3 a défini la notion de processus et décrit le comportement de tels objets ; le chapitre 4 a expliqué la gestion de la mémoire : en fait le processus et son espace-adresse, que nous avons séparés pour la commodité de l'exposé, sont indissociables. Le processus, disions-nous, est une entité active constituée d'un programme en cours d'exécution et des ressources qui lui sont nécessaires : temps de processeur, fichiers ouverts, espace de mémoire. Nous pouvons isoler conceptuellement l'aspect « programme en cours d'exécution » en le désignant comme un *fil*, au sens de « fil de la conversation » (en anglais *thread*). Le terme français « fil » n'est pas très commode pour traduire *thread* parce que son pluriel a une graphie ambiguë. L'équipe du projet Chorus a lancé le mot « activité », qui ne convient guère (à mon avis), et nous utiliserons *thread* (avec un peu de culpabilité). Les attributs qui relèvent des *threads*, par opposition aux ressources, sont les suivants :

- le compteur ordinal (ou compteur de programme, **eip** sur processeur Intel) ;
- l'état courant (actif, dormant, prêt ou terminé) ;
- les registres ;
- la pile.

Le processus possède en outre les ressources suivantes :

- espace adresse ;
- variables globales ;

– fichiers ouverts ;
– processus fils (fils comme filiation) ;
– interruptions en cours ou en attente ;
– données comptables.

Il résulte de cette définition que le processus est une entité complexe, décrite par des tables nombreuses et encombrantes : table des pages à plusieurs niveaux, table des fichiers ouverts, etc. Une commutation de contexte entre processus implique notamment la commutation d'espace adresse, donc la remise à zéro du TLB, ce qui a un impact négatif lourd sur les performances de la pagination. Le *thread*, comme nous allons le voir, est plus léger et maniable, au prix d'une plus grande promiscuité avec les *threads* rattachés au même processus.

10.6.2 Définition des *threads*

De ces considérations naquit l'idée qu'il serait bien d'avoir des processus avec plusieurs fils d'exécution, ou, si l'on voit la question sous l'angle opposé, d'avoir des « processus légers » qui n'auraient pas besoin de recevoir toutes ces ressources neuves à leur lancement et qui se contenteraient de celles déjà possédées par leur processus père, qu'ils partageraient avec leurs frères [12]. Bref, cette démarche conduit à la naissance d'une entité spéciale, un sous-processus en quelque sorte, le *thread* :

– Le *thread* appartient à un processus.
– Un processus peut avoir plusieurs *threads*.
– Les *threads* d'un même processus sont en concurrence pour l'accès au temps de processeur, elles s'exécutent en pseudo-simultanéité comme nous l'avons vu pour les processus concomitants.

12. Incidemment, cette idée n'est pas entièrement inédite. À une époque reculée (1968) IBM avait lancé un moniteur transactionnel nommé CICS *(Customer Information Control System)* ; le travail d'un tel logiciel consistait à recevoir des requêtes en provenance de milliers de terminaux, par exemple dans une banque, à interroger ou mettre à jour la base de données, à donner la réponse au terminal. Bien sûr il faut éviter que les mises à jour concomitantes ne se télescopent et veiller à ce que les temps de réponse ne soient pas trop catastrophiques. Pour des raisons de simplicité de portage sur des systèmes différents, CICS était réalisé comme un unique gros processus (*task* dans le language d'IBM) qui réalisait lui-même (avec une efficacité modeste) le partage du temps et des ressources entre les différents travaux attachés aux terminaux. Aux dernières nouvelles CICS est toujours au catalogue d'IBM, des milliers de sociétés sont spécialisées en CICS, 30 millions de personnes sont assises derrière des terminaux CICS, et cela continue à croître. CICS avait été conçu dans les années 1960 comme une solution temporaire en attendant que soit terminée la réalisation d'un SGBD plus ambitieux (IMS, *Information Management System*), aujourd'hui largement délaissé et oublié même si toujours au catalogue...

– Un *thread* s'exécute dans l'espace adresse du processus auquel elle appartient : si un *thread* modifie un octet de la mémoire, tous les *threads* de ce processus voient la modification immédiatement, comme s'ils l'avaient effectuée eux-mêmes. Un *thread* peut notamment modifier la pile d'un de ses frères !
– Tous les *threads* d'un processus partagent donc le même espace adresse, et ils voient les mêmes fichiers ouverts : si un *thread* lit des données dans un fichier séquentiel, ce qui fait progresser le curseur du fichier (voir section 5.2.2) jusqu'à la position qui suit les données juste lues, un autre *thread* qui lira le même fichier obtiendra les données suivantes, sauf si entre temps le fichier a été fermé et réouvert.

10.6.3 Avantages procurés par les *threads*

Les *threads* ainsi définies offrent de nombreux avantages : par exemple les navigateurs Web font du *multi-threading*, ce qui permet d'avoir plusieurs fenêtres ouvertes dans lesquelles se chargent simultanément plusieurs pages de données, sans encourir le prix de lancement de multiples processus ; on observe bien que la première fenêtre s'ouvre beaucoup plus laborieusement que les suivantes. Et des processus distincts ne feraient pas l'affaire, parce qu'ils ne pourraient pas partager de façon simple le cache mémoire et le cache disque, ni les fichiers de signets. En outre, comme chaque *thread* dispose de sa propre pile et de son propre état, s'il émet des demandes d'entrée-sortie bloquantes (qui le mettent en attente), un autre *thread* pourra prendre le contrôle sans que le programme ait à prévoir quoi que ce soit de particulier, et la pseudo-simultanéité sera assurée « naturellement »[13].

Le principal avantage reste la performance : créer et démarrer un *thread* demande jusqu'à cent fois moins de temps que la création d'un processus.

10.6.4 Implémentation des *threads*

Pour réaliser les *threads*, le concepteur de système a le choix entre deux possibilités : en mode utilisateur ou en mode noyau.

Threads en mode utilisateur

Les *threads* sont des objets purement internes au processus, dont le noyau n'a pas à être informé. Le processus gère totalement le partage des ressources et

13. Dans son livre de système [22] Samia Bouzefrane observe que les *threads* ou processus légers lancés dans le contexte d'un processus représentent une bonne réalisation des moniteurs de Hoare [59].

la pseudo-simultanéité entre les *threads*. Le système procure au processus une bibliothèque d'outils pour assurer cette gestion, notamment pour implanter les piles et les tables d'état des *threads*.

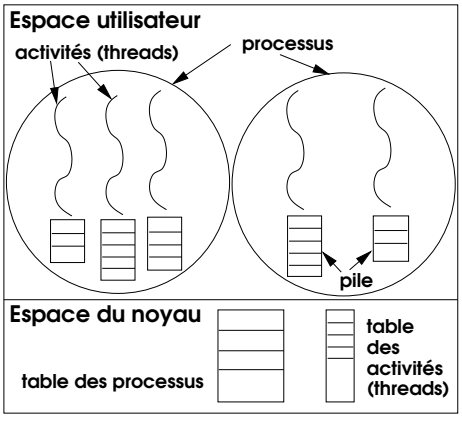


Figure 10.4 : Threads *en mode noyau.*

Threads en mode noyau

Le noyau gère les *threads* : création, ordonnancement, arrêt. Cela consomme plus de ressources que les *threads* en mode utilisateur, mais seule la gestion des *threads* en mode noyau permet le traitement correct des appels système bloquants dans un *thread* (par exemple une demande d'entrée-sortie), cas où il faut pouvoir ordonnancer un autre *thread*.

10.6.5 Inconvénients des *threads*

Les *threads* permettent une amélioration des performances de certains logiciels, au prix d'une complexité plus grande pour le développeur. Le principal problème soulevé est celui des variables globales. Un programme d'une certaine taille, nous l'avons signalé au chapitre 3 p. 46, est subdivisé en sous-programmes. Nous avons évoqué les variables au chapitre 4 p. 94. Les variables locales d'un sous-programme sont affectées sur la pile si elles sont petites (elles tiennent dans un mot), sinon elles sont affectées sur le tas mais repérées par un pointeur maintenu sur la pile. Comme nous avons dit que chaque *thread* disposait de sa propre pile, il n'y a en principe pas de risque qu'un *thread* accède à une variable locale appartenant à une autre. Mais le problème est plus difficile à résoudre pour les variables globales, visibles de tous les sous-programmes. Le problème est le même pour les accès aux fichiers. Dans ces deux cas, les précautions à prendre sont à la charge du développeur.

D'autres soucis incombent à l'auteur du système : certains sous-programmes de bibliothèque peuvent ne pas être réentrants, c'est-à-dire que leur comportement sera erroné s'ils sont appelés par un programme alors que le précédent appel par un autre programme n'est pas terminé. La cause la plus courante de ce type de problème vient de ce que le sous-programme a des variables réservées « en dur » dans son texte (au lieu d'être allouées dynamiquement sur le tas) et que ces variables reflètent l'état du calcul du premier appel, ce qui sème la confusion lors du second. Pour être réentrant, c'est-à-dire apte à être utilisé en une copie unique par deux appels concomitants, un programme doit posséder un texte invariable, c'est-à-dire non susceptible d'être modifié lors de son exécution. La désignation des objets externes qu'il manipule doit être extérieure à son texte. En bref, tout ce qui est variable doit être dans les registres, sur la pile, ou atteint au moyen d'une indirection passant par la pile ou par un registre (incidemment, sous Unix par exemple, les arguments qu'il reçoit du programme appelant sont sur la pile).

Ainsi, le sous-programme de bibliothèque qui exécute un appel système pour allouer une zone de mémoire sur le tas (`malloc` sous Unix) peut, lorsqu'il travaille dans un univers à fil d'exécution unique, laisser temporairement des tables de pointeurs dans un état incohérent, puisque s'il est interrompu ce sera par un autre processus, avec un autre espace adresse. Cette assertion cesse de tenir dans un univers à *threads* multiples... La solution consiste bien sûr à n'écrire que du code réentrant pour les sous-programmes de bibliothèque, mais le monde ne saurait être parfait...

10.7 Micro-noyaux

Dans les systèmes tels que ceux que nous avons évoqués jusqu'à présent, le noyau désigne la partie du système d'exploitation dont la présence en mémoire réelle est indispensable et qui est commune à tous les autres logiciels. Dans un système tel qu'Unix, cet ensemble est monolithique (même si des Unix modernes tels que Linux ont introduit des modules chargeables dynamiquement en cours de fonctionnement) et il est devenu assez encombrant parce qu'y ont été incorporées par commodité des fonctions assez variées.

L'idée de micro-noyau est née dans les années 1980 ; il s'agit de réduire au minimum le contenu du noyau, et de placer en dehors, c'est-à-dire dans l'espace utilisateur, tout ce qui peut l'être. Les avantages de cette démarche semblent évidents : l'interface du micro-noyau sera plus simple que celle d'un macro-noyau, et permettra de ce fait la construction d'un système plus modulaire ; il est possible d'implanter au-dessus du micro-noyau des serveurs qui utilisent ses services pour exhiber le comportement, les caractéristiques et la sémantique de systèmes divers, comme Unix, Windows, MacOS etc., ce qui rejoint la notion de

machine virtuelle et en facilite la réalisation ; en cas de panne d'un des serveurs, le système continue à fonctionner, ce qui facilite grandement tous les travaux de développement et de mise au point de système tout en augmentant la tolérance aux pannes et la capacité de redémarrage à chaud.

L'idée était aussi que tout cela fonctionne en réseau : chaque machine physique serait animée par un micro-noyau, et les serveurs en mode utilisateur seraient répartis au gré des opportunités, ce qui ouvre la voie à la construction d'architectures distribuées souples et modifiables dynamiquement.

Un peu comme l'hyperviseur de VM/CMS, un micro-noyau typique fournit des services minimum d'accès au matériel : création de *threads*, gestion minimale des interruptions, commutation de contexte, verrouillage élémentaire de sections critiques, accès brut à la mémoire et aux entrées-sorties, protection réciproque des *threads*. Les fonctions plus raffinées (ordonnancement de processus, gestion de mémoire virtuelle, système de fichiers, protocole de réseau...) sont déléguées à des serveurs en espace utilisateur.

Les communications entre le micro-noyau et les serveurs sont réalisées classiquement par passage de message, une technique qui a l'avantage de pouvoir fonctionner aussi bien par le réseau que localement. Le message contient les informations nécessaires au traitement d'un événement tel qu'une interruption, et il est déposé dans une zone de mémoire connue comme boîte à lettres, où le serveur concerné peut le récupérer. Ce mécanisme a l'avantage de fournir un bon moyen d'abstraction et de séparation des fonctions, il a aussi deux inconvénients : jusqu'à des développements récents ses performances étaient médiocres, et dès lors que le système est réparti sur plusieurs sites en réseau il est difficile de restituer ainsi la sémantique d'un noyau Unix.

Les premiers micro-noyaux célèbres furent Mach, développé à l'Université Carnegie-Mellon à Pittsburgh, Chorus, créé en France en 1979 par une équipe de l'INRIA avec notamment Hubert Zimmerman et Michel Gien, Amoeba créé à l'Université Libre d'Amsterdam par Andrew Tanenbaum.

10.7.1 Chorus

Chorus est né en 1979 comme projet de recherche à l'INRIA près de Paris. La société Chorus Systèmes a été créée en 1986 pour commercialiser une nouvelle version de ce noyau (version 3), et elle a été rachetée par Sun Microsystems en septembre 1997. Entre temps le système Chorus a connu un succès assez modéré dans le monde informatique proprement dit, mais beaucoup plus significatif dans celui des constructeurs de matériel téléphonique, dont les autocommutateurs sont en fait de vastes systèmes informatiques distribués qui semblent faits pour les micro-noyaux.

Le noyau Chorus décompose la notion de processus selon les deux axes que nous avions déjà mis en évidence à la section 10.6 :

– un axe « fil d'exécution », auquel correspondent des entités appelées *threads* ;
– un axe « ressources allouées », notamment l'espace adresse, auquel correspondent des entités appelées acteurs ; un acteur peut avoir une ou plusieurs activités qui partagent ses ressources, conformément au modèle de la section 10.6.

Chorus présente ses abstractions avec une terminologie qui témoigne, au moins dans les articles rédigés en français tels que « UNIX et la répartition : retour à la simplicité originelle ? » [8] dont les lignes qui suivent sont largement inspirées, d'un réel souci d'énonciation des concepts dans un langage précis et expressif :

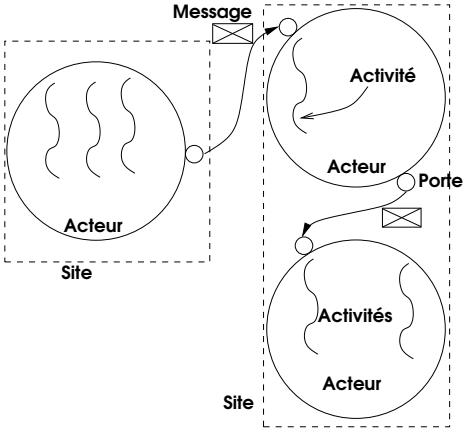


Figure 10.5 : *Abstractions du micro-noyau Chorus*

– l'acteur, unité d'allocation de ressources ;
– l'activité *(thread)*, unité d'utilisation de processeur ;
– le message, collection de données susceptible d'être envoyée ou reçue par une porte ;
– la porte *(port)*, adresse logique à laquelle des messages peuvent être envoyée ;
– le site, qui désigne une machine éventuellement reliée à d'autres sites par un système de communication (réseau ou bus).

Au-dessus du noyau les créateurs de Chorus ont développé un ensemble de serveurs destinés à constituer un sous-système Unix, tel que représenté par

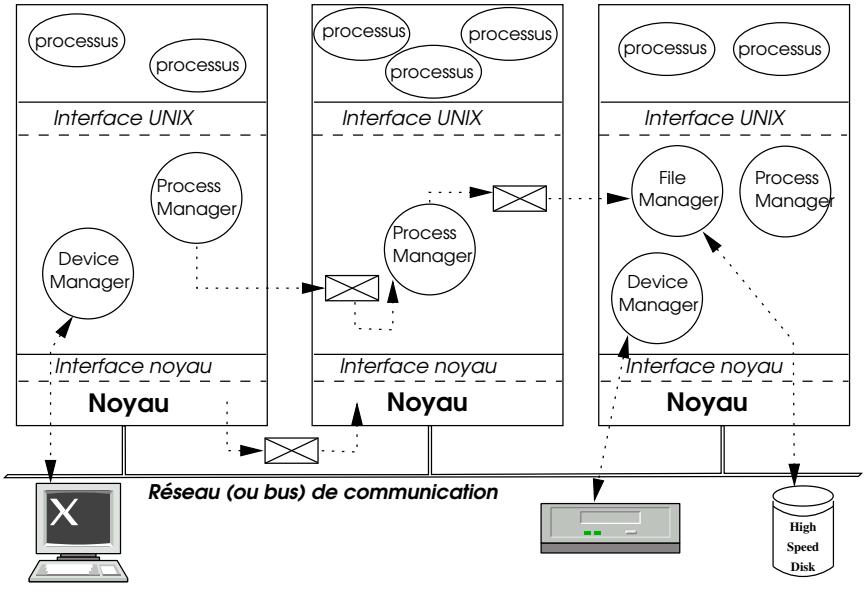


Figure 10.6 : *Serveur UNIX sur un groupe de sites Chorus*

la figure 10.6. Chaque type de ressource du système (processus, fichier...) est géré par un serveur système dédié. Les serveurs peuvent résider sur des sites différents : on voit que Chorus, encore plus que Mach, a été conçu dans la perspective de construction de systèmes répartis. Ce découpage du noyau Unix en serveurs modulaires était très intéressant tant du point de vue du génie logiciel (l'art de construire de grands systèmes informatiques) que de celui de l'architecture. L'entreprise a (partielllement) achopé sur la difficulté à restituer la sémantique du noyau Unix dans un contexte non monolithique, ainsi que sur des problèmes de performances. Tant que l'on en reste à un site unique, Chorus implémente des appels de procédures à distance (RPC, pour *Remote Procedure Call*) légers, mais cette solution n'est disponible que si l'on renonce à la répartition, et pour des processus en mode superviseur. La réalisation efficace de communications inter-processus par passage de messages ne viendra qu'après le rachat de Chorus par Sun, et ce sera une autre équipe qui s'en acquittera.

Après le rachat par Sun, certains membres de l'équipe Chorus créèrent en 2002 la société Jaluna, renommée en 2006 VirtualLogix. Le code source de ChorusOS a été publié sous licence libre par Sun et complété par Jaluna.

10.7.2 Mach

Mach est né en 1983 comme un projet de recherche de l'Université Carnegie Mellon dont les idées majeures sont exprimées dans une communication à la conférence USENIX de 1986, *Mach: A New Kernel Foundation for UNIX Development* [4]. Ce projet a suscité de grands espoirs, au point que la DARPA a à cette époque réorienté vers lui une part importante des financements qui allaient auparavant au *Computer Systems Research Group (CSRG)* de l'Université de Californie à Berkeley. Les objectifs de Mach sont les suivants :

- développer le parallélisme tant pour le système que pour les applications ;
- permettre l'usage d'espaces adresse vastes et éventuellement répartis sur le réseau, avec des dispositifs souples de mémoire partagée ;
- assurer un aspect transparent au réseau ;
- assurer la compatibilité avec les systèmes existants (Unix BSD notamment) et la portabilité.

Le noyau Mach ignore la notion classique de processus, qu'il désarticule selon les deux axes que nous avions déjà mis en évidence à la section 10.6 :

- un axe « fil d'exécution », auquel correspondent des entités appelées sans surprise *threads* ;
- un axe « ressources allouées », notamment l'espace adresse, auquel correspondent des entités appelées tâches (*tasks*) ; une tâche contient une ou plusieurs activités qui partagent ses ressources, conformément au modèle de la section 10.6.

Les abstractions de base du noyau Mach sont les suivantes (figure 10.7 :

- la tâche, unité d'allocation de ressources ;
- le *thread*, unité d'utilisation de processeur ;
- le flux *(port)*, canal de communication unidirectionnel ;
- le message, collection de données susceptible d'être envoyée ou reçue dans un flux ;
- l'objet de mémoire, unité interne de gestion de la mémoire.

Le projet Mach à l'Université Carnegie-Mellon s'est arrêté en 1994, mais il a une postérité réelle. L'Université d'Utah a repris le flambeau pendant quelques années. Le projet GNU *Hurd* vise à remplacer le noyau Unix par une collection de serveurs implantés au-dessus d'un noyau Mach. Le système MacOS-X d'Apple, et dans une certaine mesure le noyau de feu-Tru64 Unix de Compaq (ex-Digital) sont des descendants plus ou moins légitimes du micro-noyau Mach au-dessus duquel Apple et Compaq ont implanté des systèmes Unix de sensibilité plutôt BSD.

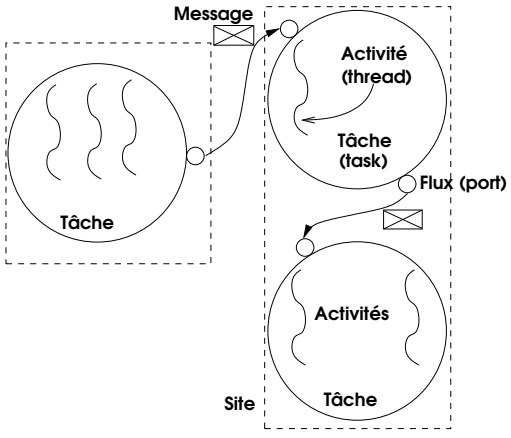


Figure 10.7 : *Abstractions du micro-noyau Mach*

10.7.3 Eumel, L3, L4

La série des systèmes Eumel, L3 et L4, développés à partir de 1977 par Jochen Liedtke à l' Université de Bielefeld, puis à partir de 1984 au GMD *(Gesellschaft für Mathematik und Datenverarbeitung)*, à partir de 1996 au centre de recherche Thomas J. Watson d'IBM et à partir de 1999 à l'Université de Karlsruhe, représente un des efforts les plus notables à la fois dans le domaine des micro-noyaux et dans celui des domaines persistants (cf. section 5.4). La disparition prématurée de Jochen Liedtke n'en a pas marqué la fin, L4 et ses avatars Fiasco, L4Ka::Pistachio et SeL4 poursuivent l'aventure, avec quelques usages industriels.

Dans son article de 1995 *On μ-Kernel Construction* [79] Liedtke pose les fondations de la « seconde génération » des micro-noyaux. Il part de la constatation que certains pionniers ont été conduits à réintégrer au noyau des fonctions qui en avaient été extraites vers le mode utilisateur, essentiellement pour des raisons de performances. Il examine donc les différents problèmes de performances rencontrés et entreprend de leur trouver des solutions qui respectent le programme initial des micro-noyaux: une architecture de système modulaire, des serveurs en mode utilisateur pour toutes les fonctions dont l'appartenance au noyau n'est pas conceptuellement indispensable, la possibilité d'implanter des stratégies variées de gestion du système en mode utilisateur.

Le premier problème de performance examiné concerne le coût de la commutation mode noyau – mode utilisateur, qui avec un noyau Mach 3 sur un antique processeur Intel 486 à 50 MHz consomme 900 cycles de processeur. Avec le noyau L3 Liedtke abaisse ce coût à 180 cycles dans le cas le plus défavorable (3 adresses non résolues par le TLB, 10 fautes de cache). L3 implémente les *threads* en mode noyau et les processus persistants, ceci pour dire que c'est un noyau

complet et non pas une simple maquette dont la rapidité aurait pour contrepartie des fonctions rudimentaires.

Liedtke examine ensuite la question de la commutation des espaces adresses. Comme nous l'avions noté à la section 4.4.5, cette opération est beaucoup plus rapide sur les processeurs qui utilisent un TLB étiqueté *(tagged TLB)* comme le MIPS R4000 ou l'Alpha, que sur les processeurs comme le Pentium ou le Motorola PowerPC qui doivent réinitialiser le TLB à chaque commutation d'espace adresse. Pour les processeurs de cette dernière catégorie, Liedtke propose d'utiliser les registres de segment, inutilisés par la plupart des systèmes d'exploitation contemporains, pour simuler un TLB étiqueté et éviter ainsi la réinitialisation du TLB. Le coût de la commutation d'espace adresse descend ainsi à moins de 50 cycles, ce qui autorise 100 000 commutations par seconde sans baisse de performance insupportable.

Le passage de messages, et de façon générale les communications interprocessus, souvent incriminées, peuvent être accélérées par la réaffectation de cadres de pages d'un espace-adresse à un autre, ce qui évite la recopie physique de zones mémoire, un grande source de gaspillage de cycles et de saturation du cache. Incidemment cette technique est aussi de nature à améliorer les performances de l'accès au réseau. Ces opérations doivent bien sûr être effectuées sous le contrôle exclusif du noyau, ne serait-ce que pour des raisons de sécurité.

Une des conclusions tirées par Liedtke de ses expériences, c'est qu'un micro-noyau, pour être efficace, ne doit pas être conçu pour être indépendant du matériel, mais doit au contraire exploiter au mieux les caractéristiques du processeur auquel il est destiné. Ainsi, le passage en mode noyau coûte un minimum de 156 cycles sur Intel 486, contre 20 cycles sur MIPS R2000, qui profite de son TLB étiqueté et d'une zone mémoire réservée au noyau non affectée à l'espace adresse de l'utilisateur : il est clair que la conception d'un noyau pour ces deux processeurs aura à tenir compte de cette différence. Pour le 486, Liedtke a été amené à organiser l'espace adresse de manière à réduire le nombre d'accès au TLB par tous les moyens, par exemple concentrer les données les plus cruciales du noyau en une page.

10.7.4 Conclusion sur les micro-noyaux

Jusqu'au milieu des années 1990, aucune conférence informatique peu ou prou orientée vers les systèmes ne pouvait avoir lieu sans plusieurs communications consacrées aux micro-noyaux. Il semblait évident à tout le monde que c'était la technologie du lendemain. Aujourd'hui on n'en parle plus guère.

Les qualités intrinsèques du modèle du micro-noyau, son adaptation à des systèmes distribués et flexibles, sa capacité à fonctionner sur des plateformes de toutes dimensions, ainsi que les améliorations apportées par les travaux de la

seconde génération, ceci combiné avec les attraits des systèmes persistants (cf. section 5.4), me donnent à penser que cette technologie réapparaîtra, sans doute combinée avec celle du code mobile à la Java.

Chapitre 11 Micro-informatique

Sommaire

11.1 Naissance et essor d'une industrie

Depuis les années 1970 des efforts considérables ont été déployés pour produire des ordinateurs accessibles économiquement et techniquement à des personnes privées. L'ensemble des idées, des techniques, des matériels et des logiciels mis en œuvre pour atteindre cet objectif a été appelé micro-informatique. Aujourd'hui le succès en est si complet que l'on peut se demander s'il ne faudrait pas dire informatique tout court: en effet les ordinateurs qui ne sont pas destinés à un usage individuel sont plutôt l'exception, on les appelle généralement des serveurs. Depuis 1993 les micro-ordinateurs offrent une puissance de calcul équivalente à celle des grands ordinateurs traditionnels, et depuis le lancement

en 2007 de l'iPhone sous système iOS, aussitôt suivi par Android de Google, on peut en dire autant des téléphones.

Si l'on a pu parler de « révolution micro-informatique », ce n'est pas à cause d'un bouleversement de la technique architecturale, car les micro-ordinateurs fonctionnent selon les mêmes principes que les serveurs et respectent l'architecture de von Neumann avec une orthodoxie plutôt plus stricte que ceux-ci. Les aspects techniques révolutionnaires résident plutôt dans l'interface personne-ordinateur, avec tout un déploiement d'interfaces graphiques et de périphériques adaptés à des usages individuels. Par exemple, une vraie nouveauté fut l'invention de la mémoire vidéo, qui associe par une interface physique l'écran du micro-ordinateur et une zone mémoire spéciale de telle sorte que toute modification de celle-ci s'affiche instantanément sur celui-là.

La micro-informatique fut en tout cas sans conteste une vraie révolution économique et sociale puisqu'elle a fait d'une machine isolée dans un sanctuaire soigneusement protégé et desservie par une caste de professionnels au langage incompréhensible un bien de consommation courante en vente dans les grands magasins et utilisé par toutes les catégories sociales (à l'exclusion, comme l'a souligné Michel Volle dans son livre *E-conomie* [136], des élites intellectuelles, économiques et politiques françaises, qui manifestent leur supériorité en proclamant ne jamais avoir touché un clavier ou une souris).

