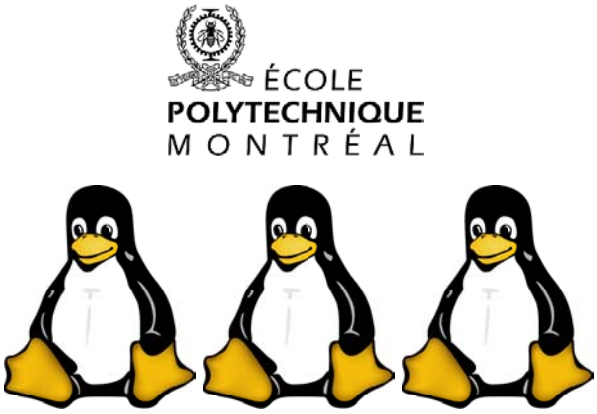
INF3600
Systèmes d'exploitation

Hanifa Boucheneb et Juan Manuel Torres-Moreno

Département de génie informatique
École Polytechnique de Montréal

version 3.90



16 avril 2003

Préface

Ce document représente la formalisation des notes de cours **INF3600-Systèmes d'exploitation**, enseigné au Département de génie informatique de l'École Polytechnique de Montréal.

Ce texte et les nombreuses applications qu'il contient visent à donner à l'étudiant[1] une idée, à la fois, assez claire que vaste de ce que sont les systèmes d'exploitation, notamment leur importance dans toute formation en informatique. Il s'adresse indistinctement aux étudiants en génie informatique, informatique, génie électrique ou sciences.

Le document aborde l'ensemble des thèmes liés aux systèmes d'exploitation réels. Les aspects théoriques présentés dans ce document constituent une synthèse des nombreux ouvrages qui existent dans la littérature. Par contre, les aspects pratiques ont systématiquement fait l'objet d'un développement privilégié bien adapté au matériel utilisé dans nos laboratoires. En effet, ce document est composé de nombreux exemples de programmes qui contribuent à mieux comprendre les fonctionnalités des systèmes d'exploitation. Pour des raisons strictement pédagogiques, et eu égard à la complexité de la matière (l'importance de l'information, la multitude de concepts...) nous avons privilégié une approche de type didactique. C'est ainsi que nous présentons :

- Les concepts principaux et les techniques fondamentales des systèmes d'exploitation.
- Les modules de base d'un système d'exploitation moderne.
- Les interactions entre ces modules et les problèmes liés à leurs implantations ainsi que les solutions adoptées dans les systèmes modernes.

1. Afin de simplifier le texte, seul le masculin a été utilisé pour désigner toute personne sans distinction du genre.

Nous avons utilisé les systèmes d'exploitation Unix et GNU/Linux pour illustrer nos exemples. Nous suggérons aussi de lectures qui pourront enrichir la matière vue.

Ce document est un support pédagogique « vivant » aussi bien pour les enseignants que pour les étudiants. Il est, en effet, appelé à faire l'objet d'une actualisation permanente tant par les suggestions attendues des uns et des autres – à la fois, pour les améliorations à apporter et les exercices – que de la prise en compte de nouvelles réalités.

Il va de soi qu'un texte n'est jamais exempt d'erreurs, quand bien même on s'entoure de toutes les précautions pour les éviter. Nous serions donc très reconnaissants aux lecteurs de nous signaler toute erreur.

Montréal, automne 2002 | hiver 2003

Hanifa Boucheneb et Juan Manuel Torres-Moreno
{hanifa.boucheneb,juan-manuel.torres}@polymtl.ca

Contenu

Le matériel est divisé en plusieurs chapitres et sections. Tout au long du document, nous présentons des nombreux exemples clairs sur les applications et conséquences de chaque concept.

Le Chapitre 1 présente un aperçu rétrospectif des concepts et fonctionnalités des systèmes d'exploitation. Un système d'exploitation est un programme qui doit permettre aux utilisateurs d'utiliser les fonctionnalités d'un ordinateur. Il doit aussi aider le programmeur à développer des logiciels de la façon la plus efficace possible. Dans les grosses machines, il est toujours en exécution. Le système constitue donc une interface entre l'utilisateur et la machine physique.

Le Chapitre 2 introduit le système d'exploitation Unix et plus particulièrement son clone Linux. Le standard Posix est aussi introduit. La présentation des commandes, processus et fichiers est faite en utilisant des exemples de programmation shell.

Le Chapitre 3 défini le concept de processus, puisque tout le logiciel d'un ordinateur est organisé en un certain nombre de processus séquentiels. Les threads ou processus légers sont aussi introduits. De nombreux exemples sont présentés et programmés sous le standard Posix.

Le Chapitre 4 présente l'ordonnancement ou planification des processus. Dans un système multi-utilisateurs à temps partagé, plusieurs processus peuvent être présents en mémoire centrale en attente d'exécution. Si plusieurs processus sont prêts, le système d'exploitation doit gérer l'allocation du processeur aux différents processus à exécuter. On verra donc que c'est l'ordonnanceur qui s'acquitte de cette tâche.

Le Chapitre 5 traite sur les processus coopératifs pour résoudre un même problème. Ces processus s'exécutent en parallèle sur un ordinateur (monoprocesseur ou multiprocesseurs) ou sur des ordinateurs différents. Ils doivent s'échanger des informations en établissant une vraie communication interprocessus. Nous étudierons les plusieurs moyens de cette com-

munication : les variables communes, les fichiers communs, les signaux, les messages et les tubes de communication.

Dans un système d'exploitation multiprogrammé en temps partagé, plusieurs processus s'exécutent en parallèle et partagent des objets comme la mémoire, les imprimantes, et d'autres. Le partage d'objets sans précaution particulière peut conduire à des résultats imprévisibles. La solution au problème s'appelle synchronisation des processus, thème qui sera abordé au Chapitre 6.

Les paquets de communication entre processus (Inter Process Communication) IPC qui offre le System V sont composés de trois mécanismes fondamentaux : les sémaphores, la mémoire partagée et les files de messages. Bien que la communication IPC ne soit pas fondée sur les descripteurs de fichiers, il existe de nombreuses applications qu'en utilisent ces techniques puissantes. Le Chapitre 7 est donc consacré à la communication IPC.

L'exécution d'un processus nécessite un ensemble de ressources qui lui sont attribuées par le système d'exploitation. L'utilisation d'une ressource passe par les étapes : demande, utilisation et libération. Des problèmes peuvent survenir lorsque les processus obtiennent des accès *exclusifs* aux ressources. Par exemple, un processus P_1 détient une ressource R_1 et attend une autre ressource R_2, utilisée déjà par un autre processus P_2, qui détient la ressource R_2 et attend la ressource R_1. On a une situation d'interblocage car P_1 attend P_2 qui attend P_1. Les deux processus vont attendre indéfiniment. Ce thème sera traité au Chapitre 8.

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. La nécessité de gérer la mémoire 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. Nous verrons dans le Le Chapitre 9 qu'il y a plusieurs schémas pour la gestion de la mémoire, avec leurs avantages et inconvénients. D'un autre côté, la mémoire virtuelle est une technique qui permet d'exécuter des programmes dont la taille excède la taille de la mémoire réelle. Avec des techniques adéquates, comme on verra dans le Chapitre 10, sa taille peut être très supérieure à celle de la mémoire physique.

Un fichier désigne un ensemble d'informations stockées sur le disque. Le système de fichiers est la partie du système d'exploitation qui se charge de gérer les fichiers. La gestion consiste en la création, la suppression, les accès en lecture et en écriture, le partage de fichiers et leur protection en

contrôlant les accès. Ce thème sera consacre au Chapitre 11.

Les fichiers de données sur les disques se répartissent dans des blocs de taille fixe. La lecture ou l'écriture d'un élément d'un fichier impliquera donc le transfert du bloc entier qui contient cet élément. Pour un accès rapide, on aura donc intérêt à implanter des techniques adéquates de stockage et le partage de fichiers. Le Chapitre 12 traite ces questions.

Nous verrons dans le Chapitre 13, que les ordinateurs disposent de périphériques d'Entrées / Sorties (terminaux, disques, imprimantes, équipements de communication, etc.). Le système d'exploitation peut considérer les périphériques comme des fichiers spéciaux, et donc intègre leur gestion au cœur du système de fichiers.

Finalement, le Chapitre 14 se veut un aperçu à la réponse de pourquoi les systèmes d'exploitation distribués ? Nous verrons que le développement de microprocesseurs puissants et l'invention des réseaux locaux à haut débit ont permis de connecter des plusieurs ordinateurs pour échanger de de l'information. Ces ordinateurs connectés par un réseau de communication constituent un système distribué. Ainsi, des concepts logiciel et matériel seront présentés. La programmation des sockets sera présentée en détail comme brique de base de la communication distribuée.

Table des matières

1 Introduction **25**

1.1 Qu'est ce qu'un système d'exploitation? 25
1.1.1 Le système d'exploitation 26
1.2 Évolution des systèmes d'exploitation 27
1.3 Interactions utilisateur / système 32
1.4 Appels système 33
1.4.1 Les processus 33
1.4.2 Les fichiers 34
1.5 Structure d'un système d'exploitation 35
1.5.1 Structure en couches 35
1.5.2 Structure monolithique 36
1.5.3 Micro-kernel 37
1.5.4 Le modèle client / serveur 37
1.5.5 Machines virtuelles 38
1.6 Exercices 40

2 Introduction au système Unix/Linux **43**

2.1 Bref historique 43
2.2 Caractéristiques 44
2.2.1 Structure d'un système Unix / Linux 45
2.3 Début de session 47
2.3.1 Interpréteurs de commandes 48
2.4 Commandes utiles d'Unix / Linux 48
2.5 Scripts shell 53
2.6 Editeurs d'Unix 60
2.7 Utilisateurs 61
2.8 Fichiers et répertoires 61
2.9 Processus 63
2.10 Exercices 68

3 Processus et threads **71**
3.1 Processus 71
3.1.1 États d'un processus 71
3.1.2 Implantation de processus 72
3.1.3 Les démons 74
3.1.4 Création et terminaison de processus 74
3.2 Services Posix pour la gestion de processus 75
3.3 Création de processus 75
3.3.1 Création de processus avec `system()` 75
3.3.2 Création de processus avec `fork()` 76
3.3.3 La famille des appels système `exec` 83
3.3.4 Les appels système `wait()`, `waitpid()` et `exit()` 86
3.3.5 Processus zombie 88
3.3.6 Commentaire sur `system()` 90
3.4 Threads 95
3.4.1 Avantages 96
3.4.2 Threads utilisateur et noyau 97
3.5 Services Posix de gestion de threads 97
3.6 Threads sous Unix (Solaris 2) 106
3.7 Exercices 111

4 Ordonnancement des processus **113**
4.1 Introduction 113
4.2 Types d'ordonnanceurs 114
4.3 Objectifs de l'ordonnanceur d'un système multi-utilisateur . 116
4.4 Ordonnanceurs non préemptifs 116
4.5 Ordonnanceurs préemptifs 119
4.5.1 Ordonnancement du plus petit temps de séjour . . . 120
4.5.2 Ordonnancement circulaire 120
4.5.3 Ordonnancement avec priorité 124
4.5.4 Files multiples (quantum variable) 125
4.6 Ordonnancement à deux niveaux 126
4.7 Ordonnancement des threads 126
4.8 Cas d'étude 127
4.8.1 MS-DOS 127
4.8.2 Unix 128
4.8.3 Linux 129
4.9 Exercices 132

5 Communication interprocessus 135
5.1 Variables et fichiers communs 135
5.2 Les signaux 136
5.2.1 Services de signaux 137
5.3 Les messages 147
5.3.1 Tubes de communication 147
5.3.2 Tubes sans nom 147
5.3.3 Tubes de communication nommés 154
5.4 Sockets 158
5.5 Exercices 160

6 Synchronisation des processus 165
6.1 Introduction 165
6.2 Objets et sections critiques 166
6.3 Masquage des interruptions 168
6.4 Exclusion mutuelle par attente active 169
6.4.1 Les variables de verrouillage 169
6.4.2 L'alternance 170
6.4.3 Solution de Peterson 171
6.4.4 L'instruction TSL 173
6.4.5 Commentaires sur l'attente active 174
6.5 Primitives SLEEP et WAKEUP 175
6.6 Problème du producteur et du consommateur 176
6.6.1 Critique des solutions précédentes 178
6.7 Sémaphores 178
6.8 Services Posix sur les sémaphores 179
6.8.1 Problème des philosophes 184
6.8.2 Problème des rédacteurs et des lecteurs 189
6.9 Compteurs d'événements 194
6.9.1 Producteur / consommateur 194
6.10 Moniteurs 196
6.10.1 Producteur / consommateur 196
6.11 Échanges de messages 197
6.12 Producteur / consommateur par tubes 197
6.13 Exclusion mutuelle (mutex) de threads 199
6.14 Sémaphores avec System V (Solaris) 204
6.15 Exercises 213

7 Communications IPC System V **219**
7.1 Files de messages 219
7.1.1 Génération des clés 220
7.1.2 Fonctions de manipulation de files 221
7.1.3 Monitorisation 226
7.2 Structure de messages 227
7.2.1 Buffer de message 227
7.2.2 `msg` du noyau 228
7.2.3 `ipc_perm` du noyau 229
7.3 Mémoire partagée 236
7.4 Sémaphores 240
7.5 Exercices 249

8 Interblocages **251**
8.1 Les processus et les ressources 251
8.2 Définition d'un interblocage 252
8.2.1 Conditions nécessaires pour l'interblocage 254
8.3 Graphe d'allocation des ressources 255
8.3.1 Réduction du graphe d'allocation des ressources . . . 256
8.4 Traitement des interblocages 257
8.5 La détection et la reprise 258
8.5.1 Algorithme de détection des interblocages 259
8.5.2 La reprise des interblocages 260
8.6 L'évitement des interblocages 261
8.6.1 Algorithme du banquier 261
8.7 La prévention des interblocages 262
8.8 Exercices 264

9 Gestion de la mémoire **269**
9.1 Introduction 269
9.1.1 Rappels sur la compilation 269
9.1.2 La gestion de la mémoire 271
9.2 Système monoprogrammé 272
9.2.1 Machine nue 272
9.2.2 Moniteur résident 272
9.3 Système multiprogrammé 274
9.3.1 Translation des adresses 275
9.4 Swapping 276
9.5 Partitions fixes sans va-et-vient 277
9.5.1 Registres limite 278

9.5.2 Registres de base et limite 278
9.5.3 Fragmentation 280
9.5.4 Files d'entrée 280
9.6 Partitions variables et va-et-vient 283
9.6.1 Gestion de la mémoire par table de bits 286
9.6.2 Gestion de la mémoire par liste chaînée 287
9.6.3 Algorithme utilisant deux listes séparées 289
9.6.4 Gestion de la mémoire par subdivision 289
9.7 Allocation d'espace de va-et-vient 290
9.8 Projection de fichiers en mémoire 291
9.9 Services Posix de projection de fichiers 291
9.9.1 Projection d'un fichier 292
9.9.2 Libération de la projection 292
9.10 Note sur MS-DOS 295
9.11 Exercices 296

10 Mémoire virtuelle 299
10.1 Introduction 299
10.2 Pagination pure 301
10.2.1 Unité de gestion de la mémoire (MMU) 302
10.2.2 Accès à la table de pages 306
10.3 Algorithmes de remplacement de page 308
10.3.1 Remplacement de page optimal 308
10.3.2 Remplacement de la page non récemment utilisée . . 309
10.3.3 Remplacement de page FIFO 309
10.3.4 Remplacement de page de l'horloge 310
10.3.5 Remplacement de la page la moins récemment utilisée 311
10.4 Nombre de cases allouées à un processus 311
10.4.1 Anomalie de Belady 311
10.5 Écroulement du système 312
10.6 Retour sur instructions 312
10.7 Segmentation 312
10.8 Segmentation paginée 313
10.9 Mémoire cache 314
10.10 Cas d'étude 314
10.10.1 Unix 314
10.10.2 Linux 316
10.10.3 MS-DOS 320
10.11 Exercises 321

11 Système de fichiers **323**
11.1 Introduction 323
11.1.1 Qu'est ce qu'un fichier? 323
11.1.2 Cycle de vie d'un fichier 324
11.1.3 Types de fichiers 324
11.2 Accès aux fichiers 325
11.2.1 Attributs des fichiers 325
11.2.2 i-nœuds 326
11.3 Services Posix sur les fichiers 326
11.3.1 Transfert rapide de données 344
11.4 Répertoires 346
11.5 Services Posix sur les répertoires 346
11.6 Répertoires sur Unix System V 350
11.7 Exercises 353

12 Stockage et partage de fichiers **355**
12.1 Stockage des fichiers 355
12.1.1 Allocation contiguë 355
12.1.2 Liste chaînée 356
12.1.3 Table d'i-nœuds 357
12.2 Le système de fichiers `/proc` 358
12.3 Partage de fichiers 361
12.3.1 Liens physiques 362
12.3.2 Liens symboliques 364
12.4 Verrouillage de fichiers 369
12.4.1 Verrouillage d'une partie de fichier 369
12.4.2 Services Posix de verrouillage de fichiers 370
12.4.3 Verrouillage avec `flock()` 375
12.5 Antémémoire 378
12.6 Cohérence du système de fichier 379
12.6.1 Au niveau des blocs 379
12.6.2 Au niveau des fichiers 380
12.7 Protection 380
12.8 Exercices 382

13 Périphériques d'Entrées/Sorties **385**
13.1 Périphériques 385
13.2 Le répertoire `/dev` 387
13.3 Les contrôleurs 389
13.4 L'accès direct à la mémoire 391

13.5 Cas du disque 391
13.5.1 Planification de disque 392
13.6 Exercices 396

14 Introduction aux systèmes distribués 399
14.1 Introduction 399
14.2 Concepts matériels 399
14.3 Concepts logiciels 403
14.4 Communication dans les systèmes distribués 406
14.5 Sockets 410
14.5.1 Appels système 416
14.5.2 Sockets ou tubes? 418
14.5.3 Sockets locales 419
14.5.4 Sockets dans le domaine Internet 422

A Opérateurs et variables shell 435

B Appels système 441

C Signaux 449

D Sites Internet 453

Table des figures

1.1 Les couches d'un système informatique. 26
1.2 Traitement par lots et organisation de la mémoire. 28
1.3 Utilisation du processeur et des périphériques. 28
1.4 Partitions de mémoire. 29
1.5 Commutation du processeur. 30
1.6 Temps partagé. 31
1.7 Structure en couches. 35
1.8 Structure monolithique 36
1.9 Structure de micro-kernel. 37

2.1 Evolution d'Unix. 44
2.2 Standard Posix 45
2.3 Architecture d'un système Unix. 45
2.4 Architecture du kernel BSD d'Unix. 46
2.5 Architecture de Linux. 47
2.6 Enchaînement parallèle des commandes cat et sort. 52
2.7 Aperçu des répertoires dans Unix/Linux. 63
2.8 Espace d'adressage. 64
2.9 Processus dans Unix/Linux. 66

3.1 Les états principaux d'un processus. 71
3.2 Modèle à cinq états d'un processus. 72
3.3 Principales entrées de PCB d'Unix. 73
3.4 Le changement de contexte. 74
3.5 Processus père et fils. 77
3.6 Services Posix : `fork()`. 78
3.7 Arbre de création de processus. 83
3.8 Services Posix : `exec`. 84
3.9 Processus zombie. 88

3.10 Processus orphelin. 89
3.11 Threads vs. processus. 96
3.12 Hiérarchie de processus légers. 98

4.1 Modèle des états actifs et non actifs. 115
4.2 États d'un processus sous Unix / Linux. 116
4.3 Ordonnancement circulaire. 121
4.4 Exemple d'ordonnancement. 123
4.5 Ordonnancement avec priorité. 124
4.6 Ordonnancement des threads. 127
4.7 Ordonnancement des threads dans Linux. 127
4.8 Ordonnancement sous Unix. 128

5.1 Processus en paralèlle reliés par un tube de communication. 148
5.2 Pipe de Lecture / Écriture. 149
5.3 Communication interprocessus. 150
5.4 Descripteurs de fichiers pendant la communication. 152
5.5 Communication interprocessus bidirectionnelle. 154

6.1 Spool d'une imprimante. 165
6.2 Somme de N nombres avec threads. 167
6.3 Problème du producteur / consommateur. 177
6.4 Exclusion mutuelle avec des sémaphores. 178
6.5 Exclusion mutuelle d'une section critique composée d'une variable a. 179
6.6 Sémaphores. 179
6.7 Problème des philosophes. 185
6.8 Problème des lecteurs et écrivains. 190
6.9 Buffer circulaire. 215

7.1 Files de messages. 220

8.1 Interblocage de deux processus. 253
8.2 Problème de circulation routière. 254
8.3 Besoins de trois processus. 255
8.4 Situation d'interblocage de trois processus. 256
8.5 Graphe d'allocation des ressources. 257
8.6 Graphe réduit d'allocation. 258
8.7 Graphe d'allocation des ressources. 264
8.8 Graphe d'allocation des ressources. 267

9.1 Phases de la compilation. 270
9.2 Machine nue. 272
9.3 Moniteur résident. 273
9.4 Organisation de la mémoire du PC sous MS-DOS. 273
9.5 Registre limite. 274
9.6 Chargement des programmes depuis les hautes positions. . 276
9.7 Registres limite. 276
9.8 Swapping. 277
9.9 Registres limite et mémoire. 278
9.10 Registres limite. 279
9.11 Registres de base et limite. 279
9.12 Fragmentation interne et externe. 280
9.13 Files d'entrée. 281
9.14 Configuration de file d'entrée unique. 282
9.15 Configuration de files multiples. 282
9.16 Exemple MVT (1). 284
9.17 Exemple MVT (2). 285
9.18 Compactage de mémoire. 286
9.19 Table de bits. 286
9.20 Bitmapt de l'état de mémoire. 287
9.21 a) État de mémoire. b) Liste chaînée. 288
9.22 Espace de va-et-vient. 290
9.23 Mémoire dans MS-DOS. 295

10.1 Espace des adresses virtuelles et espace physique. 300
10.2 Pagination pure. 301
10.3 Espace virtuel et espace physique. 302
10.4 Unité de gestion de la mémoire *MMU*. 303
10.5 Entrée d'une Table de pages. 303
10.6 Opérations du *MMU*. 304
10.7 Table de pages à deux niveaux. 305
10.8 Mémoire associative. 307
10.9 Algorithme optimale de Belady. 309
10.10 Algorithme de remplacement FIFO. 310
10.11 Algorithme de l'horloge. 310
10.12 Algorithme de remplacement MRU. 311
10.13 Segmentation simple. 313
10.14 Segmentation paginée. 314
10.15 Hiérarchie de la mémoire. 315
10.16 Espace mémoire sous Linux. 317

10.17 Segmentation paginée du 80x86 sous Linux (1). 317
10.18 Traslation d'adresse logique → adresse linéaire. 318
10.19 Segmentation paginée du 80x86 sous Linux (2). 318

11.1 Le cycle de vie d'un fichier. 324
11.2 Structure d'un i-nœud. 326
11.3 Pointeur de fichier : `lseek()`. 332

12.1 Liste chaînée. 356
12.2 Indirections d'un i-nœud. 357
12.3 i-nœud et super-bloc. 358
12.4 Partition EXT2 de Linux. 358
12.5 Fichier partagé. 362

13.1 Fichiers spéciaux et dispositifs. 386
13.2 SSTF. 393
13.3 SCAN. 394
13.4 Parcours de FCFS. 394
13.5 Parcours de SSTF. 395
13.6 Parcours de SCAN. 395
13.7 Parcours de C-SCAN. 395

14.1 Mémoire cache entre processeurs. 400
14.2 Multiprocesseurs commutés. 401
14.3 Réseau commuté omega. 402
14.4 Multicalculateurs à bus. 402
14.5 Réseau NFS. 404
14.6 Modèle ISO. 407
14.7 Architecture TCP / IP. 408
14.8 Modèle Client-serveur concurrent. 411
14.9 Modèle Client-serveur connecté. 413
14.10 Modèle Client-serveur non connecté. 414

Listings

2.1 read.sh . . . 56
2.2 mon-cat.sh . . . 56
2.3 hola.sh . . . 56
2.4 hda.sh . . . 56
2.5 while.sh . . . 57
2.6 rapport.sh . . . 57
2.7 inv.sh . . . 58
2.8 affichage.sh . . . 58
3.1 system.c . . . 76
3.2 fils.c . . . 77
3.3 chaine.c . . . 78
3.4 tfork.cpp . . . 79
3.5 fork1.c . . . 81
3.6 fork2.c . . . 81
3.7 fork-exec.cpp . . . 85
3.8 zombie.c . . . 89
3.9 parent.cpp . . . 91
3.10 fils.cpp . . . 91
3.11 texecvp.cpp . . . 92
3.12 deuxfils-1.c . . . 93
3.13 deuxfils-2.c . . . 94
3.14 thread-pid.c . . . 99
3.15 threads.cpp . . . 100
3.16 thread-param.c . . . 102
3.17 suma-mat.c . . . 104
3.18 threads.cpp . . . 107
3.19 threads2.cpp . . . 108
4.1 scheduler.c . . . 130
5.1 signaux-0.c . . . 138

5.2 signaux-1.cpp 140
5.3 sigint.c 142
5.4 sigint-2.c 143
5.5 signaux-2.c 143
5.6 ping.c 145
5.7 upipe.c 150
5.8 pipecom.cpp 152
5.9 writer.cpp 156
5.10 reader.cpp 156
5.11 fils-tube.c 162
6.1 peterson.c 171
6.2 peterson2.c 172
6.3 schema-prod-cons.c 177
6.4 sem-posix.c 180
6.5 prod-cons.cpp 182
6.6 philosophe.c 185
6.7 philosophe2.c 187
6.8 red-lec.c 191
6.9 p-c-pipe.c 198
6.10 sum.c 200
6.11 sum-thread.c 200
6.12 mutex-threads2.c 202
6.13 v-semaphore.c 204
6.14 v-prod-cons.c 206
6.15 v-philosophe.c 208
6.16 v-philosophe2.c 210
7.1 cle.c 222
7.2 schema.c 225
7.3 messages.c 229
7.4 file-w.c 231
7.5 file-r.c 232
7.6 kirk.c 233
7.7 spock.c 234
7.8 shm.c 236
7.9 mem.h 237
7.10 exemem.cc 239
7.11 posix-v-sem.c 243
7.12 ecrivain.c 245
7.13 lecteur.c 247
9.1 mmap1.c 293

11.1 copie-std.c 330
11.2 copie.c 330
11.3 seek.c 333
11.4 dup.c 335
11.5 lseek.c 336
11.6 dup-lseek.c 337
11.7 dup3.c 338
11.8 stat.c 339
11.9 estado.c 340
11.10 unlink-cs.c 343
11.11 fcopie.c 345
11.12 liste-dir.c 349
11.13 getd.c 351
12.1 env.c 359
12.2 sym.c 365
12.3 copie-mem.c 367
12.4 partage.c 371
12.5 verrou.c 372
12.6 lock.c 374
12.7 flock.c 376
13.1 sound.c 390
14.1 schema-i.c 411
14.2 schema-c.c 412
14.3 sock-client.c 419
14.4 sock-server.c 420
14.5 gethost.c 422
14.6 sock-inet.c 423
14.7 suma-client.c 426
14.8 suma-server.c 427
14.9 time-client.c 428
14.10 time-server.c 430

1
Introduction

UN système d'exploitation est un programme qui doit permettre aux utilisateurs d'utiliser les fonctionnalités d'un ordinateur. Mais le système d'exploitation doit aussi aider le programmeur à développer des logiciels de la façon la plus efficace possible. Un système d'exploitation est mis en route dès qu'on met en marche l'ordinateur. Dans les grosses machines, il est toujours en exécution. Le système constitue donc une interface entre l'utilisateur et la machine physique.

1.1 Qu'est ce qu'un système d'exploitation ?

Malgré les différences de point de vue, de forme, de taille et de type, les ordinateurs se composent de matériel et de logiciels, comme illustré à la figure 1.1. Un ordinateur sans logiciels n'est que de la ferraille plutôt chère. Le système d'exploitation est donc une composante logiciel très importante.

Le matériel d'un système informatique sont les processeurs qui exécutent les instructions, la mémoire centrale qui sauvegarde les données et les instructions à exécuter, la mémoire secondaire qui sauvegarde les informations, les périphériques d'Entrées / Sorties (clavier, souris, écran, modem, etc.) pour introduire ou récupérer des informations. Les logiciels sont à leur tour divisés en **programmes système** qui font fonctionner l'ordinateur : le **système d'exploitation** et les utilitaires (compilateurs, éditeurs, interpréteurs de commandes, etc.) et en **programmes d'application** qui résolvent des problèmes spécifiques des utilisateurs.

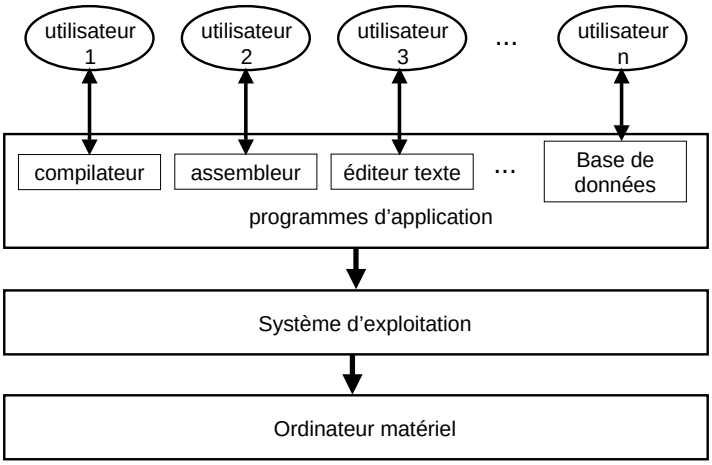


FIG. 1.1 – *Les couches d'un système informatique.*

1.1.1 Le système d'exploitation

Le **système d'exploitation** gère et contrôle les composants de l'ordinateur. Il fournit une base appelée **machine virtuelle**, sur laquelle seront construits les programmes d'application et les utilitaires au moyen des **services** ou **appels système**. Le but d'un système d'exploitation consiste donc à *développer des applications sans se soucier des détails de fonctionnement et de gestion du matériel.* Ainsi, par exemple, on peut effectuer la lecture d'un fichier par un simple appel système: `read(fd, buf, 255);` et peu importe que le fichier en question se trouve dans un disque magnétique, un DVD ou un disque en RAM.

Les fonctions d'un système d'exploitation

Les systèmes d'exploitation modernes sont constitués de centaines de milliers, voire de millions de lignes de code et requierent des milliers d'homme-années de travail. Ils ont comme fonctions principales :

- Chargement et lancement des programmes
- Gestion des processeurs, de la mémoire, des périphériques
- Gestion des processus (programmes en cours d'exécution) et des fichiers
- Protection contre les erreurs et la détection des erreurs, etc.

Exemples de systèmes d'exploitation

Il y a plusieurs types de systèmes d'exploitation actuellement :

– MS-DOS, Windows
– OS/2, Mac-OS
– Unix (AIX, Xenix, Ultrix, Solaris, etc.)
– Linux

Il y a aussi des systèmes d'exploitation éducatifs, comme Minix [1] crée par Andrew S. Tanembaum ou des simulateurs de systèmes d'exploitation comme Nachos [2].

