SYSTÈMES D'EXPLOITATION

&

RÉSEAUX INFORMATIQUES

Notes de cours 1999-2000

par

Pierre Nugues

ISMRA,
6, boulevard du Maréchal Juin
14050 Caen

Bureau C 108, téléphone 231-45-27-05
pnugues@greyc.ismra.fr
www.ensicaen.ismra.fr/~nugues

Caen, Tous droits réservés, 1999

Révision : 27/10/1999

Introduction

Ce cours présente les principaux points théoriques du fonctionnement des systèmes d'exploitation. Il les illustre par un certain nombre d'exemples de mise en œuvre qu'il tire essentiellement du système Unix et accessoirement de Windows et de Java.

L'histoire de l'informatique est très brève – les ordinateurs sont nés avec la seconde guerre mondiale – et pourtant, elle a connu des grandes évolutions. À leur apparition, les ordinateurs étaient très coûteux et réservés aux grandes entreprises; celles-ci n'en possédaient au départ que quelques exemplaires. Ces ordinateurs « centraux » sont rapidement devenu un auxiliaire d'administration et ils se sont diffusés dans les différents services (*departments* en anglais) : financier, comptabilité, etc. Pour rendre l'informatique plus adaptée et plus abordable, des fabricants se sont alors mis à produire des mini-ordinateurs « départementaux ». Ces ordinateurs fonctionnaient avec des systèmes d'exploitation qui leur étaient propres, à chaque machine ou à chaque constructeur, par exemple, MVS pour IBM ou VMS pour DEC.

Aujourd'hui, l'informatique, aussi bien dans les entreprises, que dans la recherche ou l'enseignement, utilise des machines plus petites, fonctionnant avec des systèmes d'exploitation à caractère universel. Parmi ces systèmes d'exploitation, deux se distinguent particulièrement, un système mono-utilisateur, Windows, et un autre multi-utilisateurs et multitâches, Unix. D'une manière grossière – et contestable avec l'apparition des réseaux – on peut affirmer que le premier système est destiné à des ordinateurs individuels, tandis que l'autre est réservé au travail en groupe. Les systèmes actuels gèrent, par ailleurs une interface graphique, avec comme pionnier le Finder du Macintosh. Les systèmes d'exploitation actuels ont intégré de façon généralisée le multitâches et le service à

plusieurs utilisateurs avec la généralisation des architectures client-serveur, par exemple avec OS/2 d'IBM et Windows/NT.

Parmi ces systèmes, Unix, qui est le plus ancien, est celui qui offre le plus de richesses, le plus d'homogénéité et le plus de souplesse; il dispose, dans les versions standards, d'extensions pour les réseaux et pour le graphique. Pour cette raison, nous l'avons choisi comme le centre de ce cours[1]. Par ailleurs, le système MS-DOS puis Windows, en évoluant, ont incorporé beaucoup de caractéristiques de leur prédécesseur. Les noyaux de ces systèmes se modifieront certainement avec l'évolution des techniques. Cependant, les principes sur lesquels ils se fondent, et à plus forte raison, leur « décor », devraient rester relativement stables, au moins pour les quelques années à venir.

L'étude des systèmes d'exploitation forme une part très importante de l'informatique comme discipline et, à la différence des ses autres domaines, c'est une part qui lui est propre. Ceci au contraire de l'algorithmique ou de la logique, par exemple, qui se partagent avec les mathématiques. C'est aussi une discipline technique qui plus encore que les autres est sujette au renouvellement.

On peut diviser les systèmes d'exploitation classiques en quatre parties principales :

1. les **processus**, qui correspondent à l'exécution des programmes. Ces processus pouvant s'exécuter simultanément dans un système multitâche. Le système a pour fonction de les créer, de les gérer, de les synchroniser, ainsi que de leur permettre de communiquer entre eux;

2. la **gestion de la mémoire**, qui permet de transférer les programmes et les données nécessaires à la création des processus, d'un support secondaire, par exemple un disque, vers un support central, où a lieu l'exécution des processus.

[1] Une autre raison essentielle est la disponibilité, à l'ISMRA, d'un système Unix en réseau.

Le système devra garder la trace des parties utilisées et libres de la mémoire ainsi que gérer les transferts entre les mémoires principale et secondaire;

3. le **système de fichiers**, qui offre à l'utilisateur une vision homogène et structurée des données et des ressources : disques, mémoires, périphériques. Le système gère la création des fichiers, leur destruction, leur correspondance avec les dispositifs physiques, ainsi qu'un certain nombre d'autres caractéristiques, telles que la protection. Il les organise enfin, en général, en une structure arborescente;

4. les **entrées-sorties**, qui correspondent aux mécanismes qu'utilisent les processus pour communiquer avec l'extérieur. Ces entrées-sorties font largement appel aux couches les plus proches du matériel, et dont le système tente de masquer les particularités aux utilisateurs.

Les systèmes d'exploitation modernes intègrent par ailleurs d'autres caractéristiques. Ces dernières concernent notamment deux évolutions majeures des systèmes informatiques. La première est l'interconnexion des différentes machines et des différents systèmes par des réseaux locaux ou étendus. La seconde est la disparition des écrans de textes et leur remplacement par des dispositifs à fenêtres multiples disposant de propriétés graphiques. Ces deux techniques sont, de plus, étroitement imbriquées. Les systèmes d'exploitation fonctionnent donc, ou vont fonctionner, en réseau et ils consacreront une part importante de leurs tâches à gérer le fenêtrage, le graphisme, et les interactions entre les différentes machines. Ce cours complète les 4 parties précédentes par des études sur :

5. les **réseaux** d'ordinateurs, avec les protocoles de communication, d'interconnexion et d'application. Les réseaux permettent de mettre en œuvre une nouvelle architecture informatique fondés sur des clients et des serveurs;

6. Les **systèmes répartis** avec les protocoles d'appels de procédures à distance qui leur sont associés. Les systèmes répartis actuels trouvent des applications à

des architectures clients-serveurs de fichiers ou d'applications, tels que des bases de données.

7. les **systèmes de fenêtrage** graphique, ainsi que le modèle de serveur d'écran[2].

Le système d'exploitation correspond à l'interface entre les applications et le matériel. Le programmeur d'applications n'aborde que rarement – sinon jamais – son code interne. Il l'utilise par l'intermédiaire d'« appels système ». Les appels systèmes sont souvent accessibles à partir d'un langage de programmation, notamment en C avec le système Unix. Ces appels permettent d'effectuer la plupart des opérations sur les entités du système d'exploitation et, par exemple, de créer et détruire des processus, des fichiers, de réaliser des entrées-sorties, etc. Une terminologie tend a s'imposer pour dénommer l'ensemble des appels système, qu'ils concernent un système d'exploitation ou n'importe quelle d'application informatique : les API (*Application Programming Interface*).

Un utilisateur peut lui aussi – dans une certaine mesure – manipuler un système d'exploitation, sans pour autant avoir à créer un programme. Il le fait par l'intermédiaire d'un interprète de commandes (un « shell » en anglais) muni d'une syntaxe et éventuellement programmable. Cet interprète peut accepter les lignes de commandes comme sous MS-DOS ou sous Unix. Il peut aussi gérer les « métaphores » graphiques comme avec les Macintoshes, Windows ou X-Window.

---

[2] Ce dernier point fait partie d'un autre cours pour les étudiants de l'ENSI.

Bibliographie

La bibliographie sur les systèmes d'exploitation est très abondante et elle se renouvelle très rapidement. Elle comprend à la fois des revues de recherche, des ouvrages pédagogiques et des ouvrages sur la programmation et l'utilisation d'un système particulier. La liste que nous donnons concerne ne concerne pas la recherche et n'est absolument pas exhaustive. Par ailleurs, cette recherche est largement passée des laboratoires universitaires à ceux de quelques industriels : Microsoft et IBM notamment. Notre liste fournit seulement les références que nous pensons être les plus utiles.

Littérature générale sur les systèmes d'exploitation

A. Tanenbaum, *Modern Operating Systems,* Prentice–Hall, 1992, est une référence générale très pédagogique. *Distributed Computer Systems*, Prentice Hall, 1994, examine plus en détail les systèmes répartis. *Operating Systems, 2nd ed.,* Prentice-Hall, 1997, est une référence par le même auteur qui comprend le code, très formateur, d'un système d'exploitation voisin d'Unix. Cette édition ne comprend pas les systèmes répartis.

A. Silberschatz and P. Galvin, *Operating System Concepts*, 5th ed., Addison Wesley, 1997, est un ouvrage plus conceptuel et plus théorique que le précédent. Il reste néanmoins très clair. Il est traduit en français chez Addison-Welsey sous le titre *Principes des systèmes d'exploitation* mais peut être pas l'édition la plus récente.

Sur les noyaux de systèmes d'exploitation commerciaux

M. Bach, *La conception du système Unix*, Masson, a longtemps été l'ouvrage de référence. Sa rédaction est extrêmement lourde – à éviter après le dessert – et elle doublée d'une traduction maladroite. Cet ouvrage est cependant une mine de

renseignements pour ceux qui veulent connaître les algorithmes internes d'Unix en détail.

L'ouvrage qui précède a inspiré les firmes conceptrices d'autres systèmes. Ceci a donné lieu à plusieurs ouvrages, en général plus clairs et mieux écrits que leur ancêtre :

- H.M Deitel et M.S. Kogan, *La conception d'OS/2*, Addison-Wesley, 1992.
- Helen Custer, *Au cœur de Windows NT*, Microsoft Press, 1993.

Sur la programmation des systèmes d'exploitation

Le `man` d'Unix est la meilleure référence pour le programmeur. Il n'y a rien d'équivalent sur papier.

J.M. Rifflet, *La programmation sous Unix*, 3e éd., McGraw-Hill, 1993, est une bonne référence et un ouvrage assez complet.

Charles Petzold, *Programming Windows 95*, Microsoft Press, 1996, est une bonne référence pour apprendre la programmation Windows. Elle a été la première du genre. Actuellement, il y a des dizaines d'ouvrages équivalents.

Microsoft, *Microsoft Visual C++, Version 4*, Microsoft Press, 1995, est la référence de programmation de Microsoft. Cette encyclopédie est en 6 volumes comporte tous les appels systèmes de Windows ainsi que les MFC. Elle est disponible sous forme électronique.

Apple Computer, *Inside Macintosh Series*, Addison-Wesley, 1992, 1993, 1994, est une série traitant des caractéristiques du Macintosh. Elle détaille aussi bien les mécanismes internes que les méthodes de programmation.

Sur l'utilisation du système Unix

R.S. Bourne, *Le système Unix*, InterEditions, est une référence antique, mais qui reste un modèle de clarté. L'auteur est le concepteur du premier interprète de commande d'Unix. La traduction comprend beaucoup de fautes, par exemple dans les listings de programmes.

B. Kernighan et R. Pike, *L'environnement Unix*, InterEditions, s'axe plutôt sur les outils d'Unix. Il est plus difficile à lire que le précédent et il privilégie parfois la « bidouille » info-maniaque.

J.L. Nebut, *Unix pour l'utilisateur : Commandes et langages de commandes*, Technip, 1990, est une bonne référence sur les outils d'Unix.

Sur les langages C et C++

B. Kernighan et D. Ritchie, *Le langage C*, 2e édition, Masson, est la seule référence intéressante pour ceux qui ont déjà des notions de programmation dans un autre langage. Toute la programmation système se faisant actuellement en langage C, il est indispensable d'en posséder des notions.

P.H. Winston, *On to C++*, Addison Wesley, 1994, est un excellent manuel à la fois court et clair.

B. Stroustrup, *Le langage C++*, 2e éd., Addisson Wesley, 1992, est une référence plus complète mais plus difficile à lire que la précédente.

Sur les réseaux

Apple, *Understanding Computer Networks*, Addisson Wesley, est une référence courte mais exceptionnellement claire sur les concepts des réseaux. Les illustrations sont particulièrement explicatives.

A. Tanenbaum, *Computer Networks*, 3rd ed., Prentice-Hall, 1996, est une bonne référence souvent citée.

W. R. Stevens, *Unix Network Programming*, 2nd ed, Prentice Hall, 1997, est une excellente référence de programmation. De nombreux programmes très biens expliqués.

D. Comer, *Internetworking with TCP/IP*, Prentice Hall, 1990, (3 volumes en collaboration) est bonne référence sur l'ensemble des protocoles constituant TCP/IP.

C. Hunt, *TCP/IP Network Administration*, 2nd ed, 1997, O'Reilly & Associates, est une excellente référence pour l'administration des systèmes TCP/IP.

Sur les systèmes de fenêtrage

Young, *The X-Window System*, Prentice-Hall, 1991, est une référence très pédagogique sur la programmation de X-Window avec Motif.

Sur le langage Java

D. Flanagan, *Java in a Nutshell*, 2nd ed, O'Reilly & Associates, 1997, est une excellente référence, concise et claire.

P. Niemeyer & J. Peck, Exploring Java, 2nd Edition, O'Reilly, 1997, est aussi une excellente référence.

Chapitre 1
Les processus

1.1. Structure des processus

1.1.1. Généralités

Les processus correspondent à l'exécution de tâches : les programmes des utilisateurs, les entrées-sorties, ... par le système. Un système d'exploitation doit en général traiter plusieurs tâches en même temps. Comme il n'a, la plupart du temps, qu'un processeur, il résout ce problème grâce à un pseudo-parallélisme. Il traite une tâche à la fois, s'interrompt et passe à la suivante. La commutation des tâches étant très rapide, l'ordinateur donne l'illusion d'effectuer un traitement simultané.

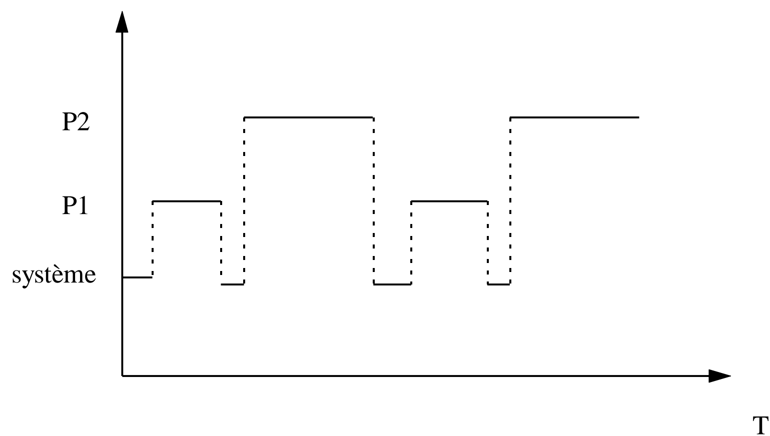


Figure 1 Le multi-tâche

Les processus des utilisateurs sont lancés par un interprète de commande. Ils peuvent eux-mêmes lancer ensuite d'autres processus. On appelle le processus créateur, le père, et les processus créés, les fils. Les processus peuvent donc se structurer sous la forme d'une arborescence. Au lancement du système, il n'existe qu'un seul processus, qui est l'ancêtre de tout les autres.

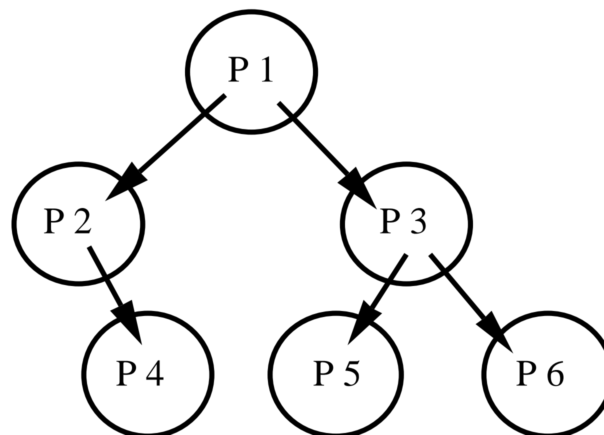


Figure 2 Le hiérarchie des processus.

Les processus sont composés d'un espace de travail en mémoire formé de 3 segments : la pile, les données et le code et d'un contexte.

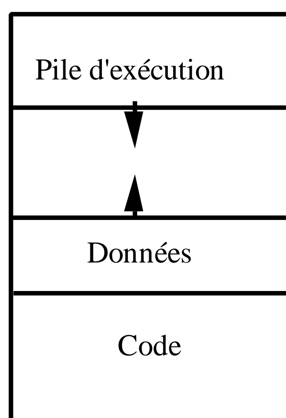


Figure 3 Les segments d'un processus.

Le code correspond aux instructions, en langage d'assemblage, du programme à exécuter. La zone de données contient les variables globales ou statiques du programme ainsi que les allocations dynamiques de mémoire. Enfin, les appels de fonctions, avec leurs paramètres et leurs variables locales, viennent s'empiler sur la pile. Les zones de pile et de données ont des frontières mobiles qui croissent en sens inverse lors de l'exécution du programme. Parfois, on partage la zone de données en données elles-mêmes et en tas. Le tas est alors réservé au données dynamiques.

Le contexte est formé des données nécessaires à la gestion des processus. Une table contient la liste de tous les processus et chaque entrée conserve leur contexte[3]. Les éléments de la table des processus de Minix, par exemple, ont la forme simplifiée du tableau ci-dessous.

Processus	Mémoire	Fichiers
Registres	Pointeur sur code	Masque `umask`
Compteur ordinal	Pointeur sur données	Répertoire racine
État du programme	Pointeur sur pile	Répertoire de travail

[3] Bach, op. cit., p. 158, et Tanenbaum, op. cit., p. 60.

Pointeur de pile	Statut de fin d'exécution	Descripteurs de fichiers
Date de création	N° de signal du proc. tué	uid effectif
Temps CP utilisé	PID	gid effectif
Temps CP des fils	Processus père	
Date de la proch. alarme	Groupe de processus	
Pointeurs sur messages	uid réel	
Bits signaux en attente	uid effectif	
PID	gid réel	
	gid effectif	
	Bits des signaux	

Tableau 1 Les éléments du contexte de Minix.

Le nombre des emplacements dans la table des processus est limité pour chaque système et pour chaque utilisateur.

La commutation des tâches, le passage d'une tâche à une autre, est réalisée par un ordonnanceur au niveau le plus bas du système. Cet ordonnanceur est activé par des interruptions :

- d'horloge,
- de disque,
- de terminaux.

À chaque interruption correspond un vecteur d'interruption, un emplacement mémoire, contenant une adresse. L'arrivée de l'interruption provoque le branchement à cette adresse. Une interruption entraîne, en général, les opérations suivantes :

- empilement du compteur ordinal, du registre d'état et peut être d'autres registres. La procédure logicielle d'interruption sauvegarde les éléments de cette pile et les autres registres dans du contexte du processus en cours;
- mise en place d'une nouvelle pile permettant le traitement des interruptions;
- appel de l'ordonnanceur;

- élection d'un nouveau programme;

Explication de l'interruption d'horloge, puis du disque.

Nous avons vu maintenant qu'un processus pouvait être actif en mémoire centrale (Élu) ou suspendu en attente d'exécution (Prêt). Il peut aussi être Bloqué, en attente de ressource, par exemple au cours d'une lecture de disque. Le diagramme simplifié des états d'un processus est donc :

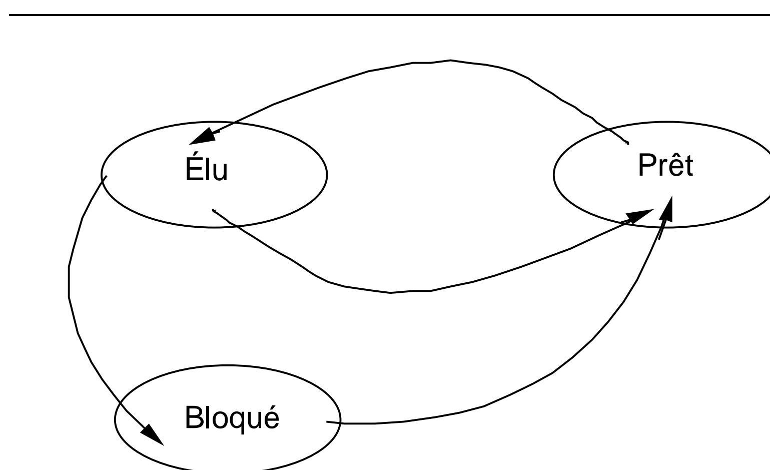


Figure 4 Les états d'un processus.

Le processus passe de l'état élu à l'état prêt et réciproquement au cours d'une intervention de l'ordonnanceur. Cet ordonnanceur se déclenchant, par exemple, sur une interruption d'horloge. Il pourra alors suspendre le processus en cours, si il a dépassé son quantum de temps, pour élire l'un des processus prêts. La transition de l'état élu à l'état bloqué se produit, par exemple, à l'occasion d'une lecture sur disque. Ce processus passera de l'état bloqué à l'état prêt lors de la réponse du disque. Ce passage correspond d'une manière générale à la libération d'une ressource.

En fait, sous Unix, l'exécution d'un processus se fait sous deux modes, le mode utilisateur et le mode noyau. Le mode noyau correspond aux appels au code du noyau : `write`, `read`, ... Le mode utilisateur correspond aux autres instructions.

Un processus en mode noyau ne peut être suspendu par l'ordonnanceur. Il passe à l'état préempté à la fin de son exécution dans ce mode – à moins d'être bloqué ou détruit –. Cet état est virtuellement le même que prêt en mémoire.

Par ailleurs, lorsque la mémoire ne peut contenir tous les processus prêts, un certain nombre d'entre eux sont déplacés sur le disque. Un processus peut alors être prêt en mémoire ou prêt sur le disque. De la même manière, on a des processus bloqués en mémoire ou bien bloqués sur disque.

Enfin, les processus terminés ne sont pas immédiatement éliminés de la tables des processus – ils ne le sont que par une instruction explicite de leur père –. Ceci correspond à l'état défunt.

Les états d'un processus Unix[4] sont alors :

[4]Bach, op.cit., p. 156.

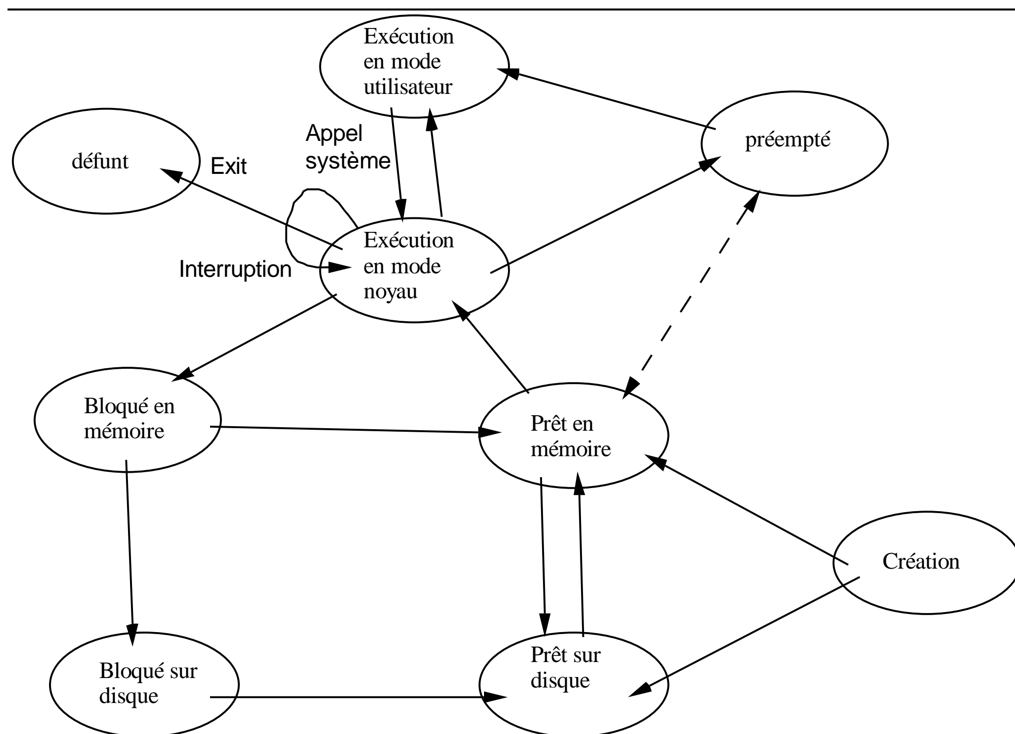


Figure 5 Les états des processus d'Unix.

1.1.2. Les processus sous Unix

Les fonctions fondamentales

Un processus peut se dupliquer – et donc ajouter un nouveau processus – par la fonction :

```
pid_t fork(void)
```

qui rend `-1` en cas d'échec. En cas de réussite, la fonction retourne `0` dans le processus fils et le n° du processus fils – *Process Identifier* ou PID – dans le père. Cette fonction transmet une partie du contexte du père au fils : les descripteurs des fichiers standards et des autres fichiers, les redirections… Les deux programmes sont sensibles aux mêmes interruptions. À l'issue d'un `fork()` les deux processus s'exécutent simultanément.

La création d'un nouveau processus ne peut se faire que par le recouvrement du code d'un processus existant grâce à la fonction :

```
int execl(char *ref, char *arg0, char *argn, 0)
```

`ref` est une chaîne de caractères donnant l'adresse du nouveau programme à substituer et à exécuter. `arg0`, `arg1`, ... , `argn` sont les arguments du programme. Le premier argument, `arg0`, reprend en fait le nom du programme.

Les fonctions `execle()` et `execlp()` ont des arguments et des effets semblables. Elle transmettent en plus respectivement la variable chemin (`PATH`) ou les variables d'environnement au complet.

Les fonctions `int execv(char *ref, char *argv[])` ainsi que `execve()` et `execvp()` sont des variantes analogues. Les paramètres du programme à lancer sont transmis dans le tableau de chaînes : `argv[][]`. Ce tableau est semblable aux paramètres de la fonction principale d'un programme C : `main(int argc, char **argv)`.

Ces fonctions rendent `-1` en cas d'échec.

Par exemple :

```
void main()
{
   execl("/bin/ls", "ls", "-l", (char *) 0);
/* avec execlp, le premier argument peut n'être
que ls et non /bin/ls */
   printf("pas d'impression\n");

}
```

La création d'un nouveau processus – l'occupation d'un nouvel emplacement de la table des processus – implique donc la duplication d'un processus existant.

Ce dernier sera le père. On substituera ensuite le code du fils par le code qu'on désire exécuter. Au départ, le processus initial lance tous les processus utiles au système.

Un processus se termine lorsqu'il n'a plus d'instructions ou lorsqu'il exécute la fonction :

```
void exit(int statut)
```

L'élimination d'un processus terminé de la table ne peut se faire que par son père, grâce à la fonction :

```
int wait(int *code_de_sortie)
```

Avec cette instruction, le père se bloque en attente de la fin d'un fils. Elle rendra le n° PID du premier fils mort trouvé. La valeur du code de sortie est reliée au paramètre d'`exit` de ce fils. On peut donc utiliser l'instruction `wait` pour connaître la valeur éventuelle de retour, fournie par `exit()`, d'un processus. Ce mode d'utilisation est analogue à celui d'une fonction. `wait()` rend `-1` en cas d'erreur.

Si le fils se termine sans que son père l'attende, le fils passe à l'état `defunct` dans la table. Si, au contraire, le père se termine avant le fils, ce dernier est rattaché au processus initial. Le processus initial passe la plupart de son temps à attendre les processus orphelins.

Grâce aux 3 instructions, `fork()`, `exec()`, et `wait()`, on peut écrire un interprète de commandes simplifié. Il prend la forme suivante :

```
while(1) {
                                   /* attend commande */
    lire_commande(commande, paramètres);
    if (fork() != 0) {
          wait(&statut);                /* proc. père */
    } else {
          execv(commande, paramètres);  /* proc. fils */
    }
}
```

Les fonctions Unix rendent `-1` en cas d'erreur. Dans ce cas, elles positionnent la variable entière globale `errno`. On peut imprimer cette valeur d'erreur et constater la cause qui est décrite dans le fichier `errno.h`. On peut aussi imprimer le texte en clair avec la fonction `perror(char*)`. Dans le cas d'une programmation professionnelle, il est indispensable de vérifier la valeur de retour de toutes les fonctions et de se protéger des erreurs par un codage du type :

```
if ((retour = fonction()) == -1) {
      perror("plantage");
      exit(0);
}
```

Les signaux

L'instruction :

```
int kill(int pid, int signal)
```

permet à un processus d'envoyer un signal à un autre processus. `pid` est le n° du processus à détruire et `signal`, le n° du signal employé. La fonction `kill` correspond à des interruptions logicielles.

Par défaut, un signal provoque la destruction du processus récepteur, à condition bien sûr, que le processus émetteur possède ce droit de destruction.

La liste des valeurs de `signal` comprend, entre autres :

Nom du signal	N° du signal	Commentaires
`SIGUP`	1	signal émis lors d'une déconnexion
`SIGINT`	2	`^C`
`SIGQUIT`	3	`^\`
`SIGKILL`	9	signal d'interruption radicale
`SIGALRM`	14	signal émis par `alarm(int sec)` au bout de sec secondes
`SIGCLD`	20	signal émis par un fils qui se termine, à son père

Tableau 2 Quelques signaux d'Unix.

Les valeurs des signaux dépendent de l'implantation. La liste complète est définies par des macros dans `signal.h`

`kill()` rend `0` en cas de succès et `-1` en cas d'échec.

Un processus peut détourner les signaux reçus et modifier son comportement par l'appel de la fonction :

```
void (*signal(int signal, void (*fonction)(int)))
```

avec `fonction` pouvant prendre les valeurs:

Nom	Action
`SIG_IGN`	le processus ignorera l'interruption correspondante
`SIG_DFL`	le processus rétablira son comportement par défaut lors de l'arrivée de l'interruption (la terminaison)
`void fonction(int n)`	le processus exécutera `fonction`, définie par l'utilisateur, à l'arrivée de l'interruption `n`. Il reprendra ensuite au point où il a été interrompu

Tableau 3 Les routines d'interruption d'Unix.

L'appel de la fonction `signal` positionne l'état des bits de signaux dans la table des processus.

Par ailleurs, la fonction `signal(SIGCLD, SIG_IGN)`, permet à un père d'ignorer le retour de ses fils sans que ces derniers encombrent la table des processus à l'état `defunct`.

Quelques fonctions d'identification d'Unix

La commande Unix `ps -ef`, donne la liste des processus en activité sur le système. Chaque processus possède un identificateur et un groupe. On les obtient par les fonctions :

```
int getpid(void)
```

et

```
int getpgrp(void)
```

Les fils héritent du groupe de processus du père. On peut changer ce groupe par la fonction :

```
int setpgrp(void)
```

Chaque processus possède, d'autre part, un utilisateur réel et effectif. On les obtient respectivement par les fonctions :

```
int getuid(void)
```

et

```
int geteuid(void)
```

L'utilisateur réel est l'utilisateur ayant lancé le processus. Le numéro d'utilisateur effectif sera utilisé pour vérifier certains droits d'accès (fichiers et envoi de signaux). Il correspond normalement au numéro d'utilisateur réel. On peut cependant positionner l'utilisateur effectif d'un processus au propriétaire du

fichier exécutable. Ce fichier s'exécutera alors avec les droits du propriétaire et non avec les droits de celui qui l'a lancé. La fonction :

```
int setuid(int euid)
```

permet de commuter ces numéros d'utilisateur de l'un à l'autre : réel à effectif et vice-versa. Elle rend `0` en cas de succès.

1.2. Les fils d'exécution

Les interfaces graphiques, les systèmes multi-processeurs ainsi que les systèmes répartis ont donné lieu à une révision de la conception usuelle des processus. Cette révision se fonde sur la notion de fils d'exécution (*thread of control*). Un processus classique est composé d'un espace d'adressage et d'un seul fil de commande. L'idée est d'associer plusieurs fils d'exécution à un espace d'adressage et à un processus. Les fils d'exécution sont donc des processus dégradés (sans espace d'adressage propre) à l'intérieur d'un processus ou d'une application. On les appelle aussi des processus poids plume ou poids léger.

1.2.1. Pourquoi des fils d'exécution

Certaines applications se dupliquent entièrement au cours du traitement. C'est notamment le cas pour des systèmes client-serveur, où le serveur exécute un `fork`, pour traiter chacun de ses clients. Cette duplication est souvent très coûteuse. Avec des fils d'exécution, on peut arriver au même résultat sans gaspillage d'espace, en ne créant qu'un fil de contrôle pour un nouveau client, et en conservant le même espace d'adressage, de code et de données. Les fils d'exécution sont, par ailleurs, très bien adaptés au parallélisme. Ils peuvent s'exécuter simultanément sur des machines multi-processeurs.

1.2.2. De quoi sont constitués les fils d'exécution

Éléments d'un fil de commande	Éléments d'un processus ordinaire
Compteur de programme	Espace d'adressage
Pile	Variables globales
Registres	Table des fichiers ouverts
Fils de commande fils	Processus fils
Statut	Compteurs
	Sémaphores

Tableau 4 Le contexte comparé d'un fil de commande et d'un processus.

Les éléments d'un processus sont communs à tous les fils d'exécution de ce processus.

1.2.3. Utilisation des fils d'exécution

L'avantage essentiel des fils d'exécution est de permettre le parallélisme de traitement en même temps que les appels système bloquants. Ceci impose une coordination particulière des fils d'exécution d'un processus. Dans une utilisation optimale, chaque fil de commande dispose d'un processeur. A. Tanenbaum, dans *Modern Operating Systems*, distingue trois modèles d'organisation possibles : le modèle répartiteur/ouvrier (*dispatcher/worker*), l'équipe et le pipeline.

Le modèle répartiteur/ouvrier

Dans ce modèle, les demandes de requêtes à un serveur arrivent dans une boîtes aux lettres. Le fil de commande répartiteur a pour fonction de lire ces requêtes et de réveiller un fil de commande ouvrier. Le fil de commande ouvrier gère la requête et partage avec les autres fils d'exécution, la mémoire cache du serveur. Cette architecture permet un gain important car elle évite au serveur de se bloquer lors de chaque entrée-sortie.

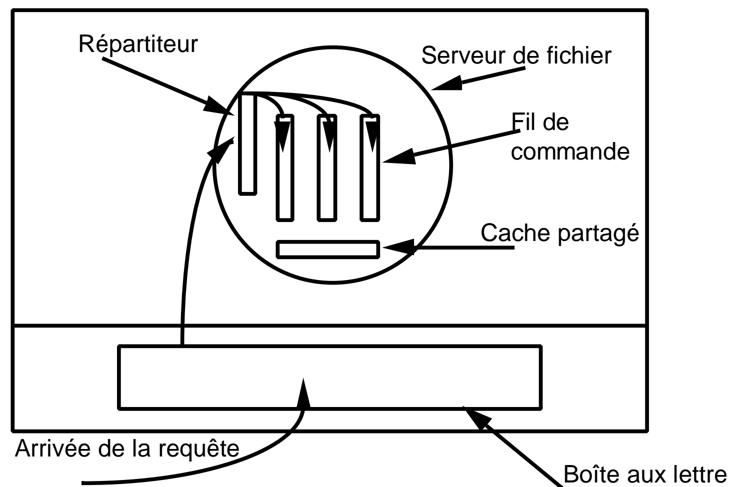


Figure 6 Le répartiteur

L'équipe

Dans le modèle de l'équipe, chaque fil de commande traite une requête. Il n'y a pas de répartiteur. À la place, on met en implante une file de travaux en attente.

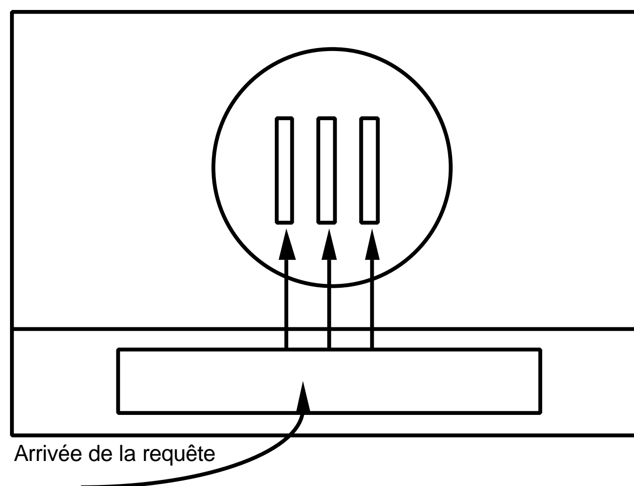


Figure 7 L'équipe

Le pipeline

Dans le modèle du pipeline, chaque fil de commande traite une partie de la requête. Plusieurs architecture matérielles implantent ce type d'organisation. Ce modèle n'est cependant pas approprié pour toutes les applications.

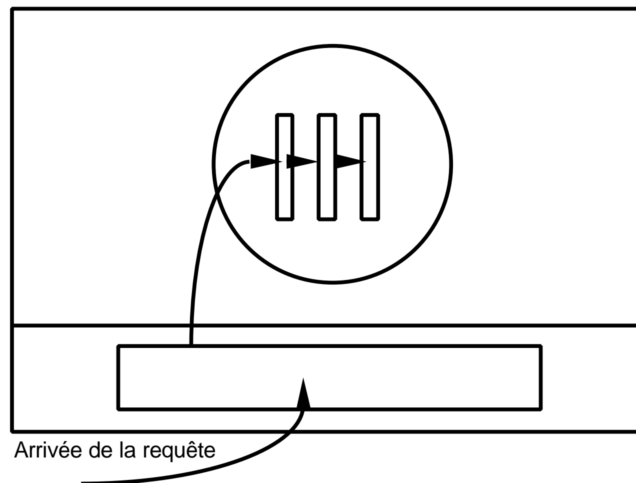


Figure 8 Le *pipeline*

1.2.4 L'implantation des fils d'exécution avec Java

Le langage Java permet de construire des applications et des appliquettes. Les appliquettes sont des petits programmes chargeables dans un navigateur de réseau. Java, grâce à ses classes de base, fournit la possibilité de créer des fils d'exécution au niveau du langage de programmation.

Créer un fil de commande

Créer un fils d'exécution peut se faire en créant une sous classe de la classe `Thread` et en créant ensuite un objet appartenant à cette sous classe[5]. L'objet

[5] On peut aussi donner à une classe Java des capacités multi-tâches en la faisant dériver par `implements` de l'interface `Runnable`. Nous n'examinerons pas ces possibilités ici.

héritera des méthodes (des fonctions membres) de la classe et elle permettront sa gestions.

Les méthodes de la classe `Thread` sont notamment :

- `start()` qui démarre le fil de commande;
- `stop()` qui l'arrête définitivement ;
- `run()` le corps du fil qui est exécuté après le lancement par `start()` ;
- `sleep()` qui le met en veille pour un temps paramétrable ;
- `suspend()` qui le suspend – qui le met en attente – jusqu'à nouvel ordre ;
- `resume()` qui reprend l'exécution du fil de commande;
- `yield()` qui laisse passer son tour d'exécution.

Le modèle d'exécution des fils de Java peut se représenter par un automate d'états.