La « révolution » manifeste un signe avant-coureur dès novembre 1960, avec la livraison du premier PDP-1 de Digital Equipment, un ordinateur individuel très bon marché pour l'époque : 120 000 dollars ! La firme Digital Equipment avait été fondée peu de temps auparavant par l'ingénieur Ken Olsen et allait contribuer de façon notable à l'évolution de l'industrie informatique en lançant l'idée de mini-ordinateur, une machine plus simple et moins chère que les grands systèmes de l'époque, bien que réalisée en logique discrète (le microprocesseur n'existe pas encore). Les gammes PDP successives allaient permettre bien des innovations, notamment le développement du contrôle informatique de processus industriels et la création du système Unix. DEC sera pendant les années 1980 le second constructeur informatique mondial derrière IBM et emploiera alors 110 000 personnes dans le monde entier. Ken Olsen ignorera deux phénomènes importants : la naissance du micro-ordinateur et le succès d'Unix, alors que sa société avait été associée à leur genèse. DEC périclitera dans les années 1990, sera racheté par Compaq, lui-même racheté en 2001 par Hewlett-Packard, et sa dernière gamme de processeurs, Alpha, sera cédée à Intel et abandonnée malgré des qualités techniques éminentes.

En novembre 1971 se produit l'événement qui va permettre la naissance de la micro-informatique : Intel produit le premier microprocesseur, le 4004. En effet, l'unité centrale du PDP-1 (sans parler de celle d'un gros ordinateur de l'époque) comporte de nombreux éléments électroniques et électriques reliés entre eux par des fils, un assemblage fait à la main par des techniciens hautement qualifiés, et

de ce fait très cher. Le microprocesseur rassemble en un seul élément (une puce, *chip* en anglais) ces nombreux circuits indépendants, et il est entièrement fabriqué par des machines automatiques, ce qui permet des prix de vente compris entre quelques Euros et quelques centaines d'Euros.

Dès 1972 les Français André Truong Trong Thi et François Gernelle conçoivent le micro-ordinateur Micral autour du microprocesseur Intel 8008[1]. En janvier 1975, MITS (*Model Instrumentation Telemetry Systems*, fondée à Albuquerque, au Nouveau-Mexique, par Ed Roberts) lance l'Altair 8800, un micro-ordinateur basé sur l'Intel 8080 et vendu en kit par correspondance pour 400 dollars.

Pour simplifier l'architecture, l'Altair possède un circuit unique nommé « bus » pour faire circuler les données entre le processeur, la mémoire centrale et les organes périphériques. Cette idée vient des PDP. Le connecteur du bus de l'Altair a 100 broches et se répand dans l'industrie sous le nom de bus S-100.

L'Altair dispose d'un traducteur du langage BASIC écrit par Bill Gates et Paul Allen, dont la société de fait « Micro-soft » apparaît ainsi en 1975.

Steve Wozniak et Steve Jobs fondent Apple le 1er avril 1976. Le 1er juillet, la carte Apple I est mise en vente au prix de 666,66 dollars. L'année suivante sort l'Apple II, basé sur le processeur Rockwell 6502 (initialement créé par *MOS Technology*) et qui comporte l'autre innovation technique importante de la micro-informatique après le microprocesseur : une mémoire vidéo directement commandée par le processeur et reliée à un moniteur à balayage analogue à un écran de télévision. Tous les éléments visuels sont enregistrés dans la mémoire vidéo, ce qui permet des affichages à l'écran très rapides, indispensables aux programmes graphiques. Le boîtier de l'Apple II comporte un fond de panier avec des connecteurs raccordés au bus dans lesquels viennent s'enficher les cartes de mémoire ou de contrôleurs de périphériques. Cette architecture matérielle fera la versatilité, la souplesse et le succès de la micro-informatique.

D'autres constructeurs lancent des micros basés sur le 6502 : Commodore, Oric, Atari, Acorn, tandis que Tandy Radio Shack utilise le Zilog Z80 pour son TRS-80.

Parallèlement au développement du matériel apparaît un des premiers systèmes d'exploitation capable de fonctionner sur plusieurs types de micros : CP/M (pour *Control Program/Microcomputers*) écrit par Gary Kildall dès 1973. Nous en reparlerons à la section suivante.

En 1979, un étudiant du MIT et de Harvard du nom de Dan Bricklin invente un nouveau type de logiciel, le *tableur* (le sien s'appelle Visicalc). Ce programme affiche sur l'écran une feuille de calcul avec des lignes et des colonnes et des

1. Sur toute cette aventure de la naissance du micro-ordinateur, et plus particulièrement sur l'épisode français, on consultera avec profit le livre d'Henri Lilen [80].

nombres dans les cases. Le programme permet de dire que telle case doit recevoir la somme des nombres contenus dans telle colonne, par exemple. L'intérêt du tableur, c'est que si l'on modifie un des nombres de la colonne, la case qui contient la somme est automatiquement modifiée, ce qui permet d'examiner rapidement toute une série d'hypothèses, pour préparer un budget par exemple. L'invention du tableur permet l'arrivée du micro dans le monde de la gestion.

Le 12 août 1981 IBM lance son micro, le PC, basé sur le 8088 d'Intel et le système MS-DOS de Microsoft, et doté d'un fond de panier et d'un bus similaires à ceux de l'Apple II. Cet événement fait sortir la micro-informatique du cercle des amateurs et des précurseurs vers le monde industriel et commercial, et permet à IBM de dominer ce nouveau marché dès 1983.

Mais des innovations techniques, cette fois dans le domaine du logiciel, vont encore modifier la face de la micro-informatique. Elles sont issues du PARC (Palo Alto Research Center) de Xerox. Steve Jobs visite le PARC en 1979, Bill Gates en 1980, ils y découvrent une technique révolutionnaire de communication entre l'homme et la machine. En 1981 Xerox lance le 8010 *Star*, premier système à offrir une interface à fenêtres, icônes, menus et souris. Apple lance en 1983 Lisa, un micro qui utilise ces techniques. Ce sont des échecs parce que Star et Lisa sont trop chères, mais en 1984 sort le Macintosh, version améliorée et moins chère de Lisa (2495 dollars). Le « Mac » présente sur son écran une métaphore du bureau ; des « icônes » figurent des dossiers, que l'on peut ouvrir ou fermer en « cliquant » dessus avec un dispositif de pointage, la souris. Dans les dossiers, des documents, qui lorsque l'on clique dessus s'ouvrent pour s'afficher dans des « fenêtres » qui ressemblent à des pages de papier posées sur le « bureau ». Cette interface graphique, fruit de recherches en psychologie et en informatique entreprises par Alan Kay à l'Université d'Utah en 1970 et poursuivies au PARC, va permettre aux « personnes ordinaires » de se servir d'un ordinateur sans trop souffrir.

Ce succès est complété par la sortie en 1985 d'une imprimante à laser de petite taille (le modèle 3800 d'IBM, sorti en 1976, pèse plusieurs centaines de kilos et coûte des centaines de milliers de dollars) dotée d'un langage de programmation graphique, PostScript, et du premier réseau de micros, deux innovations inspirées des travaux du PARC. Ces nouveautés qui mettent à la portée de presque tous des possibilités naguère réservées aux grandes entreprises et aux universités riches donnent à Apple plus de dix ans d'avance technique sur les ordinateurs d'IBM.

Si l'IBM PC est techniquement peu innovant, il a d'autres qualités : il est construit d'éléments standard en vente dans toutes les bonnes boutiques d'électronique et les spécifications de son architecture sont publiées dans une brochure. Dès juin 1982 Columbia Data Products fabrique un « clone » de PC qui marque les débuts d'une immense industrie.

L'essor prodigieux du micro stimule l'industrie du logiciel, à moins que ce ne soit l'inverse. Ce sont le traitement de texte, les tableurs et les bases de données qui assurent l'essentiel de l'activité, sans oublier les jeux.

IBM et Microsoft cherchent à combler leur retard sur l'interface homme-machine du Mac, mais ils divergent sur la méthode et, en 1987, IBM lance OS/2 et Microsoft, Windows 2. De ce schisme date la fin de la suprématie d'IBM sur le marché du PC, dominé désormais par le couple Intel-Microsoft, qui ont le monopole respectivement du processeur et du système d'exploitation.

Au milieu des années 1990, le marché des matériels micro-informatiques représente avec près de 100 milliards de dollars 60% du marché du matériel informatique. Les micros à base de processeur Intel représentent plus de 90% du parc, ils ont désormais des performances comparables à celles des stations de travail techniques pour un prix bien inférieur. Les pays du sud-est asiatique produisent une grande proportion des matériels les moins chers. Les interfaces « à fenêtres » sont d'un usage général.

En ce qui concerne l'équipement des micro-ordinateurs, la capacité de la mémoire centrale et des disques ne cesse de croître. Les lecteurs de CD-ROM, les modems et les cartes vidéo aptes au multimédia se généralisent.

Le logiciel micro-informatique est une activité dominée par Microsoft, mais cette hégémonie suscite procès et réactions, comme le développement des logiciels libres, avec notamment des Unix libres pour micro-ordinateurs (Linux, FreeBSD).

11.2 Quel système pour les micro-ordinateurs ?

Le principal problème à résoudre pour faire fonctionner les premiers micro-ordinateurs était la faible quantité de mémoire disponible. L'auteur de ces lignes a connu ses premières expériences personnelles en 1968 avec un Olivetti Programa 101, puis durant les années 1970 avec un Wang 2200 (des Wang Laboratories, créés par An Wang en 1951). Ce modèle lancé en 1972 disposait de 4K (4096 octets) de mémoire et d'un système d'exploitation qui était en fait une machine virtuelle destinée au langage BASIC, enregistrée en ROM (*Read-Only Memory*, c'est-à-dire une mémoire incorporée au matériel de telle sorte que le contenu en soit inaltérable). Les supports de données externes étaient des cassettes pour magnétophone portable (les disquettes de 8 pouces apparaîtraient plus tard). Avec de telles caractéristiques, les créateurs de micro-ordinateurs ont dû réinventer toutes les techniques de l'informatique classique de vingt ans auparavant, et il n'était pas question d'envisager des solutions luxueuses telles que la mémoire virtuelle ou la multi-programmation. Par contre ces concepteurs venaient souvent du monde de l'instrumentation électronique et ils avaient fréquemment des idées très ingénieuses pour traiter

les interruptions et faire circuler les données, ce qui eut pour conséquence que ces petits ordinateurs furent rapidement plus agiles que les gros en matière de réseau et de télécommunications.

11.2.1 Élégie pour CP/M

Nous avons déjà signalé dans les chapitres précédents que si les premiers systèmes d'exploitation, et certains qui sont encore en activité, furent écrits en assembleur pour des raisons de performance et de réduction de l'encombrement mémoire, assez tôt l'idée apparut de les écrire en langage évolué afin d'accélérer le développement, de faciliter la maintenance et d'améliorer l'intelligibilité du code produit. Le précurseur en ce domaine fut le constructeur Burroughs avec son modèle B 5000, sorti en 1962 avec un système écrit en Algol 60, suivi de Multics écrit en PL/1 et d'Unix en langage C. Le monde micro-informatique allait connaître très tôt une évolution similaire, au destin avorté.

Dès 1972, un expert indépendant sous contrat pour Intel, Gary Kildall, créait un langage évolué pour le microprocesseur 8008, baptisé PL/M ; en fait, Kildall effectuait son service militaire en tant qu'enseignant d'informatique à la *United States Naval Postgraduate School* de Monterey en Californie, il avait acheté pour 25 dollars un processeur Intel 4004, et il avait commencé à programmer cet appareil assez bizarre. Très vite il entra en contact avec l'équipe Intel, basée à quelques kilomètres, et il travailla avec eux pendant son jour de congé hebdomadaire. Comme rétribution de ce travail il reçut un système de développement complet (c'est-à-dire tout ce qu'il faut autour du processeur pour le programmer, soit les composants d'un micro-ordinateur, objet non encore inventé à l'époque), d'abord pour le 8008, puis pour le tout nouveau 8080.

PL/M était un langage d'assez bas niveau qui empruntait ses traits morphologiques et syntaxiques à PL/1 et Algol. Pour développer le compilateur PL/M, Kildall créa un système d'exploitation, CP/M : faire les choses dans cet ordre fut sans doute un exemple unique ! PL/M devint pour une vingtaine d'années le langage de programmation standard chez Intel.

À peu près à la même époque IBM avait inventé la disquette pour son usage interne[2]. Assez vite l'ingénieur à l'origine de cette invention, Alan Shugart, créa une entreprise pour commercialiser la chose, avec grand succès. Kildall eut l'idée d'utiliser ce nouveau type de périphérique comme unité de stockage de données pour micro-ordinateur et d'adapter CP/M en conséquence.

La sortie commerciale de CP/M eut lieu en 1975. Le premier fabricant d'ordinateurs à l'utiliser fut IMSAI. CP/M connut un succès important et rapide.

2. Je me rappelle l'avoir vu apparaître dans la console d'un IBM 370/155, où elle servait à charger le micro-code (voir section 9.4.7 p. 279).

Bientôt il supporta les disques durs. La société de Kildall prit le nom de *Digital Research*.

CP/M était un système mono-utilisateur à mémoire réelle, sans mémoire virtuelle. Les parties du système qui dépendaient du matériel étaient soigneusement isolées, et les caractéristiques des éléments matériels répertoriées dans des tables, de telle sorte que l'adaptation à un nouveau modèle de processeur ou de disque puisse être réalisée aussi simplement que possible. Cette organisation permit à CP/M d'être adapté avec succès pour les microprocesseurs les plus répandus de cette époque, à mots de 8 bits ou de 16 bits, tels que le Zilog Z80, l'Intel 8086 ou le Motorola 68000.

L'architecture de CP/M comporte trois sous-systèmes : CCP *(Console Command Processor)* qui régit les interactions avec l'utilisateur par l'intermédiaire d'un interpréteur de commandes (on ne rêve pas encore d'interfaces graphiques) ; BDOS *(Basic Disk Operating System)* qui gère les disques et le système de fichiers ; BIOS *(Basic Input/Output System)* contient les pilotes de périphériques et gère les aspects matériels de bas niveau, spécifiques d'une machine particulière. Typiquement le BIOS n'est pas fourni par Digital Research mais par le fabricant de l'ordinateur, ce qui annonce la pratique toujours en vigueur pour les PC à base de processeur Intel (voir section 9.4.7 et 11.2.2) ; il fournit une couche d'abstraction du système par rapport au matériel.

CP/M procure un outil supplémentaire d'abstraction des dispositifs matériels, avec la notion d'unité logique, qui permet de dissimuler à l'utilisateur les détails techniques trop horribles pris en charge par le système.

Digital Research produisit une évolution de CP/M nommée MP/M, dotée d'un noyau multi-tâche multi-utilisateur, avec des possibilités de temps partagé.

À la fin des années 1970 Digital Research fut confronté à un dilemme : si PL/M restait un langage bien adapté pour l'écriture du système d'exploitation, il fallait choisir un langage de développement pour les logiciels d'application, BASIC n'étant pas approprié. L'hésitation était entre Pascal, un langage prisé des universitaires mais dépourvu de certaines caractéristiques nécessaires à la programmation de grands logiciels (essentiellement la possibilité de construire de grands programmes à partir de modules compilés séparément, ainsi que des fonctions de traitement de fichiers et de chaînes de caractères), et PL/1, le langage qu'IBM essayait, sans grand succès d'ailleurs, de promouvoir auprès de ses clients.

PL/1 était un langage dérivé d'Algol et doté d'une grande richesse fonctionnelle, trop grande d'ailleurs, ce qui rendait la réalisation de compilateurs difficile et incitait les programmeurs inexpérimentés à produire des logiciels dépourvus de la sobriété qui fait les programmes robustes. Pour pallier la difficulté d'adapter le langage à de petits ordinateurs, le comité de normalisation de PL/1 avait défini un sous ensemble plus raisonnable, PL/1 *Subset G*, qui était un excellent langage.

C'est pour PL/1 *Subset G* que Digital Research commença en 1978 à développer un compilateur, qui fut prêt en 1980. La société avait signé son arrêt de mort.

En 1981, IBM lançait son propre micro-ordinateur, le PC. Il était construit autour du processeur Intel 8088, une version dégradée du 8086 : le 8086 avait des mots de 16 bits et un bus de données d'une largeur de 16 bits, le 8088 des mots de 16 bits mais un bus de 8 bits. Les stratèges de la compagnie ont certainement compris que CP/M, doté d'un vrai langage de développement, PL/1 *Subset G*, et qui évoluait vers un système d'exploitation multi-utilisateurs, MP/M, représentait à terme une concurrence dangereuse pour les « vrais » ordinateurs qui constituaient leur cheval de bataille. De surcroît, CP/M pouvait « tourner » sur plusieurs types de processeurs, ce qui donnait à ses utilisateurs la liberté du choix de leur fournisseur de matériel, une caractéristique non souhaitée par le département marketing d'IBM. C'est pourquoi le PC n'a pas été lancé avec CP/M, qui lui aurait parfaitement convenu, mais avec un système bien plus rudimentaire et dépourvu de langage de développement décent, MS/DOS produit par Microsoft... La légende racontera une histoire de rendez-vous manqué.

11.2.2 De MS-DOS à Windows

Les années IBM et MS-DOS

Le PC IBM naît en 1981 avec un processeur Intel 8088 et le système PC-DOS de Microsoft, qui est en fait une copie plus ou moins autorisée et incomplète de CP/M qui aurait été dépouillé de plusieurs fonctions importantes, comme l'allocation dynamique de mémoire, le partage de fichiers, la commutation de processus et la gestion de périphériques (voir à ce sujet la rubrique nécrologique de Gary Kildall par John Wharton *Gary Kildall, Industry Pioneer, Dead at 52*, [137]). La mémoire adressable ne peut pas dépasser 640 Ko. La notoriété d'IBM apporte au micro-ordinateur une respectabilité qui va permettre son entrée dans les entreprises, et va aussi attirer les éditeurs de logiciels.

Le PC a une caractéristique surprenante de la part d'IBM : toutes ses caractéristiques techniques internes et externes sont publiées dans une documentation librement accessible à qui veut bien l'acheter. Ceci permet très rapidement la réalisation de copies du PC. Si PC-DOS est le système d'exploitation réservé au PC d'IBM, les producteurs de clones peuvent les équiper de son sosie MS-DOS, parfaitement semblable.

Le BIOS du PC ne réalise pas, contrairement à celui de son modèle CP/M, une véritable abstraction du matériel, il dépend étroitement des caractéristiques du processeur, mais il constitue néanmoins une interface qui va faciliter le travail des cloneurs, en leur permettant par exemple de contourner les brevets d'IBM sur le matériel. Le BIOS permet aussi d'accueillir toutes sortes de cartes d'extension, celles-ci peuvent comporter de la mémoire ROM (*Read Only Memory*, au contenu

non-volatil et figé une fois pour toutes) qui contient des extensions au BIOS, chargées dynamiquement au démarrage ; ainsi de nouvelles fonctions peuvent être prises en compte, ce qui sera particulièrement mis à profit par les créateurs de cartes graphiques.

Durant l'année 1982 Intel lance le processeur 80286, à mots de 16 bits, bus de données de 16 bits et adresses de 24 bits, soit une capacité d'adressage de 16 777 216 octets. Ce processeur introduit des caractéristiques nouvelles dans la famille Intel : à côté d'un mode de fonctionnement compatible avec le 8086, dit *mode réel*, il est doté d'un mode inspiré de *Multics*, le *mode protégé*. Le mode protégé comporte la gestion de registres de segments pour une mémoire virtuelle segmentée à la Multics et une protection par anneaux, mais il lui manque l'accès à la mémoire auxiliaire sur disque par adressage de segment. De toutes les façons les systèmes MS-DOS et Windows n'ont pas utilisé et n'utilisent toujours pas ces possibilités du matériel. Pire, ils resteront longtemps (en fait jusqu'au lancement commercial de Windows NT en 1993) fidèles au mode réel afin de conserver la compatibilité avec les vieux logiciels MS-DOS. Le 80286 et ses successeurs possèdent un dispositif qui permet le lancement de plusieurs machines virtuelles 8086 en mode réel, et c'est cette caractéristique qui fut la plus utilisée par les systèmes Windows jusqu'à la fin du vingtième siècle, sinon après.

En 1984 IBM lance le PC/AT sur la base d'un 80286 exploité en mode réel et avec MS-DOS qui ne sait toujours utiliser que le premier mébioctet de mémoire. Cette machine connut néanmoins un grand succès, mais dont IBM ne fut pas le seul bénéficiaire parce que l'activité des fabricants de clones connut alors un grand essor.

Le schisme entre OS/2 et Windows

En 1987 IBM, désireux de reprendre le contrôle du marché du PC, lança le PS/2, doté d'une architecture fermée protégée par des brevets, d'un nouveau bus baptisé MCA incompatible avec celui de l'AT, et d'un nouveau système d'exploitation, OS/2 conçu en collaboration avec Microsoft. Le processeur était toujours le 80286 vieillissant, voire le 8086 pour les modèles d'entrée de gamme. Malgré d'indéniables qualités techniques qui corrigeaient beaucoup des défauts de l'architecture des PC sous MS-DOS, cette tentative de reprise en main fut un échec total qui aboutit à reléguer IBM dans une position subalterne et marginale sur le marché qu'il avait créé. Plus que le choix de processeurs dépassés face au nouveau modèle d'Intel, le 386 qui sortait au même moment, l'échec de la firme d'Armonk résulta d'une révolte des industriels du PC bien décidés à conserver leur indépendance, révolte soutenue par la clientèle et surtout par Microsoft et Intel, qui voyaient là une occasion de sortir de l'ombre d'IBM pour jouer les premiers rôles. Microsoft avait discrètement préparé une alternative à OS/2,

Windows2 qui sortit opportunément à ce moment. C'est le début de l'ascension de la firme de Redmond.

Windows 2 et 3 n'était qu'une surcouche cosmétique au-dessus de MS-DOS. Ce système à fenêtres, icônes et menus déroulants reposait sur l'utilisation des machines virtuelles de l'Intel 386 en mode réel, qui pouvaient être multiples, ce qui permettait de franchir la barrière de mémoire qui limitait MS-DOS. L'interface graphique faisait pâle figure à côté de celle du Macintosh, mais elle allait enclencher la conquête du grand public par les logiciels de Microsoft.

Les systèmes d'interfaces à fenêtres, que ce soit Windows, MacOS ou le système de fenêtrage X qui fonctionne comme une sur-couche au-dessus d'Unix, introduisent un nouveau style de programmation, la programmation par événements. Le programme principal exécute une boucle infinie qui attend un événement en provenance d'une fenêtre (clic de souris, déplacement de la souris, frappe d'une touche au clavier). Chaque événement est analysé et provoque le déclenchement du sous-programme chargé d'effectuer l'action appropriée.

Windows NT et 95

Windows NT (lancé donc en 1993) et Windows 95 (daté comme son nom l'indique de 1995) allaient enfin utiliser le mode protégé du processeur avec un adressage sur 32 bits et l'usage de bibliothèques de fonctions chargées dynamiquement (DLL, *Dynamic Link Libraries*) analogues aux bibliothèques partagées d'Unix. Mais Windows 95, destiné au grand public, se devait de conserver la compatibilité avec les anciens logiciels, surtout ceux destinés aux jeux, et de ce fait conservait un soubassement MS-DOS.

Faiblesses de Windows 95

Windows 95 disposait de la mémoire virtuelle, d'une gestion de processus et de possibilités de multiprogrammation, mais les bénéfices de cette modernisation étaient obérés par la présence d'un important héritage de code 16 bits dans le noyau et par le partage sans protection d'une partie de l'espace mémoire entre le noyau et tous les processus.

En outre, si Windows 95 était préemptif, son noyau n'était pas réentrant : être réentrant est une qualité statique, lexicale d'un programme, cela signifie que si un processus ou une activité *(thread)* qui l'exécute est interrompu(e) au milieu de cette exécution, il ne reste dans le texte du programme aucune trace de cette exécution inachevée et un autre processus (ou un autre *thread*) peut exécuter la même copie en mémoire de ce même programme à partir d'une autre instruction sans qu'il soit pollué par les exécutions précédentes ; en d'autres termes le texte du programme lui-même ne contient aucune information qui dépende de l'exécution d'un processus particulier ou d'un *thread* particulière,

comme par exemple une variable définie statiquement dans le code (cf. p. 306). Si Windows 95 est préemptif, cela signifie qu'un processus (ou un *thread*) peut être interrompu(e) à n'importe quel instant au profit d'un(e) autre. Si le noyau n'est pas réentrant, cela signifie que le processus interrompu peut avoir laissé une structure de données du noyau dans un état incohérent qui provoquera un comportement imprévisible du processus nouvellement ordonnancé. Bref, un système préemptif à noyau non réentrant est sujet aux pannes inexplicables de type blocage ou arrêt brutal.

Cette absence de sûreté réentrante du noyau Windows 95 présenterait de tels risques si le fonctionnement préemptif du système était effectivement utilisé que la plupart des logiciels raisonnables recouraient à des artifices pour acquérir et conserver le contrôle exclusif du processeur pendant leurs phases d'activité. Le procédé le plus populaire consistait à obtenir, en entrant dans le mode noyau, un verrou de contrôle exclusif qui contrôlait l'accès à pratiquement tout le système. La méthode était radicale, mais elle faisait perdre l'essentiel de l'intérêt de la multiprogrammation.

Atouts de Windows NT

Microsoft destinait Windows NT au marché professionnel, et afin de satisfaire les exigences supposées de cette clientèle avait recruté chez Digital Equipment David Cutler et Mark Lucovsky. Cutler avait été l'auteur du système RSX-11M des PDP-11, puis avec Lucovsky un des architectes du système VMS du VAX. La conception de Windows NT [131] commença en octobre 1988.

Windows NT visait la portabilité ; Cutler et Lucovsky pensaient que le seul moyen d'atteindre cet objectif était de développer le système simultanément pour deux architectures matérielles différentes. Ils commencèrent par la version destinée aux processeurs RISC de MIPS, dont personne ne se souciait chez Microsoft, afin d'acquérir la certitude que la version pour Intel x86 serait vraiment portable.

Toujours afin d'assurer la portabilité et l'évolutivité de leur système, Cutler et Lucovsky s'interdirent d'incorporer au noyau la sémantique de tel ou tel système ; ils décidèrent de développer plutôt un noyau réduit aux mécanismes de base et doté d'une interface de programmation (API) claire au-dessus de laquelle ils implantèrent trois sous-systèmes qui implémentaient les sémantiques respectivement des systèmes POSIX, OS/2 et Windows. Cet discipline se révéla judicieuse lorsque Microsoft décida d'abandonner définitivement OS/2 et de développer Windows de façon totalement indépendante.

Si l'on dresse la liste des dispositifs qui figurent dans Windows NT et sont absents de Windows 95 et même Windows 98, on trouve tout ce qui touche à la sécurité, le support des multi-processeurs, un système de fichiers plus perfectionné.

Le caractère préemptif et la gestion de mémoire virtuelle de NT sont dépourvus des compromis qui en font perdre une partie du bénéfice à 95 et 98.

Insuccès de Windows NT

Malgré ces qualités indéniables, le lancement de Windows NT fut un échec relatif. D'abord, ce nouveau système ne garantissait pas la compatibilité avec les vieux programmes MS-DOS. Ensuite il était cher.

Et puis Windows NT était entaché de compromis douteux. Par exemple, parmi ses dispositifs de sécurité figure un vrai système d'enregistrement des utilisateurs autorisés et de contrôle des mots de passe. Mais comme il ne faut pas perturber les habitudes culturelles de la clientèle, l'usage de ce système d'identification et d'authentification est facultatif, ce qui lui retire toute efficacité réelle. De même, NT comportait un système de protection des fichiers et d'autres objets par listes de contrôle d'accès (ACL) ; ce système, directement hérité de VMS, est robuste, mais pour qu'il ne bloque pas irrémédiablement un trop grand nombre de vieux programmes hérités du passé, il est implémenté en mode utilisateur, ce qui le rend très vulnérable aux attaques et lui retire donc tout intérêt. Et pour couronner le tout, les premières versions de NT, jusqu'au *Service Pack 3* de NT 4 en mai 1997 (en fait une nouvelle version), étaient gourmandes en mémoire, lentes et instables (les fameux écrans bleus qui s'affichaient lors des pannes système causaient le découragement des utilisateurs).