1.2 Évolution des systèmes d'exploitation

Exploitation porte ouverte : 1945-1955

Les machines à calculer étaient construites au moyen de tubes électroniques. Les machines étaient énormes, coûteuses, très peu fiables et beaucoup moins rapides car le temps de cycle se mesurait en secondes. Elles n'avaient pas de systèmes d'exploitation. Les programmes étaient écrits directement en langage machine : ils étaient chargés en mémoire, exécutés et mis au point à partir d'un pupitre de commande. Le mode de fonctionnement consistait à réserver une tranche de temps pour chaque programmeur (un seul programme en mémoire). Au début de 1950, la procédure s'est améliorée grâce à l'introduction de cartes perforées. Les problèmes évidents étaient dus au mode d'exploitation : peu économique, le matériel coûteux, la longue durée d'entrée des programmes et une mise au point pénible.

Traitement par lots : 1955-1965

Les machines étaient construites au moyen de transistors et dotées d'unités de bandes magnétiques. Elles étaient plus fiables mais toujours énormes (enfermées dans des salles climatisées). Les programmes étaient écrits en Fortran ou en assembleur sur des cartes perforées. Le mode de fonctionnement utilisé, montré à la figure 1.2, était le traitement par lots qui consiste à :

– Transférer les travaux sur une bande magnétique

1. http://www.cs.vu.nl/~ast/minix.html
2. http://www.cs.washington.edu/homes/tom/nachos/

– Monter la bande sur le lecteur de bandes
– Charger en mémoire un programme spécial (l'ancêtre des systèmes d'exploitation) qui lit puis exécute, l'un à la suite de l'autre, les programmes de la bande. Les résultats sont récupérés sur une autre bande, à la fin de l'exécution de tout le lot.
– Imprimer les résultats.

Ce mode d'exploitation nécessitait deux machines dont la plus puissante était réservée aux calculs et l'autre, moins chère, s'occupait des périphériques lents. Le problème est que le processeur restait inutilisé pendant les

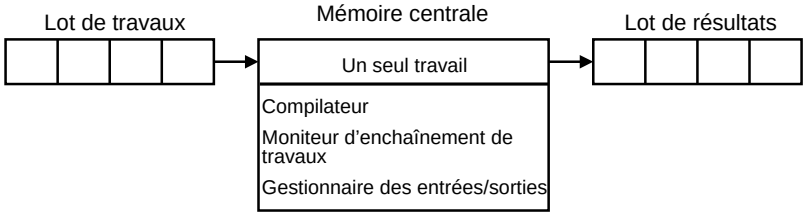


FIG. 1.2 – *Traitement par lots et organisation de la mémoire.*

opérations d'entrée et sortie (E/S), comme on le voit sur la figure 1.3. Le temps d'attente des résultats était trop long et en plus il n'y avait pas d'interaction avec l'utilisateur.

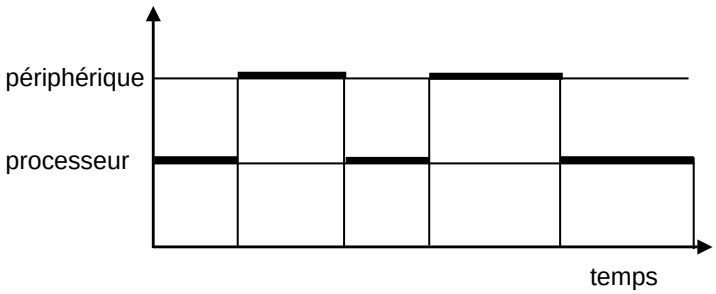


FIG. 1.3 – *Utilisation du processeur et des périphériques.*

Multiprogrammation et traitement par lots : 1965-1980

L'introduction des circuits intégrés dans la construction des machines a permis d'offrir un meilleur rapport coût / performance. L'arrivée sur le marché des unités de disques, qui offrent l'accès aléatoire et des capacités de

stockage importantes, a contribué à une série de développements des systèmes, notamment la possibilité de transférer les travaux vers le disque dès leur arrivée dans la salle machine. Cette technique s'appelle le *spool* (*Simultaneous Peripheral Operation On Line*) et est également utilisée pour les sorties. Cette notion de *spooling* ou traitement par lots subsiste dans les systèmes d'exploitation modernes.

Tous les travaux résidants sur le disque en attente d'exécution sont conservés dans le *pool* des travaux en entrée. La mémoire est organisée en un ensemble de n partitions (figure 1.4). Chaque partition peut contenir au plus un travail. Le système d'exploitation réside aussi dans une partition. S'il y a une partition libre et des travaux dans le *pool* d'entrée, le système

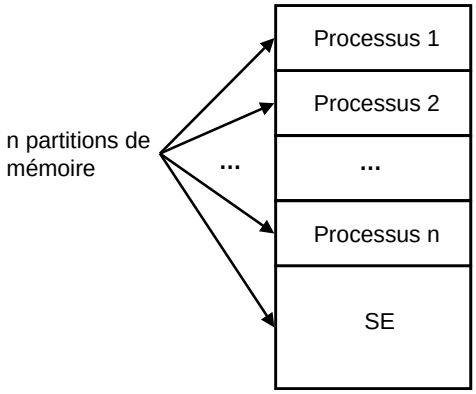


FIG. 1.4 – *Partitions de mémoire.*

d'exploitation choisit un travail puis lance son chargement en mémoire. Il conserve en mémoire plusieurs travaux et gère le partage du processeur entre les différents travaux chargés en mémoire (la multiprogrammation). Le processeur est alloué à un travail jusqu'à ce qu'il demande une E/S (premier arrivé, premier servi). Lorsqu'un travail demande une E/S (en attente de la fin d'une E/S), le processeur est alloué à un autre travail en mémoire (le suivant). A la fin d'une E/S, une interruption se produit et le système d'exploitation reprend le contrôle pour traiter l'interruption et lancer ou poursuivre l'exécution d'un travail. Dès qu'un travail se termine, le système d'exploitation peut lancer le chargement, à partir du disque, d'un nouveau travail dans la partition qui vient de se libérer.

Supposons trois travaux **A**, **B** et **C**. Dans un système multiprogrammé, trois activités peuvent être donc menées en parallèle : 1. Le chargement du travail **C** en mémoire, 2. L'exécution du travail **B** et 3. L'édition des résultats

du travail **A**.

La multiprogrammation nécessite des circuits de contrôle pour protéger chaque travail contre les intrusions et les erreurs des autres. Les ordinateurs de cette époque possédaient ces circuits.

▶ **Exemple 1.** Considérons les travaux **A**, **B** et **C** en mémoire, qui sont montrés sur la figure 1.5. L'exécution de **A** est entamée en premier puis lorsqu'il

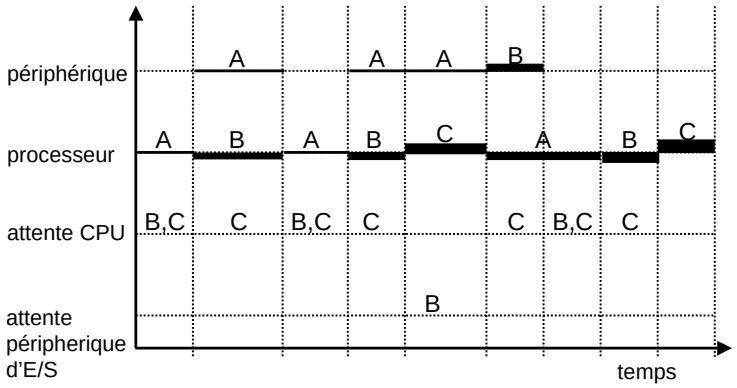


FIG. 1.5 – *Commutation du processeur.*

demande une Entrée/Sortie (E/S), le processeur commute sur **B**. A la fin de l'E/S demandée par **A**, le processeur suspend l'exécution de **B** pour commuter sur **A**. On suppose qu'**A** est prioritaire. Après un certain temps de calcul, **A** demande de nouveau une E/S; ce qui provoque la commutation du processeur sur **B**. Durant l'exécution de l'E/S de **A**, le travail **B** demande une E/S. Il se met donc en attente car le périphérique est occupé. Le processeur commute alors sur le travail **C**. Lorsque l'exécution de l'E/S demandée par **A** s'est terminée, le processeur commute sur **A** et le traitement de la demande d'E/S du travail **B** est entamé par le périphérique d'E/S.

Multiprogrammation et partage de temps : 1965-1980

Le désir d'un temps de réponse plus rapide et d'interactivité de l'exploitation a introduit la technique de partage de temps (systèmes temps partagé ou multi-utilisateurs) : plusieurs utilisateurs peuvent se connecter à la machine par l'intermédiaire de leurs terminaux et travailler en même temps.

Le processeur est alloué, à tour de rôle, pendant un certain temps à chacun des travaux en attente d'exécution. Au bout de ce temps, même si le travail en cours ne s'est pas terminé, son exécution est suspendue. Le processeur est alloué à un autre travail. Si plusieurs utilisateurs lancent à partir de leurs terminaux leurs programmes simultanément, ce mode d'exploitation donne l'impression que les programmes s'exécutent en parallèle. Le cas de trois travaux A, B et C est montré sur la figure 1.6. Les temps de réponse pour chaque utilisateur sont acceptables.

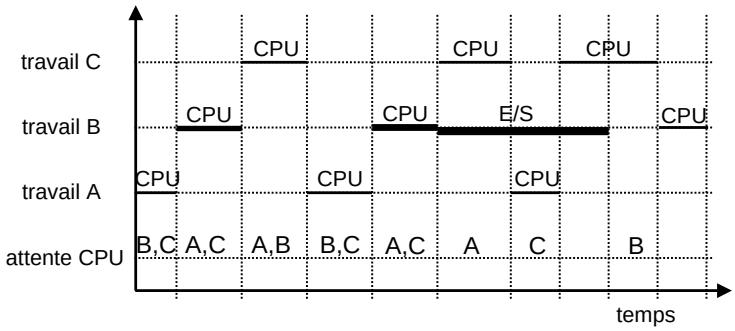


FIG. 1.6 – *Temps partagé.*

Systèmes d'exploitation d'ordinateurs personnels

Ces systèmes d'exploitation mono-utilisateur ne mettaient pas l'accent sur l'usage du processeur, ni sur la protection. Leur but était de fournir une interface conviviale et une rapidité de réaction. Ensuite, les fonctionnalités des gros systèmes ont été peu à peu transférées vers les microordinateurs.

Exploitation en réseau

Les réseaux d'ordinateurs personnels qui fonctionnent sous des systèmes d'exploitation en réseau permettent de se connecter sur une machine distante et de transférer des fichiers d'une machine à une autre. Ces systèmes d'exploitation requièrent une interface réseau, un logiciel de contrôle de bas niveau, ainsi que des programmes qui permettent une connexion distante et un accès aux fichiers qui se trouvent sur les différentes machines.

Exploitation en distribué

Les réseaux d'ordinateurs qui fonctionnent sous des systèmes d'exploitation distribués apparaissent aux yeux des utilisateurs comme une machine monoprocesseur, même lorsque ce n'est pas le cas. Le système d'exploitation distribué gère et contrôle l'ensemble des composants de tous les ordinateurs connectés (les processeurs, les mémoires, les disques, etc.).

Systèmes multiprocesseurs

Ces systèmes sont composés de plusieurs processeurs reliés au bus de l'ordinateur. Ils se caractérisent par leur capacité de traitement et leur fiabilité : la panne d'un processeur n'arrêtera pas le système

Système d'exploitation temps réel

Ce sont des systèmes spécialisés dans la conduite d'appareillage industriel ou dans la commande de processus où le temps joue un rôle critique (des contraintes temporelles strictes ou souples à respecter).

1.3 Interactions utilisateur/système

Pour un utilisateur, le système d'exploitation apparaît comme un ensemble de procédures, trop complexes pour qu'il les écrive lui-même. Les bibliothèques des **appels système** sont alors des procédures mises à la disposition des programmeurs. Ainsi un programme *C/C++* peut utiliser des appels système d'Unix/Linux comme `open()`, `write()` et `read()` pour effectuer des Entrées/Sorties de bas niveau.

L'**interpréteur de commandes** constitue une interface utilisateur/système. Il est disponible dans tous les systèmes. Il est lancé dès la connexion au système et invite l'utilisateur à introduire une commande. L'**interpréteur de commandes** récupère puis exécute la commande par combinaison d'appels système et d'outils (compilateurs, éditeurs de lien, etc.). Il affiche les résultats ou les erreurs, puis se met en attente de la commande suivante. Par exemple, la commande de l'interpréteur (shell) d'Unix suivante permet d'afficher à l'écran le contenu du fichier `archive` : `cat archive`[3]

L'introduction du graphisme dans les interfaces utilisateur a révolutionné le monde de l'informatique (attendons celle du son !). L'interface

3. La commande équivalente sous MS-DOS est : `type archive`.

graphique a été rendue populaire par le Macintosh de Apple. Elle est maintenant proposée pour la plupart des machines.

1.4 Appels système

En général, les processeurs ont deux modes de fonctionnement, le **mode superviseur** et le **mode utilisateur** :

- Le **mode superviseur** (noyau, privilégié ou maître) : pour le système d'exploitation, où toutes les instructions sont autorisées.
- Le **mode utilisateur** (esclave) : pour les programmes des utilisateurs et les utilitaires, où certaines instructions ne sont pas permises.

Ces modes de fonctionnement assurent la protection du système d'exploitation contre les intrusions et les erreurs. Ce n'est pas le cas des systèmes mono-utilisateur comme MS-DOS ou MacOS qui ont un seul mode de fonctionnement : le mode utilisateur. Ils ne sont pas protégés et donc peu fiables.

Un **appel système** consiste en une interruption logicielle (instruction TRAP) qui a pour rôle d'activer le système d'exploitation. Il a pour but : changer de mode d'exécution pour passer du mode utilisateur au mode maître, récupérer les paramètres et vérifier la validité de l'appel, de lancer l'exécution de la fonction demandée, de récupérer la (les) valeur(s) de retour et de retourner au programme appelant avec retour au mode utilisateur.

La table 1.1 énumère quelques appels système du système Unix conformes à la norme Posix[4]. Les appels système manipulent divers objets gérés par le système d'exploitation. Les processus et les fichiers sont les plus importants de ces objets.

1.4.1 Les processus

Un **processus** est un programme en cours d'exécution. Il est composé d'un **programme exécutable** (code), un **compteur ordinal** (**co**), un ensemble de données, une pile d'exécution et autres registres et informations nécessaires à l'exécution.

Les appels systèmes permettent notamment la création et l'arrêt des processus. Un processus peut créer un ou plusieurs processus fils qui, à leur tour, peuvent créer des processus fils dans une structure arborescente.

4. Posix est un ensemble de procédures standard d'Unix qui seront vues toute au long du texte.

Appel système	Description
fork	Créer un processus
waitpid	Attendre la terminaison d'un processus
wait	
execve	exécuter un autre programme
exit	Terminer l'exécution
open	Créer ou ouvrir un fichier
close	Fermer un fichier
read	Lecture de données
write	Écriture de données
Iseek	Pointeur dans un fichier
stat	Obtenir l'état des attributs
mkdir	Créer un répertoire
rmdir	Éliminer un répertoire
link	Liens vers un fichier
unlink	Éliminer un fichier
mount	Monter un système de fichiers
umount	Démonter un système de fichiers
chdir	Changer de répertoire
chmod	Changer les permissions d'accès
kill	Signaux
time	Obtenir le temps

TAB. 1.1 – *Quelques appels système Posix.*

Les processus peuvent se synchroniser et communiquer entre eux. Actuellement, on utilise de plus en plus le concept de **processus légers** (on parlera des processus et des processus légers dans le Chapitre 3). Les processus légers sont un moyen de raffiner et de diviser le travail normalement associé à un processus.

1.4.2 Les fichiers

Un **fichier** est un ensemble de données enregistrées de façon à être lues et traitées par ordinateur. Les fichiers peuvent être regroupés dans des répertoires. Un **répertoire** peut contenir soit des fichiers, soit d'autres répertoires dans une structure arborescente.

L'accès aux fichiers se fait en spécifiant un **chemin d'accès**, c'est-à dire la liste des répertoires à traverser pour accéder au fichier. Un chemin d'accès

est **absolu** si le point de départ est le répertoire racine. Un chemin d'accès est **relatif** si le point de départ est le répertoire courant. Les appels systèmes permettent de créer les fichiers et les répertoires, ainsi que de les supprimer, de les ouvrir, de les lire et de les modifier.

1.5 Structure d'un système d'exploitation

1.5.1 Structure en couches

Le système d'exploitation est structuré en couches. Chaque couche utilise les fonctions des couches inférieures. La principale difficulté est la définition des différentes couches. Par exemple, on peut l'organiser en cinq couches, comme montré sur la figure 1.7 :

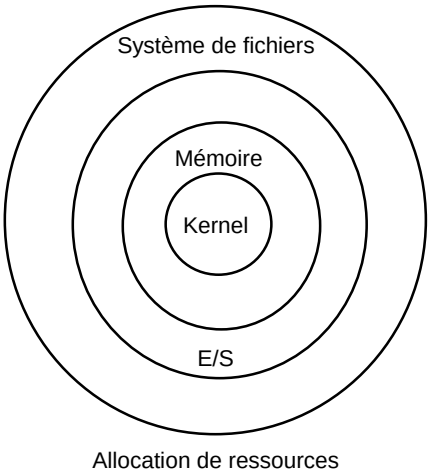


FIG. 1.7 – *Structure en couches.*

- Au plus bas niveau on trouve le **noyau**, l'interface entre le matériel et le logiciel. Il se charge, en utilisant les fonctions fournies par le matériel, de gérer la UCT, les interruptions et les processus (la communication et la synchronisation). Il doit entièrement résider en mémoire.
- Au second niveau, on trouve le gestionnaire de la mémoire qui se charge du partage de la mémoire entre les processus en attente d'exécution.
- Au troisième niveau, on a le module de gestion des entrées / sorties qui se charge de gérer tous les périphériques (clavier, écran, disques,

imprimantes, etc.).

– Au quatrième niveau, on trouve le gestionnaire de fichiers qui se charge de la gestion de l'espace du disque, de la manipulation des fichiers tout en assurant l'intégrité des données, la protection des fichiers, etc.
– Au cinquième niveau, on a le module d'allocation de ressources qui se charge d'assurer une bonne utilisation des ressources ; de comptabiliser et de fournir des statistiques sur l'exploitation des ressources principales ; de créer de nouveaux processus et leur attribuer un niveau de priorité : de permettre à chaque processus existant dans le système d'obtenir les ressources nécessaires dans des limites de temps raisonnables ; d'exclure mutuellement les processus qui demandent une ressource non partageable et d'éviter les situations de blocage.

1.5.2 Structure monolithique

Les systèmes d'exploitation sous une **structure monolithique** (figure 1.8) sont un ensemble de procédures de —presque— le même niveau : une **procédure principale** qui appelle la **procédure de service** requise, des procédures de service qui exécutent les **appels système** et un ensemble de **procédures utilitaires** qui assistent les procédures de service, par exemple la recherche de données des programmes utilisateur. Unix et Linux sont des

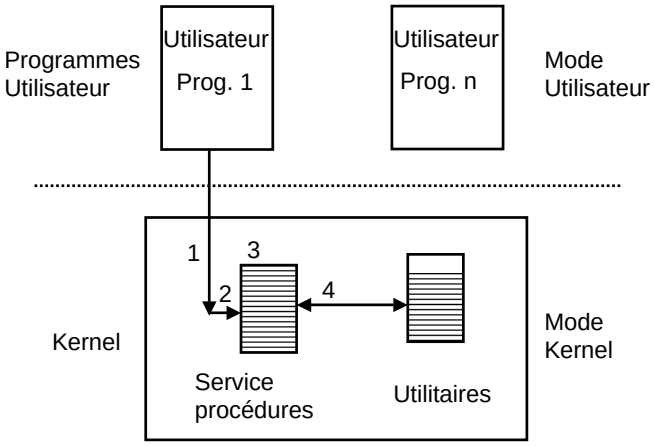


FIG. 1.8 – *Structure monolithique. (1) Appel système (Mode utilisateur ↦ Mode kernel). (2) Vérification de paramètres. (3) Appel de la procédure de service. (4) Procédure de service appel utilitaires, et puis retourner au mode utilisateur.*

exemples de systèmes monolithiques.

1.5.3 Micro-kernel

Une architecture plus moderne que celle monolitique est l'architecture **micro-kernel** (voir la figure 1.9) utilisée en MACH[5] / HURD[6], Minix et NT. L'attribut principal qui distingue les micro-kernels des kernels monoli-

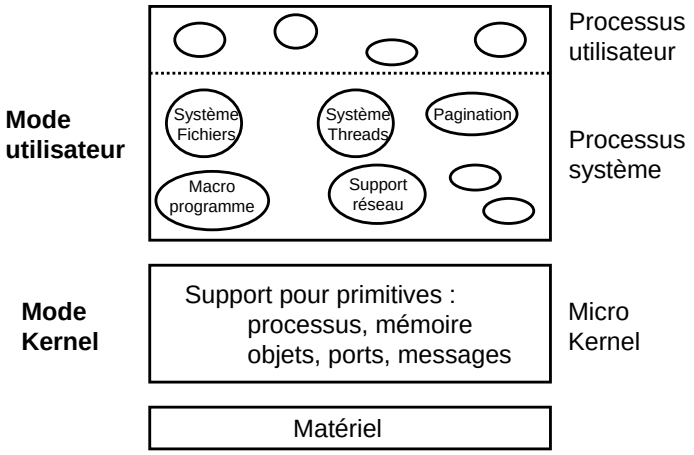


FIG. 1.9 – *Structure de micro-kernel.*

tiques est l'implantation de leurs architectures respectives en mode superviseur (*kernel mode*) et en mode usager (*user mode*). L'architecture monolitique met en œuvre tous les services du Système d'exploitation (controleurs de dispositifs, mémoire virtuelle, système de fichiers, réseaux, etc) dans le domaine du mode superviseur de l'UCT. Par contre, l'architecture micro-kernel fait une division entre les services du Système d'exploitation, en les divisant en « haut-niveau » implantés dans le domaine de l'utilisateur et « bas-niveau » implantés dans l'espace du mode superviseur.

1.5.4 Le modèle client/serveur

Dans le modèle **client/serveur** Le système d'exploitation est composé d'un noyau et d'un ensemble de serveurs. Le noyau gère la communication entre les **clients** et les serveurs. Les clients sont les demandeurs de

5. http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html
6. http://www.gnu.ai.mit.edu/software/hurd/hurd.html

services. Par exemple, pour demander un service, comme la lecture d'un bloc d'un fichier, un processus utilisateur (aussi appelé processus client) envoie une requête à un processus serveur qui effectue le travail et renvoie une réponse. Les **serveurs** s'exécutent en mode utilisateur. Ils ne peuvent donc pas accéder directement au matériel. Par conséquent, une erreur ou un bogue dans le serveur de fichiers, par exemple, n'affectera pas, en général, l'ensemble de la machine. Les dégâts se limitent au serveur.

1.5.5 Machines virtuelles

Le cœur du système se charge de la multiprogrammation en fournissant à la couche au dessus plusieurs machines virtuelles. Des systèmes d'exploitation différents peuvent tourner au dessus d'une machine virtuelle.

Exemples de machines virtuelles :

- **IBM VM** : offre à chaque usager sa propre machine virtuelle monotâche. Les machines virtuelles étaient planifiées avec du temps partagé.
- **Java** : les programmes compilés en Java tournent sur une machine virtuelle (JVM).
- **VMWare**[7] : sur PC, exécute en même temps des sessions Windows, Linux, OS/2, etc.
- **Nachos** : SE qui s'exécute dans une machine virtuelle MIPS, qui tourne sur Unix.

7. http://www.vmware.com

Sugestions de lecture

Référence : Silverwschatz A., Galvin P., Gagné G., *Applied Operating System Concepts*, Wiley, 2003.

Chapître 1

Section 1.1 What is an operating system?

Section 1.2 Batch systems : une attention particulière au concept de multiprogrammation.

Section 1.3 Time-Sharing systems : Bien saisir la différence entre multiprogrammation et temps partagé. Bien saisir la complexité des systèmes à temps partagé.

Chapître 2

Section 2.1 Computer-System Operation.

Section 2.2 I / O Structure : Le concept d'interruption est très important. En ce qui concerne le DMA, il est bon de savoir ce que c'est.

Section 2.5 Hardware protection : Important, avec une attention toute particulière aux modes utilisateur et superviseur.

Section 2.6 General system arquitecture : Le concept d'appel système est très important.

Chapître 3

Section 3.1 Systems components : Bien comprendre le concept de processus.

Section 3.3 System Calls : Très important.

1.6 Exercices

1. Indiquez si les éléments logiciels suivants se situent parmi la catégorie des programmes d'application, des programmes système, ou des programmes de systèmes d'exploitation.
   - (a) Le programme `more`
   - (b) L'éditeur de texte `emacs`
   - (c) La librairie de routines X-Window (Xlib)
   - (d) Un script de shell
   - (e) Le traitement de texte Ms-word
   - (f) La fonction `read`
2. L'ordinateur personnel compatible IBM fournit une interface aux périphériques en utilisant des routines du BIOS (Basic Input and Output System) qui est stocké en mémoire morte (ROM). Quel avantage cette interface fournit-elle aux développeurs d'applications pour la plateforme Intel 80x86?
3. Dans un environnement multiprogrammé, y-a-t-il partage des ressources entre les processus d'un même utilisateur? Expliquez.
4. Le degré de multiprogrammation représente le nombre maximal de processus qu'un système uniprocesseur peut manipuler à tout moment. Expliquez quelques facteurs matériels et logiciels qui pourraient avoir une certaine influence sur le degré de multiprogrammation d'un système.
5. Décrivez le facteur principal qui sert de mesure du temps de réponse pour un système à soumission de lots et pour un système en temps partagé.
6. Expliquez la différence fondamentale qui existe entre un système d'exploitation multiprogrammé et un système d'exploitation réseau.
7. Une fenêtre (*window*) représente un certain niveau d'abstraction de l'écran car les applications interagissent avec celle-ci et non pas directement à l'écran. Décrivez trois prototypes de fonctions qu'une application pourrait utiliser pour interagir avec une fenêtre d'écran.
8. Quelle différence existe-il entre un système temps réel et une application en temps réel?
9. Expliquez la raison principale de l'utilisation de deux modes (usager, système) d'opération dans les systèmes d'exploitation modernes.
10. Quel est le rôle d'un système d'exploitation? Les interpréteurs de

commandes et les compilateurs font-ils parties du système d'exploitation ?

11. Qu'est ce qu'un système multiprogrammé ? Un système de traitement par lots ? Un système en temps partagé ?
12. Dans le système Unix, les véritables appels système sont effectués à partir :
    (a) D'un programme utilisateur.
    (b) D'une commande shell.
    (c) D'une procédure de la bibliothèque standard.
    (d) Sont-ils exécutés en mode superviseur ou en mode utilisateur ?
13. Comment sont organisés les fichiers dans le système Unix ? Un utilisateur peut-il accéder à un fichier d'un autre utilisateur ? Si oui, comment ?
14. Est-ce qu'un Système d'Exploitation multitâche est nécessairement multiusager ? Et pour l'inverse ? Expliquez.

2
Introduction au système Unix/Linux

Le système **Unics** (*Uniplexed Information and Computing Service*) a été créé aux laboratoires AT&T de Bell, en 1969 par Ken Thompson, et modifié et baptisé par Brian Kernighan [Tan01]. Il a été une version réduite de **Multics** (*Multiplexed Information and Computing Service*). Peu après on a changé le nom Unics par Unix, et à partir de ce moment le système Unix a commencé un long chemin de développement technique.

2.1 Bref historique

Le système Unix a connu un véritable succès, lorsqu'il fut récrit en langage *C* en 1973 par Denis Ritchie et Thompson [1]. L'Université de Californie à Berkeley a obtenu une copie de la version 6 du système Unix. AT&T et Berkeley ont séparément apporté de nombreuses modifications et améliorations au système Unix (**System V** d'AT&T et **4.4BSD** de Bekerley). Une vision simplifiée de l'évolution subie par Unix est montrée sur la figure 2.1. Le projet Posix (*Portable Operating System UnIX*) de normalisation du système Unix a permis de développer un standard qui définit un ensemble de procédures. Tout système Unix conforme à Posix fournit ces procédures ou appels système standards. Ces procédures constituent la bibliothèque standard d'Unix (figure 2.2). Tout logiciel écrit en n'utilisant uniquement les procédures de la norme Posix devrait fonctionner sur tous les systèmes Unix conformes à cette norme. Une version gratuite d'Unix porte le nom de

1. Le langage *C* de Ritchie provient du langage *B* écrit par K. Thompson, et *B* à son tour de BCPL. *C* a été aussi standardisé par ANSI.

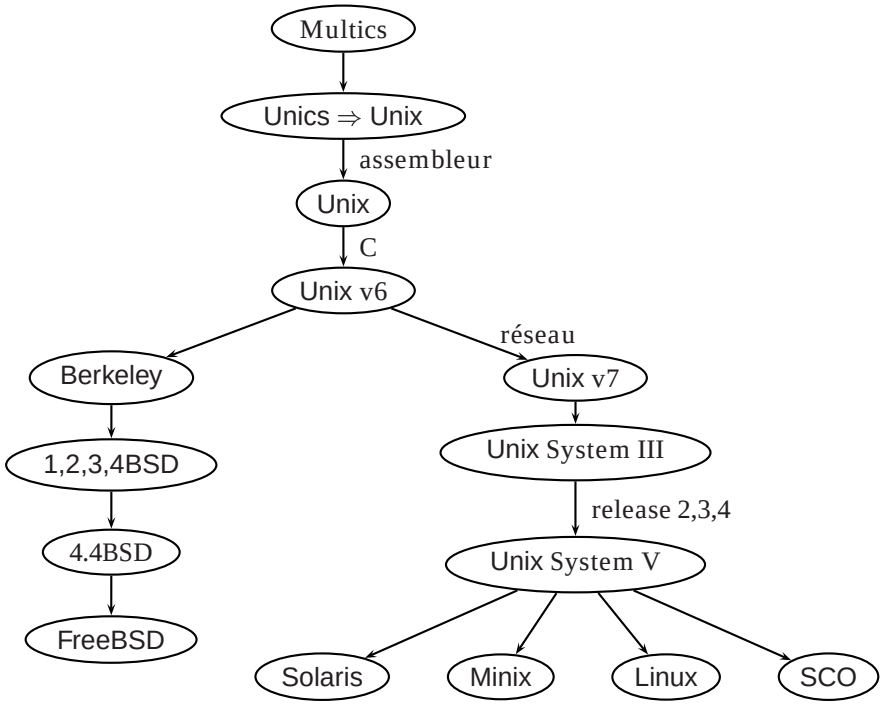


FIG. 2.1 – *Evolution d'Unix.*

Linux (code source disponible). Elle a été créée par Linus Torvalds en 1991. Par la suite, un grand nombre de programmeurs ont contribué à son développement accéléré. Conçu d'abord pour tourner sur des machines avec le processeur 80x86, Linux a migré à plusieurs autres plate-formes. Le système d'exploitation Linux peut être téléchargé à partir de plusieurs sites, par exemple : `http://www.linux.org`. Dans l'annexe D le lecteur interessé trouvera de plusieurs sites internet où l'on peut télécharger d'autres versiones de Linux.