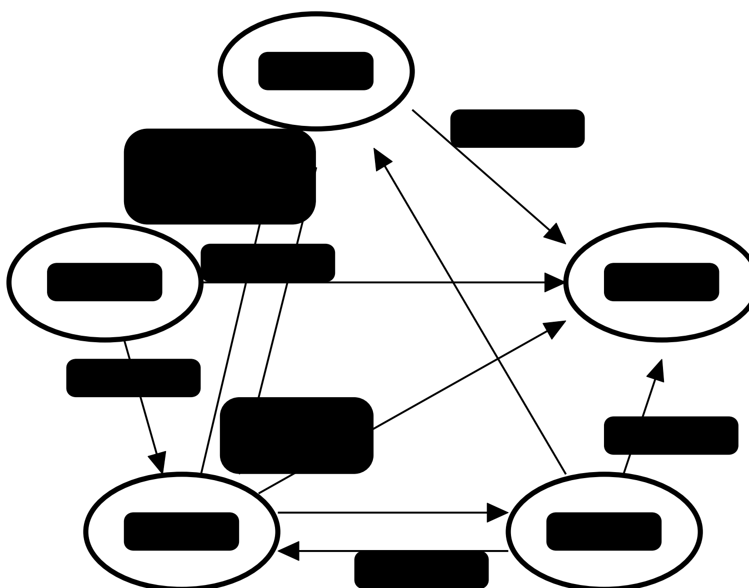


Figure 9 Les états des fils d'exécution de Java.

Un exemple

L'exemple qui suit crée et lance deux fils d'exécution `Afficheur`. Chaque fil imprime son nom à l'écran et son nombre de passage pendant qu'il s'exécute. Une fois compilé, on doit exécuter le programme avec l'interprète `java` :

```
> java Lanceur
```

```
class Lanceur {
    public static void main (String args[]) {
          Afficheur afficheur1 = new Afficheur("Moi");
          Afficheur afficheur2 =
                          new Afficheur("LAutre");
          afficheur1.start();
          afficheur2.start();
    }
}

class Afficheur extends Thread {
    public Afficheur(String str) {
          super(str);
    }
    public void run() {
          for (int i = 0; i < 10; i++) {
                System.out.println(getName()+ "  " + i + "
passage(s)");
                try {
                      sleep((int)(Math.random()*1000));
                } catch (InterruptedException e) {}
          }
          System.out.println("Terminé" + getName());
          this.stop();
    }
}
```

1.2.5. L'implantation des fils d'exécution sous Windows NT

Les fils d'exécution peuvent être implantés dans l'espace utilisateur ou dans le noyau. Actuellement, peu de systèmes d'exploitation incluent des fils d'exécution dans leur noyau. Plusieurs bibliothèques externes existent, par exemple les processus poids légers de Sun. Ce n'est pas le cas de Windows NT qui a été conçu d'emblée avec des fils d'exécution.

Windows NT peut se programmer à deux niveaux : en C ou bien en C++. La programmation en C se fait par les API du *Software Development Kit*. La programmation en C++ se fait par l'intermédiaire des *Microsoft Foundation Classes* qui encapsule les entités du système.

Un processus sous Windows est une instance d'application qui possède un espace d'adressage, de la mémoire, des fichiers et des fenêtres ouvertes. On crée un nouveau processus par la fonction `CreateProcess()`. Ce processus comprendra un fil de commande. On peut lui en ajouter d'autres avec le fonction :

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpsa, DWORD
cbStack, LPTHREAD_START_ROUTINES, lpSStartAddr,
LPVOID lpThreadParm, DWORD fdwCreate, LPDWORD
lpIDThread).
```

Un fil de commande peut s'assurer l'exclusivité d'une ressource par la fonction `WaitForSingleObject()` qui correspond à un P de sémaphore d'exclusion mutuelle. On libère le sémaphore par un `ReleaseMutex()`. Les fonctions `InterLockedIncrement()` et `InterLockedDecrement()` sont applicables à des sémaphores généralisés.

1.3. Communication entre les processus

Les processus peuvent communiquer par l'intermédiaire de fichiers. Néanmoins, certains systèmes d'exploitation fournissent des mécanismes permettant les communications directes.

Le signe « | », par exemple, de l'interprète de commandes Unix, relié à l'appel système `pipe()`, permet à deux processus de s'exécuter en même temps. Le premier fournissant des données que le second exploite au fur et à mesure de leur production.

1.3.1. Les tubes de communication avec Unix

Le système Unix offre des primitives qui permettent de synchroniser facilement des processus lecteurs et écrivains dans un tampon sans passer par des sémaphores. L'appel de la fonction du noyau `pipe()` crée un tampon de données, un tube, dans lequel deux processus pourront venir respectivement lire et écrire. La fonction fournit, d'autre part, deux descripteurs, analogues aux descripteurs de fichiers, qui serviront de référence pour les opérations de lecture et d'écriture. Le système réalise la synchronisation de ces opérations de manière interne.

Les deux processus communicants doivent partager les mêmes descripteurs obtenus par `pipe()`. Il est donc commode qu'ils aient un ancêtre commun car les descripteurs de fichiers sont hérités. Dans l'exemple suivant, deux processus, un père et un fils communiquent par l'intermédiaire d'un tube. Le père écrit les lettres de l'alphabet que lit le fils :

```
void main () {
    int df[2];          // descripteur du tube
    char c, d;          // caractère de l'alphabet
    int ret;
    pipe(df);           // On crée le tube
    if (fork() != 0) {        // On est dans le père
        close (df[0]);    // Le père ne lit pas
        for (c = 'a'; c <= 'z'; c++) {
            write(df[1], &c, 1); //Père écrit dans tube
        }
        close(df[1]); // On ferme le tube en écriture
        wait(&ret);       // Le père attend le fils
    } else {
        close(df[1]); // On ferme le tube en écriture
        while (read(df[0], &d, 1) > 0) {// Fils lit tube
            printf("%c\n", d);// Il écrit le résultat
        }
        close (df[0]);    // On ferme le tube en lecture
    }
}
```

La déclaration `int file[2]`, et l'appel de la fonction `int pipe(file)` crée un tampon interne et deux descripteurs de fichiers. L'un ouvert en lecture : `file[0]` et l'autre en écriture : `file[1]`. La fonction `pipe()` rend `0` ou –`1` en cas d'échec.

La fonction `int open(char *ref, int indicateurs)` ouvre un fichier dont le nom est contenu dans `ref` et rend un descripteur de fichier. En cas d'échec, elle rend -1. Les indicateurs (flags) sont en lecture `O_RDONLY` (0), en écriture `O_WRONLY`, (1), en lecture-écriture `O_RDWR` (2). Il existe d'autres indicateurs, par exemple, l'ajout `O_APPEND`; ils sont décrits dans le fichier `fcntl.h`. On combine les modes par un ou binaire : `O_WRONLY | O_APPEND` déclare une ouverture en écriture et ajout.

Les descripteurs de fichiers remarquables sont 0 pour l'entrée standard, le clavier, 1 pour la sortie standard, l'écran, et 2 pour la sortie d'erreur standard, l'écran encore.

`int close(int desc)` ferme le fichier de descripteur `desc`.

`int read(int desc, char *buf, int n)`, permet de lire dans le fichier de descripteur `desc`, `n` caractères qui seront contenus dans le tampon pointé par `buf`.

`int write(int desc, char *buf, int n)` permet d'écrire dans le fichier de descripteur `desc`, `n` caractères, contenus dans le tampon pointé par `buf`.

Lorsque l'interprète de commande reconnaît une commande du type : « `processus1 | processus2` », il exécute un programme de la forme :

```
void synchro(char *processus1, char *processus2) {
    int df[2];        /* descripteurs du tube */
    pipe(df);               /* création du tube */
    if (fork() != 0) {
          close(df[0]);     /* fermeture lecture */
          dup2(df[1], 1); /* df[1] devient sortie standard */
          close(df[1]);     /* inutile après redirection */
          execl(processus1, processus1, 0);
    } else {
          close(df[1]);     /* fermeture écriture */
          dup2(df[0], 0); /* df[0] devient entrée standard */
          close(df[0]);     /* inutile après redirection */
          execl(processus2, processus2, 0);

    }
}
```

La fonction `int dup2(int desc1, int desc2)` duplique le descripteur de fichier `desc1` et le rend équivalent à `desc2`. Si `desc2` est en cours d'utilisation, il est au préalable fermé par la fonction `dup2()`. La fonction `int dup(int desc)` est semblable. Elle affecte au plus petit descripteur disponible le fichier référencé par `desc`.

Pour utiliser les fonctions Unix de base, il faut en général inclure `fcntl.h`, `unistd.h` et `stdlib.h`.

1.3.2 Les tubes de communication avec Java

Avec Java, on dispose d'un mécanisme semblable à Unix. Un tube en Java se compose de deux demi-tubes dérivés des classes `PipedInputStream` et `PipedOutputStream` que l'on connecte par la fonction membre `connect()`.

Dans l'exemple qui suit, deux fils communiquent par un tube. Un fil d'exécution lit des caractères du clavier, les transmets dans le tube et un autre fil les écrit sur l'écran en les passant en majuscules. Ils font chacun dix tours de boucles.

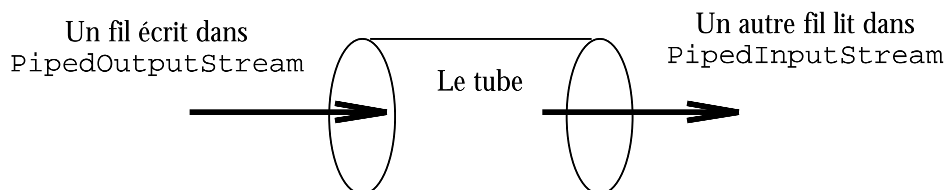


Figure 10 Les tubes en Java.

```
import java.io.*;

class ExThread {
    public static void main (String args[]) {
        PipedInputStream sortieLecture = new PipedInputStream();
        PipedOutputStream entreeEcriture = new PipedOutputStream();
        try {
            sortieLecture.connect(entreeEcriture);
        } catch (IOException EOut) {}
        new Ecrivain("Ecrivain", entreeEcriture).start();
        new Lecteur("Lecteur", sortieLecture).start();
    }
}

class Ecrivain extends Thread {
```

```
        private PipedOutputStream entreeEcriture;

        public Ecrivain(String str, PipedOutputStream entreeEcriture) {
                super(str);
                this.entreeEcriture = entreeEcriture;
        }
        public void run() {
                byte caractere[] = new byte [80];
                for (int i = 0; i < 10; i++) {
                        try {
                                System.in.read(caractere);
                        } catch (IOException eIn) {}
                        try {
                                entreeEcriture.write(caractere[0]);
                        } catch (IOException eOut) {}
                        System.out.print(getName() + " a écrit dans le tube\t" +
caractere[0]);
                        System.out.println("\t(" + i + " passage(s))");
                }
                System.out.println(getName() + " Terminé");
                this.stop();
        }

   }
   class Lecteur extends Thread {
        private PipedInputStream sortieLecture;
        public Lecteur(String str, PipedInputStream sortieLecture) {
                super(str);
                this.sortieLecture = sortieLecture;
        }
        public void run() {
                char car[] = new char[80];

                for (int i = 0; i < 10; i++) {
                        try {
                                car[0] = (char) sortieLecture.read();
                        }catch (IOException IOut) {}
                        System.out.print(getName()  +  "  a  lu  dans  le  tube\t"  +
Character.toUpperCase(car[0]));
                        System.out.println( "\t(" + i + " passage(s))");
                }
                System.out.println(getName() + " Terminé");
                this.stop();
        }

   }
```

1.3.3. Les messages

Les messages forment un mode de communication privilégié entre les processus. Ils sont utilisés dans le système pédagogique Minix de Tanenbaum. Ils sont au cœur de Mach qu'on présente comme un successeur possible d'Unix et qui a inspiré Windows NT pour certaines parties. Par ailleurs, ils s'adaptent très bien à une architecture répartie et leur mode de fonctionnement est voisin des échanges de données sur un réseau.

Les communications de messages se font à travers deux opérations fondamentales : `envoie(message)` et `reçois(message)`. (`send` et `receive`). Les messages sont de tailles variables ou fixes. Les opérations

d'envoi et de réception peuvent être soit directes entre les processus, soit indirectes par l'intermédiaire d'une boîte aux lettres.

Les communications directes doivent identifier le processus, par un n° et une machine, par exemple. On aura alors : `envoie(Proc, Message)` et `reçois(Proc, Message)`. Dans ce cas la communication est établie entre deux processus uniquement, par un lien relativement rigide et bidirectionnel. On peut rendre les liaisons plus souples en laissant vide l'identité du processus dans la fonction `reçois`.

Les communications peuvent être indirectes grâce à l'utilisation d'une boîte aux lettres (un « port » dans la terminologie des réseaux). Les liens peuvent alors unir plus de deux processus du moment qu'ils partagent la même boîte aux lettres. On devra néanmoins résoudre un certains nombre de problèmes qui peuvent se poser, par exemple, si deux processus essayent de recevoir simultanément le contenu d'une même boîte.

Les communications se font de manière synchrone ou asynchrone. Le synchronisme peut se représenter par la capacité d'un tampon de réception. Si le tampon n'a pas de capacité, l'émetteur doit attendre que le récepteur lise le message pour pouvoir continuer. Les deux processus se synchronisent sur ce transfert et on parle alors d'un « rendez-vous ». Deux processus asynchrones : P et Q, peuvent aussi communiquer de cette manière en mettant en œuvre un mécanisme d'acquittement :

P	Q
`envoie(Q, message)`	`reçois(P, message)`
`reçois(Q, message)`	`envoie(P, acquittement)`

Tableau 5 Les fonctions de messages.

1.3.4. La mémoire partagée

On peut concevoir des applications qui communiquent à travers un segment de mémoire partagée. Le principe est le même que pour un échange d'informations entre deux processus par un fichier. Dans le cas d'une zone de mémoire partagée, on devra déclarer une zone commune par une fonction spécifique, car la zone mémoire d'un processus est protégée.

Le système Unix fournit les primitives permettant de partager la mémoire. NT aussi sous le nom de fichiers mappés en mémoire. Ces mécanismes, bien que très rapides, présentent l'inconvénient d'être difficilement adaptables aux réseaux. Pour les communications locales, la vitesse est sans doute semblable à celle de la communication par un fichier à cause de la mémoire cache. Lorsqu'il a besoin de partager un espace mémoire, le programmeur préférera utiliser des fils d'exécution.

1.4. Coordination les processus

La coordination (ou exclusion) a pour but de limiter l'accès à une ressource à un ou plusieurs processus. Ceci concerne, par exemple, un fichier de données que plusieurs processus désirent mettre à jour. L'accès à ce fichier doit être réservé à un utilisateur pendant le moment où il le modifie, autrement son contenu risque de ne plus être cohérent. Le problème est semblable pour une imprimante où un utilisateur doit se réserver son usage le temps d'une impression. On appelle ce domaine d'exclusivité, une section critique.

1.4.1. L'attente active sur un verrou

La façon la plus ancienne de réaliser l'exclusion mutuelle sous Unix est d'effectuer un verrouillage sur une variable partagée. Cette opération de verrouillage doit être indivisible (atomique). Lorsque le verrou est posé sur la variable, les processus exécutent une attente active. Lorsqu'il se lève, un processus, et un seul, pose un nouveau verrou sur la variable et rentre dans la

section critique. Il sait qu'il aura l'exclusivité d'une ressource, un fichier par exemple :

Processus 1	Processus 2

`while (pos_ver(var) == échec)`
`    rien;`
`section_critique();`
`lève_ver(var);`
	`while (pos_ver(var) == échec)`
	`    rien;`
	`section_critique();`
	`lève_ver(var);`

Tableau 6 Attente active sur un verrou.

Dans les vieilles versions d'Unix, la variable de verrouillage correspondait souvent à un fichier sans droits que chacun des processus tente de créer. Lorsqu'un processus termine sa section critique, il détruit le fichier de verrouillage, laissant ainsi sa place à d'autres. On créera le verrou dans /tmp[6], par exemple, de manière à ce que tout le monde puisse le détruire. Enfin, pour que ce procédé fonctionne, il est nécessaire que tous les processus coopèrent.

```
void main()
{
    int desc;         /* descripteur fichier de ressource */
    int desc_verr;    /* descripteur fichier verrou */
    char i;
                      /* attente active */
    while ((desc_verr = creat("/tmp/verrou", 0000)) == -1)
          ;
    section_critique();
    close(desc_verr);       /* fermeture du verrou */
    unlink("/tmp/verrou");  /* destruction du verrou */

}
```

[6] Car /tmp est un répertoire dont les droits sont : rwx rwx rwx

`int creat(char *ref_fichier, int mode)` rend le descripteur du fichier ouvert en écriture, ou -1 en cas d'échec. `mode` définit les droits d'accès, `0777` p. ex. Le mode réel est obtenu par un ET binaire avec le paramètre `cmask` positionné par la fonction :

```
int umask(int cmask)
```

`int close(int desc_fichier)` ferme le fichier considéré. Cette fonction rend 0 en cas de succès et –1 en cas d'échec.

`int unlink(char *ref_fichier)` détruit le fichier dont le nom est passé en paramètre. Cette fonction rend 0 en cas de succès et –1 en cas d'échec.

Au niveau du processeur, les variables de verrouillage se réalisent par une instruction de type *Test and Set Lock* (TSL).

On n'utilise plus désormais cette façon de procéder. On préfère la fonction `flock(int fd, int opération)` qui a les mêmes fonctionnalités. Elle permet à un processus de verrouiller l'accès à un fichier de descripteur `fd` avec un verrou exclusif. Dans ce cas, `opération` est égal à `LOCK_EX`. `flock()` est bloquante si le fichier est déjà verrouillé. Le processus qui exécute cette fonction attendra s'il y a déjà un processus ou posera le verrou et passera si la voie est libre. On retire le verrou avec la même fonction. L'opération est `LOCK_UN`. Un processus bloqué passera donc lorsque le premier processus aura retiré son verrou. Le fichier d'en-tête à inclure est `<sys/file.h>`.

1.4.2. Les sémaphores

Généralités

Le concept de sémaphore permet une solution élégante à la plupart des problèmes d'exclusion. Ce concept nécessite la mise en œuvre d'une variable, le sémaphore, et de deux opérations atomiques associées `P` et `V`. Soit `séma` la variable, elle caractérise les ressources et permet de les gérer. Lorsqu'on désire

effectuer un exclusion mutuelle entre tous les processus par exemple, il n'y a virtuellement qu'une seule ressource et on donnera à `séma` la valeur initiale de 1.

Lorsqu'un processus effectue l'opération `P(séma)` :

- si la valeur de `séma` est supérieure à 0, il y a alors des ressources disponibles, `P(séma)` décrémente `séma` et le processus poursuit son exécution,
- sinon ce processus sera mis dans une file d'attente jusqu'à la libération d'une ressource.

Lorsqu'un processus effectue l'opération `V(séma)` :

- si il n'y a pas de processus dans la file d'attente, `V(séma)` incrémente la valeur de `séma`,
- sinon un processus en attente est débloqué.

`P(séma)` correspond donc à une prise de ressource et `V(séma)` à une libération de ressource. Dans la littérature, on trouve parfois d'autres terminologies, respectivement, `wait(séma)` et `signal(séma)`[7], ou `get(séma)` et `release(séma)`.

Le problème des producteurs et des consommateurs

Ce problème est un des cas d'école favoris sur les processus. Deux processus se partagent un tampon de données de taille `N`. Un premier processus produit des données et les écrit dans le tampon. Un second processus consomme les données du tampon en les lisant et en les détruisant au fur et à mesure de leur lecture. Initialement, le tampon est vide. La synchronisation de ces deux processus peut se réaliser grâce à des sémaphores.

---

[7] Le sens associé à ces dénominations, bien que très utilisées, n'apparaît pas très clairement.

Les processus producteur et consommateur doivent accéder de manière exclusive au tampon, le temps d'une lecture ou d'une écriture. Un sémaphore d'exclusion mutuelle est donc nécessaire. D'autre part, on peut considérer que les ressources du processus producteur sont les emplacements vides du tampon, alors que les emplacements pleins sont les ressources du processus consommateur. Au départ, le tampon étant vide, les ressources de consommation sont nulles, alors que les ressources de production correspondent à la taille du tampon.

On obtient le programme suivant :

```
#define N 100            /* taille du tampon */

séma mutex = 1;          /* séma d'exclusion mut. */
séma production = N;     /* séma places vides */
séma consommation = 0;   /* séma places pleines */

void consommateur() {
     while (1) {
          P(consommation);     /* une place pleine en moins */
          P(mutex);                  /* section critique */
          vider_case();
          V(mutex);                  /* fin section critique */
          V(production);       /* une place vide en plus */
     }
}

void producteur() {
     while (1) {
          P(production);       /* une place vide en moins */
          P(mutex);                  /* section critique */
          remplir_case();
          V(mutex);                  /* fin section critique */
          V(consommation);     /* une place pleine en plus */
     }
}
```

Les sémaphores sous Unix

Le système Unix, dans la plupart de ses versions, fournit maintenant des sémaphores. Les sémaphores ne correspondent cependant pas à une programmation classique de ce système. On leur préfère les verrous ou bien les tubes quand c'est possible. Par ailleurs, les paramètres des fonctions sémaphores peuvent être légèrement différents suivant les versions d'Unix. En cas de doute, consultez le manuel.

À chacune des étapes de la manipulation d'un sémaphore :

1. Déclaration : `Type Sémaphore séma;`
2. Initialisation : `séma <- 1;`
3. Opération : `P(séma);` ou bien `V(séma);`

correspond un appel système Unix.

On doit inclure les fichiers d'en-tête :

`<sys/types.h>`, `<sys/ipc.h>` et `<sys/sem.h>`.

Les sémaphores se regroupent en tableaux de type `ushort`. Par exemple : `ushort semarray[2]` est un tableau de deux sémaphores.

Le déclaration d'une variable « séma » se fait par la fonction :

```
int semget(key_t clé, int NombreSéma, int flag);
```

Cette fonction rend un entier, qui servira de référence ultérieure au sémaphore dans le programme. Elle permet l'identification universelle d'un sémaphore (à l'échelle du système). On l'utilise généralement de la façon suivante :

```
idSema = semget(ftok(UnfFichier, 'UneLettre'),
NombreSéma, IPC_CREAT | 0600);
```

L'affectation d'une variable sémaphore se fait par l'affectation du tableau. On peut ainsi initialiser plusieurs sémaphores – autant que la dimension du tableau. Par exemple :

```
semarray[0] = 1;
semarray[1] = 2;
```

puis par sa prise en compte par le système par une opération atomique :

```
  int semctl(int idSema, int NombreSéma, int
commande, ushort semarray);
```

La valeur de `commande` pour affecter les valeurs du tableau aux sémaphores est `SETALL`. Dans certaines versions d'Unix, la dernière variable `semarray` de type `ushort` est parfois remplacée par une structure `union semnum` qui doit contenir ce tableau.

`semctl()` permet d'autres opérations telles que la lecture de la valeur de sémaphores avec la commande `GETALL` et la destruction avec la commande `IPC_RMID`.

Les opérations P et V peuvent être groupées (on peut en effectuer plusieurs en même temps). Elles se font par la construction d'un tableau d'opérations de type `struct sembuf`. Avec la déclaration : `struct sembuf sops[2];` par exemple, il pourra y avoir jusqu'à deux opérations P ou V simultanées.

Les champs à affecter de la structure sont :

- `sem_num`, qui donne le rang du sémaphore dans le tableau précédent,
- `sem_op`, mis à -1 si c'est un P et à 1 si c'est un V.
- `sem_flg`, en général à 0.

La prise en compte des opérations se fait par :

```
int semop(int idSema, struct sembuf *sops, size_t NbOpérat)
```

On peut visualiser l'état des sémaphores par la commande `ipcs` ou `ipcs -a` et on peut détruire un sémaphore par la commande `ipcrm -s no_du_sémaphore`.

1.4.3 Les moniteurs

Qu'est-ce qu'un moniteur?

Un moniteur est un objet encapsulant des fonctions membre – ou des méthodes – dont une seule peut s'exécuter à un moment donné. La structure de moniteur permet d'éviter certains phénomènes d'interblocages qui peuvent apparaître avec les sémaphores. Ces interblocages sont généralement la conséquence d'erreurs de programmation vicieuses et difficiles à détecter. C'est pourquoi, on considère que l'écriture de sémaphores est délicate. La déclaration d'un moniteur est beaucoup plus facile et plus élégante par contre.

```
monitor MonMoniteur{
   char tampon[100];
   void écrivain(char *chaîne) {}
   char *lecteur() {}
}
```

Dans l'exemple précédent, en pseudo-C, quel que soit le nombre de processus qui s'exécutent et qui font appel au moniteur, seul un exemplaire de `écrivain` ou de `lecteur` pourra s'exécuter à un moment donné.

Dans le modèle de Hoare, les moniteurs disposent en plus d'un mécanisme qui permet à un processus de s'arrêter sur une condition particulière, de laisser sa place et de reprendre son exécution quand la condition est remplie. On arrête un

processus par `wait(condition)` et on le relance par `notify(condition)`.

```
monitor MonMoniteur{
    char tampon[100];
    void écrivain(char *chaîne) {
        ...
        if (condition == false) wait(condition);
        ...
    }
    char *lecteur() {
        ...
        notify(condition);
        ...
    }
}
```

Un processus exécutant `écrivain()` avec `condition` à faux, s'arrêtera et laissera sa place, permettra à un autre processus d'exécuter une fonction du moniteur et attendra jusqu'à ce qu'un processus exécutant `lecteur()` le relance par `notify()`.

Synchroniser les fils d'exécution

Les fils d'exécution s'exécutent dans le même espace mémoire. Ils peuvent tous accéder aux variables communes d'une classe simultanément. Comme pour les processus, ça peut conduire à des effet indésirables. Le langage Java ne dispose pas de sémaphores qu'il remplace par des moniteurs.

En Java, tout héritier de la classe `Object` est un moniteur potentiel – et donc toute classe, car `Object` est la classe mère –. Pour réaliser une exclusion et assurer un accès unique à une méthode, on utilise le mot clé

synchronized. Le compilateur associe un « moniteur » à chaque objet qui possède des variables ou des méthodes précédées de synchronized. Il supervisera un accès unique soit au niveau d'un objet de la classe, soit au niveau de la classe elle-même si elle est statique (static). Ainsi un seul fil de commande pourra exécuter une même méthode synchronisée à un moment donné. Et une seule des méthodes synchronisées d'un objet pourra s'exécuter. On peut aussi déclarer un bloc {... } comme synchronized sur un objet, par exemple

```
synchronized (monObjet) {

// les intructions

}
```

Lorsqu'une méthode s'exécute sans avoir de données disponibles, elle peut bloquer les autres ou abandonner l'exécution sans avoir rien fait. Pour éviter ceci, on dispose de la fonction wait(), mais sans variable. Une fonction qui exécute ce wait()va alors s'arrêter, libérer le moniteur et attendre. Une autre méthode pourra prendre sa place.

Une fonction qui a exécuté un wait() pourrait rester bloquée à l'infini. C'est pourquoi, on dispose de la fonction notify() qui permet de relancer une méthode bloquée sur un wait() (n'importe laquelle). Une fonction qui exécute un notify() relancera une et une seule des fonctions en attente. notifyAll() permet de relancer toutes les méthodes bloquées par un wait().

Un exemple

Grâce aux moniteurs, on peut facilement implanter le programme de producteurs et de consommateurs. Pour ceci, les fils d'exécution les représentant doivent partager un tampon commun où les fils producteurs viendront écrire des messages et les fils consommateurs viendront les lire.

On peut réaliser ceci par une classe `Tampon` qui dispose de fonctions `put()` et `get()` synchronisées qui respectivement écrivent et lisent les messages :

- `put()` écrit dans le tampon. S'il y a quelque chose, il attend qu'il se vide.
- `get()` lit la valeur du tampon et le met à zéro. S'il n'y a rien, il attend qu'il se remplisse.