La désaffection qui accueillit Windows NT et la persévérance des clients à utiliser les vieux systèmes 16 bits basés sur MS-DOS conduisit Microsoft à prolonger leur vie sous les noms de Windows 98 et Windows Me. Ces anciens systèmes, notoirement faibles du point de vue de la sécurité, ont vécu encore des années. En principe Windows XP aurait dû sonner l'heure de la réunification entre les systèmes grand public et les systèmes d'entreprise.

Windows 2000, ses successeurs

On trouvera une description détaillée de Windows 2000 dans le livre de Solomon et Russinovich *Inside Windows 2000* [125], mais l'exposé de Tanenbaum dans son livre *Modern Operating Systems* [129] constitue déjà une introduction substantielle.

Windows 2000, en fait la version 5 de Windows NT, était un système plein de bonnes résolutions. Le modèle de sécurité et son implémentation ont été revus, conformes aux standards (IPSec, Kerberos, LDAP, X509), le service de noms était fondé sur le DNS à la mode Internet, le système de fichiers NTFS était doté de fonctions de chiffrement applicables aux fichiers individuels, aux répertoires ou à toute une partition. Chaque processus peut contrôler un ou plusieurs *threads* , qui sont gérées par le noyau. En fait on pourrait plutôt reprocher à Windows

2000 sa profusion : plus de 30 millions de lignes de code, des centaines d'appels système pour le système au sens restreint du terme (gestion des processus, de la mémoire, des I/O, de la synchronisation, des fichiers et de la protection), des milliers d'appels système pour l'interface graphique.

Autant dire que c'est un système peu intelligible, ce qui est parfois aussi gênant qu'un système indigent : ainsi l'implémentation des dispositifs de protection et de sécurité est obscure, ce qui ne donne à l'ingénieur de sécurité d'autre choix que de leur faire une confiance aveugle, avec les inconvénients que nous avons soulignés à la p. 234.

Un des aspects intéressants de Window 2000 est sa couche d'abstraction du matériel (HAL, pour *Hardware Abstraction Layer*) située sous le noyau, et qui regroupe tous les aspects du système trop dépendants d'un matériel particulier, comme les interfaces avec les périphériques, la gestion de l'accès direct à la mémoire pour les opérations d'entrée-sortie, des interruptions, de l'horloge, des verrous de synchronisation des multi-processeurs, l'interface avec le BIOS, etc. Bref, Windows rejoint enfin le niveau d'abstraction de CP/M.

Symétriquement, l'accès des programmes en mode utilisateur aux services système se fait à travers une couche d'interface constituée de la bibliothèque dynamique `NTDLL.DLL`.

Windows 2000 dispose d'un système perfectionné de mémoire virtuelle, assez proche de celui de VMS. Les structures internes des versions ultérieures de Windows, jusqu'à Windows 10, sont directement dérivées de Windows 2000, même si les interfaces sont assez différentes.

François Anceau a publié une excellente synthèse de l'évolution du PC dans son article *La saga des PC Wintel* [6], où le lecteur pourra trouver de nombreuses données supplémentaires.

À la fin de l'année 2002 une proportion écrasante des ordinateurs en service dans le monde (il y en avait un milliard) fonctionnait avec une version ou une autre de Microsoft Windows. En 2013, Cisco (qui est bien placé pour le savoir) estimait à 8,7 milliards le nombre d' « objets » connectés à l'Internet[3], et si l'on observe la répartition des accès au Web selon le système d'exploitation émetteur de la requête, Windows ne représente plus en 2018 que quelques 35 % des accès, sévèrement concurrencé par MacOS et surtout Android, un système d'exploitation créé par Google, dérivé de Linux et équipé d'un système Java spécifique (machine virtuelle Dalvik initialement, ultérieurement remplacée par un environnement d'exécution en code natif ART, pour *Android Runtime*) pour le développement d'applications. Android est utilisé pour les smartphones et les tablettes.

3. `https://blogs.cisco.com/news/cisco-connections-counter/`

11.3 La saga des processeurs Intel

La naissance et l'essor des micro-ordinateurs, qui sont aujourd'hui les ordinateurs tout court, est inséparable de celle des processeurs Intel, qui méritent ici une section particulière.

Pour compléter les informations de cette section on pourra consulter l'excellent article de Samuel « Doc TB » Demeulemeester « L'épopée des microprocesseurs - Un demi-siecle d'évolution » [44] dans « Canard PC Hardware ».

Cette section emprunte certaines informations à l'article de Tom R. Halfhill dans le numéro de février 2009 de « Microprocessor Report » et à une conférence de Richard S. Tedlow, professeur à Harvard Business School.

11.3.1 Quand les microprocesseurs étaient du bricolage

En 1968 Robert Noyce et Gordon Moore quittent Fairchild, l'entreprise qu'ils avaient créée et où Noyce avait inventé en 1958 le circuit intégré [4], pour créer une nouvelle entreprise, Intel, et y inventer le microprocesseur. Cette date est historique parce qu'elle inaugure la troisième révolution industrielle [5], dont Michel Volle [6] ne manque pas de nous rappeler qu'elle est gouvernée par des lois économiques différentes et qu'il lui faudra un système éducatif, une organisation du travail et un encadrement législatif différents, qui lui manquent aujourd'hui, avec les troubles qui en résultent pour l'éducation, pour l'emploi et pour le droit des affaires.

Sur le moment personne ne perçoit l'importance de l'événement. Le premier microprocesseur, le 4004, sorti d'usine en novembre 1971, est certes un ordinateur complet dans un seul circuit intégré, mais un ordinateur très rudimentaire et très lent. Les industriels de la « grande informatique » (ou de la moyenne...) ne disposaient pas d'instruments d'optique suffisamment puissants pour simplement voir cet objet, alors de là à comprendre qu'il allait révolutionner leur activité et conduire la plupart d'entre eux à disparaître, le pas était grand.

Les gens qui fabriquent des microprocesseurs, et ceux qui les utilisent en les incorporant, au prix de mille astuces, dans des appareils de plus en plus ingénieux, passent pour d'aimables bricoleurs, ainsi dès 1972 les Français André Truong Trong Thi et François Gernelle qui conçoivent le micro-ordinateur Micral autour du microprocesseur Intel 8008. En janvier 1975, MITS (Model Instrumentation Telemetry Systems, fondée à Albuquerque, au

4. Jack Kilby, chez Texas Instruments, avait fait la même invention indépendamment la même année.

5. La première, à la fin du XVIII^{e} siècle, reposait sur la chimie, la métallurgie et la machine à vapeur ; la seconde, à la fin du XIX^{e} siècle, sur l'électricité industrielle et le moteur à combustion interne.

6. Cf. `https://michelvolle.blogspot.com/2015/03/le-secret-de-liconomie.html`

Nouveau-Mexique, par Ed Roberts) lance l'Altair 8800, un micro-ordinateur basé sur l'Intel 8080 et vendu en kit par correspondance pour 400 dollars : cela ne peut pas être sérieux...

Si les industriels de l'informatique ne font pas grand cas du microprocesseur, ceux de l'électronique et leurs clients n'ont pas mis longtemps à comprendre le parti à en tirer. Remplacer une douzaine de circuits intégrés connectés les uns aux autres par une seule puce programmée et reprogrammable en cas de besoin divise les coûts par dix et simplifie la maintenance.

La conception d'un nouveau processeur ne demande alors que quelques semaines de travail à une dizaine d'ingénieurs, parce que l'architecture n'en est pas très complexe : le 4004 ne comporte que 2 300 transistors, son successeur le 8008 en aura 3 300, le 8080, lancé en 1974, à peu près 6 000. Ces chiffres sont à comparer aux 731 millions de transistors sur une puce de 263 mm^2 de l'Intel Core i7 de 2008, dont la conception a représenté une charge de l'ordre de 1 000 années-hommes (et en 2015 l'Intel Core i3/i5/i7, architecture Skylake, a 1 750 000 000 transistors en 14nm).

En bref, les premiers microprocesseurs sont perçus comme des objets simples, bon marché, faciles à concevoir, pratiques, mais bassement utilitaires. Jusqu'au début des années 1980 d'ailleurs l'essentiel du chiffre d'affaires d'Intel provient des mémoires et d'autres types de composants.

11.3.2 Accords de seconde source et offensive japonaise

Une caractéristique intéressante du marché des microprocesseurs dans sa première décennie était la présence systématique d'accords de seconde source. Les donneurs d'ordres, qui étaient en général des entreprises de beaucoup plus grande taille que les fournisseurs, voulaient des garanties de régularité des approvisionnements, et aussi faire pression sur les prix, et pour ce faire ils imposaient à Intel et aux autres fabricants de microprocesseurs, tels Zilog ou Motorola, de céder à d'autres sociétés la licence qui leur permettrait de fabriquer leurs produits. Ainsi le 8086 d'Intel, lancé en 1978 (29 000 transistors, dessiné en 3µm) était fabriqué en seconde source par AMD, Fujitsu, Harris Semiconductor et quelques autres. Le dessin du microprocesseur n'était pas considéré comme un actif de grande valeur, et le céder à un concurrent ne soulevait pas d'objection.

Le lancement en 1981 du micro-ordinateur IBM PC, basé sur un processeur Intel 8088, version dégradée du 8086, allait bouleverser les conditions économiques de ce marché, en transformant la micro-informatique, jusque là destinée à un public de hobbyistes, en industrie d'importance mondiale.

On aurait pu penser que cette évolution allait faire la fortune d'Intel : elle a failli causer sa ruine et sa disparition. En effet la croissance rapide du marché des microprocesseurs avait attiré de grandes entreprises japonaises telles que NEC, Fujitsu et Hitachi, ou américaines comme Texas Instruments ou National

Semiconductors, lesquelles disposaient de capacités d'investissement sans commune mesure avec celles d'Intel, ce qui provoqua une baisse des prix et une diminution importante de la profitabilité de cette activité. En 1983 Intel n'était plus que le dixième producteur mondial de circuits intégrés et son déclin semblait inéluctable.

Un des facteurs de la supériorité des Japonais, outre leur capacité supérieure d'investissement, résidait dans leurs meilleures performances en production : le taux de microprocesseurs défectueux en sortie de leurs usines était de deux ordres de grandeur inférieur à celui d'Intel. Pour dessiner des processeurs sur une galette de silicium (le *wafer*), on utilise une machine très complexe et onéreuse, le stepper[7], dont à l'époque le premier fabricant mondial était Nikon au Japon, et Nikon collaborait directement avec les fabricants de semi-conducteurs japonais, qui avaient ainsi accès en priorité aux innovations. Aujourd'hui le marché mondial du stepper se partage entre Nikon, Canon, l'américain Ultratech, le néerlandais ASML[8] et quelques autres.

11.3.3 Comment l'industrie américaine fit face au Japon

Il faut rappeler qu'à cette époque le marché des grands systèmes IBM représentait les trois quarts du marché informatique mondial, et que les industriels japonais s'en étaient approprié une part très significative. Encore en 1989, Fujitsu était le numéro 2 du marché derrière IBM, NEC le numéro 4 derrière Digital Equipment alors à son apogée, Hitachi le numéro 6. L'emprise japonaise se renforçait aux deux extrémités du marché, semi-conducteurs et gros ordinateurs, et elle semblait invincible. On trouvera des chiffres et quelques hypothèses sur les facteurs qui ont permis que l'industrie américaine redresse sa position sur le site de l'auteur[9]. Ici nous allons plus précisément examiner comment Intel est redevenu le leader mondial d'une industrie qu'il avait créée.

Parmi les facteurs du redressement, il faut noter que les industriels américains des semi-conducteurs ont évité une attitude qui fut fatale à de nombreuses entreprises informatiques : la suffisance. Très tôt ils ont reconnu la force de la menace japonaise et ils ont su surmonter leurs rivalités et, jusqu'à un certain point, unir leurs forces pour la juguler. Dès 1977 avait été créée la Semiconductor Industry Association[10], qui elle-même créa en 1982, à l'initiative de Bob Noyce, une filiale destinée à organiser et à financer la recherche

7. Cf. `http://en.wikipedia.org/wiki/Stepper`
8. Cf. `http://en.wikipedia.org/wiki/ASML_Holding`
9. Cf. `https://laurentbloch.net/MySpip3/Industrie-electronique-et-informatique-americaine`
10. Cf. `http://www.sia-online.org/abt_history.cfm`

pré-compétitive, Semiconductor Research Corp.[11] (SRC). Les statuts de SRC prévoyaient explicitement le refus d'accepter des membres non-américains.

La SIA et SRC ne furent en aucun cas des organisations potiches. Elles reçurent des financements importants en provenance des industriels et des pouvoirs publics, notamment du *Department of Defense* (DoD), et s'engagèrent dans une politique de collaboration active avec les Universités ; ces actions eurent pour fruits de nombreuses innovations techniques et le redressement de la courbe de progrès de l'industrie américaine, innovations et progrès dont les bénéfices étaient explicitement réservés aux entreprises américaines.

11.3.4 Le tournant du 386

La création de la SIA et de SRC ont contribué à rétablir la position de l'industrie américaine des semi-conducteurs en général, mais il reste à expliquer le redressement particulier d'Intel, qui en ce début des années 1980 était en grand péril, et c'est là que l'analyse du professeur Tedlow nous éclaire.

En 1982 Intel lança le 286 (134 000 transistors, 1,5µm), sur la base duquel IBM lança le PC AT, dont Tedlow nous fait observer qu'il est le premier signe du transfert du leadership technologique d'IBM à Intel, dans la mesure où les seules innovations apportées par le PC AT proviennent du 286. Le 286 fait l'objet d'accords de seconde source avec IBM, AMD, Harris (Intersil), Siemens et Fujitsu.

Pour succéder au 286, Intel voulait produire un processeur à architecture 32 bits, et non plus 16 bits comme le 8086 et le 286, ou *a fortiori* 8 bits comme le vieux 8080[12]. Une architecture 32 bits serait de nature à abolir les limitations techniques gênantes des systèmes 16 bits, notamment en termes de taille de la mémoire adressable, mais un impératif était donné aux ingénieurs de l'équipe de conception : il fallait que le nouveau processeur, le 386, soit compatible avec ses prédécesseurs, c'est-à-dire que les logiciels qui fonctionnaient avec le 8086 ou avec le 286 soient encore utilisables.

Pour atteindre cet objectif ambitieux, Intel réunit une équipe brillante et dépensa 100 millions de dollars, le double de ce qu'avait coûté le design du 286. La définition de l'architecture prit un an : cet investissement devait s'avérer durable, puisque selon toute vraissemblance le processeur de l'ordinateur avec lequel vous lisez cet article, Mac ou PC, est un descendant direct du 386, c'est-à-dire qu'il exécute les mêmes instructions élémentaires.

11. Cf. `http://www.src.org/member/about/history.asp`

12. Quand on dit que l'architecture d'un processeur est à 32 bits, cela signifie, sans trop entrer dans les détails, que les nombres entiers relatifs qu'il sait traiter ont 31 chiffres binaires, soit une valeur absolue de l'ordre de 2 milliards, que la taille de sa mémoire peut théoriquement atteindre 4 milliards de caractères, et que les échanges de données entre le processeur, la mémoire et les organes prériphériques se font par paquets de 32 bits, soit quatre octets. Sous réserve bien sûr de dispositifs techniques ingénieux pour dépasser ces limites.

Le 80386[13] (275 000 transistors, 1,5µm) fut lancé en 1985 et fut un immense succès technique, mais pas seulement. Comme IBM restait fixé au 286, pour lequel il avait développé un système d'exploitation spécialement adapté, OS/2, d'autres industriels furent les premiers à produire des ordinateurs à base de 386, au premier rang desquels Compaq. Et Microsoft lança la première version de Windows pour ces ordinateurs, qui se révélèrent rapidement plus puissants et plus faciles à utiliser que ceux d'IBM : le couple Windows-386 sonnait le glas de l'hégémonie d'IBM sur le marché du PC.

11.3.5 Fin des accords de seconde source

Microsoft, IBM et Compaq ne furent pas les seuls à être affectés par les innovations du 386 : Intel aussi, bien sûr.

Pour protéger ses investissements et permettre aux prix de remonter, Intel (en la personne de son *Chief executive officer* Andy Grove) décida de ne pas accorder de licence de seconde source pour le 386. Cette décision révolutionnaire, mal comprise à l'époque, était accompagnée de mesures de réorganisation interne, notamment pour améliorer la qualité de la production. Une autre décision fut prise, qui prit à rebrousse-poil beaucoup d'ingénieurs d'Intel : abandonner la production des mémoires, qui représentait alors la principale source de profit de l'entreprise.

Andy Grove avait compris que le centre de gravité de l'industrie micro-électronique s'était déplacé des États-Unis au Japon, et il suscita une initiative destinée à gagner pour Intel des clients japonais, qui représentaient le critérium de l'exigence technique. Pour gagner le marché international des microprocesseurs, il fallait battre les concurrents japonais sur leur propre terrain, afin de convaincre la clientèle internationale de la supériorité des produits Intel.

La décision de garder le monopole du 386 fut prise à un moment où Intel était en train de regagner le leadership qu'il avait perdu tant dans la technologie que pour la qualité et la capacité de production. Cette décision fut à l'origine de la remontée d'Intel vers la première place dans l'industrie des semi-conducteurs, qu'il occupe aujourd'hui solidement avec un chiffre d'affaires 2008 de 33,767 milliards de dollars, double de celui du second, Samsung avec 16,902 milliards de dollars.

11.3.6 Fin de l'intégration verticale

Un effet collatéral de la nouvelle orientation d'Intel, de son renouveau et du succès du 386 fut la fin du modèle alors en vigueur dans l'industrie

13. Cf. `https://en.wikipedia.org/wiki/Intel_80386`

informatique : l'intégration verticale. Jusqu'alors, tant IBM que Control Data ou Digital Equipment concevaient et fabriquaient les éléments électroniques de l'unité arithmétique et logique de leurs ordinateurs, de sa mémoire et de ses périphériques tels que disques, dérouleurs de bande et autres imprimantes, et ils en produisaient le logiciel, depuis le système d'exploitation jusqu'au traitement de texte. À partir de 1986, l'industrie informatique devint essentiellement une industrie d'assemblage, et même IBM achète la plus grande partie de ses processeurs, de ses mémoires et de ses disques, sans parler du logiciel. Intel et Microsoft tiennent en main les cartes maîtresses de ce jeu, ils en tirent les ficelles, parce que leur technologie incorpore bien plus de valeur ajoutée que les usines d'assemblage, au demeurant fort bien conçues, de Dell[14].

11.3.7 Conversion silencieuse à l'architecture RISC

Comme nous l'avons signalé ci-dessus dans la section consacrée à l'architecture RISC (*Reduced Instruction Set Computer*, cf. p. 276), celle-ci a révolutionné la conception des microprocesseurs en en améliorant considérablement les performances ainsi que les délais de conception et de mise au point, grâce à sa simplicité. À partir de 1985 la plupart des constructeurs d'ordinateurs lancèrent leur propre architecture RISC : MIPS (créé pour la circonstance), Hewlett-Packard, Sun, IBM et Digital Equipment Corporation (DEC). Au tournant de la décennie 1990, il était communément admis que le temps de l'architecture CISC (*Complex Instruction Set Computer*) était révolu.

C'était sans compter avec le poids de l'existant : pendant le même laps de temps, le marché de l'ordinateur personnel devenait décisif pour l'industrie, le parc de logiciels grand public pesait lourdement sur ce marché, et ces logiciels étaient écrits pour des processeurs Intel x86 d'architecture CISC ; cette architecture allait donc continuer à dominer le marché, et le dominerait encore aujourd'hui sans l'essor des téléphones portables, en quasi-totalité équipés de processeurs ARM d'architecture RISC.

Mais les ingénieurs d'Intel avaient compris que s'ils ne voulaient pas être irrémédiablement distancés en termes de performances, ils devaient tirer partie des avantages du RISC. La solution retenue sous le nom d'architecture P6, sous la conduite de Robert Colwell, fut de construire un processeur doté d'unités d'exécution RISC entourées d'un micro-code de traduction qui présenterait l'interface habituelle de l'architecture x86 CISC, au prix d'une certaine dégradation des performances et d'une surconsommation électrique importante. Le premier processeur P6 fut le Pentium Pro, présenté en novembre 1995, puissant mais cher, suivi de versions plus accessibles. Aujourd'hui (2018) tous les processeurs Intel reposent sur ce principe architectural (cf. l'article de Samuel « Doc TB »

14. Cf. `https://laurentbloch.net/MySpip3/Crise-du-modele-Dell-Computer`

Demeulemeester *L'épopée des microprocesseurs - Un demi-siecle d'évolution* [44] dans *Canard PC Hardware* pour plus de détails).

11.4 Une alternative : MacOS

MacOS est apparu avec le Macintosh en 1984. En 2002, Apple s'est engagé dans un processus destiné à lui substituer MacOS X, qui est en fait un système totalement nouveau bâti sur un micro-noyau Mach 3 surmonté d'un serveur Unix BSD [15] (cf. p. 311). Nul doute que ce changement soit bénéfique : un système Unix sur un micro-noyau représente une base architecturale solide, mais un changement de système est une opération lourde qui implique des modifications dans toute la gamme de logiciels disponibles. Apple a fait montre dans cette affaire de sa grande maîtrise des interfaces homme-machine, parce que la plupart des utilisateurs ne se sont pas aperçus de ce changement de système, pourtant radical !

MacOS (de version 9 ou antérieure, dans les lignes qui suivent c'est de ces versions que nous parlons, à l'exclusion de MacOS X et des versions ultérieures macOS) souffrait des mêmes défauts architecturaux que Windows 95 ou 98, mais à un degré moindre de gravité. Comme ceux-ci il s'agissait d'un système au caractère préemptif incertain et au code le plus souvent non réentrant, de ce fait sujet aux blocages et aux arrêts brutaux causés par des conflits ou des étreintes fatales entre processus, si ce n'est tout simplement par le déclenchement d'une interruption asynchrone à un moment où le système n'est pas dans un état convenable pour la recevoir. De fait, la « multiprogrammation coopérative » entre programmes pseudo-simultanés n'est possible de façon sûre que si la commutation entre processus a lieu à des emplacements bien déterminés du code, lors de l'appel au sous-programme de bibliothèque `WaitNextEvent`, et en effectuant à cette occasion des manipulations de données enfouies profondément dans le système. Bref, MacOS n'était pas un vrai système multi-tâches.

Le « vieux » MacOS disposait d'un espace mémoire unique où cohabitaient sans protection le système et les programmes lancés par l'utilisateur. Le système était accompagné d'une vaste bibliothèque de fonctions généralement connue sous le nom de *Toolbox*, et fonctionnait en étroite symbiose avec des éléments codés en mémoire ROM (*Read-Only Memory*, une mémoire incorporée au matériel de telle sorte que le contenu en soit inaltérable) et protégés par des brevets, ce qui a empêché la production d'ordinateurs compatibles avec le Macintosh par

15. Ce changement d'orientation technique décisif fut la conséquence d'un changement politique tout aussi décisif à la tête d'Apple : Steve Jobs avait été évincé en 1983 de l'entreprise qu'il avait créée au profit d'un « vrai manager » venu de Pepsi Cola, John Sculley, qui en restera le dirigeant jusqu'en 1993. Jobs a créé an 1985 une autre entreprise, NeXT, qui n'a pas eu de grand succès commercial, mais au sein de laquelle fut élaboré l'OS qui allait devenir MacOS X après l'échec pitoyable du « vrai manager ».

d'autres industriels, sauf pendant la courte période où Apple a vendu des licences. La *Toolbox* n'était pas réentrante et faisait un usage systématique de variables d'état globales, ce qui rendait très problématique par exemple le développement d'applications en Java qui auraient utilisé les possibilités de *multithreading* de ce langage. D'ailleurs l'implémentation de Java sous MacOS a toujours été réputée problématique. Comme sous Windows 95 et 98, les développeurs ont tant bien que mal résolu ces problèmes en ayant recours à de longues sections critiques protégées par des verrous de contrôle exclusif.

Si la situation engendrée par ces lacunes des anciens MacOS a été moins calamiteuse que dans le cas de Windows 95 et 98, c'est pour une série de raisons contingentes. D'abord, le système MacOS et tous les Macintosh qu'il devait faire fonctionner étaient conçus par une seule entreprise en un seul lieu, Apple à Cupertino. Ceci permettait une grande cohérence dans les développements et évitait de livrer au client des systèmes trop incertains. Ensuite, Apple a su coordonner le travail de tous les développeurs de logiciels et de matériels extérieurs à la société de telle sorte qu'ils respectent tous les mêmes règles d'interface. C'est ce qui fait l'agrément d'usage du Macintosh : quel que soit le logiciel que l'on utilise, les mêmes fonctions sont toujours réalisées de la même façon, en cliquant au même endroit sur un article de menu qui porte le même nom. Microsoft est venu un peu tard à cette discipline. De ce fait le Macintosh doté de MacOS, même une version antique, est beaucoup plus agréable à utiliser et déclenche beaucoup moins d'appels au secours en direction du service d'assistance qu'un système sous Windows.

MacOS X est un système entièrement nouveau qui repose sur d'excellentes fondations techniques : c'est un Unix BSD assis sur un micro-noyau Mach et surmonté d'une interface homme-machine aussi réussie que les précédentes versions de MacOS. Il a permis à Apple de reconquérir sur les machines sous Windows un terrain alors réduit à 2 ou 3% du marché. Il a surmonté la concurrence des solutions à base d'Unix libres, moins onéreuses à l'achat. La facilité d'usage par le naïf est un critère vital, et la réponse à ces incertitudes a donc été oui. Parce que durant quelques années, grâce à l'architecture du PC à processeur Intel et à Windows qui échouait à en dissimuler les détails intimes à l'utilisateur, le monde a été plein de comptables qui potassaient les niveaux d'interruption associés à leur cartes graphiques et de présidents d'universités qui expérimentaient les combinaisons possibles de configurations de leurs disques IDE pendant les heures de travail, ce qui avait indubitablement un coût très supérieur à la valeur ajoutée résultante.

11.5 Autre alternative : Unix

En fait pour être complet il faudrait dire « Unix libre sur PC de super-marché ». Microsoft et Intel, rendons leur cette justice, ont rendu possible le PC à 300 Euros. Rappelons-nous également qu'en 1980 Bill Gates pensait qu'Unix était le système d'avenir pour les micro-ordinateurs, et que pour cette raison il avait acquis une licence Unix pour lancer sa version de ce système : Xenix.

Maintenant sur une telle machine il est possible d'installer un autre système que Windows, Linux le plus souvent, en tout cas pour l'utilisateur final. Le principal avantage de Windows, c'est que lorsque vous achetez le PC il est déjà installé, avec en général quelques logiciels en plus, dont le traitement de texte habituel dont le lecteur n'imagine peut-être pas qu'il puisse être remplacé par autre chose. Il faut un cœur bien accroché pour formater son disque dur et entreprendre d'y installer un système téléchargé sur une clé USB, d'autant plus que le traitement de texte en question n'y est pas. Le particulier isolé hésitera sans doute, mais si son voisin d'amphithéâtre ou son collègue de laboratoire lui promettent aide ou assistance, il franchira peut-être le pas. À la fin de l'année 2002 on estimait à vingt millions les ordinateurs qui fonctionnaient sous Linux, devenus 70 millions en 2013 : ce n'est pas si peu, mais si Linux équipe la moitié des serveurs en entreprise et 100 % des super-calculateurs [16], il a du mal à conquérir le grand public, sauf sous la forme Android, qui équipait selon Gartner 79% des smartphones et des tablettes vendus en 2013. Des interfaces homme-machines qui rappellent celle de Windows sont apparues (Gnome, KDE, Xfce...) et ne fonctionnent pas plus mal que l'original.