FreeBSD est une autre version gratuite d'Unix. Il est un systeme d'exploitation pour x86 compatibles, DEC Alpha, et architectures PC-98. Il est dérivé de BSD Unix et développé par une grande communauté d'individus (`http://www.freebsd.org`).

2.2 Caractéristiques

Unix est un système interactif et multi-utilisateurs (multiprogrammé en temps partagé). Plusieurs utilisateurs peuvent disposer, en même temps, de

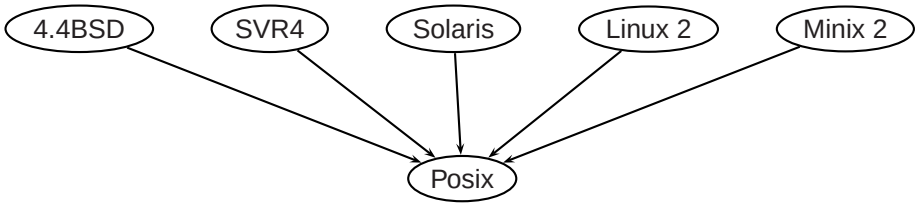


FIG. 2.2 – *Standard Posix*

la puissance de calcul du système. Le système Unix se charge de contrôler et de gérer l'usage des ressources en les attribuant, à tour de rôle, aux différents utilisateurs. Il permet la création, la communication et la synchronisation des processus. Unix est un système d'exploitation ouvert, portable et disponible sur différentes plate-formes. Linux est en plus gratuit et on a le droit d'étudier et de modifier le code source.

2.2.1 Structure d'un système Unix/Linux

Un système informatique sous Unix / Linux est constitué de couches de logiciels, comme illustré à la figure 2.3.

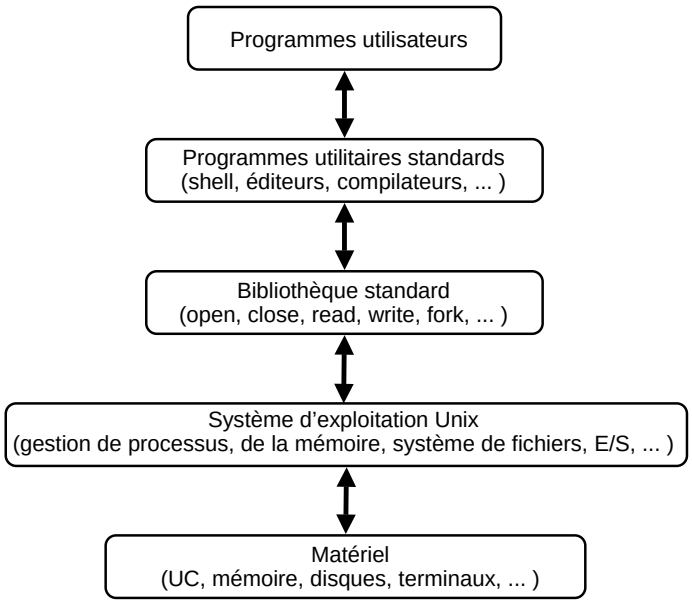


FIG. 2.3 – *Architecture d'un système Unix.*

Le système d'exploitation, appelé **noyau** ou *kernel*, gère le matériel et fournit aux programmes une interface d'appels système. Le kernel de BSD est montré à la figure 2.4. Les appels système permettent aux programmes de créer et de gérer des processus et des fichiers. A chaque appel système correspond une procédure de bibliothèque que l'utilisateur peut appeler (bibliothèque standard). Chaque procédure se charge de placer les paramètres de l'appel système correspondant en un endroit prédéfini comme les registres du processeur, et de provoquer une **interruption logicielle** (instruction TRAP) pour passer du **mode utilisateur** au **mode noyau** et activer ainsi le système d'exploitation.

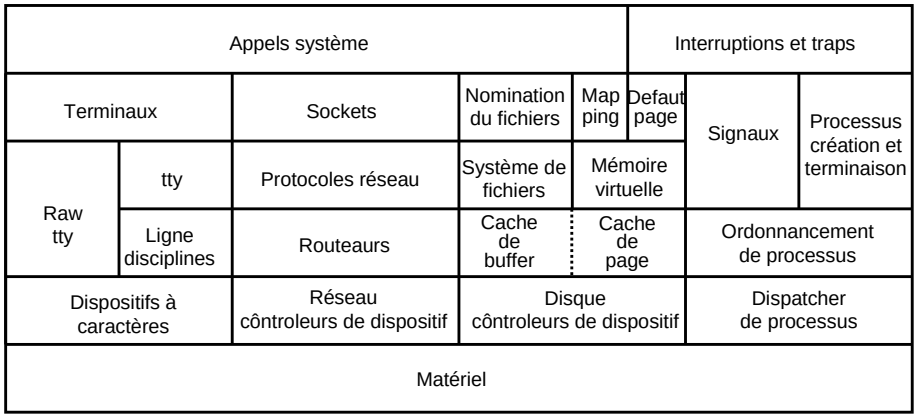


FIG. 2.4 – *Architecture du kernel BSD d'Unix.*

La procédure de bibliothèque a pour but de masquer les détails de l'instruction TRAP et de faire apparaître les appels de procédure comme des appels de procédures ordinaires. Par exemple `read(f, b, 50)` sera utilisé pour lire `50` caractères dans un stockage temporaire ou *buffer* `b`, à partir d'un fichier `f` présentement ouvert.

Lorsque le système d'exploitation prend le contrôle suite au TRAP, il vérifie la validité des paramètres et effectue dans ce cas le traitement demandé. A la fin du traitement, il place un code de statut, indiquant si le traitement a réussi ou échoué, dans un registre, puis redonne le contrôle à la procédure de bibliothèque avec retour au mode utilisateur. La procédure de bibliothèque remet, à son tour, le contrôle à l'appelant en lui transmettant le code de statut. Les utilisateurs peuvent développer des programmes et invoquer des appels système, grâce la bibliothèque standard et à un ensemble d'utilitaires (shells, éditeurs, compilateurs, etc.) fournis avec le sys-

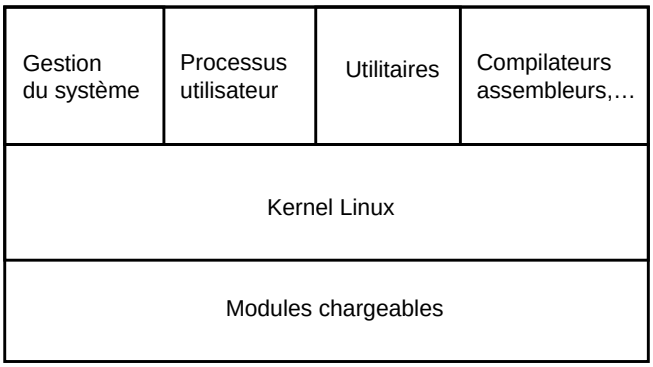


FIG. 2.5 – *Architecture de Linux.*

tème Unix/Linux. Par exemple, pour exécuter l'appel système READ, un programme *C/C++* peut appeler la procédure de la bibliothèque standard `read()` qui va effectuer le véritable appel système READ.

Remarque : Les véritables appels système s'exécutent en **mode noyau**, ce qui assure la protection du système d'exploitation contre les tentatives d'intrusions et les erreurs.

2.3 Début de session

Lorsqu'un utilisateur demande à se connecter, le système l'invite à introduire son nom d'utilisateur (code) et son mot de passe. Si ces données sont correctes, le système ouvre une session de travail et lance l'**interpréteur de commandes** (processus shell) qui affiche à l'écran, aussitôt après son initialisation, une invitation puis se met en attente d'ordres de l'utilisateur. Dépendamment du shell utilisé, ceci peut être un simple symbole :

```
$
```

ou bien, l'invitation peut montrer le nom de la machine. Dans notre cas l'une des machines s'appelle `leibnitz` :

```
leibnitz>
```

Lorsque l'utilisateur introduit une commande, le shell vérifie si elle est correcte, puis crée, si c'est le cas, un processus qui a pour tâche d'exécuter la commande. Par exemple la commande :

```
leibnitz>cp source destination
```

copiera le fichier `source` dans le fichier `destinantion`. Le shell attend la terminaison du processus créé avant d'inviter l'utilisateur à introduire une

nouvelle commande.

2.3.1 Interpréteurs de commandes

Il existe plusieurs interpréteurs de commandes ou shell populaires pour le système Unix. Parmi eux on trouve :

- Le shell de Bourne : `sh`
- Le shell de Korn : `ksh`
- Le C shell : `csh`

Le shell de Korn englobe celui de Bourne. Bon nombre de commandes sont communes aux trois shell's.

Le shell `bash`, écrit par Brian Fox, fait partie du projet **GNU**, acronyme récursif de **G**NU's **N**ot **U**nix, démarré par Richard Stallman en 1984[2]. Le projet GNU voulait créer un système complet et compatible Unix mais distribué librement. GNU avait développé plusieurs programmes comme le compilateur `gcc`, l'éditeur `emacs`, `make`, `tar`, l'interface graphique X-Windows. En 1991 le kernel Linux a rejoint le projet et on a appelé le tout **GNU/Linux**. Le shell `bash` (acronyme de *Bourne-Again SHell*) est compatible avec le standard Posix, et compatible aussi avec le shell `sh`. Il incorpore beaucoup des caractéristiques du Korn shell `ksh` et le C shell `csh`.

2.4 Commandes utiles d'Unix/Linux

Dans la table 2.1 nous montrons quelques commandes Posix très utilisés dans Unix / Linux.

`man`

```
man [-s section] mot
```

La commande `man` donne une aide en ligne. Elle affiche les informations correspondantes au titre donné, figurant dans le manuel standard de Linux. Ce manuel est composé de 8 sections.

▶ **Exemple 1.** La commande :

```
leibnitz> man chmod
```

sélectionne la section `1` des commandes, et :

```
leibnitz> man -s 2 chmod
```

2. http://www.gnu.org

Commande	Description
who	Afficher la liste des utilisateurs connectés
who ami	Afficher l'utilisateur de la session courante
date	Afficher la date
ps	Afficher la liste des processus de l'utilisateur
kill	Stopper un processus
passwd	Créer ou changer de mot de passe
pwd	Afficher le nom du répertoire de travail
mkd	Créer un répertoire
cd	Changer de répertoire de travail
cat	Fusionner une liste de fichiers et afficher le résultat
head	Afficher le début d'un fichier
grep	Afficher les lignes des fichiers référencés qui contiennent une chaîne de caractères donnée
wc	Compter le nombre de mots, lignes ou caracteres.
sleep	Dormir pendant un certain temps (secondes)
find	Rechercher un fichier
ls	Afficher le contenu d'un répertoire
cp	Copier un fichier dans un autre
ln	Ajouter à un fichier existant un autre nom
mv	Renommer un fichier
rm	Supprimer un fichier
rmdir	Supprimer un répertoire
chmod	Changer les protections d'un fichier

TAB. 2.1 – *Commandes Posix d'Unix.*

sélectionne la section 2 des appels système. Évidemment si l'on veut comprendre comment `man` fonctionne, il faut taper :

```
leibnitz> man man
```

`sort`

La commande `sort` réalise la lecture des lignes provenant du terminal (jusqu'à ce que l'utilisateur tape `CRTL-D` indiquant la fin de fichier), trie les lignes par ordre alphabétique et affiche le résultat à l'écran.

▶ **Exemple 2.** Utilisation de `sort`

```
leibnitz> sort
Martin 16 HULL
Robert 17 Ottawa
Catherine 15 Montréal
^D
Catherine 15 Montréal
Martin 16 HULL
Robert 17 Ottawa
leibnitz>
```

Redirection des entrées/sorties

A chaque processus créé (commande ou programme en exécution) sont associées une entrée et deux sorties standards :

- L'entrée standard : le clavier
- La sortie standard : l'écran du terminal
- La sortie erreur standard : l'écran du terminal

Le shell permet de modifier ou faire la **redirection** des Entrées / Sorties d'un processus.

Redirection de l'entrée standard

On fait la redirection avec l'opérateur `<` :

```
commande < fichier
```

▶ **Exemple 3.** Cette commande trie le fichier `source`, puis affiche le résultat à l'écran :

```
leibnitz> sort < source
```

Redirection de la sortie standard

On peut rediriger la sortie standard avec l'opérateur > ou » :

```
commande > nom_fichier
commande >> nom_fichier
```

▶ **Exemple 4.** La liste des utilisateurs connectés au système est récupérée dans le fichier `users` avec la commande suivante. Si le fichier existe déjà, il y aura une erreur :

```
leibnitz> who > users
```

Si le fichier existe déjà, la liste est insérée à la fin :

```
leibnitz> who >> users
```

▶ **Exemple 5.** Cette commande lit les lignes provenant du terminal puis les recopie dans le fichier `dest` (jusqu'à ce que l'utilisateur tape `CRTL-D`, pour indiquer la fin de fichier) :

```
leibnitz> cat > dest
```

Redirection des sorties standards (erreur et normal)

```
(commande) >& nomfich
(commande) >> & nomfich
```

▶ **Exemple 6.** Redirection des sorties.

```
leibnitz> (ls ggg)>& er
leibnitz> cat er
ggg: Ce fichier ou ce répertoire n'existe pas
leibnitz> (lhiumo ) >>& er
leibnitz> cat er
ggg: Ce fichier ou ce répertoire n'existe pas
lhiumo : Commande introuvable
leibnitz>
```

Enchaînement séquentiel des commandes

Le shell permet de lancer plusieurs commandes (processus) de façon séquentielle avec l'opérateur ; point-virgule :

```
commande_1 ; commande_2 ; .... ; commande_n
```

▶ **Exemple 7.** Cette commande permet d'afficher le répertoire de travail puis son contenu. Après elle va créer le répertoire (cours) dans le répertoire de travail. Elle affiche le contenu du répertoire de travail, ensuite elle change au répertoire de travail (cours), affiche le nouveau répertoire de travail (cours), puis elle revient au répertoire père, et finalement affiche le nouveau répertoire de travail :

```
leibnitz> pwd;ls;mkdir cours;ls;cd cours;pwd;cd ..;pwd
```

Enchaînement parallèle des commandes

De façon similaire, on peut lancer plusieurs commandes en parallèle avec l'opérateur | et de diriger la sortie de l'un vers l'entrée d'un autre. On appelle ceci **pipelines** :

```
commande_1 | commande_2 | ... | commande_n
```

▶ **Exemple 8.** La commande suivante (voir figure 2.6) crée deux processus cat et sort. cat fusionne les fichiers src1 et src2 et envoie le résultat à sort ; sort trie les données reçues avant de les stocker dans le fichier dest. Les deux processus s'exécutent en parallèle :

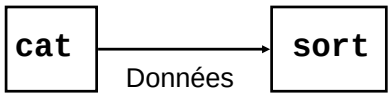


FIG. 2.6 – *Enchaînement parallèle des commandes cat et sort.*

```
leibnitz> cat src1 src2 | sort > dest
```

▶ **Exemple 9.** Utilisation enchaînée.

```
leibnitz> grep ter *.t|sort|head -20|tail -5 >fich
```

La commande `grep` envoie à `sort` les lignes contenant la chaîne « `ter` » de tous les fichiers se terminant par `.t`, puis `sort` trie les lignes reçues et les transmet à `head`, qui à son tour envoie les 20 premières à `tail`. Finalement, la commande `tail` écrit les cinq dernières dans le fichier `fich`.

▶ **Exemple 10.** Les sorties (erreur ou non) de la compilation/édition de liens du programme sont re-dirigées vers l'entrée du processus `more`. Ce dernier affiche à l'écran les résultats progressivement :

```
leibnitz> gcc prog.c | & more
```

Tâches de fond

Le shell permet de lancer une `commande` en tâche de fond en utilisant l'opérateur &. Dans ce cas, il n'attend pas la fin de l'exécution de la `commande` :

▶ **Exemple 11.** Cette commande fera la recherche récursive à partir du répertoire racine de tous les fichiers de nom `bin`. Les chemins d'accès absolus des fichiers trouvés sont stockés dans le fichier `result` :

```
leibnitz> find / -name bin -print >result &
```

2.5 Scripts shell

Chaque shell d'Unix a son propre langage de programmation conçu pour manipuler des fichiers et des processus. Ce langage comporte des structures de contrôle semblables à celles des langages de programmation classiques. Les programmes shell sont appelés **script shell**. Dans l'Annexe A vous trouverez une liste complète des commandes et des opérateurs `test [ ]` de `if`.

Syntaxe de certains commandes shell

Le shell peut effectuer des commandes répétitives sous forme de boucles `for` ou `while`, ainsi que prendre des décision avec `if` ou `case` :

– Boucle `for` :

```
for name [in word] do list; done
```

- Boucle `while`:
  `while list ; do list ; done`
- Décision `case`:
  `case mot in [(pattern[|pattern]...) liste ;;]... esac`
- Décision `if`:
  `if liste; then liste; [elif liste; then liste]... fi`

Autorisation de l'exécution du script

Il faut donner au script le mode adéquat pour que l'interpréteur de commandes puisse l'exécuter.

▶ **Exemple 12.** On peut créer un fichier `script0`:

```
leibnitz> cat > script0
for NAME in Jill Richard Sam ; do
   echo "Hola $NAME";
done
^D
leibnitz>
```

puis on peut changer le mode à `u+x` pour le rendre exécutable:

```
leibnitz> ls -l script0
-rw------- 1 jmtorres professeur 326 août 27 19:32 script0
leibnitz> chmod u+x script0
leibnitz> ls -l script0
-rwx------ 1 jmtorres professeur 326 août 27 19:32 script0
leibnitz>
```

Commentaires

Les scripts shell peuvent avoir des commentaires s'ils sont précédés du signe #

```
#*************************************************
# Fichier : script.sh
# Auteur(s) : Juan Manuel Torres
# Cours : Systèmes d'exploitation
# École Polytechnique de Montréal
#*************************************************
```

En particulier le *commentaire* `#!/bin/bash` indique au shell qu'on veut utiliser `bash` pour l'exécution de nôtre script.

Variables

Les variables peuvent être utilisées comme dans d'autres langages de programmation. L'assignation de la valeur à une variable este très simple. Par exemple, l'instruction :

```
$ REP=repertoire
```

stocke le mot `repertoire` dans a variable `REP`. Conventionnelement tous les noms de variables doivent être en majuscules, mais ceci n'est pas impossé par le shell. Une fois la variable définie, on peut l'utiliser librement :

```
$ REP=repertoire
$ echo $REP
$ repertoire
$ date > $REP
$ cat $REP
$ Wed Feb 19 16:01:13 EST 2003
```

Pour supprimer une variable, il suffit d'utiliser la commande `unset` :

```
$ REP=repertoire
$ echo $REP
$ repertoire
$ unset REP
$ echo $REP
$
```

Les paramètres de commandes

Les arguments d'une procédure sont récupérés dans les variables `$1`, `$2`, `...`, `$9`. `$0` garde le nom utilisé pour appeler la commande. Le nombre effectif d'arguments est donné par `$#`. `$*` et `$@` donnent la liste de tous les arguments.

Lecture interactive

La commande `read` permet de lire une ligne de l'entrée standard et d'affecter les valeurs des variables dont les noms sont donnés en arguments, à raison d'un mot par variable, la dernière variable se voyant affectée tout le reste de la ligne (s'il reste des mots !).

▶ **Exemple 13.**

Listing 2.1: *read.sh*

```
#!/bin/bash

read a b c
echo "Le premier mot : " $a
echo "Le deuxième mot : " $b
echo "Le reste de la ligne est : " $c
```

`read` est une commande qui rend faux quand la fin du fichier est rencontrée. On peut ainsi s'en servir pour lire un fichier en totalité à l'aide d'une boucle `while`.

▶ **Exemple 14.**

Listing 2.2: *mon-cat.sh*

```
#/bin/bash

while read ligne;
do
    echo "$ligne"
done
```

▶ **Exemple 15.** Ce script montre le message `"Hola $NOM"` pour chacun des noms de la liste {`Patricia`, `Richard`, `Jean`} :

Listing 2.3: *hola.sh*

```
#!/bin/bash

for NOM in Patricia Richard Jean ; do
    echo "Hola $NAME";
done
```

▶ **Exemple 16.** Ce script montre la liste de fichiers qui commencent par `hda*` dans le répertoire `/dev/` :

Listing 2.4: *hda.sh*

```
#!/bin/bash

for FILENAME in /dev/hda*; do
```

```
  echo "J'ai trouvé le fichier $FILENAME"
done
```

▶ **Exemple 17.** La boucle `while` permet d'évaluer des expressions :

Listing 2.5: *while.sh*

```
#!/bin/bash

i=1
while true do
  echo $i
  i=$(expr $i + 1)
  if [ $i == 6 ]
  then
      exit 0
  fi
done
```

Ce script donnera comme résultat :

```
leibnitz>
1
2
3
4
5
leibnitz>
```

▶ **Exemple 18.** Ce script trouve la liste de rapports textes de 1997 à 1999 qui contiennent le mot `Linux`. Ensuite, pour tous ces fichiers, il affiche le nom en traitement, puis il copie les 20 dernières lignes dans `TEMP`, finalement il envoie, par `courriel` avec un sujet adéquat le fichier `TEMP` :

Listing 2.6: *rapport.sh*

```
#!/bin/bash

for FILE in `grep -l Linux Rapport-199[7-9].txt`; do
 echo "Traitement de $FILE"
 tail -20 $FILE > TEMP
 mail -s "20 lignes de $FILE" juan-manuel.torres@polymtl.ca
        < TEMP
```

```
done
```

▶ **Exemple 19.** Ce script renvoi le texte inversé tapé par l'utilisateur. Le `-n` de `echo` sert à ne pas sauter à la ligne suivante après l'affichage du message :

Listing 2.7: *inv.sh*

```
#!/bin/bash

echo "Tapez du texte ; Ctl-D fin"
echo -n "entre: "
while read; do
  TEXT=`echo "$REPLY" | rev`
  echo "Inverse : $TEXT"
  echo -n "Votre entree: "
done
```

▶ **Exemple 20.** Test de type de fichier. Le script `affichage.sh` récupère les fichiers du répertoire courant (`set 'ls'`) dans les variables `$i`. La variable `$0` contient le nom du premier fichier, `$1` le nom du second et ainsi de suite. Il parcourt les noms des fichiers (`for FICH in $*`). Pour chaque nom, il teste si c'est un répertoire (`if [ -d $FICH ]`). Si c'est le cas, il affiche le nom du répertoire suivi du texte `'est un répertoire'` (`echo "$FICH est un répertoire"`). Dans le cas contraire, il vérifie s'il s'agit d'un fichier normal (`if [ -f $FICH ]`). En cas de succès, il affiche un message puis propose d'afficher le contenu du fichier `$FICH` :

Listing 2.8: *affichage.sh*

```
#!/bin/bash

set `ls`
for FILENAME in $*; do
  if [ -d $FILENAME ]; then
      echo "J'ai trouve le repertoire $FILENAME"
  elif [ -f $FILENAME ]; then
      echo "$FILENAME est un fichier. Affichage ? (o | n)"
      read rep
      case $rep in
      o | O) cat $FILENAME ;;
```

```
      n | N) echo "Pas de visualisation de $FILENAME" ;;
      *) echo "Reponse incorrecte"
      esac
  fi
done
```

Exécution du `affichage.sh`:

```
leibnitz> cat fich
bonjour ici fich
a bientôt
leibnitz>
leibnitz> affichage.sh
fichier trouve. Affichage ? (o ou n)
o
bonjour ici fich
a bientôt
leibnitz>
```

Petite arithmétique shell

Dans la table 2.2 nous montrons un petit aperçu des capacités arithmétiques de `bash`.

Opérateurs	Opération
+, -, *	Addition, différence, multiplication
/, %	division, reste
==, !=, <=, >=	égal, différent, inférieur ou égal, supérieur ou égal
<, >	Comparaisons
>>, <<	Décalages
!, ~	Négation logique et de bits
&, |	AND, OR
&&, ||	AND, OR à niveau de bits

TAB. 2.2 – *Capacités arithmétiques* `bash`.

▶ **Exemple 21.** Opérations arithmétiques:

```
leibnitz> bash
bash-2.05a$ v1=5
bash-2.05a$ v2=7
```

```
bash-2.05a$ v3=$v1+$v2
bash-2.05a$ v4=$((v1+v2))
bash-2.05a$ echo $v1
5
bash-2.05a$ echo $v2
7
bash-2.05a$ echo $v3
5+7
bash-2.05a$ echo $v4
12
bash-2.05a$ echo "$v3=$v4"
5+7=12
```

Il existe aussi une autre commande `expr` qui permet d'évaluer une expression arithmétique composée de constantes et de variables shell.

▶ **Exemple 22.** Utilisation d'`expr`.

```
i=1
i=$(expr $i + 1)
echo "i=${i}"
i=$((i+1))
echo "i=${i}"
```

aura comme résultat :

```
a=2
a=3
```

2.6 Editeurs d'Unix

Plusieurs éditeurs fonctionnent sous Unix : `emacs`, `vi`, `xedit`, `pico` et `CDE Text Editor` parmi d'autres. `vi` est l'éditeur standard sur la majorité des systèmes Unix. Il a comme vertu de fonctionner là où d'autres éditeurs échouent. Malgré le fait qu'il n'est pas très convivial, il est très performant et permet de faire l'édition de très gros fichiers. `vi` utilise deux modes de fonctionnement : le **mode insertion** utilisé pour entrer du texte ; et le **mode commande** utilisé pour manipuler le fichier.

Pour créer un fichier `nomfich` avec l'éditeur `vi`, il faut :

- Lancer la commande : `vi nomfich`
- Appuyer sur la touche `i` pour passer au mode insertion.
- Taper votre texte.
- Appuyer sur `<esc>` pour passer au mode commande.
- Pour la sauvegarde, taper `:w`

– Pour quitter, taper `:q`
– Pour quitter sans modifier, taper `:q!`

Il existe plusieurs clons de `vi`, par exemple : VIM (http://www.vim.org), stevie (http://www.scytale.com/waffle/wafiles.shtml), bvi (http://bvi.sourceforge.net) et bien d'autres, adaptés à plusieurs platte-formes, non seulement Unix. Le lecteur interessé peut consulter le site http://www.vi-editor.org où c'est bien documenté l'histoire et l'utilisation de cet éditeur.

2.7 Utilisateurs

Chaque **utilisateur du système** Unix/Linux est identifié par un numéro unique appelé `UID` (*user identifier*). Un utilisateur particulier, appelé le **superviseur**, **administrateur** ou `root`, dispose de certains privilèges que les autres n'ont pas. Il peut accéder à tous les fichiers et effectuer certains appels système réservés.

Le système Unix/Linux offre la possibilité de constituer des **groupes d'utilisateurs**. Chaque groupe est identifié par un numéro unique appelé `GID` (*group identifier*). Les `UID` et `GID` d'un utilisateur servent, par exemple, à définir les droits d'accès aux fichiers de l'utilisateur. L'utilisateur propriétaire d'un fichier peut permettre aux membres de son groupe l'accès en lecture au fichier et l'interdire aux autres utilisateurs.

2.8 Fichiers et répertoires

Fichiers

L'accès aux **fichiers** se fait en spécifiant le **chemin d'accès**, qui peut être absolu ou relatif par rapport au répertoire racine. Par exemple, supposons que le répertoire `'cours'` se trouve dans le répertoire racine. Le répertoire `'cours'` contient le répertoire `inf3600` qui, à son tour, contient le fichier introduction. Dans le système Unix/Linux le chemin d'accès absolu au fichier `introduction` est : `/cours/inf3600/introduction`
Si `inf3600` est le répertoire courant, le chemin d'accès relatif est :

```
introduction
```

La commande `cd` d'Unix permet de changer de répertoire :

```
cd /cours
```

Le répertoire courant est maintenant `cours`. Le chemin d'accès relatif au fichier introduction devient :

```
inf3600/introduction
```

Les deux commandes suivantes réalisent chacune l'affichage du contenu du fichier `introduction` :

```
cat inf3600/introduction
cat /cours/inf3600/introduction
```

Chaque fichier a un **propriétaire** (en général le créateur) qui appartient à un groupe d'utilisateurs. Pour contrôler les accès aux fichiers, le système Unix affecte, à chaque fichier, 9 **bits de protection**. Ce code indique les autorisations d'accès en lecture, en écriture et en exécution du fichier pour le propriétaire (3 premiers bits), le groupe du propriétaire (les trois bits suivants) et les autres (les trois derniers bits). Il faut ouvrir un fichier avant de le lire ou d'y écrire. Le système vérifie alors les droits d'accès et retourne, si l'accès est autorisé, un entier, appelé **descripteur de fichier**, qu'il faudra utiliser dans toutes les opérations ultérieures de manipulation du fichier. Si l'accès est refusé, un code d'erreur est retourné. Les appels système liés aux fichiers permettent de créer, d'ouvrir, de lire, de récupérer les caractéristiques (taille, type,...) d'un fichier.

▶ **Exemple 23.** Le code `rwxr-x-x` (ou encore `751`) d'un fichier indique que :

- Le propriétaire peut lire, écrire, et exécuter le fichier (`rwx` ou encore `7`)
- Les membres du groupe peuvent lire et exécuter le fichier, mais ils ne peuvent y écrire (`r-x` ou encore `5`)
- Les autres utilisateurs peuvent exécuter le fichier, mais ils ne peuvent ni le lire ni y écrire (`-x` ou encore `1`).

Lorsqu'on crée un fichier, le système lui attribue par défaut un code de protection. Ce code est calculé en se basant sur la valeur de `umask`.

▶ **Exemple 24.** Pour connaître la valeur du code de protection il faut taper :

```
leibnitz> umask
```

ou bien :

```
leibnitz> umask 77
```

pour le modifier. Dans le cas de nos machines Linux, le code de protection par défaut est `22`. C'est à dire que les nouveaux fichiers sont crées avec les permissions `0666 & ~ 022 = 0644 = rw-r-r-`

Répertoires

Les fichiers peuvent être regroupés dans des répertoires. Les répertoires peuvent contenir d'autres répertoires, sous une structure arborescente (figure 2.7). Les répertoires sont aussi protégés par un code de 9 bits qui indiquent si les accès en lecture, en écriture et en recherche sont autorisés pour le propriétaire, le groupe et les autres. Le répertoire `" / "` est nommé répertoire racine ou *root*. Il est composé de plusieurs sous-répertoires, comme illustré à la figure 2.7 :

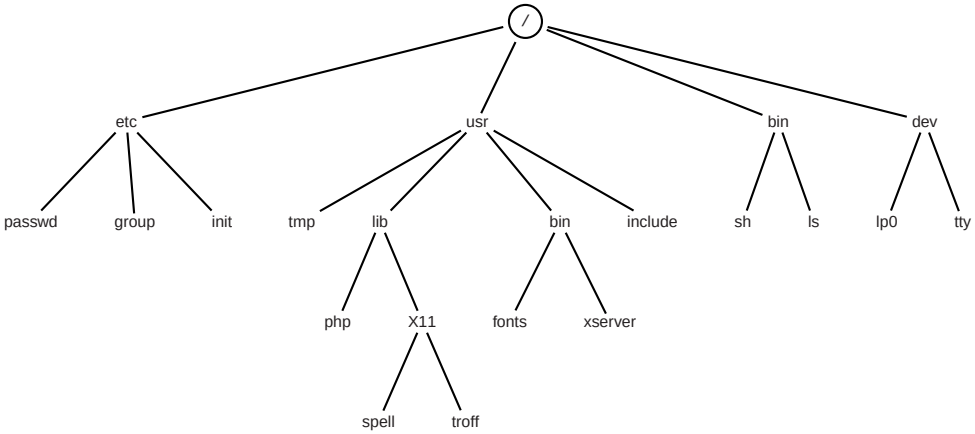


FIG. 2.7 – *Aperçu des répertoires dans Unix/Linux.*

- `/bin` : contient les commandes utilisateur communes, telles que `ls`, `sort` et `date`
- `/dev` : contient les fichiers représentant les points d'accès aux périphériques de votre système comme `tty`, `fd`, `hd`, `ram`, `cd`, etc.
- `/etc` : contient les commandes et les fichiers d'administration
- `/lib` : contient les librairies partagées ou des liens vers ces librairies
- `/tmp` : contient les fichiers temporaires
- `/usr` ou `/home` : contient les répertoires assignés à chaque utilisateur

2.9 Processus

Un **processus** est un programme qui s'exécute. Tout processus sous Unix/Linux a un espace d'adressage constitué de trois segments : le **texte**, les **données** et la **pile**, comme montré sur la figure 2.8.