```
class Tampon {
    private String data = null;
    public synchronized boolean put(String put_data) {
    // Si quelqu'un a écrit quelque chose
    // on attend
    while (data != null) {
          try {
                wait();
          } catch (InterruptedException e) {}
    }
    data = put_data;
    // Les autres peuvent se réveiller
    notifyAll();
    return true;
    }
    public synchronized String get() {
          while (data == null) {
                try {
                      wait();
                } catch (InterruptedException e) { }
          }
          String get_data = data;
          data = null;
          notifyAll();
          return get_data;
    }
}
```

1.5. Ordonnancement

L'ordonnancement règle les transitions d'un état à un autre des différents processus. Cet ordonnancement a pour objectifs :

1. de maximiser l'utilisation du processeur;
2. d'être équitable entre les différents processus;
3. de présenter un temps de réponse acceptable;
4. d'avoir un bon rendement;
5. d'assurer certaines priorités;

1.5.1. Le tourniquet

Cet algorithme est l'un des plus utilisés et l'un des plus fiables. Chaque processus prêt dispose d'un quantum de temps pendant lequel il s'exécute. Lorsqu'il a épuisé ce temps ou qu'il se bloque, par exemple sur une entrée-sortie, le processus suivant de la file d'attente est élu et le remplace. Le processus suspendu est mis en queue du tourniquet.

Le seul paramètre important à régler, pour le tourniquet, est la durée du quantum. Il doit minimiser le temps de gestion du système et cependant être acceptable pour les utilisateurs. La part de gestion du système correspond au rapport de la durée de commutation sur la durée du quantum. Plus le quantum est long plus cette part est faible, mais plus les utilisateurs attendent longtemps leur tour. Un compromis peut se situer, suivant les machines, de 100 à 200 ms.

1.5.2. Les priorités

Dans l'algorithme du tourniquet, les quanta égaux rendent les différents processus égaux. Il est parfois nécessaire de privilégier certains processus par rapport à d'autres. L'algorithme de priorité choisit le processus prêt de plus haute priorité.

Ces priorités peuvent être statiques ou dynamiques. Les processus du système auront des priorités statiques (non-modifiables) fortes. Les processus des utilisateurs verront leurs priorités modifiées, au cours de leur exécution, par l'ordonnanceur. Ainsi un processus qui vient de s'exécuter verra sa priorité baisser. Pour un exemple d'exécution avec priorités, on pourra consulter Bach[8].

1.5.3. Le tourniquet avec priorités

On utilise généralement une combinaison des deux techniques précédentes. À chaque niveau de priorité correspond un tourniquet. L'ordonnanceur choisit le tourniquet non vide de priorité la plus forte et l'exécute.

Pour que tous les processus puissent s'exécuter, il est nécessaire d'ajuster périodiquement les différentes priorités.

1.5.4 L'ordonnancement des fils d'exécution

Les fils d'exécution sont soumis à un ordonnancement. Dans Windows NT, les fils d'exécution ont 32 niveaux de priorité qui sont soit fixes soit dynamiques. La priorité la plus haute est toujours celle qui s'exécute. Dans le cas d'une priorité dynamique, les valeurs de cette priorité varient entre deux bornes. Elle augmente, par exemple, lors d'une attente d'entrée-sortie. On peut changer la priorité des fils d'exécution dans Windows NT par la fonction `SetThreadPriority()`.

Avec Java, les niveaux de priorité varient entre `Thread.MIN_PRIORITY` et `Thread.MAX_PRIORITY`. La priorité normale est `Thread.NORM_PRIORITY`. Comme avec NT, la priorité la plus haute est toujours celle qui s'exécute. Certains interprètes utilisent un tourniquet qui gère les fils de même priorité sans limite de temps pour le fil qui s'exécute. D'autres interprètes limitent l'exécution d'un fils à un quantum de temps, puis passent à un

---

[8] op. cit., p. 267-270.

autre fil d'une même priorité. Le modèle d'exécution peut être assez différent suivant les machines, par exemple entre l'IBM 580 et le Sun E3000.

Chapitre 2
La mémoire

2.1. Introduction

La mémoire principale est le lieu où se trouvent les programmes et les données quand le processeur les exécute. On l'oppose au concept de mémoire secondaire, représentée par les disques, de plus grande capacité, où les processus peuvent séjourner avant d'être exécutés.

De manière encore plus vive que pour les autres ressources informatiques, le prix des mémoires a baissé et la capacité unitaire des circuits a augmenté. Cependant la nécessité de la gérer de manière optimale est toujours fondamentale, car en dépit de sa grande disponibilité, elle n'est, en général, jamais suffisante. Ceci en raison de la taille continuellement grandissante des programmes.

2.1.1. La multiprogrammation

Le concept de multiprogrammation s'oppose à celui de monoprogrammation. La monoprogrammation ne permet qu'à un seul processus utilisateur d'être exécuté. Cette technique n'est plus utilisée que dans les micro-ordinateurs. On trouve alors en mémoire, par exemple dans le cas de MS-DOS : le système en mémoire basse, les pilotes de périphériques en mémoire haute (dans une zone allant de 640 ko à 1 Mo) et un programme utilisateur entre les deux.

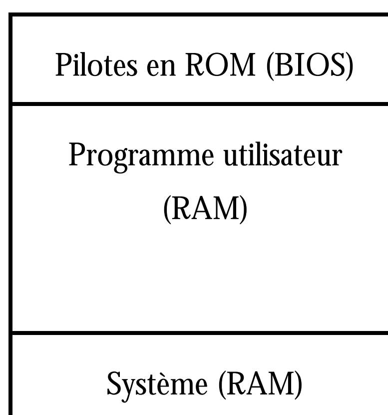


Figure 11 L'organisation de la mémoire du DOS.

La multiprogrammation autorise l'exécution de plusieurs processus indépendant à la fois[9]. Cette technique permet d'optimiser le taux d'utilisation du processeur en réduisant notamment les attentes sur des entrées-sorties. La multiprogrammation implique le séjour de plusieurs programmes en même temps en mémoire et c'est cette technique qui a donné naissance à la gestion moderne de la mémoire.

2.1.2. Les registres matériels

La gestion de la mémoire est presque impossible sans l'aide du matériel. Celui-ci doit notamment assurer la protection. Dans les systèmes multi-utilisateurs, par exemple, on doit interdire à un utilisateur d'accéder n'importe comment au noyau du système, ou aux autres programmes des utilisateurs.

[9] Il est à noter que MS-DOS, bien qu'il ne soit pas multiprogrammable, n'est pas non plus un système monoprogrammé *stricto sensu*; c'est une créature qui tient des deux.

Pour assurer une protection fondamentale, on dispose, sur la plupart des processeurs[10], de deux registres délimitant le domaine d'un processus :

- le registre de base et
- le registre de limite.

La protection est alors assurée par le matériel qui compare les adresses émises par le processus à ces deux registres.

2.2. Concepts fondamentaux

2.2.1. Production d'un programme

Avant d'être exécuté, un programme doit passer par plusieurs étapes. Au début, le programmeur crée un fichier et écrit son programme dans un langage source, le C par exemple. Un **compilateur** transforme ce programme en un module objet. Le module objet représente la traduction des instructions en C, en langage machine. Le code produit est en général relogeable, commençant à l'adresse 00000 et pouvant se translater à n'importe quel endroit de la mémoire en lui donnant comme référence initiale le registre de base. Les adresses représentent alors le décalage par rapport à ce registre.

On peut rassembler les modules objets dans des bibliothèques spécialisées, par exemple au moyen des commandes `ar` et `ranlib` (pour avoir un accès direct) sous Unix ou `TLIB` avec le compilateur C de Borland. On réunit ensuite les bibliothèques dans un répertoire, en général `/usr/lib` sous Unix.

[10] Malheureusement pas sur le 8086 ce qui a eu des conséquences considérables sur les systèmes DOS et Windows.

Les appels à des procédures externes sont laissés comme des points de branchements. L'**éditeur de liens** fait correspondre ces points à des fonctions contenues dans les bibliothèques et produit, dans le cas d'une liaison statique, une image binaire. Certains systèmes, notamment Unix, autorisent des liaisons dynamiques et reportent la phase d'édition de liens jusqu'à l'instant de chargement. Il faut alors construire les objets et les bibliothèques d'une manière légèrement différente. Grâce à cette technique, on peut mettre les bibliothèques à jour sans avoir à recompiler et on encombre moins les disques.

Le **chargeur** effectue les liaisons des appels système avec le noyau, tel que l'appel `write` par exemple. Enfin, il charge le programme en mémoire.

Outre le code compilé – le texte – et la zone de données initialisées, le fichier exécutable contient un certain nombre d'autres informations. Dans le cas du système Unix, il y a notamment un en-tête composé d'un nombre « magique » (un code de contrôle de type), de diverses tailles (texte, données,...), du point d'entrée du programme, ainsi qu'une table de symboles pour le débogage. Dans le cas de MS-DOS, plusieurs formats cohabitent, notamment COM et EXE.

2.2.2. Principes de gestion

Pour effectuer le chargement, le système **alloue** un espace de mémoire libre et il y place le processus. Il libérera cet emplacement une fois le programme terminé.

Dans beaucoup de cas, il n'est pas possible de faire tenir tous les programmes ensemble en mémoire. Parfois même, la taille d'un seul programme est trop importante. Le programmeur peut, dans ce cas, mettre en œuvre une stratégie de recouvrement (*overlay*) consistant à découper un programme important en modules et à charger ces modules quand ils sont nécessaires. Ceci est cependant très fastidieux et nécessite, pour chaque nouveau programme, un redécoupage.

Les systèmes d'exploitation modernes mettent en œuvre des stratégies qui libèrent le programmeur de ces préoccupations. Il existe deux stratégies principales pour gérer les chargements : le **va-et-vient** et la **mémoire virtuelle**.

2.3. L’allocation

Avant d’implanter une technique de gestion de la mémoire centrale par va-et-vient, il est nécessaire de connaître son état : les zones libres et occupées; de disposer d’une stratégie d’allocation et enfin de procédures de libération. Les techniques que nous allons décrire servent de base au va-et-vient; on les met aussi en œuvre dans le cas de la multiprogrammation simple où plusieurs processus sont chargés en mémoire et conservés jusqu’à la fin de leur exécution.

2.3.1. État de la mémoire

Le système garde la trace des emplacements occupés de la mémoire par l’intermédiaire d’une table de bits ou bien d’une liste chaînée. La mémoire étant découpée en unités, en blocs, d’allocation.

2.3.1.1. Tables de bits

On peut conserver l’état des blocs de mémoire grâce à une table de bits. Les unités libres étant notées par 0 et ceux occupées par un 1. (ou l’inverse).

0	0	1	1	0	0	

Figure 12

La technique des tables de bits est simple à implanter, mais elle est peu utilisée. On peut faire la remarque suivante : plus l’unité d’allocation est petite, moins on a de pertes lors des allocations, mais en revanche, plus cette table occupe de place en mémoire.

2.3.1.2. Listes chaînées

On peut représenter la mémoire par une liste chaînée de structures dont les membres sont : le type (libre ou occupé), l'adresse de début, la longueur, et un pointeur sur l'élément suivant.

Pour une mémoire ayant l'état suivant :

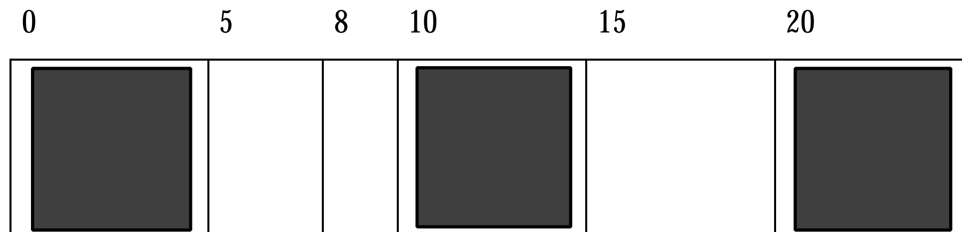


Figure 13

on aurait la liste :

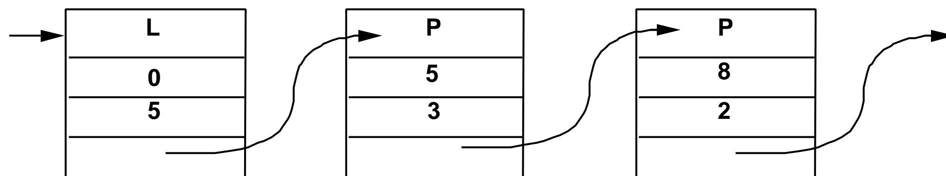


Figure 14

On peut légèrement modifier ce schéma en prenant deux listes : l'une pour les processus et l'autre pour les zones libres. La liste des blocs libres peut elle même se représenter en réservant quelques octets de chaque bloc libre pour contenir un pointeur sur le bloc libre suivant.

Le système MS-DOS utilise une variante de ce procédé grâce à un en-tête de 16 octets qui précède chaque zone (arène) en mémoire. Les en-têtes contiennent notamment le type de l'arène (un pointeur sur le contexte du processus[11] ou 0) et sa taille.

2.3.2. Politiques d'allocation

L'allocation d'un espace libre pour un processus peut se faire suivant trois stratégies principales : le « premier ajustement » (*first fit*), le « meilleur ajustement » (*best fit*), et le « pire ajustement » (*worst fit*).

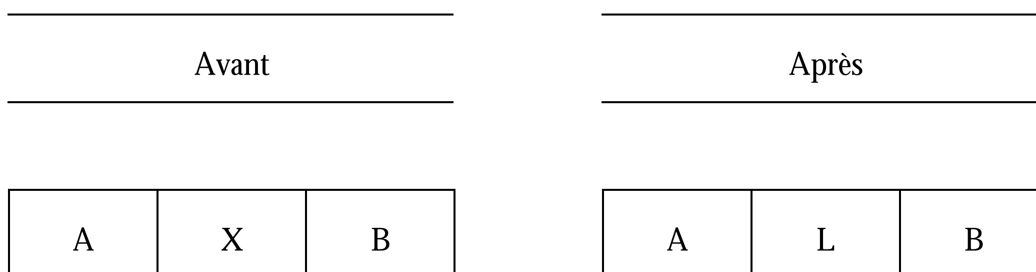
Dans le cas du « premier ajustement », on prend le premier bloc libre de la liste qui peut contenir le processus qu'on désire charger. Le « meilleur ajustement » tente d'allouer au processus l'espace mémoire le plus petit qui puisse le contenir. Le « pire ajustement » lui prend le plus grand bloc disponible et le fragmente en deux.

Des simulations ont montré que le « premier ajustement » était meilleur que les autres. Paradoxalement, le « meilleur ajustement », qui est plus coûteux, n'est pas optimal car il produit une fragmentation importante.

2.3.3. Libération

La libération se produit quand un processus est évacué de la mémoire. On marque alors le bloc à libre et on le fusionne éventuellement avec des blocs adjacents.

Supposons que X soit le bloc qui se libère, on a les schémas de fusion suivants :



[11] *Program Segment Prefix* ou PSP.

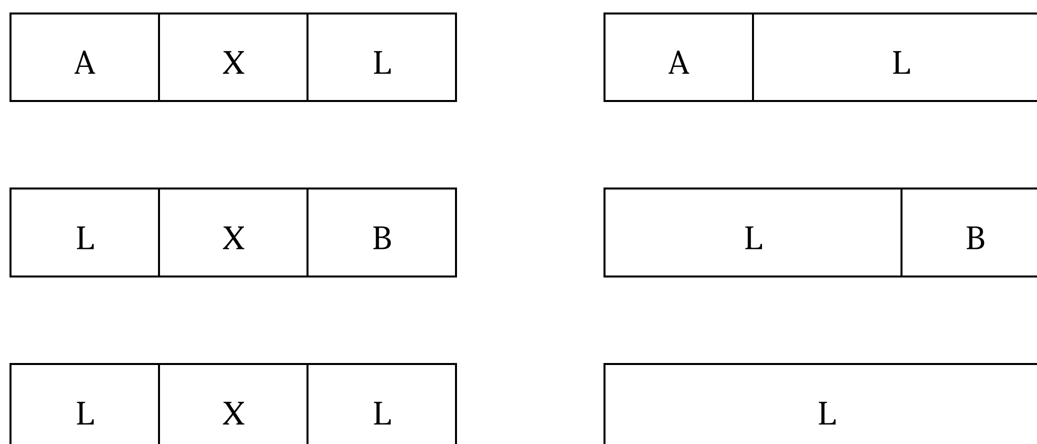


Figure 15

La récupération de mémoire

La fragmentation de la mémoire est particulièrement dommageable car elle peut saturer l'espace disponible rapidement. Ceci est particulièrement vrai pour les gestionnaires de fenêtrage. Pour la diminuer, on peut compacter régulièrement la mémoire. Pour cela, on déplace les processus, par exemple, vers la bas de la mémoire et on les range l'un après l'autre de manière contiguë. On construit alors un seul bloc libre dans le haut de la mémoire. Cette opération est coûteuse et nécessite parfois des circuits spéciaux.

Par ailleurs, une fois qu'un objet ou une zone a été utilisé, le programmeur système doit récupère la mémoire « à la main » par une libération du pointeur sur cette zone – `free()`. Ceci est une source d'erreurs car on oublie parfois cette opération. Si on alloue dans une fonction, il n'y a plus moyen d'accéder au pointeur après le retour de la fonction. Le bloc de mémoire est alors perdu et inutilisable. On a une « fuite de mémoire » – *memory leak*.

Certains langages ou systèmes incorporent la récupération automatique de mémoire – *garbage collection*. Ils libèrent ainsi le programmeur de cette tâche. C'est le cas de Java. Il est alors très facile d'éliminer immédiatement une zone par l'affectation `objet = null`. Sans ça, le récupérateur doit déterminer tout seul qu'une zone n'a plus de référence dans le suite du programme. La récupération

automatique de mémoire a fait l'objet de controverses mais les machines devenant plus rapides, cette querelle est sans doute dépassée[12].

Les algorithmes de récupération de mémoire sont nombreux et variés. Ils sont différents suivant les implantations de langage Java. Nous en présentons un simplifié : *stop and copy*. Un fil d'exécution de basse priorité est associé à la récupération de mémoire. On dit que la récupération est asynchrone. Quand il décide de récupérer, il arrête les autres fils. Il balaye les objets en mémoire et répertorie ceux qui ne sont plus référencés par un pointeur valide. Il les marque à libre. Pour ceci, le récupérateur doit tenir compte des zones non référencées et des structures cycliques inutilisées qui pourraient ne jamais être libérées. Il gère les structures cycliques en parcourant les graphes de pointeurs pour déterminer les zones accessibles et par complément les zones inaccessibles. Il compacte ensuite la mémoire libre en déplaçant les zones qui sont en cours d'utilisation.

L'algorithme de récupération peut poser des problèmes car périodiquement le système s'arrête brutalement pour laisser la place au récupérateur. On peut demander son démarrage explicite (synchrone) par la méthode `java.lang.Runtime.gc()`. La récupération est aussi dite synchrone lorsqu'il n'y a plus de mémoire dans le tas. Pour pouvoir appeler le récupérateur, il faut lancer l'interprète avec l'option : `-noasyncgc`. On obtient la quantité de mémoire libre avec la méthode `Runtime.freeMemory()`. Le tas est par défaut de 1 Mo. On peut positionner sa valeur en lançant l'interprète avec l'option `-ms16m` (pour 16 Mo).

2.4. Le va-et-vient

Le va-et-vient est mis en œuvre lorsque tous les processus ne peuvent pas tenir simultanément en mémoire. On doit alors en déplacer temporairement certains

[12] Selon les tenants du Prolog (ou du Lisp), la récupération est trop importante pour la laisser aux programmeurs et selon ceux du C++, elle est trop importante pour la laisser au système.

sur une mémoire provisoire, en général, une partie réservée du disque (*swap area* ou *backing store*).

Sur le disque, la zone de va-et-vient d'un processus peut être allouée à la demande dans la zone de va-et-vient générale (*swap area*). Quand un processus est déchargé de la mémoire centrale, on lui recherche une place. Les places de va-et-vient sont gérées de la même manière que la mémoire centrale. La zone de va-et-vient d'un processus peut aussi être allouée un fois pour toute au début de l'exécution. Lors du déchargement, le processus est sûr d'avoir une zone d'attente libre sur le disque.

Le système exécute pendant un certain quantum de temps[13] les processus en mémoire puis déplace un de ces processus au profit d'un de ceux en attente dans la mémoire provisoire. L'algorithme de remplacement peut être le tourniquet.

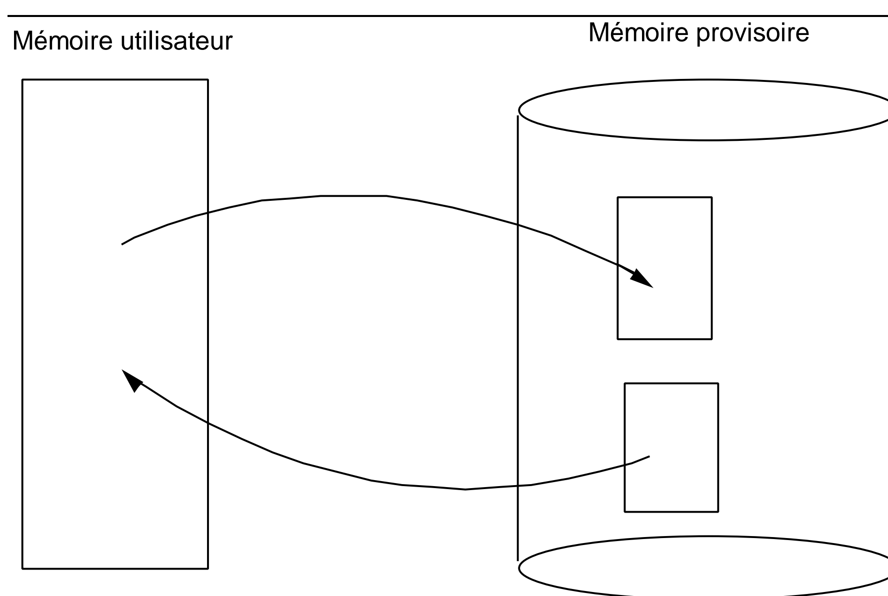


Figure 16

[13] Ce quantum est bien sûr plus long que le quantum de commutation de tâches vu au chapitre sur les processus. Il tient compte du fait que les temps de chargement et de déchargement sont beaucoup plus longs que les temps de commutation de processus.

Le système de va-et-vient, s'il permet de pallier le manque de mémoire nécessaire à plusieurs utilisateurs, n'autorise cependant pas l'exécution de programmes de taille supérieure à celle de la mémoire centrale.

2.5. La pagination

La pagination permet d'avoir en mémoire un processus dont les adresses sont non contiguës. Pour réaliser ceci, on partage l'espace d'adressage du processus et la mémoire physique en pages de quelques kilo-octets. Les pages du processus sont chargées à des pages libres de la mémoire.

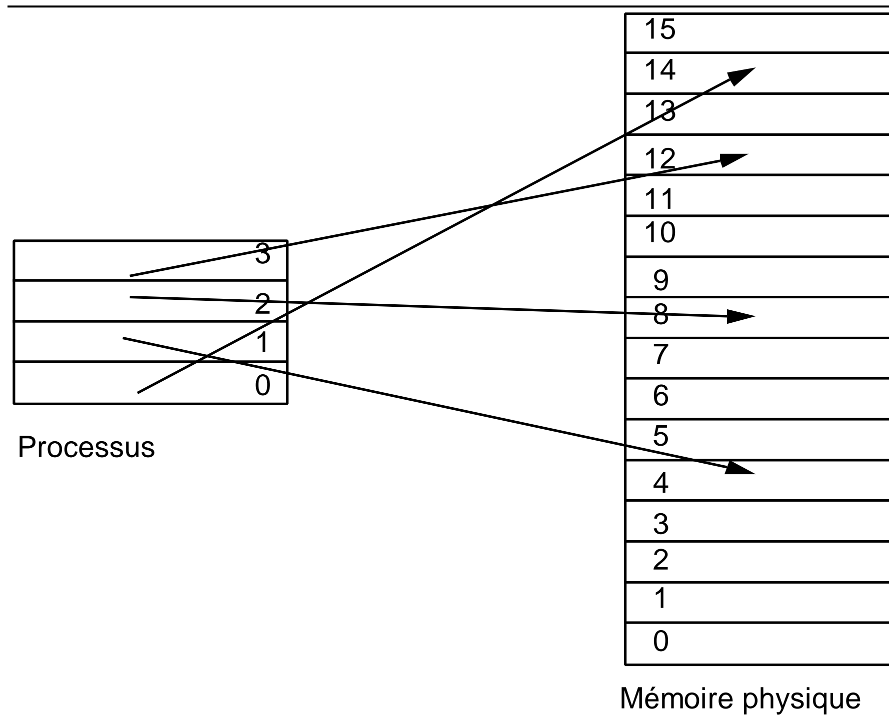


Figure 17

On conserve l'emplacement des pages par une table de transcodage :

0	14
1	4
2	8
3	12

La pagination permet d'écrire des programmes ré-entrants, c'est à dire où certaines pages de codes sont partagées par plusieurs processus.

2.6. La segmentation

Alors que la pagination propose un espace d'adressage plat et indifférencié, (ceci est offert par la famille de µ-processeurs 68000), la segmentation partage les processus en segments bien spécifiques. On peut ainsi avoir des segments pour des procédures, pour la table de symboles, pour le programme principal, etc. Ces segments peuvent être relogeables et avoir pour origine un registre de base propre au segment. La segmentation permet aussi le partage, par exemple du code d'un éditeur entre plusieurs processus. Ce partage porte alors sur un ou plusieurs segments.

Un exemple réduit d'architecture segmentée est donné par la famille 8086 qui possède 3 segments : le code (Code Segment), la pile (Stack Segment) et les données (Data Segment).

2.7. La mémoire virtuelle

2.7.1. Présentation

La mémoire virtuelle permet d'exécuter des programmes dont la taille excède la taille de la mémoire réelle. Pour ceci, on découpe (on « pagine ») les processus ainsi que la mémoire réelle en pages de quelques kilo-octets (1, 2 ou 4 ko généralement).

L'encombrement total du processus constitue l'espace d'adressage ou la mémoire virtuelle. Cette mémoire virtuelle réside sur le disque. À la différence de la pagination présentée précédemment, on ne charge qu'un sous-ensemble de pages en mémoire. Ce sous ensemble est appelé l'espace physique (réel).

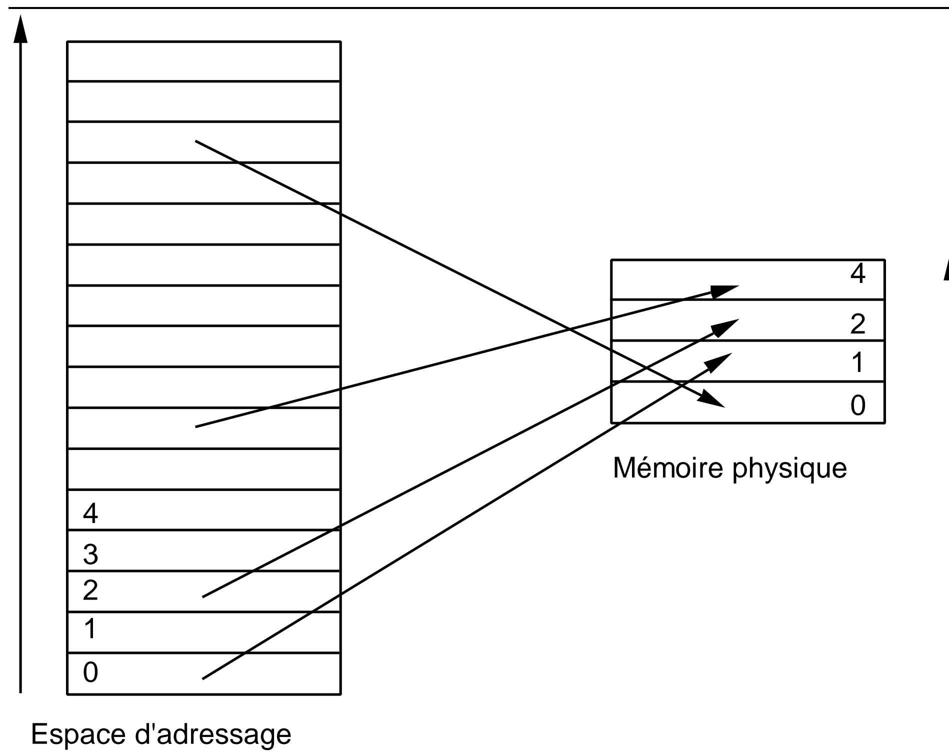


Figure 18

Lorsqu'une adresse est générée, elle est transcodée, grâce à une table, pour lui faire correspondre son équivalent en mémoire physique. Ce transcodage est effectué par des circuits matériels de gestion : *Memory Management Unit* (MMU). Si cette adresse correspond à une adresse en mémoire physique, le MMU transmet sur le bus l'adresse réelle, sinon il se produit un **défaut de page**. Pour pouvoir accéder à la page dont on a généré l'adresse, on devra préalablement la charger en mémoire réelle. Pour cela, on choisit parmi les pages réelles une page « victime »; si cette dernière a été modifiée on la reporte en mémoire virtuelle (sur le disque) et on charge à sa place la page à laquelle on désirait accéder.

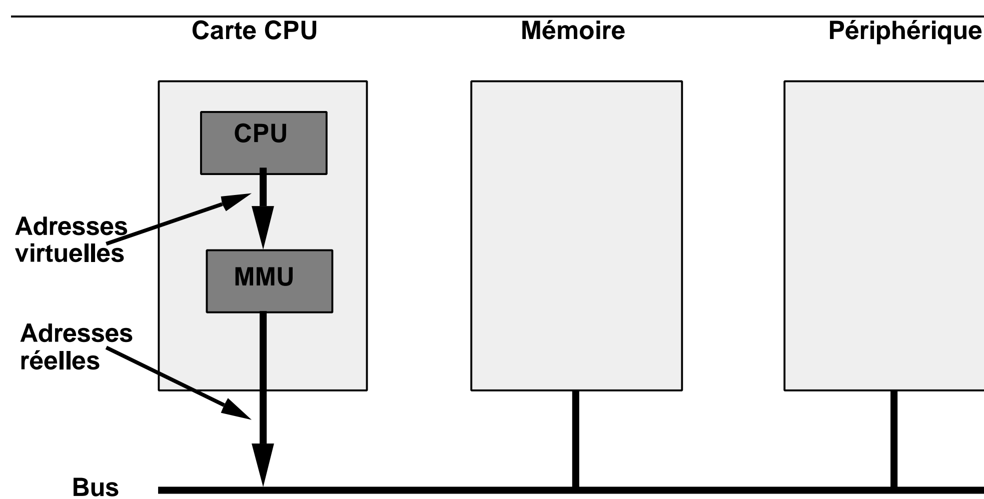


Figure 19 L'unité de gestion de mémoire.

Il est plus facile d'implanter l'algorithme en utilisant des tailles correspondant à des puissances de deux. Pour une adresse virtuelle ou réelle, on réserve les bits de poids forts nécessaires pour coder les pages réelles et virtuelles. Les bits de poids faibles codent les décalages à l'intérieur de chacune de ces pages. Par exemple, supposons que les pages fassent 4 ko; que la taille du processus fasse 16 pages; que la taille allouée en mémoire soit de 4 pages. Il faut 2 bits pour pouvoir coder les pages réelles, 4 bits pour les pages virtuelles et 12 bits pour les décalages.

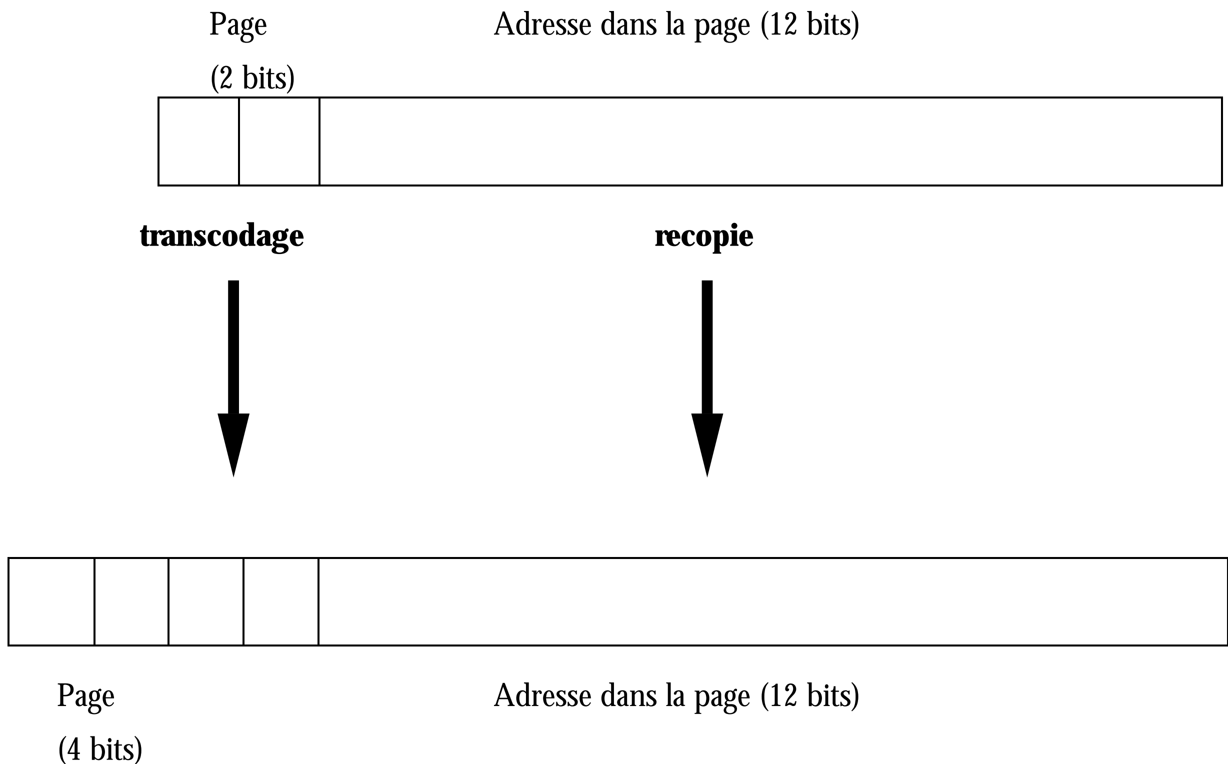


Figure 20 Structure des adresses mémoire de l'espace d'adressage

Pour réaliser le transcodage, on conserve des tables contenant les données nécessaires avec notamment : un bit pour marquer le présence de la page en mémoire réelle et un bit de modification pour signaler si on a écrit dans la page. Dans ce dernier cas, la page devra être reportée sur le disque si on désire la remplacer par une autre.

P. Virt.	P. Réel.	Présent	Modif.
0	01	0	
1	–	X	
2	10	0	
3	–	X	
4	–	X	
15	00		

Figure 21

2.7.2. Algorithmes de remplacement de pages

L'algorithme de remplacement de page optimal consiste à choisir comme victime, la page qui sera appelée le plus tard possible. On ne peut malheureusement pas implanter cet algorithme, mais on essaye de l'approximer le mieux possible, avec de résultats qui ont une différence de moins de 1 % avec l'algorithme optimal.

La technique FirstIn-FirstOut est assez facile à implanter. Elle consiste à choisir comme victime, la page la plus anciennement chargée.

L'algorithme de remplacement de la page la moins récemment utilisée (*Least Recently Used*) est l'un des plus efficaces. Il nécessite des dispositifs matériels particuliers pour le mettre en œuvre. On doit notamment ajouter au tableau une colonne de compteurs. Par logiciel, on peut mettre en œuvre des versions dégradées.

2.7.3. Autres considérations

2.7.3.1. L'écroulement (Thrashing)

On peut allouer les pages de mémoire physique en nombre égal pour chaque processus. Par exemple, si la mémoire totale fait 100 pages et qu'il y a cinq processus, chaque processus recevra 20 pages. On peut aussi allouer les pages proportionnellement aux tailles des programmes. Si un processus est deux fois plus grand qu'un autre, il recevra le double de pages.

Ces techniques d'allocation, utilisées telles quelles, peuvent provoquer un écroulement du système. En effet, ce dernier n'est viable que si les défauts de pages sont contenus au dessous d'une limite relativement basse. Si le nombre de processus est trop grand, l'espace propre à chacun sera insuffisant et ils passeront leur temps à gérer des défauts de pages.

On peut limiter le risque d'écroulement en considérant des abaques de comportement. Le nombre de défauts de pages en fonction du nombre de pages allouées à la forme d'une hyperbole. Si un processus provoque trop de défauts de pages (au dessus d'une limite supérieure) on lui allouera plus de pages; au-dessous d'une limite inférieure, on lui en retirera. Si il n'y a plus de pages disponibles et trop de défauts de pages, on devra suspendre un des processus.

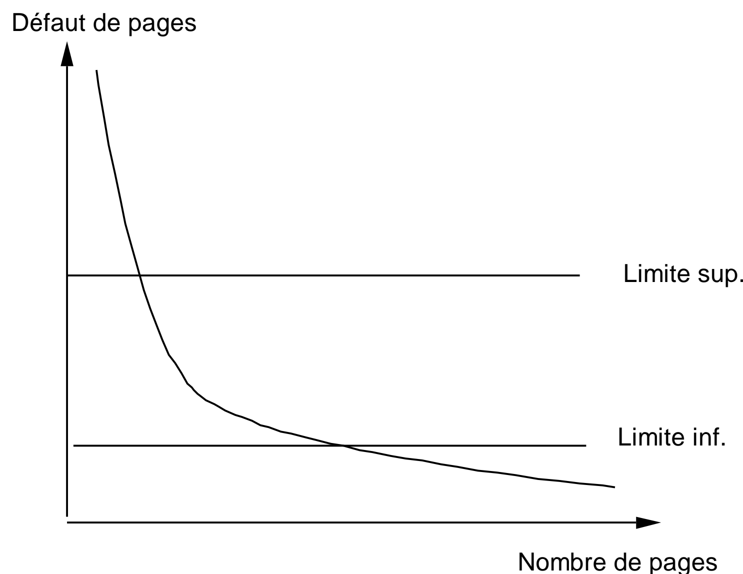


Figure 22

2.7.3.2. L'ensemble de travail

Pour déterminer l'espace viable d'un processus, on utilise le modèle de l'ensemble de travail. L'ensemble de travail est constitué par les zones du processus auxquelles on accède sur un court instant de temps (une dizaine de références à la mémoire). Par exemple, le mise à jour d'un individu dans une base de données, utilisera certaines pages de code et la page correspondant à l'individu.

Des simulations ont montré que cet ensemble de travail est relativement stable à un instant donné. Une allocation optimale pourrait chercher à allouer à chaque processus en fonctionnement autant de pages que nécessite son espace de travail. Dans ces conditions, les défauts de pages seront provoquées uniquement lors des changements d'espace de travail. En fait ce modèle n'est utilisé que pour la prépagination.

2.7.3.3. L'allocation locale ou globale

Lorsqu'on retire une page de la mémoire centrale, on peut choisir la plus ancienne :

- du point de vue global (la plus ancienne du système);
- du point de vue local (la plus ancienne du processus).

En général, l'allocation globale produit de meilleurs résultats.

2.7.3.4. La prépagination

Lors du lancement d'un processus ou lors de sa reprise après une suspension, on provoque obligatoirement un certain nombre de défauts de pages. On peut essayer de les limiter en enregistrant, par exemple, l'ensemble de travail avant une suspension. On peut aussi essayer de le deviner. Par exemple, au lancement d'un programme, les premières pages de codes seront vraisemblablement exécutées.

2.7.3.5. Le retour sur instructions

Sur la plupart des processeurs, les instructions se codent sur plusieurs opérandes. Si un défaut de page se produit au milieu d'une instruction, le processeur doit revenir au début de l'instruction initiale et la réexécuter. On devra alors lui donner les moyens de déterminer l'adresse du premier octet et éventuellement d'annuler certaines incrémentations.

Ce retour sur instruction n'est possible qu'avec l'aide du matériel. Par exemple, le 68010 possède un registre qui mémorise les adresses des premières instructions et les incréments. Il rend possible la pagination. Le 68000 n'en dispose pas et ne peut pas le faire.

2.8. Un exemple de gestion de la mémoire sur un micro-processeur : le 386

L'examen du processeur 386 est intéressant car il combine des techniques de mémoire paginée à des techniques de segmentation[14]. Il dispose de 16 k segments indépendants ayant une capacité allant jusqu'à 1 milliard de mots de 32 bits.

La mémoire virtuelle est fondée sur deux tables :

- la table locale de descripteurs, (LDT), propre à chaque programme. Elle contient ses segments avec notamment les segments de code, de pile et de données;
- la table globale de descripteurs, (GDT), unique pour tout le système et partagée par tous les processus. Elle contient les segments du système et notamment ceux du noyau.

Le 386 dispose de six registres de segments. Pour accéder à un segment particulier, par exemple le segment de code d'un processus, il charge un sélecteur dans l'un des registres. Ce sélecteur correspond à un indice dans l'une des deux tables. Chaque entrée de ces tables contient l'adresse de base du segment, l'adresse de limite ainsi que certains autres champs. Grâce à l'adresse de base, le microprocesseur peut convertir les décalages générés en adresses linéaires de 32 bits.

Le 386 dispose d'une pagination optionnelle avec des pages de 4 ko. Lorsqu'elle est activée, l'adresse précédente est interprétée comme une adresse virtuelle et elle est convertie en adresse réelle. La pagination se fait sur deux niveaux. Pour ceci l'adresse générée est interprétée en fonction de 3 champs :

10 bits	10 bits	12 bits
Dir	Page	Offset

[14] C'est par ailleurs le processeur 32 bits le plus répandu.

Figure 23

Dir est un indice dans un tableau contenant un pointeur sur une table de pages. Dans cette table Page est un autre indice contenant lui aussi un pointeur (double indirection) sur une page réelle. À l'intérieur de cette page réelle, Offset est l'adresse de l'élément recherché.

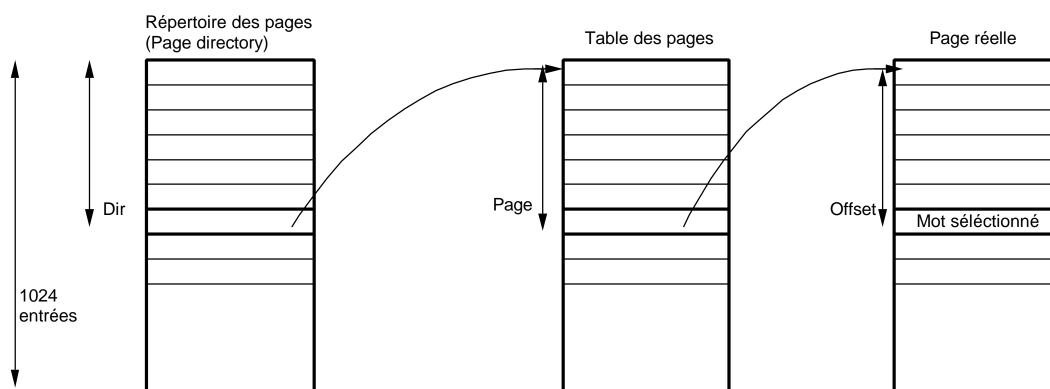


Figure 24

2.9. Appels système Unix

Cette partie est incomplète en 1999

`fork()` entraîne une allocation de mémoire

`exec()` entraîne une modification de mémoire

`wait()` entraîne une libération de mémoire

Les fichiers exécutables Unix contiennent notamment le texte et les données globales initialisées; les données non initialisées (BSS) sont allouées au chargement. En mémoire, les segments de code ne sont pas modifiables dans la

plupart des langage compilés[15] et sont partageables sur la plupart des systèmes modernes. Cette dernière caractéristique optimise l'occupation de la mémoire. En revanche, les autres segments (données et pile) sont propres à chaque processus.

Les segments de données et de pile sont continuellement modifiés au cours de l'exécution et leur taille peuvent varier. Les appels modifiant ces tailles sont les suivants :

`int brk(caddr_t addr)` permet de déplacer la frontière de la zone de données à `addr`. Elle peut être utilisée par `malloc`. Elle rend -1 si échec.

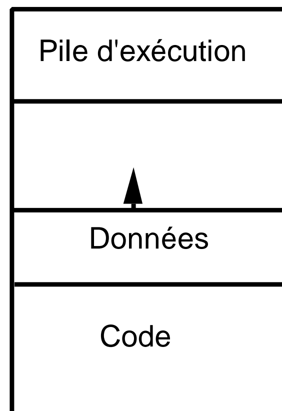


Figure 25

`caddr_t sbrk(int incr)` est similaire à `brk`. Au lieu de fixer l'adresse, elle étend la zone de données de `incr`.

`void *malloc(int incr)` alloue un espace contigu de taille `incr` dans la zone de données. Elle rend un pointeur sur la zone et `NULL` en cas d'échec. `Malloc` est une fonction assez primitive d'allocation de mémoire. Elle utilise l'algorithme du premier ajustement et ne recompacte pas les blocs libres de la zone de données. Elle entraîne de la fragmentation. Dans l'ouvrage sur le

---

[15] Les Prolog compilés sont des exceptions notables.

langage C de Kernighan et Ritchie, on trouve le code d'implantation de `malloc`. Il est intéressant à lire.

`void *calloc(int n_obj, int size)` rend un tableau.

`void realloc(void *p, int size)` réétend la zone allouée à une variable à `size`. Elle rend `NULL` si échec.

`void free(void *p)` libère la zone pointée par `p`.

2.10. La mémoire du DOS

Le système MS-DOS dispose d'un modèle de mémoire nettement plus compliqué que celui d'Unix. Il faut interpréter cette complexité comme un défaut : la simplicité d'Unix n'est en aucune façon synonyme de médiocrité, bien au contraire. La gestion de mémoire du DOS est, par ailleurs, très dépendante du matériel sous-jacent – Intel 8086 – qu'elle exploite jusqu'à la corde. L'architecture originale sur laquelle le DOS a été développé disposait d'un bus d'adresses de 16 bits, permettant d'adresser jusqu'à 64 ko de mémoire. Ultérieurement, l'adressage s'est fait à partir d'une base dont la valeur est contenue dans des registres spécifiques de 16 bits. Ces registres correspondent généralement à des fonctions de pile (SS), de données (DS) et de code (CS). Les registres contiennent les bits de poids forts d'une adresse de 20 bits, ce qui correspond à un intervalle de 1 Mo.

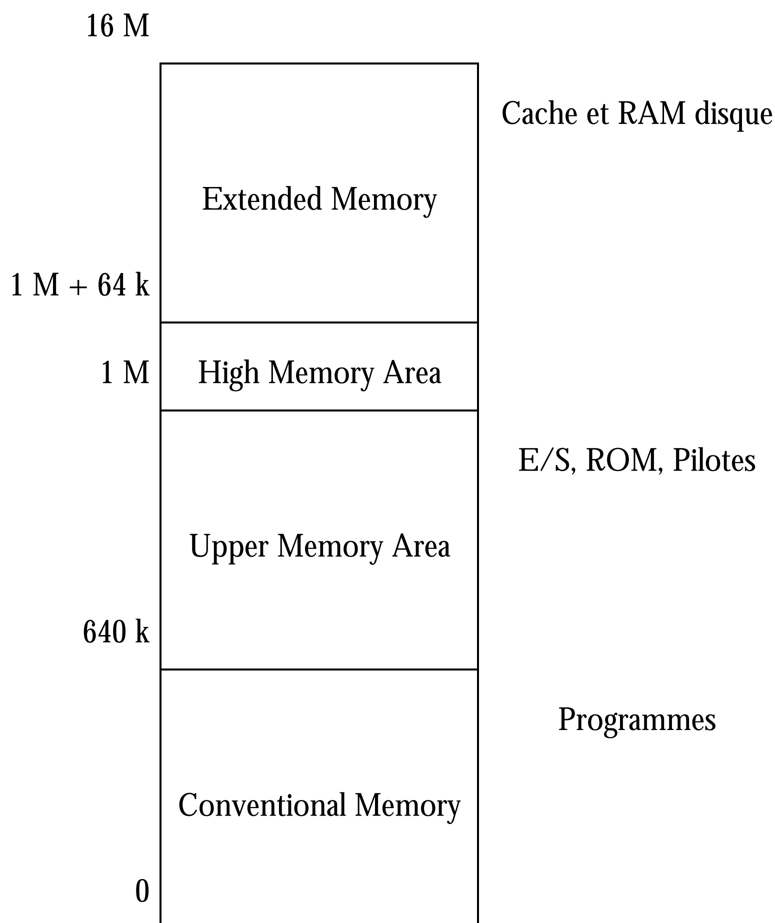


Figure 26

L'adressage du DOS est en fait restreint à 640 ko, la partie supérieure (Upper Memory Area) étant réservée à divers programmes de pilotage d'entrées-sorties.

Le modèle de mémoire du DOS permet des adresses au delà de 1 Mo. En prenant comme base, la valeur 0xFFFF, il est possible d'aller jusqu'à 1 Mo + 64 ko. Ceci correspond à la zone de mémoire haute (High Memory Area). L'exploitation de cette mémoire dépend de la manière – variable – dont est câblée la ligne d'adresse 21.

Les processeurs Intel actuels permettent des adressages étendus. Le 286 autorise 16 Mo; les 386 et suivants, 4 Go. Pour être compatible avec les processeurs 8086, Intel a fournit un mode de fonctionnement qui permet d'exploiter les machines actuelles avec l'ancienne gestion de mémoire : le mode «

réel ». Ce mode limite la taille à 640 ko. Cependant des modifications de MS-DOS permette d'exploiter la mémoire étendue pour des disques RAM, par exemple.

2.11 La mémoire de Windows

Les premières versions de Windows ont beaucoup souffert de l'architecture du 8086 et de l'héritage du DOS. Cependant la gestion de mémoire de Windows devient maintenant plus facile avec l'interface de programmation Windows. En effet, avec cette API, le programmeur voit la mémoire comme une zone d'adresses plate.

Windows réalise automatiquement le recompactage des données de la mémoire par un mécanisme de récupération des zones libres. Ceci est absolument nécessaire car le fenêtrage nécessite de multiples créations et destructions de mémoire et sans ce système, la mémoire se trouverait très vite en « mille morceaux ».

Après une opération de recompactage, dans le meilleur des cas, les blocs libres ne forment plus qu'un seul et grand segment. Dans la réalité, le compactage n'est souvent pas total, mais il permet une allocation ultérieure plus facile. Le recompactage à lieu régulièrement à l'initiative du système d'exploitation ou bien pour allouer de la mémoire qui à un moment donné ferait défaut. La récupération des zones libres existait déjà dans certains langage de programmation tels que le Prolog ou le Lisp et dans des systèmes d'exploitation tels que celui du Macintosh.

L'allocation de mémoire utilise des pointeurs de pointeurs – des *handles*. Ces handles (`HGLOBAL`) sont conservés dans un segment de mémoire : le BurgerMaster[16]. Le handle qui référence le bloc mémoire sera constant, en revanche, le pointeur sur la mémoire réelle lui sera variable au gré du système

[16] C'était le restaurant préféré des développeurs de Windows, paraît-il.

d'exploitation. Pour manipuler les données, on devra verrouiller le segment et utilisateur les pointeurs qui alors seront constants.

HGLOBAL GlobalAlloc(UNIT fuFlags, DWORD cbBytes) alloue un segment de longueur cbBytes avec les options fuFlags. Parmi ces options GMEM_MOVEABLE signale que le segment est relogeable, GMEM_FIXED signale que le segment est fixe. Si il y un échec, la fonction rend NULL et on en peut connaître la raison par la fonction GetLastError.

LPVOID GlobalLock(HGLOBAL) verrouille un segment et rend un pointeur. Si il y un échec, la fonction rend NULL.

BOOL GlobalUnlock(HGLOBAL) déverrouille un segment. En fait elle décrémente le compteur des verrous sur le segment. Si ce compteur passe à zéro, la valeur rendue est FALSE, sinon TRUE.

HGLOBAL GlobalFree(HGLOBAL) libère un segment. En cas de succès, la fonction rend NULL.

La mémoire de Windows NT

Windows NT reprend le mécanisme de tas de Windows mais il lui en ajoute un autre plus rapide. Un processus est créé avec un tas particulier et il peut en créer d'autres qui lui seront propres. Les fonctions qui manipulent les tas sont les suivantes :

HANDLE GetProcessHeap(void) permet de retrouver le tas par défaut de l'application. C'est dans ce tas que les appels GlobalAlloc trouvent de la place libre.

HANDLE HeapCreate(DWORD flOptions, DWORD dwInitialSize, DWORD cbMaximumSize) crée un nouveau tas privé pour le processus qui l'exécute. Le options possibles sont 0,

HEAP_GENERATE_EXCEPTION et HEAP_NO_SERIALIZE. Les deux autres paramètres sont généralement à 0.

`LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes)` permet d'allouer une zone de mémoire dans un tas. Cette zone est fixe et non déchargeable.

`DWORD HeapSize(HANDLE hHeap, DWORD dwFlags, LPCVOID lpMem)` retourne la taille d'un bloc dans un tas.

`DWORD HeapReAlloc(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem, DWORD dwBytes)` permet de modifier la taille d'une zone de mémoire dans un tas.

`BOOL HeapFree(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem) permet` de détruire une zone de mémoire dans un tas.

`BOOL HeapDestroy(HANDLE hHeap)` permet de libérer la totalité d'un tas.

Chapitre 3
Le système de fichiers

3.1. Qu'est ce qu'un fichier?

3.1.1. Une définition

Un fichier, dans un premier temps, peut s'assimiler à un ensemble de données sur disque. Cette vision n'est cependant que partielle et la plupart des systèmes d'exploitation, notamment Unix, en offre une notion plus générale. Dans cet exemple, les fichiers recouvrent aussi bien les données, sur une grande variété de supports, les programmes, l'ensemble des caractères frappés au clavier, ou l'horloge du système. Le système d'exploitation réalise alors la mise en correspondance de ces « fichiers » avec divers dispositifs physiques.

Les fichiers d'un système d'exploitation représentent une partie des objets que celui-ci manipule et avec lequel il échange des informations ; l'autre partie étant composée des processus. Il est important de distinguer le point de vue algorithmique sur les fichiers qui concerne toujours un ensemble de données, de celui que nous examinons ici.

L'algorithmique décrit la mise en œuvre d'un certain nombre de manipulations telles que la structuration de l'ensemble des données, l'insertion d'éléments, etc. Les systèmes d'exploitation considèrent le système de fichiers essentiellement comme une interface avec des objets relativement statiques. Cette interface dispose d'une structure propre, souvent hiérarchique, permettant de désigner un ensemble de données, mais aussi des périphériques.

L'interface du système de fichiers aura pour objectif de banaliser le plus possible les objets qu'elle recouvre, ainsi le système Unix masque complètement les disques locaux et les périphériques sous une arborescence unique. Il nomme aussi ses objets d'une manière uniforme. On copie par exemple le texte : « `abcd` », tapé au clavier, dans un fichier par la commande :