Plus fondamentalement, la question est de savoir si le système d'exploitation payant a un avenir. Microsoft répond oui, bien sûr, et pour XP a introduit de façon systématique des redevances périodiques pour qui veut disposer des nouvelles versions du système. Cette politique me rappelle l'IBM des années 1970, qui détenait plus de 90% du marché et ne connaissait pas de limite à la domination sur le client. On a vu la suite. Il est sûr en tout cas que le logiciel libre occupe aujourd'hui une place telle qu'il ne s'évaporera pas en une nuit, et que dans le domaine plus particulier du système d'exploitation il fait peser une hypothèque assez lourde sur l'avenir du logiciel privé.

La domination absolue et éternelle du marché par une seule firme est un fantasme propre au monde de l'informatique : IBM hier, Microsoft aujourd'hui, ou sur des secteurs plus spécialisés Oracle et Cisco. Même dans ses rêves les plus euphoriques le président de General Motors a toujours su qu'il y aurait Ford ou Toyota, celui de Boeing qu'il y aurait EADS.

16. `http://en.wikipedia.org/wiki/Usage_share_of_operating_systems`

Michel Volle [136] nous a bien expliqué le mécanisme de formation des monopoles dans l'industrie informatique : elle fonctionne sous le régime de la *concurrence monopolistique* évoquée p. 254 :

« Lorsque le rendement d'échelle est croissant le coût unitaire le plus bas sera celui de l'entreprise qui produit la plus grande quantité. Elle sera donc en mesure d'évincer ses concurrents en proposant un prix inférieur au leur : ce marché obéit au régime du "monopole naturel".

« On pourrait donc s'attendre à ce que tous les marchés soient dans l'iconomie dominés par un monopole. Il n'en est cependant rien car les entreprises disposent d'une arme qui leur permet de résister : la diversification du produit en variétés. »

C'est dans l'illusion de la pérennité de ces monopoles que réside le caractère fantasmatique de la croyance. Parce que dans la réalité diachronique la vitesse de l'innovation technologique fait et défait les positions les plus solides, les plus monopolistes. Tôt ou tard Microsoft suivra la voie d'IBM, voire celle de Digital Equipment.

Risquons l'hypothèse que ce fantasme (heureusement régulièrement démenti par les faits) soit engendré par une proximité inquiétante (et d'ailleurs surestimée) entre l'esprit que nous prêtons à l'ordinateur et le nôtre. Toute pluralité du démiurge de cet esprit introduit une sensation d'insécurité semble-t-il intolérable. Nous aspirons à l'unité des processeurs et des mémoires. Que le voisin soit « sous un autre système » nous perturbe, nous le lui faisons savoir avec véhémence, parfois. Voici donc enfin l'explication des controverses lors des dîners en ville évoquées dans les premières lignes de ce livre : j'espère ainsi ne pas l'avoir écrit en vain.

Chapitre 12 Le micrologiciel *(firmware)*

Sommaire

12.1 Sous le système, le micrologiciel

Tout au long de cet ouvrage nous avons décrit le fonctionnement du système d'exploitation, ainsi que celui du matériel informatique et du réseau dans la mesure où cela était nécessaire pour comprendre le système. Mais il existe un autre élément moins visible mais tout aussi nécessaire au fonctionnement de l'ordinateur, le *micrologiciel* (*firmware* en anglais), que nous avons déjà évoqué au chapitre 2 p. 19. À l'origine des temps (années 1970) il était constitué du BIOS *(Basic Input/Output System)*, qui s'est perfectionné pour devenir UEFI *(Unified Extensible Firmware Interface)*, implanté dans un composant électronique distinct du processeur, branché sur la carte-mère.

À partir de 2008 Intel a introduit *Intel Management Engine*, dont un composant est *Intel Active Management Technology* (AMT, les autres industriels tels AMD et ARM disposent de technologies équivalentes) ; il s'agit en fait d'un ordinateur complet, implanté dans un composant électronique distinct, avec un véritable système d'exploitation, qui supervise tout le fonctionnement de l'ordinateur, notamment du point de vue de la sécurité. Nous allons examiner ces dispositifs.

12.2 Dispositif d'amorçage du système

À la page 33 nous avons donné une première description du dispositif de démarrage d'un ordinateur et d'amorçage *(boot)* de son système d'exploitation. Tous les ordinateurs contemporains utilisent pour réaliser la première phase de cette fonction un logiciel assez particulier généralement intitulé BIOS *(Basic Input/Output System)*. Le BIOS a été inventé en 1975 par Gary Kildall pour son système d'exploitation CP/M (cf. p. 320).

Le BIOS est enregistré dans un élément de mémoire non-volatile physiquement distinct des autres composants de la carte mère de l'ordinateur ; depuis les années 1990 c'est en général de la mémoire Flash, analogue à celle des clés USB et des disques SSD, ce qui permet de le modifier, et ce qui introduit par conséquent un risque de modification malveillante. La carte mère et le processeur sont configurés de sorte qu'à la mise sous tension ce programme soit chargé dans la mémoire vive et commence son exécution. L'action de ce programme consiste essentiellement à aller chercher sur une mémoire externe (disque dur, clé USB…) préparée à cet effet un autre programme un peu moins petit, à le recopier en mémoire centrale et à en déclencher l'exécution. Ce processus est connu sous le nom de *boot-strap* ou simplement *boot*, mais il est permis de dire amorçage. Comme indiqué p. 33, c'est ce programme de *boot* qui va charger en mémoire le noyau du système d'exploitation, éventuellement à partir d'un serveur de *boot* sur le réseau. Afin de simplifier les opérations du BIOS (rappelons-nous que ces mécanismes avaient été conçus en des temps de processeurs lents et de mémoire de faible capacité), le programme de *boot* était le plus souvent enregistré dans le premier secteur du premier disque visible sur le bus [1]. Ce secteur contenait aussi la table des partitions « ancien style » du disque, il est nommé *Master Boot Record* (MBR, cf. p. 114, où nous avons abordé ces questions sous l'angle du système de fichiers). Nous allons voir que les systèmes plus modernes avec des mémoires et

1. Il est possible d'interrompre le déroulement de la phase d'amorçage avant le démarrage du système d'exploitation afin de changer de périphérique de démarrage, par exemple pour installer un nouveau système d'exploitation à partir d'une clé USB. Ceci se fait par pression, pendant le démarrage, sur la touche `F2`, ou `F9`, ou `F10`, ou `Escape`, ou `Delete`, ou `Suppr` (cela dépend du système et du modèle de matériel).

des disques plus spacieux n'utilisent plus vraiment le MBR (sauf s'ils comportent toujours un BIOS pour pouvoir démarrer à partir de supports à l'ancienne mode tels que cédéroms), mais que la table des partitions et le logiciel d'amorçage occupent des espaces plus vastes ailleurs sur le disque.

Nous avons vu p. 322 comment, lors du lancement en 1981 du PC, IBM en a publié toutes les spécifications techniques, et notamment celles de la version simplifiée du BIOS qu'ils avaient adoptée (avec le code source!). Ce BIOS était à l'échelle des capacités des mémoires et des disques de l'époque, avec notamment au plus quatre partitions physiques sur le disque dur, lequel pouvait comporter (jusque dans les années 1990) au maximum 1024 cylindres, 256 têtes, 63 secteurs par piste, 2,2 téraoctets de capacité totale. Il a fallu longtemps jongler avec ces limites, jusqu'à ce que le constructeur Intel et à sa suite AMD, American Megatrends, Apple, Dell, HP, IBM, Insyde, Microsoft, Phoenix Technologies, etc. créent l'*UEFI Forum* pour promouvoir l'*Unified Extensible Firmware Interface*[2] (UEFI, « Interface micrologicielle extensible unifiée », standard publié en 2006, destinée à se substituer au vieux BIOS).

12.3 UEFI pour remplacer le BIOS

L'*Unified Extensible Firmware Interface* (UEFI, « Interface micrologicielle extensible unifiée » donc) vient abolir les limitations énoncées ci-dessus pour le démarrage de l'ordinateur et l'amorçage du système d'exploitation. Elle est accompagnée du système de partitionnement GPT, pour *GUID Partition Table (Globally Unique IDentifier Partition Table)*, capable de gérer un disque de capacité 9,4 ZB ($9{,}4 \times 10^{21}$ octets) ou 8 ZiB (9 444 732 965 739 290 427 392 octets).

Vient en sus un autre dispositif : *Secure Boot*, destiné à empêcher le démarrage sur cet ordinateur d'un système non signé par une autorité d'accréditation, en l'occurrence Microsoft. Là les choses commencent à paraître moins séduisantes. Cette élévation de Microsoft au rang d'autorité universelle chargée de délivrer ou de refuser le droit de fonctionnement à tout système d'exploitation est un privilège exorbitant. Inutile de dire que la communauté du logiciel libre n'a pas apprécié. Dans sa grande bonté et mansuétude, Microsoft a accepté de vendre des certificats aux éditeurs des principales distributions GNU/Linux pour la modique somme de US $99, somme symbolique, mais justement c'est un symbole lourd de conséquences, puisqu'il institue la suzeraineté de Microsoft sur ceux qui accepteront d'être ses vassaux. À ce jour Ubuntu, RedHat et Fedora ont accepté de passer sous les fourches caudines, mais pas Debian. Cela dit il est possible, dans le menu du BIOS, de désactiver l'option *Secure Boot* et d'activer

2. Cf. `https://fr.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface`

`Launch CSM` (pour *Compatibility Support Module*) qui permet de s'affranchir de cette contrainte, mais en perdant une partie des avantages novateurs d'UEFI et de GPT.

Lorsque l'on achète un ordinateur équipé de Windows, le vendeur n'a en général pas envisagé que l'acheteur puisse souhaiter y installer un autre système d'exploitation, soit en remplacement de Windows, soit en partageant le disque de démarrage entre Windows et un autre système, par exemple GNU/Linux, selon le principe dit de *double boot*. Il appartient alors à l'heureux propriétaire de cet ordinateur de configurer son disque de façon à faire de la place pour le nouveau système qu'il souhaite y installer, à créer les partitions adéquates, et à le doter d'un logiciel d'amorçage qui permette de choisir au démarrage de l'ordinateur le système d'exploitation à lancer.

La suite de ce chapitre exposera la manière de configurer un disque de démarrage porteur de deux systèmes d'exploitation, en l'occurrence GNU/Linux et Windows, avec le logiciel d'amorçage GNU GRUB, sur une machine UEFI/GPT. Nous avons déjà décrit en partie cette procédure au chapitre 5 p. 114, nous examinerons ici essentiellement ce qui est nouveau avec UEFI et GPT.

12.4 Le logiciel d'amorçage GNU GRUB

12.4.1 Installation de GRUB

GNU GRUB *(GRand Unified Bootloader)* est le logiciel d'amorçage de référence pour la plupart des distributions Linux et pour Oracle Solaris. Il peut lancer d'autres systèmes d'exploitation, tels que Windows ou les différentes variantes de BSD, et il peut charger des images de systèmes par le réseau. Il supporte divers formats d'exécutables, diverses géométries de disques, tous les systèmes de fichiers Unix, ainsi que les systèmes de fichiers Windows FAT et NTFS. Surtout, il permet d'avoir sur le même disque plusieurs systèmes d'exploitation et de choisir au démarrage celui que l'on veut utiliser *(dual boot)*.

Depuis 2010 la version supportée par le projet GNU est GRUB 2, émanation du projet japonais PUPA *(Preliminary Universal Programming Architecture)* lancé en 1999 par Yoshinori K. Okuji dans le but de porter GRUB sur des architectures autres qu'Intel x86, de le rendre plus compact et modulaire, de le doter d'un langage de script, d'autoriser les caractères non-ASCII, et quelques autres améliorations.

En général l'installation de GRUB est réalisée par la procédure d'installation de toute distribution récente de GNU/Linux. Il est bien sûr possible de faire la même chose « à la main » avec les programmes utilitaires **grub-install** et **grub-**

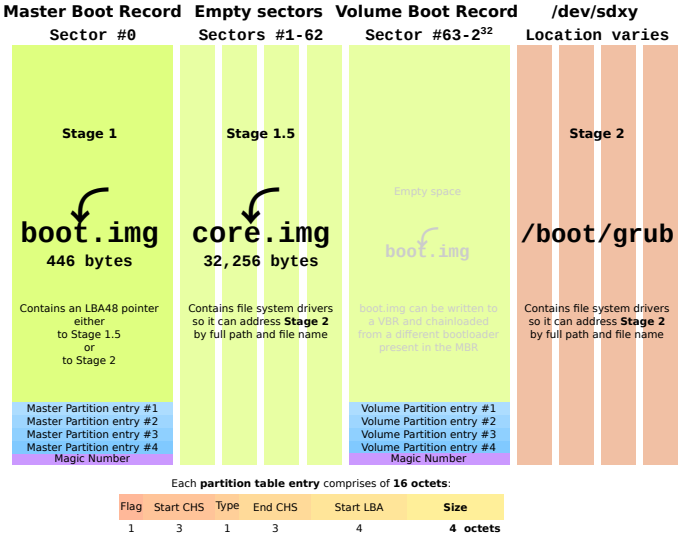


Figure 12.1 : *Organisation d'un disque avec une table de partitions dans le MBR et GRUB (auteur : Shmuel Csaba Otto Traian, source : Wikimedia Commons, licence CC BY-SA 3.0).*

mkconfig, voire en écrivant son propre fichier `/boot/grub/grub.cfg`, mais nous n'entrerons pas ici dans ces détails[3].

Un bon schéma vaut mieux qu'un long discours, en voici un qui résume la configuration de GRUB sur un ordinateur avec un BIOS à l'ancienne, et en dessous ce qu'il en est avec UEFI et GPT.

12.4.2 Partition du disque

> *Les sections qui suivent décrivent les opérations de préparation d'un disque préalablement à l'installation d'un système GNU/Linux, soit seul, soit à côté d'un système Windows existant. De telles opérations dépendent étroitement des caractéristiques du matériel et de la distribution Linux utilisés, de ce fait les indications données ici ne peuvent avoir valeur de mode d'emploi à suivre à la lettre, mais plutôt de description des problèmes à résoudre et de pistes à suivre pour y parvenir.*

Avec les BIOS à l'ancienne, la table des partitions stockée dans le MBR est limitée à quatre partitions. Pour contourner cette limite, il faut créer une parti-

3. Si vraiment on en a envie, on pourra se reporter au site du toujours excellent Chris Hoffman [62].

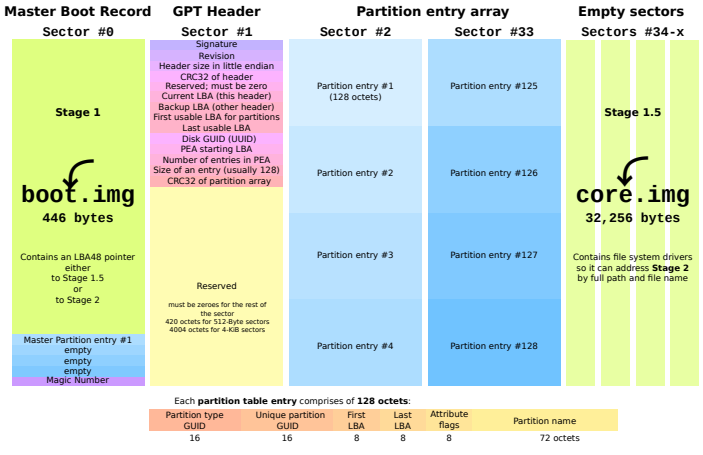


Figure 12.2 : *Organisation d'un disque avec une table de partitions GPT et GRUB (auteur : Shmuel Csaba Otto Traian, source : Wikimedia Commons, sous licence CC BY-SA 3.0).*

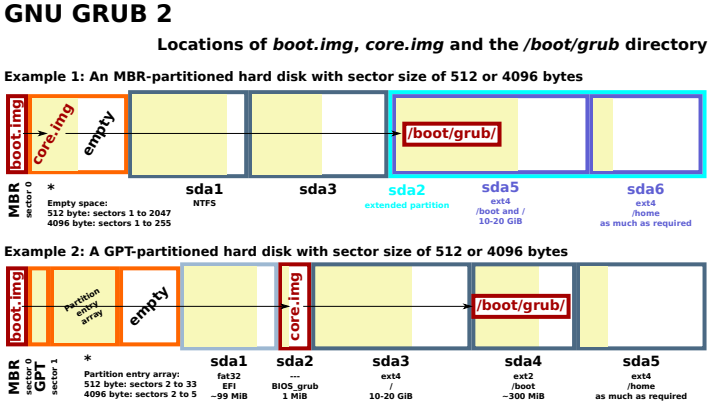


Figure 12.3 : *Partitions d'un disque dual-boot avec GRUB (auteur : Shmuel Csaba Otto Traian, source : Wikimedia Commons, sous licence CC BY-SA 3.0).*

tion dite « étendue » (partition 2 dans la figure ci-dessous), en quelque sorte une super-partition, à l'intérieur de laquelle il est possible de créer des partitions supplémentaires. Nous avons décrit au chapitre 5 p. 114 le processus de préparation du disque pour une telle configuration, qui correspond à la partie supérieure de la figure 12.3. Avec UEFI et GPT c'est plus compliqué, cette complexité est un obstacle sérieux à l'installation d'un système d'exploitation libre, même si les distributions Linux récentes facilitent le processus ; le résultat devra être conforme à la partie inférieure de la figure 12.3.

Installer Linux tout seul sur un disque

Plaçons-nous dans le cas de vouloir installer un système Linux sur un disque dur dont toutes les partitions auraient été effacées, au moyen par exemple du logiciel **gparted**. Les opérations à effectuer devront être les suivantes :

1. Créer une clé USB bootable avec le système Linux désiré dont on aura téléchargé l'image ISO, par exemple Ubuntu pour fixer les idées. Cela se fait au moyen du logiciel **UNetbootin**, disponible sous Windows, MacOS et Linux.
2. Démarrer l'ordinateur à partir de la clé USB.
3. À partir du menu proposé, lancer Ubuntu en mode « essayer sans installer ».
4. Lancer le logiciel de gestion de partition **gparted**.
5. Au début du volume créer une partition étiquetée `efi`, 250 mégaoctets suffisent, type `FAT32`, avec le drapeau `boot` (attention, il ne doit y avoir qu'une seule partition `efi` sur un disque).
6. Derrière cette partition `efi` créer une partition non formatée d'un mégaoctet avec le drapeau `bios_grub`.
7. Puis créer les partitions Linux habituelles, par exemple pour une machine personnelle 30 gigaoctets pour la partition racine `/`, 4 Go pour la partition de *swap*, le reste à partager entre `/home` et `/usr/local`. Pour un serveur il faudra une partition à part suffisamment vaste pour `/var`, qui reçoit les journaux d'exploitation et les bases de données.

Il faudra sans doute aussi accéder au menu du BIOS au moyen d'une pression, pendant le démarrage, sur la touche `F2`, ou `F9`, ou `F10`, ou `Escape`, ou `Delete`, ou `Suppr` (cela dépend du système), et une fois ce menu affiché, désactiver l'option `Secure Boot`[4] si ce n'est fait. Si la distribution Linux utilisée refuse de s'installer en mode UEFI, il faudra, toujours dans le BIOS, activer l'option `Launch CSM` (CSM = *Compatibility Support Module*), qui revient au mode de l'ancien BIOS. Il est impossible d'être plus précis parce que cela dépend des versions de BIOS ou UEFI, de la distribution Linux utilisée, etc. Il est possible que certaines opérations énumérées ci-dessus soient effectuées par la procédure d'installation de Linux. Dans tous les cas de figure, pendant la procédure d'installation, afin de garder la maîtrise de l'organisation du disque, lorsque l'on arrivera à l'étape qui demande si l'on veut installer Linux en écrasant le contenu antérieur du disque, ou si l'on

4. Ou pas : cela dépend de la distribution Linux utilisée, avec Ubuntu `Secure Boot` semble accepté. Il faudra de toute façon sans doute faire des essais, en commençant par les configurations les plus conformes à celle de départ et en descendant petit à petit vers le BIOS à l'ancienne mode. Les distributions Linux récentes de la famille Ubuntu semblent s'installer sans grandes manipulations.

veut l'installer à côté de ce qui existe déjà, ou faire autre chose, il faut choisir *faire autre chose*, qui proposera le choix d'utilisation des partitions existantes, et éventuellement la création d'autres partitions.

On pourra consulter des éléments complémentaires d'analyse sur le site d'Éric Buist [25].

Installer Linux à côté d'un Windows existant

Si le disque de l'ordinateur contient un système Windows que l'on souhaite conserver, il faut commencer par réduire la taille d'une partition Windows pour récupérer de l'espace pour Linux. Encore une fois **gparted** est le logiciel qui convient à cette tâche.

Il faudra peut-être aussi désactiver l'option `Secure Boot` et activer l'option `Launch CSM` (CSM = *Compatibility Support Module*, cf. ci-dessus). Encore une fois ces procédures ne sont pas normalisées et dépendent des versions des systèmes utilisés. Là aussi, pendant la procédure d'installation, lorsque l'on arrivera à l'étape qui demande si l'on veut installer Linux en écrasant le contenu antérieur du disque, ou si l'on veut l'installer à côté de ce qui existe déjà, ou faire autre chose, il faut choisir *faire autre chose*, qui proposera le choix d'utilisation des partitions existantes, et éventuellement la création d'autres partitions.

Voici un exemple de configuration de disque où cohabitent Linux et Windows, telle que restituée par le logiciel **fdisk**:

Disque /dev/sda : 232,9 GiB, 250 059 350 016 octets, 488 397 168 secteurs
Unités : sectors of 1 * 512 = 512 octets
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 6A2E11D5-54AB-4C7D-A5E3-7101294F5A3B

Périphérique	Start	Fin	Secteurs	Size	Type
/dev/sda1	1716224	1748991	32768	16M	Microsoft reserved
/dev/sda2	2875392	3796991	921600	450M	Windows recovery
/dev/sda3	4001792	4923391	921600	450M	Windows recovery
/dev/sda4	5128192	6049791	921600	450M	Windows recovery
/dev/sda5	6049792	6254591	204800	100M	EFI System
/dev/sda6	6254592	171670607	165416016	78,9G	Microsoft basic data
/dev/sda7	238075904	289929215	51853312	24,7G	Linux filesystem
/dev/sda8	289929216	487372575	197443360	94,2G	Linux filesystem
/dev/sda9	487372800	488396799	1024000	500M	Windows recovery
/dev/sda10	179482624	238075903	58593280	28G	Linux filesystem
/dev/sda11	171671552	179482623	7811072	3,7G	Swap Linux

Partition table entries are not in disk order.

12.5 La face obscure de l'architecture x86

Intel Management Engine

En octobre 2015 la chercheuse en sécurité informatique Joanna Rutkowska a publié un article retentissant intitulé *Intel x86 considered harmful* [113], qui étudie principalement les mécanismes peu connus du démarrage du microprocesseur, avant le lancement du système d'exploitation, et qui fait l'inventaire des failles de sécurité potentielles exploitables par un attaquant. Le résultat est très impressionnant.

Lors de la mise sous tension d'un ordinateur il faut que lui soit donnée une information d'amorçage: que faire pour commencer? Plus précisément: quelle est l'adresse de la première instruction à exécuter? On imagine bien que la modification de cette adresse par un attaquant pourrait compromettre irrémédiablement tout le fonctionnement ultérieur de quelque logiciel que ce soit, et notamment de tout système d'exploitation, aussi sûr soit-il.

Plus généralement, tout détournement malveillant d'une étape du processus d'amorçage peut corrompre toutes les opérations ultérieures, parce qu'à ce stade du démarrage aucun dispositif de protection matériel ou logiciel n'est activé, et qu'ainsi le BIOS (ou son successeur plus moderne) a accès sans restriction en mode privilégié à toute la mémoire et à tous les périphériques. Il est par exemple concevable qu'un BIOS corrompu par un attaquant lance une version corrompue du système d'exploitation. Cette question du démarrage est donc cruciale, d'autant plus qu'elle est généralement mal documentée et mal connue. C'est tout le mérite de Joanna Rutkowska d'avoir appliqué ses capacités d'investigation et de critique à ce processus, et d'avoir ainsi montré qu'en dépit d'une complexité considérable ajoutée au cours des dernières années pour en améliorer la sécurité, le système est encore vulnérable. D'où le titre de l'article.

Depuis 2014, comme nous l'avons déjà signalé ci-dessus, la plupart des ordinateurs sont livrés avec un programme de démarrage dit *Unified Extensible Firmware Interface* (UEFI), qui est essentiellement un BIOS écrit selon des principes plus modernes que ses prédécesseurs et doté de fonctions plus étendues, telles que la possibilité de disques de plus grande capacité avec un plus grand nombre de partitions. Un inconvénient majeur d'UEFI est la décision de Microsoft d'imposer aux vendeurs d'ordinateurs certifiés pour Windows 8 ou 10 l'obligation de livrer leur matériel configuré pour démarrer en mode dit *Secure Boot* contrôlé par une clé de chiffrement privée détenue par Microsoft et utilisée pour signer le noyau du système. Cette situation rend nettement plus difficile l'installation d'un système d'exploitation libre sur ces machines.

12.5.1 Système d'exploitation souterrain

Multiples sous-systèmes en micro-code

Pour corriger certaines failles de sécurité, les ingénieurs d'Intel ont multiplié les dispositifs : *Trusted Platform Module* (TPM), *Trusted Execution Technology* (TXT), exécution du code SMM *(System Management Mode)* sous contrôle d'un hyperviseur, *Active Management Technology* (AMT), *Boot Guard*, *Secure Boot* pour UEFI, et pour couronner l'édifice *Intel Management Engine* (ME). Il s'agit en fait de systèmes implantés dans le micrologiciel du processeur, c'est-à-dire hors de portée de l'utilisateur même muni de tous les privilèges sur son système. L'article de notre auteure en fait une analyse approfondie.

Ce qui est à noter, c'est que des systèmes comme AMT et ME sont actifs en permanence, *y compris lorsque l'ordinateur est arrêté* (même lorsque l'ordinateur est hors tension mais branché au secteur, certains éléments restent sous tension, comme en témoigne par exemple le clignotement du LED du connecteur au réseau Ethernet RJ45) ! C'est dire à quel niveau de contrôle arrivent ces techniques, dont la puissance est telle qu'il est à souhaiter qu'elles ne puissent pas être détournées par des malfaisants.

Intel Management Engine (ME)

Nous ne traiterons pas ici de tous ces dispositifs, pour lesquels nous renvoyons à l'article, et nous n'évoquerons que ME (*Intel Management Engine*), parce que c'est le dernier en date et qu'il englobe en quelque sorte tous les autres. ME est un petit ordinateur incorporé à *tous* les processeurs Intel contemporains. Il est impossible de l'enlever ou de le désactiver, et d'ailleurs, à supposer que l'on trouve un moyen de l'inhiber, le processeur deviendrait pratiquement inutilisable parce que beaucoup de ses fonctions dépendent de ME.

Plus qu'un simple microcontrôleur, ME est une infrastructure d'accueil complète pour toutes sortes de sous-systèmes, et de fait AMT est désormais implanté sur ME, ainsi que *Boot Guard*, PTT (*Platform Trust Technology*, la version pour ME de TPM), et d'autres à venir. ME procure bien sûr un hyperviseur qui permet par exemple l'exécution de code SMM en bac à sable (pour une récapitulation de tous ces sigles on pourra se reporter au document *Intel Hardware-based Security Technologies for Intelligent Retail Devices* [65]).

ME partout et pour tout ?

Joanna Rutkowska souligne les effets pervers de cette prise de contrôle par ME de tous les aspects du fonctionnement du système.

D'abord, imposer à tous les utilisateurs de processeurs Intel une technologie fermée et opaque en prétendant qu'elle offrirait un niveau de sécurité inégalable,

sans discussion possible, est une attitude arrogante et peu convaincante sur le fond.

Ensuite, si cette idée s'impose, et il semble que l'on n'ait guère le choix (AMD développe des technologies comparables sous le nom *Platform Security Processor*, PSP), cela conduira à ce qu'elle appelle la « zombification » des systèmes d'exploitation tels que Windows, Linux, OS-X, etc., réduits au rôle d'interfaces avec l'utilisateur pour balader des fenêtres, réagir aux déplacements de souris et jouer de la musique, cependant que les véritables traitements de données seront effectués derrière les portes closes (par des clés de chiffrement détenues par Intel) de ME, sans que l'utilisateur sache quels sont les algorithmes utilisés, et avec quel niveau de sécurité.