- Le segment de **texte** contient les instructions en langage machine (le

code) produit par le compilateur, l'éditeur de liens et l'assembleur. Ce segment est inaccessible en écriture. Il peut être partagé entre plusieurs processus. Il est de taille fixe.

- Le segment de **données** contient l'espace de stockage des variables du programme. Il est à son tour composé de deux parties :
  - Le segment de données système appartient à Unix est inaccessible en mode utilisateur. En mode kernel, les appels système y trouvent des informations du type descripteurs de fichiers ouverts, zone de sauvegarde des registres, informations de comptabilité. On ne trouve pas de buffers d'entrée sortie dans ce segment.
  - Le segment de données utilisateur est lui aussi composé de deux parties, contenant les données non initialisées explicitement et celles initialisées (BSS). La taille de cet espace peut augmenter ou diminuer durant l'exécution du processus.
- Le segment de **pile** sert à l'allocation des variables locales, à la sauvegarde des adresses de retour des sous-programmes, à la sauvegarde des registres, etc. Il contient aussi au départ, les variables d'environnement (shell) et la ligne de commande envoyée pour demander l'exécution du programme. Un programme peut connaître tous ses paramètres.

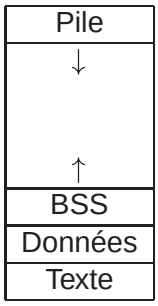


FIG. 2.8 – *Espace d'adressage.*

Chaque processus possède un numéro qui lui est propre, le `pid` (en anglais *process identifier*). C'est un numéro unique que l'on appelle l'identifiant de processus. Bien qu'il s'agit d'un chiffre entier de 32 bits, seulement sont utilisés des `pid` entre 0 et 32767, par raisons de compatibilité. Si le nombre de processus dépasse 32767, le noyau doit alors recycler des `pid`. Il existe un appel système qui retourne le `pid` du processus en cours

d'exécution :

```
int getpid()
```

`getpid()` retourne le numéro de processus – pid – du processus appelant. La valeur de retour de la fonction est le numéro du processus courant.

Visualisation de processus

Sur Unix/Linux les processus peuvent être affichés au moyen de la commande `ps` :

```
leibnitz> ps
PID TTY          TIME CMD
6100 pts/2    00:00:00 tcsh  --- le shell sur ce terminal
6150 pts/2    00:00:00 ps    --- le programme ps lui même
leibnitz>
```

La commande `pstree` permet de visualiser l'arbre de création de processus :

```
leibnitz> pstree
init-+-acpid
     |-agetty
     |-atd
     |-2*[automount]
     |-bdflush
     |-crond
     |-gpm
     |-identd---identd---5*[identd]
     |-inetd-+-2*[in.ftpd]
     |       |-in.telnetd---tcsh---pstree
     |       `-in.telnetd---tcsh---more
     ...
     |-kswapd
     |-kupdated
     |-lockd
     |-lpd
     ...
     |-sendmail
     |-snmpd
     |-sshd
     |-syslogd
     `-ypbind---ypbind---2*[ypbind]
```

En particulier on peut observer les démons [3] `init`, `getty` (accès de terminaux), `sendmail` (courrier), `lpd` (imprimante), et `pstree` lui même. Les appels systèmes d'Unix/Linux permettent la création, l'arrêt des processus, la communication et la synchronisation des processus.

Démarrage d'Unix/Linux

Au démarrage d'un système Unix/Linux, un programme **amorce** appelé **boot** est chargé du disque en mémoire grâce à un petit programme, en général câblé, qui passe ensuite le contrôle au programme amorce. Ce dernier détermine les caractéristiques du matériel et effectue un certain nombre d'initialisations. Il crée ensuite le processus `0` qui réalise d'autres initialisations (par exemple celles du système de fichiers) et crée deux processus : `init` de `PID 1` et le démon des pages de `PID 2` (voir figure 2.9).

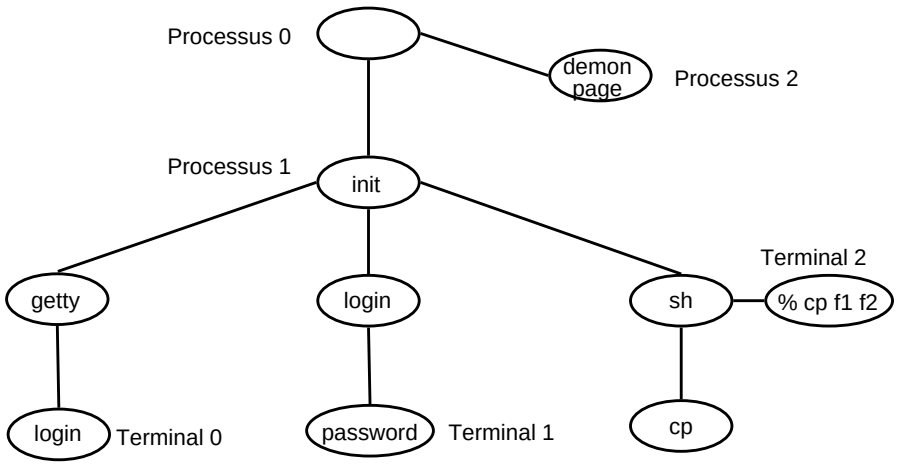


FIG. 2.9 – *Processus dans Unix/Linux.*

3. Les démons sont des processus particuliers qui tournent tout le temps. Ils seront traités dans le Chapitre 3 *Processus et threads.*

Sugestions de lecture

Référence : Silverwschatz A., Galvin P., Gagné G., *Applied Operating System Concepts*, Wiley, 2003.

Chapître 20

Section 20.1 History.

Section 20.2 Design Principles.

Section 20.3 Programmer interface.

Section 20.4 User interface.

2.10 Exercices

1. Expliquez la différence fondamentale qui existe entre un script et un `makefile`. Peut-on effectuer les opérations d'un `makefile` à l'aide d'un script?
2. Que fait-il la commande shell :
   `cat < f1 > f2`
3. Donner la commande qui permet de réaliser ce qui suit :
   (a) Afficher les utilisateurs connectés au système.
   (b) Afficher votre code utilisateur.
   (c) Afficher votre login utilisateur.
   (d) Afficher le nom du répertoire courant.
   (e) Afficher le contenu du répertoire courant.
   (f) Créer un répertoire de nom `laboratoire`.
   (g) Afficher le contenu du répertoire courant.
   (h) Changer de répertoire pour passer à celui que vous venez de créer.
   (i) Afficher le nom du répertoire courant.
   (j) Créer un fichier de nom `fich1` en utilisant la commande `cat`. Le contenu du fichier est :

```
#include <iostream.h>

#define long 180
int main( )
{
    char chaine[long];
    cin.getline(chaine,long);
    cout << chaine << endl;
    return 0;
}
```

   (k) Afficher le code de protection du fichier crée.
   (l) Ajouter la permission en exécution pour le propriétaire et le groupe.
   (m) Changer le nom du fichier en prog1.cpp
   (n) Compiler le fichier avec la commande :
   `g++ -o prog1 prog1.cpp`
   (o) Exécuter le programme: `prog1`

(p) Supprimer la permission en exécution du programme `prog1.cpp` pour le groupe.
(q) Rediriger la sortie standard du programme vers un fichier nommé `resultat`.
(r) Afficher le contenu du fichier `resultat`.

4. Que fait chacune des commandes suivantes :
   (a) `man`
   (b) `ls`
   (c) `man man`
   (d) `ls *.cpp`
   (e) `file laboratoire`
   (f) `ls | sort`
   (g) `ls */*.cpp`
   (h) `ls | wc .w`
   (i)
   ```
   cat >> prog1.cpp
   ...
   //fin du programme
   ^D
   ```

5. Écrire la commande qui permet d'afficher :
   (a) les 2 premières lignes du fichier `prog1.cpp`
   (b) les 3 dernières lignes du fichier `prog1.cpp`
   (c) la 4ième ligne du fichier `prog1.cpp`
6. Éditer avec l'éditeur `vi` un programme *C/C++* nommé `prog2.cpp` qui lit une ligne tapée à l'écran et affiche le nombre de mots dans la ligne suivi de la ligne.
   (a) Compiler et exécuter le programme.
   (b) Exécuter la commande : `prog1 | prog2`
7. La commande `date` affiche la date en français (anglais). Écrire un script qui traduit la date en anglais (français). La date se présente sous la forme :
   `lundi, 4 septembre 2000, 09:38:09 EDT`
8. Écrire un script `mesure` qui constituera un fichier comportant la date et le nombre d'utilisateurs à ce moment. Un pointage sera fait toutes les 10 secondes. La commande sera utilisée de la façon suivante :
   `mesure fichier_log &`

3
Processus et threads

LE concept **processus** est le plus important dans un système d'exploitation. Tout le logiciel d'un ordinateur est organisé en un certain nombre de processus séquentiels. Un processus est un programme qui s'exécute et possède des compteurs de son exécution, des registres, des variables et une pile d'exécution. Son exécution est, en général, une alternance de calculs effectués par le processeur et de requêtes d'Entrée / Sortie effectuées par les périphériques.

3.1 Processus

3.1.1 États d'un processus

Lorsqu'un processus s'exécute, il change d'état, comme on peut le voir sur la figure 3.1. Il peut se trouver alors dans l'un des trois états principaux :

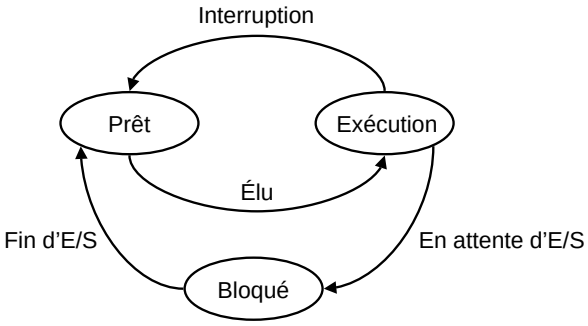


FIG. 3.1 – *Les états principaux d'un processus.*

– **Élu** : en cours d'exécution.

– **Prêt** : en attente du processeur.

– **Bloqué** : en attente d'un événement.

Initialement, un processus est à l'état prêt. Il passe à l'état exécution, lorsque le processeur entame son exécution. Un processus passe de l'état exécution à l'état prêt, lorsqu'il est suspendu provisoirement pour permettre l'exécution d'un autre processus. Il passe de l'état exécution à l'état bloqué, si le processus ne peut plus poursuivre son exécution (demande d'une E/S). Il se met alors en attente d'un événement (fin de l'E/S). Lorsque l'événement survient, il redevient prêt.

Le modèle à trois états est complété par deux états supplémentaires : **Nouveau** et **Fin**, qui indiquent respectivement la création d'un processus dans le système et son terminaison, comme illustré à la figure 3.2.

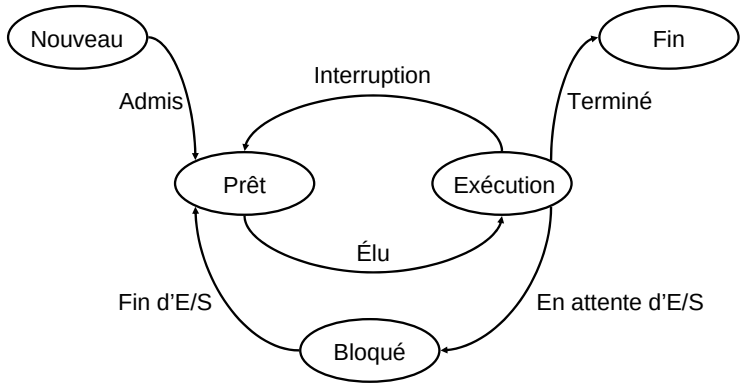


FIG. 3.2 – *Modèle à cinq états d'un processus.*

Consulter le Chapitre 4, *Ordonnancement des processus* pour plus de détails sur les états de processus en Unix/Linux.

3.1.2 Implantation de processus

Pour gérer les processus, le système d'exploitation sauvegarde dans des structures de données adéquates. Il existe une table des informations concernant tous les processus créés. Il y a une entrée par processus dans la table, appelée le **Bloc de Contrôle de Processus** (**PCB**).

Bloc de Contrôle de Processus

Chaque entrée de la table **PCB** comporte des informations sur :

- Le `pid` du processus.
- L'état du processus.
- Son compteur ordinal (adresse de la prochaine instruction devant être exécutée par ce processus).
- Son allocation mémoire.
- Les fichiers ouverts.
- Les valeurs contenues dans les registres du processeur.
- Tout ce qui doit être sauvegardé lorsque l'exécution d'un processus est suspendue.

Sur la figure 3.3, nous montrons les principales entrées de la **PCB**.

Gestion des processus	Gestion de la mémoire	Gestion des fichiers
Registres	Pointeur seg. code	Répertoire racine
Compteur ordinal (`co`)	Pointeur seg. données	Répertoire de travail
Priorité	Pointeur seg. pile	`fd`
Pointeur de pile (`sp`)		`uid`
État du processus		`gid`
Date de lancement		
Temps UCT utilisé		
Temps UCT des fils		
Signaux		
`pid`		
`ppid`		

FIG. 3.3 – *Principales entrées de la* ***Table des processus*** *d'Unix. Les entrées* `co`*, états de processus,* `pid` *et* `ppid` *seront étudiées dans la présent chapitre. Celles des descriptifs de fichiers (*`fd`*), signaux, priorité, temps, gestion de mémoire et répertoires seront étudiées prochainement.*

Changement de contexte de processus

La figure 3.4 illustre le changement de contexte entre processus. Le passage de mode utilisateur au mode *kernel* implique une gestion adéquate des **PCB**. Le changement de contexte sera expliqué plus en détail dans le Chapitre 4, *Ordonnancement de processus.*

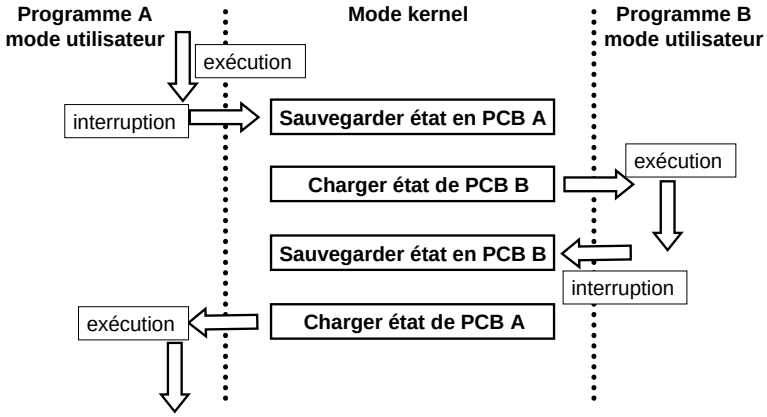


FIG. 3.4 – *Le changement de contexte.*

3.1.3 Les démons

Les **démons** sont des processus particuliers. Ils s'exécutent toujours en tâches de fond (*background*). Ceci implique que son père ne attend pas la fin de son exécution. Ils ne sont associés à aucun terminal ou processus `login` d'un utilisateur. Les démons sont toujours à l'écoute et attendent qu'un événement se produise. Ils réalisent des tâches de manière périodique.

Les démons démarrent au début du chargement du système d'exploitation et ne meurent pas. Les démons ne travaillent pas : ils lancent plutôt d'autres processus pour effectuer les tâches. Comme exemples de démons on retrouve le **voleur de pages** de la mémoire virtuelle (voir Chapitre 10 sur la gestion de la mémoire) ou le **démon des terminaux** qui lance `getty`, puis `login` à l'invite des utilisateurs.

3.1.4 Création et terminaison de processus

Le système d'exploitation fournit un ensemble d'appels système qui permettent la création, la destruction, la communication et la synchronisation des processus. Les processus sont créés et détruits dynamiquement. Un processus peut créer un ou plusieurs processus fils qui, à leur tour, peuvent créer des processus fils sous une forme de structure arborescente. Le processus créateur est appelé processus père.

Dans certains systèmes, comme MS-DOS, lorsqu'un processus crée un fils, l'exécution du processus père est suspendue jusqu'à la terminaison du

processus fils. C'est ce qu'on appelle l'**exécution séquentielle**. Par contre dans les systèmes du type Unix, le père continue à s'exécuter en concurrence avec ses fils. C'est ce qu'on appelle **exécution asynchrone**. Un processus fils créé peut partager certaines ressources comme la mémoire ou les fichiers avec son processus père ou avoir ses propres ressources. Le processus père peut contrôler l'usage des ressources partagées et peut avoir une certaine autorité sur ses processus fils. Également, il peut suspendre ou détruire ses processus fils. Il peut également se mettre en attente de la fin de l'exécution de ses fils. L'espace d'adressage du processus fils est obtenu par duplication de celui du père. Il peut exécuter le même programme que son père ou charger un autre programme. Un processus se termine par une demande d'arrêt volontaire (appel système `exit()`) ou par un arrêt forcé provoqué par un autre processus (appel système `kill()`). Lorsqu'un processus se termine toutes les ressources systèmes qui lui ont été allouées sont libérées par le système d'exploitation.

3.2 Services Posix pour la gestion de processus

Le standard Posix définit un nombre relativement petit d'appels système pour la gestion de processus.

- `pid_t fork()` : Création de processus fils.
- `int execl()`, `int execlp()`, `int execvp()`, `int execle()`, `int execv()` : Les services `exec()` permettent à un processus d'exécuter un programme (code) différent.
- `pid_t wait()` : Attendre la terminaison d'un processus.
- `void exit()` : Finir l'exécution d'un processus.
- `pid_t getpid()` : Retourne l'identifiant du processus.
- `pid_t getppid()` : Retourne l'identifiant du processus père.

En Linux, le type `pid_t` correspond normalement à un `long int`.

3.3 Création de processus

3.3.1 Création de processus avec `system()`

Il y a une façon de créer sous-processus en Unix/Linux, en utilisant la commande `system()`, de la bibliothèque standard de *C* `<stdlib.h>`. Comme arguments elle reçoit le nom de la commande (et peut-être une liste d'arguments) entre guimets.

Listing 3.1: *system.c*

```
#include <stdlib.h>

int main()
{
  int return_value;
  /* retourne 127 si le shell ne peut pas s'exécuter
     retourne -1 en cas d'erreur
     autrement retourne le code de la commande */
  return_value = system("ls -l /");
  return return_value;
}
```

Il crée un processus fils en lançant la commande :

```
ls -l /
```

dans un shell. Il faut retenir que `system()` n'est pas un appel système, mais une fonction *C*. Ce qui rend l'utilisation de la fonction `system()` moins performante qu'un appel système de création de processus. Ceci sera discuté à la fin de la section.

3.3.2 Création de processus avec `fork()`

Dans le cas d'Unix, l'appel système `fork()` est la seule véritable façon qui permet de créer des processus fils :

```
#include <unistd.h>
int fork();
```

`fork()` est le seul moyen de créer des processus, par duplication d'un processus existant [1]. L'appel système `fork()` crée une copie exacte du processus original, comme illustré à la figure 3.5. Mais maintenant il se pose un problème, car les deux processus père et fils exécutent le même code. Comment distinguer alors le processus père du processus fils ? Pour résoudre ce problème, on regarde la valeur de retour de `fork()`, qui peut être :

– **0** pour le processus fils

1. Linux a introduit un autre appel système pour la création de contexte d'un processus : `clone()`. C'est un appel système très puissant, avec un nombre d'arguments pour gérer la mémoire par le programmeur. Malheureusement `clone()` ne fait pas partie du standard Posix, et ne sera pas étudié dans notre contexte. Le lecteur intéressé peut consulter, par exemple, les articles du Linux Journal, http://www.linuxjournal.com

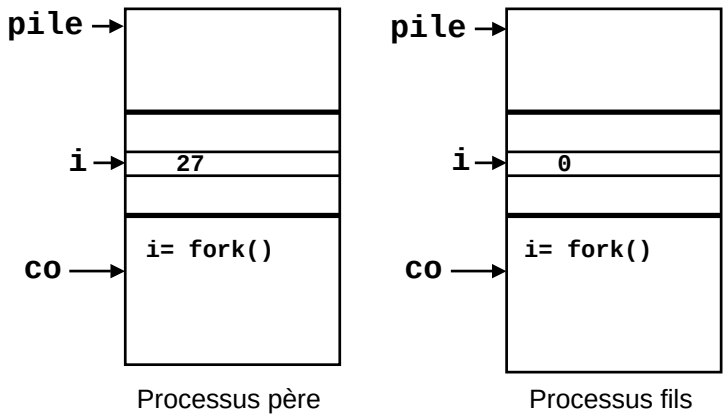


FIG. 3.5 – *Processus père et son fils cloné.* `i=27` *après l'appel* `fork()` *pour le père et* `i=0` *pour le fils. L'image mémoire est la même pour tous les deux, notamment le compteur ordinal* `co`*.*

- **Strictement positive** pour le processus père et qui correspond au `pid` du processus fils
- **Négative** si la création de processus a échoué, s'il n'y a pas suffisamment d'espace mémoire ou si bien le nombre maximal de créations autorisées est atteint.

Ce branchement est montré à la figure 3.6. Le père et le fils ont chacun leur propre image mémoire privée. Après l'appel système `fork()`, la valeur de `pid` reçoit la valeur `0` dans le processus fils mais elle est égale à l'identifiant du processus fils dans le processus père.

Observez attentivement les exemples suivants :

▶ **Exemple 1.** Après l'exécution de `fils.c` le père et le fils montrent leur `pid` :

Listing 3.2: *fils.c*

```
#include <sys/types.h>   // pour fork()
#include <unistd.h>

int main()
{
 int fils_pid;

 if ((fils_pid=fork())==0)
```

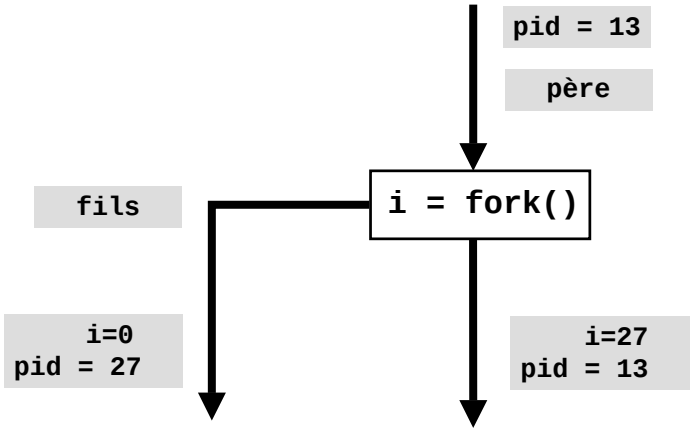


FIG. 3.6 – *Services Posix : `fork()`.*

```
      printf("Je suis le fils avec pid %d\n",getpid());
  else
     if( fils_pid > 0)
         printf("Je suis le père avec pid %d\n", getpid());
}
```

L'exécution de `fils.c` montre les `pid` du père et du fils :

```
leibnitz> gcc -o fils fils.c
leibnitz> fils
Je suis le père avec pid 4130
leibnitz> Je suis le fils avec pid 4131
```

▶ **Exemple 2.** Étudiez en détail le programme suivant. Il créera une chaîne de $n-1$ processus :

Listing 3.3: *chaine.c*

```
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
 int i, n=10, fils_pid;
```

```
  for ( i=1; i<n; i++)
  {
10    if ( fils_pid =fork ()) // c'est le père
        break;
      printf ("Processus %d avec père %d \n",getpid(), getppid());
    }
}
```

Regardez deux exécutions du programme `chaine.c`. Pourquoi ce comportement different dans chaqu'une?:

```
leibnitz> gcc -o chaine chaine.c
leibnitz> chaine
leibnitz> Processus 4336 avec père 1
Processus 4337 avec père 1
Processus 4338 avec père 1
Processus 4339 avec père 1
Processus 4340 avec père 1
Processus 4341 avec père 1
Processus 4342 avec père 4341
Processus 4343 avec père 4342
Processus 4344 avec père 4343

leibnitz> chaine
leibnitz> Processus 4346 avec père 1
Processus 4347 avec père 1
Processus 4348 avec père 4347
Processus 4349 avec père 4348
Processus 4350 avec père 4349
Processus 4351 avec père 4350
Processus 4352 avec père 4351
Processus 4353 avec père 4352
Processus 4354 avec père 4353
```

▶ **Exemple 3.** Le programme `tfork.cpp` suivant, créera deux processus qui vont modifier la variable partagée globale `a`:

Listing 3.4: *tfork.cpp*

```
#include <sys/types.h> // pour le type pid_t
#include <unistd.h>    // pour fork
#include <stdio.h>     // pour perror , printf
```

```
int a=20;
int main(void)
{
    pid_t p ;

    // création d'un fils
    switch (p = fork())
    {
        case -1:
            // le fork a échoué
            perror("le fork a échoué !");
            break;
        case 0 :
            // Il s'agit du processus fils
            printf("ici processus fils , le PID %d.\n", getpid());
            a+=10;
            break;
        default :
            // Il s'agit du processus père
            printf("ici processus père, le PID %d.\n", getpid());
            a+=100;
    }
    // les deux processus exécutent cette instruction
    printf("Fin du processus %d avec a = %d.\n", getpid(),a );
    return 0;
}
```

Deux exécutions du programme `tfork.cpp` montrent que les processus père et fils sont concurrents :

```
leibnitz> gcc -o tfork tfork.cpp
leibnitz> tfork
ici processus père, le pid 12339.
ici processus fils, le pid 12340.
Fin du Process 12340 avec a = 30.
Fin du Process 12339 avec a = 120.
leibnitz>
leibnitz> tfork
ici processus père, le pid 15301.
Fin du Process 15301 avec a = 120.
ici processus fils, le pid 15302.
Fin du Process 15302 avec a = 30.
```

```
leibnitz>
```

▶ **Exemple 4.** Les exemples suivants meritent d'être etudiés en détail. Quelle est la différence entre les programmes `fork1.c` et `fork2.c` suivants ? Combien de fils engendreront-ils chacun ?

Listing 3.5: *fork1.c*

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

int main()
{
 int i,n=5;
 int childpid;

  for (i=1; i<n; i++)
  {
   if ((childpid=fork())==-1) break;
      printf("Processus %d avec père %d, i=%d\n",getpid(),
             getppid(),i);
  }
  return 0;
}
```

Listing 3.6: *fork2.c*

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

int main()
{
 int i,n=5;
 int childpid;

 for (i=1; i<n; i++)
 {
   if ((childpid=fork())<=0) break;
      printf("Processus %d avec père %d, i=%d\n",getpid(),
```

```
                getppid(), i );
  }
  return 0;
}
```

La sortie de `fork2.c` est assez comprehensible. Le shell a un `pid` de 759, père de `fork2`, lui avec un `pid` = 904 :

```
leibnitz> gcc -o fork2 fork2.c
leibnitz> fork2
Processus 904 avec père 759, i=1
Processus 904 avec père 759, i=2
Processus 904 avec père 759, i=3
Processus 904 avec père 759, i=4
```

Par contre, la sortie de `fork1.c` est complexe. Le shell a toujours un `pid` de 759, mais `fork2` donne naissance a 15 fils. Pire encore : il y a des fils qui ont été adoptés par `init`, car son père est mort. La figure 3.7 montre l'arbre de création de processus pour cet exemple.

```
leibnitz> gcc -o fork1 fork1.c ; fork1
Processus 874 avec père 759, i=1
Processus 874 avec père 759, i=2
Processus 875 avec père 874, i=1
Processus 875 avec père 874, i=2
Processus 876 avec père 874, i=2
Processus 876 avec père 874, i=3
Processus 877 avec père 875, i=2
Processus 877 avec père 875, i=3
Processus 874 avec père 759, i=3
Processus 875 avec père 874, i=3
Processus 878 avec père 876, i=3
Processus 878 avec père 876, i=4
Processus 879 avec père 877, i=3
Processus 879 avec père 877, i=4
Processus 880 avec père 874, i=3
Processus 881 avec père 875, i=3
Processus 882 avec père 1, i=4
Processus 883 avec père 879, i=4
Processus 884 avec père 874, i=4
Processus 885 avec père 875, i=4
Processus 886 avec père 876, i=4
Processus 887 avec père 877, i=4
Processus 888 avec père 880, i=4
Processus 889 avec père 881, i=4
```

```
Processus 874 avec père 759, i=4
leibnitz> Processus 875 avec père 1, i=4
Processus 876 avec père 1, i=4
Processus 877 avec père 1, i=4
Processus 880 avec père 1, i=4
Processus 881 avec père 1, i=4
```

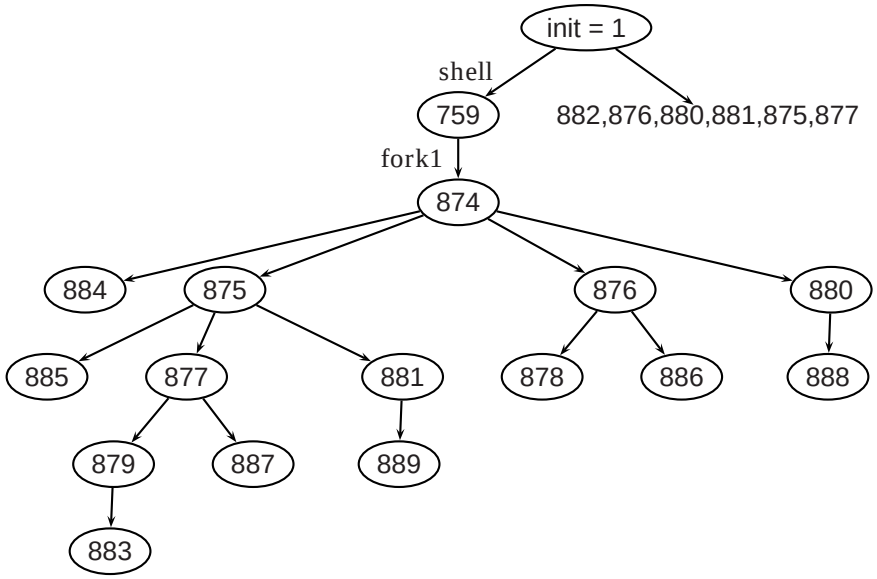


FIG. 3.7 – *L'arbre montre la création de processus de* `fork1.c`*, où les processus non encerclés ont été adoptés par* `init` *suite à la mort de leur pères.*

3.3.3 La famille des appels système `exec`

Un **processus fils** créé peut remplacer son code de programme par un autre programme. Le système Unix offre une famille d'appels système `exec` qui permettent de changer l'image d'un processus (figure 3.8). Tous les appels système `exec` remplacent le processus courant par un nouveau processus construit à partir d'un fichier ordinaire exécutable. Les segments de texte et de données du processus sont remplacés par ceux du fichier exécutable.

```
#include <unistd.h>
int execl(const char *path, const char *argv, ...);
int execv(const char *path, const char *argv[]);
```

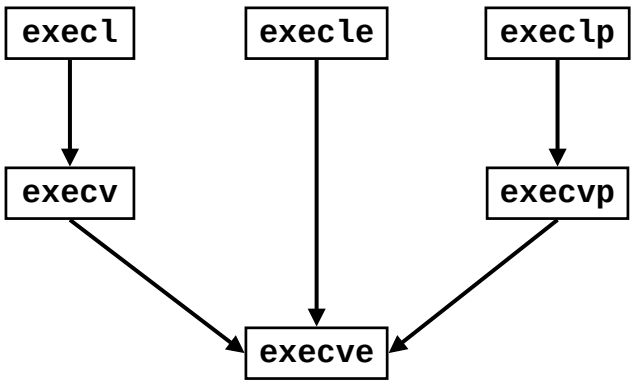


FIG. 3.8 – *Services Posix : exec.*

```
int execle(const char *path, const char *argv,
           const char *envp[]);
int execlp(const char *file, const char *argv, ...);
int execvp(const char *file, const char *argv[]);
```

- `execl()` : permet de passer un nombre fixé de paramètres au nouveau programme.
- `execv()` : permet de passer un nombre libre de paramètres au nouveau programme.
- `execle()` : même fonctionnement qu'`execl()` avec en plus, un argument `envp` qui représente un pointeur sur un tableau de pointeurs sur des chaînes de caractères définissant l'environnement.
- `exelp()` : Interface et action identiques à celles d'`execl()`, mais la différence vient du fait que si le nom du fichier n'est pas un nom complet — par rapport à la racine — le système utilisera le chemin de recherche des commandes — les chemins indiqués par la variable `PATH` — pour trouver dans quel répertoire se trouve le programme.
- `exevp()` : Interface et action identiques à celles d'`execv()`, mais la différence vient du fait que si le nom de fichier n'est pas un nom complet, la commande utilise les répertoires spécifiés dans `PATH`.