```
$ cat > fichier
abcd
^D
```

et sur un terminal, par la commande :

```
$ cat > /dev/tty
abcd
^D
```

Le système de fichiers répartis « NFS » permet des opérations similaires à travers un réseau.

3.1.2. La structure d'un système de fichiers

1. Le modèle hiérarchique

Les systèmes d'exploitation modernes adoptent une structure hiérarchique des fichiers. Chaque fichier appartient à un groupe d'autres fichiers et chaque groupe appartient lui même à un groupe d'ordre supérieur. On appelle ces groupes, des répertoires, ou des dossiers[17], suivant les terminologies.

Le schéma de la structure générale d'un système de fichiers prend l'aspect d'un arbre, formé au départ d'un répertoire « racine » recouvrant des périphériques et

[17] Ce terme, *folder* en anglais, consacré par le MacIntosh et repris par Windows 95, est nettement plus cohérent que le précédent. Cependant, il est peu utilisé par les informaticiens « purs et durs », c'est pourquoi nous garderons la désignation « répertoire ».

notamment un ou plusieurs disques. Dans chacun des répertoires on pourra trouver d'autres répertoires ainsi que des fichiers de données ordinaires.

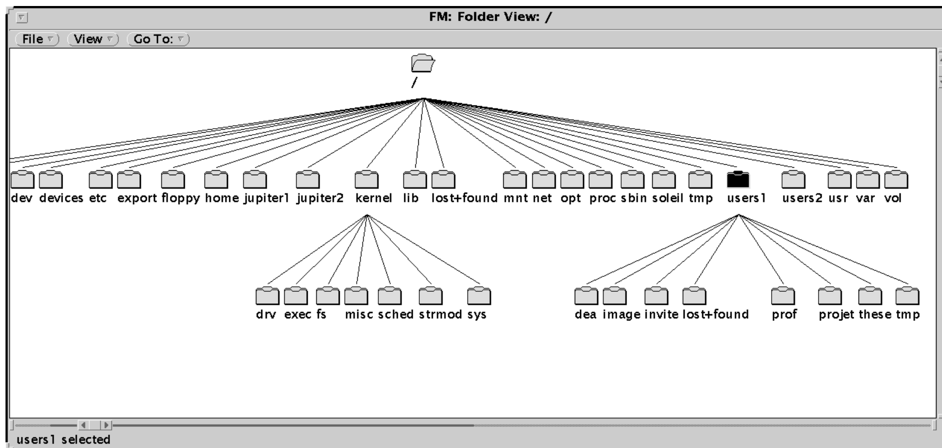


Figure 27 Une « vue » du gestionnaire de fichiers d'*Open Look*

2. Les répertoires et l'adressage d'un fichier

Les répertoires sont, eux aussi, des fichiers, constitués des noms et des références de tous les fichiers qu'ils contiennent. Cette structure permet alors de construire l'arborescence du système. Pour désigner un fichier quelconque, il suffit de spécifier l'enchaînement des répertoires nécessaires à son accès, à partir de la racine. Dans le système Unix, les répertoires de cet enchaînement sont séparés par une oblique : « / ». Dans le système DOS, par une contre-oblique : « \ ».

Dans le système Unix, chaque répertoire contient aussi sa propre référence, ainsi que celle du répertoire immédiatement supérieur. « **.** » désigne le répertoire courant, et « **..** », le répertoire supérieur. L'inclusion de ces deux références permet de désigner un fichier quelconque, relativement au répertoire courant.

Pour effectuer des entrées-sorties par une désignation relative, un processus ne peut s'exécuter que dans un répertoire. La référence de ce fichier est donc conservée dans la table de chaque processus.

En dehors des fichiers ordinaires et des répertoires, nous avons vu que le système Unix possède des fichiers spéciaux de périphériques blocs et caractères. Il connaît aussi les tubes nommés, qui sont des structures FIFO (FirstIn-FirstOut) permettant la communication entre processus.

La commande `ls -l` permet de vérifier le type des fichiers : « – » désigne les fichiers ordinaires, « `d` » les répertoires, « `c` » les périphériques caractères, « `b` » les périphériques blocs, et « `p` » les tubes nommés.

3.1.3. Accès aux éléments d'un fichier

Les fichiers de données sur disques se composent d'un ensemble de blocs, comprenant un nombre fixes d'octets. La première structure possible de ces fichiers correspond à la suite des octets des blocs; ces blocs étant ordonnés. L'accès à un octet se fait alors par un déplacement à partir de l'origine du premier bloc : le début du fichier. Le système Unix, ainsi que le système DOS ne connaissent que cet accès.

Certains systèmes structurent les fichiers sous la forme d'enregistrements séquentiels de tailles fixes, l'ensemble des blocs étant toujours ordonné linéairement. Ce principe est à peu près le même que le précédent. Il permet de lire, de substituer un élément ou d'en ajouter d'autres à la fin, mais on ne peut pas en détruire ou en insérer de nouveaux au milieu du fichier.

L'accès séquentiel indexé est une méthode plus élaborée, mise en œuvre, entre autres, sur les systèmes MVS d'IBM. On structure les blocs des fichiers sous la forme d'un arbre. Cette structure permet l'insertion d'éléments à n'importe quel endroit d'un fichier.

3.2. Les blocs du disques

Les fichiers de données sur les disques se répartissent dans des blocs de taille fixe correspondant à des unités d'entrées-sorties du contrôleur de ce

périphérique. La lecture ou l'écriture d'un élément d'un fichier impliquera donc le transfert du bloc entier qui contient cet élément. On peut formater les disques, – les rendre utilisables par le système d'exploitation –, avec une taille particulière de blocs. Cette taille résulte d'un compromis entre la vitesse d'accès aux éléments des fichiers et l'espace perdu sur le disque.

Lors d'un transfert de données d'un disque vers l'espace d'adressage d'un processus, le temps de lecture ou d'écriture sur le disque est négligeable devant le temps d'accès au bloc, et ceci quelque soit la taille du bloc. Pour un accès rapide, on aura donc intérêt à prendre des blocs de grande taille. Cependant, les fichiers, y compris les fichiers de 1 octet, ont une taille minimale de 1 bloc. Si un disque comprend beaucoup de fichiers de petite taille et si les blocs sont de grandes dimensions, l'espace gaspillé sera alors considérable.

Des études sur de nombreux systèmes ont montré que la taille moyenne d'un fichier est de 1 Ko[18]. Ce chiffre recouvre bien sûr de grandes variations entre, par exemple, une base de données et un ordinateur à usage pédagogique. Il conduit à des tailles courantes de blocs de 512, de 1024, ou de 2048 octets.

Chaque disque conserve, dans un ou plusieurs blocs spécifiques, un certain nombre d'informations de fonctionnement, telles par exemple que le nombre de ses blocs, leur taille,... . Il mémorise aussi en général l'ensemble de ses blocs ainsi que leur état dans une table. Si cette table est binaire, un disque de *n* blocs devra alors réserver une table de *n* bits; la position de chaque bit indiquant si le bloc est libre ou si il est utilisé par un fichier, par exemple, 0 pour un bloc occupé et 1 pour un bloc libre. Certains systèmes stockent l'ensemble des blocs libres dans une liste chaînée.

3.3. La répartition physique des fichiers en blocs

[18] Tanenbaum, op. cit., p. 285.

À chaque fichier correspond une liste de blocs contenant ses données. L'allocation est en générale non contiguë et les blocs sont donc répartis quasi-aléatoirement sur le disque. Les fichiers conservent l'ensemble de leurs blocs suivant deux méthodes : la liste chaînée et la table d'index.

3.3.1. La liste chaînée

L'ensemble des blocs d'un fichier peut être chaînée sous la forme d'une liste. Chaque bloc contiendra des données ainsi que l'adresse du bloc suivant. Le fichier devant mémoriser indépendamment le numéro du 1er bloc. Par exemple, si un bloc comporte 1024 octets et si le numéro d'un bloc se code sur 2 octets, 1022 octets seront réservés aux données et 2 octets au chaînage du bloc suivant.

Figure 28

Cette méthode rend l'accès aléatoire aux éléments d'un fichier particulièrement inefficace lorsqu'elle est utilisée telle quelle. En effet pour atteindre un élément sur le bloc *n* d'un fichier, le système devra parcourir les *n-1* blocs précédents.

Le système MS-DOS utilise des listes chaînées. Il conserve le premier bloc de chacun des fichiers dans son répertoire. Il optimise ensuite l'accès des blocs suivants en gardant leurs références dans une Table d'Allocation de Fichiers (FAT). Chaque disque dispose d'une FAT et cette dernière possède autant d'entrées qu'il y a de blocs sur le disque. Chaque entrée de FAT contient le numéro du bloc suivant.

Avec la table suivante[19] :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
X	X	EOF	13	2	9	8	L	4	12	3	L	EOF	EOF	L	BE	...

Figure 29

où « XX » indique la taille du disque, « L » désigne un bloc libre et « BE » un bloc endommagé, le fichier commençant au bloc 6, sera constitué des blocs : 6 → 8 → 4 → 2.

Le parcours de la FAT est nettement plus rapide que la chaîne des blocs. Cependant si elle n'est pas constamment, tout entière en mémoire, elle ne permet pas d'éviter les entrées-sorties du disque.

3.3.2. La table d'index

Le système Unix répertorie chaque fichier par un numéro unique pour tout un disque. À chaque numéro, correspond un enregistrement, un nœud d'index, (i-node), comportant un nombre fixé de champs. Parmi ces champs, certains, 13 au total, mémorisent l'emplacement physique du fichier. Les 10 premiers champs (sur les 13) contiennent les numéros des 10 premiers blocs composant le fichier.

Nœud d'index	bloc n° 1	bloc n° 2	bloc n° 3	bloc n° 4	...								

Figure 30

Pour les fichiers de plus de 10 blocs, on a recours à des indirections. Dans le cas du système Unix, le bloc n° 11 contient le numéro d'un bloc composé lui-même d'adresses de blocs de données. Si les blocs ont une taille de 1024 octets et

[19] Reprise de Tanenbaum, op. cit., p. 288.

si ils sont numérotés sur 4 octets, le bloc n° 11 pourra désigner jusqu'à 256 blocs. Au total, le fichier utilisant la simple indirection aura alors une taille maximale de 266 blocs. De la même manière, le bloc n° 12 contient une adresse, un pointeur, à double indirection, et le bloc n° 13, un pointeur à triple indirection. Avec l'exemple que nous avons donné, un fichier peut avoir une taille maximale de 16 Go quand il utilise ces pointeurs à triple indirection.

Bloc n° 11	Bloc n° 12	Bloc n° 13
Pointeur à simple indirection	Pointeur à double indirection	Pointeur à triple indirection

Figure 31

3.4. Les dossiers ou les « répertoires »

Les répertoires des systèmes de fichiers hiérarchiques sont des fichiers dont le contenu est particulier. Ils permettent de référencer et de retrouver physiquement les fichiers immédiatement en dessous d'eux dans la hiérarchie. La structure la plus commune des répertoires prend la forme d'un arbre, que nous avons illustré au paragraphe 1. On peut parfois mettre en œuvre des structures plus complexes, de graphes acycliques ou de graphes généralisés.

Un graphe permet à deux répertoires de référencer le même fichier. On a ainsi le moyen de partager un fichier entre, par exemple, deux utilisateurs. En général, on construit les répertoires de manière à éviter les cycles :

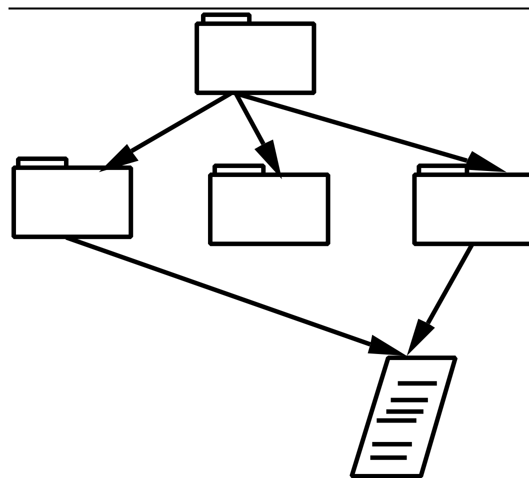


Figure 32

Cependant, on peut parfois construire des structures cycliques, telle que celles-ci :

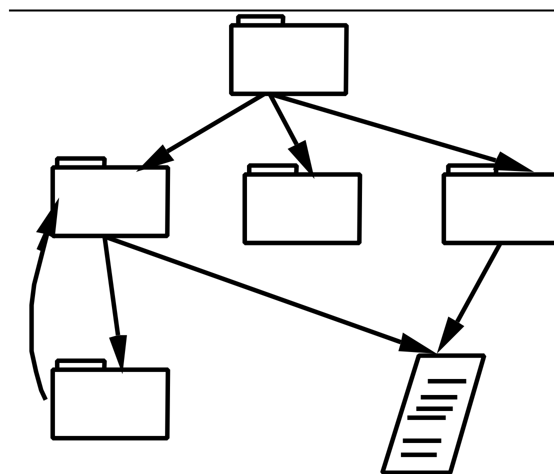


Figure 33

3.4.1. Les répertoires du système MS-DOS

Les répertoires du système MS-DOS possèdent une entrée par fichier et chaque entrée à la structure suivante :

Nom du fichier	Extension	Attributs	Réservé	Heure	Date	N° du 1er bloc	Taille

Figure 34

Le numéro du premier bloc, *cluster* dans la terminologie MS-DOS, indexe la Fat et permet de retrouver la suite des éléments du fichier.

3.4.2. Les répertoires d'Unix

Les répertoires du système Unix disposent, eux aussi, d'une entrée par fichier. Chaque entrée possède au moins les deux champs suivants, le numéro du nœud d'index et le nom du fichier. Par exemple :

26	.
59	..
267	toto
534	titi

Figure 35

La structure exacte d'une entrée varie suivant les versions d'Unix. Dans la version *System V*, elle décrite par le fichier d'en-tête `/usr/include/sys/dir.h`.

On peut visualiser, et interpréter, le contenu du répertoire courant, par exemple, grâce à la commande : « `od -c .` »

3.4.3. Structure et manipulation des nœud d'index

Le numéro du nœud d'index renvoie à un enregistrement sur le disque contenant les numéros des blocs, comme on l'a présenté au paragraphe

précédent. Cet enregistrement contient aussi des informations, concernant la gestion des fichiers[20], dont les principales sont les suivantes :

Nœud d'index
Uid du propriétaire
Gid du propriétaire
Type du fichier
Permissions
Nombre de liens
Taille du fichier
Date de création
Date du dernier accès
Date de dernière modif.
13 numéros de blocs et de pointeurs ...

Figure 36

Lorsqu'on manipule des fichiers, les nœuds d'index correspondants, auxquels s'ajoutent quelques champs supplémentaires, sont chargés en mémoire. Parmi les champs supplémentaires, on trouve notamment divers indicateurs, de verrouillage du nœud en mémoire, d'attente de processus sur le nœud, de modification en mémoire, du nœud ou du fichier, non encore reportée sur le disque. Le nœud en mémoire contient aussi, le numéro du périphérique (du disque) sur lequel se trouve le fichier, le numéro du nœud d'index, le nombre de fois où le fichier a été chargé en mémoire (le nombre d'ouvertures en cours)[21].

Pendant l'exécution d'un appel système concernant un fichier, le nœud d'index se verrouille. Deux processus ne peuvent donc pas effectuer, simultanément, une opération sur le même fichier. En cas de conflit, l'un des deux doit attendre. Ceci évite que le nœud ne se retrouve dans un état incohérent. Cependant ce nœud n'est pas verrouillé entre les appels système et des processus peuvent donc se

[20] Bach, p. 64, op. cit.

[21] Bach, p. 65, op. cit.

partager un fichier ouvert. Si un utilisateur désire un accès exclusif, il devra mettre en œuvre, par exemple, un sémaphore.

La position courante – de lecture et d'écriture – des processus dans un fichier, n'est pas contenue dans le nœud d'index, mais dans une table séparée, globale à tous les processus. Cette organisation est nécessaire pour que deux processus puissent lire ou écrire à des endroits différents du fichier. On repère les entrées de cette table par les descripteurs des fichiers[22], rendus par `open`, et que manipule `read`, `write`,... . En revanche, la position courante est partagée par les fils d'un processus, lorsque le fichier a été ouvert avant la création de ces fils. Les fils se réfèrent alors au fichier par le même numéro de descripteur. Grâce à ces propriétés, on peut, par exemple, établir une communication à travers un tube.

[22] Ces descripteurs de fichiers indicent une table privée (propre à chaque processus) dont les éléments pointent sur la cellule de la table globale où se trouve la position courante.

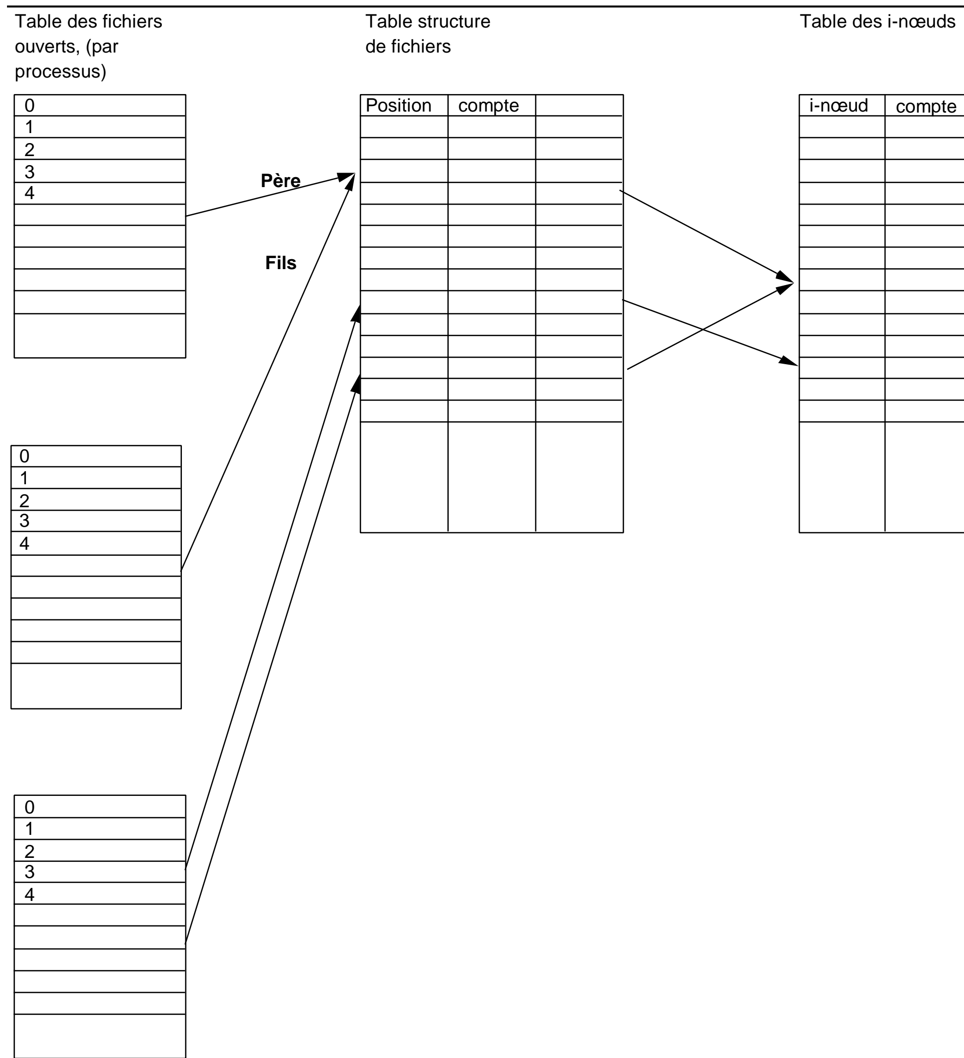


Figure 37

3.4.4. Le partage de fichiers par liens avec Unix

Un mécanisme permet de construire des structures de répertoires ayant la forme de graphes acycliques orientés et ainsi de partager des fichiers. Pour cela, plusieurs répertoires doivent référencer le même nœud d'index, éventuellement sous un nom différent. On appelle chaque nouveau référencement d'un nœud, la création d'un lien sur le fichier correspondant.

Une fois les liens établis, le fichier pourra être désigné sous l'un quelconque des noms. Pour sa part, le nœud d'index conservera, dans un de ses champs, le nombre de fichiers qui le référence, indépendamment des protections sur le

fichier. L'ajout d'une référence incrémentera le nombre de liens du nœud. L'élimination d'un fichier par un utilisateur décrémentera ce nombre. La destruction effective du nœud, et par là du fichier, aura lieu lorsque le compte des liens sera nul.

3.5. La mémoire cache

Les processus ne peuvent pas manipuler directement les données du disque. Ils sont obligés, pour cela, de les déplacer en mémoire centrale. Les transferts s'effectuent par blocs et il est souvent inutile d'effectuer autant d'entrées-sorties sur disque que d'accès au fichier. La plupart des systèmes de fichiers gère les entrées-sorties grâce à une mémoire intermédiaire : la mémoire cache ou l'antémémoire. Cette mémoire cache fait correspondre des blocs tampons en mémoires aux blocs du disque.

La stratégie générale d'utilisation de la mémoire cache est la suivante : on lui alloue un certain nombre de blocs en mémoire centrale. Lorsque l'on accède à un élément d'un fichier, on examine la suite de ces blocs. Si le bloc désiré – celui qui contient l'élément du fichier – se trouve dans la mémoire cache, on pourra y lire ou y écrire directement l'élément, sinon, on produira un bloc libre de la mémoire cache, on y chargera le bloc du disque et on effectuera les opérations de lecture ou d'écriture. Un bloc – un tampon – pourra être partagé dans la mémoire cache par plusieurs processus et il ne pourra s'y trouver qu'en un seul exemplaire.

Dans le système Unix[23], les blocs tampons sont reliés par un double chaînage des blocs libres et des blocs occupés. Ils possèdent, d'autre part, un en-tête indiquant le numéro du périphérique, le numéro du bloc, ainsi que l'état du tampon :

[23] Bach, chap. 3, op. cit.

N° du périph.	N° du bloc	État	Ptr sur le bloc de données	Ptr sur tampon occupé suiv.	Ptr sur tampon occupé préc.	Ptr sur tampon libre suiv.	Ptr sur tampon libre préc.

Figure 38

Le système Unix ordonne ses tampons suivant l'ordre dernière utilisation. En cas de demande de chargement d'un bloc du disque, il éliminera le moins récemment utilisé (algorithme LRU). Un codage par hachage accélère l'accès.

Il existe plusieurs politiques de recopie des blocs de la mémoire cache sur le disque lorsqu'ils ont été modifiés, en fonction d'un compromis entre la sécurité et la vitesse. Dans le système Unix, les nœuds d'index et les répertoires sont recopiés immédiatement après leur modification. Les blocs de données ordinaires sont recopiés si leur tampon atteint la fin de la liste LRU ou automatiquement par un démon toutes les 15 secondes. On peut déclencher ce démon par l'appel système par l'appel de la fonction `sync()`. Les tampons du système MS-DOS sont à recopie immédiate.

3.6. La structure physique d'un disque Unix

Chaque disque physique au format Unix possède la structure suivante :

Bloc démarrage	Super bloc	Table bits i-nœuds	Table bits blocs données	nœuds d'index (plusieurs blocs...)	Données (plusieurs blocs...)

Figure 39

Le bloc de démarrage contient le code nécessaire à tout disque pour se lancer.

Le super bloc donne des informations sur les fichiers, telles que le nombre de nœuds d'index, le nombre de blocs, le premier bloc de données, la taille maximale de fichiers, etc.

Le super bloc est chargé en mémoire où on lui ajoute quelques informations supplémentaires, telles que le numéro du périphérique, un indicateur de mise à jour, etc.

3.7. La structure logique d'un disque Unix

Le système Unix permet de réunir plusieurs disques sous une arborescence unique.

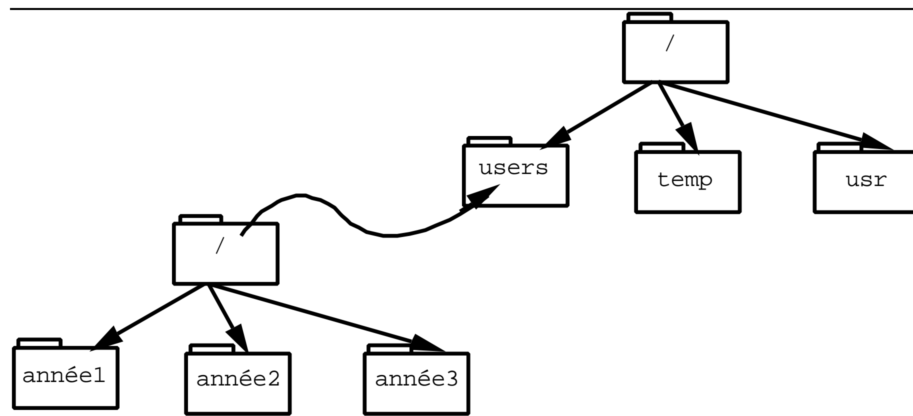


Figure 40

On réalise cet attachement par un « montage », grâce à la commande :

```
/etc/mount /dev/fd1 /users
```

où `/dev/fd1` désigne le disque à monter.

Le système « monté » devient :

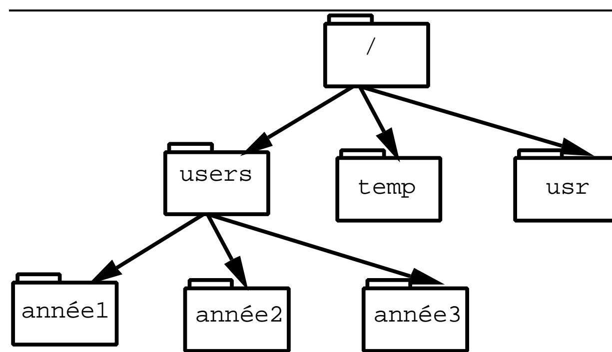


Figure 41

Une table des volumes montés garde la trace des différents périphériques[24]. Elle contient les éléments suivants :

N° du périph. monté	Ptr sur le tampon du super bloc	Ptr sur le i-nœud du système monté	Ptr sur le i-nœud du répert. de montage

Figure 42

Elle permet d'associer les disques à l'arborescence du système. Ainsi l'utilisateur peut parcourir les répertoires sans en supposer leur organisation physique.

3.8. Réparer le système de fichier

Les disques magnétiques sont des dispositifs délicats et il est fréquent de retrouver certains de leurs blocs, ou parfois même leurs références, dans un état incohérent.

Pour vérifier cette cohérence, on peut reconstruire la table des blocs occupés, en examinant tous les fichiers et en incrémentant le compte d'un bloc chaque fois

[24] Bach, p. 126, op. cit.

qu'il est référencé. Si le système est cohérent, la table des blocs occupés doit être complémentaire de celle des blocs libres.

Si un bloc n'apparaît ni dans la table des blocs libres, ni dans celle des blocs occupés, le bloc est dit manquant. On peut le réparer en le rajoutant à la liste des blocs libres.

Si un bloc appartient à deux ou plusieurs fichiers, le système est sans doute profondément incohérent. On peut effectuer une tentative de réparation en recopiant le bloc défectueux à des emplacements libres autant de fois qu'il est référencé puis en affectant chacun de ces nouveaux blocs à l'un des fichiers qui les référençaient.

On peut aussi vérifier la cohérence du point de vue des liens. Chaque numéro de nœud d'index doit apparaître autant de fois dans la structure arborescente qu'il possède de liens.

Sur la plupart des systèmes Unix, le programme `fsck` effectue cette tâche à chaque démarrage, si nécessaire.

3.9. La sécurité

Ce paragraphe est incomplet en 1999

3.9.1. La protection physique

3.9.2. La protection des fichiers

3.9.3. les listes d'accès

3.10. Les fonctions d'Unix

Les appels systèmes d'Unix sont en majorité destinés à la gestion des fichiers et des répertoires. Nous allons en donner ici une liste, ainsi qu'une description succincte. En cas d'erreur, les appels système rendent en général -1.

`int creat(char *ref, int mode)` crée, ou tronquera, le fichier `ref` avec les droits d'accès `mode`, masqués par le résultat de la fonction `umask`. `creat` alloue un nouveau nœud d'index, inscrit la référence dans le répertoire parent et ouvre le fichier en écriture. La fonction alloue un élément des tables globale et privée des fichiers. Elle rend un descripteur entier indexant cette dernière.

Nous avons décrit `int open(char *ref, int indicateurs)` dans les chapitres sur les processus et les entrées-sorties. Du point de vue du système de fichiers, `open`, alloue un élément des tables globale et privée des fichiers. La fonction rend un descripteur indiçant la table privée et elle positionne, par défaut, le déplacement à 0 dans la table globale.

`int read(int df, char *tampon, int nb)` lit `nb` octets du fichier de descripteur `df` et les place dans `tampon`. Elle rend le nombre de caractères réellement lus. La fonction `int write(int df, char *tampon, int nb)` lui est similaire.

`int close(int df)` libère, ou décrémente, les différents emplacements correspondants des tables de fichiers, ainsi que des nœuds d'index en mémoire.

`int lseek(inf df, long déplacement, int origine)` permet de positionner la lecture ou l'écriture à un endroit particulier du fichier de descripteur `df`, en fonction d'un déplacement `déplacement` par rapport à `origine`. `origine` peut prendre 3 valeurs, 0 indique le début du fichier, 1 la position courante et 2 la fin du fichier.

`int mknod(char *réf, int mode, int dev)` permet de créer un nœud d'index attaché à un fichier spécial.

int chdir(char *réf) permet à un processus de changer de répertoire de travail.

int chroot(char *réf) change le nœud d'index de la racine du système.

int chown(char *réf, int propr, int group) change le propriétaire et le groupe du fichier réf.

int chmod(char *réf, int mode) change les droits du fichier réf.

int stat(char *ref, struct stat *tampon) fournit le contenu du nœud d'index du fichier réf, à l'exception des numéros des blocs. Cette information sera rendue dans la structure tampon. Sa définition se trouve dans le fichier /usr/include/sys/stat.h

int fstat(int df, struct stat *tampon) est semblable à la fonction précédente, mais manipule le descripteur d'un fichier ouvert.

int dup(int df) rend le plus petit descripteur de fichiers disponible et le fait pointer sur même emplacement que df dans la table globale des fichiers.

int mount(char *périph, char *répert, int options) rattache le périphérique de fichier spécial périph au répertoire répert. Les options permettent le rattachement en lecture seule.

int umount(char *périph) détache le fichier spécial périph.

int link(char *fichier1, char *fichier2) crée la référence fichier2 et la rend équivalente à fichier1. Cet appel aura pour effet d'incrémenter le compte lien du nœud d'index de fichier1.

int unlink(char *fichier) détruit la référence fichier. Si le compte lien du nœud d'index correspondant devient nul, le fichier sera éliminé du disque.

`int access(char *réf, int mode)` permet d'effectuer des tests sur les fichiers.

`int fcntl(int df, int commande, int argument)` est une fonction à usages multiples. Elle permet de modifier finement le comportement des fichiers et notamment de verrouiller des enregistrements.

`int pipe(&df[0])` crée un tube de communication. Voir le chapitre sur les processus.

3.11. Les fonctions de Windows NT

Le système de fichiers de Windows NT est capable de fonctionner avec trois types de format : la FAT, NTFS et CDFS. Windows NT fournit un ensemble complet de fonctions pour gérer le système de fichier. Nous en en présentons ici un aperçu.

`DWORD GetCurrentDirectory()` permet d'obtenir le répertoire courant.

`BOOL SetCurrentDirectory()` permet de positionner le répertoire courant.

`BOOL CreateDirectory()` permet de créer un répertoire.

`BOOL RemoveDirectory()` permet de détruire un répertoire

`HANDLE CreateFile()` permet de créer un fichier

`BOOL CopyFile()` copie un fichier dans un autre.

`BOOL DeleteFile()` détruit un fichier.

`BOOL MoveFile()` déplace un fichier.

`BOOL ReadFile()` lit des données dans un fichier.

`BOOL WriteFile()` écrit des données dans un fichier.

`DWORD SetFilePointer()` positionne le point de lecture dans un fichier.

`BOOL SetEndOfFile()`

`BOOL FlushFileBuffer()` vide le tampon et force l'écriture sur le disque.

`BOOL LockFile()` verrouille un segment de fichier

`BOOL UnlockFile()` déverrouille un segment de fichier

3.12. L'interface du langage C

Le langage C fournit une interface de manipulation de fichiers, disposant d'une mémoire cache et permettant des formats de type courant : entiers, réels, ... Les fonctions de manipulation correspondent à l'ouverture, la fermeture, l'écriture, ...

L'interface du langage C réserve un tableau d'en-têtes, décrits par la structure `_iobuf`, dans le fichier `stdio.h`. À l'ouverture d'un fichier, il correspondra l'affectation d'un de ces en-têtes. Le tampon de l'en-tête, désigné par le champ `_base` de la structure sera alors alloué.

`_filbuf` est la fonction de remplissage. Elle fait passer les données, de la mémoire cache des blocs au tampon pointé par `_base`.

`_flsbuf` est la fonction de vidage du tampon, dans la mémoire cache des blocs. Elle intervient quand la tampon est plein ou à la rencontre d'un retour à la ligne "`\n`".

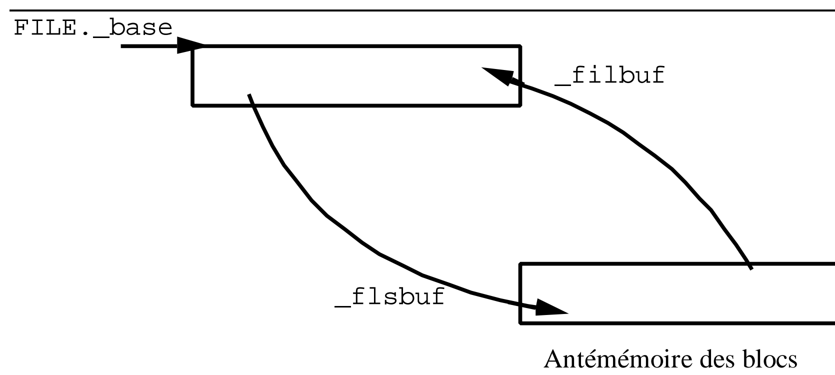


Figure 43

Les principales fonctions de manipulation des fichiers sont :

`FILE *fopen(char *nom, char *mode)` ouvre un fichier et recherche un en-tête vide dans la table.

`int fclose(FILE *)` libère l'en-tête.

`int getc(FILE *)` lit un caractère dans le tampon.

`int putc(char c, FILE *)` écrit un caractère dans le tampon.

`int fscanf(char *,` lit un ensemble de variables suivant un format.

`int fprintf(char *,` écrit un ensemble de variables suivant un format.

`int fseek(FILE *, long, int)` effectue un déplacement dans le fichier.

`fflush(FILE *)` vide immédiatement le tampon de la structure `_iobuf` dans la mémoire cache des blocs.

Le chapitre 8 du *Langage C*, de Kernighan & Ritchie fournit une description complète de la structure de données, ainsi que le code de la fonction `fopen()` écrit à partir de fonctions Unix. Dans ce chapitre de ce livre, les structures sont un peu simplifiées. La partie qui suit est un exemple réel :

```
/*  @(#)stdio.h 1.16 89/12/29 SMI; from UCB 1.4 06/30/83 */

#ifndef FILE
#define   BUFSIZ      1024
#define _SBFSIZ 8
extern    struct      _iobuf {
    int   _cnt;
    unsigned char *_ptr;
    unsigned char *_base;
    int   _bufsiz;
    short _flag;
    char  _file;            /* should be short */
} _iob[];

#define   _IOFBF      0
#define   _IOREAD     01
#define   _IOWRT      02
#define   _IONBF      04
#define   _IOMYBUF    010
#define   _IOEOF      020
#define   _IOERR      040
#define   _IOSTRG     0100
#define   _IOLBF      0200
#define   _IORW 0400
#define   NULL  0
#define   FILE  struct _iobuf
#define   EOF   (-1)

#define   stdin (&_iob[0])
#define   stdout      (&_iob[1])
#define   stderr      (&_iob[2])