Comment savoir, par exemple, si Intel n'a pas décidé de confier le séquestre de ses clés de chiffrement à un tiers de confiance, par exemple une agence de sécurité gouvernementale américaine ? ME pourra-t-il filtrer et analyser nos messages électroniques, nos communications par Skype, nos recherches sur le Web ? Le générateur de nombres pseudo-aléatoires de ME comporte-t-il des faiblesses, voulues ou non ? Répondre à de telles questions est déjà difficile aujourd'hui, mais Joanna Rutkowska attire notre attention sur le fait qu'avec ME la rétro-ingénierie nécessaire à leur compréhension et à leur analyse sera d'une difficulté accrue d'un ordre de grandeur.

Et si ME était corrompu ?

C'est la question qu'il faut toujours se poser à propos d'un système tout-puissant : tant que ses actions sont bénéfiques tout va bien[5], mais s'il est détourné vers des actions maléfiques, intentionnellement ou par suite d'une attaque réussie, alors l'empire du mal risque d'être absolu.

Or, pour qui voudrait entreprendre une action maléfique, ME est l'infrastructure idéale, nous dit Joanna Rutkowska : un *rootkit* implanté dans ME aurait le contrôle total des traitements effectués par le système et des données traitées (y compris les clés privées en transit par la mémoire), et il serait pratiquement indétectable. Que ce *rootkit* soit implanté par la mafia ou par la NSA ne change rien au problème.

5. Sous réserve d'un accord unanime pour déterminer ce qui est bénéfique et ce qui ne l'est pas, et l'on sait que ce n'est vrai que dans l'imagination des dictateurs totalitaires, c'est même la définition du totalitarisme.

12.5.2 Idées pour un système plus sûr

La démarche de Joanna Rutkowska lui permet d'élaborer les idées qui mènent à un système plus sûr, par exemple :

- utilisation intensive des techniques d'isolation des différents artefacts fonctionnels les uns par rapport aux autres : virtualisation et bac à sable *(sandboxing)* en particulier ;
- exécuter en mode non-privilégié tout ce qui peut l'être, et son article démontre que c'est possible pour presque tout ce qui a trait aux périphériques, ce qui annulerait les menaces de type *Evil Maid* et DMA par exemple.

Joanna Rutkowska ne s'est pas contentée de proférer des idées, qu'elle réunit sous le vocable compartimentation, elle les a mises en pratique en organisant le laboratoire *The Invisible Things* qui a créé le système d'exploitation Qubes OS [114] fondé sur ces principes. On remarquera que cette idée de décomposer les systèmes en sous-systèmes le plus possible indépendants les uns des autres et dotés de privilèges réglés au minimum rejoint le courant des micro-noyaux, une technologie un peu oubliée mais dont les qualités de modularité, de flexibilité et d'aptitude au calcul réparti devraient permettre le retour au premier plan.

Elle constate avec dépit que de façon générale les industriels et les éditeurs de systèmes d'exploitation ne suivent pas ces principes et continuent à produire des systèmes monolithiques et donc vulnérables.

Il n'en reste pas moins que même en supposant toutes ces idées mises en œuvre, la question du démarrage reste entière, parce qu'elle est entre les mains des industriels qui conçoivent et fabriquent processeurs et cartes mères. La multiplication par Intel de technologies concurrentes ou complémentaires ne fait que compliquer la question : *Trusted Platform Module* (TPM), *Trusted Execution Technology* (TXT), exécution du code SMM *(System Management Mode)* sous contrôle d'un hyperviseur, *Boot Guard*, *Secure Boot* pour UEFI, et pour couronner l'édifice *Intel Management Engine* (ME), qui est incorporé de façon irréversible à *tous* les processeurs Intel contemporains et qui est un véritable système d'exploitation implanté en micro-code et qui vit sous le système d'exploitation que l'utilisateur croit avoir choisi.

Avec *Intel Management Engine* le vrai système d'exploitation est dans les entrailles du processeur, et Windows, Linux ou OS-X ne sont plus que des systèmes de gestion de fenêtres, d'affichage de vidéo et de diffusion de musique. Ce qui est grave dans tout cela, c'est que non seulement l'utilisateur (et même le fabricant d'ordinateur) est privé de toute liberté de choix de son système, mais qu'en outre le système imposé n'offre pas et ne peut pas offrir les garanties de sécurité auxquelles il prétend.

Conclusion

Le projet à l'origine de ce livre était de s'adresser à un public assez large, exposé à l'usage de l'informatique mais peu curieux de ses arcanes, sans doute souvent agacé par ses défaillances ou ses mystères, afin d'attirer son attention sur « les enjeux des batailles politiques qui, en ce moment, font rage », pour reprendre les mots de la préface que Christian Queinnec a bien voulu lui consacrer. J'envisageais de parler aussi peu que possible de technique et d'aboutir à un texte bref, à la limite du pamphlet. Le lecteur qui aura atteint les présentes lignes jugera de l'écart entre la visée initiale et le résultat.

Parler de technique : aussi peu que possible. Là gît la difficulté. Décrire les enjeux intellectuels et économiques induits par les systèmes d'exploitation pour des lecteurs qui en ignorent à peu près tout sans leur en expliquer les principes aurait été de la cuistrerie, pour reprendre la terminologie utilisée par Michel Volle dans son ouvrage *Le métier de statisticien* [134] désormais accessible en ligne où il signale deux écueils qui menacent le spécialiste qui parle au peuple : pédanterie et cuistrerie. J'ai entrepris une description des grands principes des systèmes, ce qui n'allait bien sûr pas sans ceux des ordinateurs et des réseaux, aussi peu que possible, bien sûr. Je rejoignais ainsi un projet suggéré par Dominique Sabrier, celui d'un ouvrage destiné à un public curieux mais non spécialiste des systèmes d'exploitation. Bref, voici un ouvrage d'introduction engagé : il y a si longtemps que la littérature engagée a disparu que l'adjectif est libre, on peut le reprendre.

Ai-je été aussi bref que possible ? Je crains que non, le sujet me plaisait trop. Me semblait possible une évocation historique des principaux systèmes d'exploitation des origines à nos jours : il est vite apparu qu'il y aurait fallu le triple de volume, au moins. Ce pan de l'histoire de l'informatique constitue un champ de recherche à lui seul, à ma connaissance encore fort peu défriché.

Le parti-pris de décrire de façon aussi générale que possible les mécanismes et l'architecture des systèmes en les illustrant d'exemples empruntés de façon non systématique à telle ou telle réalisation particulière a engendré un phénomène de sélection dont le résultat n'est pas indifférent. Cette sélection a bien sûr été biaisée par mon expérience personnelle : il y a des systèmes passionnants dont je n'ai eu qu'une connaissance livresque, tels Tenex et Tops-20 qui animaient les PDP-10 de Digital Equipment, et qui de ce fait

n'apparaissent pas dans ce livre. D'autres, peut-être les meilleurs d'un certain point de vue, sont si discrets que je les ai utilisés pendant des années sans pratiquement m'apercevoir de leur existence, comme MacOS ou Pick, et du coup je n'ai pas grand-chose à en dire, si ce n'est qu'ils m'ont rendu de bons et loyaux services.

Si l'OS 360 et Unix reviennent souvent dans mes exemples c'est bien sûr dû à une fréquentation plus longue et plus intime de ces systèmes que de tel ou tel autre, mais pas seulement. J'ai surtout emprunté à IBM sa gestion de mémoire virtuelle et son traitement des interruptions parce qu'ils sont d'une sobriété et d'une clarté conceptuelle parfaites, ce qui n'est pas si répandu. Cette mémoire virtuelle a été conçue pour être ajoutée à un système existant, ce qui imposait de réduire les interférences avec les autres composants au strict minimum et permettait en contre-partie une conception parfaitement libre du poids du passé : d'où une élégance que l'on peine à trouver dans la gestion de mémoire d'Unix, il faut le dire. Le système de fichiers qu'Unix a hérité de Multics atteint par contre un dépouillement esthétique qui n'est surpassé que par les systèmes persistants qui ignorent avec hauteur la notion même de fichier. Et Multics, que je n'ai pratiqué que pendant une courte période, m'a néanmoins fait découvrir une manière nouvelle en informatique, que j'ai retrouvée plus tard avec Unix, surtout d'ailleurs sous sa forme Linux.

Pendant dix ans j'ai travaillé avec le système VMS que *Digital Equipment (DEC)* avait créé pour les VAX. J'ai beaucoup aimé ce système stable et robuste, je l'ai même défendu au-delà du raisonnable, et je me suis demandé pourquoi j'avais si peu parlé de lui dans ce livre. J'ai eu la réponse en lisant *Inside Windows 2000* [125] de Solomon et Russinovich. Windows 2000 et VMS ont le même concepteur principal, David Cutler, un homme qui sait visiblement réaliser des systèmes de grande envergure et très fiables. Et la description des structures internes de Windows 2000 m'a irrésistiblement rappelé le cours VMS que j'avais suivi chez Digital quelques années plus tôt : les solutions retenues sont visiblement raisonnables, quelquefois même un peu trop lorsqu'elle engendrent une complexité considérable en prévision de cas de figure exceptionnels, on se dit qu'il n'y a vraiment aucune chance pour que cela tombe en panne, mais cela manque de délié, c'est un bloc massif qui résiste à l'intellection. Peut-être est-ce d'ailleurs le but : pour un industriel, qu'il soit *Digital* ou *Microsoft*, le système est un secret de fabrique et il ne faut pas que les concurrents puissent trop facilement le contrefaire ou en faire l'ingénierie inverse.

Multics et Unix, pour des raisons longuement développées ci-dessus et qui tiennent à leur origine universitaire ou proche de l'Université, ont sans doute mis longtemps à acquérir les qualités industrielles que VMS et Windows 2000 avaient pratiquement de naissance, mais leur architecture plus explicite, et pour Unix la plus grande ouverture de la structure interne, ont permis la naissance d'une véritable communauté intellectuelle dont tous les bénéfices apparaissent au grand

jour dans le mouvement du logiciel libre. Ne pas voir les origines lointaines de ce mouvement prive d'y rien comprendre, comme le montrent à l'envi les élucubrations de la presse générale ou technique qui mélange allègrement les activités des développeurs du libre, des pirates du réseau (en jouant sur les acceptions multiples du terme *hacker*) et des adolescents adeptes de jeux électroniques comme s'il ne s'agissait que d'une seule et même chose. Il ne m'échappe pas que cette confusion peut viser un but, fût-ce de flatter une clientèle.

Autre chose m'est apparu tandis que j'écrivais ce livre : l'instant était favorable à une telle entreprise pédagogique parce que nous sommes dans une période de consolidation et de simplification. Le paysage technique de l'informatique était sinon plus complexe du moins plus hétérogène il y a une quinzaine d'années. Les progrès rapides et implacables de la technologie micro-électronique et des disques magnétiques ont laminé de nombreuses filières d'innovation dont la rentabilité supposée était trop faible, que ce soit dans le domaine du matériel (processeurs vectoriels ou systoliques, multi-processeurs complexes) ou dans celui du logiciel (micro-noyaux, systèmes d'exploitation distribués). La stagnation des caractéristiques des processeurs n'est pas pour demain, mais il existe tout un stock d'innovations aujourd'hui au placard dont beaucoup ressurgiront sous une forme ou sous une autre. En attendant, la (relative) unification des techniques de base et la concentration du monde des systèmes autour de trois ou quatre variétés principales implantées sur trois ou quatre modèles de processeurs ont simplifié la tâche de l'auteur. Jusqu'à la prochaine flambée innovatrice qui sera déclenchée par une percée technologique…

Une chose en tout cas est certaine : l'invasion de domaines de plus en plus nombreux de notre vie professionnelle et privée par les systèmes d'exploitation va se poursuivre, et même s'ils sauront se faire de plus en plus discrets, voire transparents, c'est-à-dire opaques, tout en ignorer sera de plus en plus imprudent.

Annexe A Numération binaire

A.1 Définitions

Ce chapitre d'annexe emprunte une partie de son contenu à mon livre Initiation à la programmation avec Scheme*, publié en 2020 par les Éditions Technip, avec l'aimable autorisation de l'éditeur.*

Le premier procédé utilisé par l'humanité pour représenter graphiquement les nombres a sûrement été le système des « bâtons », que l'on peut appeler numération unaire. Il est encore en usage pour marquer les points au ping-pong : pour noter dix-sept points on trace dix-sept bâtons, regroupés par paquets de cinq pour faciliter la lecture. Il n'en reste pas moins que l'encombrement de la notation est proportionnel à la grandeur du nombre envisagé, ce qui est vite malcommode.

Le système que nous utilisons communément est appelé numération de position. Dans la représentation d'un nombre le chiffre le plus à droite est celui des unités, le second à partir de la droite celui des dizaines, le troisième celui des centaines, etc. Ainsi :

$$147 = 7.10^0 + 4.10^1 + 1.10^2 = 7 + 40 + 100$$

La numération de position a été inventée à Sumer il y a 4 000 ans, mais sa diffusion a été laborieuse. Notre système utilise la base 10, c'est-à-dire que les chiffres successifs à partir de la droite sont les coefficients des puissances successives de 10, mais tout nombre supérieur ou égal à 2 serait une base convenable. Les premiers comptables sumériens utilisaient la base 60 : il était logique, alors que la numération de position était une science de pointe, une acquisition intellectuelle difficile, d'utiliser une base de valeur élevée, ce qui permettait pour les usages élémentaires (nombres inférieurs à 60) de se ramener à l'ancien système, plus accessible.

Les anciens Gaulois utilisaient la base 20 dont nous voyons la trace dans les termes quatre-vingt et Quinze-Vingt, vieux noms de nombres celtiques.

La Chine antique utilisait les bases 2, 10 et 12. L'ouvrage classique de Marcel Granet *La pensée chinoise* consacre un volumineux chapitre à l'usage

des nombres par les Chinois. Ils maîtrisaient une arithmétique tout à fait respectable, mais cette discipline était tenue en piètre estime par rapport à l'usage noble des nombres : la divination par la numérologie.

Soit B un entier supérieur ou égal à 2 et N un entier strictement positif : tout entier p peut être écrit de façon unique sous la forme :

$$p = \sum_{i=0}^{N-1} d_i B^i$$

où les d_i sont des entiers compris entre 0 et $B-1$. C'est un théorème dont la démonstration est laissée en exercice au lecteur.

B est appelé la *base* de notre système de numération, $(d_0,d_1,...,d_{N-1})$ est appelé décomposition en base B du nombre p, on la notera $(d_{N-1}...d_1d_0)_B$, ou lorsqu'il n'y a pas de confusion possible sur la base utilisée simplement $d_{N-1}...d_1d_0$. Les d_i sont les *chiffres* de notre système et il est de bon ton de leur faire correspondre à chacun un symbole spécial. Si B est inférieur à 10 les chiffres arabes habituels feront l'affaire. N est le nombre de chiffres de notre nombre p en base B, une donnée importante en informatique. Ainsi le nombre 42 s'écrit en base 10 :

$$42 = 2 \times 10^0 + 4 \times 10^1 = (42)_{10} = 42$$

et en base 2 :

$$(42)_{10} = 0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5 = (101010)_2$$

Cela marche aussi bien sûr avec les chiffres après la virgule, qui sont les coefficients des puissances négatives de la base B dans la décomposition du nombre.

A.2 Petits exemples binaires

Voyons ce que donne la base 2, qui nous intéresse tout particulièrement. Énumérons les premiers nombres :

Notation décimale	Notation binaire
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
...	...
15	1111
16	10000
...	...

Il sera commode de se rappeler que $2^{10} = 1024 \simeq 10^3$, $2^{20} \simeq 10^6$, etc.

Voyons l'addition : $2 + 3$ s'écrit donc $10 + 11$ et se calcule ainsi, selon la méthode habituelle :

- un plus zéro donne un et je ne retiens rien ;
- un plus un donne deux, je pose zéro et je retiens un ;
- un plus zéro donne un, le résultat s'écrit 101 et vaut bien cinq.

A.3 Conversion entre bases quelconques

Il est parfois nécessaire de convertir un nombre p écrit dans la base B vers la base B'. La méthode, laborieuse, est la suivante :

1. Dans la base de départ B diviser (au sens de la division entière qui donne un quotient et un reste) p par la nouvelle base B'. Remarquer que le reste obtenu est forcément inférieur à B'. Diviser le quotient obtenu à nouveau par B', puis recommencer ainsi de suite jusqu'à l'obtention d'un quotient nul.
2. Si $B' > B$, convertir tous les restes de B en B'. Si $B' < B$ c'est inutile. En toute rigueur l'algorithme décrit est récursif, mais en pratique le nombre de cas à examiner est réduit et les calculs peuvent se faire de tête.
3. Écrire les restes successifs de droite à gauche : c'est le résultat cherché, l'écriture de p dans la base B'.

Voici quelques exemples. La division est posée comme vous avez appris à l'école : le dividende est à gauche du trait vertical, le diviseur est à droite, le résultat est inscrit sous le dividende, chaque reste partiel est dans le colonne de

droite, et le résultat de la conversion est le nombre constitué des chiffres qui sont les restes, dans l'ordre inverse de leur obtention, c'est-à-dire du bas vers le haut :

1000_{10} :

$1000 \lfloor \underline{2}$	0
$500 \lfloor \underline{2}$	0
$250 \lfloor \underline{2}$	0
$125 \lfloor \underline{2}$	1
$62 \lfloor \underline{2}$	0
$31 \lfloor \underline{2}$	1
$15 \lfloor \underline{2}$	1
$7 \lfloor \underline{2}$	1
$3 \lfloor \underline{2}$	1
$1 \lfloor \underline{2}$	1

$= 1111101000_2$

1000_{10} :

$1000 \lfloor \underline{3}$	1
$333 \lfloor \underline{3}$	0
$111 \lfloor \underline{3}$	0
$37 \lfloor \underline{3}$	1
$12 \lfloor \underline{3}$	0
$4 \lfloor \underline{3}$	1
$1 \lfloor \underline{3}$	1

$= 1101001_3$

1000_{10} :

$1000 \lfloor \underline{4}$	0
$250 \lfloor \underline{4}$	2
$62 \lfloor \underline{4}$	2
$15 \lfloor \underline{4}$	3
$3 \lfloor \underline{4}$	3

$= 33220_4$

1000_{10} :

$1000 \lfloor \underline{5}$	0
$200 \lfloor \underline{5}$	0
$40 \lfloor \underline{5}$	0
$8 \lfloor \underline{5}$	3
$1 \lfloor \underline{5}$	1

$= 13000_5$

1000_{10} :

$1000 \lfloor \underline{6}$	4
$166 \lfloor \underline{6}$	4
$27 \lfloor \underline{6}$	3
$4 \lfloor \underline{6}$	4

$= 4344_6$

1000_{10} :

$1000 \lfloor \underline{7}$	6
$142 \lfloor \underline{7}$	2
$20 \lfloor \underline{7}$	6
$2 \lfloor \underline{7}$	2

$= 2626_7$

1000_{10} :

$1000 \lfloor \underline{8}$	0
$125 \lfloor \underline{8}$	5
$15 \lfloor \underline{8}$	7
$1 \lfloor \underline{8}$	1

$= 1750_8$

1000_{10} :

$1000 \lfloor \underline{9}$	1
$111 \lfloor \underline{9}$	3
$12 \lfloor \underline{9}$	3
$1 \lfloor \underline{9}$	1

$= 1331_9$

1000_{10} :

$1000 \lfloor \underline{10}$	0
$100 \lfloor \underline{10}$	0
$10 \lfloor \underline{10}$	0
$1 \lfloor \underline{10}$	1

$= 1000_{10}$

1000_{10} :

$1000 \lfloor \underline{11}$	10
$90 \lfloor \underline{11}$	2
$8 \lfloor \underline{11}$	8

$= 82\mathrm{A}_{11}$

1000_{10} :

$1000 \lfloor \underline{12}$	4
$83 \lfloor \underline{12}$	11
$6 \lfloor \underline{12}$	6

$= 6\mathrm{B}4_{12}$

1000_{10} :

$1000 \lfloor \underline{13}$	12
$76 \lfloor \underline{13}$	11
$5 \lfloor \underline{13}$	5

$= 5\mathrm{BC}_{13}$

1000_{10} :

$1000 \lfloor \underline{14}$	6
$71 \lfloor \underline{14}$	1
$5 \lfloor \underline{14}$	5

$= 516_{14}$

1000_{10} :

$1000 \lfloor \underline{15}$	10
$66 \lfloor \underline{15}$	6
$4 \lfloor \underline{15}$	4

$= 46\mathrm{A}_{15}$

1000_{10} :

$1000 \lfloor \underline{16}$	8
$62 \lfloor \underline{16}$	14
$3 \lfloor \underline{16}$	3

$= 3\mathrm{E}8_{16}$

1024_{10} :

$1024\lfloor\underline{2}$	0
$512\lfloor\underline{2}$	0
$256\lfloor\underline{2}$	0
$128\lfloor\underline{2}$	0
$64\lfloor\underline{2}$	0
$32\lfloor\underline{2}$	0
$16\lfloor\underline{2}$	0
$8\lfloor\underline{2}$	0
$4\lfloor\underline{2}$	0
$2\lfloor\underline{2}$	0
$1\lfloor\underline{2}$	1

$= 10000000000_2$

1024_{10} :

$1024\lfloor\underline{16}$	0
$64\lfloor\underline{16}$	0
$4\lfloor\underline{16}$	4

$= 400_{16}$

6561_{10} :

$6561\lfloor\underline{3}$	0
$2187\lfloor\underline{3}$	0
$729\lfloor\underline{3}$	0
$243\lfloor\underline{3}$	0
$81\lfloor\underline{3}$	0
$27\lfloor\underline{3}$	0
$9\lfloor\underline{3}$	0
$3\lfloor\underline{3}$	0
$1\lfloor\underline{3}$	1

$= 100000000_3$

1000000_{10} :

$1000000\lfloor\underline{111}$	1
$9009\lfloor\underline{111}$	18
$81\lfloor\underline{111}$	81

$= 81.18.1._{111}$

1000000000_{10} :

$1000000000\lfloor\underline{111}$	1
$9009009\lfloor\underline{111}$	27
$81162\lfloor\underline{111}$	21
$731\lfloor\underline{111}$	65
$6\lfloor\underline{111}$	6

$= 6.65.21.27.1._{111}$

C'est un algorithme, il est donc programmable. Pour les chiffres après la virgule c'est un peu plus compliqué parce qu'il faut prévoir le cas des nombre dont l'écriture dans la nouvelle base nécessite une infinité de nombres après la virgule, et alors s'arrêter.

A.4 Représentation informatique des nombres entiers

Nous allons maintenant dire quelques mots de la façon dont sont représentés dans la mémoire d'un ordinateur les nombres entiers.

De façon usuelle un entier est stocké dans un mot mémoire. La taille du mot d'un ordinateur donné détermine donc la valeur absolue maximum utilisable sur cet ordinateur, ce qui nous rappelle qu'en informatique nous sommes contraints

de demeurer dans un univers fini. Une machine à mots de 32 bits autorisera des entiers compris entre $-2\,147\,483\,648$ et $+2\,147\,483\,647$.

Principe de représentation

La représentation des nombres est en général caractéristique de l'architecture d'un ordinateur donné, et non pas du langage de programmation utilisé. Les lignes qui suivent décrivent la représentation des nombres entiers en « virgule fixe » et valent pour la plupart des ordinateurs contemporains et la plupart des langages de programmation.

Si la représentation des entiers positifs se fait selon la notation de position usuelle et n'appelle pas de remarques particulières, celle des nombres négatifs se fait par la méthode du « complément à la base », qui appelle une description. Cette dernière méthode, plus complexe au premier abord, simplifie la conception des algorithmes de calcul comme nous allons le voir.

Représentation physique (chaîne de chiffres binaires)	Nombre représenté
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Tab. A.1 : *Représentation des entiers*

Soit un ordinateur dont l'architecture matérielle met à notre diposition, pour représenter les entiers, des emplacements de n positions en base B, B paire. Nous pouvons représenter B^n nombres différents : nous prenons ceux compris entre $-\frac{B^n}{2}$ et $\frac{B^n}{2} - 1$, ce qui revient à partager l'espace des représentations disponibles en deux parties égales, une pour les nombres négatifs et une pour les nombres positifs. Le plus grand nombre positif représentable a une valeur

absolue plus faible de 1 que celle du plus petit nombre négatif représentable, parce que 0 est « avec » les nombres positifs. La représentation se fera comme suit :

- Les nombres compris entre 0 et $\frac{B^n}{2}-1$, soit $\frac{B^n}{2}$ nombres, seront représentés selon la notation usuelle de position ; remarquons que le chiffre de poids fort (rang n) est toujours 0 pour ces nombres.
- Pour représenter au moyen des combinaisons restantes les $\frac{B^n}{2}$ nombres compris entre $-\frac{B^n}{2}$ et -1 nous leur ferons correspondre, dans cet ordre, les nombres à n chiffres binaires compris entre $\frac{B^n}{2}$ et B^n-1. Cela revient à dire que le chiffre de poids fort (rang n) est toujours égal à 1 et qu'un nombre négatif $-p$ sera représenté par le nombre obtenu en remplaçant chacun des chiffres de p par son complément à 1 (c'est-à dire en remplaçant chaque 1 par un 0 et chaque 0 par un 1) et en additionnant 1 au résultat, ce que l'on appelle le complément à 2.
- Prenons un exemple avec comme base $B = 2$ et $n = 4$ chiffres possibles. Les entiers représentés seront tels que décrits dans la table A.1. Le nombre $+5$ est représenté par les chiffres suivants : `0101`. Le complément à 1 de cette combinaison de chiffres nous donne : `1010`. Additionnons 1 pour avoir le complément à 2 : `1011`, qui représente -5. Si nous additionnons les deux nombres en abandonnant la dernière retenue (puisque nous n'avons que 4 chiffres par nombre) :

```
  0101
+ 1011
= 0000
```

  ce qui est conforme à notre attente.

L'intérêt de cette notation réside dans le fait que l'addition peut se faire selon le même algorithme, quel que soit le signe des opérandes, il suffira « d'oublier » la retenue éventuelle qui donnerait un $n+1^{\text{ième}}$ chiffre. Évidemment, si le calcul excède la capacité physique de la représentation, il y aura une erreur.

A.4.1 Notation hexadécimale

La représentation des nombres en base 2 est très commode pour les ordinateurs mais moins pour les humains, parce qu'elle est encombrante et peu lisible. La conversion entre base 2 et base 10 est laborieuse. Mais la conversion entre la base 2 et une base puissance de 2 est beaucoup plus maniable. La règle des faisceaux (que nous ne démontrerons pas ici) nous apprend que chaque groupe de n chiffres d'un nombre binaire correspond à un chiffre de ce nombre converti en base 2^n. Les valeurs de n souvent utilisées sont 3 et 4, soient les notations octale et hexadécimale. Les chiffres de la notation octale sont les chiffres arabes de 0 à 7, ceux de la notation hexadécimale les chiffres arabes de 0 à 9 et les lettres majuscules de A à F qui notent respectivement les nombres 10 à 15.

Ainsi le nombre binaire :

$$0111\ 1111\ \ 0011\ 1000$$

soit en décimal $32\ 568$, s'écrit-il en hexadécimal :

$$7\text{F}\ 38$$

On notera qu'un octet correspond à un nombre compris entre 0 et 255, représenté en hexadécimal par deux chiffres. Ce mode de représentation est très utilisé par les informaticiens.

A.5 Types fractionnaires

A.5.1 Les « réels »

Ces types usurpent volontiers le qualificatif « réel », et correspondent aux nombres en virgule flottante de l'ordinateur utilisé, dont la notice du constructeur et celle de l'auteur du compilateur comportent une description. Ils servent à représenter les nombres « avec des chiffres après la virgule ».

La norme IEEE 754 définit deux formats de nombres fractionnaires que l'on retrouve sur la plupart des ordinateurs. Elle est le plus généralement implantée physiquement sur l'ordinateur, c'est-à-dire que les lignes qui suivent ne s'appliquent pas à un langage particulier, mais à l'utilisation de la plupart des ordinateurs et de la plupart des langages de programmation.