La convention Unix veut que chaque chaîne ait la forme `nom=valeur`. Ainsi, les arguments pour `execv()` peuvent être passés, par exemple :

```
char *arguments[4]
...
 arguments[0]="/bin/ls";
```

```
  arguments[1]="-l";
  arguments[2]="/etc";
  arguments[3]="NULL";
  execv("/bin/ls", arguments);
 ...
```

Après l'exécution d'un appel système de la famille `exec()`, l'image mémoire d'un processus est écrasée par la nouvelle image mémoire d'un programme exécutable, mais le `pid` ne change pas. Le contenu du contexte utilisateur qui existait avant l'appel à `exec()` n'est plus accessible.

▶ **Exemple 5.** Le programme `fork-exec.cpp` montre l'utilisation des appels système `fork()` et `execvp()` :

Listing 3.7: *fork-exec.cpp*

```
#include <unistd.h>      // pour sleep
#include <stdio.h>
#include <sys/types.h>

int  traiter (char* programme, char** args)
{
    pid_t fils_pid;

    // Dupliquer le processus
    fils_pid = fork ();
    if (fils_pid != 0) // Père
        return fils_pid;
    else
    {
      // Executer programme avec les arguments a partir du PATH
      execvp (programme, args);
      // Retourner en cas d'erreur
      perror("Erreur dans execvp");
      exit (1);
    }
}

int main(void)
{
    // Liste d'arguments pour la comande "ls"
    char* args[] =
```

```
    {
      "ls",         // argv [0] = nom du programme
      "-l",
      "/",
      NULL          // La liste doit finir avec NULL
    };

    // traite le processus fils avec la commande "ls". Ignorer le
    // pid retourné du fils
    traiter ("ls", args);
    printf("Programme principal terminé\n");
    return 0;
}
```

Exécution de `fork-exec.cpp`:

```
leibnitz> gcc -o fork-exec fork-exec.cpp
leibnitz> fork-exec
Programme principal terminé
leibnitz> total 113
drwxr-xr-x    2 root  bin     4096 Aug 22 12:00 bin
drwxr-xr-x    2 root  root    4096 Jan 20 15:46 boot
drwxr-xr-x    2 root  root    4096 Aug 22 16:43 cdrom
drwxr-xr-x   14 root  root   40960 Jan 20 16:10 dev
drwxr-xr-x   33 root  root    4096 Jan 22 15:55 etc
drwx------    2 root  root   16384 Aug 22 11:18 lost+found
drwxr-xr-x    5 root  root    4096 Mar 16  2002 mnt
...
drwxrwxrwt    8 root  root    4096 Jan 22 16:44 tmp
drwxr-xr-x   20 root  root    4096 Jan 20 16:04 usr
drwxr-xr-x   17 root  root    4096 Aug 22 16:43 var
leibnitz>
```

3.3.4 Les appels système `wait()`, `waitpid()` et `exit()`

Ces appels système permettent au processus père d'attendre la fin d'un de ses processus fils et de récupérer son status de fin. Ainsi, un processus peut synchroniser son exécution avec la fin de son processus fils en exécutant l'appel système `wait()`. La syntaxe de l'appel système est:

```
pid = wait(status);
```

où `pid` est l'identifiant du processus fils et `status` est l'adresse dans l'espace utilisateur d'un entier qui contiendra le status de `exit()` du processus fils.

```
#include <sys/wait.h>
int wait (int *status);
int waitpid(int pid, int *status, int options);
void exit(int return_code);
```

- `wait()` : Permet à un processus père d'attendre jusqu'à ce qu'un processus fils termine. Il retourne l'identifiant du processus fils et son état de terminaison dans `&status`.
- `waitpid()` : Permet à un processus père d'attendre jusqu'à ce que le processus fils numéro `pid` termine. Il retourne l'identifiant du processus fils et son état de terminaison dans `&status`.
- `void exit()` : Permet de finir volontairement l'exécution d'un processus et donne son état de terminaison. Il faut souligner qu'un processus peut se terminer aussi par un arrêt forcé provoqué par un autre processus avec l'envoi d'un **signal**[2] du type `kill()`. Une description détaillée de `kill()` et d'autres appels système se trouve dans l'Annexe C.

Le processus appelant est mis en attente jusqu'à ce que l'un de ses fils termine. Quand cela se produit, il revient de la fonction. Si `status` est différent de 0, alors 16 bits d'information sont rangés dans les 16 bits de poids faible de l'entier pointé par `status`. Ces informations permettent de savoir comment s'est terminé le processus selon les conventions suivantes :

- Si le fils est stoppé, les 8 bits de poids fort contiennent le numéro du signal qui a arrêté le processus et les 8 bits de poids faible ont la valeur octale `0177`.
- Si le fils s'est terminé avec un `exit()`, les 8 bits de poids faible de `status` sont nuls et les 8 bits de poids fort contiennent les 8 bits de poids faible du paramètre utilisé par le processus fils lors de l'appel de `exit()`.
- Si le fils s'est terminé sur la réception d'un signal, les 8 bits de poids fort de `status` sont nuls et les 7 bits de poids faible contiennent le numéro du signal qui a causé la fin du processus.

2. Les signaux et d'autres types de *Communication Interprocessus* seront étudiés au Chapitre 5.

Dans le fichier d'en-tête `<sys/wait.h>` sont definis différents macros qui permettent d'analyser la valeur de status afin de determiner la cause de terminaison du processus. En particulier, le processus père peut obtenir la valeur des 8 bits de poids faible du paramètre qui reçoit depuis `exit()` de la part du fils en utilisant le macro :

`WEXITSTATUS(status)`

La définition de ce macro se trouve dans `<sys/wait.h>`:

```
#define WEXITSTATUS(s) (((s)>>8)&0xFF)
```

Evidemment, si le processus appelant `wait()` n'a pas de processus fils vivants, une erreur est produite.

3.3.5 Processus zombie

L'exécution asynchrone entre processus parent et fils a des certaines conséquences. Souvent, le fils d'un processus se termine, mais son père ne l'attend pas. Le processus fils devient alors un **processus zombie**, comme illustré à la figure 3.9. Le processus fils existe toujours dans la table des processus, mais il n'utilise plus les ressources du *kernel*.

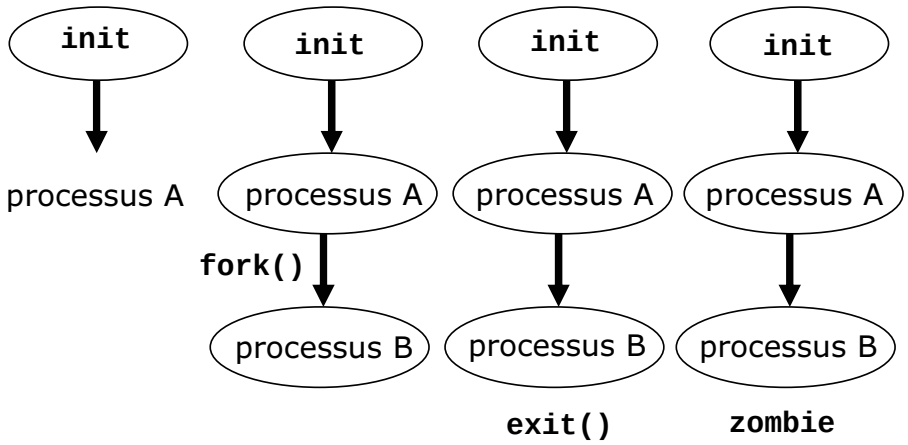


FIG. 3.9 – *Processus zombie : le fils meurt et le père n'a pas fait* `wait()`.

L'exécution de l'appel système `wait()` ou `waitpid()` par le processus père élimine le fils de la table des processus. Il peut y avoir aussi des terminaisons prématurées, où le processus parent termine avant la terminaison d'un de ses processus fils (figure 3.10). Dans cette situation, ses processus fils sont adoptés par le processus `init`, dont le `pid` = 1.

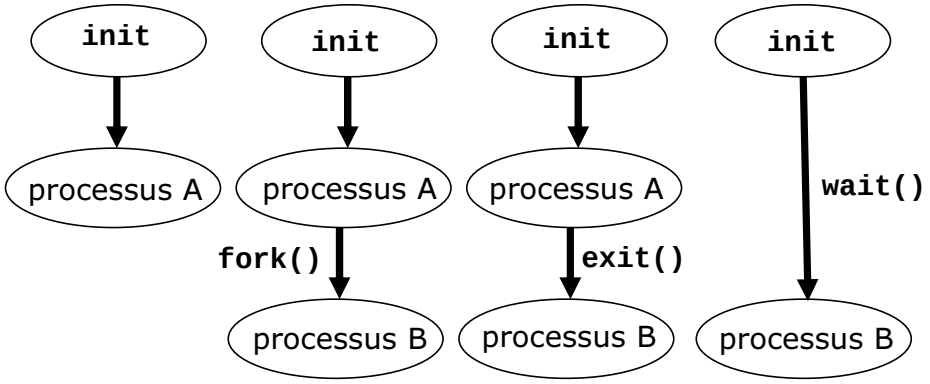


FIG. 3.10 – *Processus orphelin : le père meurt avant le fils.* `init` *adopte l'enfant.*

▶ **Exemple 6.** Engendrer des zombies est relativement facile. Le programme `zombie.c` montre la création d'un processus fils zombie pendant 30 secondes :

Listing 3.8: *zombie.c*

```
#include <unistd.h>      // pour sleep
#include <stdlib.h>
#include <sys/types.h>

int main()
{
    pid_t pid;

    // processus fils
10  pid = fork();
    if (pid > 0)
    {
      // Père : dormir 30 secondes
      sleep(30);
    }
    else
    {
      // Fils : quitter immediatement
       exit (0);
```

```
20    }
      return 0;
}
```

Exécution de `zombie.c` en tâche de fond : la commande `ps` permet de constater son état zombie avant et après son disparition définitive :

```
leibnitz> gcc -o zombie zombie.c
leibnitz> zombie &
[1] 5324
leibnitz> ps -u jmtorres
  PID TTY          TIME CMD
 2867 pts/0    00:00:00 tcsh
 5324 pts/0    00:00:00 zombie
 5325 pts/0    00:00:00 zombie <defunct>
 5395 pts/0    00:00:00 ps
leibnitz>
[1]    Done                          zombie
leibnitz>
leibnitz> ps -u jmtorres
  PID TTY          TIME CMD
 2867 pts/0    00:00:00 tcsh
 5396 pts/0    00:00:00 ps
leibnitz>
```

3.3.6 Commentaire sur `system()`

L'exécution de la fonction de bibliothèque `system()` est essentiellement un `fork()` + `exec()` + `wait()`. Et elle est peut être encore pire que cela, car elle invoque un shell pour exécuter des commandes, ce qui est très coûteux en ressources, car le shell doit charger des scripts et des initialisations. Bien que `system()` puisse s'avérer utile pour des petits programmes qui s'exécutent très occasionnellement, on la déconseille fortement pour d'autres applications. Par exemple, le seul appel :

```
system("ls -l /usr")
```

est un gaspillage incroyable de ressources. D'abord on effectue un `fork()`, puis un `exec()` pour le shell. Puis le shell s'initialise et effectue un autre `fork()` et un `exec()` pour lancer `ls`. Il est mieux d'utiliser des appels système pour la lecture des répertoires (voir par exemple le Chapitre 11 *Système de fichiers*, Section 11.5 *Services Posix sur les répertoires*.) qui sont moins coûteux que la fonction `system()`.

▶ **Exemple 7.** Les programmes `parent.cpp` et `fils.cpp` montrent la création des processus avec la combinaison `fork()` et `execl()` :

Listing 3.9: *parent.cpp*

```
#include <unistd.h>      // pour fork et execl
#include <sys/wait.h>    // pour wait
#include <stdio.h>       // pour printf

int main(void)
{
   int p, child, status;

   p=fork();
   if ( p == -1)
      return -1;
   if ( p>0) // il s'agit du père car p > 0
   {
      printf ("Ici le père [%d], mon fils[ %d]\n", getpid (), p);
      if ((child=wait(&status))>0)
         printf("Ici père [%d], fin du fils[%d]\n",getpid(),child);
                  // Dormir pendant une seconde
      sleep(1);
      printf("Le père [%d] se termine\n", getpid());
   }
   else
   { // il s'agit du fils
      if ((status=execl(
                  "/home/ada/users/jmtorres/inf3600/logiciel/a.out",
                  "a.out",
                  NULL))==-1)
         printf("désolé le programme n'existe pas : %d\n",status);
      else
         printf("cette instruction n'est jamais exécutée\n");
   }
   return 0;
}
```

Listing 3.10: *fils.cpp*

```
#include <stdio.h>
#include <unistd.h>
```

```
int main()
{
    printf("ici programme fils [%d] \n",getpid());
    return 0;
}
```

Exécution des programmes `parent.cpp` et `fils.cpp`. Remarque : Le fichier exécutable de `fils.cpp` est `a.out` :

```
leibnitz> gcc fils.cpp
leibnitz> gcc -o parent.cpp
leibnitz> parent
Ici le père 19909, mon fils 19910
ici programme fils 19910
Ici père 19909, fin du fils 19910
Le père 19909 se termine
leibnitz>
```

▶ **Exemple 8.** Utilisation d'`execvp()` :

Listing 3.11: *texecvp.cpp*

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main (int argc , char * argv[])
{
  if ( fork () == 0) // il s'agit du fils
        { // exécute un autre programme
              execvp( argv[1], &argv [1]) ;
              fprintf (stderr , "on ne peut exécuter %s\n ", argv [1]);
        }
        else if ( wait(NULL)>0 )
              printf(" Le père détecte la fin du fils \n");
  return 0;
}
```

Exécution du programme `texecvp.cpp` :

```
leibnitz> gcc -o texecvp texecvp.cpp
```

```
leibnitz> texecvp date
Wed Sep  4 15:42:06 EDT 2002
 Le père détecte la fin du fils
leibnitz> texecvp ugg+kju
on ne peut pas exécuter ugg+kju
  Le père détecte la fin du fils
leibnitz> gcc -o fils fils.cpp
leibnitz> fils
 ici programme fils [20498]
leibnitz> texecvp fils
 ici programme fils [20500]
 Le père détecte la fin du fils
leibnitz>
```

▶ **Exemple 9.** Création de deux fils (version 1), et utilisation des 8 bits de poids fort :

Listing 3.12: *deuxfils-1.c*

```
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>

void fils(int i);
int main()
{
   int status;
   if(fork()) // création du premier fils
10       {
         if(fork()==0) // création du second fils
            fils (2);
         }
   else fils (1);
   if(wait(&status)>0)
         printf("fin du fils %d\n", status »8);
   if (wait(&status)>0)
         printf("fin du fils %d\n", status »8);
   return 0;
20 }

void fils(int i)
{
```

```
    sleep (2) ;
    exit(i ) ;
}
```

Deux exécutions du programme `deuxfils-1.c`:

```
leibnitz> gcc -o deuxfils-1 deuxfils-1.c
leibnitz> deuxfils-1
fin du fils 2
fin du fils 1
leibnitz> deuxfils-1
fin du fils 1
fin du fils 2
leibnitz>
```

▶ **Exemple 10.** Création de deux fils (version 2), avec la simulation d'attente d'un troisième fils inexistant et l'utilisation du macro `WEXITSTATUS`:

Listing 3.13: *deuxfils-2.c*

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void fils (int i);
int main()
{
  int    status;
  pid_t pid;

  // Creation du premier fils
  pid = fork();
  if (pid == -1) perror("fork");
  else if (pid == 0) fils (1);

  // Creation du second fils
  pid = fork();
  if (pid == -1) perror("fork");
  else if (pid == 0) fils (2);

  // Attente de terminaison des deux fils
```

```
  if(wait(&status)>0)
    printf("fin du fils %d\n", WEXITSTATUS(status));
  else perror("wait");

  if(wait(&status)>0)
    printf("fin du fils %d\n", WEXITSTATUS(status));
  else perror("wait");

  // Ici on attend la fin d'un troisieme fils qui n'existe pas
  // Une erreur sera retournee
  if (wait(&status)>0)
    printf("fin du fils %d\n", WEXITSTATUS(status));
  else perror("wait");

  return 0;
}

// Routine pour les fils
void fils (int i)
{
    sleep(2);
    exit(i);
}
```

Exécution du programme `deuxfils-2.c`:

```
leibnitz> gcc -o deuxfils-2 deuxfils-2.c
leibnitz> deuxfils-2
fin du fils 2
fin du fils 1
wait: No child processes
leibnitz>
```

3.4 Threads

Le modèle de processus décrit précédemment est un programme qui s'exécute selon un chemin unique avec un seul compteur ordinal. On dit qu'il a un **flot de contrôle unique** ou un seul **thread**. De nombreux systèmes d'exploitation modernes offrent la possibilité d'associer à un même

processus plusieurs chemins d'exécution ou multithread (figure 3.11). Ils permettent ainsi l'exécution simultanée des parties d'un même processus. Chaque partie correspond à un chemin d'exécution du processus. Le processus est vu comme étant un ensemble de ressources (code exécutable, segments de données, de fichiers, de périphériques, etc.) que ces parties appelées flots de contrôle ou processus légers (*threads* en anglais) partagent. Chaque flot de contrôle (thread) a cependant, en plus des ressources communes, sa propre zone de données ou de variables locales, sa propre pile d'exécution, ses propres registres et son propre compteur ordinal.

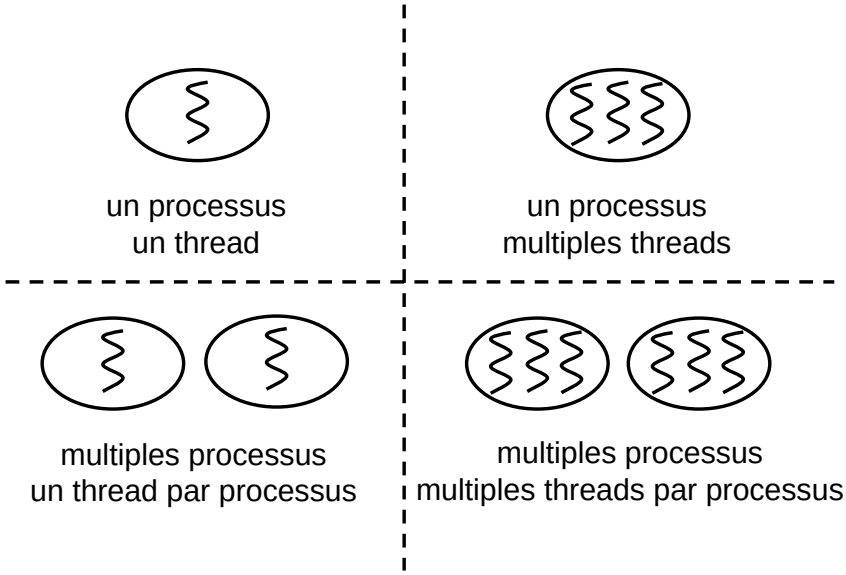


FIG. 3.11 – *Threads vs. processus.*

3.4.1 Avantages

Comparativement aux processus à un flot de contrôle unique, un *thread* ou processus léger avec plusieurs flots de contrôle présente plusieurs avantages, notamment :

- **Réactivité**. Le processus léger continue à s'exécuter même si certaines de ses parties sont bloquées.
- **Partage de ressources**.

– **Économie d'espace mémoire et de temps**. Par exemple sous Solaris, la création d'un processus est 30 fois plus lente que celle d'un processus thread.

3.4.2 Threads utilisateur et noyau

La majorité des systèmes permettent le **multiflot** (*multithreading*). Ils sont offerts soit au niveau utilisateur, soit au niveau noyau (voir le Chapitre 4 *Ordonnancement des processus*, pour les détails sur l'ordonnancement des threads et des processus).

Les threads utilisateur sont supportés au-dessus du noyau et sont implantés par une bibliothèque de threads au niveau utilisateur (par exemple `pthread` sous Linux ou `thread` dans Solaris). Ils sont portables sur différentes plate-formes. Ils sont gérés par une application où le blocage du thread peut bloquer le processus complet. Le changement de contexte est rapide.

Les threads noyau sont directement supportés par le noyau du système d'exploitation. Le système d'exploitation se charge de leur gestion et le changement de contexte est lent.

Les threads combinés sont implantés par le système d'exploitation (utilisateur et système). Les threads utilisateur sont associés à des threads système. La plupart des tâches gestion s'effectuent sous le mode utilisateur.

Les opérations concernant les threads sont notamment la création, la terminaison, la suspension et la relance.

3.5 Services Posix de gestion de threads

Linux ne fait pas de distinction entre les processus et les threads. Un thread est un processus qui partage un certain nombre de ressources avec le processus créateur : l'espace d'adressage, les fichiers ouverts ou autres. Pour la gestion **Posix** de threads, Linux utilise la bibliothèque `pthread`, qui doit être appelée par l'éditeur de liens.

Création de threads

```
int pthread_create (pthread_t *thread ,
                    pthread_attr_t *attr,
                    void *nomfonction,
```

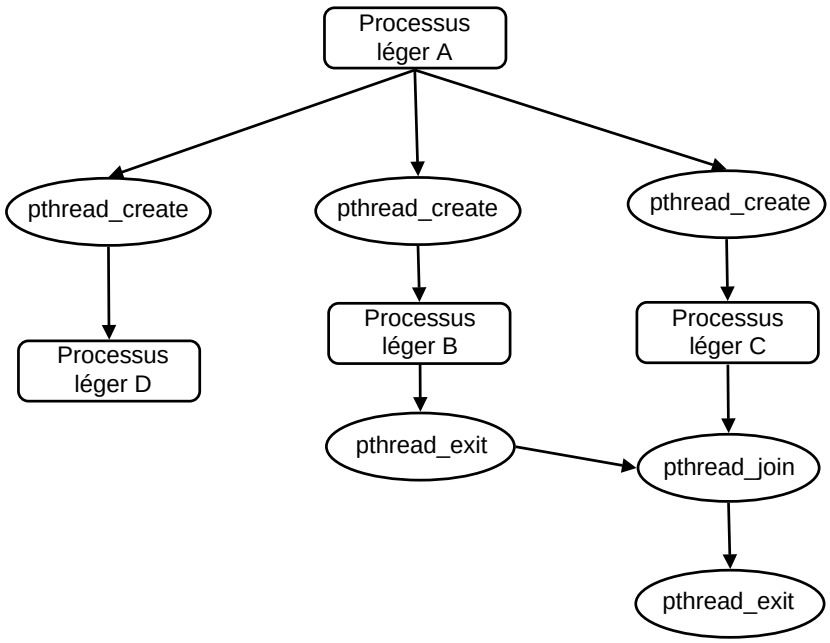


FIG. 3.12 – *Hiérarchie de processus légers.*

```
                void *arg );
```

Le service `pthread_create()` crée un processus léger qui exécute la fonction `nomfonction` avec l'argument `arg` et les attributs `attr`. Les attributs permettent de spécifier la taille de la pile, la priorité, la politique de planification, etc. Il y a plusieurs formes de modification des attributs.

Suspension de threads

```
int pthread_join(pthread_t *thid,  void **valeur_de_retour);
```

`pthread_join()` suspend l'exécution d'un processus léger jusqu'à ce que termine le processus léger avec l'identificateur `thid`. Il retourne l'état de terminaison du processus léger.

Terminaison de threads

```
void pthread_exit(void *valeur_de_retour);
```

`pthread_exit()` permet à un processus léger de terminer son exécution, en retournant l'état de terminaison.

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
```

`pthread_attr_setdetachstate()` sert à établir l'état de terminaison d'un processus léger :

- Si `detachstate = PTHREAD_CREATE_DETACHED` le processus léger libérera ses ressources quand il terminera.
- Si `detachstate = PTHREAD_CREATE_JOINABLE` le processus léger ne libérera pas ses ressources. Il sera donc nécessaire d'appeler `phtread_join()`.

▶ **Exemple 11.** Le programme `thread-pid.c` montre l'implantation de threads dans GNU/Linux, ainsi que la récupération du `pid` du thread.

Listing 3.14: *thread-pid.c*

```
#include <unistd.h>     // pour sleep
#include <pthread.h>    // pthread_create , pthread_join , pthread_exit
#include <stdio.h>

void *fonction(void *arg)
{
   printf("pid du thread fils = %d\n", (int) getpid());

   while(1); // forever
   return NULL;
}

int main()
{
 pthread_t thread;

  printf("pid de main = %d\n", (int) getpid());
  pthread_create(&thread, NULL, &fonction, NULL);
  while(1); // forever

 return 0;
}
```

Étant donné que aussi bien `main()` que la `fonction_thread()` tournent tout le temps, cela nous permet de regarder de près des détails de leur exécution. Le programme doit être compilé avec la librairie `-lpthread` L'exécution en tâche de fond de `pthread-pid.c` est :

```
leibnitz> gcc -o pthread-pid thread-pid.c -lpthread
leibnitz> pthread-pid &
[1] 24133
leibnitz> pid de main = 24133
pid du thread fils = 24136
leibnitz> ps x
 PID TTY      STAT   TIME COMMAND
23928 pts/2    S      0:00 -tcsh
24133 pts/2    R      0:53 pthread-pid
24135 pts/2    S      0:00 pthread-pid
24136 pts/2    R      0:53 pthread-pid
24194 pts/2    R      0:00 ps x
```

Le `pid 24133` correspond au processus de `main()`, `24136` est le PID du `fonction_thread()` elle même, mais qui est le processus `24135` qui — en plus— dort ? D'un autre coté, la seule façon de terminer l'exécution du programme précèdent, c'est l'utilisation de la commande `kill`, avec le `pid` correspondant :

```
leibnitz> kill 24133
leibnitz> [1]    Terminated    pthread-pid
```

▶ **Exemple 12.** Le programme `threads.cpp` montre l'utilisation des variables globales, ainsi que le passage de paramètres lors des appels de threads.

Listing 3.15: *threads.cpp*

```
#include <unistd.h>     // sleep
#include <pthread.h>    // pthread_create , pthread_join , pthread_exit
#include <stdio.h>

int glob=0;
void* increment(void *); void* decrement(void *);

int main()
{
        pthread_t tid1, tid2;
```

```
        printf("ici  main[%d], glob = %d\n", getpid(),glob);

    // création d'un thread pour increment
    if ( pthread_create(&tid1, NULL, increment, NULL) != 0)
            return -1;
    printf( "ici  main : creation du thread[%d] avec succès\n", tid1);

    // création d'un thread pour decrement
    if ( pthread_create(&tid2, NULL, decrement, NULL) != 0)
            return -1;
    printf( "ici  main : creation du thread [%d] avec succès\n", tid2);

    // attendre la fin des threads
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("ici  main : fin des threads, glob = %d \n",glob);
  return 0;
}

void* decrement(void *)
{
    int dec=1;
    sleep(1);
    glob= glob - dec ;
    printf("ici  decrement[%d], glob = %d\n", getpid(), glob);
    pthread_exit(NULL);
}

void* increment (void *)
{
    int inc=2;
    sleep(10);
    glob=glob + inc;
    printf("ici  increment[%d], glob = %d\n",getpid(), glob);
    pthread_exit(NULL);
}
```

Résultat du programme `threads.cpp`:

```
leibnitz> gcc -o pthread threads.cpp -lpthread
leibnitz> pthread
```

```
ici main[21939], glob = 0
ici main : creation du thread[1026] avec succès
ici main : creation du thread [2051] avec succès
ici increment[21941], glob = 2
ici decrement[21942], glob = 1
ici main : fin des threads, glob = 1
leibnitz>
```

Le programme `thread-param.c` suivant illustre le passage de paramètres entre threads.

▶ **Exemple 13.** Passage de paramètres.

Listing 3.16: *thread-param.c*

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

void *thread_function(void *arg);
char message[] = "Hola mundo";

int main()
{
        int res;
        pthread_t a_thread;
        void *thread_result;

        res = pthread_create(&a_thread, NULL, thread_function,
                (void *)message);
        if (res != 0)
        {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }

        printf("En attente que le thread termine...\n");
        res = pthread_join(a_thread, &thread_result);
        if (res != 0)
        {
            perror("Echec dans l'union du thread");
```

```
            exit(EXIT_FAILURE);
        }
        printf("Thread termine %s\n", (char *)thread_result);
        printf("Le message est %s\n", message);
        exit(EXIT_SUCCESS);
}

void *thread_function(void *arg)
{
        printf("La fonction du thread demarre. L'argument est '%s'\n",
                (char *) arg);
        sleep(5);
        strcpy(message, "Adios!");
        pthread_exit("Merci par le temps de CPU");
}
```

Résultat du programme `thread-param.c`:

```
leibnitz> gcc -o thread-param thread-param.c -lpthread
leibnitz> thread-param
leibnitz> Le thread demarre. L'argument est 'Hola mundo'
En attente que le thread termine...
Thread terminé Merci pour le temps de l'UCT
Le message est Adios!
leibnitz>
```

Algorithmes vectoriels

Il y a plusieurs exemples d'utilisation des threads pour le traitement parallèle de l'information. Un **serveur de fichiers**[3] par exemple, nécessite la gestion et la manipulation de plusieurs requêtes en même temps, qui peuvent être effectuées par l'utilisation des threads. Ainsi, la création d'une thread à chaque requête est envisageable : les threads effectuent les traitements nécessaires et envoient les résultats au client.