#ifdef lint     /* so that lint likes (void)putc(a,b) */
extern int putc();
extern int getc();
#else
#define   getc(p)     (--(p)->_cnt>=0? ((int)*(p)->_ptr++):_filbuf(p))
#define   putc(x, p)  (--(p)->_cnt >= 0 ?\
    (int)(*(p)->_ptr++ = (unsigned char)(x)) :\
    (((p)->_flag & _IOLBF) && -(p)->_cnt < (p)->_bufsiz ?\
          ((*(p)->_ptr = (unsigned char)(x)) != '\n' ?\
                (int)(*(p)->_ptr++) :\
                _flsbuf(*(unsigned char *)(p)->_ptr, p)) :\
          _flsbuf((unsigned char)(x), p)))
#endif
#define   getchar()   getc(stdin)
#define   putchar(x)  putc((x),stdout)
#define   feof(p)           (((p)->_flag&_IOEOF)!=0)
#define   ferror(p)   (((p)->_flag&_IOERR)!=0)
#define   fileno(p)   ((p)->_file)
#define   clearerr(p) (void) ((p)->_flag &= ~(_IOERR|_IOEOF))

extern FILE     *fopen();
extern FILE     *fdopen();
extern FILE     *freopen();
extern FILE     *popen();
extern FILE     *tmpfile();
extern long     ftell();
```

```
extern char     *fgets();
extern char     *gets();
extern char     *sprintf();
extern char     *ctermid();
extern char     *cuserid();
extern char     *tempnam();
extern char     *tmpnam();

#define L_ctermid     9
#define L_cuserid     9
#define P_tmpdir      "/usr/tmp/"
#define L_tmpnam      25          /* (sizeof(P_tmpdir) + 15) */
# endif
```

Chapitre 4
Les entrées-sorties

4.1. Comment le processeur communique avec l'extérieur

4.1.1. Généralités

L'ordinateur peut effectuer des entrées-sorties – échanger des données – avec des organes externes :

- électroniques, tels que les mémoires,
- magnétiques, tels que les disques ou les disquettes,
- mécaniques, tels que le clavier, les imprimantes,...

Les périphériques – les organes externes – mis en œuvre sont de nature très différente et l'un des objectifs du système de gestion des entrées-sorties est de préserver une relative homogénéité dans leur mode d'accès. Les entrées-sorties forment une très part importante, au moins en volume de code, des systèmes d'exploitation. Leur étude, qui présente beaucoup plus d'aspects techniques que théoriques, est cependant parfois négligée.

On classe les périphériques d'entrées-sorties en deux catégories principales :

- les périphériques blocs;
- les périphériques caractères.

On gère les données des périphériques blocs par l'intermédiaire de blocs de tailles fixes, couramment de 512 ou 1024 octets et parfois plus. L'accès à chacun de ces blocs étant aléatoire. Les disques correspondent, à peu près, aux périphériques blocs.

Les autres périphériques sont donc, en général, des périphériques caractères. Ils fournissent ou acceptent une suite de caractères sans réelle structure. Les terminaux, les écrans, les mémoires sont des périphériques caractères.

On doit noter que cette classification des périphériques en deux catégories est assez grossière et, dans certains cas, plutôt arbitraire. Certains périphériques ne correspondent ni à l'une ni à l'autre définition, d'autres étant recouverts, par contre, par les deux définitions, les dérouleurs de bandes par exemple.

4.1.2. Les contrôleurs

La plupart du temps, le processeur ne commande pas directement les périphériques. Il utilise pour cela un circuit spécialement adapté, un « contrôleur », de caractéristiques propres à chaque périphérique. Les commandes du processeur au périphérique s'opéreront alors par l'intermédiaire de ce contrôleur. Ces commandes correspondent directement aux mécanismes physiques du périphérique telles, par exemple, que le déplacement du bras d'un lecteur de disque. Par ailleurs, les contrôleurs sont souvent d'une grande complexité électronique.

On relie les contrôleurs au bus de l'ordinateur et on leur alloue, à chacun, un certain nombre d'adresses. À ces différentes adresses, on émettra des commandes de pilotage ou on effectuera des entrées-sorties de données. Le contrôleur reconnaîtra ses adresses grâce à une logique de décodage. Dans certains ordinateurs, les contrôleurs ne sont pas reliés au bus mais à des voies d'entrées-sorties spécifiques.

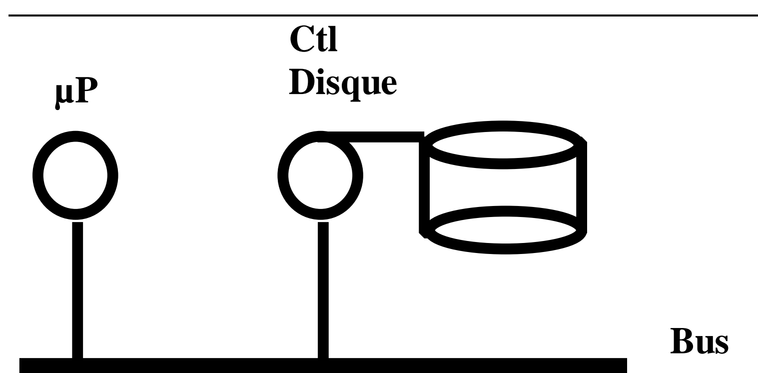


Figure 44

Les contrôleurs et le processeur opèrent en parallèle. Le contrôleur dispose d'une mémoire tampon, pour lui permettre, par exemple, de lire des données en provenance du périphérique d'entrée pendant que le processeur traite une autre tâche. Le contrôleur signale l'achèvement de son travail en émettant une interruption qui lui est propre. À la réception de cette interruption, le processeur se branche à un emplacement fixe. La valeur de cet emplacement est contenue dans une adresse – un **vecteur** – spécifique à chaque interruption[25]. À l'adresse d'interruption, une routine réalise, en général, le transfert des données vers la mémoire principale, puis rend la « main » au processus précédent.

Contrôleur	Adresses d'entrées-sorties	Vecteurs (IRQ)
Horloge	040-043	0
Clavier	060-063	1
Port COM 1	3F8-3FF	4

Table 1. Les adresses et les vecteurs de quelques périphériques de l'IBM PC.

[25] Et donc à ce contrôleur.

4.1.3. Un exemple simple de fonctionnement : écrire sur un disque

Pour transférer des données sur un disque, le pilote de ce disque émet des commandes, aux adresses adéquates, dans les registres du contrôleur. Ces commandes sont, la mise en route éventuellement, le positionnement de la tête sur le bon cylindre, puis l'instruction d'écriture, avec les paramètres nécessaires, notamment le secteur où écrire le bloc. Le pilote transmet ensuite les données à écrire, l'une après l'autre, dans les tampons du contrôleur puis il se bloque en attendant la fin de l'écriture effective.

Pendant ce temps, le processeur pourra élire une autre tâche. Le contrôleur écrira seul, sur le disque, les données contenues dans ses tampons et quand il aura terminé son action, il émettra une interruption. L'interruption débloquera le pilote et lancera l'ordonnanceur. Ce dernier pourra activer le pilote qui vérifiera alors que l'opération s'est bien déroulée.

4.1.4. L'accès direct à la mémoire

Pour les périphériques blocs, le transfert des données entre la mémoire et le contrôleur n'est pas, en général, réalisé par le processeur, mais par un circuit spécialisé d'accès direct à la mémoire (DMA).

Ce circuit DMA est attaché au périphérique. Il utilise des instants particuliers du cycle de fonctionnement du processeur pour écrire les données de la mémoire centrale vers les tampons du contrôleur, ou bien l'inverse. Il permet ainsi d'occuper le processeur à d'autres tâches de traitement. Avant de le lancer, le processeur doit bien sûr lui communiquer, le sens du transfert, l'adresse d'origine de la zone mémoire où on désire lire ou écrire des données, ainsi que le nombre d'octets à transférer.

4.2. Les différents niveaux des logiciels d'entrées-sorties

Le traitement des entrées-sorties se fait sur plusieurs niveaux, des plus proches du matériel aux plus proches de l'utilisateur. Les niveaux proches de l'utilisateur doivent s'efforcer, autant que possible, de masquer les particularités physiques des périphériques.

Les niveaux purement matériels sont constitués des interruptions émises par les différents périphériques vers le processeur. Ces interruptions correspondent à des indications, par exemple de fin de traitement. Le système détermine l'origine des interruptions et fait passer le pilote destinataire, en général, de l'état bloqué à l'état prêt. Les interruptions ont aussi souvent pour effet de relancer l'ordonnanceur. Si le pilote est de nouveau élu, et il le sera de toute façon à un moment, il transmettra au processus demandeur, les résultats de l'entrée-sortie, et il le débloquera à son tour. Du point de vue conceptuel, ces déblocages de ressources se formalisent par des opérations sur des sémaphores :

```
V(ressource).
```

Les pilotes de périphériques s'occupent de traduire les requêtes des entrées-sorties dans des commandes propres à chaque contrôleur. Ils gèrent aussi une partie des erreurs matérielles. Ils s'intègrent enfin, dans le cas d'Unix, dans le système de fichiers. Les fichiers pouvant recouvrir à la fois des données sur disques, des périphériques,... Ce système de fichiers à l'avantage de présenter une interface uniforme aux entrées-sorties :

- d'adressage,
- de protection,
- d'allocation et de libération des périphériques,
- de signalisation des erreurs, etc.

Pour l'utilisateur, les fonctions d'interface, les appels système, correspondent alors à l'ouverture ou à la fermeture d'un fichier, à la lecture ou à l'écriture de données, enfin, au positionnement des conditions d'échange pour les périphériques caractères.

4.3. Les dispositifs matériels d'entrées-sorties

Cette partie est incomplète en 1999

4.3.1. Les disques

Les disques sont constitués d'un ou plusieurs cylindres. Chaque cylindre possède un certain nombre de pistes. Un disquette simple face a une piste par cylindre, une double face en possède deux. Chaque piste est divisée en secteurs d'angles égaux. En général, on a un bloc par secteur, parfois il faut plusieurs secteurs pour un bloc. Les paramètres temporels importants sont les temps moyens de positionnement du bras et le temps de rotation.

Plusieurs algorithmes permettent d'ordonnancer le bras du disque. Le plus simple est celui qui sert les demandes dans leur ordre d'arrivée. (First Come-First Served). L'un des plus efficace est celui de l'ascenseur. Il traite les demandes en les ordonnant en fonction de la position de leur cylindre sur le disque, de l'intérieur vers l'extérieur et en opérant des balayages entre les deux secteurs extrêmes.

4.3.2. Les terminaux

Les terminaux, de type VT100, sont reliés à l'ordinateur par une voie RS-232 asynchrone. La transmission s'opère en série par l'intermédiaire d'un circuit UART (*Universal Asynchronous Receiver Transmitter*). La vitesse de transmission est, en général, paramétrable et s'exprime, de manière impropre, en bauds.

Les terminaux plus récents, tels que les consoles graphiques, ou les terminaux X, ont une mémoire vidéo. Les éléments qui composent l'écran, les caractères, les formes géométriques, les fenêtres, les images, sont transformés en une matrice de points. La matrice est transférée dans la mémoire vidéo. Un dispositif balaie cette mémoire périodiquement pour l'afficher.

Les logiciels des terminaux peuvent opérer en mode canonique, c'est à dire en filtrant et en interprétant certains caractères, tel que l'effacement. Il peuvent aussi tout transmettre au processeur, il s'agit alors du mode non-canonique.

4.3.3. Les mémoires

4.3.4. Les horloges

Les horloges sont un composant fondamental des systèmes à temps partagé. Elles fonctionnent en émettant des interruptions périodiques qui agissent, notamment, en déclenchant l'ordonnanceur.

*4.3.5. L'espace de va-et-vient (*swap area*)*

4.3.6. Une interface normalisée : SCSI

Cette interface tend à se répandre de plus en plus. Elle permet des transfert de 8/16 ou 32 bits en parallèle à des vitesses de 2 ou 5 MHz. Les dispositifs sont reliés sous la forme d'un chapelet : « daisy chain ». Le premier se connectant sur le port de l'ordinateur, le second sur le port de sortie du premier, etc. Beaucoup de disques magnétiques ou de disques optiques CD-ROM ont une interface SCSI.

4.4. Les interblocages

Un interblocage survient quand deux processus détiennent chacun une ressource en exclusivité et ont besoin, pour continuer, de la ressource de l'autre. Les interblocages dépassent largement le champ des entrées-sorties, mais ils peuvent se produire assez souvent à cette occasion.

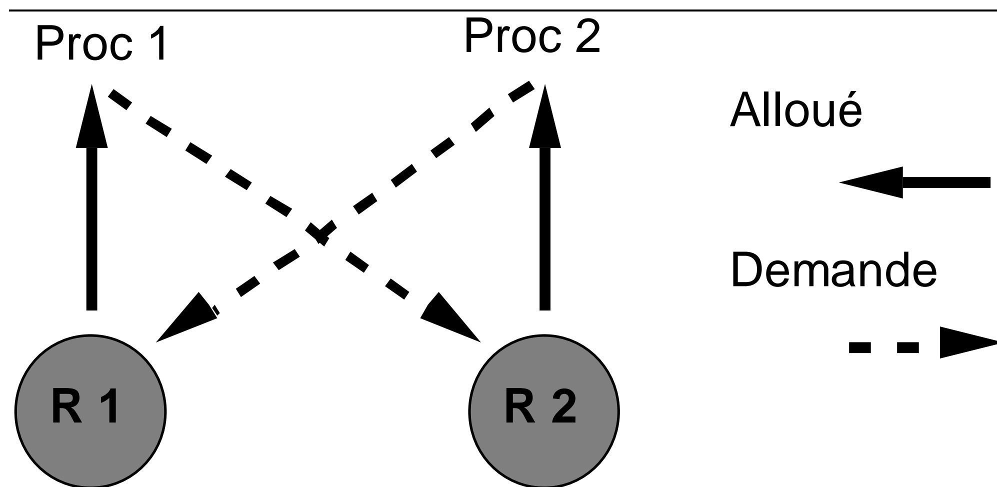


Figure 45

La résolution des interblocages constituent un point théorique très étudié. On dispose à son sujet d'une littérature abondante, mais parfois peu lisible. Cette débauche intellectuelle n'a cependant pas fourni, pour l'instant, de solution complétement satisfaisante et la plupart des systèmes – notamment Unix – ne cherche pas à traiter ce phénomène. Cependant, si on ne peut pas résoudre le problème dans sa généralité, on devra tenir compte de certaines techniques qui minimisent les risques.

4.4.1. Comment les interblocages se produisent

Des auteurs ont montré que les interblocages se produisent quand quatre conditions sont remplies :

1. l'exclusion mutuelle pour la prise d'une ressource;
2. la possibilité de détenir une ressource et d'attendre pour obtenir d'autres ressources;
3. l'absence de réquisition; on ne peut pas retirer d'autorité une ressource allouée;
4. Il se produit un cycle dans le graphe orienté représentant la détention et l'attente.

On peut faire l'autruche et ignorer les possibilités d'interblocages. Cette « stratégie » est celle de la plupart des systèmes d'exploitation courants. On peut aussi tenter de traiter les interblocages, en détectant les circularités ou les processus bloqués pendant trop longtemps et en les éliminant avec tous les dangers que cela comporte. On peut enfin tenter de prévenir les interblocages, de manière statique en considérant leurs conditions d'apparition, ou de les éviter dynamiquement en allouant les ressources des processus avec précaution.

4.4.2. La prévention des interblocages

Pour prévenir les interblocages, on doit éliminer une des quatre conditions nécessaires à leur apparition.

Pour éviter l'exclusion mutuelle, il est parfois possible de sérialiser les requêtes portant sur une ressource. Par exemple, pour les imprimantes, les processus « spoulent » leurs travaux dans un répertoire spécialisé et un démon d'impression les traitera, en série, l'un après l'autre.

Pour ce qui concerne la deuxième condition, elle pourrait être évitée si les processus demandaient leur ressources à l'avance. Ceci est en fait très difficile à réaliser dans la pratique car l'allocation est, en général, dynamique. Empêcher cette condition serait donc particulièrement coûteux.

La troisième condition n'est pas raisonnablement traitable pour la plupart des ressources sans dégrader profondément le fonctionnement du système. On peut cependant l'envisager pour certaines ressources dont le contexte peut être sauvegardé et restauré.

Enfin, on peut résoudre le problème de l'attente circulaire en numérotant les ressources et en n'autorisant leur demande, par un processus, que lorsqu'elles correspondent à des numéros croissants.

4.4.3. L'évitement

L'algorithme du banquier est une technique permettant d'allouer dynamiquement des ressources à des processus, en prenant soin qu'aucun des processus en cours ne puisse rester indéfiniment bloqué.

Considérons le cas où on dispose de plusieurs exemplaires de la même ressource. On construit un tableau où, pour chaque processus, on note le nombre maximal de ressources demandables et ainsi que le nombre de ressources allouées à un instant *t*. Par analogie avec une banque, le nombre maximal est, en quelque sorte, le crédit total du processus et le nombre alloué, son encours. La somme totale des crédits – des ressources – peut être supérieure au nombre de ressources réellement disponibles.

Dans ces conditions, on est certain de pouvoir terminer si la somme du nombre maximal de ressources allouable à un processus et des ressources allouées aux autres processus est inférieure ou égale au nombre total de ressources réellement disponibles. En effet, le premier processus pourra se terminer à condition de suspendre les autres. Il relâchera alors ses ressources et permettra aux suivants de poursuivre.

Par exemple, si le nombre total de ressources est de 6 et qu'on se trouve dans l'état suivant :

Processus	Res. allouées	Res. maxi.
P 1	1	3
P 2	0	1
P 3	2	3
P 4	1	4

Les ressources allouées sont au nombre de 4, pour rester dans un état sûr, on doit suspendre P1, P3 et P4, et permettre à P2 de terminer, puis laisser, soit P1, soit P3 se terminer.

On peut généraliser cet algorithme pour des ressources de plusieurs types.

4.5. Les fonctions Unix

L'écriture d'un pilote d'entrées-sorties est une tâche assez complexe. Nous n'examinerons que les grandes lignes de la structure des entrées-sorties sous Unix. Pour plus de détails, le lecteur pourra considérer les exemples se trouvant sur le serveur Hewlett-Packard dans les répertoires : `/usr/lib/drivers` ou `/systems/DRIVERS`.

4.5.1. Les périphériques et Unix

Le système Unix désigne chaque périphérique par un fichier. De manière courante, on rassemble les fichiers de tous les périphériques dans le répertoire : `/dev` *(device).* Ces fichiers sont appelés « spéciaux ».

On obtient leurs caractéristiques avec la commande : `ls -l`, ceci donne, par exemple, sur un ordinateur Sun du Laboratoire d'Informatique de l'ISMRA :

```
total 11
-rwxr-xr-x  1 root         9499 Feb  8  1990 MAKEDEV
crw-rw-rw-  1 root      37,  69 Mar 11 16:13 audio
crw-rw-rw-  1 root      69, 128 Mar 11 16:13 audioctl
crw-rw-rw-  1 root      68,   0 Mar 11 16:18 cgnine0
crw-rw-rw-  1 root      31,   0 Mar 11 16:18 cgtwo0
crw--w----  1 pierre     0,   0 Mar 25 21:49 console
crw-rw-rw-  1 root      11,   0 Mar 11 16:13 des
crw-r-----  1 root       7,   0 Mar 11 16:11 drum
crw-rw----  1 root      41,   0 Mar 11 16:11 dump
crw-r-----  1 root       3,  11 Mar 11 16:11 eeprom
crw-rw-rw-  1 root      22,   0 Mar 11 16:11 fb
brw-rw-rw-  2 root      16,   2 Mar 11 16:11 fd0
brw-rw-rw-  1 root      16,   0 Mar 11 16:11 fd0a
brw-rw-rw-  1 root      16,   1 Mar 11 16:11 fd0b
brw-rw-rw-  2 root      16,   2 Mar 11 16:11 fd0c
crw-rw-rw-  1 root      32,   0 Mar 11 16:18 gpone0a
crw-rw-rw-  1 root      32,   1 Mar 11 16:18 gpone0b
crw-rw-rw-  1 root      32,   2 Mar 11 16:18 gpone0c
crw-rw-rw-  1 root      32,   3 Mar 11 16:18 gpone0d
crw-rw-rw-  1 root      29,   0 Mar 11 16:11 kbd
crw-------  1 root      16,   0 Mar 11 16:11 klog
crw-r-----  1 root       3,   1 Mar 11 16:11 kmem
srw-rw-rw-  1 root            0 Mar 25 18:02 log
crw-------  1 root       3,   4 Mar 11 16:11 mbio
crw-------  1 root       3,   3 Mar 11 16:11 mbmem
crw-r-----  1 root       3,   0 Mar 11 16:11 mem
crw-rw-rw-  1 root      13,   0 Mar 11 16:11 mouse
crw-------  1 root      37,  40 Mar 11 16:11 nit
crw-rw-rw-  3 root      18,   4 Mar 11 16:12 nrmt0
crw-rw-rw-  3 root      18,   5 Mar 11 16:12 nrmt1
...
crw-rw-rw-  1 root      21,   1 Mar 25 21:49 ptyp1
crw-rw-rw-  1 root      21,   2 Mar 25 14:13 ptyp2
crw-rw-rw-  1 root      21,   3 Mar 25 14:10 ptyp3
...
crw-rw-rw-  1 root      21,  10 Mar 11 16:13 ptypa
crw-rw-rw-  1 root      21,  11 Mar 11 16:13 ptypb
```

Le première lettre indique que c'est un fichier spécial de périphérique et elle décrit son type :

- « c » désigne un périphérique caractère;
- « b » désigne un périphérique bloc.

Viennent ensuite les droits de lecture, d'écriture et d'exécution pour respectivement : le propriétaire, les membres de son groupe, et les autres utilisateurs.

Après ces droits, le système indique le nom du propriétaire du fichier. On remarque que c'est root pour la plupart de ces fichiers.

Le premier chiffre, à la suite du propriétaire, désigne le majeur. Il correspond au type du périphérique. Les terminaux VT100 sont un type de périphérique, les lecteurs de disquettes 360 ko des PC en sont un autre, les dérouleurs de bandes XyLogics 472, un autre encore. Par ailleurs, le majeur est aussi un index dans une table qui permet, pour un appel système donné, de retrouver la fonction de pilotage réelle du périphérique.

Le chiffre suivant est le mineur, il permet de distinguer les périphériques de même type et de les relier à une adresse[26].

On crée un nouveau fichier, correspondant à un périphérique, par la fonction `mknod`. On doit préciser en argument le nom de ce périphérique, par exemple `/dev/perif`, le type `c(aractère)` ou `b(loc)`, le n° de majeur pour retrouver les fonctions de pilotage dans la table et le mineur pour retrouver son adresse. Ceci donne une commande du type :

```
mknod /dev/perif c 2 13
```

4.5.2. Les fonctions d'entrées-sorties du noyau d'Unix

Les fonctions réelles d'entrées-sorties : d'ouverture, de fermeture, de lecture et d'écriture, sont contenues dans deux tables faisant partie du noyau. Ces fonctions sont propres à chaque périphérique. Elles réalisent le pilotage direct du contrôleur du périphérique et elles sont mises en correspondance avec les appels système.

La table *character device switch*[27] permet de piloter les périphériques caractères. Elle fait correspondre, aux fonctions génériques de manipulation de fichiers : `open`, `close`, `read`, `write` et `ioctl`, des fonctions destinées au

---

[26] Pour déterminer exactement le mineur d'un périphérique, reportez-vous, par exemple, à un manuel d'administration Unix.

[27] Table provenant de Bach, p. 333, op. cit.

périphérique qui effectue les entrées-sorties. La table permet leur substitution au moment de l'appel système[28]. Ces fonctions sont indicées par le n° du majeur.

fonctions \ n° majeur	open	close	read	write	ioctl
0	conopen	conclose	conread	conwrite	conioctl
1	dzbopen	dzbclose	dzbread	dzbwrite	dzbioctl
2	syopen	nullclose	syread	sywrite	syioctl
3	nulldev	nulldev	mmread	mmwrite	nodev
4	gdopen	gdclose	gdread	gdwrite	nodev
5	gtopen	gtclose	gtread	gtwrite	nodev

La console est désignée, dans cette table, par le périphérique n° 0. Le noyau l'ouvre effectivement par `conopen`. La mémoire centrale possède le majeur n° 3, la fonction `nulldev` n'a en réalité aucun effet, la mémoire étant toujours ouverte. Sur le HP-9000, ces routines se trouvent dans le fichier : `/etc/conf/conf.c`.

De la même manière, on dispose de la table *block device switch*[29] pour piloter les périphériques blocs :

fonctions \ n° majeur	open	close	strategy

[28] "leur" introduit ici une ambiguïté de langage, il peut être transitif ou intransitif. Il faut comprendre, bien sûr, : "on substitue les fonctions aux appels systèmes."

[29] Table provenant de Bach, p. 333, op. cit.

0	gdopen	gdclose	gdstrategy
1	gtopen	gtclose	gtstrategy

La fonction `strategy` permet de gérer les lectures et les écritures des blocs. Le sens de cette fonction apparaîtra plus clair après la partie consacrée aux fichiers.

4.5.3. Les appels systèmes d'Unix

Les appels système fournissent une interface uniforme pour les opérations d'entrées-sorties, quelque soit le périphérique. Certaines de leurs actions sont du domaine de ce chapitre, d'autres sont du ressort du système de fichiers que nous examinerons dans un chapitre ultérieur.

Les appels systèmes ont accès, par l'intermédiaire du nom du fichier, ou de leur descripteur, une fois ouvert, à un nœud d'information[30], (*i-node*). Ce nœud contient, notamment, le majeur et le mineur du périphérique.

La fonction `int open(char *ref, int mode)` recherchera le nœud d'information du périphérique dont le nom est la chaîne de caractères pointée par `ref` et lui allouera un descripteur. Ultérieurement, le processus exécutant l'appel, ainsi que le noyau, référenceront ce descripteur comme celui d'un autre fichier. Le système substituera ensuite à `open,` une fonction spécifique d'ouverture, indicée par le majeur du fichier. Avec la table que nous avons donnée en exemple, si le majeur en 0, cette fonction sera `conopen.` La fonction spécifique d'ouverture prendra alors le mineur comme paramètre et ouvrira directement le contrôleur.

---

[30] Le chapitre sur les fichiers clarifiera la notion de nœud d'information.

À la différence des fichiers ordinaires, les ouvertures des fichiers spéciaux peuvent être bloquantes. On peut cependant s'affranchir de ces éventuels blocages en introduisant un délai d'attente maximal, grâce, par exemple, à la fonction : `int alarm(int n)`. Cette fonction émettant une interruption `SIGALRM`, `n` secondes après son exécution. Nous donnons un exemple de la mise en œuvre de ce mécanisme, un peu plus bas, dans l'alinéa décrivant la fonction `read`.

`int close(int desc)` ferme la connexion matérielle avec le contrôleur du périphérique et le remet à zéro. (Seulement si aucun autre utilisateur n'a gardé ce périphérique encore ouvert). La fonction libère aussi le descripteur de fichier `desc`.

`int read(int desc, char *buf, int n),` permet de lire dans le fichier (ici le périphérique) de descripteur `desc, n` caractères et de les déplacer dans une zone contiguë en mémoire commençant à l'adresse `buf`[31]. `buf[0]` désignera le 1er octet transféré, `buf[i - 1]`, le ième octet[32]. La fonction `read` rend le nombre de caractères réellement lus.

Dans le cas de terminaux, par défaut, la lecture, comme l'ouverture, est blocante et le transfert des données, des pilotes vers le processus, ne peut s'effectuer qu'après la frappe d'un caractère `newline`, (`return` ou `enter`). Pour une ligne vide, un seul `return`, `read` retournera alors 1. Un 0 correspondra à une fin de fichier (`^D`).

On peut débloquer les lectures en utilisant la fonction `alarm()`. Le programme qui suit attendra pendant 5 secondes l'arrivée de caractères. Passé ce

---

[31] En langage C, ceci se déclare par un tableau de d'octets - de caractères -, par exemple, `char buf[512]`.

[32] Ceci, car en langage C, il y a une correspondance entre le nom du tableau et l'adresse du début du tableau. Par ailleurs, les indices d'un tableau commencent à 0.

délai, la fonction `alarm()` émettra une interruption qui activera la fonction `erreur()`. Le programme se terminera alors en affichant un message d'erreur :

```
#include <signal.h>

void erreur();

main () {
   char table[50];          /* la table à lire */
   int nread;          /* le nombre de car. lus */
   signal(SIGALRM, erreur);
   alarm(5);      /* On arme le compteur */
   nread = read(0, table, 10);
   alarm(0);      /* On désarme le compteur */
   table[nread + 1] = '\0';
   printf("%s", table);
}

void erreur(n)
int n;
{
   printf("Temps dépassé\n");
   exit(1);
}
```

On peut aussi éviter les blocages grâce au positionnement du pilote par la fonction `ioctl`. Le transfert pourra alors se régler suivant deux conditions :

- après un nombre déterminé de caractères frappés au clavier, 2 par exemple ;
- après un temps déterminé d'attente, 5/10e de seconde par exemple.

`int write(int desc, char *buf, int n)` permet d'écrire dans le fichier de descripteur `desc`. L'appel transférera au maximum `n` caractères, contenus dans le tampon pointé par `buf`. La fonction `write` rend le nombre de caractères réellement écrits.

`int ioctl(int desc, int commande, struct termio *arg)` permet de régler les paramètres des périphériques caractères. Elle a pour homologue la commande `stty -a` de l'interprète de commandes. `desc` est le descripteur du périphérique ouvert, `commande` est l'opération de lecture ou d'écriture des paramètres de réglage :

- `TCGETA` pour lire les paramètres,
- `TCSETA` pour les positionner.

Les paramètres sont eux contenus dans la structure : `arg`. La définition de cette structure fait partie de la bibliothèque `termio.h.` Ces principaux champs sont :

- `c_iflag`, qui règle le mode d'entrée, notamment les conversions majuscules-minuscules, les traitements du `newline` et du `return`;

- `c_oflag`, qui est l'homologue en sortie de `c_iflag`;

- `c_cflag`, qui positionne les vitesses de transmission, le contrôle de parité, etc;

- `c_cc[NCC]`, qui donne les correspondances entre des touches et des signaux ou certaines constantes, par exemple, la fin de fichier et `^D`, l'interruption au clavier et `^C`, etc. En mode non-canonique, on peut aussi positionner deux conditions de déblocage de la lecture selon le nombre de caractères rentrés ou bien le temps d'attente;

- `c_line`, permet de positionner certains modes, avec ou sans écho, canonique ou non canonique, etc.

Pour une maîtrise plus complète de l'interface utilisateur des entrées-sorties, on pourra aussi examiner les fonctions, `fcntl` et les bibliothèques `stdio.h` `fcntl.h.` Cette liste n'est pas limitative.

4.5.4. Démarrage de pilotes

Au départ du système, le processus initial se charge de lancer les pilotes de périphériques. Pour chaque console en particulier, un processus « démon », `getty` se chargera de scruter les éventuelles entrées.

Ce processus, lorsqu'il détectera une tentative de connexion, vérifiera le mot de passe de l'utilisateur dans le fichier `/etc/passwd`, puis il sera remplacé par l'interprète de commande en cas de réussite. `getty` se régénérera par l'indication `respawn` au moment de la déconnexion de l'utilisateur (fin de l'interprète).

On trouve les ordres de lancement de divers démons dans le fichier `/etc/inittab`. Par exemple pour la machine ensi 1 de l'ISMRA, les démons des terminaux, correspondant à la console graphique, ainsi qu'au terminal VT100 qui lui est attaché, sont situés à ses deux dernières lignes. (# indique un commentaire).