Un type fractionnaire est défini sur un sous-ensemble borné, incomplet et fini des rationnels. En effet, le « nombre de chiffres après la virgule » est limité par la taille physique d'une représentation concrète. Un tel type peut être utilisé pour représenter approximativement les nombres réels.

A.5.2 Principe de représentation

Les nombres fractionnaires sont représentés dans les registres des ordinateurs selon le principe de la virgule flottante. Ce principe est inspiré de la notation familière aux scientifiques, qui préfèrent écrire 197.10^6 plutôt que 197000000.

Soit un système de numération de base B (entier positif), un nombre x pourra être représenté par le doublet :

$$[m,p] \text{ tel que : } x = m\ .\ B^p$$

m, la mantisse du nombre x, est un nombre positif compris entre :

$$1 \text{ (compris) et } B \text{ (exclus),}$$

ou nul si $x = 0$. Ceci correspondrait, en notation décimale usuelle, à des nombres tels que :

$$1{,}000000000 \text{ à } 1{,}9999999999$$

Cette mantisse m sera représentée par un nombre fixe de S chiffres binaires : elle pourra donc prendre 2^S valeurs différentes.

L'exposant p sera un entier compris entre deux valeurs MIN et MAX.

Les quatre entiers B (la base), N (le nombre de chiffres significatifs de la mantisse), MIN et MAX (les bornes de l'exposant) suffisent à définir un système de virgule flottante. Tout nombre réel de l'intervalle :

$$]-B^{MAX}, +B^{MAX}[$$

sera approché par un nombre représentable exactement, c'est-à-dire de la forme :

$$x = m \,.\, B^p$$

La norme IEEE 754 définit deux types de nombres en virgule flottante, en simple ou double précision. Le tableau ci-dessous donne aussi les caractéristiques de la double précision sur Cray YMP, qui ne respecte pas la norme.

	Simple précision	Double précision	*Cray* (double)
B	2	2	2
MIN	-126	-1022	-16382
MAX	127	1023	16383
S	24	53	48
plus petite valeur absolue	$1{,}1754944.10^{-38}$	$2{,}225073858507201.10^{-308}$	
plus grande valeur absolue	$3{,}4028235.10^{+38}$	$1{,}797693134862317.10^{+308}$	

On remarquera qu'avec des emplacements de même taille physique pour placer les nombres, Cray privilégie la largeur de l'intervalle utilisable (ce que l'on appelle la « dynamique » de la représentation) aux dépens de la précision. D'autre part la représentation IEEE est dite « normalisée », c'est-à-dire que le premier chiffre de la mantisse (devant la virgule) est toujours égal à 1 et que l'on peut donc se dispenser de le stocker, ce qui assure 53 chiffres significatifs sur 52 bits. La virgule flottante Cray n'est pas normalisée, ce qui accroît encore la dynamique et diminue la précision.

Voici à titre d'illustration le format physique d'un nombre à la norme IEEE simple précision :

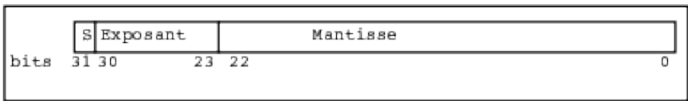


Figure A.1 : *Format d'un nombre en virgule flottante*

S — le bit de signe, 0 pour un nombre positif, 1 pour un nombre négatif ;

Exposant — exposant binaire « biaisé », c'est-à-dire que s'il est représenté sur E chiffres binaires ($E = 8$ ici), on ajoute à sa valeur effective 2^{E-1}, afin de n'avoir à représenter que des valeurs positives ;

Mantisse — une valeur fractionnaire. Un bit à 1 implicite figure « à gauche » du bit 22. La virgule est à droite du bit implicite.

En fait, la norme IEE754 est plus complexe que le résumé que nous en donnons, et elle admet des variantes. Les valeurs conventionnelles suivantes sont définies, ici en simple précision (les valeurs des mantisses et des exposants sont les configurations binaires physiques) :

Nom	Valeur	Signe	Exposant	Mantisse
zéro positif	$+0$	0	0	0
zéro négatif	-0	1	0	0
infini positif	$+\infty$	0	255	0
infini négatif	$-\infty$	1	255	0
NaN (not a number)	n/a	1 ou 0	255	$\neq 0$

A.5.3 Exemple

Un exemple simple emprunté au toujours précieux livre de Bertrand Meyer et Claude Baudoin *Méthodes de programmation* [89] illustrera quelques aspects intéressants de ce type de représentation. Soit un système où $B = 2$, $N =$

3, $MIN = -1$ et $MAX = 2$, les nombres susceptibles d'être représentés exactement sont les suivants :

	$p=-1$	$p=0$	$p=+1$	$p=+2$
$m=(1{,}00)_2=1$	$1/2$	1	2	4
$m=(1{,}01)_2=5/4$	$5/8$	$5/4$	$5/2$	5
$m=(1{,}10)_2=3/2$	$3/4$	$3/2$	3	6
$m=(1{,}11)_2=7/4$	$7/8$	$7/4$	$7/2$	7

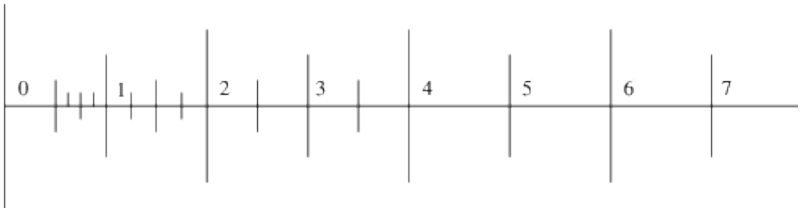


Figure A.2 : *Axe des nombres représentés*

Seules les valeurs positives ont été représentées dans le tableau et sur le graphique, les valeurs négatives s'en déduiraient par symétrie. On remarquera que la « densité » des nombres représentés exactement (ou la précision absolue de la représentation) est variable. Le lecteur pourra se convaincre facilement de ce qui suit :

- si l'on représente les nombres réels par un tel ensemble de nombres, l'opérateur d'égalité n'est pas utilisable (non plus que l'inégalité d'ailleurs) ; au mieux peut-on vérifier que la différence entre deux nombres est inférieure à un seuil que l'on se donne, et encore à condition de s'assurer que les chiffres fractionnaires qui produisent la différence sont significatifs ; pour un exposé complet et original de la question on se reportera utilement au livre de Michèle Pichat et Jean Vignes [102] ;
- la soustraction risque de provoquer une grande perte de précision dans les calculs (cas de deux nombres « grands » mais peu différents) ;
- il est dangereux d'additionner ou de soustraire des nombres d'ordres de grandeur différents, parce qu'une « mise au même exposant » sera nécessaire, au prix de la précision ;
- des changements de variable judicieux peuvent augmenter la qualité des résultats ;
- le premier chiffre de la mantisse vaut toujours 1 (on dit que la virgule flottante est normalisée), il sera donc sous-entendu dans le matériel.

Pour donner un tour plus concret à cet exposé, nous empruntons au *Numerical Computations Guide* de *Sun Microsystems* la table suivante, qui donne pour la représentation IEEE 754 simple précision la taille des intervalles entre deux nombres représentés exactement consécutifs, et ce pour différents ordres de grandeur :

x	$nextafter$	Gap
0.0	$1.1754944e-38$	$1.1754945e-38$
$1.0000000e+00$	1.0000001	$1.1920929e-07$
$2.0000000e+00$	$2.0000002e.00$	$2.3841858e-07$
$1.6000000e+01$	$1.6000002e+01$	$1.9073486e-06$
$1.2800000e+02$	$1.2800002e+02$	$1.5258789e-05$
$1.0000000e+20$	$1.0000001e+20$	$8.7960930e+12$
$9.9999997e+37$	$1.0000001e+38$	$1.0141205e+31$

Annexe B Semi-conducteurs et circuits logiques

B.1 Transistor

Nous avons vu que l'unité centrale de l'ordinateur, et notamment l'unité arithmétique et logique, était constituée de circuits logiques. Les circuits logiques réalisent matériellement les opérations de la logique, et à partir de là les opérations arithmétiques élémentaires. Il suffit pour réaliser les circuits logiques nécessaires à toutes les opérations d'un dispositif unique, dit semi-conducteur, qui en fonction d'un courant de commande laisse passer ou bloque un courant entre une source et un collecteur. C'est ce que nous allons montrer.

Le premier semi-conducteur fut la triode, inventée en 1906 par Lee De Forest et utilisée dans la construction de l'ENIAC et des premiers ordinateurs comme dans celle des anciens postes de radio « à lampes » et de luxueux amplificateurs, mais nous passerons tout de suite à son équivalent moderne, le transistor, dont l'invention aux *Bell Laboratories* en 1947 vaudra le prix Nobel 1956 à John Bardeen, Walter Houser Brattain et William Shockley.

Le but des lignes qui suivent n'est pas de donner un cours d'électronique, mais de donner à comprendre que les opérations élémentaires effectuées par les ordinateurs sont des objets physiques, et pourquoi cela peut fonctionner effectivement. Je n'entreprendrai pas l'explication des phénomènes physiques en jeu dans le transistor, que toute bonne encyclopédie en ligne ou sur papier révélera au lecteur, et je me bornerai au modèle donné par la figure B.1, qui correspond à un transistor bipolaire. Les circuits actuels utilisent plutôt des transistors à effet de champ, qui autorisent des densités plus élevées, mais avec des circuits plus complexes, et, répétons-le, le but de ce chapitre n'est pas un cours d'électronique. Pour une présentation analogue avec des transistors à effet de champ, on pourra par exemple se reporter au site d'Olivier Carton [1]. Signalons aussi la réédition récente de l'ouvrage de Paolo Zanella, Yves Ligier et Emmanuel

1. `http://www.liafa.univ-paris-diderot.fr/~carton/Enseignement/Architecture/`

Lazard, *Architecture et technologie des ordinateurs*[2], aux Éditions Dunod, et la conférence de François Anceau [7] dans le cadre d'un séminaire au Conservatoire des Arts et Métiers.

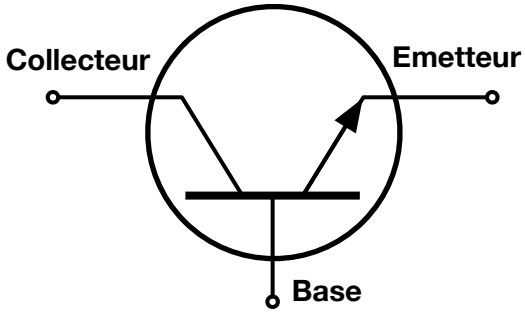


Figure B.1 : *Modèle du transistor bipolaire NPN : quand la base est mise à une tension positive, le courant passe du collecteur à l'émetteur ; quand la base est mise à une tension négative ou nulle, le courant ne passe pas.*

Quand la base est à une tension positive, le courant passe du collecteur à l'émetteur ; quand la base est à une tension négative ou nulle, le courant ne passe pas.

B.2 Algèbre de Boole

Munis du dispositif très simple qu'est le semi-conducteur (qu'il soit réalisé par une triode ou des relais peu importe), les ingénieurs des premiers circuits logiques (George Stibitz des Bell Labs en 1937, l'Allemand Konrad Zuse en 1938, et à plus grande échelle Eckert et Mauchly pour l'ENIAC) s'attaquèrent aux opérations de l'algèbre de Boole. Le mathématicien britannique George Boole (1815 – 1864) avait imaginé de formaliser la logique d'Aristote au moyen d'une algèbre d'événements qui depuis porte son nom.

Soit un ensemble d'événements A,B,C,.... À chaque événement correspond une proposition : l'événement considéré a eu lieu. Nous considérons un ensemble d'événements qui ont entre eux un certain rapport de contenu, en ceci qu'ils sont liés au résultat d'une seule et même épreuve. À chaque épreuve est attaché un certain ensemble de résultats possibles ; de chacun des

2. `http://www.dunod.com/informatique-multimedia/fondements-de-linformatique/architectures-des-machines/ouvrages-denseignemen/architecture-et-tec`

événements on doit pouvoir affirmer, pour chaque résultat de l'épreuve, s'il a eu lieu ou non[3].

Si deux événements A et B, pour chaque résultat de l'épreuve, sont toujours ou tous deux réalisés, ou tous deux non–réalisés, nous dirons qu'ils sont identiques, ce qui s'écrit $A = B$.

La non–réalisation d'un événement A est aussi un événement, qui s'écrira $\overline{A}$.

Rényi prend pour épreuve l'exemple du tir sur une cible et propose de partager la cible en quatre quadrants par un diamètre vertical et un diamètre horizontal. L'événement A sera réalisé si le coup frappe la moitié supérieure de la cible, l'événement B si le coup frappe la moitié droite de la cible.

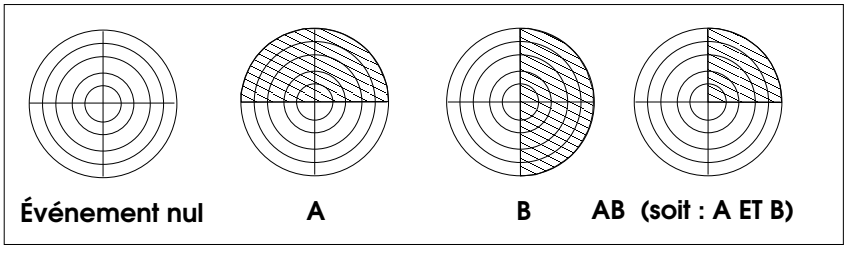


Figure B.2 : *Exemples d'événements*

Si A et B ont eu lieu tous les deux, il s'agit d'un nouvel événement, C, qui est justement « A et B ont eu lieu tout les deux », qui a lieu si le coup frappe le quadrant supérieur droit de la cible. C'est le *produit* de deux événements, que nous noterons A ET B ou $A \wedge B$ ou simplement selon l'élégante notation de Rényi :

$C = AB$

De même, on peut se demander si au moins un des deux événements A et B a eu lieu. La proposition « au moins un des deux événements A et B a eu lieu » est vraie si le coup ne frappe pas le quadrant inférieur gauche de la cible. Cet événement qui se produit quand au moins un des deux événements A et B a lieu est appelé la *somme* de A et B et s'écrit A OU B ou $A \vee B$ ou simplement : $C = A + B$.

Nous n'irons guère plus loin en algèbre de Boole. Le lecteur pourra vérifier les propriétés algébriques du produit et de la somme, qui sont « bonnes ». Nous pouvons donner les tables de vérité de ces opérations :

3. J'emprunte ce résumé de l'algèbre de Boole au *Calcul des probabilités* du mathématicien hongrois Alfred Rényi.

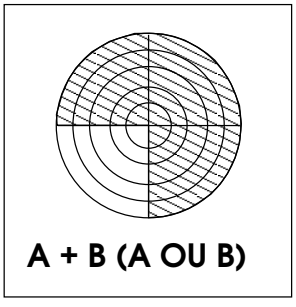


Figure B.3 : *Ou logique*

x	**y**	**xy**	**x+y**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

x	$\overline{x}$
0	1
1	0

B.3 Réalisation des opérations booléennes

Cette section doit beaucoup au livre de Patrick de Miribel *Principe des ordinateurs*[91]. Emmanuel Lazard a réalisé les schémas et les textes explicatifs qui leur correspondent.

Par convention, le vrai sera représenté par la valeur 1 et le faux par la valeur 0. À la valeur 1 correspondra un courant positif et à la valeur 0 un courant nul.

Les circuits ci-dessous comportent des résistances, symbolisées par des fils en zigzag, qui comme leur nom l'indique font obstacle au passage du courant. Si le courant trouve un chemin plus facile, comme par exemple un transistor à l'état passant, il ne franchira pas la résistance (plus exactement, le courant qui franchira la résistance sera faible et inférieur au seuil qui le rendrait efficace). Mais s'il n'y a pas d'autre chemin, par exemple parce que le transistor est à l'état bloqué, le courant franchira la résistance.

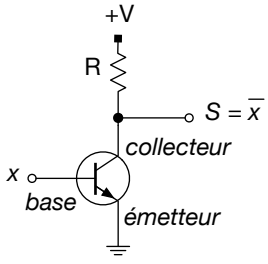


Figure B.4 : *Circuit NON*

B.3.1 Circuit NON

Si $x = 0$, la base du transistor est à un potentiel nul, le transistor est bloqué ; via la résistance, le courant positif va arriver en $\overline{x}$, qui vaudra donc 1, ce qui est bien le contraire de 0.

Si $x = 1$, le courant positif atteint la base du transistor qui devient passant. De ce fait, le point $\overline{x}$ est directement relié à la masse, donc à une tension nulle et vaudra 0, ce qui est le résultat voulu.

B.3.2 Circuit OU

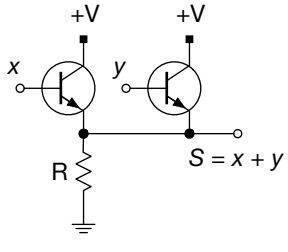


Figure B.5 : *Circuit OU*

Nous avons deux transistors en parallèle : pour que le courant positif parvienne à la sortie notée $x + y$ et lui confère ainsi la valeur 1, ou le vrai, il suffit que l'un des deux transistors soit passant. Pour cela il suffit que l'une des deux entrées, x ou y, soit positive : en effet un courant positif en x par exemple l'emportera sur la mise à la masse générée par R. C'est bien le résultat attendu.

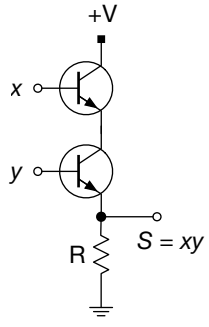


Figure B.6 : *Circuit ET*

B.3.3 Circuit ET

Nous avons deux transistors en série : pour que le courant positif atteigne la sortie notée xy il faut que les deux transistors soient passants, et donc que les deux entrées x et y soient positives, ce qui est bien le résultat voulu, conforme à la sémantique du ET.

B.3.4 Complétude de cette réalisation

Il existe d'autres opérations booléennes, mais il est aisé de démontrer qu'elles peuvent toutes se ramener à une composition des trois opérations que nous venons de voir. Il existe d'autres façons de réaliser une algèbre de Boole complète, notamment avec la seule opération NON ET (NAND), souvent utilisée par les circuits contemporains : plus touffue pour le lecteur humain, elle donne des résultats strictement équivalents à ceux que nous venons de décrire. Ce circuit est décrit par la figure B.7.

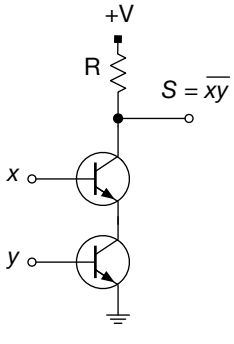


Figure B.7 : *Circuit NON ET* (NAND)

Comme les circuits NON OU (NOR) et OU EXCLUSIF (XOR) sont aussi utiles, notamment pour réaliser la mémoire, les voici dans les figures B.8 et B.9.

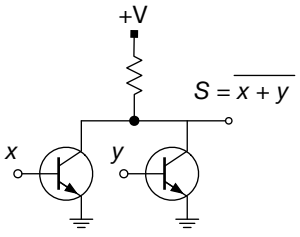


Figure B.8 : *Circuit NON OU* (NOR)

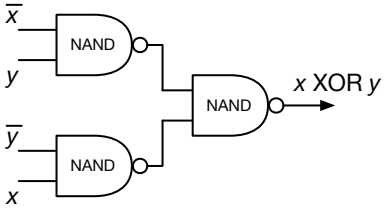


Figure B.9 : *Circuit OU EXCLUSIF* (XOR)

B.4 Construction de l'arithmétique

Munis d'une réalisation électronique de l'algèbre de Boole, nous allons montrer que nous pouvons réaliser les opérations de l'arithmétique binaire. En fait, nous allons montrer comment réaliser un opérateur électronique capable d'additionner deux chiffres binaires et de donner un chiffre de somme et un chiffre de retenue. En combinant plusieurs exemplaires de ce circuit de base il est possible de construire un additionneur à plusieurs chiffres. L'addition donne la multiplication et la soustraction, qui donne la division : autant dire que l'on a tout. Le lecteur peu assuré de sa connaissance de l'arithmétique binaire pourra se reporter à la section A.2 et plus généralement à l'annexe A.

Voici la table de vérité du semi-additionneur binaire. Soient s le chiffre de somme et r la retenue. Leibniz avait déjà remarqué la conséquence simplificatrice de l'usage de la numération binaire : la retenue et le chiffre de

somme n'ont que deux valeurs possibles, 0 ou 1[4].

x	y	s	r
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

De cette table nous pouvons inférer, par comparaison avec les tables des opérations logiques ci-dessus :

$$r = xy$$

$$s = (x + y) \cdot \overline{xy}$$

$$s = (x\ OU\ y)\ ET\ NON\ (x\ ET\ y)$$

Opérations que nous pouvons réaliser par le circuit de la figure B.10.

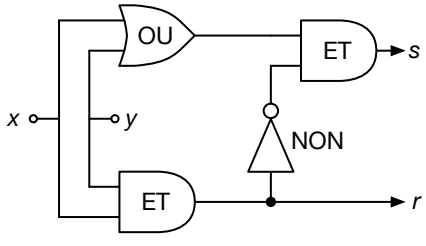


Figure B.10 : *Semi–additionneur binaire*

B.5 Construction de la mémoire

Jusque dans les années 1970 la mémoire était réalisée à partir d'éléments statiques, le plus souvent des tores de ferrite dont l'orientation du champ magnétique représentait conventionnellement la valeur d'un bit.

Aujourd'hui la mémoire est réalisée avec des circuits logiques, le plus souvent des portes NON OU (NOR). Voici la table de vérité de NON OU, qui

4. Cf. le texte de Leibniz ici : `https://laurentbloch.net/MySpip3/L-arithmetique-binaire-par-Leibniz-98`

comme son nom l'indique donne des résultats opposés à ceux du OU :

x	y	x NOR y
0	0	1
0	1	0
1	0	0
1	1	0

Une position de mémoire élémentaire, qui représente un bit, est obtenue en combinant deux circuits NON OU de telle sorte que la sortie de l'un alimente l'entrée de l'autre, et réciproquement. Un tel dispositif est appelé une *bascule* (*latch* en anglais), représentée par la figure B.11.

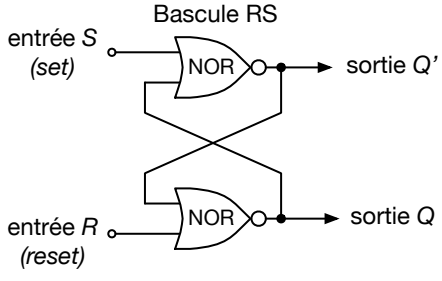


Figure B.11 : *Élément de mémoire : bascule statique*

Selon la table d'opération ci-dessus, la sortie d'une porte NON OU vaut 1 si toutes ses entrées sont à 0. Selon les tensions appliquées à ses entrées R (comme *reset*, remettre le bit à 0) et S (comme *set*, allumer le bit à 1), les sorties Q et Q' ont les valeurs indiquées dans la table ci-dessous :

S	R	Q	Q'	
1	0	1	0	Set (allumer)
0	0	1	0	Le bit vaut 1
0	1	0	1	Reset (éteindre)
0	0	0	1	Le bit vaut 0
1	1	0	0	État interdit

Le dernier état correspondrait à une situation où l'on demanderait au circuit de positionner le bit simultanément à 0 et à 1, ce qu'il semble raisonnable d'exclure. Q' a toujours la valeur complémentaire de celle de Q, soit NON Q, noté $\overline{Q}$

Un transistor non alimenté perd son état : un tel circuit doit être alimenté périodiquement, ce que l'on appelle le rafraîchissement. À ce détail technique

près on observera que les deux états possibles de ce circuit sont stables, même après le retour à un potentiel nul des entrées R et S.

On observera que la combinaison de deux objets élémentaires, ici deux portes NON OU, crée un objet qui excède de beaucoup ses composants en richesse conceptuelle : une position de mémoire est un objet beaucoup plus complexe qu'une porte logique, l'algèbre de Boole ne peut pas en rendre compte.

Annexe C Universitaires et ingénieurs avant et après Unix

Une controverse intellectuelle tacite

Unix survient une vingtaine d'années après l'invention de l'ordinateur, et une dizaine d'années après que quelques pionniers eurent compris qu'avec l'informatique une nouvelle science naissait, qu'ils eurent tenté de la faire reconnaître comme telle, et qu'ils eurent échoué dans cette tentative. Certains traits d'Unix et certains facteurs de son succès procèdent de cet échec, et c'est de cette histoire qu'il va être question ici, selon la perception que j'en ai eue de ma position de praticien. Cette perception me venait de façon rétrospective, aussi l'ordre chronologique n'est-il pas toujours respecté dans cet exposé. Ce qui suit est le récit de l'élaboration d'une vision personnelle, qui assume sa part de subjectivité.

C.1 Avant l'informatique

Entre 1936 et 1938 à Princeton Alan Turing avait bien conscience de faire de la science, mais ne soupçonnait pas que ses travaux de logique seraient un jour considérés comme la fondation théorique d'une science qui n'existait pas encore, l'informatique.

Dans les couloirs de l'IAS il croisait John von Neumann, parfois ils parlaient travail, mais pas du tout de questions de logique, domaine que von Neumann avait délibérément abandonné après la publication des travaux de Gödel [37]. En fait, ils étaient tous les deux mathématiciens, et ils parlaient des zéros de la fonction $\zeta(s)$ et de l'hypothèse de Riemann (RH). Les efforts d'Alonzo Church pour éveiller l'intérêt de ses collègues pour les travaux sur les fondements de son disciple Turing ("*On computable numbers, with an application to the Entscheidungsproblem*") rencontraient peu de succès, la question apparaissait clairement démodée.

Ce n'est qu'à la rencontre de Herman Goldstine, et par son entremise des concepteurs de l'ENIAC Eckert et Mauchly, à l'été 1944, que von Neumann s'intéressa aux calculateurs, et sans le moins du monde établir un lien entre cet

intérêt et les travaux de Turing dont il avait connaissance. Néanmoins, si Turing avait jeté les bases de l'informatique, von Neumann allait inventer l'ordinateur.

Samuel Goyet [55] a eu, lors d'une séance du séminaire *Codes sources*, une formule frappante et qui me semble exacte : avant von Neumann, programmer c'était tourner des boutons et brancher des fiches dans des tableaux de connexion, depuis von Neumann c'est écrire un texte ; cette révolution ouvrait la voie à la science informatique.

Lors d'une séance précédente du même séminaire, Liesbeth De Mol [92] avait analysé les textes de von Neumann et d'Adele et Herman Goldstine, en montrant que tout en écrivant des programmes, ils n'avaient qu'une conscience encore imprécise du type d'activité à laquelle ils s'adonnaient.

C'est donc une douzaine d'années après le *First Draft of a Report on the EDVAC* [98] que s'est éveillée la conscience de l'arrivée d'une science nouvelle, et j'en retiendrai comme manifestations les plus explicites la naissance du langage de programmation Algol, puis la naissance et la diffusion du système d'exploitation Multics. Il faudra encore une bonne quinzaine d'années pour que la bonne nouvelle se répande quelque peu parmi les praticiens de l'informatique, dont l'auteur de ces lignes, d'abord sous les espèces de la *Programmation structurée* [39, 9], qui semait l'espoir d'une sortie du bricolage. Inutile de préciser que l'esprit de bricolage est encore présent parmi les praticiens.

C.2 Algol et Multics

Algol et Multics sont les cadavres dans le placard d'Unix, même si les meurtriers ne sont pas clairement identifiés.

C.2.1 Algol

La composition des comités de rédaction des rapports Algol successifs [10] et la teneur de leurs travaux [100] me semblent marquer un point de non retour dans la constitution de l'informatique comme science.

Jusqu'alors la programmation des ordinateurs était considérée un peu comme un bricolage empirique, qui empruntait sa démarche à d'autres domaines de connaissance.