La manipulation des matrices et des vecteurs est un autre champ particulièrement propice pour les algorithmes parallèles. La multiplication de deux matrices **a** et **b** de m colonnes = n lignes = 3, par exemple, peut être parallélisé, puisque :

3. Le modèle client-serveur sera étudié au Chapitre 14 *Introduction aux systèmes distribués*

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} =$$

$$\begin{pmatrix} a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31} & a_{11}b_{12}+a_{12}b_{22}+a_{13}b_{32} & a_{11}b_{13}+a_{12}b_{23}+a_{13}b_{33} \\ a_{21}b_{11}+a_{22}b_{21}+a_{23}b_{31} & a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32} & a_{21}b_{13}+a_{22}b_{23}+a_{23}b_{33} \\ a_{31}b_{11}+a_{32}b_{21}+a_{33}b_{31} & a_{31}b_{12}+a_{32}b_{22}+a_{33}b_{32} & a_{31}b_{13}+a_{32}b_{23}+a_{33}b_{33} \end{pmatrix}$$

Ainsi, chaque élément de la matrice résultat peut être un thread. On aura donc 9 thread dans cet exemple.

▶ **Exemple 14.**

Le programme 3.17 `suma-mat.c` illustre l'utilisation des vecteurs de threads pour faire la somme de deux matrices `A` et `B` de `N` éléments. La somme de chaque ligne est calculée avec des threads indépendants en parallèle. Les composantes sont générées aléatoirement avec `srand()`.

Listing 3.17: *suma-mat.c*

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 4                          // Taille des matrices
#define M 10                         // Nb Maximum

void *somme_vectorielle(void *arg);    // Somme des matrices
void imprime(int matrice[N][N]);
pthread_t tid[N];                    // Vecteurs threads
int A[N][N], B[N][N];                // Matrices
int C[N][N];                         // C=A+B
int fila ;                           // La file des 2 matrices

int main(void)
{
 int j, k, semilla;
 // Generation des matrices
 semilla = time(NULL);
 srand(semilla);
 for(j=0; j<=N-1; j++)
  for(k=0; k<=N-1; k++)
```

```
      {
        A[j][k] = rand()%(M); // aleatoire 0-M
        B[j][k] = rand()%(M);
      }
   for(j=0; j<=N-1; j++)
    {
      printf("Thread file %d\n",j);
      pthread_create(&tid[j], NULL, somme_vectorielle, (void *) j);
    }
   for(j=0; j<=N-1; j++)
   {
     printf("Attendre thread %d\n",j);
     pthread_join(tid[j], NULL);
   }
   printf("\n\nMatrice A");
   imprime(A);
   printf("\n\nMatrice B");
   imprime(B);
   printf("\n\n Matriz C=A+B");
   imprime(C);
 return 0;
}

void imprime(int matrice[N][N])
{
 int j, k;
 for(j=0; j<=N-1; j++)
 {
  printf("\n");
  for(k=0; k<=N-1; k++)
    printf("%d \t",matrice[j][k]);
 }
}

void *somme_vectorielle(void *arg)
{
 int f, k;
 f = (int) arg;
 printf("Somme de file %d\n",f);
 for(k=0; k<=N-1; k++)
```

```
    C[f][k] = A[f][k]+B[f][k];

 pthread_exit(NULL);
}
```

Exécution du programme `suma-mat.c`:

```
leibnitz> suma-mat
Thread file 0
Thread file 1
Thread file 2
Somme de file 0
Somme de file 2
Thread file 3
Somme de file 3
Attendre thread 0
Attendre thread 1
Somme de file 1
Attendre thread 2
Attendre thread 3

Matrice A
6       6       0       9
5       5       7       7
2       7       0       6
4       1       6       5

Matrice B
0       3       9       7
9       3       5       6
6       8       6       4
4       8       2       3

Matriz C=A+B
6       9       9       16
14      8       12      13
8       15      6       10
8       9       8       8       leibnitz>
```

3.6 Threads sous Unix (Solaris 2)

Sous Unix System V de Solaris existe la bibliothèque `thread` pour la gestion de processus légers. Le processus et ses threads ont leur propre

`pid`. Chaque thread a en plus un `tid`.

▶ **Exemple 15.** Le programme 3.18 `threads.cpp` montre l'utilisation des variables globales et le passage de paramètres.

Listing 3.18: *threads.cpp*

```
#include <unistd.h>     // sleep
#include <pthread.h>    // pthread_create , pthread_join , pthread_exit
#include <stdio.h>

int glob=0;
void* increment(void *); void* decrement(void *);

int main()
{
        pthread_t tid1, tid2;
        printf("ici  main[%d], glob = %d\n", getpid(),glob);

      // création d'un thread pour increment
      if ( pthread_create(&tid1, NULL, increment, NULL) != 0)
            return -1;
      printf( "ici  main : creation du thread[%d] avec succès\n", tid1);

      // création d'un thread pour decrement
      if ( pthread_create(&tid2, NULL, decrement, NULL) != 0)
            return -1;
      printf( "ici  main : creation du thread [%d] avec succès\n", tid2);

      // attendre la fin des threads
      pthread_join(tid1,NULL);
      pthread_join(tid2,NULL);
      printf("ici  main : fin des threads, glob = %d \n",glob);
   return 0;
}

void* decrement(void *)
{
      int dec=1;
      sleep (1);
      glob= glob - dec ;
```

```
        printf(" ici  decrement[%d], glob = %d \n", getpid(), glob);
        pthread_exit(NULL);
}

void* increment (void *)
{
        int inc=2;
        sleep(10);
        glob=glob + inc;
        printf(" ici  increment[%d], glob = %d \n",getpid(), glob);
        pthread_exit(NULL);
}
```

Trois exécutions du programme `threads.cpp`, sur la machine Unix `jupiter` :

```
jupiter% gcc -o threads threads.cpp -lthread
jupiter% threads
ici main[18745], glob = 0
ici increment[18745], glob = 2
ici decrement[18745], glob = 1
ici main, fin des threads, glob = 1
jupiter% threads
ici main[18751], glob = 0
ici decrement[18751], glob = -1
ici increment[18751], glob = 1
ici main, fin des threads, glob = 1
jupiter% threads
ici main[18760], glob = 0
ici increment[18760], glob = 1
ici decrement[18760], glob = 1
ici main, fin des threads, glob = 1
```

▶ **Exemple 16.**

Le programme 3.19 `threads2.cpp` effectue le passage de paramètres à un thread :

Listing 3.19: *threads2.cpp*

```
#include <unistd.h> //pour  sleep
#include <thread.h> //  thr_create  et  thr_join
#include <stdio.h>

int glob=0;
```

```
void* increment(void *inc);
void* decrement(void *dec);

int main()
{
    int inc = 2, dec = 1;
    printf("ici main[%d], glob = %d\n", getpid(),glob);
    thr_create(NULL,0,increment, &inc,0,NULL);
    thr_create(NULL,0,decrement, &dec,0,NULL);
    while((thr_join(NULL,NULL,NULL)==0));
    printf("ici main, fin des threads, glob = %d \n",glob);
    return 0;
}

void* decrement(void *dec)
{
    int locd=*(int *) dec;
    sleep(1);
    glob -= locd;
    printf("ici decrement[%d], glob = %d\n",getpid(),glob);
    return (NULL);
}

void* increment (void *inc)
{
    int loci=*(int *) inc;
    sleep(1);
    glob += loci;
    printf("ici increment[%d], glob = %d\n",getpid(), glob);
    return (NULL);
}
```

Exécution programme `threads2.cpp`:

```
jupiter% gcc threads2.cpp -lthread
jupiter% a.out
ici main[19451], glob = 0
ici decrement[19451], glob = 1
ici increment[19451], glob = 2
ici main, fin des threads, glob = 1
```

Suggestions de lecture

Référence : Silverwschatz A., Galvin P., Gagné G., *Applied Operating System Concepts*, Wiley, 2003.

Chapître 4

Section 4.1 Process Concept.

Section 4.3 Operations on Processes.

Chapître 5

Section 5.1 Overview.

Section 5.2 Benefits.

Section 5.3 User and Kernel Threads.

Section 5.4 Multithreading Models.

Section 20.5 Process management : lire la sous-section 20.5.1 Process Control Blocks.

3.7 Exercices

1. Dans le système Unix, est-ce que tout processus a un père?
2. Que se passe-t-il lorsqu'un processus devient orphelin (mort de son père)?
3. Quand est-ce qu'un processus passe à l'état **zombie**?
4. Pour lancer en parallèle plusieurs traitements d'une même application, vous avez le choix entre les appels système `pthread_create()` et `fork()`. Laquelle des deux possibilités choisir? Pourquoi?
5. Qu'affiche chacun des segments de programme suivants:

(a)
```
for (i=1; i<=4; i++)
{
    pid = fork();
    if (pid != 0) printf("%d\n", pid);
}
```

(b)
```
for (i=1; i<=4; i++)
{
    pid = fork();
    if (pid == 0) break;
    else printf("%d\n", pid);
}
```

(c)
```
for (i=0; i<nb; i++)
{
    p = fork();
    if (p < 0) exit(1);
    execlp("prog", "prog",NULL);
}
wait(&status);
```

(d)
```
for (i=1; i<=nb; i++)
{
    p1 = fork();
    p2 = fork();
    if (p1 < 0) exit(1);
    if (p2 < 0) exit(1);
    execlp("prog1", "prog1",NULL);
    execlp("progc", "prog",NULL);
}
wait(&status);
```

(e) Quel est le nombre de processus créés, dans chaque cas?

6. Écrire un programme qui lit à l'écran le nombre de fils à créer, puis les crée l'un à la suite de l'autre. Chaque fils affiche à l'écran son `pid` (`getpid()`) et celui de son père (`getppid()`). Le processus créateur doit attendre la fin de ses fils. Lorsqu'il détecte la fin d'un fils, il affiche le `pid` du fils qui vient de se terminer.
7. Écrire un programme qui lance en créant un processus le programme `prog2.cpp`, puis se met en attente de la fin d'exécution du programme.
8. On sait qu'un thread est créé par un processus parent, mais quelle différence existe-il entre un thread et un processus fils?
9. Quelle serait un avantage principal à utiliser des threads plutôt que des processus fils?
10. Expliquez un exemple d'exécution de programme dans lequel il existe de l'incohérence entre les trois niveaux de mémoire, *i.e.* registres, mémoire principale, et mémoire secondaire.
11. Lorsqu'un nouveau processus passe en état d'exécution, tous les registres de l'UCT doivent être initialisés ou restaurés avec les valeurs au moment de l'interruption du processus. Expliquez la raison pour laquelle le registre compteur ordinal (i.e. *program counter*) est le dernier à être initialisé.
12. Quelle différence existe-il entre un programme exécutable et un processus?
13. Est-ce qu'un processus peut exécuter plusieurs programmes, et est-ce qu'un programme peut être exécuté par plusieurs processus à la fois? Expliquez.
14. Considérez un système ayant les caractéristiques suivantes :
    (a) Système d'exploitation multiprogrammé.
    (b) Un seul UCT.
    (c) Nombre limité de ressources.
    (d) Un processus peut quitter volontairement l'état d'exécution pour revenir à l'état prêt.
    (e) Le système peut bloquer à tout moment un processus en état d'exécution s'il a besoin des ressources utilisées par ce processus.

    Dessinez le diagramme d'état de ce système.

4
Ordonnancement des processus

Dans un système multi-utilisateurs à temps partagé, plusieurs processus peuvent être présents en mémoire centrale en attente d'exécution. Si plusieurs processus sont prêts, le système d'exploitation doit gérer l'allocation du processeur aux différents processus à exécuter. C'est l'ordonnanceur qui s'acquitte de cette tâche.

4.1 Introduction

Tout le logiciel d'un ordinateur peut être vu comme un ensemble de processus dont l'exécution est gérée par un processus particulier : l'ordonnanceur (*scheduler* en anglais). Un ordonnanceur fait face à deux problèmes principaux :

– le choix du processus à exécuter, et
– le temps d'allocation du processeur au processus choisi.

Un système d'exploitation multitâche est *préemptif* lorsque celui-ci peut arrêter (réquisition) à tout moment n'importe quelle application pour passer la main à la suivante. Dans les systèmes d'exploitation préemptifs on peut lancer plusieurs applications à la fois et passer de l'une à l'autre, voire lancer une application pendant qu'une autre effectue un travail.

Il y a aussi des systèmes d'exploitation dits multitâches, qui sont en fait des « multi-tâches coopératifs ». Quelle est la différence ? Un *multitâche coopératif* permet à plusieurs applications de fonctionner et d'occuper des plages mémoire, laissant le soin à ces applications de gérer cette occupation, au risque de bloquer tout le système. Par contre, avec un « multi-tâche préemptif », le noyau garde toujours le contrôle (qui fait quoi, quand et comment), et se réserve le droit de fermer les applications qui monopolisent les ressources du système. Ainsi les blocages du système sont inexistantes.

4.2 Types d'ordonnanceurs

Il est possible de distinguer trois types d'ordonnanceurs : à long terme, à moyen terme et a court terme. Leurs principales fonctions sont les suivantes :

À long terme : L'ordonnanceur fait la sélection de programmes à admettre dans le système pour leur exécution. Les programmes admis deviennent des processus à l'état **prêt**. L'admission dépend de la capacité du système (degré de multiprogrammation) et du niveau de performance requis.

À moyen terme : Il fait la sélection de processus déjà admis à débarquer ou rembarquer sur la mémoire. Il effectue ses tâches de géstion en fonction du degré de multiprogrammation du système, et aussi en base aux requêtes d'E/S des périphériques.

À court terme : L'ordonnanceur à court terme a comme tâche la gestion de la file des processus prêts. Il décide — en fonction d'une certaine politique — la sélection du prochain processus à exécuter. Il effectue aussi le changement de contexte des processus. Il peut implanter un ordonnancement préemptif, non préemptif, ou coopératif. L'ordonnanceur est activé par un événement : interruption du temporisateur, interruption d'un périphérique, appel système ou signal.

À partir du modèle à 5 états montré sur la figure 3.2 du chapitre 3, l'ordonnanceur à moyen terme introduit le concept d'**états actifs** et **non actifs**, et deux états supplémentaires : **Prêt en suspension** et **Attente en suspension**.

Prêt en suspension : Un processus qui est **prêt à s'exécuter** est mis en **prêt en suspension** (envoyé en mémoire secondaire). Un processus **prêt en suspension** ne peut pas s'exécuter en UCT car il n'est pas en mémoire principale.

Attente en suspension : Un processus qui attend un événement est mis en **attente en suspension** (swap)

En fonction de l'ordonnanceur implanté, il existe un certain nombre d'états supplémentaires. La figure 4.1 illustre les états actifs et non actifs. Les premiers sont en compétition par le processeur. Les seconds ne sont pas encore en compétition par le processeur, car ils attendent la fin d'un événement d'Entrée/Sortie, mais dès que ceci sera terminé, ils deviendront des états actifs. Sur la figure 4.2 on montre les états d'un processus sous

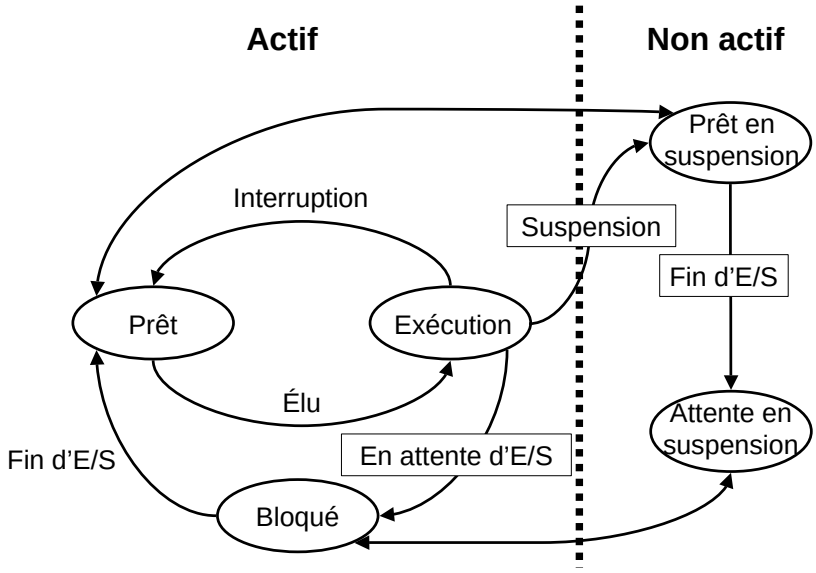


FIG. 4.1 – *Modèle des états actifs et non actifs.*

Unix / Linux.

Attente en suspension → bloqué : Un processus bloqué ne peut pas s'exécuter de toute façon, ainsi le placer dans la mémoire semble gaspiller cette ressource. Dans quelques contextes, la décision peut être appropriée : le processus peut être un processus prioritaire ou le système peut être légèrement chargé avec une quantité de mémoire significative et inutilisée, disponible. Dans l'un ou l'autre cas, déplacer un processus **attente en suspension** (*swapped-blocked*) à l'état **bloqué** (ou attente active) permet au processus d'exécuter plus tôt, une fois que la condition de blocage est satisfaite.

Prêt → prêt en suspension ou bloqué → attente en suspension : L'ordonnanceur à moyen terme met un processus en stockage temporaire (*swap*) pour laisser mémoire à d'autres processus.

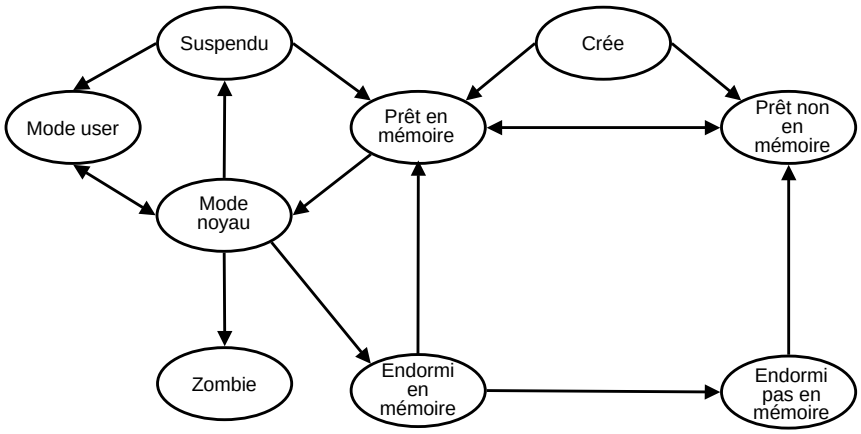


FIG. 4.2 – *États d'un processus sous Unix/Linux.*

4.3 Objectifs de l'ordonnanceur d'un système multi-utilisateur

Les objectifs d'un ordonnanceur d'un système multi-utilisateur sont, entre autres :

- S'assurer que chaque processus en attente d'exécution reçoit sa part de temps processeur.
- Minimiser le temps de réponse.
- Utiliser le processeur à 100%.
- Utilisation équilibrée des ressources.
- Prendre en compte des priorités.
- Être predictibles.

Ces objectifs sont parfois complémentaires, parfois contradictoires : augmenter la performance par rapport à l'un d'entre eux peut diminuer la performance considérée à partir d'autre. Il est impossible de créer un algorithme qui optimise tous les critères de façon simultanée.

4.4 Ordonnanceurs non préemptifs

Dans un système à **ordonnancement non préemtif** ou sans réquisition, le système d'exploitation choisit le prochain processus à exécuter, en général, le **Premier Arrivé est le Premier Servi** PAPS (ou *FCFS First-Come*

First-Served) ou le **plus court d'abord** (*SJF Short Job First*). Il lui alloue le processeur jusqu'à ce qu'il se termine ou il se bloque (en attente d'un événement). Il n'y a pas de réquisition.

Si l'ordonnanceur fonctionne selon la stratégie **SJF**, il choisit parmi le lot de processus à exécuter, le plus court d'abord (plus petit temps d'exécution). Cette stratégie est bien adaptée au traitement par lots de processus dont les temps maximaux d'exécution sont connus ou fixés par les utilisateurs car elle offre un meilleur temps moyen de séjour. Le temps de séjour d'un processus (temps de rotation ou de virement) est l'intervalle de temps entre la soumission du processus et son achèvement.

Considérons par exemple un lot de quatre processus dont les temps respectifs d'exécution sont a, b, c et d. Le premier processus se termine au bout du temps a ; le deuxième processus se termine au bout du temps $a+b$; le troisième processus se termine au bout du temps $a+b+c$; le quatrième processus se termine au bout du temps $a+b+c+d$. Le temps moyen de séjour $< t >$ est : $< t >= \frac{4a+3b+2c+d}{4}$. On obtient un meilleur temps de séjour pour $a \leq b \leq c \leq d$. Toutefois, l'ordonnancement du plus court d'abord est optimal que si les travaux sont disponibles simultanément.

▶ **Exemple 1.** Considérons cinq travaux A, B, C, D et E, dont les temps d'exécution et leurs arrivages respectifs sont donnés dans la table 4.1. Faire un schéma qui illustre son exécution et calculer le temps de séjour de chaque processus, le temps moyen de séjour, le temps d'attente et le temps moyen d'attente en utilisant :

1. Premier arrivé premier servi (**PAPS**)
2. Le plus court d'abord (**SJF**)

Processus	Temps d'exécution	Temps d'arrivage
A	3	0
B	6	1
C	4	4
D	2	6
E	1	7

TAB. 4.1 – *Données d'ordonnancement.*

1. PAPS. Schéma d'exécution :

A	A	A	B	B	B	B	B	B	C	C	C	C	D	D	E
1				5					10					15	

Au temps 0, seulement le processus A est dans le système et il s'exécute. Au temps 1 le processus B arrive mais il doit atteindre que A

termine car il a encore 2 unités de temps. Ensuite B s'exécute pendant 4 unités de temps. Au temps 4, 6, et 7 les processus C, D et E arrivent mais B a encore 2 unités de temps. Une fois que B a terminé, C, D et E entrent au système dans l'ordre [1].

Le temps de séjour pour chaque processus est obtenu soustrayant le temps d'entrée du processus du temps de terminaison. Ainsi :

Processus	Temps de sejour
A	3-0 = 3
B	9-1 = 8
C	13-4 = 9
D	15-6 = 9
E	16-7 = 9

Le temps moyen de séjour est : $\frac{(3+8+9+9+9)}{5} = \frac{38}{5} = 7{,}6$

Le temps d'attente est calculé soustrayant le temps d'exécution du temps de séjour :

Processus	Temps d'attente
A	3-3 = 0
B	8-6 = 2
C	9-4 = 5
D	9-2 = 7
E	9-1 = 8

Le temps moyen d'attente est : $\frac{(0+2+5+7+8)}{5} = \frac{23}{5} = 4{,}4$

Il y a cinq tâches exécutées dans 16 unités de temps, alors $16/5 = 3{,}2$ **unités de temps par processus**.

2. Le plus court d'abord. Schéma d'exécution :

A	A	A	B	B	B	B	B	B	E	D	D	C	C	C	C
1				5					10					15	

Pour la stratégie **SJF** nous aurons la séquence d'exécution A,B,E,D,C, et le temps de séjour est :

Processus	Temps de séjour
A	3-0 = 3
B	9-1 = 8
E	10-7 = 3
D	12-6 = 6
C	16-4 = 12

$\frac{(3+8+3+6+12)}{5} = \frac{32}{5} = 6{,}4$

1. Bien entendu, on suppose qu'il n'y a aucun blocage.

Processus	Temps d'attente
A	3-3 = 0
B	8-6 = 2
E	3-1 = 2
D	6-2 = 4
C	12-4 = 8

Le temps moyen d'attente est : $\frac{(0+2+2+4+8)}{5} = \frac{16}{5} = 3{,}2$
Il y a cinq tâches exécutées dans 16 unités de temps, alors $16/5 = 3{,}2$ **unités de temps par processus**.

Comment appliquer cette technique aux processus interactifs ?

Chaque processus se comporte comme suit : attend une commande, l'exécute, attend la commande suivante, et ainsi de suite. Alors parmi les processus prêts, le processus élu est celui dont la commande à exécuter est *la plus courte en temps*. Le temps d'exécution de la prochaine commande de chaque processus est estimé en se basant sur le comportement passé du processus : Si T_0 est le temps d'exécution estimé pour la première commande et T_i le temps d'exécution mesuré pour la i^{ieme} commande, les estimations successives sont :

T_0 pour la première ;
$T_0/2 + T_1/2$ pour la seconde ;
$T_0/4 + T_1/4 + T_2/2$ pour la troisième ;
$T_0/8 + T_1/8 + T_2/4 + T_3/2$, ...

Les ordonnanceurs non préemptifs ne sont pas intéressants pour les systèmes multi-utilisateurs car les temps de réponse ne sont pas toujours acceptables.

4.5 Ordonnanceurs préemptifs

Dans un schéma d'**ordonnanceur préemptif** ou avec réquisition, pour s'assurer qu'aucun processus ne s'exécute pendant trop de temps, les ordinateurs ont une horloge électronique qui génère périodiquement une interruption. A chaque interruption d'horloge, le système d'exploitation reprend la main et décide si le processus courant doit poursuivre son exécution ou s'il doit être suspendu pour laisser place à un autre. S'il dé-

cide de suspendre l'exécution au profit d'un autre, il doit d'abord sauvegarder l'état des registres du processeur avant de charger dans les registres les données du processus à lancer. C'est qu'on appelle la *commutation de contexte* ou le *changement de contexte*. Cette sauvegarde est nécessaire pour pouvoir poursuivre ultérieurement l'exécution du processus suspendu.

Le processeur passe donc d'un processus à un autre en exécutant chaque processus pendant quelques dizaines ou centaines de millisecondes. Le temps d'allocation du processeur au processus est appelé **quantum**. Cette commutation entre processus doit être rapide, c'est-à-dire, exiger un temps nettement inférieur au **quantum**.

Le processeur, à un instant donné, n'exécute réellement qu'un seul processus, mais pendant une seconde, le processeur peut exécuter plusieurs processus et donne ainsi l'impression de parallélisme (pseudo-parallélisme).

Problèmes :

- Choix de la valeur du quantum.
- Choix du prochain processus à exécuter dans chacune des situations suivantes :
  1. Le processus en cours se bloque (passe à l'état Attente).
  2. Le processus en cours passe à l'état Prêt (fin du quantum...).
  3. Un processus passe de l'état Attente à l'état Prêt (fin d'une E/S).
  4. Le processus en cours se termine.

4.5.1 Ordonnancement du plus petit temps de séjour

L'**ordonnancement du plus petit temps de séjour** ou ***Shortest Remaining Time*** est la version préemptive de l'algorithme SJF. Un processus arrive dans la file de processus, l'ordonnanceur compare la valeur espérée pour ce processus contre la valeur du processus actuellement en exécution. Si le temps du nouveau processus est plus petit, il rentre en exécution immédiatement.

4.5.2 Ordonnancement circulaire

L'**algorithme du tourniquet**, **circulaire** ou *round robin* montré sur la figure 4.3 est un algorithme ancien, simple, fiable et très utilisé. Il mémorise

dans une file du type FIFO *(First In First Out)* la liste des processus prêts, c'est-à-dire en attente d'exécution.

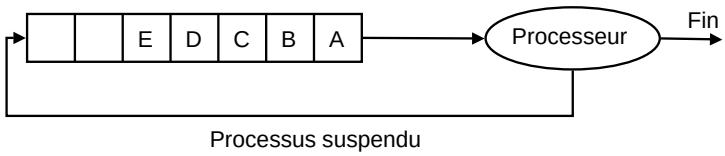


FIG. 4.3 – *Ordonnancement circulaire.*

Choix du processus à exécuter

Il alloue le processeur au processus en tête de file, pendant un quantum de temps. Si le processus se bloque ou se termine avant la fin de son quantum, le processeur est immédiatement alloué à un autre processus (celui en tête de file). Si le processus ne se termine pas au bout de son quantum, son exécution est suspendue. Le processeur est alloué à un autre processus (celui en tête de file). Le processus suspendu est inséré en queue de file. Les processus qui arrivent ou qui passent de l'état bloqué à l'état prêt sont insérés en queue de file.

Choix de la valeur du quantum

Un quantum trop petit provoque trop de commutations de processus et abaisse l'efficacité du processeur. Un quantum trop élevé augmente le temps de réponse des courtes commandes en mode interactif. Un quantum entre 20 et 50 ms est souvent un compromis raisonnable [2].

▶ **Exemple 2.** Soient deux processus A et B prêts tels que A est arrivé en premier suivi de B, 2 unités de temps après. Les temps de UCT nécessaires pour l'exécution des processus A et B sont respectivement 15 et 4 unités de temps. Le temps de commutation est supposé nul. Calculer le temps de séjour de chaque processus A et B, le temps moyen de séjour, le temps d'attente, le temps moyen d'attente, et le nombre de changements de contexte pour :

- SRT
- Round robin (quantum = 10 unités de temps)

2. Le quantum était de 1 seconde dans les premières versions d'Unix.

– Round robin (quantum = 3 unités de temps)

Solution SRT :

A	A	B	B	B	B	A	A	A	A	A	A	A	A	A	A	A	A	A
1				5					10					15				

Processus	Temps de séjour
A	19-0 = 19
B	6-2 = 4

Temps moyen de séjour = $\frac{19+4}{2} = 11{,}5$

Processus	Temps d'attente
A	19-15 = 4
B	4-4 = 0

Le temps moyen d'attente = $\frac{4+0}{2} = 2$

Il y a 3 changements de contexte.

Round robin (quantum = 10) :

A	A	A	A	A	A	A	A	A	A	B	B	B	B	A	A	A	A	A
1				5					10					15				

Processus	Temps de séjour
A	19-0 = 19
B	14-2 = 12

Temps moyen de séjour = $\frac{19+12}{2} = 15{,}5$

Processus	Temps d'attente
A	19-15 = 4
B	12-4 = 8

Le temps moyen d'attente = $\frac{4+8}{2} = 6$

Il y a 3 changements de contexte.

Round robin (quantum = 3) :

A	A	A	B	B	B	A	A	A	B	A	A	A	A	A	A	A	A	A
1				5					10					15				

Processus	Temps de séjour
A	19-0 = 19
B	10-2 = 8

Temps moyen de séjour = $\frac{19+8}{2} = 13{,}5$

Processus	Temps d'attente
A	19-15 = 4
B	8-4 = 4

Le temps moyen d'attente = $\frac{4+4}{2} = 4$
Il y a 5 changements de contexte.

Dans le trois solutions (SRT, RR Q_t=10 et RR Q_t=3), il y a 2 tâches exécutées dans 19 unités de temps, alors $19/2 = 9{,}5$ **unités de temps par processus**.

▶ **Exemple 3.** Soient trois processus **A**, **B** et **C**. Le processus **A** arrive en premier suivi de **B** (3 unités de temps après), puis **C** (4 unités de temps après **B**). On suppose que l'exécution du processus **A** nécessite 8 unités de temps UCT. L'exécution du processus **B** nécessite dans l'ordre 5 unités de temps UCT, 2 unités de temps d'E/S et 3 unités de temps UCT. L'exécution du processus **C** nécessite 4 unités de temps UCT. Le quantum est de 3 unités de temps. Le temps de commutation est de 1 unité de temps.

1. Montrer comment les trois processus vont utiliser le processeur, dans le cas d'un ordonnancement circulaire?
2. Calculer le temps de séjour pour chacun des trois processus et le temps de séjour moyen des trois processus.
3. Que se passe-t-il, si on augmente la valeur du quantum? (Q_t = 5).
4. Que se passe-t-il, si on diminue la valeur du quantum? (Q_t = 2).

Solution. Voir la table 4.2 et la figure 4.4.

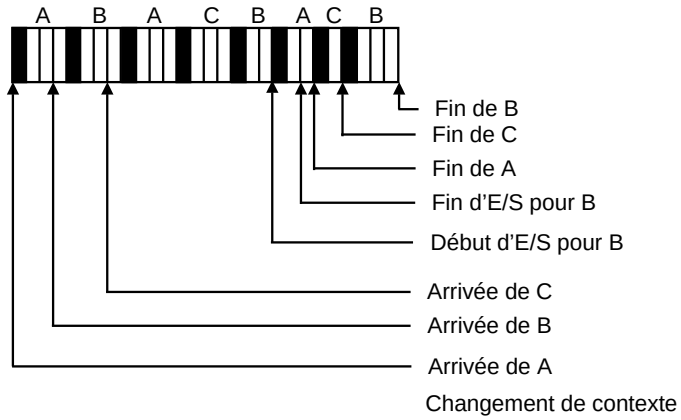


FIG. 4.4 – *Le quantum $Q_t = 3$. Le temps de séjour pour A est (22 - 0) unités de temps. Le temps de séjour pour B est (28 - 3) unités de temps Le temps de séjour pour C est (24 - 7) unités. Le temps de séjour moyen est* $(22+25+17)/3 = 21{,}33$.

Q_t	A	B	C	Commutations	Temps moyen
2	26	27	14	10	22,3
3	22	25	17	8	21,33
5	16	22	14	5	17,33

TAB. 4.2 – *Solution des trois processus A, B, C.*

4.5.3 Ordonnancement avec priorité

L'algorithme *round robin* permet une répartition équitable du processeur. Cependant il n'est pas intéressant si certains processus sont plus importants ou urgents que d'autres. L'**ordonnanceur à priorité** attribue à chaque processus une priorité. Le choix du processus à élire dépend des priorités des processus prêts.

Les processus de même priorité sont regroupés dans une file du type FIFO. Il y a autant de files qu'il y a de niveaux de priorité. L'ordonnanceur choisit le processus le plus prioritaire qui se trouve en tête de file. En général, les processus de même priorité sont ordonnancés selon l'algorithme du tourniquet.

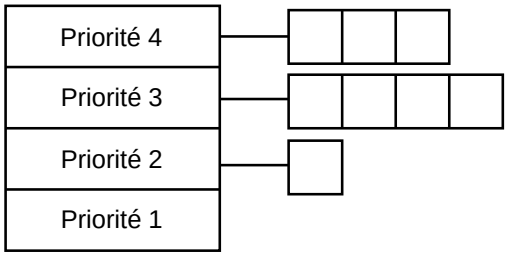


FIG. 4.5 – *Ordonnancement avec priorité.*

Attribution et évolution des priorités

Pour empêcher les processus de priorité élevée de s'exécuter indéfiniment, l'ordonnanceur diminue régulièrement la priorité du processus en cours d'exécution.

La priorité du processus en cours est comparée régulièrement à celle du processus prêt le plus prioritaire (en tête de file). Lorsqu'elle devient inférieure, la commutation a lieu. Dans ce cas, le processus suspendu est inséré en queue de file correspondant à sa nouvelle priorité. L'attribution

et l'évolution des priorités dépendent des objectifs fixés et de beaucoup de paramètres.

▶ **Exemple 4.** Les processus qui font beaucoup d'E/S (qui sont souvent en attente) doivent acquérir le processeur dès qu'ils le demandent, afin de leur permettre de lancer leur requête suivante d'E/S. Lorsqu'un processus passe de l'état élu à l'état bloqué, sa priorité est recalculée. Sa nouvelle valeur est le rapport : quantum / temps réellement utilisé par le processus.

Les processus qui ont le plus grand rapport sont les plus prioritaires :

- Si le quantum = 100 ms et le temps utilisé = 2 ms, la nouvelle priorité est 50
- Si le quantum = 100 ms et le temps utilisé = 50 ms, la nouvelle priorité est 2

4.5.4 Files multiples (quantum variable)

Pour éviter qu'il y ait beaucoup de commutations pour les processus consommateurs de temps UCT, il est préférable d'allouer un plus grand quantum à ces processus. Lorsqu'un processus passe à l'état élu : Pour la première fois, le processeur lui est alloué pendant un quantum. Pour la seconde fois, le processeur lui est alloué pendant 2 quantum... Pour la n^{ieme} fois, le processeur lui est alloué pendant $2^{(n-1)}$ quantum.

Chaque processus a une priorité. Cette dernière dépend du nombre de quantum qui lui sera alloué lors de sa prochaine activation. Les processus dont le nombre de quantum est le plus petit sont les plus prioritaires. Les processus prêts sont répartis selon leur priorité dans des files (FIFO).

D'abord on fait l'élection du processus le plus prioritaire qui se trouve en tête de file. Lorsque l'exécution d'un processus est suspendue pour la n-ième fois, sa priorité est recalculée (2^n) puis il est inséré à la queue de la file appropriée.

▶ **Exemple 5.** Considérons un processus qui doit effectuer des calculs pendant 100 quantum. Ce processus obtient successivement 1, 2, 4, 8, 16, 32 et 64 quantum. Mais il utilisera 37 des 64 derniers quantum. Le nombre de changements de contexte passe de 100 (cas du tourniquet) à 7. Le processeur passe moins de temps à commuter et a donc un meilleur temps de réponse. Surtout dans le cas où la commutation nécessite des transferts sur disque.

Un processus qui descend de plus en plus dans les files à priorité s'exécute de moins en moins fréquemment et favorise ainsi les processus courts en mode interactif. Pour ne pas défavoriser un processus qui s'était exécuté pendant assez longtemps avant de devenir interactif, on peut lui attribuer la plus haute priorité dès qu'un retour chariot est tapé sur le terminal associé au processus (les premières versions d'Unix utilisaient cette solution).

4.6 Ordonnancement à deux niveaux

Lors d'une commutation, si le processus élu n'est pas en mémoire, il faut le charger en mémoire. Le temps de commutation est deux à trois fois plus élevé que celui des processus qui se trouvent en mémoire centrale. Pour éviter ces va-et-vient entre le disque et la mémoire lors d'une commutation, l'**ordonnanceur à deux niveaux** déplace les processus entre le disque et la mémoire (haut niveau) et choisit le processus à exécuter parmi ceux qui sont en mémoire (bas niveau). Périodiquement, l'ordonnanceur de haut niveau retire de la mémoire les processus qui y sont restés assez longtemps et les remplace par des processus qui sont restés sur le disque pendant trop de temps. Pour élire un processus, l'ordonnanceur de bas niveau se restreint au processus en mémoire.

4.7 Ordonnancement des threads

L'idée derrière les threads est de permettre l'exécution « parallèle » de plusieurs tâches d'un même processus. L'ordonnancement, dans ce cas, peut se faire au niveau des processus ou au niveau des processus légers (threads).

Dans le premier cas (voir figure 4.6), l'ordonnanceur choisit un processus parmi ceux qui sont prêts. Le processeur exécute alors le thread ordonnanceur du processus élu qui va se charger d'ordonnancer les threads du processus élu. Le thread ordonnanceur utilise une des méthodes d'ordonnancement précédentes (tourniquet, à priorité). La seule restriction est l'absence d'une horloge pour suspendre l'exécution d'un thread qui monopolise le processeur. Dans le second cas (figure 4.7), l'ordonnanceur choisit un processus léger parmi tous ceux qui sont prêts (il fait une abstraction des processus). Ceci est le cas de Linux.

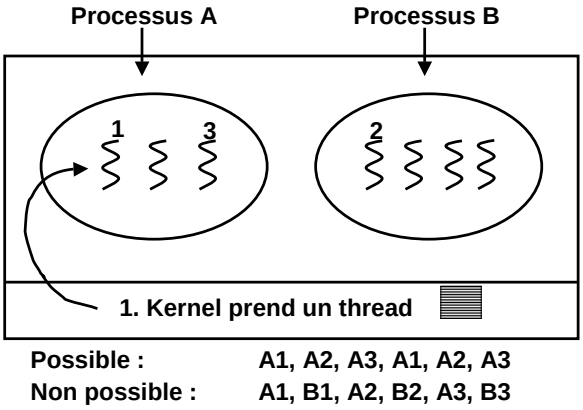


FIG. 4.6 – *Ordonnancement des threads.*

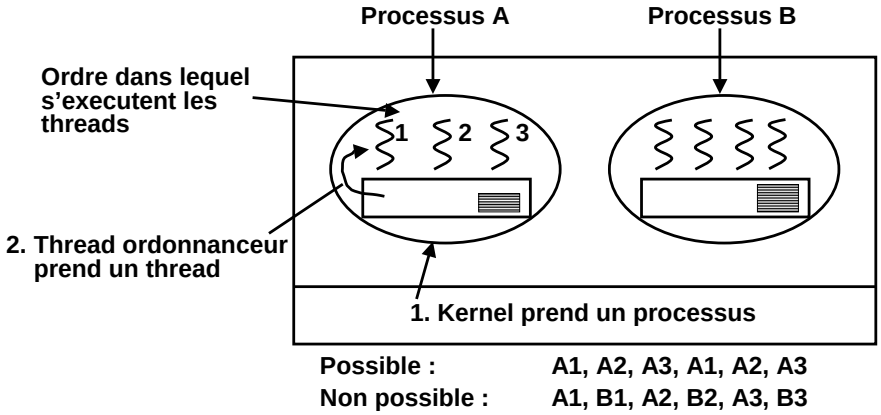


FIG. 4.7 – *Ordonnancement des threads dans Linux.*

4.8 Cas d'étude

4.8.1 MS-DOS

MS-DOS est un système mono-utilisateur, et il n'est non plus un système multiprogrammé. Il peut y avoir, à un moment donné, plusieurs processus en mémoire mais un seul est à l'état prêt, tous les autres étant en attente de la fin d'un fils. L'ordonnanceur des processus du système MS-DOS est très simple : il exécute un processus jusqu'à ce qu'il se termine ou crée un fils. Le processus créateur est suspendu jusqu'à ce que le processus créé se termine.

4.8.2 Unix

C'est un ordonnanceur à deux niveaux. L'ordonnanceur de bas niveau se charge d'élire un processus parmi ceux qui résident en mémoire. L'ordonnanceur de haut niveau se charge des transferts de processus entre la mémoire centrale et le disque. L'ordonnanceur de bas niveau utilise plusieurs files, une priorité est associée à chaque file (plusieurs niveaux de priorité). Les processus prêts qui sont en mémoire sont répartis dans les files selon leur priorité. Les priorités des processus s'exécutant en mode utilisateur sont positives ou nulles, alors que celles des processus s'exécutant en mode noyau sont négatives. Les priorités négatives sont les plus élevées, comme illustré à la figure 4.8.

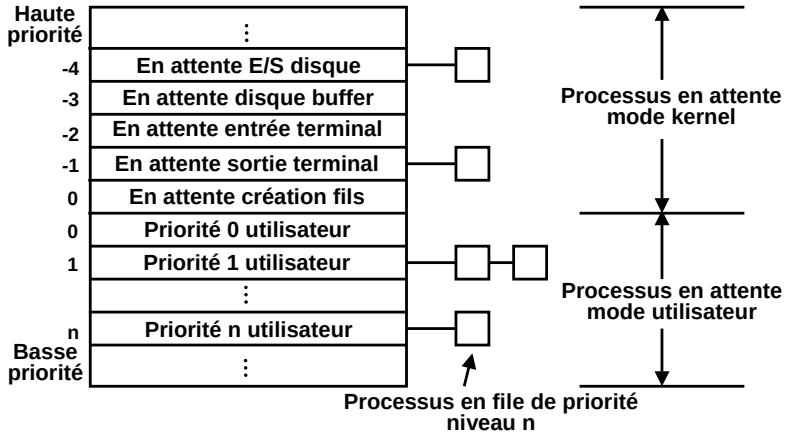


FIG. 4.8 – *Ordonnancement sous Unix.*

L'ordonnanceur de bas niveau choisit le processus le plus prioritaire qui se trouve en tête de file. Le processus élu est alors exécuté pendant au maximum un quantum (100 ms). S'il ne se termine pas ou ne se bloque pas au bout de ce quantum, il est suspendu. Le processus suspendu est inséré en queue de sa file. La priorité initiale d'un processus utilisateur est en général égale à 0. La commande `nice` permet d'attribuer une priorité plus basse à un processus (> 0). Les priorités des processus prêts en mémoire sont recalculées périodiquement toutes les secondes. La priorité d'un processus est recalculée, dans certaines versions, en ajoutant à sa valeur de base (en général 0 pour les processus utilisateurs), la moitié du temps UCT utilisé (temps en nombre de pulsations d'horloge) et la valeur de `nice`.

Lorsqu'un processus passe de l'état bloqué à l'état prêt, il se voit attribuer une forte priorité (négative). Après le calcul des priorités, les proces-

sus sont déplacés vers les files d'attente correspondant à leurs nouvelles priorités.

4.8.3 Linux

Le changement de contexte c'est le mécanisme qui permet le traitement multi-tâche. En Linux il existe la fonction `switch_to` dans le fichier `linux/kernel/sched.c`. Cette fonction sauvegarde dans la table PCB le descripteur du processus en exécution. Elle récupère aussi le contexte du nouveau processus admis. Cette fonction est appelée par `schedule()` Dans plusieurs systèmes d'exploitation le noyau est un processus, c'est pourquoi la façon d'y accéder est par un changement de contexte. Le noyau de Linux n'est pas un processus. Le noyau protège ainsi la consistance de ses structures de données en bloquant de façon approprié, en interdisant les changements de contexte —- sous n'importe quelle condition — et en contrôlant toutes les interruptions..

L'ordonnanceur de Linux complète celui d'Unix, en classant les processus légers en trois catégories et en ajoutant deux autres techniques. Il distingue ainsi trois classes de processus légers :

1. **Tourniquet temps réel** ou `SCHED_RR` : les processus légers de cette classe sont gérés selon l'algorithme du tourniquet avec un quantum variable.
2. **FIFO temps réel** ou `SCHED_FIFO` : ils sont les plus prioritaires.
3. **Temps partagé** ou `SCHED_OTHER` : les processus légers de cette classe sont ordonnancés selon la méthode du système Unix traditionnel. Ces processus légers s'exécutent uniquement si aucun processus légers des classes `1` et `2` n'est prêt.

Si un processus léger de la classe FIFO temps réel est à l'état élu, son exécution est suspendue uniquement dans l'une des situations suivantes :

1. Un processus léger plus prioritaire de la même classe devient prêt. Ce dernier devient élu.
2. Il se bloque ou il se termine.
3. Il cède volontairement le processeur avec `sched_yield()` sans se bloquer. Le processus sera alors déplacé à la fin de la liste des processus prêts de sa priorité, et un autre processus sera exécuté. Si le processus est le seul avec une priorité élevée, il continuera son exécution.

Durant l'exécution d'un processus léger de la deuxième ou de la troisième classe, son quantum est décrémenté de `1` à chaque top d'horloge.

S'il est toujours à l'état élu lorsque son quantum passe à `0`, il est alors suspendu. Il peut être également suspendu, si un processus plus prioritaire devient prêt.

▶ **Exemple 6.** Le programme 4.1 `scheduler.c` permet de connaître la politique d'ordonnancement pour un processus. Il utilise la fonction Posix `sched_getscheduler()`, définie en :