```
init:2:initdefault:
stty::sysinit:stty 9600 clocal icanon echo opost onlcr ienqak ixon icrnl
ignpar </dev/systty
brc1::bootwait:/etc/bcheckrc </dev/console >/dev/console 2>&1 # fsck, etc.
brc2::bootwait:/etc/brc >/dev/console 2>&1      # boottime commands
link::wait:/bin/sh -c "rm -f /dev/syscon; \
                      ln /dev/systty /dev/syscon" >/dev/console 2>&1
cwrt::bootwait:cat /etc/copyright >/dev/syscon      # legal requirements
meas::bootwait:/usr/contrib/bin/Micreate >/dev/syscon # measurement interface
rc  ::wait:/etc/rc </dev/console >/dev/console 2>&1 # system initialization
powf::powerwait:/etc/powerfail >/dev/console 2>&1   # power fail routines
lp  ::off:nohup sleep 999999999 </dev/lp & stty 9600 </dev/lp
cons:32456:respawn:/etc/getty -h console console    # system console
# xlog:2:respawn:/usr/bin/X11/xlogin                  # X Windows login
co:2:respawn:/etc/getty -h ttyc0 9600               # Serie carte mere
```

4.6. Les « streams »

Lors d'une entrée-sortie, il est parfois nécessaire, avant d'accéder à un dispositif final, de passer par d'autres matériels. Ces matériels nécessitent, bien sûr un système de pilotage intermédiaire. Ceci particulièrement, quand on désire

accéder à un périphérique via un réseau. Ces pilotages multiples peuvent conduire à des implantations *ad hoc*, lourdes et hétéroclites.

D. Ritchie a conçu les « streams » pour décomposer les pilotes de périphériques en couches rationnelles. Les « streams » sont des ensembles de modules de traitement qui pilotent les étapes matérielles respectives. Par exemple un terminal à travers un réseau comprendra un module de pilotage du terminal et un module de pilotage du réseau. Ces « streams » sont actuellement très en vogue pour l'écriture de pilotes de périphériques sous Unix.

Les « streams » sont comparables à un flux bidirectionnel de données passant à travers un empilement de modules commençant par une tête et finissant par le pilote proprement dit. Le module de tête correspond à l'interface utilisateur des appels système. Les modules intermédiaires, qui peuvent ne pas exister, sont constitués d'une paire de files transmettant les données. Une file est descendante, de l'utilisateur vers le périphérique, l'autre est montante. Les commandes entre les modules se font sous la forme de messages.

Chaque module a une fonction bien précise, par exemple, transcoder les caractères ASCII et EBCDIC, ou bien effectuer un filtrage canonique. Enfin, le module de queue réalise le pilotage brut du périphérique.

L'un des intérêts majeurs des « streams » réside dans leur construction dynamique. On peut empiler et dépiler les modules à volonté par une extension de la fonction `ioctl`.

Chapitre 5
Les réseaux informatiques

5.1. Qu'est-ce qu'un réseau?

5.1.1. Introduction

Les réseaux de communications informatiques connaissent un développement accéléré. Ces réseaux ont d'abord été indépendants des systèmes d'exploitation, puis des laboratoires de recherches les ont intégrés dans leur noyau. La plupart des constructeurs ont suivi ce mouvement et les réseaux jouent maintenant un rôle fondamental dans la presque totalité des systèmes d'exploitation commerciaux.

La théorie des réseaux, en tant que telle, est un ensemble assez vaste que nous n'aborderons que dans ses liens directs avec l'informatique; ces liens étant déjà d'une complexité importante[33]. Dans ce cadre, les réseaux combinent des caractéristiques qui leur sont propres, et notamment la transmission de données, à des concepts propres aux systèmes d'exploitation, tels par exemple que la répartition de la mémoire ou du calcul.

On se réfère généralement au terme de réseaux informatiques lorsque les communications entre les ordinateurs sont explicites, par exemple par une

[33] À titre d'exemple, l'inclusion des réseaux dans le système Unix en a doublé la taille. Cette taille a encore été multipliée par deux avec les systèmes de fenêtrage.

commande de transfert d'un fichier, d'une machine vers une autre, ou bien par un courrier envoyé à un autre utilisateur. Ce niveau est le plus simple et le plus ancien.

L'informatique répartie, qui fournit la matière du chapitre suivant, elle, masque, lorsqu'elle concerne les fichiers, la réalité physique des emplacements des disques. La communication de données est alors invisible à l'utilisateur. Ce dernier appréhende son système de fichiers par l'arborescence habituelle sans pouvoir distinguer ce qui local de ce qui est lointain[34].

5.1.2. Pourquoi un réseau

Les réseaux informatiques ont eu comme première fonction de permettre l'accès à des ordinateurs distants par l'intermédiaire de modems. Le programme mis en œuvre consiste alors, pour l'utilisateur à distance, à simuler la connexion d'un terminal local, tout en transmettant les données à travers le réseau. Par les mêmes techniques, on peut accéder à des informations distantes, telles que des bases de données par exemple. La transmission de fichiers ou l'envoi de messages se font par des techniques voisines. Elles évitent les disquettes, les bandes magnétiques ou les lettres par la poste.

Dans leurs versions plus élaborées, les réseaux permettent de partager des ressources telles que des imprimantes ou des disques. Ce partage suit le modèle du client et du serveur. Le serveur étant le logiciel ou le matériel qui fournit le service. On construit ainsi des serveurs d'impression, de disques, de fichiers,... Celui qui utilise le service est appelé le client. Dans le cadre d'une informatique répartie, il y a une ubiquité de ces serveurs et de ces clients, qui représentent virtuellement tous les processus et tous les matériels.

---

[34] Par exemple, combien d'élèves de l'ISMRA savent, en débutant, si les données qu'ils manipulent sont au pied de leur machine, ou dans la salle d'à côté.

5.1.3. Perspective historique

L'extension des réseaux informatiques se place dans la perspective de la réduction des tailles et des coûts des matériels informatiques. À ses débuts, elle n'était pas partagée. Un ordinateur central, disposait des données et des programmes. Ces programmes étaient bien souvent présent en quelques exemplaires seulement à la surface du globe. Les seules opérations de réseau résidaient dans des connexions de terminaux à distance.

Avec les mini-ordinateurs, les réseaux se sont développés, surtout pour diffuser les logiciels et les données lorsqu'ils n'étaient pas envoyés par la poste. Ils s'agissait alors de transfert de fichiers, de messagerie, ... Ce modèle est très répandu dans le monde scientifique et celui des grandes entreprises. Il commence à se diffuser largement dans les petites entreprises et dans le public.

Avec la micro-informatique, il est devenu impossible de maintenir la cohérence de toutes les données et de toutes les versions des logiciels. Chaque utilisateur devant en avoir un exemplaire local (sur son propre disque) à jour. Les données et les programmes communs sont donc implantées, souvent de manière unique, sur un des disques du réseau : le serveur. Par l'intermédiaire d'un « montage » à distance de ce disque serveur, l'utilisateur accède à ces données communes de la même manière que pour un disque local. Le partage est souvent invisible. La plupart des micro-ordinateurs des entreprises, petite ou grande, sont connectés de la sorte.

5.1.4. La constitution d'un réseau

Un réseau est constitué de « nœuds », tels que des ordinateurs, des imprimantes, ... Chaque nœud possède une adresse à laquelle on peut associer un nom. Ces nœuds sont reliés entre eux par un support de transmission. Ce support peut être formé de paires de métal torsadées, de câbles coaxiaux, de fibres optiques, ou bien être l'atmosphère, ... Des signaux émis selon un certain codage,

et regroupés en trames d'un certain nombre d'octets, transitent sur le support. Les signaux peuvent être diffusés en bande de base ou bien modulés.

Les nœuds sont agencés suivant une architecture, une topologie. Cette topologie peut être un bus, une étoile ou bien un anneau. On doit différencier la topologie physique, qui correspond à l'agencement des nœuds, de la topologie logique, qui elle est le mode de circulation des données. Pour TokenRing, l'anneau à jeton d'IBM, la topologie physique est une étoile, la topologie logique est un anneau.

Les nœuds accèdent au support :

- de manière aléatoire, en détectant une porteuse pour la plupart des réseaux Ethernet,
- de manière semirépartie, en se saisissant d'un jeton, pour l'anneau d'IBM ou bien,
- par l'intermédiaire d'une commande centrale et d'une commutation de données dans le réseau téléphonique et dans les réseaux locaux modernes.

Ces nœuds sont rattachés au support par un dispositif matériel de connexion, une carte électronique, qui se charge, en général, de gérer l'accès.

Pour que les données parviennent à leur destinataire, les trames doivent être munies d'informations supplémentaires, telle que leurs adresses de destination et d'expédition, des commandes diverses, un code de détection d'erreur,...

Les trames sont échangées de trois manières :

1. En mode « non connecté » ou datagramme, les données sont expédiées de manière indépendante, « du mieux qu'on peut », en espérant qu'elles arrivent à bon port. On peut comparer ce mode à l'envoi d'un lettre.
2. En mode non connecté avec un acquittement. On peut comparer ce mode à l'envoi d'un lettre avec un accusé de réception.

3. En mode « connecté », l'expéditeur demande d'abord une connexion avec le destinataire. Si ce dernier l'accepte, il retourne un acquittement. Les données sont ensuite envoyées sous la forme d'une séquence ordonnée et elles sont acquittées de manière régulière. Enfin la connexion est rompue à l'initiative de l'une des deux parties.

En fait, seules les méthodes 1 et 3 sont largement utilisées pour la transmission de données. La méthode 2 est utilisée par les RPC du chapitre suivant.

Les réseaux sont locaux, lorsqu'ils sont construits autour d'un support simple, un bus par exemple, ou un anneau. Au delà, ils sont dits étendus. Les réseaux locaux sont souvent privés alors que la plupart des réseaux étendus sont publics. Les réseaux locaux peuvent être interconnectés par l'intermédiaire de réseaux étendus.

Pour assurer leur transport d'une machine à une autre ou d'un réseau à un autre, les données sont fragmentées. Ces données fragmentées, les paquets, sont échangées, elles aussi, avec des informations supplémentaires, en mode connecté ou non connecté.

L'envoi de commandes d'un ordinateur à un autre se fait par un appel de procédure à distance, un paquet de données spéciales. Au besoin les données sont transcodées pour leur assurer une représentation universelle. Ceci sera présenté au chapitre suivant.

Les applications principales actuelles, pour ce qui concerne l'informatique, sont les transfert de fichiers, la messagerie électronique, les systèmes clients/serveur de fichiers et d'écran.

5.2. Les protocoles de communication

5.2.1. Le modèle de la division en couches

Au départ, la communication ne faisait appel qu'à la transmission de données. Plusieurs protocoles ont vu le jour permettant de transférer des fichiers par le réseau téléphonique. Ces protocoles fonctionnent en point à point, avec une connexion préalable des deux parties. Ils vérifient la bonne réception des données – transmises par paquets – et procèdent à des retransmissions si nécessaire. Parmi les protocoles les plus répandus, on trouve X-Modem et Kermit.

Ces protocoles sont insuffisant pour des réseaux informatiques complexes. L'organe de normalisation international, l'ISO, a conçu une architecture en couches qui tente de prendre en compte la complexité des réseaux pour laquelle, il a publié un certain nombre d'avis et de recommandations. Bien que la plupart des applications commerciales n'adoptent pas les normes établies par l'ISO, elles se réfèrent souvent à son modèle de découpage[35].

Les protocoles informatiques de communication se situent aux différents niveaux conceptuels du modèle ISO. On parle parfois de réseaux TCP/IP ou Ethernet ce qui constitue un abus de langage car en fait cela ne décrit que le protocole d'un niveau d'intervention – une ou deux couches – bien spécifique du réseau.

On distingue couramment trois de niveaux principaux, de conceptualisation croissante, qui vont des signaux physiques aux applications logicielles :

- les services de connexion (1, 2); Ethernet est un exemple de protocole intervenant à ces niveaux.
- les services de transport (3, 4 et 5); TCP/IP est un exemple de protocole fournissant ces services.

---

[35] Il faut éviter de considérer les couches de l'ISO comme un modèle absolu. Ce modèle est assez ancien et souvent dépassé. Les transmissions modernes asynchrones, comme ATM, utilisent une autre architecture, par exemple. Il a autant de pertinence que la séparation entre logiciel et matériel. Le modèle ISO est d'ailleurs mort du point de vue commercial.

- les services d'application (6, 7); NFS fournit un service de fichiers intervenant à ces niveaux. Ce service est fondé sur les appels de procédures à distance RPC et la représentation universelle XDR.

Les protocoles cités précédemment constituent une « pile » nécessaire à la réalisation d'une application répartie. Cette pile est souvent associée au nom de TCP/IP et elle est très répandue dans le monde Unix.

N°	Nom	Définition de la fonction fournie	Protocoles dominants sous Unix ou Windows
7	Application	Serveur de disque, courrier, transfert	NFS, SMTP, FTP, Telnet
6	Présentation	Représentation universelle	XDR
5	Session	Appel de procédures à distance	RPC de Sun ou Microsoft
4	Transport	Transport de bout en bout, fiable, ou non	TCP, UDP
3	Réseau	Découpage en paquets, adressage universel	IP
2	Liaison	Trame, accès, adressage local	Ethernet,Token Ring
1	Physique	Connexion, support, signaux	Ethernet, Token Ring

Tableau 7 Les couches réseau de l'OSI.

Il existe d'autres piles. Dans le monde Windows, la plus courante est celle de Novell NetWare. Elle associe IPX (3), éventuellement SPX (4), avec une « émulation » des SMB (7). Ceci pour fixer quelques éléments de vocabulaire. Il

est à noter que les réseaux forme un domaine constitué quasi uniquement de signes cryptiques et que malheureusement, il faut « faire avec ».

Les dialogues, d'un utilisateur à un autre autre, s'établissent aussi entre les couches inférieures correspondantes. L'information, si elle est émise à partir de la couche application, devra traverser toutes les couches du modèle. Chacune des couches de l'expéditeur (resp. du destinataire) ajoutera les données nécessaires à son fonctionnement et établira un échange (en mode connecté ou datagramme) avec la couche correspondante du destinataire (resp. de l'expéditeur). Au niveau le plus bas, il y aura beaucoup plus d'informations échangées, que de données utiles aux utilisateurs. C'est le mécanisme d'encapsulation des données à l'émission et de désencapsulation à la réception.

5.2.2. Un protocole simple : X-Modem

Le protocole X-MODEM permet de transférer des fichiers à travers un réseau de communication. Il a été défini au départ pour des micro-ordinateurs CP/M mais il s'est diffusé bien au-delà. Il est encore très utilisé, notamment pour les communications utilisant les ports séries asynchrones et les réseaux téléphoniques commutés.

L'échange de données se fait sous la forme de trame. Chacune de ces trames devant respecter un format contenant un en-tête, le n° de séquence de la trame (sur un octet), le complément à 255 de ce n° de séquence, les données elles-mêmes, ainsi qu'un code de contrôle (*checksum* sur un octet). La taille des données de la trame est constante et fixée à 128 octets.

SOH	N° BLOC	255- N°	Données (128 octets)	CT

Figure 46 La trame X-Modem.

La transmission des données se fait se fait dans un seul sens à la fois. Les fichiers sont donc découpés en blocs de 128 octets. Le dernier paquet doit

contenir des caractères « espace » de bourrage car le protocole ne prend pas en compte la longueur du champ de données.

Le schéma d'échange du protocole est le suivant :

L'émetteur ouvre la session par une caractère ACK auquel le récepteur répond, si il est d'accord par un NAK.

L'émetteur envoie ensuite ses blocs de données un par un. À la réception de chaque bloc, le récepteur répond par un ACK, si le code de contrôle correspond ou un NAK sinon. L'émetteur retransmet le paquet si il n'a pas reçu d'acquittement au bout d'un délai donné.

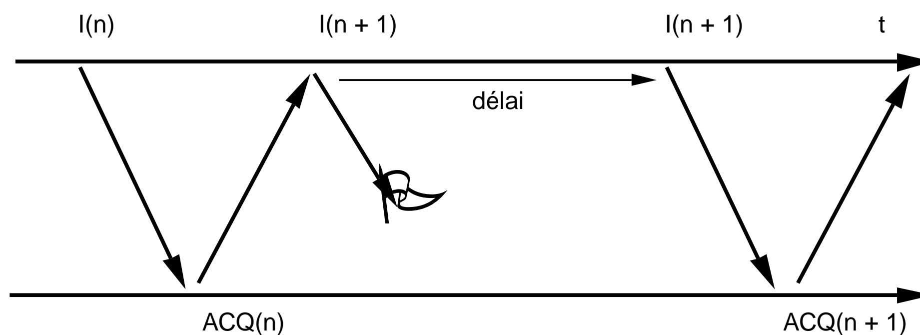


Figure 47 Un échange.

À la fin de la transmission, l'émetteur envoie de EOT si tout est normal ou CAN sinon. À la réception d'un de ces caractères, le récepteur répond respectivement par un EOT ou par un CAN.

Le code de contrôle d'une trame est un « ou exclusif » effectué sur tout les octets de données.

5.2.3. Ethernet

Ethernet est une norme de transmission intervenant aux niveaux 1 et 2 du modèle OSI. Elle permet les échanges de données à l'échelle d'un réseau local.

Le support physique d'Ethernet est un câble coaxial épais ou fin, ou bien une paire torsadée. Le câblage est en bus ou en étoile. Son accès est à détection de porteuse avec détection des collisions : CSMA/CD. Cette méthode d'accès est actuellement supplantés par les techniques de commutation. La fréquence actuelle de transmission est de 10 MHz. Des versions rapides d'Ethernet apparaissent avec une fréquence de 100 MHz.

L'adressage des nœuds Ethernet se fait sur 48 bits. Ces adresses sont, en général, inscrites dans la ROM des cartes de communication, et elles sont attribuées par la société Xerox. Un type décrivant le contenu de la trame est codé sur 2 octets. Un code de contrôle est ajouté à la fin. Il est long de 4 octets. La longueur totale des trames doit être comprise entre 64 et 1518 octets. Les trames sont précédées d'un préambule de 8 octets de synchronisation.

L'échange de trames entre les parties se fait par des datagrammes dont le format est :

Source	Dest	Type	Données	CRC

Figure 48 La trame Ethernet.

5.2.4. Le protocole d'interconnexion IP

Internet Protocol (IP) permet les communications d'un réseau à un autre avec une échelle qui peut être mondiale. Il réalise le découpage des données en paquets et les expédie sous la forme de datagrammes lui aussi.

Par ailleurs, pour passer d'un réseau à un autre, on a besoin d'un adressage unique et universel pour différencier chaque machine. Le protocole IP gère cet adressage international. En France, c'est l'Inria qui distribue les adresses IP.

Les adresses Internet permettent de numéroter un réseau. Elles ont une taille de 4 octets. La répartition entre la taille de l'adresse du réseau et celle de la machine hôte se fait sous quatre formes correspondant aux classes A, B, C et D[36] :

Classe A :

net (1 octet)	host (3 octets)
0xxxxxxx	yyyyyyyy.zzzzzzzz.tttttttt

Classe B :

net (2 octets)	host (2 octets)
10xxxxxx.yyyyyyyy	zzzzzzzz.tttttttt

Classe C :

net (3 octets)	host (1 octet)
110xxxxx.yyyyyyyy.zzzzzzzz	tttttttt

Classe D :

Message de diffusion à un groupe (multicast)
1110xxxx.yyyyyyyy.zzzzzzzz.tttttttt

Figure 49 Les classes IP.

36

Les trois premières formes sont les plus répandues. L'ENSI de Caen dispose d'un réseau d'enseignement de classe C dont l'adresse est : `193.49.200.tttt`. Dans chacune des trois premiers type d'adressage, on peut désigner le réseau lui-même en mettant à 0 l'adresse machine ou toutes les machines, en mettant à 1 cette adresse. On pourra ainsi diffuser des messages à toutes les machines d'un réseau (broadcast). Pour l'ENSI de Caen, l'adresse du réseau et donc `193.49.200.0` et l'adresse de toutes les machines est : `193.49.200.255`. Enfin, il y a une adresse de bouclage pour les communications internes d'une machine : `127.0.0.1`.

La correspondance entre les adresses Ethernet et Internet, sur un réseau local, se fait grâce à l'algorithme *Address Resolution Protocol* (ARP).

Le protocole Internet rajoute un en-tête aux paquet de la forme suivante :

<table>
<tr><td>0</td><td>4</td><td>8</td><td>16</td><td>19   24</td></tr>
<tr><td>Version</td><td>IHL</td><td>Service</td><td colspan="2">Longueur totale</td></tr>
<tr><td colspan="3">n° d'identification d'un même fragment</td><td>Flags</td><td>Décalage de fragmentation</td></tr>
<tr><td colspan="2">Durée de vie</td><td>Protocole</td><td colspan="2">Contrôle d'en-tête</td></tr>
<tr><td colspan="5">Adresse d'origine</td></tr>
<tr><td colspan="5">Adresse de destination</td></tr>
<tr><td colspan="5">Options</td></tr>
</table>

Figure 50 L'en-tête IP.

À l'intérieur de l'en-tête, on peut noter les significations des champs suivants :

- Version est le n° de version du protocole, 4 ou 5 actuellement, 6 à venir.
- Internet Header Length (IHL) est la longueur de l'en-tête Internet.

- Service n'est pas implanté sur la plupart des passerelles.
- Les informations de fragmentation permettent de coordonner la suite des paquets. Cette fragmentation est parfois nécessaire pour tenir compte des limites des protocoles sous jacents: 1500 octets pour Ethernet, 128 octets pour certains protocoles. Elle comprennent un identificateur, un indicateurs et un décalage dans le cas ou il y a fragmentation.
    - l'identificateur est un numéro unique
    - les indicateurs qui indique si on peut fragmenter ou non (DT). Ils donnent des détails sur la fragmentation, avec un bit MF (more flag),
    - le décalage donne la position du fragment dans la suite des paquets.
- Les adresses sont sur 4 octets. Elles auront 16 octets dans la version IPv6.

Pour la transmission des entiers Internet adopte une norme big endian : les octets de poids fort sont transmis en premier. Il faut faire attention à ce que cet ordre n'est pas le même sur toutes les machines et notamment sur les processeurs Intel.

5.2.5. Le protocole de transport TCP/UDP

Entre deux ordinateurs, la communication passe par des ports d'entrées-sorties. Les systèmes d'exploitation sous-jacents étant multi-tâches, chaque machine peut gérer plusieurs ports au même moment.

Il existe deux modes de liaison entre ces ports, soit le transfert par datagramme, soit le transfert fiable. La norme fournit UDP un service datagramme, alors que TCP réalise un service de transport fiable.

Le protocole TCP permet d'ouvrir et de terminer une communication. Les paquets sont transmis et en retour on obtient des accusés de réception. L'envoi de paquets prend en fait un peu d'avance sur les acquittement et déroulement de

l'échange fait appel à une fenêtre glissante. Les paquet comportent deux compteurs correspondant aux octets transmis (Séquence) et aux octets acquittés (Acknowledge).

Les paquets TCP comprennent un en-tête du type suivant :

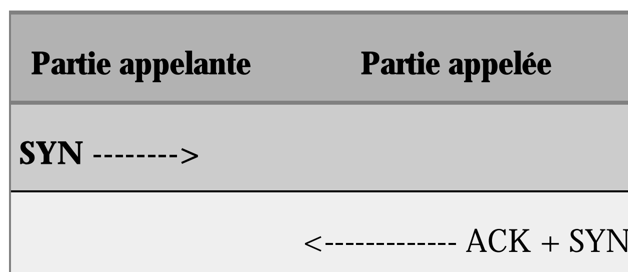
0 16

<table>
<tr><td colspan="3">Source Port</td><td colspan="2">Destination Port</td></tr>
<tr><td colspan="5">Sequence N°</td></tr>
<tr><td colspan="5">Acknowledgment N°</td></tr>
<tr><td>Data Offset
(Taille en-tête)</td><td>Reserved</td><td>Code</td><td colspan="2">Window</td></tr>
<tr><td colspan="3">Checksum</td><td colspan="2">Urgent pointer</td></tr>
<tr><td colspan="4">TCP options</td><td>Padding</td></tr>
<tr><td colspan="5">Données 64 Ko</td></tr>
</table>

Figure 51 La trame TCP.

Le type des messages est contenu dans le champ Code en positionnant un bit : URG, ACK, PSH, RST, SYN, et FIN.

La connexion est un échange à trois temps :

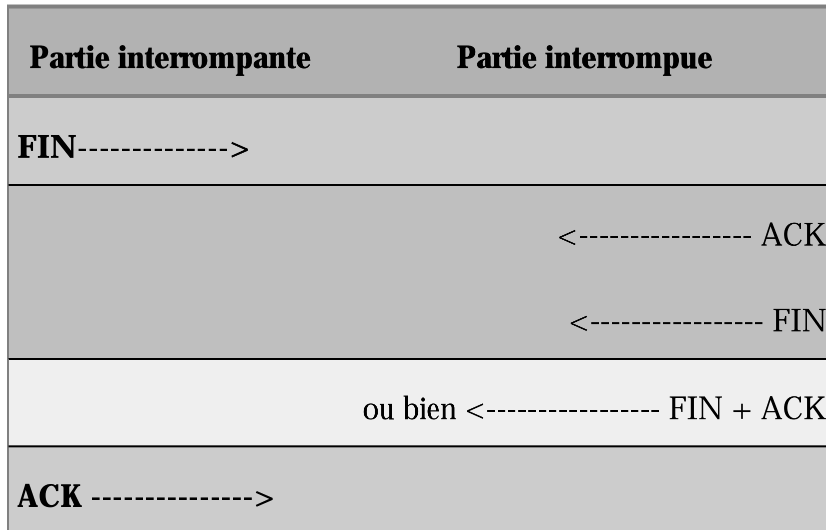


Partie appelante	Partie appelée
SYN -------->	
	<------------- ACK + SYN

ACK -------->

Figure 52 La connexion TCP.

La déconnexion est un échange à trois temps :



Partie interrompante	**Partie interrompue**
FIN-------------->	
	<----------------- ACK <----------------- FIN
	ou bien <----------------- FIN + ACK
ACK --------------->	

Figure 53 La déconnexion TCP.

La conception du protocole de transmission TCP utilise la notion d'automate. Il peut se modéliser par le dessin suivant :

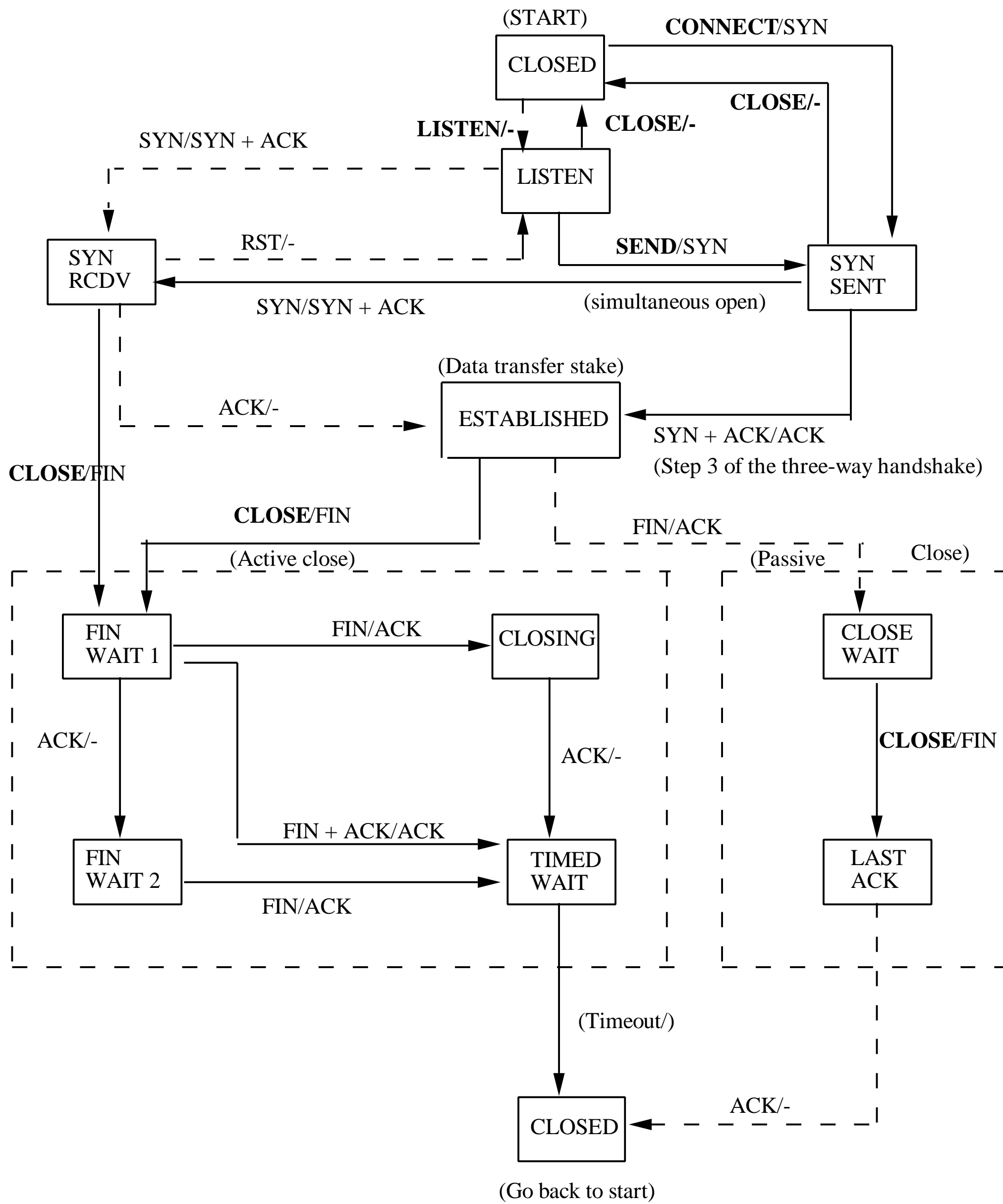


Figure 54 L'automate d'états de TCP (d'après Tanenbaum).

Le protocole UDP est un simple appendice de IP : il rajoute un n° de port à ce dernier protocole. C'est ce protocole qui est en général utilisé pour construire les applications réparties.

0 16

Source Port	Destination Port
Length	Checksum

Figure 55 L'en-tête UDP.

Les programmes TCP/UDP doivent s'adresser à des ports fournissant des services dont les n° sont connus. En revanche, les ports utilisés par le clients sont en général attribués par le système. La commande `netstat` permet de visualiser ces attributions.

5.3. L'architecture d'un système client-serveur

Au-dessus des protocoles de transmission, on peut construire des applications réparties reposant sur un (ou plusieurs) serveur et des clients. Les serveurs (machines) fournissent un service (programme), par exemple l'heure, le transfert de fichiers, la connexion, l'écriture sur un disque... à des programme clients (sur les machines clientes) qui utilisent ces services.

Les deux appels de base mis en jeu sont d'une part les demandes du client et d'autre part les réponses du serveur. Dans un protocole comme TCP/IP ou UDP/IP, le client émet sa demande vers le serveur en la munissant de la bonne adresse de machine et de la bonne adresse du service[37] : le port. Le serveur à l'écoute de ce port (en lecture du port) détecte la demande et fournit la réponse munie de l'adresse de la machine cliente et de son port de demande.

[37]Une machine peut héberger plusieurs services. Il est donc nécessaire de la numéroter.

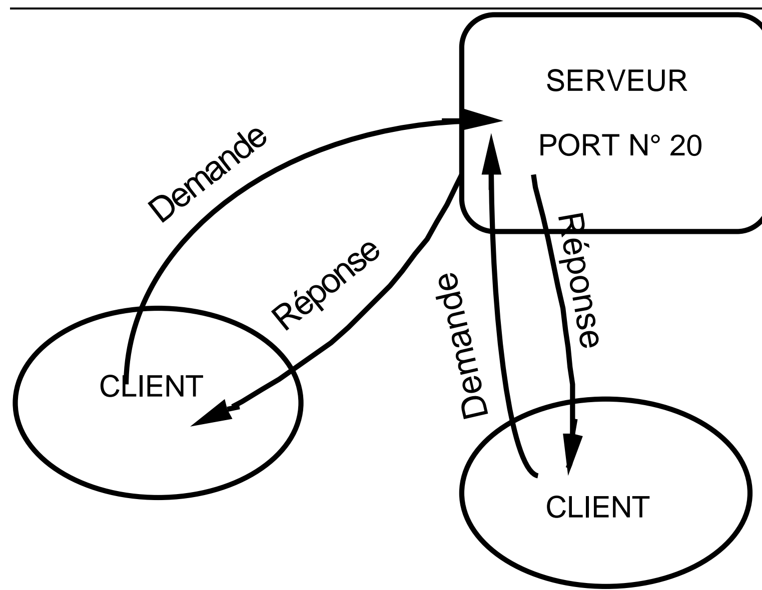


Figure 56 L'architecture d'un système client-serveur.

La boucle fondamentale d'un serveur de création, lecture, écriture de fichier est la suivante :

```
while (1) {
   receive(source, &code);
   switch(code) {
        case CREATE:   a = creat(…;   break;
        case READ:     a = read(…     break;
        case WRITE:    a = write(…    break;
        default;
   }
   send(dest, a);
}
```

5.4. Communiquer par les réseaux publics

5.4.1. Le réseau téléphonique

(Non disponible)

Le protocoles de modems, les commandes Hayes, les protocoles de communication, UUCP, MNP4 et MNP5.

5.4.2. Internet et ATM

(Non disponible)

5.5. Les interconnexions

Les niveaux d'interconnexion

L'interconnexion des réseaux peut avoir lieu à différents niveaux de la hiérarchie OSI. Cette distiction classique est cependant de moins en moins valide, notamment depuis l'arrivée des commutateurs. Classiquement on a :

- les répéteurs (1) régénèrent les signaux;
- les ponts (2) passent des données d'une branche de réseau à une autre. Ils fonctionnent souvent par auto-apprentissage et filtrage. On trouve maintenant à leur place des commutateurs de réseau qui généralisent les techniques des ponts à plusieurs branches : 4, 8, 32, etc.
- les routeurs (3) peuvent orienter les données en fonctions de tables préalablement définies. Ils utilisent de plus en plus des techniques de commutation.
- Les passerelles (7) traduisent les protocoles.

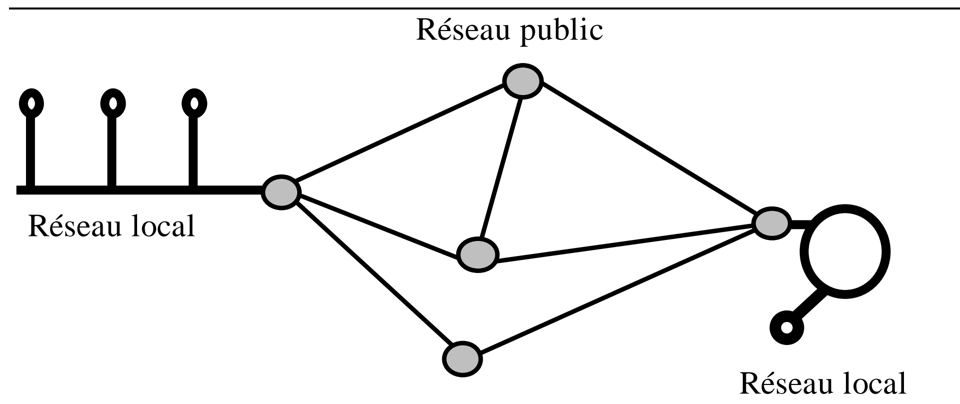


Figure 57 Interconnexions.

Les principes et les normes de routage

Non disponible en 1999

5.6. Les interfaces de programmation Berkeley

5.6.1. Les fichiers du système

- les ordinateurs du réseau sont dans le fichier : `/etc/hosts`
- les services connus du système sont dans : `/etc/services`
- les protocoles connus du système sont dans : `/etc/protocols`
- les réseaux connus du système sont dans : `/etc/networks`

Un extrait des services de `/etc/services` :

```
tcpmux          1/tcp
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
systat          11/tcp          users
daytime         13/tcp
```

```
daytime         13/udp
netstat         15/tcp
chargen         19/tcp          ttytst source
chargen         19/udp          ttytst source
ftp-data        20/tcp
ftp             21/tcp
telnet          23/tcp
smtp            25/tcp          mail
time            37/tcp          timserver
time            37/udp          timserver
name            42/udp          nameserver
```

Le fichier de protocoles `/etc/protocols`

```
#
# Internet (IP) protocols
#
ip              0       IP              # internet protocol, pseudo protocol
number
icmp            1       ICMP            # internet control message protocol
ggp             3       GGP             # gateway-gateway protocol
tcp             6       TCP             # transmission control protocol
egp             8       EGP             # exterior gateway protocol
pup             12      PUP             # PARC universal packet protocol
udp             17      UDP             # user datagram protocol
hmp             20      HMP             # host monitoring protocol
xns-idp         22      XNS-IDP         # Xerox NS IDP
rdp             27      RDP             # "reliable datagram" protocol

#
# Internet (IPv6) extension headers
#
ipv6            41      IPv6            # IPv6 in IP encapsulation
ipv6-route      43      IPv6-Route      # Routing header for IPv6
ipv6-frag       44      IPv6-Frag       # Fragment header for IPv6
esp             50      ESP             # Encap Security Payload for IPv6
ah              51      AH              # Authentication Header for IPv6
ipv6-icmp       58      IPv6-ICMP       # IPv6 internet control message
protocol
ipv6-nonxt      59      IPv6-NoNxt      # No next header extension header for
IP v6
ipv6-opts       60      IPv6-Opts       # Destination Options for IPv6
```

Un service de Pages Jaunes, lorsqu'il est installé, s'efforce de rendre unique ces fichiers sur le réseau. On accède alors à leur contenu, non pas par la commande

more, par exemple, more /etc/hosts, mais par ypcat, par exemple, ypcat hosts. Les fonctions du système qui peuvent manipuler ces fichiers ne devraient néanmoins pas être modifiées.

5.6.2. Les sockets

Les interfaces de programmation permettent d'implanter des transactions bidirectionnelles entre un fournisseur de service: le **serveur** et un usager de ce service, le **client**.

Le serveur fournira son service sur le port de communication d'une machine. Certains ports sont réservés pour des services bien connus, tel que ftp. D'autres sont libres pour les les programmeurs – et les étudiants – de systèmes. Les ports supérieurs à IPPORT_RESERVED sont normalement libres.

Le client enverra sa demande au serveur, à l'adresse de la machine et au port du service désiré.

Le système Unix fournit une interface de programmation permettant Des transactions sous la norme TCP/IP, en mode datagramme ou connecté. D'autres versions ont utilisÈ la norme IP/TP4 de l'OSI. Ce sont les interfaces TLI. Ces dernières étant en voie de disparition, nous ne considérerons que la norme TCP/IP.

Les « sockets » sont les prises de communication des transactions. Elles ressemblent, du point de vue de l'utilisateur, à des entrées-sorties vers un fichier. À chaque « socket » correspond donc un descripteur entier qui servira de référence à ces entrées-sorties. L'appel de la fonction socket permet l'obtention de ce descripteur :

```
int socket(int family, int type, int protocol);
```

La famille, pour Internet, est AF_INET, le type est SOCK_DGRAM pour le mode datagramme, ou SOCK_STREAM, pour le mode connecté, le protocole est

généralement mis à `0`. Ceci indique qu'à l'ouverture de la socket, le système choisira le protocole approprié au mode. L'appel rend `-1` en cas d'erreur.

Contrairement aux descripteurs de fichiers, ces sockets ne sont pas liées à un dispositif physique. On doit leur associer, leur attacher dans le jargon en usage, une adresse, c'est à dire un numéro de machine Internet et un numéro de port TCP. L'adresse, lorsqu'elle est complète, comprend les éléments suivants : `<protocole de communication, adresse machine locale, port local, adresse machine distante, port distant>`. Distant ici est un abus de langage, car un système client/serveur peut fonctionner sur une même machine.

L'attachement complet est un processus assez complexe qui peut se réaliser de plusieurs façons. On peut par exemple attacher la partie locale par la fonction `bind()` et la partie distante par des fonctions comme `connect()` ou `accept()`. La fonction `socket` fournit simplement le protocole de communication. Il nous reste à détailler les structures de données nécessaires aux autres éléments.

On manipule les ensembles <adresse, port> lointains ou locaux par une structure : « adresses[38] de sockets » s'inscrivant dans une forme générique :

```
struct sockaddr {
        short     sa_family;
        char      sa_data[14];
};
```

Pour le domaine Internet, les sockets ont pour adresses :

```
struct sockaddr_in {
        short     sin_family;
        short     sin_port;      /* port TCP */
```

[38] Répétons le « adresse » ici se réfère à la partie soit locale, soit lointaine des deux entités de la communication.

```
        struct in_addr sin_addr;
        char      sin_data[8];
};
```

L'adresse Internet prend la forme :

```
struct in_addr {
        u_long    s_addr;
};
```

On peut obtenir, grâce au fichier `/etc/hosts`, l'adresse Internet de chaque machine du réseau à partir de son nom. L'appel de la fonction `struct hostent *gethostbyname(char *name)` fournit ce service dans un programme. Elle rend un pointeur sur une structure de type `hostent` :

```
struct hostent {
        char *h_name;
        char **h_alias;
        int h_addrtype;
        int h_length;  /* taille de l'adresse */
        char **h_addr_list; /*liste des adresses */
};
```

La variable globale `h_errno` est positionnée par le système en cas d'erreur.

De la même manière, la fonction `struct servent *getservbyname(char *service)` analyse le fichier `/etc/services`.

Les fichiers à inclure pour la compilation d'un programme sont :

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
```

Les appels système, une fois la socket créée, doivent réaliser l'attachement de cette socket à une adresse ou à un port, avant de pouvoir échanger des messages à travers elle.

L'attachement de la partie locale se réalise par :

```
int bind(int socket, (struct sockaddr *) sockaddr,
                int sockaddrelen)
```

`bind()` n'est nécessaire que pour attacher une socket à un port bien particulier sans pour autant réaliser d'envoi. Ceci pour se mettre en écoute, par exemple.

En revanche, il est inutile, notamment dans le cas d'un serveur, à l'instant de l'attachement, de préciser l'adresse locale. On lui permet ainsi d'être lancé sur n'importe quel machine. On remplit alors le champ `sin_addr.s_addr` de l'adresse par `INADDR_ANY`.

`bind()` rend -1 en cas d'erreur.

Pour les clients, on doit bien sûr remplir le champ du port lointain demandé ainsi que l'adresse de la machine fournissant le service. On peut remplir ce champ adresse: `sin_addr.s_addr` à partir du champ: `h_addr_list[0]` de la structure `hostent`. On réalise la copie par la fonction `memcpy()`. On peut aussi convertir une notation pointée et affecter directement le membre adresse Internet par :

```
unsigned long inet_addr(char *notation_pointée);
```

Les conversions entre adresses réseau et machines se font par les fonctions `ntohl()`, `ntohs()`, `htons()` et `htonl()`.

5.6.3. Le mode datagramme

Dans le cas de datagrammes, l'envoie de message se fait par les fonctions :

```
int sendto(int socket, char *message, int length,
int flags, (struct sockaddr *) dest, int destlength);
```

`sendto` rend le nombre de caractères envoyés.

L'émission se fait par le port qui a été attaché au préalable à la socket. Si cela n'a pas été fait, le système attribuera un port libre. L'attachement sera alors réalisé dynamiquement et de manière complète (pour la partie locale et lointaine) grâce à la fourniture de `(struct sockaddr *) dest`. Il sera maintenu ultérieurement.

La réception se fait par :

```
int recvfrom(int socket, char *message, int length,
     int flags, (struct sockaddr *) from, int
                   *fromlength);
```

`recvfrom` rend le nombre de caractères reçus.

Pour d'obscures raisons, on doit donner une valeur initiale à `fromlength`. En général, on prend `sizeof(struct sockaddr)`.

Pour ne pas bloquer le client on doit mettre en place des dispositifs de temporisation. D'autres schémas sont possibles. Les réponses complètes dans `man recv` ou `getsockopt`.

5.6.4. Évolution de l'attachement des adresses

Partie Serveur

1. Avant l'échange

Serveur		
Parties	locale	distante
Port TCP UDP	Port serveur	?
Adresse IP	INADDR_ANY	?

L'attachement du port local est réalisé par `bind()`. On ne précise pas, par contre, d'adresse locale, et on indique `INADDR_ANY`, car le processus doit pouvoir fonctionner sur n'importe quelle machine. Ce champ sera automatiquement rempli au cours des échanges.

2. Après une réception

Serveur		
Parties	locale	distante
Port TCP UDP	Port serveur	Port client
Adresse IP	Adresse serveur	Adresse client

La réception du datagramme permet de compléter les champs manquants. Ces champs sont récupérés dans le programme par la fonction `recvfrom()`.

Partie Client

1. Avant l'échange

Client		
Parties	locale	distante
Port TCP UDP	?	Port serveur
Adresse IP	?	Adresse serveur

Ici, on doit inclure dans la structure de données, l'adresse et le port de la machine distante – du serveur –.

2. À la transmission

Client		
Parties	locale	distante
Port TCP UDP	Port client	Port serveur
Adresse IP	Adresse client	Adresse serveur

L'appel de la fonction `sendto()` complète, en interne, la partie locale de l'association et permet la fabrication du datagramme. Les fonctions d'écriture et de lecture ultérieures pourront utiliser les

adresses ainsi créés.

5.6.5. Le mode connecté

Dans le cas d'un échange par transport fiable, le serveur doit préciser combien il accepte de connexions en attente par :

```
int listen(int socket, int nbconnex);
```

L'acceptation des connexions, de la part de ce serveur, se fait par la fonction :

```
int accept(int socket, (struct sockaddr *) from,
           int *fromlength);
```

qui renvoie un descripteur analogue à celui d'un fichier. On utilise ce descripteur de la même manière pour les opérations de lecture et d'écriture. De même que pour le mode non-connecté, `fromlength` doit être initialisé.

Le serveur traite chacune des connexions en créant un processus à l'aide d'un `fork()`. Ce processus se termine avec un `exit()` en fin de connexion.

Du côté du client, les connexions sont réalisées par la fonction :

```
int connect(int socket, (struct sockaddr *) dest,
            int destlength);
```

`connect` renvoie -1 en cas d'erreur sinon 0.

Les échanges de données se font par les fonctions :

`int read(int desc, char *tampon, int nbcar);` et

`int write(int desc, char *tampon, int nbcar);`

Ici, il n'est plus nécessaire de préciser les adresses d'expédition, car les attachements ont été réalisés de manière complète par les fonctions de connexion.