Fortran, le premier langage dit évolué, était conçu comme la transposition la plus conforme possible du formalisme mathématique, du moins c'était son ambition, déçue comme il se doit [1], et le caractère *effectuant* du texte du programme,

1. Un énoncé mathématique est essentiellement déclaratif, il décrit les propriétés d'une certaine entité, ou les relations entre certaines entités. Un programme informatique est essentielle-

qui le distingue radicalement d'une formule mathématique, plutôt que d'être signalé, était soigneusement dissimulé, notamment par l'emploi du signe = pour désigner l'opération d'affectation d'une valeur à une variable, variable au sens informatique du terme, lui aussi distinct radicalement de son acception mathématique. La conception même du langage obscurcissait la signification de ses énoncés, ce que l'on peut pardonner à John Backus parce qu'il s'aventurait dans un domaine jamais exploré avant lui. Mais cette affaire du signe = n'est pas si anecdotique qu'il y paraît, nous y reviendrons.

De même, Cobol se voulait le plus conforme possible au langage des comptables, et RPG cherchait à reproduire les habitudes professionnelles des mécanographes avec leurs tableaux de connexions.

Algol rompt brutalement avec ces compromis, il condense les idées de la révolution de la programmation annoncée par Alan Perlis [100, p. 75] en tirant toutes les conséquences (telles que perçues à l'époque) de la mission assignée au texte d'un programme : effectuer un calcul conforme à l'idée formulée par un algorithme. La syntaxe du langage ne doit viser qu'à exprimer cette idée avec précision et clarté, la lisibilité du texte en est une qualité essentielle, pour ce faire il est composé de mots qui composent des phrases. L'opération d'affectation := est clairement distinguée du prédicat d'égalité =.

La composition du comité Algol 58 est significative, la plupart des membres sont des universitaires, l'influence des industriels est modérée, Américains et Européens sont à parité (j'ai ajouté au tableau quelques personnalités influentes qui n'appartenaient pas formellement au comité IAL initial, ou qui ont rejoint ultérieurement le comité Algol 60, ou qui simplement ont joué un rôle dans cette histoire) :

---

ment impératif (ou performatif), il décrit comment faire certaines choses. Il est fondamentalement impossible de réduire l'un à l'autre, ou vice-versa, ils sont de natures différentes. Il est par contre possible, dans certains cas, d'établir une *relation* entre le texte d'un programme et un énoncé mathématique, c'est le rôle notamment des systèmes de preuve de programme. Ou, comme l'écrivent Harold Abelson et Gerald Jay Sussman [2, p. 22], *"In mathematics we are usually concerned with declarative (what is) descriptions, whereas in computer science we are usually concerned with imperative (how to) descriptions"*.

Friedrich L. Bauer	1924		Professeur Munich
Hermann Bottenbruch	1928	PhD	Ingénieur
Heinz Rutishauser	1918	PhD	Professeur ETH
Klaus Samelson	1918	PhD	Professeur Munich
John Backus	1924	MS	Ingénieur IBM
Alan Perlis	1922	PhD	Professeur Yale
Joseph Henry Wegstein	1922	MS	NIST
Adriaan van Wijngaarden	1916		U. Amsterdam
Peter Naur	1928	PhD	Professeur
Mike Woodger	1923	U. College	NPL
Bernard Vauquois	1929	Doctorat	Prof. U. Grenoble
Charles Katz	1927	MS U. Penn	Ingénieur Univac
Edsger W. Dijkstra	1930	PhD	U. of Texas
C. A. R. Hoare	1934		Prof. Oxford
Niklaus Wirth	1934	PhD	Professeur ETH
John McCarthy	1927	PhD	Prof. Stanford
M.-P. Schützenberger	1920	Doctorat	Professeur
Louis Bolliet	1928	Ingénieur	IMAG
Jacques Arsac	1929	ENS	Professeur

Les années de naissance sont significatives : il s'agit de la génération 1925 (plus ou moins).

S'ils ne figurent pas au sein des comités, Edsger W. Dijkstra et Jacques Arsac ont contribué (avec Dahl, Hoare et beaucoup d'autres) à la systématisation de leurs idées sous la forme d'une doctrine, la *programmation structurée* [39], qui a contribué à extraire ma génération de l'ignorance dans laquelle elle croupissait. Elle est souvent et abusivement réduite à une idée, le renoncement aux instructions de branchement explicite (`GOTO`), promulgué par un article célèbre de Dijkstra [45], *Go to Statement Considered Harmful*. Il s'agit plus généralement d'appliquer à la programmation les préceptes du *Discours de la Méthode*, de découper les gros programmes compliqués en petits programmes plus simples, et ainsi d'éviter la programmation en « plat de spaghettis », illisible et donc impossible à maintenir.

Rien n'exprime mieux l'aspiration de cette époque à la création d'une science nouvelle que le livre de Dijkstra *A Discipline of Programming* [47] : l'effort pour exprimer de façon rigoureuse les idées les plus ardues de la programmation va de pair avec la recherche d'un formalisme qui ne doive rien aux notations mathématiques.

Pierre-Éric Mounier-Kuhn, dans un article de 2014 [95], narre le succès initial rencontré dans les années 1960 en France par Algol 60, puis son déclin au cours des années 1970, parallèle à celui connu en d'autres pays. À la lecture de l'article on peut deviner certaines causes de cette désaffection : lors de son discours de réception du Prix Turing 1980, C.A.R. Hoare faisait l'éloge d'Algol W (l'ancêtre de Pascal) et la critique d'Algol 68, mais les deux avaient sombré. L'éclatement en

chapelles de la communauté Algol a sûrement contribué à en écarter les acteurs du monde de l'entreprise, pour qui la stabilité et la pérennité d'un système informatique est un critère de choix fondamental.

Les algolistes français, comme beaucoup des premiers informaticiens universitaires, étaient souvent des mathématiciens théoriciens déçus qui n'avaient jamais écrit un seul programme, leur souci de rigueur axiomatique n'était tempéré par aucun pragmatisme. Ils auraient voulu être reconnus comme des égaux par les mathématiciens, mais, souvent peu convaincus eux-mêmes de la valeur de leur propre discipline, ils ne pouvaient guère qu'échouer, non sans avoir au passage détourné pour longtemps la discipline informatique (telle qu'enseignée dans les universités françaises) vers des chemins de traverse mathématiques où elle n'avait rien à gagner.

Bref, Algol 68 était un beau monument intellectuel, difficile d'accès, peu apte à séduire constructeurs d'ordinateurs et utilisateurs en entreprise avec des problèmes concrets à résoudre en temps fini.

C.2.2 Multics, un système intelligent

Multics est à la science des systèmes d'exploitation ce qu'est Algol à celle des langages de programmation : un échec public, mais la source d'idées révolutionnaires et toujours d'actualité qui sont à l'origine d'une science des systèmes d'exploitation. Idées dont certaines ont d'ailleurs été largement empruntées par les créateurs d'Unix. Un récit de l'aventure Multics figure ci-dessus, au chapitre 8 p. 235.

C.2.3 Avenir de Multics

Il n'est paradoxal qu'en apparence de prédire l'avenir de ce système disparu. Certaines des idées les plus brillantes de Multics sont restées inabouties du fait des limites des ordinateurs de l'époque. Celle qui me tient le plus à cœur consiste à évacuer la notion inutile, voire nuisible, de *fichier*, pour ne garder qu'un seul dispositif d'enregistrement des données, la *mémoire* (mieux nommée en anglais *storage*), dont certains *segments* seraient pourvus de l'attribut de *persistance* [14].

Le fichier est un objet rétif à toute définition conceptuelle consistante, hérité de la mécanographie par les cartes perforées et la puissance d'IBM, mais sans aucune utilité pour un système d'exploitation d'ordinateur moderne. Il n'est que de constater la difficulté qu'il y a à expliquer à un utilisateur ordinaire pourquoi il doit sauvegarder ses documents en cours de création, et pourquoi ce qui est dans la mémoire est d'une nature différente de ce qui est sur le disque dur, terminologie qui révèle une solution de continuité injustifiée.

Les systèmes d'exploitation de demain, je l'espère, n'auront plus de fichiers, c'est déjà le cas de systèmes à micro-noyaux comme L4.

Pour Unix, tout est fichiers, y compris la mémoire. Cette unification conceptuelle est un progrès, certes. Les zélotes d'Unix défendent cette conception de la manière suivante : avec un système de mémoire virtuelle, la différence entre mémoire et fichier ne serait pas dans la persistance ou non mais dans le mode d'accès. L'accès à la mémoire se fait par des addresses, l'accès à un fichier se fait par un nom le plus souvent reflètant une structure (quasi) arborescente (le quasi est pour les liens durs Unix qui font que plusieurs chemins peuvent désigner la même chose). D'ailleurs les deux visions peuvent s'échanger. Je répondrais ceci : avec une mémoire segmentée comme celle de Multics, et conforme à ce que permettent les processeurs actuels, rien n'interdit de donner un nom aux segments persistants, ce qui permet l'unification évoquée ci-dessus, d'une façon plus satisfaisante (IMHO).

C.3 Unix

C.3.1 Les auteurs d'Unix

Que nous apprend Salus, auquel j'emprunte le générique avec lequel j'ai construit le tableau ci-desous ? Thompson et Ritchie étaient chercheurs dans une entreprise industrielle. Au fur et à mesure de leur apparition, les noms de ceux qui ont fait Unix, parmi eux Kirk McKusick, Bill Joy, Eric Allman, Keith Bostic, sont toujours accompagnés d'un commentaire de la même veine : ils étaient étudiants *undergraduates* ou en cours de PhD, et soudain ils ont découvert qu'Unix était bien plus passionnant que leurs études. Bref, les auteurs d'Unix n'ont jamais emprunté ni la voie qui mène les ingénieurs perspicaces vers les fauteuils de Directeurs Généraux, ni celle que prennent les bons étudiants vers la *tenure track*, les chaires prestigieuses, voire le Nobel.

Ken Thompson	1932	Berkeley, MS 1966	Ingénieur Bell Labs
Dennis Ritchie	1941	Harvard, PhD 1968	Ingénieur Bell Labs
Brian Kernighan	1942	Princeton (PhD)	Ingénieur Bell Labs
Stephen Bourne	1944	PhD. Trinity Col.	Ingénieur Bell Labs.
Keith Bostic	1959		Ingénieur Berkeley
Joseph Ossanna	1928	BS Wayne State U.	Ingénieur Bell Labs
Douglas McIlroy	1932	Cornell, MIT PhD	Ingénieur Bell Labs
Kirk McKusick	1954	PhD Berkeley	Ingénieur Berkeley
Eric Allman	1955	MS UC Berkeley	Ingénieur Berkeley
Bill Joy	1954	MS UC Berkeley	Ingénieur Sun
Özalp Babaoğlu	1955	PhD Berkeley	Prof. Bologne
John Lions	1937	Doc. Cambridge	Ing. Burroughs
Robert Morris	1932	MS Harvard	Ingénieur Bell Labs
Mike Lesk		PhD. Harvard	Ingénieur Bell Labs
Mike Karels		BS U. Notre Dame	Ingénieur Berkeley

Si l'on compare ce tableau avec celui des concepteurs d'Algol, on constate une différence de génération (1943 contre 1925), une plus faible propension à soutenir une thèse de doctorat, le choix de carrières d'ingénieur plutôt que d'universitaire, même si ces carrières se déroulent souvent dans un environnement de type universitaire. On notera que Joseph Ossanna avait aussi fait partie de l'équipe qui a réalisé Multics. On notera aussi la disparition des Européens, présents presque à parité dans le groupe Algol.

C.3.2 Unix occupe le terrain

Ces épisodes ont influencé tant la pratique informatique que le milieu social auxquels je participais, en général avec au moins une décennie de décalage. Cette expérience, renforcée par quelques décennies de discussions et de controverses avec quantité de collègues, m'a suggéré une hypothèse: au cours des années 1970, la génération d'Algol et de Multics a finalement perdu ses batailles, ses idées n'ont guère convaincu le monde industriel, dominé à l'époque par IBM et l'ascension des constructeurs japonais et de *Digital Equipment*. Et la génération Unix a occupé le terrain laissé vacant, par une stratégie de guérilla (du faible au fort), avec de nouvelles préoccupations et de nouveaux objectifs. Pendant ce temps les géants de l'époque ne voyaient pas venir la vague micro-informatique qui allait profondément les remettre en cause.

La mission d'un universitaire consiste, entre autres, à faire avancer la connaissance en élucidant des problèmes compliqués par l'élaboration de théories et de concepts nouveaux. Les comités Algol et le groupe Multics ont parfaitement rempli cette mission en produisant des abstractions de nature à généraliser ce qui n'était auparavant que des collections d'innombrables recettes empiriques, redondantes et contradictoires. L'élégance d'Algol resplendit surtout dans sa clarté et sa simplicité.

La mission d'un ingénieur consiste en général à procurer des solutions efficaces à des problèmes opérationnels concrets. Nul ne peut contester que ce souci d'efficacité ait été au cœur des préoccupations des auteurs d'Unix, parfois un peu trop, pas tant d'ailleurs pour le système proprement dit que pour son langage d'implémentation, C.

C.3.3 Inélégances du langage C

Certains traits du langage C me sont restés inexplicables jusqu'à ce que je suive un cours d'assembleur VAX, descendant de leur ancêtre commun, l'assembleur PDP 11. J'ai compris alors d'où venaient ces modes d'adressage biscornus et ces syntaxes à coucher dehors, justifiées certes par la capacité exiguë des mémoires disponibles à l'époque, mais de nature à décourager l'apprenti. Je préfère

la clarté, mais il est vrai que l'obscurité peut être un moyen de défense de techniciens soucieux de se mettre à l'abri des critiques.

Sans trop vouloir entrer dans la sempiternelle querelle des langages de programmation, je retiendrai deux défauts du langage C, dus clairement à un souci d'efficacité mal compris : l'emploi du signe = pour signifier l'opération d'affectation, et l'usage du caractère NUL pour marquer la fin d'une chaîne de caractères.

À l'époque où nous étions tous débutants, la distinction entre l'égalité et l'affectation fut une affaire importante. En venant de Fortran, il est clair que les idées sur la question étaient pour le moins confuses, et qu'il en résultait des erreurs cocasses ; après avoir fait de l'assistance aux utilisateurs pour leurs programmes Fortran je parle d'expérience. L'arrivée d'une distinction syntaxique claire, avec Algol, Pascal, LSE ou Ada, sans parler de Lisp, permettait de remettre les choses en place, ce fut une avancée intellectuelle dans la voie d'une vraie réflexion sur les programmes.

Les auteurs de C en ont jugé autrement : ils notent l'affectation = et l'égalité ==, avec comme argument le fait qu'en C on écrit plus souvent des affectations que des égalités, et que cela économise des frappes. Ça c'est de l'ingénierie de haut niveau ! Dans un article du bulletin 1024 de la SIF [101], une jeune doctorante explique comment, lors d'une présentation de l'informatique à des collégiens à l'occasion de la fête de la Science, une collégienne lui a fait observer que l'expression `i = i+1` qu'elle remarquait dans le texte d'un programme était fausse. Cette collégienne avait raison, et l'explication forcément controuvée qu'elle aura reçue l'aura peut-être écartée de l'informatique, parce qu'elle aura eu l'impression d'une escroquerie intellectuelle. Sa question montrait qu'elle écoutait et comprenait ce que lui disaient les professeurs, et là des idées durement acquises étaient balayées sans raison valable. Évidemment, on me répondra que la mission des ingénieurs n'est pas l'éducation des masses, mais ce n'est pas non plus de leur rendre inintelligible ce qui n'est déjà pas facile.

Pour la critique de l'usage du caractère NUL pour marquer la fin d'une chaîne de caractères je peux m'appuyer sur un renfort solide, l'article de Poul-Henning Kamp *The Most Expensive One-byte Mistake* [68]. Rappelons que ce choix malencontreux (au lieu de représenter une chaîne comme un doublet {longueur, adresse}) est à l'origine des erreurs de débordement de zone mémoire, encore aujourd'hui la faille favorite des pirates informatiques. Poul-Henning Kamp énumère dans son article les coûts induits par ce choix : coût des piratages réussis, coût des mesures de sécurité pour s'en prémunir, coût de développement de compilateurs, coût des pertes de performance imputables aux mesures de sécurité supplémentaires…

C.3.4 Élégance d'Unix

Si le langage C ne manque pas de défauts, il est néanmoins possible d'écrire des programmes C élégants, et les différentes variantes du noyau Unix en sont des exemples.

La première édition en français du manuel de système d'Andrew Tanenbaum [129] comportait le code source intégral de Minix, une version allégée d'Unix à usage pédagogique, qui devait servir d'inspiration initiale à Linus Torvalds pour Linux. Pour le commentaire du code source de Linux on se reportera avec profit au livre exhaustif et d'une grande clarté de Patrick Cegielski [29] (la première édition reposait sur la version 0.01 du noyau, beaucoup plus sobre et facile d'accès que la version ultérieure commentée pour la seconde édition à la demande de l'éditeur).

On trouvera à la fin de cette annexe les codes sources des routines principales des *schedulers* de ces systèmes. Le scheduler est l'élément principal du système d'exploitation, il distribue le temps de processeur aux différents processus en concurrence pour pouvoir s'exécuter.

Les codes de ces systèmes sont aujourd'hui facilement disponibles en ligne, ici [128] et là [132] par exemple. Par souci d'équité (d'œcuménisme ?) entre les obédiences on n'aura garde d'omettre BSD [111], ici la version historique 4BSD. Quelques extraits figurent à la fin du présent texte.

L'élégance d'Unix réside dans la sobriété et la simplicité des solutions retenues, qui n'ont pas toujours très bien résisté à la nécessité de les adapter à des architectures matérielles et logicielles de plus en plus complexes. Ainsi, la totalité des 500 super-ordinateurs les plus puissants du monde fonctionnent sous Linux, ce qui implique la capacité de coordonner plusieurs milliers de processeurs, 40 460 pour le chinois Sunway TaihuLight qui tenait la corde en 2016, dont chacun héberge plusieurs centaines de processus : le scheduler ultra-concis du noyau Linux v0.01 dont on trouvera le texte ci-dessous en serait bien incapable.

Autre élégance des versions libres d'Unix (Linux, FreeBSD, NetBSD, OpenBSD…) : le texte en est disponible, et les paramètres variables du système sont écrits dans des fichiers de texte lisible, ce qui permet à tout un chacun de les lire et d'essayer de les comprendre.

Conclusion

Finalement les universitaires orphelins d'Algol et de Multics se sont convertis en masse à Unix, ce qui assurait son hégémonie sans partage. La distribution de licences à un coût symbolique pour les institutions universitaires par les *Bell Labs*, ainsi que la disponibilité de compilateurs C gratuits, furent pour beaucoup dans ce ralliement, à une époque (1982) où le compilateur Ada que j'avais acheté

à *Digital Equipment*, avec les 80 % de réduction pour organisme de recherche, coûtait 500 000 francs.

Unix a permis pendant un temps la constitution d'une vraie communauté informatique entre universitaires et ingénieurs, ce qui fut positif. Mon avis est moins positif en ce qui concerne la diffusion du langage C: pour écrire du C en comprenant ce que l'on fait, il faut savoir pas mal de choses sur le système d'exploitation et l'architecture des ordinateurs, savoirs qui ne peuvent s'acquérir que par l'expérience de la programmation. C n'est donc pas un langage pour débutants.

Si le langage C est relativement bien adapté à l'écriture de logiciel de bas niveau, typiquement le système d'exploitation, je reste convaincu que son usage est une torture très contre-productive pour les biologistes (ce sont ceux que je connais le mieux) et autres profanes obligés de se battre avec `malloc`, `sizeof`, `typedef`, `struct` et autres pointeurs dont ils sont hors d'état de comprendre la nature et la signification. La conversion à C n'a pas vraiment amélioré la pratique du calcul scientifique, la mode récente de Python est alors un bienfait parce qu'au moins ils pourront comprendre (peut-être) le sens des programmes qu'ils écrivent.

Annexes : codes sources

Scheduler du noyau Linux v0.01 :

```
void schedule(void)
{
  int i,next,c;
  struct task_struct ** p;
/*check alarm, wake up interruptible tasks that have got
a signal*/
  for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
    if (*p) {
      if ((*p)->alarm && (*p)->alarm < jiffies) {
          (*p)->signal |= (1<<(SIGALRM-1));
          (*p)->alarm = 0;
        }
      if ((*p)->signal && (*p)->state==TASK_INTERRUPTIBLE)
        (*p)->state=TASK_RUNNING;
    }
/* this is the scheduler proper: */
  while (1) {
    c = -1;
    next = 0;
    i = NR_TASKS;
    p = &task[NR_TASKS];
    while (--i) {
      if (!*--p)
```

```
        continue;
      if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
        c = (*p)->counter, next = i;
    }
    if (c) break;
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
      if (*p)
        (*p)->counter = ((*p)->counter >> 1) +
            (*p)->priority;
  }
  switch_to(next);
```

Avec l'aide de Patrick Cegielski [29, p. 196] on observe que la variable c représente la priorité dynamique de la tâche[2] considérée. Le scheduler parcourt la table des tâches, et parmi celles qui sont dans l'état `TASK_RUNNING`, c'est-à-dire disponibles pour s'exécuter, il sélectionne celle qui a la priorité dynamique la plus élevée et lui donne la main : `switch_to(next);`

Scheduler de 4BSD (extrait) :

```
struct thread *
runq_choose(struct runq *rq)
{
  struct rqhead *rqh;
  struct thread *td;
  int pri;

  while ((pri = runq_findbit(rq)) != -1) {
    rqh = &rq->rq_queues[pri];
    td = TAILQ_FIRST(rqh);
    KASSERT(td != NULL,
      ("runq_choose: no thread on busy queue"));
    CTR3(KTR_RUNQ,
        "runq_choose: pri=%d thread=%p rqh=%p",
         pri, td, rqh);
    return (td);
  }
  CTR1(KTR_RUNQ, "runq_choose: idlethread pri=%d", pri);

  return (NULL);
}
```

2. Dans le contexte de Linux v0.01 tâche est synonyme de processus. Ultérieurement il peut y avoir une certaine confusion entre processus, tâche et *thread*, mais il s'agit toujours d'un programme candidat à l'exécution auquel le scheduler doit décider de donner ou non la main. La valeur de la variable `jiffies` est le nombre de tops d'horloge depuis le démarrage du système.

Suite page suivante

Scheduler de Minix (extrait) :

```
int main(void)
{
  /* Main routine of the scheduler. */
  message m_in; /* incoming message itself is kept here. */
  int call_nr;  /* system call number */
  int who_e;  /* caller's endpoint */
  int result; /* result to system call */
  int rv;
  int s;

  /* SEF local startup. */
  sef_local_startup();

  if (OK != (s=sys_getmachine(&machine)))
    panic("couldn't get machine info: %d", s);
/* Initialize scheduling timers,
   used for running balance_queues */
  init_scheduling();

/* This is SCHED's main loop - get work and do it,
   forever. */
  while (TRUE) {
    int ipc_status;

/*Wait for the next message, extract useful information
   from it.*/
    if (sef_receive_status(ANY, &m_in, &ipc_status) != OK)
      panic("SCHED sef_receive error");
    who_e = m_in.m_source;  /* who sent the message */
    call_nr = m_in.m_type;  /* system call number */
```

Suite page suivante

Scheduler de Minix (suite) :

```
    /* Check for system notifications, special cases.*/
    if (is_ipc_notify(ipc_status)) {
      switch(who_e) {
        case CLOCK:
          expire_timers(m_in.NOTIFY_TIMESTAMP);
          continue; /* don't reply */
        default :
          result = ENOSYS;
      }

      goto sendreply;
    }

    switch(call_nr) {
    case SCHEDULING_INHERIT:
    case SCHEDULING_START:
      result = do_start_scheduling(&m_in);
      break;
    case SCHEDULING_STOP:
      result = do_stop_scheduling(&m_in);
      break;
    case SCHEDULING_SET_NICE:
      result = do_nice(&m_in);
      break;
    case SCHEDULING_NO_QUANTUM:
  /* This message was sent from the kernel, don't reply*/
      if (IPC_STATUS_FLAGS_TEST(ipc_status,
        IPC_FLG_MSG_FROM_KERNEL)) {
        if ((rv = do_noquantum(&m_in)) != (OK)) {
          printf("SCHED: Warning, do_noquantum "
            "failed with %d\n", rv);
        }
        continue; /* Don't reply */
      }
      else {
        printf("SCHED: process %d faked "
          "SCHEDULING_NO_QUANTUM message!\n",
            who_e);
        result = EPERM;
      }
      break;
    default:
      result = no_sys(who_e, call_nr);
    }
```

Suite page suivante

```
sendreply:
```

```
    /* Send reply. */
    if (result != SUSPEND) {
      m_in.m_type = result;      /* build reply message */
      reply(who_e, &m_in);     /* send it away */
    }
  }
  return(OK);
}

/*===========================================*
 *        reply              *
 *===========================================*/
static void reply(endpoint_t who_e, message *m_ptr)
{
  int s = send(who_e, m_ptr);    /* send the message */
  if (OK != s)
    printf("SCHED: unable to send reply to %d: %d\n",
           who_e, s);
}

/*===========================================*
 *             sef_local_startup           *
 *===========================================*/
static void sef_local_startup(void)
{
  /* No init callbacks for now. */
  /* No live update support for now. */
  /* No signal callbacks for now. */

  /* Let SEF perform startup. */
  sef_startup();
}
```

Index

B

C

D

E

J

K

L

M

T

Bibliographie

[1] Janet Abbate. *Inventing the Internet*. Cambridge, MA, USA : MIT Press, 2000. isbn : 0262511150.

[2] Harold Abelson, Gerald Jay Sussman et Julie Sussman. *Structure and Interpretation of Computer Programs*. Cambridge, Massachusetts : MIT Press, 1985. url : https://mitpress.mit.edu/sicp/.

[3] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge : Cambridge University Press, 1996.

[4] Mike Acetta et al. "Mach: A New Kernel Foundation for UNIX Development". In : *Proceedings of the Summer 1986 USENIX Conference*. 1986.

[5] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. Reading, Massachusetts : Addison-Wesley, 2013.

[6] François Anceau. "La saga des PC Wintel". In : *Technique et science informatiques* 19.6 (2000).

[7] François Anceau. *La logique, des MOS aux Circuits Intégrés*. Nov. 2013. url : https://project.inria.fr/minf/files/2013/11/2-Des-MOS-aux-Circuits-Int%C3%A9gr%C3%A9s1.pdf (visité le 12/05/2018).

[8] François Armand et al. "Unix et la répartition : retour à la simplicité originelle ?" In : *Actes de la Convention UNIX'90*. 1990.

[9] Jacques J. Arsac. "Syntactic Source to Source Transforms and Program Manipulation". In : *Commun. ACM* 22.1 (jan. 1979), p. 43-54. issn : 0001-0782. doi : 10.1145/359046.359057. url : https://doi.acm.org/10.1145/359046.359057.

[10] J. W. Backus et al. "Report on the Algorithmic Language ALGOL 60". In : *Commun. ACM* (). Sous la dir. de Peter Naur. url : https://www.masswerk.at/algol60/report.htm.

[11] Georg T. Becker et al. "Stealthy Dopant-Level Hardware Trojans: Extended Version". In : *Journal of Cryptographic Engineering* (2014). url : https://www.emsec.rub.de/media/crypto/veroeffentlichungen/2015/03/19/beckerStealthyExtended.pdf.

[12] C. Gordon Bell et Allen Newell. *Computer Structures: Reading and Examples*. New York : McGraw-Hill, 1971.

[13] Didier Bert, Henri Habrias et Véronique Viguié Donzeau-Gouge. “Méthode B (numéro spécial)”. In : *Technique et science informatique* 22.1 (2003).

[14] Laurent Bloch. *Mémoire virtuelle, persistance : faut-il des fichiers ?* Fév. 2014. URL : https://laurentbloch.net/MySpip3/Memoire-virtuelle-persistance-faut.

[15] Laurent Bloch. *Révolution cyberindustrielle en France*. Paris : Economica, 2015. URL : https://laurentbloch.net/MySpip3/Revolution-cyberindustrielle-en-307.

[16] Laurent Bloch. *L'Internet, vecteur de puissance des États-Unis ? — Géopolitique du cyberespace, nouvel espace stratégique*. Paris : Éditions du Diploweb, 2017. URL : https://www.diploweb.com/_Laurent-BLOCH_.html.

[17] Laurent Bloch. *La pensée aux prises avec l'informatique – Systèmes d'information*. Texte intégral et documents complémentaires disponibles en ligne. Paris : Laurent Bloch, 2017. URL : https://laurentbloch.net/MySpip3/-Systemes-d-information-.