```
#include <sched.h>
int sched_getscheduler(int pid);
```

où `pid` est l'identificateur d'un processus courant.

Listing 4.1: *scheduler.c*

```
#include <sched.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
  int ordonnancement;
  int pid;
  if (argc != 2)
    {
      fprintf (stderr, "Syntaxe %s Pid \n", argv[0]);
       exit (1);
    }
  pid = atoi(argv[1]);
  ordonnancement = sched_getscheduler(pid);
  fprintf (stdout, "ordonnancement = %d \n", ordonnancement);
  switch(ordonnancement)
  {
    case SCHED_RR : fprintf (stdout, "Round robin \n"); break;
    case SCHED_FIFO : fprintf(stdout, "FIFO \n"); break;
    case SCHED_OTHER : fprintf(stdout, "Autre \n"); break;
    default          : fprintf (stdout, "??? \n" );    break;
  }
  return 0;
}
```

Suggestions de lecture

Référence : Silverwschatz A., Galvin P., Gagné G., *Principles appliqués des systèmes d'exploitation avec Java*, Vuibert, Paris, France, 2002.

Chapître 6 : Ordonnancement d'exécution.

Chapître 20 : Le système Unix.

4.9 Exercices

1. Citez quatre événements qui provoquent l'interruption de l'exécution d'un processus en cours, dans le système Unix.
2. Quel est le rôle de l'ordonnanceur?
3. Décrire brièvement l'ordonnanceur du système Unix. Favorise-t-il les processus interactifs?
4. Expliquez une raison pour laquelle certains systèmes (tel Unix) assignent des priorités internes plutôt qu'externes aux processus.
5. Identifiez l'algorithme d'ordonnancement qui est le plus utilisé, et spécifiez la raison.
6. Identifiez l'algorithme d'ordonnancement sans réquisition qui fournit la meilleure performance. Pourquoi?
7. Expliquez la différence fondamentale entre un algorithme d'ordonnancement à réquisition et sans réquisition. Lequel fournit la meilleure performance?
8. Quel sera l'effet sur la performance du système d'un temps de quantum trop long et trop petit.
9. Expliquez la différence fondamentale existant entre un ordonnanceur de bas-niveau et un ordonnanceur de haut-niveau. Existe-il des systèmes d'exploitation qui utilisent un seul ordonnanceur? Expliquez aussi.
10. Expliquez quel mécanisme permet à un processus d'effectuer un partage volontaire de l'UCT. Quels sont les avantages et désavantages du partage volontaire?
11. Expliquez comment le système détermine le moment d'effectuer un changement de contexte entre deux processus. Durant un changement de contexte, quels sont les processus (utilisateur ou système) qui sont chargés et déchargés?
12. Considérez la charge de processus montré sur la table 4.3 suivante. Pour chacun des algorithmes d'ordonnancement spécifiés ci-bas, dessinez un diagramme montrant l'état d'occupation de l'UCT, des périphériques, et des files d'attente. Aussi, calculez les temps de virement et d'attente. Supposez un temps de changement de contexte instantané (infiniment petit). Finalement, comparez les performances des divers algorithmes pour cette charge de processus.

    a) L'algorithme du premier-arrivé-premier-servi (PAPS).

    b) L'algorithme du tourniquet avec un quantum de 5 unités de temps.

Processus	Temps d'arrivée	Temps de service	Durée des E/S	Périodes des E/S
P0	0	9	2	5,9
P1	2	6	0	-
P2	5	5	1	4

TAB. 4.3 – *Charge de processus.*

c) L'algorithme du tourniquet avec un quantum de 5 unités de temps et des niveaux de priorité externe de 3, 1, 2 respectivement pour P0, P1, P2.

d) L'algorithme du tourniquet avec un quantum de 5 unités de temps et deux UCTs fonctionnant en parallèle.

e) L'algorithme du tourniquet avec un quantum de 5 unités de temps et des niveaux de priorité interne initiaux de 20 respectivement pour P0, P1, P2. Au moment d'un changement de contexte, la priorité du processus débarquant diminue de 1 unité. Aussi, un processus qui est bloqué dans une file d'attente des E/S perd 2 unités de priorité. Si le processus n'attend pas pour effectuer son E/S alors il ne perd que 1 unité de priorité. Lorsqu'un processus dans la file d'attente de l'UCT a plus haute priorité qu'un processus en exécution, l'UCT est alors préempté avant la fin du quantum.

5
Communication interprocessus

Les processus coopèrent souvent pour traiter un même problème. Ces processus s'exécutent en parallèle sur un même ordinateur (monoprocesseur ou multiprocesseurs) ou bien sur des ordinateurs différents. Ils doivent alors s'échanger des informations (**communication interprocessus**). Il existe plusieurs moyens de communication interprocessus. Nous pouvons citer, parmi d'autres, les variables communes, les fichiers communs, les signaux, les messages et les tubes de communication.

5.1 Variables et fichiers communs

Les processus partagent un ensemble de données communes qui sont soit en mémoire (variables ou segments de données) ou sur disque (fichiers). Chaque processus peut accéder en lecture ou en écriture à cet ensemble de données appelé **espace de données commun**. Des problèmes peuvent se poser lorsque plusieurs processus essayent d'accéder en même temps à un espace commun. Si deux processus ou plus lisent ou écrivent des données partagées, le résultat dépend de l'ordonnancement des processus. C'est ce qu'on appelle des *accès concurrents*.

Supposons deux processus qui partagent une variable `v`. Le premier processus incrémente de `1` la valeur de `v` alors que l'autre la décrémente de `1`. La valeur initiale de `v=1`.

```
P1 : v=++;
P2 : v=--;
```

Les instructions correspondant en assembleur :

```
P1 :   movl $v, %eax       # load v
```

```
        movw (%eax), %bx
        incw %bx            # bx++;
        movw %bx, (%eax)    # store v

P2 :    movl $v, %eax       # load v
        movw (%eax), %bx
        decw %bx            # bx--;
        movw %bx, (%eax)    # store v
```

Supposons que les processus `P1` et `P2` s'exécutent en temps partagé.

- Le processus `P1` est en cours d'exécution.
- Il est suspendu juste après l'exécution de l'instruction `incw %bx`.
- Le processus `P2` est élu et exécute les instructions `load v` et `decw %bx` et `store v`. Le processus P2 se termine avec `v = 0`.
- Le processus `P1` est ensuite élu et exécute l'instruction `store v` et donc `v=2`.

Pour éviter de tels problèmes, il est nécessaire de contrôler les accès concurrents à des objets communs. Nous verrons dans ce chapitre les solutions qui permettent de résoudre les conflits d'accès à des objets communs. Notamment nous allons traiter les différents moyens de communication interprocessus offerts par le système Unix / Linux.

5.2 Les signaux

Les **interruptions logicielles** ou **signaux** sont le mécanisme qui est utilisé par le système d'exploitation pour aviser les processus utilisateurs de l'occurrence d'un événement important. De nombreuses erreurs détectées par le matériel comme l'exécution d'une instruction non autorisée (comme la division par 0, par exemple) ou l'emploi d'une adresse non valide, sont converties en signaux qui sont envoyés au processus fautif. Ce mécanisme permet à un processus de réagir à cet événement sans être obligé d'en tester en permanence l'arrivée.

Les processus peuvent indiquer au système ce qui doit se passer à la réception d'un signal. On peut ainsi ignorer le signal, ou bien le prendre en compte, ou encore le laisser tuer le processus. Certains signaux ne peuvent être ni ignorés, ni capturés. Si un processus choisit de prendre en compte les signaux qu'il reçoit, il doit alors spécifier la procédure de gestion de signal. Quand un signal arrive, la procédure associée est exécutée. A la fin de

l'exécution de la procédure, le processus s'exécute à partir de l'instruction qui suit celle durant laquelle l'interruption a eu lieu.

En résumé, les signaux sont utilisés pour établir une communication minimale entre processus, une communication avec le monde extérieur et faire la gestion des erreurs. Ainsi, un processus peut :

- **Ignorer le signal.**
- **Appeler une routine de traitement fourni par le noyau.** Cette procédure provoque normalement la mort du processus. Dans certains cas, elle provoque la création d'un fichier `core`, qui contient le contexte du processus avant de recevoir le signal. Ces fichiers peuvent être examinés à l'aide d'un débogueur.
- **Appeler une procédure spécifique créée par le programmeur.** Tous les signaux ne permettent pas ce type d'action.

Tous les signaux ont une routine de service, ou une action par défaut. Cette action peut être du type :

- A : terminaison du processus
- B : ignorer le signal
- C : Créer un fichier `core`
- D : Stopper le processus
- E : La procédure de service ne peut pas être modifiée
- F : Le signal ne peut pas être ignoré

Chaque signal est identifié au moyen d'un numéro entier positif. La table 5.1 décrit les signaux les plus importants. Dans l'**Annexe C** le lecteur trouvera des tables plus complètes des signaux sur Linux et Unix.

Signal	Num	Type	Description
SIGKILL	9	AEF	Termine l'exécution du processus
SIGSTOP	19	DEF	Stoppe le processus
SIGCONT	18		Continue l'exécution du processus
SIGCHILD	17	B	Un processus fils a fini
SIGUSR1	10	A	Signal d'utilisateur
SIGUSR2	10	A	Signal d'utilisateur

TAB. 5.1 – *Quelques signaux Unix.*

5.2.1 Services de signaux

Dans le cas du système Unix/Linux, un processus utilisateur peut également envoyer un signal à un autre processus. Les deux processus appar-

tiennent au même propriétaire, ou bien le processus émetteur du signal est le super-utilisateur. Les services Posix de gestion des signaux utilisent la bibliothèque `<signal.h>`.

`kill()`

L'envoi d'un signal peut se faire avec l'appel système `kill()` :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid t pid, int signal);
```

`kill()` envoi le signal numéro `signal` à un processus identifié par son `pid`. En cas d'erreur il retourne -1, et 0 autrement. `pid` peut prendre comme valeurs :

- Si `pid` > 0 : processus `pid`.
- Si `pid` = 0 : groupe de l'émetteur.
- Si `pid` =-1 : tous les processus (seulement root peut le faire).
- Si `pid` < 0 : au groupe `gid` = |`pid`|

▶ **Exemple 1.** Le programme 5.1 `signaux-0.c` montre l'utilisation du masquage des signaux.

Listing 5.1: *signaux-0.c*

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sighandler(int signum);

int  main(void)
{
 char buffer[256];
10

 if(signal(SIGTERM,&sighandler) == SIG_ERR )
 {
     printf("Ne peut pas manipuler le signal\n");
     exit (1);
 }
 while(1)
 {
```

```
    fgets(buffer,sizeof(buffer), stdin);
    printf("Input: %s",buffer);
  }
 return 0;
}

void sighandler(int signum)
{
   printf("Masquage du signal SIGTERM\n");
}
```

Nous allons montrer l'exécution de `signaux-0.c` dans deux terminaux, afin de montrer le masquage de signaux et d'essayer de tuer le processus. D'abord, exécution sur un terminal :

```
leibnitz> gcc -o signal-0 signal-0.c
leibnitz> signal-0
HOLA
Input: HOLA
KJHKHBNFYNBknbjhhgdt6565
Input: KJHKHBNFYNBknbjhhgdt6565
ls -l
Input: ls -l
who
Input: who
...
```

Dans un autre terminal (`pts/1`), on peut essayer de tuer le processus 22096 :

```
leibnitz> ps -u jmtorres
 PID TTY          TIME CMD
20652 pts/0    00:00:00 tcsh
21964 pts/0    00:00:00 a.out
21966 pts/1    00:00:00 tcsh
22096 pts/0    00:00:00 signal-0
22098 pts/1    00:00:00 ps
leibnitz> kill 22096
leibnitz> kill 22096
leibnitz> kill -9 22096
leibnitz>
```

Et dans le premier terminal (`pts/0`) nous continuerons à voir des messages, jusqu'à ce que le signal non ignorable 9 arrive :

```
...
Who am i
```

```
Input: who am i
kill
Input: kill
Masquage du signal SIGTERM
Masquage du signal SIGTERM
Killed
leibnitz>
```

Il faut dire que `Ctrl-C` ou bien `Ctrl-Z` auraient aussi tué le processus.

▶ **Exemple 2.** Le programme 5.2 `signaux-1.cpp` illustre l'utilisation des signaux `SIGUSR1` et `SIGUSR2` :

Listing 5.2: *signaux-1.cpp*

```
#include <signal.h>
#include <iostream.h>
#include <unistd.h>
#include <wait.h>

static void action(int sig);
void main()
{
 int i, pid, etat;
 // Spécification de l'action du signal
 if(signal(SIGUSR1, action) == SIG_ERR)
    cout << "Erreur de traitement du code de l'action" << endl;
 if(signal(SIGUSR2, action) == SIG_ERR)
    cout << "Erreur de traitement du code de l'action" << endl;
 if((pid = fork()) == 0)
 {
        kill(getppid(), SIGUSR1); // Envoyer signal au parent.
        for(;;)
         pause();                  // Mise en attente d'un signal
 }
 else
 {
        kill(pid, SIGUSR2);     // Envoyer un signal à l'enfant
        cout << "Parent : terminaison du fils" << endl;
        kill(pid, SIGTERM);     // Signal de terminaison à l'enfant
        wait(&etat);            // attendre la fin de l'enfant
```

```
        cout « "Parent : fils terminé" « endl;
 }
}
// Fonction définissant le code de
// l'action à l'attrape d'un signal
static void action(int sig)
{
  switch (sig)
  {
     case SIGUSR1 :
          cout « "Parent : signal SIGUSR1 reçu" « endl;
        break;
     case SIGUSR2 :
          cout « "Fils : signal SIGUSR2 reçu" « endl;
          break;
     default : break;
  }
}
```

Une exécution de `signaux-1.cpp` est montrée à continuation :

```
leibnitz> g++ -o signaux-1 signaux-1.cpp
leibnitz> signaux-1
Parent : terminaison du fils
Fils : signal SIGUSR2 reçu
Parent : terminaison du fils
Parent : signal SIGUSR1 reçu
Parent : fils terminé
leibnitz>
```

Cependant, si l'on essai à nouveau une autre exécution, la sortie probablement, sera tout à fait différente :

```
leibnitz> signaux-1
Parent : terminaison du fils
Parent : fils terminé
leibnitz>
```

Pour quoi ce comportement étrange et comment le résoudre ?

`sleep()` et `pause()`

Les appel système `sleep()` et `pause()` peuvent être utiles lorsqu'on travaille avec des signaux.