Les fermetures de connexions se font par :

```
int close(int desc);
```

5.6.6. Schéma de programmes en mode connecté

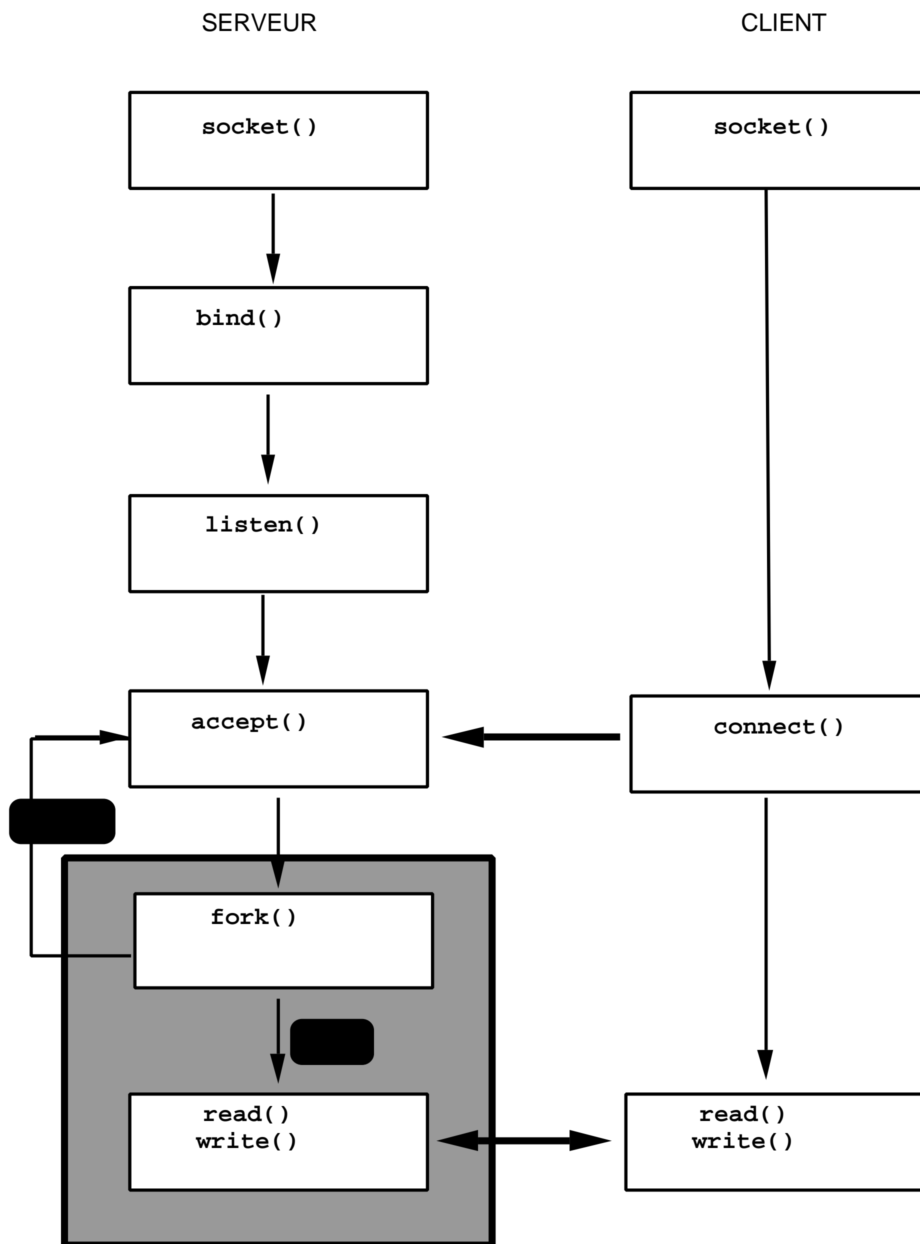


Figure 58 Résumé des appels systèmes pour les sockets.

5.7 Les interfaces de programmation de Java

5.7.1 Vue d'ensemble

Le langage Java permet une intégration facile des programmes au réseau Internet. Alors que l'interface Berkeley est souvent anachronique, les appels Java sont plus simples et mieux conçus. Comme Berkeley, Java comporte les transmissions en mode connecté et en mode datagramme. Nous n'examinerons ici que les échanges connectés.

Dans le mode connecté, le client se connectera au serveur qui établira pour lui une connexion particulière. Le serveur créera pour ceci un fil d'exécution qui jouera un rôle semblable à celui d'un processus sous Unix. Dans le cas du mode non connecté, le principe est quasiment le même qu'avec les appels Berkeley.

Les appels au réseau de Java sont intégrés au paquetage `java.net`. Ils comprennent la classe `Socket`, pour le client, et `ServerSocket` pour le serveur.

5.7.2 Un exemple de client et de serveur en mode connecté

Le client

On se connecte à un service par le constructeur de `Socket` :

```
try {
      Socket maSoquette = new Socket("www.ensicaen.ismra.fr", 2001);
} catch (UnknownHostException e) {
      // pas d'« host »
} catch (IOException e) {
```

```
    // erreur lors de la connexion
}
```

et ensuite on redirige les entrées-sorties sur des flux. Avec les flux de base, ça donne :

```
InputStream in = maSoquette.getInputStream();

OutputStream out = maSoquette.getOutputStream();
```

On écrit et on lit alors dans la soquette avec les fonctions `read()` et `write()` :

```
Byte cara = in.read();

out.write(100);
```

On peut utiliser d'autres flux d'entrées-sorties plus élaborés comme `InputStreamReader()` et `PrintWriter()`.

Le serveur

On lance un serveur et le met en écoute par le constructeur `ServerSocket`. On accepte les clients par `accept()` et on ferme le serveur principal et les serveurs dérivés par `close()`. Chacun des clients est dérivé de `Socket` :

```
try {

    // On lance le serveur au port 2001

    ServerSocket serveur = new ServerSocket(2001);

    while (true) {

        // On accepte les clients

        Socket client = serveur.accept();
```

```
            // dans le code qui suit
            // on traite le client
            ...
      }
} catch (IOException e) {}
serveur.close();
```

Avec ce code, le serveur ne peut traiter qu'un seul client. Pour gérer toutes les connexions en parallèle, on devra créer un nouveau fil d'exécution à chaque arrivée d'un client. Pour ceci, on associera les connexions à des objets issus d'une classe dérivée de `Thread`.

```
ConnexionServeur extends Thread {
      Socket client;
      ConnexionServeur(Socket client) {
            this.client = client;
            // On doit aussi rediriger la soquette vers
            // des descripteurs d'entrée-sortie
            // getInputStream et getOutputStream
      }
      public void run() {
            //On traite la connexion ici
      ...
            client.close();
```

```
    }
}
```

Et on modifie le serveur pour démarrer le fil d'exécution chaque fois qu'il accepte une nouvelle connexion :

```
    while (true) {
        Socket client = serveur.accept();
        new ConnexionServeur(client).start();
    }
```

De même que pour les clients, le serveur créé devra rediriger les flux s'il a besoin de les traiter, par exemple par :

```
InputStream in = client.getInputStream();
OutputStream out = client.getOutputStream();
```

On écrit et on lit avec les fonctions `read()` et `write()` :

```
Byte cara = in.read();
out.write(100);
```

5.7.3 Les classes de Java pour le réseau

Le client `Socket`

Constructeurs :

- `protected Socket()`
- `protected Socket(SocketImpl impl)`

- `public Socket(String host, int port)`
- `public Socket(InetAddress address, int port)`
- `public Socket(String host, int port, InetAddress localAddr, int localPort)`
- `public Socket(InetAddress address, int port, InetAddress localAddr, int localPort)`
- `public Socket(String host, int port, boolean stream)`
- `public Socket(InetAddress host, int port, boolean stream)`

Méthodes :

- `public InetAddress getInetAddress()`
- `public InetAddress getLocalAddress()`
- `public int getPort()`
- `public int getLocalPort()`
- `public InputStream getInputStream()`
- `public OutputStream getOutputStream()`
- `public synchronized void close()`
- `public String toString()`
- `public static synchronized void setSocketImplFactory(SocketImplFactory fac)`

Le serveur `ServerSocket`

Constructeurs :

- `public ServerSocket(int port)`
- `public ServerSocket(int port, int backlog)`
- `public ServerSocket(int port,int backlog, InetAddress bindAddr)`

Méthodes :

- `public InetAddress getInetAddress()`
- `public int getLocalPort()`
- `public Socket accept()`
- `protected final void implAccept(Socket s)`
- `public void close()`
- `public synchronized void setSoTimeout(int timeout)`
- `public synchronized int getSoTimeout()`
- `public String toString()`
- `public static synchronized void setSocketFactory(SocketImplFactory fac)`

Chapitre 6
Les systèmes répartis

6.1 Introduction

L'informatique répartie s'oppose à la fois à l'informatique centralisée, celle des gros ordinateurs, et à l'informatique individuelle, celle des micro-ordinateurs. Elle pallie certains désavantages de cette dernière par :

- le partage des données grâce à un accès individuel, en lecture, par le réseau à des fichiers communs situés sur un disque quelconque ainsi que par le transfert de fichiers d'un disque à un autre.
- le partage des applications. Par exemple pour l'exploitation individuelle, par le réseau, d'un seul logiciel de base de données sur le disque d'une des machines connectées.
- le partage du travail par l'exécution d'un programme dans la mémoire d'une autre machine.
- le partage des ressources : chaque utilisateur connecté peut utiliser une même imprimante.
- les communications : envoi par le réseau de courrier dans une boîte aux lettres électronique à un ou plusieurs utilisateurs connectés. Accès par le réseau téléphonique à des services d'informations : annuaires, banques de données, etc.

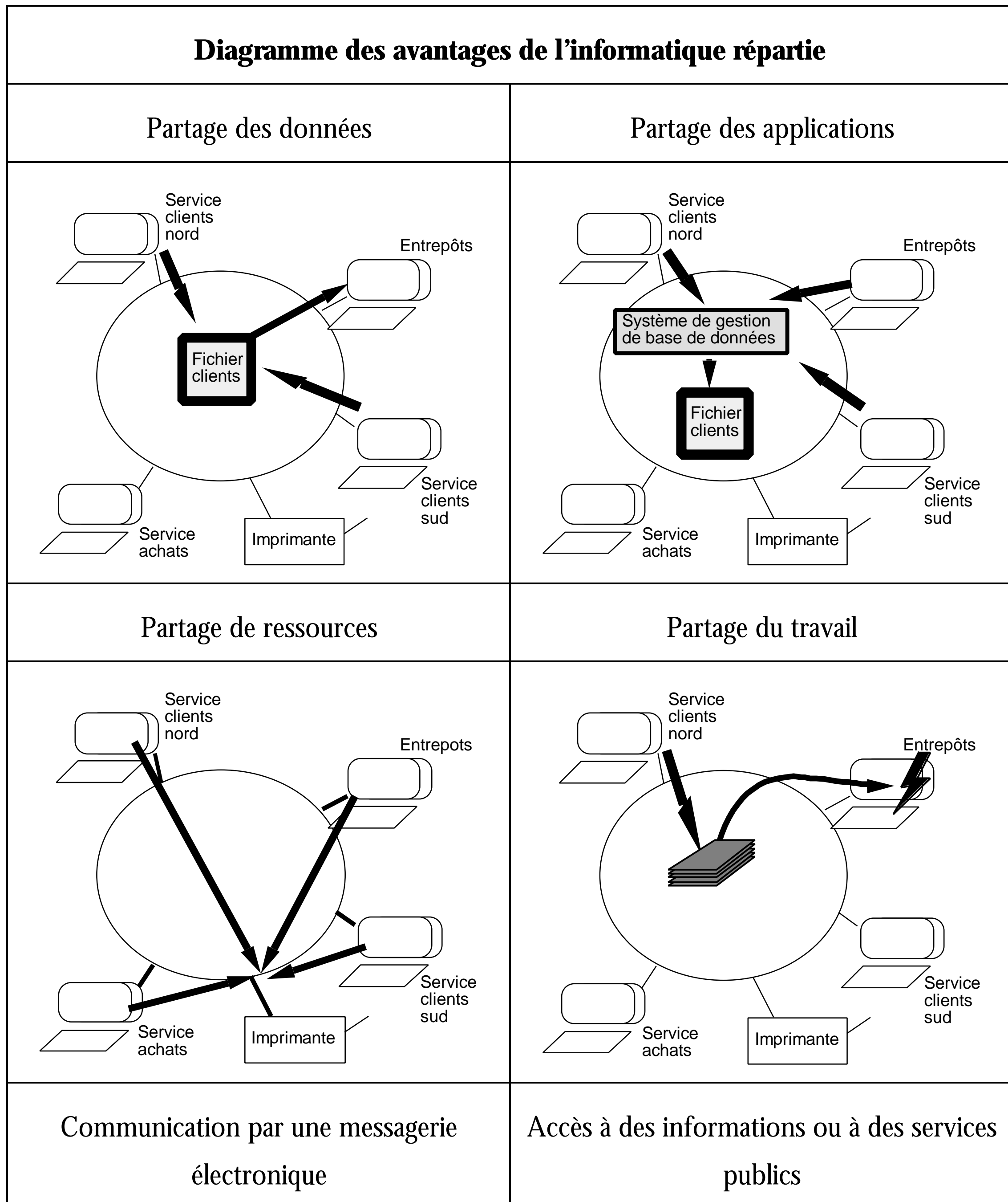


Diagramme des avantages de l'informatique répartie
Partage des données
Service clients nord
Entrepôts
Fichier clients
Service clients sud
Service achats
Imprimante
Partage des applications
Service clients nord
Entrepôts
Système de gestion de base de données
Fichier clients
Service clients sud
Service achats
Imprimante
Partage de ressources
Service clients nord
Entrepots
Service clients sud
Service achats
Imprimante
Partage du travail
Service clients nord
Entrepôts
Service clients sud
Service achats
Imprimante
Communication par une messagerie électronique
Accès à des informations ou à des services publics

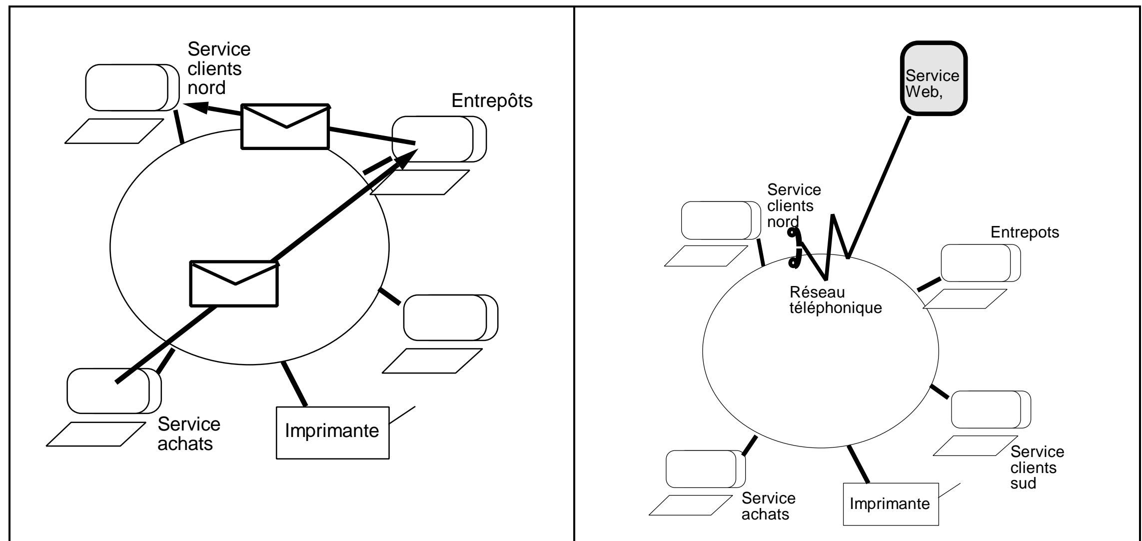


Figure 59 Les avantages d'une informatique répartie

6.1.1 Pourquoi une informatique répartie

L'informatique répartie se répand aux dépens de l'informatique centralisée pour les raisons suivantes :

- elle parfois moins chère, dans la mesure où elle utilise des circuits standards;
- elle est « ouverte » grâce à des protocoles communs;
- elle peut être plus fiable car les pannes d'une machine n'affectent pas les autres utilisateurs;
- elle est à géométrie variable car on peut ajouter des machines ou en enlever facilement (*scalable*)

Elle a aussi certains points faibles qui restent en suspens :

- plusieurs problèmes fondamentaux des systèmes d'exploitation de réseaux ne sont encore totalement résolus;
- beaucoup de logiciels d'application ne sont pas encore prêts;

- la fragilité du système est reportée sur le réseau. Il en devient le tendon d'Achille;

- la sécurité devient un problème fondamental.

L'informatique répartie a pour objectif d'être « transparente » à l'utilisateur et de permettre sa mobilité. Les machines sont banales et un utilisateur n'a pas à connaître la localisation précise des ressources. L'informatique répartie, dans ses applications commerciales, se résume souvent aux serveurs de fichiers. C'est eux qui eux qui ont « popularisé » ce concept.

6.1.2 Les applications de communication

Avant l'informatique répartie, certaines applications de communication se sont répandues. Elles sont relativement plus anciennes et sont fondées directement sur le modèle client-serveur. Elles mettent en œuvre – explicitement du point de vue de l'utilisateur – deux machines. Il existe des applications fondées sur TCP/IP :

- Telnet, « émulation » de terminal

- Ftp, transfert de fichiers

- SMTP, messagerie (le `mail` d'Unix).

Il existe aussi des commandes incluses dans certains systèmes d'exploitation, par exemple les "*remote*" commandes de Unix BSD[39] :

- rlogin machine (connexion à distance)

- rsh machine commande (exécution d'une commande à distance).

---

[39] Ces commandes sont maintenant disponibles sur beaucoup de systèmes Unix.

- rcp machine1:fichier1 machine2:fichier2 (copie d'un fichier d'une machine à une autre).

6.2 La construction d'une informatique répartie

6.2.1 Les appels de procédures à distance (Remote Procedure Calls)

L'informatique répartie est construite autour d'appels de procédures à distance pour pallier les déficiences du modèle client-serveur. En effet, celui-ci fonctionne sur le concept explicite d'entrées-sorties par les opérations `send` et `receive`. Les systèmes centralisés ignorent ces opérations. Ceci peut entraîner une « non transparence » des programmes.

Les appels de procédures à distance (*Remote Procedure Calls*) imitent les appels classiques, mais les autorisent à s'exécuter sur une autre machine. L'information est transportée à travers le réseau. Elle correspond, pour l'appelant, à la fonction et à ses paramètres; pour l'appelé, aux données retournées. Les opérations d'entrées-sorties sont invisibles au programmeur. Elles sont gérées par des procédures d'interfaces : les talons (*stubs*).

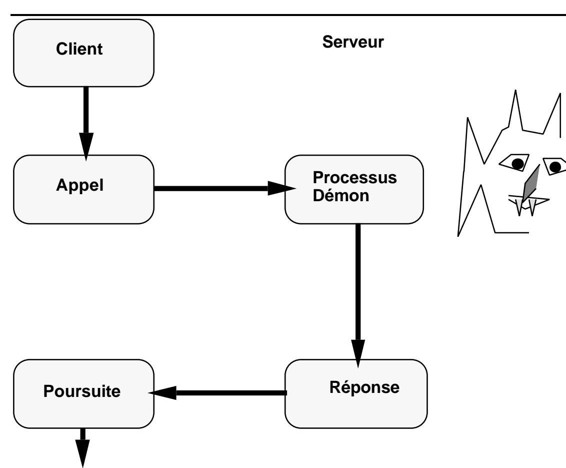
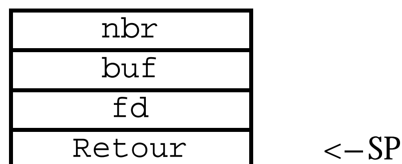


Figure 60 Un appel à distance

6.2.2 Les mécanismes de passage de paramètres

Les mécanismes de passage de paramètres sont cependant légèrement modifiés. Dans le cas classique d'une fonction : `n = read(fd, buf, nbr);` les paramètres sont poussés dans la pile à l'appel :



Au retour, ils sont éliminés (dépilés).

Les paramètres appelés par valeurs, comme `fd` ou `nbr`, sont copiés dans la pile. Les paramètres appelés par référence, comme `buf`, sont transmis par leur pointeur.

Une autre forme de passage par copie/restauration passe les paramètres par valeur mais les recopie au retour. Cette forme est similaire à l'appel par référence sauf si le même paramètre apparaît plusieurs fois dans les arguments, par exemple avec 2 tableaux.

6.2.3 Le mécanisme de passage de paramètres dans les RPC

Les passages de paramètres dans les RPC tâchent de ressembler le plus possible à ceux des appels ordinaires. Au lieu de faire appel au noyau pour accéder au disque, un talon (*client stub*) encapsule les paramètres dans un message. Il appelle le noyau qui émet un `send` puis se bloque en réception avec un `receive`.

Le talon du serveur (*server stub*) désencapsule les paramètres et appelle la procédure correspondante. Ce talon construira le message de résultats que le noyau renverra à travers le réseau. La talon du client désencapsule les paramètres

qui lui parviennent et retourne au programme appelant comme une procédure locale.

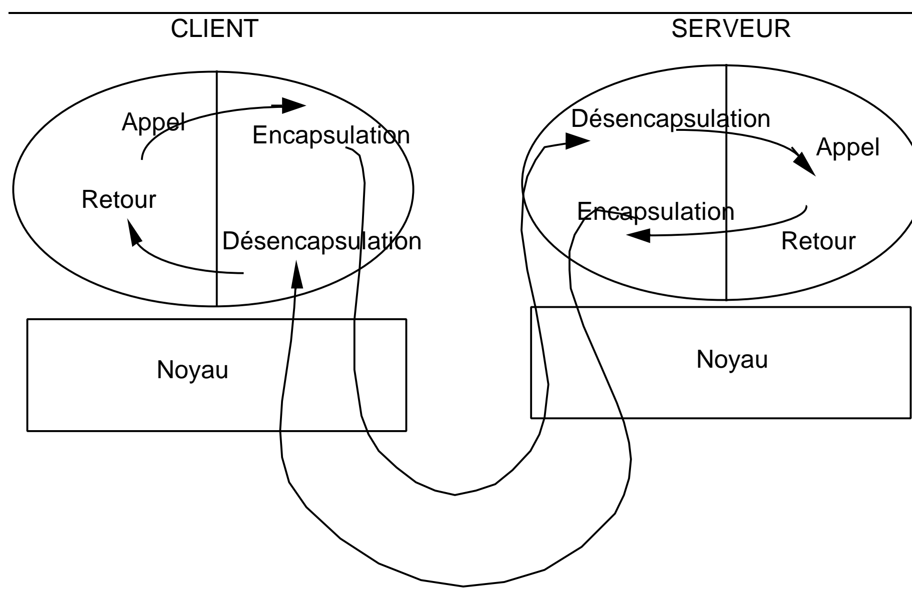


Figure 61 Le mécanisme de passage de paramètres dans les RPC

L'appel de fonction le plus simple se traduit par la transmission du nom de la procédure et des paramètres.

```
fonction(a, b)    ->   fonction, a, b
```

Dans le cas d'une procédure, il est impossible de transmettre les pointeurs. Il faut alors transmettre tous les paramètres dans le message : `read(fd, buf, nbr)->    read, fd, buf[0], buf[1], buf[2], buf[3],… buf[nbr - 1], nbr`

Le passage par paramètres devient un passage par copie/restauration. Le talon (*stub*) gère les différents passages par un triage des paramètres (*marshaling*). Il ne peut cependant pas se débrouiller de structures complexes telles que des graphes.

On peut optimiser ces procédures en distinguant les paramètres d'entrée, des paramètres de sortie. Ceci économise un transfert des données dans un des deux

sens. Les données en entrée du serveur lui seront transmises à l'envoi. Les données en sortie (du serveur) ne transiterons qu'au retour.

Enfin pour assurer une compréhension universelle (de toutes les machines), il faut passer par une représentation intermédiaire, par exemple XDR (*eXternal Data Representation*).

6.2.4 L'enregistrement des services

Pour faciliter les mises à jour des services, leurs différentes caractéristiques sont centralisées. Pour ceci, les serveurs doivent enregistrer auprès d'un aiguilleur (*binder* ou *portmapper*) : leur nom, leur n° de version et un identificateur entier. Pour se localiser physiquement, le serveur donne aussi son adresse (Ethernet, Internet ou autre). Le client à la recherche d'un service émettra vers l'aiguilleur une requête. Ce dernier lui retournera l'adresse du service. le client pourra alors adresser sa demande au serveur.

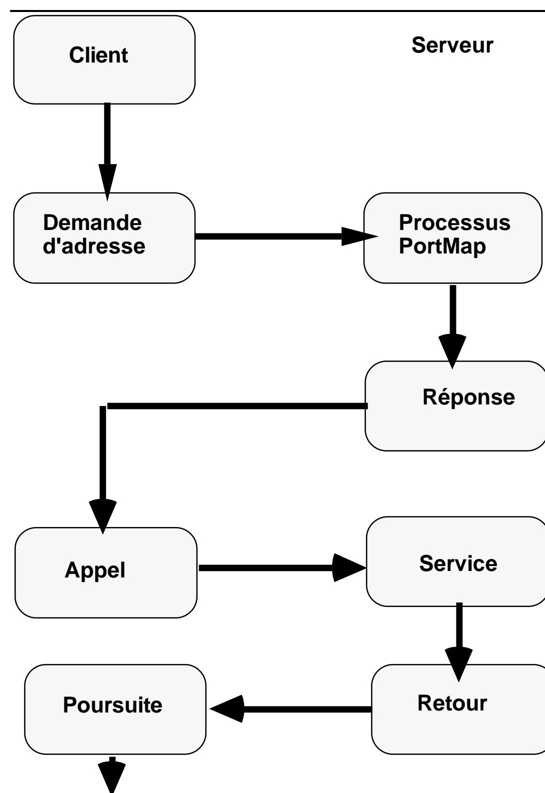


Figure 62 Schéma d'exécution

6.2.5 Les défaillances

Plus que pour les systèmes centralisés, les systèmes répartis sont susceptibles de défaillances. Ceci à cause de la multiplicité des matériels. Les principales défaillances sont : l'impossibilité de localiser le serveur; les messages de requêtes perdues; les messages de réponses perdues; les plantages du serveur; les plantages du client.

6.2.5.1 les messages perdus

Les messages de requêtes perdus sont aisément réparables par une simple retransmission. On peut pallier ce type de défaillance par l'implantation d'un compteur (*time out*) qui déclenchera une réexpédition si un envoi n'a pas été acquitté.

Les messages de réponses perdus sont en revanche plus difficiles à traiter. Certaines opérations peuvent se répéter sans dommage, par exemple la lecture d'un fichier. Elles sont dites idempotentes. Certaines autres opérations ne le sont pas, par exemple, envoyer un courrier. Il est alors plus difficile de remédier à la perte d'un message.

6.2.5.2 Les plantages du serveur

Les plantages du serveur sont aussi très dépendants de l'idempotence. Le plantage peut, en effet, avoir lieu avant ou après l'exécution. On est alors revenu au problème des pertes de messages.

Pour maîtriser ce type de pannes, on peut implanter 3 types de sémantique d'exécution pour les RPC. Aucune d'elles cependant n'est totalement satisfaisante :

- la procédure est exécutée <u>exactement une fois</u>. Ceci est la sémantique des appels locaux. Elle est impossible à réaliser pour les RPC.

- La procédure est exécutée au plus une fois. Si le client obtient une réponse, il sait qu'il y a une exécution, sinon il n'a aucune certitude.

- La procédure est exécutée au moins une fois. Le client répète sa requête tant qu'il n'a pas eu de réponse correcte.

6.2.5.3 Les plantages du client

Un client qui se plante peut laisser chez le serveur des processus orphelins. Plusieurs méthodes ont été décrites dans la littérature :

- l'extermination, qui consiste à éliminer tous les orphelins d'un client lors de son redémarrage;

- la réincarnation qui élimine brutalement tous les calculs répartis lors du redémarrage d'une station;

- l'expiration qui limite le temps de calcul pour chaque service. Si ce temps est dépassé, le serveur doit s'assurer que le client existe toujours.

En pratique, ces méthodes sont difficilement implantables.

6.2.6 L'implantation

6.2.6.1 Les protocoles

Les protocoles connectés sont préférables pour des transmissions sur une longue distance. Pour les réseaux locaux où le temps de réponse est crucial, on choisit souvent des protocoles non-connectés. Le protocole le plus utilisé est UDP/IP. Ces deux protocoles sont dépassés techniquement car ils ajoutent de nombreux champs inutiles. Trois champs seulement sont nécessaires : la destination, la source et la longueur. Cependant le poids du matériel existant fait qu'ils seront difficilement remis en cause.

6.2.6.2 Les performances

Les performances dépendent de 5 parties principales : l'encapsulation de l'appel, le traitement du noyau du client et l'envoi, le transit, la réception par le noyau du serveur, la désencapsulation du message.

Des mesures ont montré que pour une RPC nulle, les parties sont relativement égales à l'exception du transit qui prend peu de temps. Pour une trame Ethernet pleine, les 5 parties sont à peu près égales.

6.2.6.3 Les acquittements

Des appels de procédures peuvent être perdus parce que le message est perdu sur le réseau ou bien parce que le récepteur est submergé (*overrun*).

Pour savoir si le traitement a été effectué ou non, les appels de procédures utilisent un compteur afin éventuellement de réitérer leur demande. La durée de la temporisation est un compromis à réaliser :

- un courte temporisation peut entraîner des retransmissions inutiles;
- une longue temporisation peut créer des attentes importantes pour des paquets perdus.

6.3 Les RPC de Sun

6.3.1 Introduction

Plusieurs normes d'appels de procédures à distances ont été définies sur Unix. Parmi elles on trouve les RPC de SUN, le protocole Courier de Xerox, les RPC de Apollo : Network Computing System. En fait, les RPC de la société Sun sont de loin les plus répandues. Sun a étendu ce concept à Java sous le nom d'invocation de méthodes à distance (RMI).

Les RPC de Sun sont construites au-dessus des sockets de Berkeley. Elles comprennent une bibliothèque : `rpclib` et un utilitaire de génération de talons :

`rpcgen`. Elles sont normalement synchrones. Il est néanmoins possibles de les rendre asynchrones. Elles autorisent les passage de paramètres par valeur ou par pointeurs. Elles n'acceptent pas les variables globales. L'appel de la procédure à distance utilisera un port quelconque de la machine cliente. Le service utilise aussi un port quelconque.

La représentation des données pour la transmission est effectué par le protocole XDR (*eXternal Data Representation*). Les RPC de Sun utilisent souvent un transport en mode non-connecté : les datagrammes UDP associées à IP. Les paquets sont transférés par blocs de 8 ko. Il est cependant possible d'utiliser TCP/IP.

6.3.2 *Le* portmapper

Les services utiliseront des ports d'écoute quelconques. Ces services s'enregistreront auprès d'un processus d'aiguillage : le PortMapper. Pour obtenir l'adresse d'un service, le client adressera ses demandes au PortMapper dont le port est constant : n° 111. Le PortMapper aiguillera la demande vers le bon service. Le client adressera alors sa requête au service. Le service renverra les résultats au client à son port d'émission.

6.3.3 *La nomenclature des appels*

Chaque appel transmet un identificateur de requête, des identificateurs de services et un identificateur d'authentification. Les services sont identifiés par 3 entiers de 32 bits : un n° de programme, un n° de procédure et un n° de version.

Les n° de programme se codent dans les intervalles suivants :

Min	Max	Type
`0x00000000`	`0x1fffffff`	Sun
`0x20000000`	`0x3fffffff`	Utilisateurs

`0x40000000`	`0x5fffffff`	Transitoire
`0x60000000`	`0x7fffffff`	Sun

Les n° des procédures commencent à 1. Le n° 0 est réservé à la procédure nulle. Les versions commencent à 1.

6.3.4 La gestion des erreurs

Les RPC de Sun ont une sémantique d'appel se référant à l'idempotence. Pour implanter de facto la sémantique « au plus une fois », il faut utiliser un transport TCP. Les RPC gèrent un temporisateur paramétrable qui répétera une requête un certain nombre de fois avant de retourner une erreur. On peut paramétrer ces temporisations en termes de durée et en terme de nombre de reprises.

6.3.5 Les niveaux de programmation

On peut programmer des RPC à 3 niveaux différents. Le 1er niveau ne permet que l'utilisation de RPC déjà enregistrées. Le 2e ne permet que l'utilisation de UDP. Il offre cependant une grande variété de fonctionnalités qui sont souvent suffisantes pour les cas courants. Le 3e donne accès à toutes les possibilités des RPC.

6.3.6 La structure des programmes

Les programmes serveurs auront la structure suivante :

- création d'un descripteur RPC (handle);
- enregistrement auprès du processus PortMap;
- traitement des appels (cette partie est similaire à celle d'une procédure locale).

Les programmes clients auront la structure suivante :

- création d'un descripteur RPC (handle);
- appel;
- libération du descripteur.

La « fabrication » de ces programmes est largement facilitée par `rpcgen`.

6.3.7 Les fonctions serveur de la bibliothèque

Création d'un descripteur (*handle*) UDP :

```
SVCXPRT *svcudp_create(int sock)
```

création d'un descripteur TCP :

```
SVCXPRT *svctcp_create(int sock, int sendz, int recvz)
```

Destruction d'un descripteur :

```
void svc_destroy(SVCXPRT *xptr)
```

Enregistrement :

```
bool_t svc_register(SVCXPRT *xptr, u_long prognum, u_long versnum, void (*dispatch)(), u_long protocol).
```

`protocol` = `IPPROTO_UDP` ou `IPPROTO_TCP`

Suppression d'un service :

```
bool_t svc_unregister(u_long prognum, u_long versnum)
```

Réponse à un client :

```
bool_t svc_sendreply(SVCXPRT *xptr, xdrproc_t outproc, char *out)
```

6.3.8 Les fonctions client de la bibliothèque

Création d'un client TCP ou UDP :

`CLIENT *clnt_create(char *host, u_long prognum, u_long versnum, char *protocol)`

avec `protocol` = `"udp"` ou `"tcp"`

Création d'un client UDP :

`CLIENT *clntudp_create(struct sockaddr_in *addr, u_long prognum, u_long versnum, struct timeval wait, int *psock)`

avec `wait` déterminant la tempo de retransmission (5 s par défaut) et `psock` le socket de retour

Création d'un client TCP :

`CLIENT *clnttcp_create(struct sockaddr_in *addr, u_long prognum, u_long versnum, int *psock, int sendz, int recvz)`

avec `sendz` et `recvz` déterminant les tailles des tampon TCP.

Destruction d'un descripteur :

`void clnt_destroy(CLIENT *xptr)`

Appel d'une procédure :

`enum clnt_stat clnt_call(CLIENT *cl, u_long procnum, xdrproc_t inproc, char *in, xdrproc_t outproc, char *out, struct timeval wait)`

où `inproc` et `outproc` sont les procédures XDR d'encodage et de décodage; `in` et `out` sont les données d'entrées-sorties; `wait` est le temporisateur total de l'appel (nombre maxi de retransmission)

Contrôle général de la transmission :

`bool_t clnt_control(CLIENT *cl, int request, char *info)`

Cette fonction est analogue aux `ioctl`.

6.3.9 Compléments sur les fonctions

Gestion d'erreurs

`clnt_perror(), clnt_pcreateerror()` et `svcerr_system()` permettent d'afficher des erreurs. Elles sont analogues à la fonction `perror()`

Authentification

Les paramètres d'authentification sont transmis avec chaque requête. Il est possible de ne pas en tenir compte (traitement par défaut); de vérifier l'UID et le GID avec `AUTH *authunix_create_default()` ou de coder les données avec le DES.

Gestion du démon d'aiguillage (PortMapper) :

- On peut obtenir la liste des ports attribués et des n° de programmes avec `pmap_getmaps()`.
- On peut associer un programme à un port avec `pmap_set()` et le dissocier avec `pmap_unset()`.

Bibliothèques d'inclusions : `<rpc/rpc.h>`

La commande `rpcinfo` permet d'obtenir des informations sur les fonctions enregistrées par le démon d'aiguillage.

6.4 Les modèles de représentations universelles

Pour s'affranchir de la représentation des machines, on utilise une représentation universelle pour transporter les données. L'OSI a normalisé une représentation ASN.1 (*Abstract Syntax Notation 1*) qui n'est pas utilisée dans les systèmes répartis. (Elle l'est en revanche dans la gestion des réseaux).

XDR (eXternal Data Representation) fait partie de l'ensemble des protocoles de Sun. Elle permet de représenter les données sous une forme universelle et de

réaliser les communications en mémoire, à travers des fichiers ou à travers le réseau.

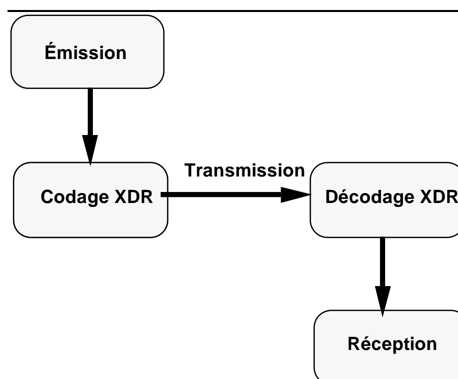


Figure 63 Le codage et le décodage XDR

Le codage XDR ne transporte pas la description du type codé, au contraire de ASN.1. Ceci signifie que la procédure de décodage doit savoir ce qu'elle décode.

6.4.1 Le codage XDR

Le codage XDR est du type big-endian. L'octet de poids fort est à l'adresse la plus petite. Par exemple pour le chiffre 5 :

oct. 0	oct. 1	oct. 2	oct. 3
0	0	0	5

Les réels sont codés au format IEEE sur 32 bits.

6.4.2 Le transcodage XDR

L'encodage XDR transforme les données d'émission dans une suite d'octets (un flot) à sa norme de représentation. La procédure de transformation : encodage ou décodage, s'appelle un filtre.

Pour réaliser les flots et les filtres, on dispose de fonctions rassemblées dans une bibliothèque dont l'en-tête est `<rpc/xdr.h>`

6.4.3 Les flots

On peut créer des flots d'entrées-sorties standards ou des flots en mémoire.

Création d'un flot :

```
void xdrstdio_create(XDR *xdr_handle, FILE *fichier, enum xdr_op op)
```

où op est XDR_ENCODE, XDR_DECODE ou XDR_FREE.

Création d'un flot en mémoire :

```
void xdrmem_create(XDR *xdr_handle, char *tampon, int taille_tampon, enum xdr_op op)
```

où op est aussi XDR_ENCODE, XDR_DECODE ou XDR_FREE.

En fonction de l'opérateur, on écrit ou on lit ensuite dans le flot par des filtres de type :

```
bool_t xdr_type(XDR *xdr_handle, type *value)
```

Par exemple : xdr_float, xdr_long, etc. Les données sont lues ou écrite dans le tampon ou le fichier, suivant que la création a été faite par xdrmem_create ou xdrstdio_create.

6.4.4 Les filtres de XDR

On dispose de filtres de base et on peut construire des filtres composites.

Les filtres de base correspondent au types simples :

```
bool_t xdr_type(XDR *xdr_handle, type *value)
```

Les filtres composites traitent les types suivants :

string, opaque, bytes, vector, array, reference, pointer,...

Les appels aux filtres reprennent les paramètres des filtres simples et en ajoutent d'autres correspondant aux types :

`bool_t xdr_string(XDR *xdr_handle, type *value, int max)`

À partir de ces types on peut construire des types complexes. Grâce à ces types, on peut ensuite lire ou écrire dans les flots des structures quelconques.

6.4.5 Réalisation pratique

En pratique on construit les filtres grâce au générateur `rpcgen`. `rpcgen` prend en entrée un fichier suffixé « `.x` » (par exemple `alpha.x`) contenant la déclaration des structures dans un langage très voisin du C. Par exemple :

```
struct structure {
    float a;
    string tab<10>;
    int mat[10];
};
```

`rpcgen` créera un fichier `alpha.h` qui contiendra les structures C correspondantes, ainsi qu'un fichier `alpha_xdr.c` avec les flots pour ces objets. La fonction d'encodage et de décodage portera le nom de `xdr_structure(XDR *xhandle, struct structure *in/*out)`.