[18] Laurent Bloch. *Initiation à la programmation et aux algorithmes avec* Python. Paris : Technip, 2020.

[19] Laurent Bloch. *Initiation à la programmation et aux algorithmes avec* Scheme. Un livre de programmation consacré à *Scheme*, un dialecte moderne et élégant de LISP. Paris : Technip, 2020.

[20] Laurent Bloch et al. *Sécurité informatique – Pour les DSI, RSSI et administrateurs*. Paris : Éditions Eyrolles, 2016. URL : https://laurentbloch.net/MySpip3/Securite-informatique.

[21] Corrado Böhm et Giuseppe Jacopini. “Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules”. In : *Communications of the ACM (CACM)* 9.5 (mai 1966).

[22] Samia Bouzefrane. *Les systèmes d'exploitation: cours et exercices corrigés Unix, Linux et Windows XP avec C et JAVA*. Les systèmes d'exploitation constituent un domaine si complexe que de multiples approches ne l'épuisent pas. Il faut donc lire plusieurs livres, et notamment celui-ci. L'ouvrage de Samia Bouzefrane conjugue une approche conceptuelle rigoureuse et systématique, indispensable pour qui veut y voir clair, à des exemples et des exercices corrigés très concrets que le lecteur pourra tester sur son ordinateur. Paris : Dunod, 2003.

[23] Daniel P. Bovet et Marco Cesati. *Le noyau Linux*. Pour qui veut savoir vraiment comment fonctionne notre système d'exploitation préféré, et en outre beaucoup de détails intimes sur la vie des ordinateurs (notamment les processeurs Intel). Paris : O'Reilly, 2001.

[24] Frederick Brooks. *Le mythe de l'homme-mois*. S'il faut neuf mois à une femme pour faire un enfant, deux femmes ne peuvent pas y arriver en quatre mois et demie. Au-delà du rappel au bon sens dont bien des managers ont vraiment besoin, Brooks, qui fut le concepteur principal de l'OS/360, jette un regard (auto-)critique sur la plus ambitieuse entreprise d'ingénierie des cinquante dernières années : l'écriture des systèmes d'exploitation. Paris : Thomson (pour la traduction française), 1996.

[25] Éric Buist. *GRUB et l'UEFI: peuvent-ils être réunis?* URL : https://www.ericbuist.com/info/grub-uefi.xhtml (visité le 17/06/2017).

[26] Maarten Bullynck, Edgar G. Daylight et Liesbeth De Mol. "Why Did Computer Science Make a Hero out of Turing?" In : *Commun. ACM* 58.3 (fév. 2015), p. 37-39. ISSN : 0001-0782. DOI : 10.1145/2658985. URL : https://doi.acm.org/10.1145/2658985.

[27] Franck Cappello. "P2P : Développements récents et perspectives". In : *6èmes journées réseau JRES*. 2005. URL : https://2005.jres.org/slides/152.pdf.

[28] Rémy Card, Éric Dumas et Franck Mével. *Programmation Linux 2.0*. Une description détaillée de l'incarnation d'un système d'exploitation pour qui veut participer à son développement ou simplement le comprendre. Paris : Eyrolles, 1998.

[29] Patrick Cegielski. *Conception de systèmes d'exploitation – Le cas Linux*. Une analyse détaillée de Linux, avec un commentaire juxtalinéaire du code source du noyau. Dans la grande tradition de John Lions et Andrew Tanenbaum, un livre bien sûr indispensable. Paris : Eyrolles, 2003.

[30] Vinton G. Cerf et Judy O'Neill. "Oral history interview with Vinton Cerf". In : (avr. 1990). URL : https://conservancy.umn.edu/handle/11299/107214 (visité le 08/08/2018).

[31] David Chisnall. "C Is Not a Low-level Language - Your computer is not a fast PDP-11". In : *acmqueue* 16 (avr. 2018). URL : https://queue.acm.org/detail.cfm?id=3212479.

[32] Société ClearSy. "Atelier B". In : (juin 2004). URL : https://www.atelierb.societe.com/.

[33] E. F. Codd. "Multiprogram Scheduling: Parts 1 and 2. Introduction and Theory". In : *Commun. ACM* 3.6 (juin 1960), p. 347-350. ISSN : 0001-0782. DOI : 10.1145/367297.367317. URL : https://doi.acm.org/10.1145/367297.367317.

[34] Marie Coris. “Impact des logiciels libres sur l’industrie du logiciel : vers un nouveau modèle productif ?” In : *Actes du congrès JRES*. Sous la dir. de Roland Dirlewanger. 2001.

[35] Thomas Cormen et al. *Introduction à l’algorithmique*. Une somme d’une complétude impressionnante ; si les exposés mathématiques des algorithmes sont d’une grande clarté, le passage à la programmation (en pseudo-code) est souvent difficile. Paris : Dunod (pour la traduction française), 2009.

[36] Intel Corp.. *Intel IA-64 Architecture Software Developer’s Manual*. Santa Clara, Calif., USA : Intel Corp., 2001.

[37] Leo Corry. “Turing’s Pre-war Analog Computers: The Fatherhood of the Modern Computer Revisited”. In : *Commun. ACM* 60.8 (juill. 2017), p. 50-58. issn : 0001-0782. doi : 10.1145/3104032. url : https://doi.acm.org/10.1145/3104032.

[38] CROCUS. *Systèmes d’exploitation des ordinateurs*. Cet ouvrage collectif, quoique assez ancien, conserve un intérêt certain par sa rigueur dans l’introduction des concepts et du vocabulaire, et en a acquis un nouveau, de caractère historique, par la description de systèmes aujourd’hui disparus. Paris : Dunod, 1975.

[39] O. J. Dahl, E. W. Dijkstra et C. A. R. Hoare, éd. *Structured Programming*. London, UK, UK : Academic Press Ltd., 1972. isbn : 0-12-200550-3. url : https://dl.acm.org/citation.cfm?id=1243380.

[40] Jeffrey Dean. “Large-Scale Distributed Systems at Google: Current Systems and Future Directions”. In : *Large Scale Distributed Systems and Middleware (LADIS)*. Sous la dir. d’ACM SIGOPS. 2009. url : https://www.sigops.org/sosp/sosp09/ladis.html.

[41] Jeffrey Dean. *Latency Numbers Every Programmer Should Know*. 2012. url : https://gist.github.com/jboner/2841832 (visité le 29/04/2018).

[42] Alan Dearle et David Hulse. “Operating system support for persistent systems: past, present and future”. In : *Software Practice and Experience* 30 (2000), p. 295-324.

[43] Harvey M. Deitel. *An Introduction to Operating Systems*. Reading, Massachusetts : Addison-Wesley, 1984.

[44] Samuel « Doc TB » Demeulemeester. “L’épopée des microprocesseurs - Un demi-siècle d’évolution”. In : *Canard PC Hardware* 37 (juill. 2018), p. 48-61.

[45] Edsger Wybe Dijkstra. “Letters to the Editor: Go to Statement Considered Harmful”. In : *Commun. ACM* 11.3 (mars 1968), p. 147-148. issn : 0001-0782. doi : 10 . 1145 / 362929 . 362947. url : https://doi.acm.org/10.1145/362929.362947.

[46] Edsger Wybe DIJKSTRA. "The structure of the THE multiprogramming system". In : *Communications of the ACM (CACM)* 11.5 (mai 1968). URL : https://www.acm.org/classics/mar96/.

[47] Edsger Wybe DIJKSTRA. *A Discipline of Programming*. 1st. Englewood Cliffs, NJ, USA : Prentice-Hall PTR, 1976. ISBN : 013215871X.

[48] Gilles DUBERTRET. *Initiation à la cryptographie*. Paris : Vuibert, 2002.

[49] Albert DUCROCQ et André WARUSFEL. *Les mathématiques – Plaisir et nécessité*. Paris : Vuibert, 2000.

[50] Kjeld Borch EGEVANG et Paul FRANCIS. *RFC 1631 – The IP Network Address Translator (NAT)*. Rapp. tech. IETF, mai 1994. URL : https://www.ietf.org/rfc/rfc1631.txt.

[51] D. R. ENGLER, M. F. KAASHOEK et J. Jr. O'TOOLE. "Exokernel: An operating system architecture for application-level resource management". In : *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. 1995, p. 251-266.

[52] D.A. FAIRCLOUGH. "A unique microprocessor instruction set". In : *IEEE Micro* (mai 1982).

[53] Kurt GÖDEL et Jean-Yves GIRARD. *Le théorème de Gödel*. Paris : Éditions Le Seuil, 1989.

[54] Herman H. GOLDSTINE. *The Computer – from Pascal to von Neumann*. Princeton, NJ : Princeton University Press, 1972.

[55] Samuel GOYET. *Les interfaces de programmation (API) web : écriture informatique, industrie du texte et économie des passages*. Juin 2017. URL : https://codesource.hypotheses.org/241.

[56] Thomas HAIGH. "Actually, Turing Did Not Invent the Computer". In : *Commun. ACM* 57.1 (jan. 2014), p. 36-41. ISSN : 0001-0782. DOI : 10.1145/2542504. URL : https://doi.acm.org/10.1145/2542504.

[57] Brinch HANSEN. *Operating System Principles*. Prentice Hall, 1973.

[58] John L. HENNESSY et David A. PATTERSON. *Computer architecture: a quantitative approach*. San Mateo, Calif., USA : Morgan Kaufman Publishers, 2017.

[59] C. Antony R. HOARE. "Monitors: An Operating System Structuring Concept". In : *Communications of the ACM (CACM)* 17.10 (oct. 1974). URL : https://www.acm.org/classics/feb96/.

[60] C. Antony R. HOARE. *Processus séquentiels communicants*. Prentice Hall (Masson pour la traduction française), 1985.

[61] Andrew HODGES. *Alan Turing: the enigma (Alan Turing : l'énigme de l'intelligence)*. New-York, USA : Simon et Schuster (Payot, Paris pour la traduction), 1983.

[62] Chris HOFFMAN. *How to Configure the GRUB2 Boot Loader's Settings*. Sept. 2014. URL : https://www.howtogeek.com/196655/how-to-configure-the-grub2-boot-loaders-settings/ (visité le 17/06/2017).

[63] Antony HOSKING et Quintin CUTTS. "Special issue: persistent object systems". In : *Software Practice and Experience* 30-4 (2000).

[64] Christian HUITEMA. *Routing in the Internet*. Upper Saddle River, NJ, USA : Prentice Hall, 2000.

[65] INTEL. *Intel Hardware-based Security Technologies for Intelligent Retail Devices*. Nov. 2015. URL : https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/security-technologies-4th-gen-core-retail-paper.pdf.

[66] Youngbin JIN et Ben LEE. "Chapter One - A comprehensive survey of issues in solid state drives". In : sous la dir. d'Ali R. HURSON. T. 114. Advances in Computers. Elsevier, 2019, p. 1-69. DOI : https://doi.org/10.1016/bs.adcom.2019.02.001. URL : https://www.sciencedirect.com/science/article/pii/S0065245819300117.

[67] Robert E. KAHN et Judy O'NEILL. "Oral history interview with Robert E. Kahn". In : (avr. 1990). URL : https://conservancy.umn.edu/handle/11299/107387 (visité le 08/08/2018).

[68] Poul-Henning KAMP. *The Most Expensive One-byte Mistake*. Juill. 2011. URL : https://queue.acm.org/detail.cfm?id=2010365.

[69] Donald E. KNUTH. *The Art of Computer Programming*. Reading, Massachusetts : Addison-Wesley, 2011.

[70] Xeno KOVAH et Corey KALLENBERG. *How Many Million BIOSes Would you Like to Infect?* Juin 2015. URL : https://legbacore.com/Research_files/HowManyMillionBIOSesWouldYouLikeToInfect_Whitepaper_v1.pdf.

[71] Sacha KRAKOWIAK. *Principes des systèmes d'exploitation des ordinateurs*. Par un des auteurs du CROCUS, ce livre en est une suite ou une mise à jour, avec les mêmes qualités appliquées à d'autres objets. Paris : Dunod, 1987.

[72] Sacha KRAKOWIAK et Jacques MOSSIÈRE. *La naissance des systèmes d'exploitation*. Avr. 2013. URL : https://interstices.info/jcms/nn_72288/la-naissance-des-systemes-d-exploitation.

[73] Julia KRISTEVA. *Le Langage, cet inconnu - Une initiation à la linguistique*. Un tour d'horizon complet et accessible de la linguistique et de son histoire. Des proximités surprenantes avec l'informatique. Paris : Le Seuil, 1969.

[74] Emmanuel LAZARD. *Pratique performante du langage C*. Une approche systématique et qui n'évite pas les difficultés, pour un langage qui n'en manque pas. Paris : Ellipses, 2013.

[75] Emmanuel LAZARD et Pierre MOUNIER-KUHN. *Histoire illustrée de l'informatique*. Paris : EDP Sciences, 2016.

[76] Emmanuel LAZARD, Paolo ZANELLA et Yves LIGIER. *Architecture et technologie des ordinateurs*. Un tour complet et approfondi des questions techniques, à jour des dévelopements récents. Paris : Dunod, 2013.

[77] Josh LERNER et Jean TIROLE. "The Simple Economics of Open Source". In : *National Bureau of Economic Research* (2000). URL : https://www.people.hbs.edu/jlerner/simple.pdf.

[78] I.M. LESLIE et al. "The design and implementation of an operating system to support distributed multimedia applications". In : *IEEE Journal of Selected Areas in Communications* 14.7 (1996), p. 1280-1297.

[79] Jochen LIEDTKE. "On μ-Kernel Construction". In : *Proceedings of the 15th ACM Symposium on Operating System Principles*. 1995.

[80] Henri LILEN. *La saga du micro-ordinateur*. Le titre de ce livre est vraiment mérité : passionnant et passionné, illustré de documents parfois inédits, il rend notamment justice aux précurseurs français de cette industrie et éclaire le processus de décision au sein de grandes entreprises industrielles. Paris : Vuibert, 2003.

[81] John LIONS. *Lion's Commentary on Unix' 6th Edition*. Ce livre légendaire et longtemps clandestin parce qu'il comportait le code source du noyau Unix, propriété d'AT&T, mérite toujours d'être lu, même si Unix a évolué. Menlo Park, Calif. : Peer-to-Peer Communications, 1996.

[82] Keith LOEPERE. *Mach 3 Kernel Principles*. Boston : Open Software Foundation, 1991.

[83] Anil MADHAVAPEDDY et David J. SCOTT. "Unikernels : The Rise of the Virtual Library Operating System". In : *Communications of the ACM* 57.1 (), p. 61-69. URL : https://anil.recoil.org/2014/01/13/unikernels-in-cacm.html.

[84] Alexander MCKENZIE. "INWG and the Conception of the Internet: An Eyewitness Account". In : *IEEE Ann. Hist. Comput.* 33.1 (jan. 2011), p. 66-71. ISSN : 1058-6180. DOI : 10.1109/MAHC.2011.9. URL : https://dx.doi.org/10.1109/MAHC.2011.9.

[85] Marhall Kirk MCKUSICK, George V. NEVILLE-NEIL et Robert N.M WATSON. *The Design and Implementation of the FreeBSD Operating System*. Reading, Massachusetts : Addison-Wesley, 2014.

[86] Baptiste MÉLÈS. "Unix selon l'ordre des raisons : la philosophie de la pratique informatique". In : *Informatique, Philosophie, Mathématiques*. Sébastien Maronne. Toulouse, France, nov. 2013. URL : https://journals.openedition.org/philosophiascientiae/897.

[87] Alfred J. MENEZES, Paul C. van OORSCHOT et Scott A. VANSTONE. *Handbook of Applied Cryptography*. Une introduction complète au sujet, disponible en consultation sur le WWW. Boca Raton, Floride, États-Unis : CRC Press, 1996. URL : https://www.cacr.math.uwaterloo.ca/hac/.

[88] Bertrand MEYER. "The Ethics of Free Software". In : *Software Development Magazine* (mars 2000). URL : https://se.ethz.ch/~meyer/publications/softdev/ethics.pdf.

[89] Bertrand MEYER et Claude BAUDOIN. *Méthodes de programmation*. Une approche certes datée par le style, mais on y trouvera un exposé incisif des problèmes toujours au cœur de l'informatique. Paris : Eyrolles, 1984.

[90] MICRODESIGN RESOURCES, éd. *Microprocessor Report*. La revue mensuelle avec édition hebdomadaire sur le WWW du micro-processeur et de ses évolutions techniques et industrielles. Informée, compétente, beaucoup de détail technique exposé avec clarté. Sunnyvale, Calif. : Cahners Electronics Group, 1987. URL : https://www.linleygroup.com.

[91] Patrick de MIRIBEL. *Principes des ordinateurs*. Ce livre d'initiation comporte quelques-unes de ces intuitions pédagogiques qui font comprendre une fois pour toutes une question difficile, comme le fonctionnement des circuits électroniques et des ordinateurs. Paris : Dunod, 1975.

[92] Liesbeth De MOL. *Un code source sans code ? Le cas de l'ENIAC*. Juin 2016. URL : https://codesource.hypotheses.org/219.

[93] René MOREAU. *Ainsi naquit l'informatique*. Paris : Dunod, 1987.

[94] Pierre-Éric MOUNIER-KUHN. *L'Informatique en France, de la Seconde Guerre mondiale au Plan Calcul. L'émergence d'une science*. Presses de l'Université Paris-Sorbonne, 2010.

[95] Pierre-Éric MOUNIER-KUHN. "Algol in France: From Universal Project to Embedded Culture". In : *IEEE Annals of the History of Computing* 36.4 (oct. 2014).

[96] Pierre-Éric MOUNIER-KUHN et Georges-Louis BARON. "Computer Science at the CNRS and in French Universities: A Gradual Institutional Recognition". In : *Annals of the History of Computing* 12.2 (avr. 1990).

[97] Jean-Louis NEBUT. *UNIX pour l'utilisateur*. Face à l'océan des livres-mode d'emploi inodores et sans saveur, celui-ci introduit des concepts (ceux de la programmation) dans un univers d'où ils sont souvent bannis. Unix y apparaît sous un jour nouveau, doté d'une cohérence non limitée à sa structure interne, et du coup compréhensible même à qui n'en a pas lu le noyau. L'exercice (organiser cet apparent fouillis) était difficile. Paris : Éditions Technip, 1990.

[98] John von NEUMANN. *First Draft of a Report on the EDVAC*. Rapp. tech. Texte fondamental, longtemps d'un accès difficile, maintenant disponible en ligne. University of Pennsylvania, juin 1945. URL : https://fab.cba.mit.edu/classes/862.16/notes/computation/vonNeumann-1945.pdf.

[99] John von NEUMANN. *The Computer and the Brain*. Traduction française : La Découverte, Paris 1992. Ce texte d'une conférence que la mort de l'auteur a empêché d'être prononcée réfute le réductionnisme qui fleurit souvent sous de

tels titres, énumère les différences fondamentales entre l'activité du cerveau et le fonctionnement des machines numériques, ouvre de nouvelles problématiques sur des questions rebattues telle que l'existence des objets de la mathématique et de la logique. New Haven, Connecticut : Yale University Press, 1957.

[100] Alan J. Perlis. "History of Programming Languages I". In : sous la dir. de Richard L. Wexelblat. New York, NY, USA : ACM, 1981. Chap. The American Side of the Development of ALGOL, p. 75-91. isbn : 0-12-745040-8. doi : 10 . 1145 / 800025 . 1198352. url : https://doi.acm.org/10.1145/800025.1198352.

[101] Anne-Charlotte Philippe. "Quand une doctorante fait des heures supplémentaires". In : *1024 – Bulletin de la société informatique de France* Hors-série numéro 1 (fév. 2015). url : https://www.societe-informatique-de-france.fr/wp-content/uploads/2015/03/1024-hs1-philippe.pdf.

[102] Michèle Pichat et Jean Vignes. *Ingénierie du contrôle de la précision des calculs sur ordinateur*. Certains résultats scientifiques publiés à l'issue de traitements informatiques sont simplement des erreurs de calcul. Cet ouvrage donne une vision approfondie des causes possibles de telles erreurs et propose des méthodes pour les éviter, notamment la méthode originale CESTAC de contrôle et estimation stochastique des arrondis de calcul. Paris : Éditions Technip, 1993.

[103] Louis Pouzin. "Presentation and Major Design Aspects of the CYCLADES Computer Network". In : *Computer Communication Networks* (1973). Sous la dir. de R. Grimsdale et F. Kuo. Réimpression en 1975.

[104] W. Curtis Preston. *SANs and NAS*. Sebastopol, California : O'Reilly, 2002.

[105] John S. Quarterman. *The Matrix - Computer Networks and Conferencing Systems Worldwide*. Bedford, Massachusetts : Digital Press, 1990.

[106] Christian Queinnec. *ABC d'Unix*. Ce livre épuisé mais disponible sur le réseau sous licence FDL *(Free Documentation License)* accomplit un exercice délicat : dégager les concepts de la philosophie d'Unix, sans se noyer dans les détails ni omettre rien d'important. Paris : Eyrolles, 1985. url : https://pages.lip6.fr/Christian.Queinnec/Books/ABCdUNIX/uunix-toc.html.

[107] Denis Réal et al. "La rétroconception de puces électroniques, le bras armé des attaques physiques". In : *MISC*. Hors-série 7 (mai 2013).

[108] Yakov Rekhter et al. *RFC 1918 – Address Allocation for Private Internets*. Rapp. tech. IETF, fév. 1996. url : https://www.ietf.org/rfc/rfc1918.txt.

[109] Alfred Rényi. *Calcul des probabilités*. Ce livre qui a fait date dans son domaine contient un exposé particulièrement incisif et élégant de l'algèbre de Boole. Budapest [Paris] : Jacques Gabay [pour la traduction], 1966.

[110] Dennis M. Ritchie. “The Evolution of the Unix Time-sharing System”. In : *Lecture Notes in Computer Science* 79 (1980). Language Design and Programming Methodology.

[111] Jeffrey Roberson et Jake Burkholder. *FreeBSD 11.1 Scheduler*. 2017. URL : https://github.com/freebsd/freebsd/blob/master/sys/kern/sched_ule.c.

[112] Mark E. Russinovich. *Windows internals*. Redmond, État de Washington : Microsoft Press, 2016.

[113] Joanna Rutkowska. *Intel x86 considered harmful*. Oct. 2015. URL : https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf.

[114] Joanna Rutkowska, Marek Marczykowski et Wojciech Porczyk. *Qubes OS Project*. Sept. 2015. URL : https://www.qubes-os.org/.

[115] Peter H. Salus. *A Quarter Century of UNIX*. Un récit très empathique de l’aventure Unix. Reading, Massachusetts : Addison-Wesley, 1994.

[116] Peter H. Salus. *Casting the Net*. Une source de première main sur les hommes qui ont fait l’Internet. Reading, Massachusetts : Addison-Wesley, 1995.

[117] Guillaume Saupin. “Primalité et cryptographie”. In : *GNU/Linux Magazine* 214 (avr. 2018).

[118] Valérie Schafer. *La France en réseaux (vol. 1, 1960-1980)*. Paris : Nuvis, 2012. 386 p. URL : https://laurentbloch.net/MySpip3/La-France-en-reseaux-1960-1980.

[119] Valérie Schafer et Bernard Tuy. *Dans les coulisses de l’Internet. RENATER, 20 ans de technologie, d’enseignement et de recherche*. Paris : Armand Colin, 2013. 238 p.

[120] Manuel Serrano. *Vers une programmation fonctionnelle* praticable. Une réflexion pratique non dépourvue d’aperçus théoriques sur la construction de logiciels. Disponible en ligne. 2000. URL : https://laurentbloch.net/MySpip3/IMG/gz/hdr-serrano-ps.gz.

[121] Sudharsan Seshadri et al. “Willow: a user-programmable SSD”. In : *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO : USENIX Association, oct. 2014, p. 67-80. URL : https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-seshadri.pdf.

[122] Claude E. Shannon. “A mathematical theory of communication”. In : *Bell System Technical Journal* 27, p. 379-423 et 623-656 (juill. 1948). URL : https://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf.

[123] Avi Silberschatz, Peter Galvin et Greg Gagne. *Principes des systèmes d’exploitation : avec Java*. Paris : Vuibert (pour la traduction française), 2008.

[124] Simon SINGH. *The Code Book (Histoire des codes secrets)*. Paris : JC Lattès (pour la traduction française), 1999.

[125] D.A. SOLOMON et M.E. RUSSINOVICH. *Inside Windows 2000*. Redmond, État de Washington : Microsoft Press, 2000.

[126] Pyda SRISURESH et Kjeld Borch EGEVANG. *RFC 3022 – Traditional IP Network Address Translator (Traditional NAT)*. Rapp. tech. IETF, jan. 2001. URL : https://www.ietf.org/rfc/rfc3022.txt.

[127] *Tails: The amnesic incognito live system*. Nov. 2015. URL : https://tails.boum.org/.

[128] Andrew TANENBAUM et Nick COOK. *Minix 3.2.1, Scheduler main routine*. 2017. URL : https://homepages.cs.ncl.ac.uk/nick.cook/csc2025/minix/3.2.1/usr/src/servers/sched/main.c.

[129] Andrew S. TANENBAUM et Herbert BOS. *Modern Operating Systems*. Andrew Tanenbaum est l'auteur à conseiller en tout premier lieu à qui veut approfondir le sujet des systèmes d'exploitation. Non seulement la teneur est de toute première qualité, mais la clarté de l'exposé est proprement éblouissante. Upper Saddle River, New Jersey : Pearson, 2014.

[130] Andrew S. TANENBAUM et David J. WETHERALL. *Computer Networks*. Encore plus que sur les systèmes, Andrew Tanenbaum est l'auteur de base pour qui veut pénétrer les arcanes des réseaux informatiques. Tout devient clair. Upper Saddle River, New Jersey : Pearson, 2013.

[131] Paul THURROTT. *Windows Server 2003: The Road To Gold*. Jan. 2003. URL : https://www.itprotoday.com/article/windows-server/windows-server-2003-the-road-to-gold-part-one-the-early-years-127432 (visité le 21/06/2020).

[132] Linus TORVALDS. *Linux v0.01 Scheduler*. 2017. URL : https://kernel.googlesource.com/pub/scm/linux/kernel/git/nico/archive/+/v0.01/kernel/sched.c.

[133] Alan TURING et Jean-Yves GIRARD. *La machine de Turing*. Une introduction abordable mais sans concessions, puis le texte historique. Paris : Éditions Le Seuil, 1995.

[134] Michel VOLLE. *Le métier de statisticien*. Ce livre, disponible en ligne, outre une introduction de première main aux questions et aux enjeux suscités par l'exercice de la statistique publique, introduit une réflexion originale sur la déontologie du spécialiste confronté à la mission d'informer le public non spécialisé. Paris : Economica, 1984. URL : http://www.volle.com/ouvrages/metier/tabmetier.htm.

[135] Michel VOLLE. *Modèle en couches*. 2000. URL : http://www.volle.com/ouvrages/e-conomie/couches.htm (visité le 29/04/2018).

[136] Michel VOLLE. *iconomie*. Une analyse économique informée et pénétrante des nouvelles technologies par un des maîtres de l'économétrie et de la statistique. Paris : Economica, 2014.

[137] John WHARTON. "Gary Kildall, Industry Pioneer, Dead at 52". In : *Microprocessor Report* 8-10 (1994).

[138] WIKIPÉDIA. "Network address translation". In : *Wikipédia* (2017). URL : https://fr.wikipedia.org/wiki/NAT.

[139] WIKIPÉDIA. "Modèle OSI". In : *Wikipédia* (2018). URL : https://fr.wikipedia.org/wiki/Mod%C3%A8le_OSI.

[140] WIKIPÉDIA. "Pair à pair". In : *Wikipédia* (2018). URL : https://en.wikipedia.org/wiki/Peer-to-peer.

[141] Ludwig WITTGENSTEIN. *Tractatus logico-philosophicus*. Paris : Gallimard (pour la traduction française), 1918.

[142] Hubert ZIMMERMANN. "High Level Protocols Standardization: Technical and Political Issues". In : *Proceedings of Third International Conference on Computer Communication*. North-Holland, 1976. URL : https://interstices.info/jcms/c_30585/dune-informatique-centralisee-aux-reseaux-generaux-le-tournant-des-annees-1970.