6.5 Le générateur *RPCGEN*

`rpcgen` permet d'obtenir des programmes répartis à partir de leur spécification. On décrit les données et les programmes grâce à un langage voisin du C dans un fichier suffixé « `.x` » (`prog.x`). On traite ce fichier par le générateur `rpcgen`. Il produit :

- un fichier d'en-tête : `prog.h`
- deux fichiers clients : `prog_clnt.c` et `prog_xdr.c`

- deux fichiers serveurs : `prog_svc.c` et `prog_xdr.c`

Par défaut, `rpcgen` génère qu code conforme à la norme K & R 1978. On peut créer du C ANSI avec la commande `rpcgen -C`. Les programmes `client.c` et `serveur.c` restent à la charge du programmeur.

6.5.1 Les données de RPCGEN

Le fichier de données est suffixé « `.x` »

1. On définit les données en entrée et en sortie. Une fonction a au plus un paramètre d'entrée et un de sortie. On doit donc avoir recours à des structures pour des appels avec plusieurs paramètres.

2. Les structures sont éventuellement redéfinies grâce à `typedef`

```
typedef données en entrée

typedef données en sortie
```

3. On déclare ensuite le programme et l'ensemble des procédures appelées à distance.

```
program PROGRAMME {
   version VERSION {
        typeOut PROC_UN(typeIn) = 1;
        typeOut PROC_DEUX(typeIn) = 2;
        ...
   } = 1;
} = 0x20000010;
```

6.5.2 Le développement des programmes (le client)

Le squelette général d'un client est le suivant :

```
#include <rpc/rpc.h>
#include "prog.h"
void client()
```

```
{
   CLIENT    *cl;       /* RPC handle */
   typeIn    entrée;    /* param entrée */
   typeOut   *out;           /* param sortie */
   char      *serveur;

   cl = clnt_create(serveur, PROGRAMME, VERSION ,
"udp");

   out = proc_un_1(&entrée, cl);
   clnt_destroy(cl);
}
```

6.5.3 Le développement des programmes (le serveur)

Le nom des services est généré automatiquement par `rpcgen`. à partir du fichier « `.x` ».

```
#include <rpc/rpc.h>
#include "prog.h"
TypeOut *proc_un_1(entrée)
TypeIn *entrée;
{
        static typeOut sortie;
                       /* Doit être static */
        …
        …
        return &sortie;
}
```

6.5.4 Quelques points à retenir

1. Le nom des procédures à distance doit être écrit en majuscules.

2. Les appels ne doivent pas comporter plus d’un paramètre.

3. Les passages des paramètres et leur retour doivent se faire par des pointeurs.
4. Le paramètre de retour des procédures de service doit être déclarée en `static`.

6.5.5 La production de code

L'outil `rpcgen` produit :

- un fichier d'en-tête : `prog.h`
- deux stubs : `prog_clnt.c` et `prog_svc.c`
- un filtre XDR : et `prog_xdr.c`

Le programmeur produit :

- `serveur.c`
- `client.c`

Le compilateur produit :

- le client avec : `client.c`, `prog_clnt.c`, `prog_xdr.c`
- le serveur avec : `serveur.c`, `prog_svc.c`, `prog_xdr.c`.

6.5.6 Les performances

De nombreuses mesures de performances ont été effectuées pour évaluer les RPC. De multiples facteurs peuvent jouer dans cette évaluation. En général, un remplacement tel quel est environ 10 fois plus lent avec une RPC.

Le temps de traitement d'une procédure nulle est de quelques millisecondes lors d'un appel blocant (5 à 10). Cette durée donne un ordre d'idée du seuil à partir duquel il est rentable de transférer des calculs sur un ordinateur très puissant.

6.5.7 Un exemple

Le fichier de description du squelette :

```
program DATE_PROG {
        version DATE_VERS {
                long BIN_DATE(void) = 1;
                string STR_DATE(long) = 2;
        } = 1;
} = 0x31234567;
```

Le programme d'appel du client

```
#include <rpc/rpc.h>
#include "date.h"

void main(int argc, char **argv) {
    CLIENT *cl;
    char *server;
    long *lresult;
    char **sresult;

    server = argv[1];
    if  ((cl  =  clnt_create(server,  DATE_PROG,  DATE_VERS,
"udp")) == NULL) {
          clnt_pcreateerror(server);
          exit(2);
    }
    if ((lresult = bin_date_1(NULL, cl)) == NULL) {
          clnt_perror(cl, server);
          exit(3);
```

```
    }
    printf("time on host %s = %ld\n", server, *lresult);
    if ((sresult = str_date_1(lresult, cl)) == NULL) {
          clnt_perror(cl, server);
          exit(4);
    }
    printf("time on host %s = %s\n", server, *sresult);
    clnt_destroy(cl);
    exit(0);
}
```

Les fonctions du serveur :

```
#include <rpc/rpc.h>
#include "date.h"

long *bin_date_1() {
    static long timeval;
    long time();
    timeval = time((long *) 0);
    return(&timeval);
}

char **str_date_1(long *bintime) {
    static char *ptr;
    char *ctime();
    ptr = ctime(bintime);
    return(&ptr);
}
```

6.6 Les invocations de méthodes à distance (RMI)

6.6.1 L'architecture

L'invocation de méthodes à distance est une variante des appels de procédures à distance pour Java. Les RMI tirent complètement parti des objets de Java et ils les « distribuent » sur le réseau. On peut ainsi appeler les méthodes d'objets lointains. On dit alors qu'on les invoque. On crée les *stubs* d'une manière similaire aux RPC avec le générateur `rmic`.

Les spécifications de l'objet distant doit être faite par une `interface` Java et hériter de la classe `java.rmi.Remote`. Ceci est semblable aux fichiers « `.x` » des RPC. Elles doivent obligatoirement tenir compte d'exceptions possibles :

```
import java.rmi.*

public interface MonObjetDistant extends java.rmi.Remote {

       public UneClasse méthode1() throws java.rmi.RemoteException;

       public UneClasse méthode2() throws java.rmi.RemoteException;

}
```

L'implantation de l'objet distant, comparable aux fonctions serveur des RPC, hérite de `java.rmi.UnicastRemoteObject` et doit « implanter » l'interface précédente.

```
public class CodeObjetDistant implements MonObjetDistant extends
java.rmi.UnicastRemoteObject {

       // Constructeur

       public CodeObjetDistant()throws java.rmi.RemoteException{… //code}

       public UneClasse méthode1() throws java.rmi.RemoteException{…}

       public UneClasse méthode2() throws java.rmi.RemoteException{…}

// Autres méthodes destinées au fonctionnement de la classe

}
```

On nomme un objet distant par l'adresse `rmi://nom de la machine/nom de l'objet`. L'enregistrement des services se fait par l'intermédiaire de la classe `Naming` et du programme `rmiregistry`.

6.6.2 Un exemple

Rédaction en cours

6.7 Les serveurs de fichiers

6.7.1 Introduction

6.6.1.1 Le partage des fichiers

Un système d'exploitation de réseaux permet de monter des fichiers à distance. Le montage porte sur des répertoires et un répertoire partagé implique – en général – le partage de tout ses fils. Suivant les systèmes d'exploitation, différentes machines peuvent effectuer des montages différents ou similaires.

6.6.1.2 Le « montage » des répertoires

À partir de deux systèmes de fichiers séparés :

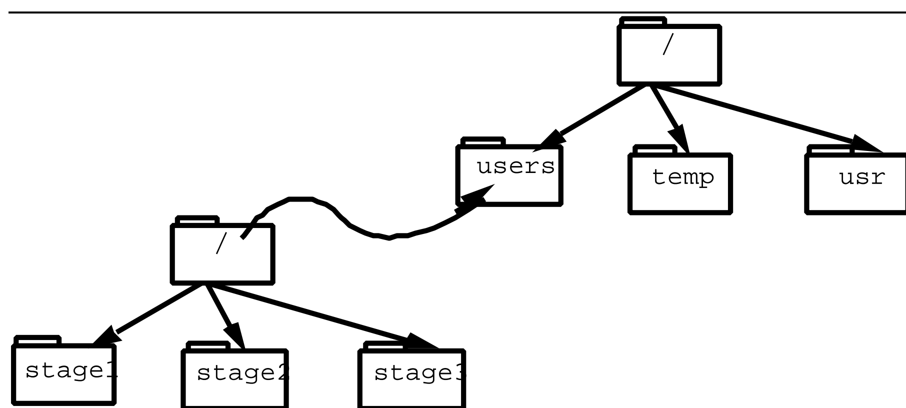


Figure 64 Le montage de disques

Le montage produit le système « monté » suivant :

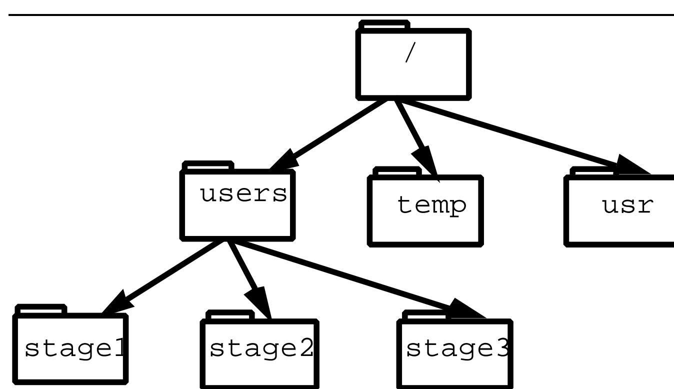


Figure 65 Les disques montés

6.7.2 NFS

Le système *Network File System* de Sun est l'un des systèmes d'exploitation de réseau les plus répandus. Il utilise les protocoles de RPC de la société du même nom et il fonctionne avec UDP/IP. Il repose sur le montage de parties d'arborescences de serveurs par des clients. Ces clients peuvent être avec ou sans disques. Le partage intervient lorsque deux clients « montent » la même arborescence.

Avec NFS, la machine fournissant le disque (le serveur) indique les répertoires disponibles (partageables) dans `/etc/exports`. Les machines utilisant le

disque (les clients) indiquent les fichiers qu'elles utilisent dans le fichier /etc/fstab.

6.6.2.1 Les protocoles de NFS : les poignées

NFS autorise les matériels et les systèmes d'exploitation hétérogènes. Les différentes machines doivent simplement implanter les protocoles spécifié par lui. Les manipulations des fichiers entre les différentes machines se font par l'intermédiaire de « poignées » (*handles*) de fichiers Les poignées de répertoires sont fournies au montage de l'arborescence. Le client expédie au serveur l'arborescence (le chemin) qu'il désire monter. Si cette arborescence est exportée, le serveur retourne la poignée : type du système de fichier, n° du disque, i-nœud du répertoire, droits.

6.6.2.2 Les RPC de montage

Les procédures de montage sont implantées sous la forme d'appels de procédures à distance.

```
program MOUNTPROG {
   version MOUNTVERS {
        void MOUNTPROC_NULL(void) = 0;
        fhstatus MOUNTPROC_MNT(dirpath) = 1;
        mountlist MOUNTPROC_DUMP(void) = 2;
        void MOUNTPROC_UMNT(dirpath) = 3;
        void MOUNTPROC_UMNTALL(void) = 4;
        exportlist MOUNTPROC_EXPORT(void) = 5;
   } = 1;
} = 100005;
```

6.6.2.3 Les protocoles de NFS : les opérations

Les opérations concernent des manipulations de fichiers et de répertoires. Ces manipulations sont voisines de celles des appels au système de fichiers d'Unix. Elles comprennent la création de fichiers, les lectures, écritures, accès aux attributs de : tailles, droits, date,... Elles ne comprennent pas les opérations

d'ouverture et de fermeture de fichiers: `open` et `close`. NFS est un protocole « sans état » (stateless). Pour effectuer une lecture, le client va d'abord envoyer au serveur le nom de ce fichier et recevoir une poignée. Les opérations ultérieures adresseront au serveur la poignée, le décalage, et le nombre d'octets.

6.6.2.4 Les avantages d'un protocole « sans états »

L'avantage des systèmes sans état tient au fait que le serveur n'a pas à se souvenir de ses connexions avec ses clients. C'est un processus sans mémoire. Le système est ainsi beaucoup plus résistant au défaillances de machines. Si le serveur se « plante » et redémarre, il n'aura perdu aucune informations sur les fichiers ouverts (puisqu'il n'en a pas).

6.6.2.5 Les inconvénients d'un protocole « sans états »

L'inconvénient principal de ce type de systèmes est qu'il ne respecte pas la sémantique des systèmes d'exploitation courants. En particulier le comportement de NFS est différent sur certains points de celui d'Unix. La pose des verrous sur les fichiers « ouverts » nécessite certaines adaptations. Les opérations de partage des fichiers – du fait des mémoires cache – peuvent avoir des résultats imprévisibles.

6.6.2.6 Les RPC des opérations de NFS

```
program NFS_PROGRAM {
   version NFS_VERSION {
        void NFSPROC_NULL(void) = 0;
        attrstat NFSPROC_GETATTR(fhandle) = 1;
        attrstat NFSPROC_SETATTR(sattrargs) = 2;
        void NFSPROC_ROOT(void) = 3;
        diropres NFSPROC_LOOKUP(diropargs) = 4;
        readlinkres NFSPROC_READLINK(fhandle) = 5;
        readres NFSPROC_READ(readargs) = 6;
        void NFSPROC_WRITECACHE(void) = 7;
        attrstat NFSPROC_WRITE(writeargs) = 8;
```

```
          diropres NFSPROC_CREATE(createargs) = 9;
          stat NFSPROC_REMOVE(diropargs) = 10;
          stat NFSPROC_RENAME(renameargs) = 11;
          stat NFSPROC_LINK(linkargs) = 12;
          stat NFSPROC_SYMLINK(symlinkargs) = 13;
          diropres NFSPROC_MKDIR(createargs) = 14;
          stat NFSPROC_RMDIR(diropargs) = 15;
          readdirres  NFSPROC_READDIR(readdirargs)  =
16;
          statfsres NFSPROC_STATFS(fhandle) = 17;
     } = 2;
  } = 100003;
```

6.7.3 L'implantation de NFS

L'implantation de NFS par Sun est fondée sur trois couches : les appels système; le système de fichiers virtuel; le noyau. La couche d'appel système vérifie la rectitude syntaxique des appels. La couche de fichiers virtuels va interfacer les i-nœuds d'Unix avec les références des fichiers distants. Le noyau va exécuter localement les commandes provenant du client.

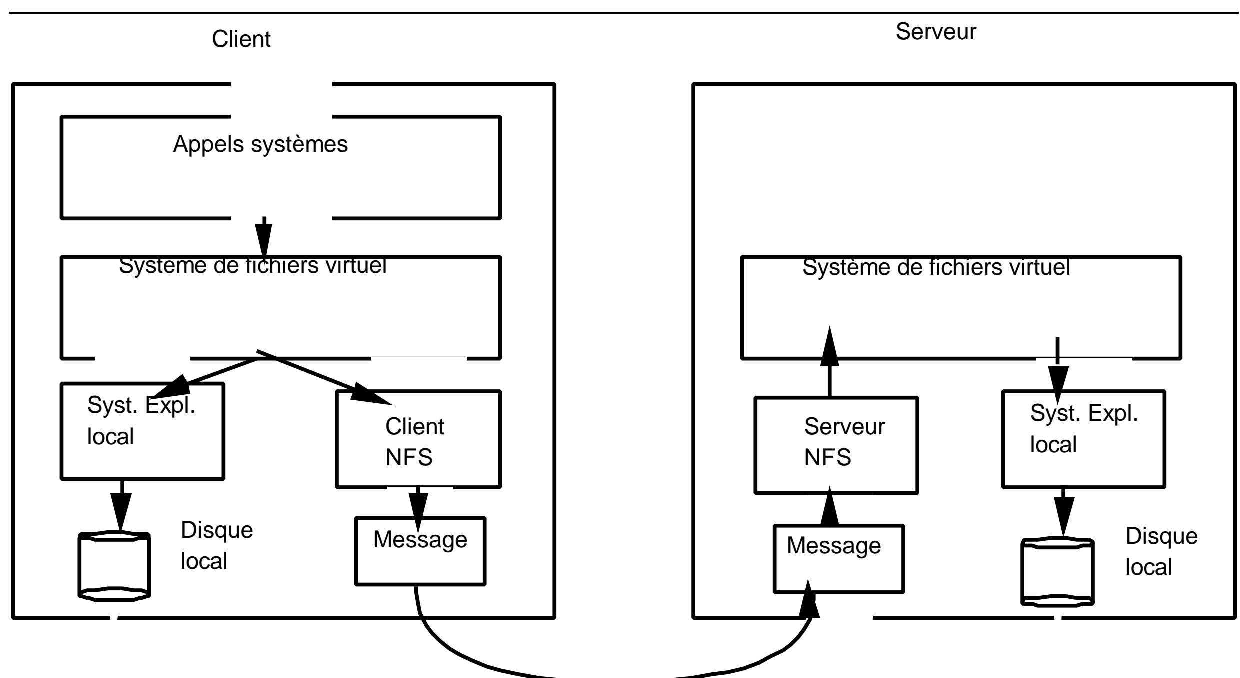


Figure 66 Un serveur de fichiers

6.6.3.1 Les v-nœuds

Les nœuds virtuels constituent l'interface entre les appels systèmes et les systèmes d'exploitation locaux ou distants.

Chaque v-nœud pointe soit sur un i-nœud (pour Unix) dans le cas ou le fichier est local, soit sur un r-nœud (*remote*) dans le cas ou le fichier est distant.

Le r-nœud contient la poignée du fichier lointain. À partir du v-nœud il est donc possible de retrouver les caractéristiques des différents fichiers.

6.6.3.2 L'ouverture d'un fichier avec NFS

Pour effectuer une ouverture (du côté du client), le système analyse l'enchaînement de répertoires. Si le fichier est distant, à un certain moment, l'arborescence correspond à un système monté. Le client demandera au serveur une poignée pour le fichier distant. Le client créera un r-nœud pour le fichier ouvert où il conservera la poignée. Pour chaque fichier ouvert, il correspondra donc au v-nœud : soit un i-nœud, soit un r-nœud. L'appel système récupérera un descripteur entier relié au v-nœud du client. Le serveur, quant à lui, ne conservera aucune trace.

6.6.3.3 La lecture et l'écriture d'un fichier avec NFS

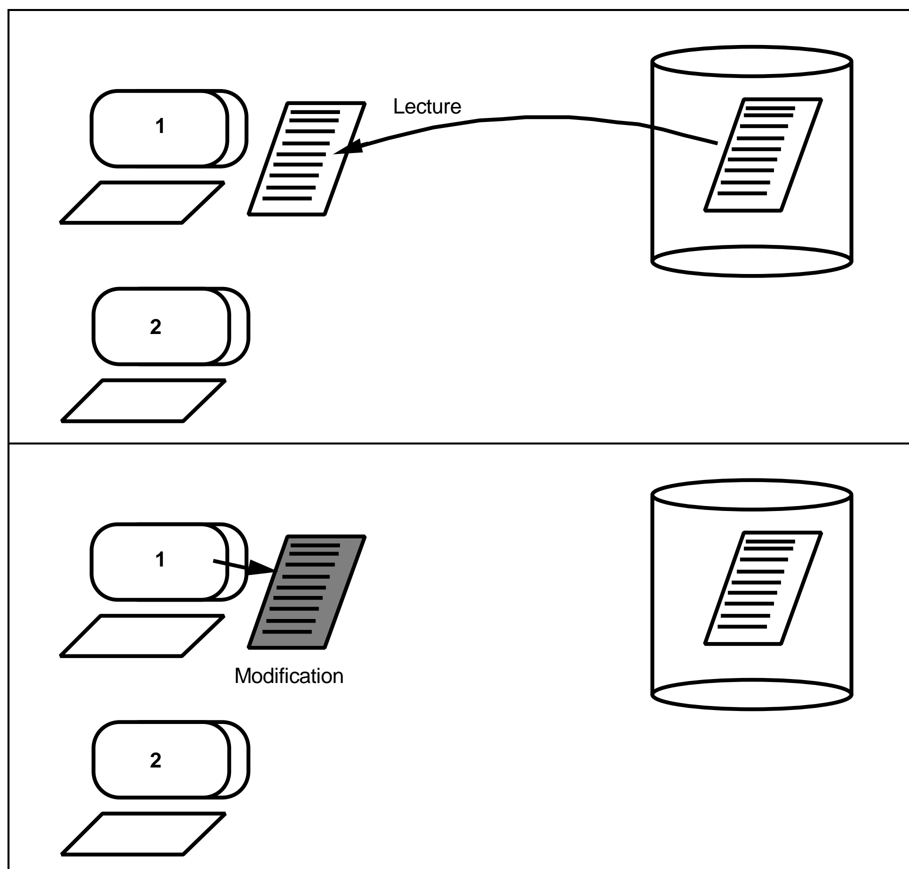
Les appels ultérieurs concernant une lecture ou une écriture détermineront par le v-nœud si le fichier est distant ou local. Dans le cas de fichiers distants, les données seront transférées par blocs de 8 ko. Elles seront « cachées » par le client. Ceci signifie, dans le cas d'un écriture, que le transfert vers le serveur ne s'effectue que lorsque le bloc est plein.

La mémoire cache améliore notablement les performances du système. Dans le cas d'un système d'exploitation centralisé – comme Unix –, les processus

accédant à la mémoire accéderont au même cache. En revanche, le cache pose un certain nombre de problème aux systèmes répartis.

6.6.3.4 Le problème du cache

Le cache risque de rendre les données de fichiers partagés incohérentes. À la différence des systèmes centralisés, un bloc de données manipulé par plusieurs machines sera recopié sur toutes les machines qui l'exploitent. Dans le cas de lectures multiples, ceci est sans inconvénients. Dans le cas d'écritures, il peut se produire des effets inattendus, car les modifications ne sont pas immédiatement reportées sur toutes les machines.



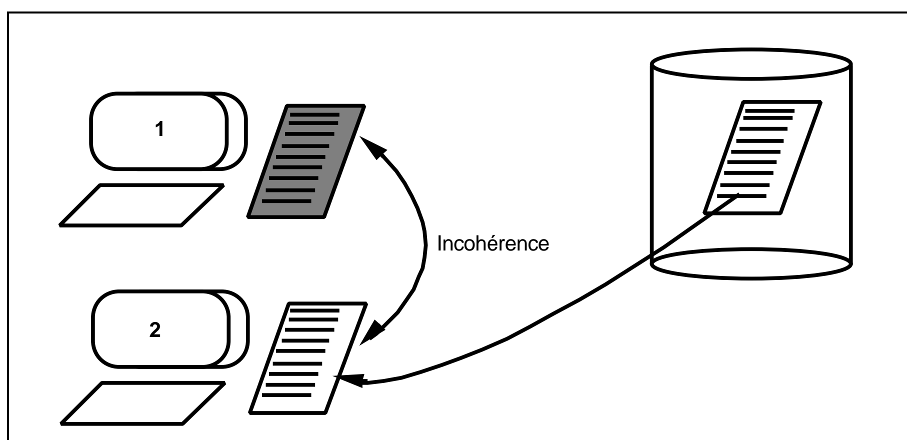


Figure 67 La cohérence du cache

6.6.3.5 *Comment NFS pallie ce problème*

NFS ne résout pas totalement les incohérences potentielles dues au cache; il tente seulement de les limiter.Pour ceci, les blocs lus n'ont qu'une durée de vie de 3 s. Les blocs modifiés dans le cache sont retournés au serveur par un démon s'exécutant toutes les 30 s. Chaque fois qu'un fichier est ouvert et qu'il y correspond un cache, le client compare la date de dernière modification et la date de cache. Dans le cas où cette dernière est plus ancienne, il effectuera une nouvelle lecture des données.

Ce précautions n'éliminent cependant pas totalement les risques. NFS n'implante pas la sémantique d'Unix. Notamment, un fichier peut être créé sur une machine et rester invisible aux autres pendant 30 s.

6.8 Les processus dans les systèmes répartis

non disponible en 1999

6.9 Les autres services

Certains paragraphes ne sont pas disponibles en 1999

6.9.1 Les messageries

6.8.1.1 SMTP, POP et IMAP

6.8.1.2 X.400 et X.500

6.9.2 Les serveurs d’impression

6.9.3 Répartition et nommage

6.8.3.1 Les pages jaunes

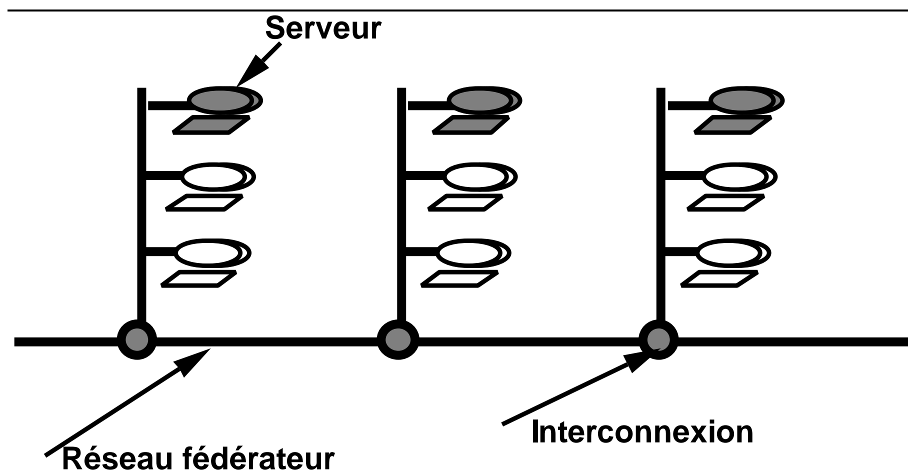
Un service de Yellow Pages s’efforce de rendre unique sur le réseau le fichier `/etc/hosts`.

ypcat hosts group passwd etc.

6.8.3.2 Les serveurs de noms

6.9.4 La gestion l’administration des réseaux

Les réseaux, au départ locaux, deviennent de taille de plus en plus importante. Ils sont, à proprement parler, le système informatique de l’entreprise. Pour maîtriser leur croissance, on les segmente, en général, en sous-réseaux. Les sous-réseaux sont reliés entre-eux par des épines dorsales. Les épines dorsales ont souvent pris la forme de réseaux rapides, utilisant des normes telles que FDDI. Actuellement, elles sont remplacées par des commutateurs Ethernet 10 ou 100 MHz. À l’avenir, ces commutateurs utiliseront sans doute la norme ATM.



La croissance en taille des réseaux rend leur administration de plus en plus compliquée.

6.8.4.1 Netstat et autres

Sur la plupart des systèmes Unix répartis, plusieurs commandes sont disponibles pour connaître l'état du réseau :

- `ifconfig` permet de configurer l'interface, en général, carte Ethernet
- `arp` permet de connaître les associations d'adresses Ethernet/IP
- `netstat` permet d'obtenir certaines statistiques sur l'état des transmissions et du routage;
- `ping` permet de connaître les temps d'aller et retour entre deux machines
- `route` permet le contrôle des tables de routage.
- `traceroute` permet de connaître les enchaînements de routeurs entre deux destinations.
- `rpcinfo` donne des indications sur les RPC;
- `nfsstat` donne des informations sur NFS;

netstat fournit 3 types d'informations :

1. le premier correspond à la liste des sockets en service avec l'indication de leur état et de leur protocole : netstat -a;
2. le second fournit les structures de données du réseau :
   - netstat -v les statistiques sur la carte
   - netstat -s les statistiques par protocole
   - netstat -i les statistiques par interface
   - netstat -r les routes
   - netstat -m les mémoires tampons
3. le troisième affiche de manière continue des statistiques sur l'état des interfaces (netstat intervalle).

La commande route permet de régler manuellement les entrées de la table de routage. De cette manière on peut ajouter ou détruire des routes. On indique alors le réseau ou la machine de destination, la passerelle et le nombre de saut pour l'atteindre.

rpcinfo donne l'état des appels de procédures à distances. rpcinfo -p machine donne les démons s'exécutant sur la machine appelée. Parmi les programmes : 100003 est NFS, 100005 est le démon de montage à distance, 100000 est le portmapper, 100002 est rusers... On peut tester les procédures 0 (nulle) d'un démon sur une machine donnée grâce aux options -u (udp) et -t (tcp). On peut aussi tester un service et une version particulière sur toutes les machines grâce à un message de diffusion : broadcast (-b).

nfsstat permet d'obtenir certaines statistiques sur les appels systèmes de NFS. nfsstat -m donne des temps moyen d'exécution pour les lectures et écritures. nfsstat sans options, donne :

- pour le serveur, le nombre total d'appels à distance et d'appel système reçus;
- pour le client, le nombre total d'appels à distance et d'appel systèmes envoyés;
- pour chacun il détaille les pourcentages de chacun des appels système;
- il indique les divers échecs.

Les utilitaires précédents disponibles sur Unix ne le sont pas forcément sur les autres systèmes. Ils sont donc liés à un système d'exploitation et non pas aux applications de réseaux. Ils ne permettent pas facilement la téléopération, par exemple le redémarrage d'une station. Ces protocoles ne disposent pas de la possibilité d'alerte d'erreurs à une machine de gestion, par exemple, le plantage d'une station, l'engorgement d'un routeur, etc.

6.8.4.2 SNMP

SNMP (*Simple Network Management Protocol*) est un protocole de gestion de réseau qui tend à devenir un standard *de facto*. Il opère au niveau de la couche application ce qui présente des avantages : indépendance du réseau sous-jacent, indépendance vis-à-vis du matériel,... Ceci présente aussi des inconvénients : impossibilité d'atteindre les machines distantes si la couche de transport est en panne.

SNMP est proposé par de nombreux fournisseurs de matériels. D'autres protocoles seraient susceptibles de le concurrencer, notamment CMIP (*Common Management Information Protocol*) de l'OSI, mais ils n'ont gagné une reconnaissance comparable. SNMP est plus rudimentaire que ses concurrents, mais il a le mérite d'exister et d'être accepté.

SNMP est né d'une initiative pour administrer les réseaux de la recherche américaine (National Science Foundation). Les premières réalisations datent de 1987-1988. Elles se destinaient au départ à la simple gestion de passerelles TCP/IP. De ce fait SNMP est étroitement lié à TCP/IP et aux constructeurs américains.

SNMP est composé :

- de stations gérantes ou d'administration;
- de stations gérées. Ces stations sont des nœuds IP dotées d'un agent (d'espionnage) SNMP qui enregistre des statistiques.

SNMP définit – entre autres – :

- Un protocole d'échange entre les entités gérantes ou gérées.
- Une base d'informations MIB (*Management Information Base*) sur chaque entité.

La structure de l'information de gestion définit 3 types d'opérations :

- la consultation d'informations;
- le positionnement de variables d'un nœud distant;
- les alarmes.

La consultation ou le positionnement se font par l'exploration de la machine gérée. Pour ceci, sur chaque station gérée, un programme serveur, rentrera en communication avec le programme client de l'administrateur. Les alarmes sont des communications d'informations à l'initiative des stations gérées.

Sur chaque station on trouve une base de données constituée d'objets à gérer la MIB (Management Information Base). Les objets concernent principalement les différents protocoles IP. Elles gèrent aussi différentes couches et protocoles : MIB Ethernet, MIB TokenRing,... Les objets possèdent des variables de fonctionnement, propres, telles que les erreurs de transmission, le temps de bon fonctionnement,... Ces variables sont pour beaucoup statistiques et elles sont normalisées par un nom et un identificateur unique. Cette normalisation permet aux primitives de fonctionner correctement. Plusieurs normalisations de bases coexistent MIB 1, MIB 2 et RMON.

Une norme complémentaire définit comment on gère les variables de la MIB : le SMI (*Structure of Management Information*). Elle détermine l'organisation et la représentation des objets. Ces objets sont structurés sous la forme d'un arbre. SMI définit de manière précise des types tels que l'adresse IP et des règles de référencement des tableaux. La SMI impose l'utilisation de la norme ASN.1 pour définir les différentes variables.

Les objets se situent dans la structure suivante :

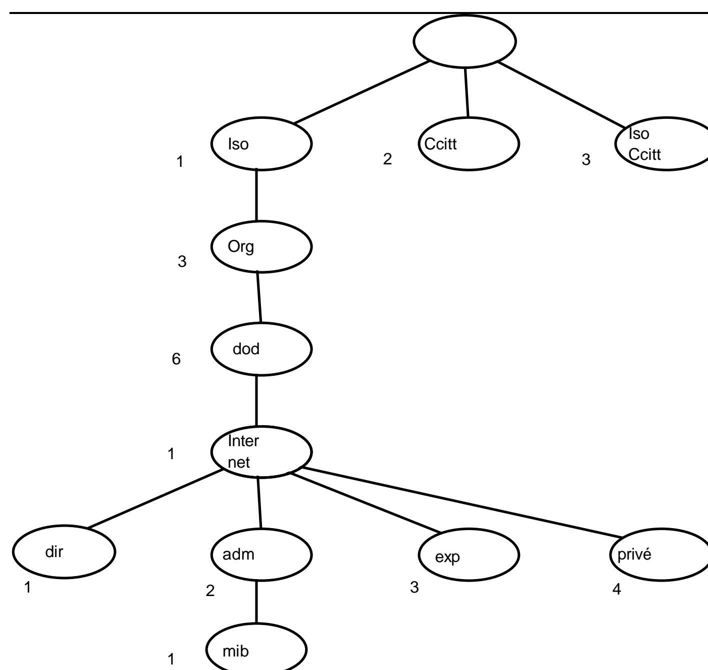


Figure 68 La structure des informations MIB

On accède aux objets par l'enchaînement des noms de la hiérarchie ou par ses indices. Par exemple, les objets de la MIB sont désignés par : 1.3.6.1.2.1...

La MIB est divisée en 8 sous-domaines à l'intérieur desquels sont situées les variables :

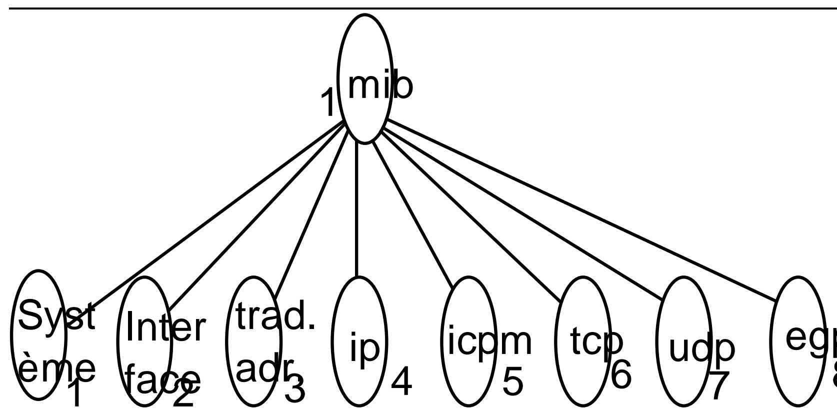


Figure 69 La sous structure Internet

`Système` : variables relatives au système d'exploitation. `Interface` : carte d'interface avec le réseau. `Traduction d'adresse` : par exemple correspondance ARP. `IP, ICMP` : protocoles de IP. `TCP, UDP` : protocoles TCP. `EGP` : protocole de routage.

Quelques exemples de variables :

Variables MIB	Catégorie	Signification
sysUpTime	Système	Durée depuis le dernier redémarrage
ifMtu	Interface	Max. Transfer Unit de l'interface
ipInReceives	ip	Nombre de datagrammes reçus
ipRoutingTable	ip	Tables de routages
icpmInEchos	icpm	Nombre de demande d'échos Icpm reçues
tcpInSegs	tcp	Nombre de segments tcp reçus
udpInDatagrams	udp	Nombre de datagrammes udp reçus

...

6.8.4.3 ASN.1

ASN.1 (Abstract Syntax Notation) est une norme OSI de définition universelle de types. Les variables sont membres de 4 classes :

- UNIVERSAL
- CONTEXT_SPECIFIC
- APPLICATION
- PRIVATE

Parmi les types universels (UNIVERSAL), ASN.1 distingue les types primitifs et les types construits.

Types primitifs notables:

- BOOLEAN
- INTEGER
- BITSTRING
- OCTETSTRING
- IA5String
- ANY

Les types construits ressemblent aux structures du C. Parmi les types notables :

- SEQUENCE : liste ordonnée d'éléments de même type (équivalent à un structure);

- SEQUENCEOF : liste ordonnée d'éléments de même type – en nombre variable;
- SET : ensemble d'éléments de type différent – non ordonnée en nombre fixe;
- SETOF : ensemble d'éléments de même type;
- CHOICE : choix d'un élément parmi une liste.

Comme exemple de définition ASN.1, une table de routage s'écrira :

```
ipAddrTable ::= SEQUENCEOF ipAddrEntry
ipAddrEntry ::= SEQUENCE {
        ipAdEntAddr              ipAddress,
        ipAdEntIndex             INTEGER,
        ipAdEntNetMask           ipAddress,
        ipAdEntBcastAddr         ipAddress,
        ipAdEntReasmMaxSize INTEGER
}
```

Les primitives de SNMP sont peu nombreuses, paradoxalement?

Les primitives de consultation sont :

- get request(station, variable)
- get response(station, valeur)
- get next request (*parcours d'une table sans connaître le nom de la variable*).

La primitive de positionnement est : `set(station, variable, valeur)`. La primitive d'alerte est : `trap(station, variable, valeur)`. Les dialogues s'effectuent en mode non-connecté. Les opérations sont atomiques.

Structure ASN.1 des primitives :

```
PDUs-SNMP ::=
   CHOICE {
        get-request
        get-next-request
        get-response
        set-request
        trap
   }
```

La conception de SNMP donne à ce protocole plusieurs avantages :

- il est stable parce que la définition est figée. On ne rajoutera pas de commandes ésotériques pour tenir compte de tel ou tel matériel;
- il est simple à mettre en œuvre et à comprendre;
- il est souple, car il peut inclure de nouveaux matériels par l'ajout de caractéristiques dans la base de données.

Pour redémarrer une station, par exemple, on positionne la variable de temps depuis le dernier redémarrage à 0.

La syntaxe ASN.1 ne permet pas d'indexer les tables. On retrouve les éléments par l'intermédiaire d'un identificateur particulier, par exemple l'adresse Ip, dans le cas de ipAddrTable. L'accès au membre « ipAdEntNetMask », d'une certaine adresse «128.10.2.3» se notera : `iso.org.dod.internet.mgmt.mib.ip.ipAddrTable.ipAddrEntry.ipAdEntNetMask.128.10.2.3`

On peut accéder à la table sans connaître aucune des adresses par la commande `get next request` et l'argument : `iso.org.dod.internet.mgmt.mib.ip.ipAddrTable.ipAddrEntry.ipAdEntNetMask`

Les messages SNMP ont la structure suivante :

```
Message-SNMP ::=
   SEQUENCE {
        version INTEGER {
             version 1 (0)
        },
        community OCTETSTRING,
        données ANY
   }
```

La zone de données se décompose en unité de données de protocoles (PDU). Les PDU de SNMP sont les suivants :

```
PDUs-SNMP ::=
   CHOICE {
        get-request
        get-next-request
        get-response
        set-request
        trap
}
```

La primitive `get request` est composée des champs suivants :

```
PDU-get request ::=
   IMPLICIT SEQUENCE {
        request-id
        error status
        error index
        variable bindings
}
```

request id est l'identificateur de la demande. error status et error index sont normalement à 0 lors d'une demande. variable bindings est une liste de couples (variable, valeur), par exemple (nom, nil).