```
#include <unistd.h>
```

```
int pause(void);
int sleep(unsigned int seconds);
```

Un processus peut se mettre en veille et attendre l'arrivé d'un signal particulier pour exécuter le gestionnaire, ou bien l'émetteur utilise `sleep()` pour se synchroniser avec le récepteur.

`pause()` permet que le processus passe à l'état suspendu. Il va quitter cet état au moment de recevoir une signal. Après avoir reçu le signal, celle-ci recevra un traitement adéquat et le processus continuera s'exécutant à l'instruction suivant à `pause()`. Il retourne toujours la valeur -1. Cet appel fournit une alternativa à l'attente active.

▶ **Exemple 3.** Le programme 5.3 `sigint.c` montre l'utilisation incorrecte du signal `SIGINT` car on fait une attente active `while(num_signal<5);`

Listing 5.3: *sigint.c*

```
#include <stdio.h>
#include <signal.h>

int num_signal = 0;

void traiter_SIGINT()
{
  num_signal++;
  printf("Signal CTRL-C capturé!\n");
}

int main()
{
  if (signal(SIGINT, traiter_SIGINT) == SIG_ERR)
  {
        printf("Erreur dans le gestionnaire\n");
        exit (1);
  }
  while (num_signal < 5); // attente active
  printf("\n%d signaux SIGINT sont arrivés\n", num_signal);
  return 0;
}
```

Par contre, le programme 5.4 `sigint-2.c` utilise `pause()` pour éviter ce gaspillage de CPU :

Listing 5.4: *sigint-2.c*

```
#include <stdio.h>
#include <signal.h>

int num_signal = 0;

void traiter_SIGINT()
{
  num_signal++;
  printf("Signal CTRL-C capturé!\n");
}

int main()
{
  if (signal(SIGINT, traiter_SIGINT) == SIG_ERR)
 {
        printf("Erreur dans le gestionnaire\n");
        exit (1);
  }
  while (num_signal < 5) pause(); // attente NON active
  printf("\n%d signaux SIGINT sont arrivés\n", num_signal);
  return 0;
}
```

▶ **Exemple 4.** Le programme 5.5 `signaux-2.c` montre l'utilisation des signaux ignorables `SIGINT` et `SIGQUIT`, ainsi que le signal non ignorable `SIGKILL`:

Listing 5.5: *signaux-2.c*

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

// Déclaration des fonctions à associer aux signaux
void trait_sigint ()
{
  printf("bien reçu SIGINT, mais je vais l'ignorer\n");
}

```

```
void trait_sigquit ()
{
   printf("bien reçu SIGQUIT, mais je vais l'ignorer\n");
}

int main()
{
   int pid;

   pid = fork ();
   if(pid==0) // processus fils
   {     // Associer au signal SIGINT le traitement trait_sigint
      signal(SIGINT, trait_sigint );
        // Associer au signal SIGQUIT le traitement trait_sigquit
      signal(SIGQUIT, trait_sigquit);
      printf("je suis toujours en vie\n");
      sleep(20);
      printf("premier réveil du fils \n");
      sleep(120);
      printf("second réveil du fils \n");
      sleep(500);
      printf("troisième réveil du fils \n");
   }
   else
   {
      sleep(1);
      // émission du signal SIGINT (interrupt)
      kill(pid, SIGINT);
      sleep(2);
      // emission du signal SIGQUIT (quit)
      kill(pid, SIGQUIT);
      sleep(5);
      // émission du signal SIGKILL (kill)
      // ce signal ne peut être ni capturé
      // ni ignoré par le destinataire
      kill(pid,SIGKILL);
   }
   return 0;
}
```

L'exécution de `signaux-2.c` :

```
leibnitz> gcc -o signaux-2 signaux-2.c
leibnitz> signaux-2
 je suis toujours en vie
 bien reçu SIGINT, mais je vais l'ignorer
premier réveil du fils
bien reçu SIGQUIT, mais je vais l'ignorer
second réveil du fils
leibnitz>
```

▶ **Exemple 5.** Le programme 5.6 `ping.c` effectue un ping-pong des signaux entre un processus et son fils. Le signal `SIGUSR1` est envoyé par le fils au père et `SIGUSR2` est envoyé du père au fils. Le premier signal est envoyé par le père.

Listing 5.6: *ping.c*

```
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void handlerSignal();
void sendSignal(int pid, int sig);

int Proc1, cptr, limite;
main(int argc, char **argv)
{
    if (argv[1]==0)
        {
          printf("format : ping n\n");
          exit (1);
        }
    limite=atoi(argv[1]);
    cptr=0;
    Proc1=fork();
    switch (Proc1)
        {
        case 0:
```

```
                signal(SIGUSR2, handlerSignal);
                printf("Fils => signal envoyé, PID=%d\n", getpid());
                pause();
                while(cptr<limite-1)
                {
                        cptr++;
                        sendSignal(getppid(), SIGUSR1);
                        pause();
                }
                cptr++;
                sendSignal(getppid(), SIGUSR1);
                exit (0);
                break;
        default :
                signal(SIGUSR1, handlerSignal);
                printf("Père => signal envoyé, PID=%d\n", getpid());
                cptr++;
                sendSignal(Proc1, SIGUSR2);
                pause();
                while(cptr<limite)
                {
                        cptr++;
                        sendSignal(Proc1, SIGUSR2);
                        pause();
                }
                break;
        }
}

void handlerSignal(int sig)
{
    printf("\t\t[%d] gestionnaire %d => signal capté\n", getpid(), sig);
    fflush(stdout);
}

void sendSignal(int pid, int sig)
{
 sleep(1);
 if ( kill(pid, sig)==-1)
 {
```

```
        printf("ERREUR kill PID=%d\n", pid);
        fflush(stdout);
        exit (1);
 }
 printf("#%d [%d] signal %d envoyé à %d\n", cptr, getpid(), sig, pid);
}
```

Exemple d'exécution de `ping.c` avec l'envoi de deux messages :

```
leibnitz> gcc -o ping ping.c
leibnitz> ping 2
Père => signal envoyé, PID=4825
Fils => signal envoyé, PID=4826
#1 [4825] signal 12 envoyé à 4826
                [4826] gestionnaire 12 => signal capté
#1 [4826] signal 10 envoyé à 4825
                [4825] gestionnaire 10 => signal capté
#2 [4825] signal 12 envoyé à 4826
                [4826] gestionnaire 12 => signal capté
#2 [4826] signal 10 envoyé à 4825
                [4825] gestionnaire 10 => signal capté
leibnitz>
```

5.3 Les messages

Un autre mécanisme de communication entre processus est l'**échange de messages**. Chaque message véhicule des données. Un processus peut envoyer un message à un autre processus se trouvant sur la même machine ou sur des machines différentes. Unix offre plusieurs mécanismes de communication pour l'envoie de messages : les **tubes de communication** sans nom, nommés et les **sockets**.

5.3.1 Tubes de communication

Les **tubes de communication** ou *pipes* permettent à deux ou plusieurs processus d'échanger des informations. On distingue deux types de tubes : les tubes sans nom (*unamed pipe*) et les tubes nommés (*named pipe*).

5.3.2 Tubes sans nom

Les **tubes sans nom** sont des liaisons unidirectionnelles de communication. La taille maximale d'un tube sans nom varie d'une version à une

autre d'Unix, mais elle est approximativement égale à 5Ko. Les tubes sans nom peuvent être créés par le shell ou par l'appel système `pipe()`.

Les tubes sans nom du shell sont créés par l'opérateur binaire « | » qui dirige la sortie standard d'un processus vers l'entrée standard d'un autre processus. Les tubes de communication du shell sont supportés par toutes les versions d'Unix.

▶ **Exemple 6.** La commande shell suivante crée deux processus qui s'exécutent en parallèle et qui sont reliés par un tube de communication pipe (figure 5.1). Elle détermine le nombre d'utilisateurs connectés au système en appelant `who`, puis en comptant les lignes avec `wc` :

```
leibnitz> who | wc -l
```

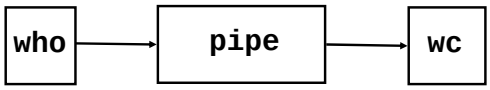


FIG. 5.1 – *Processus en paralèlle reliés par un tube de communication.*

Le premier processus réalise la commande `who`. Le second processus exécute la commande `wc -l`. Les résultats récupérés sur la sortie standard du premier processus sont dirigés vers l'entrée standard du deuxième processus via le tube de communication qui les relie.

Le processus réalisant la commande `who`, dépose une ligne d'information par utilisateur du système sur le tube d'information. Le processus réalisant la commande `wc -l`, récupère ces lignes d'information pour en calculer le nombre total. Le résultat est affiché à l'écran. Les deux processus s'exécutent en parallèle, les sorties du premier processus sont stockées sur le tube de communication. Lorsque le tube devient plein, le premier processus est suspendu jusqu'à ce qu'il y ait libération de l'espace nécessaire pour stocker une ligne d'information. De façon similaire, lorsque le tube devient vide, le second processus est suspendu jusqu'à ce qu'il y ait au moins une ligne d'information sur le tube.

Création d'un tube sans nom

Un tube de communication sans nom est créé par l'appel système :

```
int pipe(int fildes[2]);
```

L'appel système `pipe()` retourne après création du tube deux descripteurs de fichiers : le descripteur pour les lectures du tube et le descripteur pour les écritures sur le tube. L'accès au tube se fait via les descripteurs. Seul le processus créateur du tube et ses descendants (ses fils) peuvent accéder au tube. Si le système ne peut pas créer de tube pour manque d'espace, l'appel système `pipe()` retourne la valeur `-1`, sinon il retourne la valeur `0`.

▶ **Exemple 7.** Les déclarations suivantes créent le tube montré sur la figure 5.2 :

```
int fd[2];
pipe(fd);
```

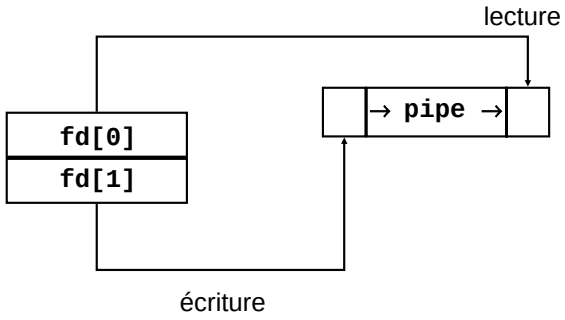


FIG. 5.2 – *Pipe de Lecture/Écriture.*

Les tubes sans nom sont, en général, utilisés pour la communication entre un processus père et ses processus fils, avec un d'entre eux qui écrit sur le tube, appelé processus écrivain, et l'autre qui lit à partir du tube, appelé processus lecteur (voir figure 5.3). La séquence d'événements pour une telle communication est comme suit :

- Le processus père crée un tube de communication sans nom en utilisant l'appel système `pipe()`
- Le processus père crée un ou plusieurs fils en utilisant l'appel système `fork()`
- Le processus écrivain ferme le fichier, non utilisé, de lecture du tube
- De même, le processus lecteur ferme le fichier, non utilisé, d'écriture du tube
- Les processus communiquent en utilisant les appels système `write()` et `read()`

– Chaque processus ferme son fichier lorsqu'il veut mettre fin à la communication via le tube

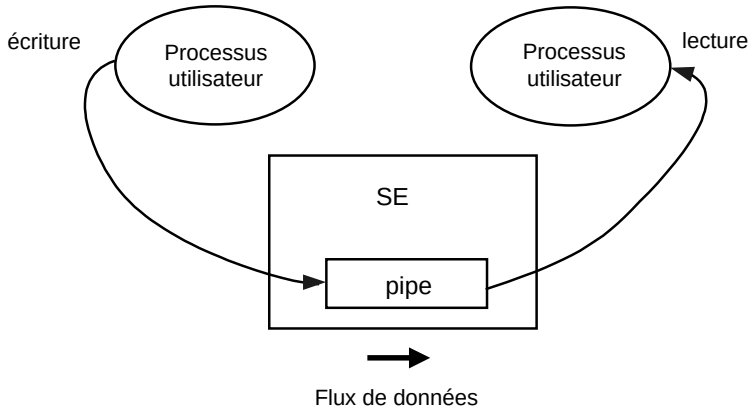


FIG. 5.3 – *Création d'un tube de communication entre deux processus.*

▶ **Exemple 8.** Dans le programme 5.7 `upipe.c` le processus père crée un tube de communication pour communiquer avec son processus fils. La communication est unidirectionnelle du processus fils vers le processus père.

Listing 5.7: *upipe.c*

```
#include <sys/types.h> // types
#include <unistd.h>     // fork , pipe , read , write , close
#include <stdio.h>
#define R 0
#define W 1

int main()
{
int fd[2];
char message[100];    // pour récupérer un message
int nboctets;
char *phrase = "message envoyé au père par le fils";

pipe(fd);              // création d'un tube sans nom
if (fork()==0)   // création d'un processus fils
{
```

```
    // Le fils ferme le descripteur non utilisé de lecture
    close(fd[R]);
    // dépôt dans le tube du message
    write(fd[W],phrase, strlen(phrase)+1) ;
    // fermeture du descripteur d'écriture
    close (fd[W]) ;
}
else
{
    // Le père ferme le descripteur non utilisé d'écriture
    close(fd[W]) ;
    // extraction du message du tube
    nboctets = read (fd[R], message,100) ;
    printf ("Lecture %d octets : %s\n", nboctets, message) ;
    // fermeture du descripteur de lecture
    close (fd[R]) ;
}
return 0;
}
```

Exécution de `upipe.c` :

```
leibnitz> gcc -o upipe upipe.c
leibnitz> upipe
Lecture 35 octets : message envoyé au père par le fils
leibnitz>
```

Remarques : Le processus fils de l'exemple précédent a inclus le caractère nul dans le message envoyé au processus père. Le père peut facilement le supprimer. Si le processus écrivain envoie plusieurs messages de longueurs variables sur le tube, il est nécessaire d'établir des règles qui permettent au processus lecteur de déterminer la fin d'un message, c'est-à-dire, d'établir un **protocole de communication**. Par exemple, le processus écrivain peut précéder chaque message par sa longueur ou bien terminer chaque message par un caractère spécial comme retour chariot ou le caractère nul.

▶ **Exemple 9.** Le programme 5.8 `pipecom.cpp` réalise l'exécution parallèle de deux commandes shell. Un tube est créé pour diriger la sortie standard de la première commande vers l'entrée standard de la deuxième (voir

figure 5.4). Ce programme utilise `dup2()`. `dup()` et `dup2()` sont des appels système qui seront étudiés et détail au Chapitre 11 *Système de fichiers,* mais on peut avancer pour le moment qu'ils s'avèrent très utiles pour la duplication de la sortie standard, dont nous aurons besoin.

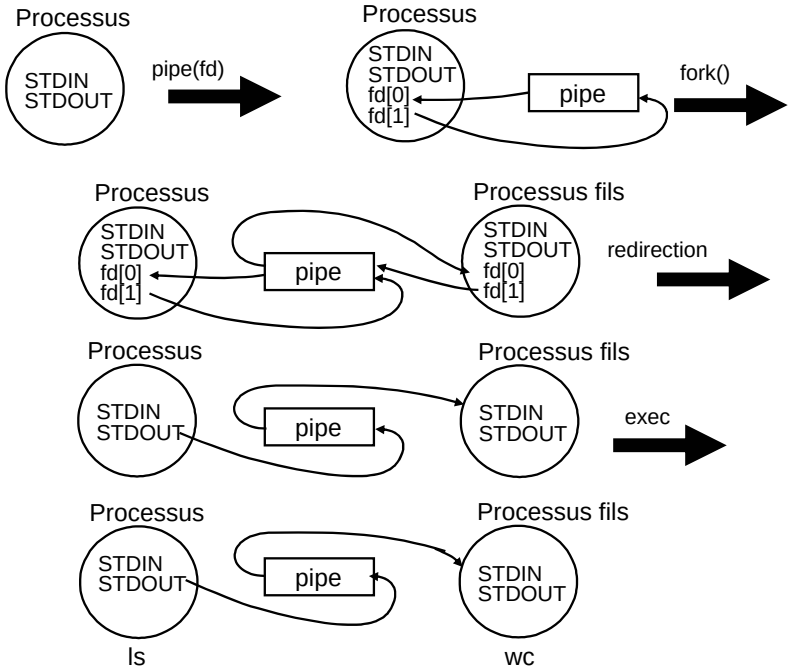


FIG. 5.4 – *Usage des descripteurs de fichiers pendant la communication par tube unidirectionnelle entre deux processus.*

Listing 5.8: *pipecom.cpp*

```
#include <unistd.h> //pour fork , close ...
#include <stdio.h>
#define R 0
#define W 1

int main(int argc, char *argv [])
{
  int fd[2];
  pipe(fd);      // création d'un tube sans nom
  if (fork() !=0)
  {          // Le père ferme le descripteur non utilisé de lecture
```

```
            close(fd[R]);
        // duplication de la sortie standard (son descripteur est 1)
            dup2(fd[W], 1);
        // fermeture du descripteur d'écriture
            close (fd[W]);
        // exécute le programme écrivain
            if(execlp(argv[1], argv[1], NULL) ==-1);
                perror("execlp");
20  }
    else // processus fils (lecteur)
    {
        // fermeture du descripteur non utilisé d'écriture
            close(fd[W]);
        // duplication de l'entrée standard (son descripteur est 0)
            dup2(fd[R],0);
            close (fd[R]); // fermeture du descripteur de lecture
        // exécute le programme lecteur
            execlp(argv[2], argv[2], NULL);
30          perror("connect");
    }
    return 0;
}
```

Exécution de `pipecom.cpp` en utilisant la commande `wc` pour compter le nombre de lignes, mots et caractères de la sortie de `who` sur la machine `leibnitz`:

```
leibnitz> g++ -o pipecom pipecom.cpp
leibnitz> who
venon    pts/0    Nov 28 10:13
jmtorres pts/1    Nov 28 10:22
leibnitz> pipecom who wc
leibnitz>       2      10      62
leibnitz>
```

Communication bidirectionnelle

La communication bidirectionnelle entre processus est possible en utilisant deux tubes : un pour chaque sens de communication, comme illustré à la figure 5.5.

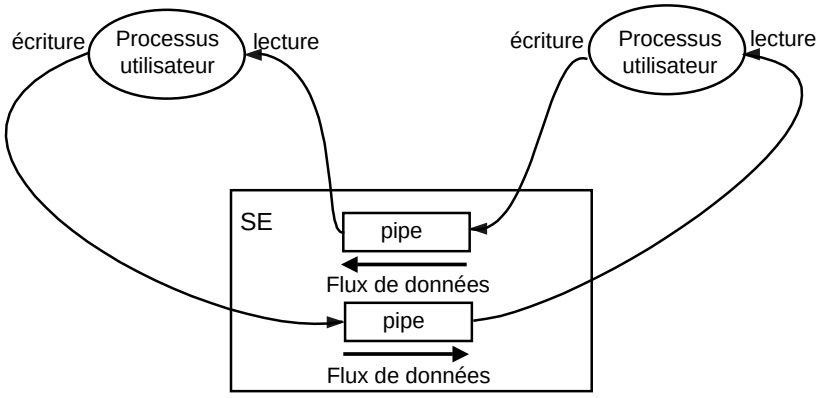


FIG. 5.5 – *Création d'un tube de communication bidirectionnelle entre deux processus.*

5.3.3 Tubes de communication nommés

Le **System V** supporte un autre type de tubes de communication, beaucoup plus performants que ceux sans nom. Ils sont appelés les **tubes nommés** (*named pipes*). Les tubes de communication nommés fonctionnent aussi comme des files du type FIFO (*First In First Out*). Ils sont plus intéressants que les tubes sans nom car ils offrent, en plus, les avantages suivants :

- Ils ont chacun un nom qui existe dans le système de fichiers (une entrée dans la Table des fichiers).
- Ils sont considérés comme des fichiers spéciaux.
- Ils peuvent être utilisés par des processus indépendants, à condition qu'ils s'exécutent sur une même machine.
- Ils existeront jusqu'à ce qu'ils soient supprimés explicitement.
- Ils sont de capacité plus grande environ 40 Ko.

Ainsi, deux processus sans relation parent-enfant peuvent échanger des données au moyen des tubes nommés.

Appels système `mknod()` et commande `mknod`

Les tubes de communication nommés sont créés par la commande shell `mknod` ou par l'appel système `mknod()` :

```
int mknod(char *nom, mode_t mode, dev_t périphérique);
```

où nom contient le nom du tube. mode peut être S_IFREG : fichier régulier vide ou S_IFIFO : tube. [1]
Ainsi, le segment de code suivant permet de créer un tube nommé de type S_IFIFO avec les permissions 0666 :

```
if(mknod(nom_tube, S_IFIFO | 0666, 0) != 0)
  {
         printf("Impossible de créer %s\n", nom_tube);
         exit (1);
  }
```

Il faut dire que l'utilisation de l'appel système mknod() est assez rare. On préfère plutôt la commande mknod. Elle est utilisée à continuation pour créer un tube nommé tube :

```
leibnitz> mknod tube p
leibnitz>
```

On peut faire l'affichage des attributs du tube créé comme on le fait avec les fichiers :

```
leibnitz> ls -l tube
prw-r--r--    1 jmtorres prof           0 Oct  2 11:36 tube|
leibnitz>
```

De la même façon, il est possible de modifier des permissions d'accès :

```
leibnitz> chmod g+rw tube
leibnitz> ls -l tube
prw-rw-r--    1  jmtorres prof          0 Oct  2 11:36 tube|
leibnitz>
```

Remarque : Dans les permissions p indique que tube est un tube.

Une fois le tube créé, il peut être utilisé pour réaliser la communication entre deux processus. Pour ce faire, chacun des deux processus ouvre le tube, l'un en mode écriture et l'autre en mode lecture.

▶ **Exemple 10.** Le programme 5.9 writer.cpp envoie un message sur le tube mypipe. De son coté, le programme 5.10 reader.cpp lit un message à partir du même tube.

---

1. Il y à d'autres modes, mais nous ne serons pas intéressés ici.

Listing 5.9: *writer.cpp*

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>

#define N 100
int main(void)
{
 int fd;
 char message[N];

 sprintf(message, "bonjour du writer [%d]", getpid());
 // Ouverture du tube  mypipe en mode écriture
 fd = open("mypipe", O_WRONLY);
 printf("ici  writer[%d] \n", getpid());
 if (fd!=-1)
   {   // Dépot d'un message sur le tube
       write(fd, message, strlen(message)+1);
   }
 else
   printf("désolé, le tube n'est pas disponible\n");
 close(fd);
 return 0;
}
```

Listing 5.10: *reader.cpp*

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>

#define N 100
int main()
{
  int fd,n;
  char message[N];
       // ouverture du tube mypipe en mode lecture

  fd = open("mypipe", O_RDONLY);
  printf("ici  reader[%d] \n",getpid());
```

```
  if (fd!=-1)
  { // récupérer un message du tube , taille maximale =100
     n = read(fd,message,100);
     if(n>0) // n est le nombre de caractères lus
          printf("%s\n", message);
  }
  else
     printf("désolé, le tube n'est pas disponible\n");
  close(fd);
  return 0;
}
```

Après avoir crée le tube `mypipe` avec `mknod`, on peut compiler séparément les deux programmes :

```
leibnitz> mknod mypipe p
leibnitz> ls -l mypipe
prw-r--r--    1 jmtorres prof      0   Nov 28 10:30 mypipe|
leibnitz> gcc -o writer writer.cpp
leibnitz> gcc -o reader reader.cpp
```

Il est possible de lancer leurs exécutions en arrière plan avec &. Les processus `writer` et `reader` ainsi créés communiquent via le tube de communication `mypipe` :

```
leibnitz> writer& reader&
[1] 14951
[2] 14952
leibnitz> ici writer[14951]

[1]    Done                          writer
leibnitz> ici reader[14952]
bonjour du writer [14951]

[2]    Done                          reader
leibnitz>
```

ou le lancement de l'exécution de deux `writer` et d'un `reader` :

```
leibnitz> writer& writer& reader&
[1] 14953
[2] 14954
[3] 14955
leibnitz> ici writer[14953]
ici writer[14954]
```

```
[2]    Done                          writer
leibnitz>
[1]    Done                          writer
leibnitz> ici reader[14955]
bonjour du writer [14953]

[3]    Done                          reader
leibnitz>
```

5.4 Sockets

Les tubes de communication permettent d'établir une communication unidirectionnelle entre deux processus d'une même machine. Le système Unix/Linux permet la communication entre processus s'exécutant sur des machines différentes au moyen de **sockets**.

Les **sockets** ou **prises** sont les mécanismes traditionnels de communication interprocessus du système Unix. Elles permettent la communication entre processus s'exécutant sur des machines différentes connectées par un réseau de communication. Par exemple, l'utilitaire `rlogin` qui permet à un utilisateur d'une machine de se connecter à une autre machine, est réalisé au moyen de **sockets**. Les sockets sont également utilisées pour imprimer un fichier se trouvant sur une autre machine et pour transférer un fichier d'une machine à autre. Des exemples de communication interprocessus en utilisant les **sockets** seront vus dans le Chapitre 14, *Introduction aux systèmes distribués.*

Suggestions de lecture

Référence : Silverwschatz A., Galvin P., Gagné G., *Principles appliqués des systèmes d'exploitation avec Java*, Vuibert, Paris, France, 2002.

Chapître 4 : Gestion de processus.

Chapître 7 : Synchronisation de processus.

Chapître 20 : Le système Unix.

5.5 Exercices

1. Complétez le programme suivant afin de rendre la communication *bidirectionnelle* entre 2 processus sans laisser de zombies. Le père dort 2 secondes après avoir envoyé son message au fils et le fils dort 4 secondes après avoir envoyé son message au père. On n'a pas besoin de variables additionnelles.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#define R 0
#define W 1

int main()
{
int fdp[2],          // père
    fdf[2];          // fils
char message[100];   // pour récupérer un message
char *phrasef = "message envoyé au père par le fils";
char *phrasep = "message envoyé au fils par le père";

// À COMPLÉTER !!

return 0;
}
```

2. Le programme suivant doit afficher le poème de Vignault, « Mon pays ». A l'aide d'une famille de processus, faire que le programme effectue TOUJOURS la sortie correcte du poème (vers dans leur ordre).
   Restrictions:
   (a) Utiliser **seulement** 3 PROCESSUS.
   (b) Pas besoin des variables additionnelles.
   (c) Chaque processus doit afficher **au moins** un vers. Pas de processus bidons.
   (d) Montrer le père et le pid de chaque processus avant l'affichage du vers.

```
#include <unistd.h>
#include <stdio.h>

int main ()
{
```

```
  int statut2; // statut du 2eme processus
  int statut3; // statut du 3eme processus
  char *vers1 = "Mon pays ce n'est pas un pays,
                 c'est l'hiver\n";
  char *vers2 = "Mon jardin ce n'est pas un jardin,
                 c'est la plaine\n";
  char *vers3 = "Mon chemin ce n'est pas un chemin,
                 c'est la neige.\n";
  char *vers4 = "Mon pays ce n'est pas un pays,
                 c'est l'hiver !\n";

  printf ("-- Mon pays, récité par 3 processus\n") ;
  switch (fork())
    {
      // A compléter !
    }
  return 0;
}
```

3. Le code C/C++ ci-dessous effectue 3 milliards de tours de boucle, et à chaque 100 ms (i.e. 100000 us) le processus est interrompu pour afficher le nombre de tours de boucle effectué à date. La temporisation s'effectue à l'aide de l'appel système `setitimer(...)` qui envoie un signal `SIGPROF` au processus à chaque 100 ms. Le prototype de `setitimer()` est :

   ```
   int setitimer(ITIMER_PROF,const struct itimerval *value,
                 NULL);
   ```

   Complétez le code ci-dessous, de façon à ce que la fonction `sighandler()` soit exécutée à la réception du signal `SIGPROF`. Le signal 29 `SIGPROF` sert au declenchement d'un temporisateur profilé (`setitimer`).
   `sighandler()` doit afficher le nombre de tours de boucle effectué et doit aussi compter le nombre d'interruptions du temporisateur. Avant de terminer son exécution, le processus affiche alors le nombre total des interruptions durant les 3 milliards de tours de boucle.

   ```
   #include <sys/time.h>
   #include <signal.h>
   #include <iostream.h>
  // Définition du gestionnaire du signal de temporisation
   static void sighandler(int sigtype);
   long i = 0; // Compteur
   ```

```
...
// Valeurs du temporisateur par intervalle
struct itimerval v;

int main()
{
   ...
   // Initialisation du temporisateur
   //Valeur des répétitions en secondes ou microsecondes
   v.it_interval.tv_sec = 0;
   v.it_interval.tv_usec = 100000;
   // Valeur initiale en secondes ou microsecondes
   v.it_value.tv_sec = 0;
   v.it_value.tv_usec = 100000;

   printf("\ndébut de la temporisation\n");
   setitimer(ITIMER_PROF, &v, NULL);

for (i=0; i < 3000000000; i++);

   exit(0);
}
...
```

4. En utilisant le diagramme 5.5 sur la communication bidirectionnelle, écrivez le code pour permettre à deux processus d'échanger des informations.
5. Le programme `fils-tube.c` implante la communication entre 2 processus fils par un tube. Le premier processus (producteur) génère une valeur y qui est envoyée au second processus (consommateur). Tracez manuellement le programme et affichez les résultats possibles. Ensuite, exécutez le code en enlevant l'appel système sleep et observez les résultats. Si vous n'observez pas alors une alternance d'exécution entre les processus producteur et consommateur, expliquez-en la raison. Finalement, utilisez l'appel système gettimeofday pour mesurer le temps d'exécution (système et utilisateur en microsecondes) de chacun des processus et expliquez pourquoi ce temps n'est pas exactement le même pour chacun des processus. Si votre temps d'exécution est `0`, alors incrémentez la valeur du paramètre de `sleep()`.

Listing 5.11: *fils-tube.c*

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void)
{
int
    pid1, pid2,    // identificateurs des processus fils
    p1, p2,        // identificateurs des processus terminés
    code_retour1,
    code_retour2, // Codes de terminaison des processus
    x = 0,         // donnée non partagée entre les processus
    y = 0,         // donnée partagée entre les processus
    i,             // indice
    tube[2];       // tube entre les processus
    pipe(tube);    // création du tube
    // Bloc du processus producteur
    pid1=fork();
    // Si pid = 0, alors exécuter code du processus créé
    if (pid1==0)
    {
        for (i = 0; i < 10; ++i)
        {
            // afficher les données
            printf("Producteur : x=%d y=%d\n", ++x, --y);
            // Envoyer y au consommateur via le tube
            write(tube[1], &y, 1);
            // Bloquer le producteur pour 2 secondes
            sleep(2);
        }
        exit(0); // Terminer l'exécution du producteur
    }
    // Bloc du processus consommateur
    pid2=fork();
    // Si pid = 0, exécuter code du processus créé
    if(pid2==0)
    {
      for (i = 0; i < 10; ++i)
      {
          // Extraire y du tube
```

```
            read(tube[0], &y , 1);
            // afficher les données
            printf("Consommateur : x=%d y=%d \n", x++, y);
            // Bloquer le consommateur pour 2 secondes
            sleep (2);
          }
          exit (0); // Terminer l'exécution du producteur
      }
      // Bloc du processus parent
      // Exécution du parent concurremment avec ses fils
      // Afficher les valeurs de x et y
      printf("Parent : x=%d y=%d \n", x, y);
      // Synchroniser le processus parent avec les fils
      p1 = wait(&code_retour1);
      p2 = wait(&code_retour2);
      // Indiquer l'ordre de terminaison des fils
      if ( p1 == pid1)
          printf("\nProducteur a terminé en premier.");
      else
          printf("\nConsommateur a terminé en premier.");
      return 0;    // Terminer le processus parent
} // main
```

6
Synchronisation des processus

DAns un système d'exploitation multiprogrammé en temps partagé, plusieurs processus s'exécutent en pseudo-parallèle ou en parallèle et partagent des objets (mémoires, imprimantes, etc.). Le partage d'objets sans précaution particulière peut conduire à des résultats imprévisibles. La solution au problème s'appelle **synchronisation des processus**.

6.1 Introduction

Considérez par exemple, le fonctionnement d'un *spool* d'impression montré sur la figure 6.1, qui peut être délicat.

FIG. 6.1 – *Spool d'une imprimante.*

Quand un processus veut imprimer un fichier, il doit placer le nom de ce fichier dans un répertoire spécial, appelé répertoire de *spool*. Un autre processus, le démon d'impression, vérifie périodiquement s'il y a des fichiers à imprimer. Si c'est le cas, il les imprime et retire leur nom du répertoire. Supposons que :

– Le répertoire d'impression ait un très grand nombre d'emplacements, numérotés 0,1,2,...,N. Chaque emplacement peut contenir le nom d'un fichier à imprimer.

– Tous les processus partagent la variable `in` qui pointe sur le prochain emplacement libre du répertoire.
– Deux processus A et B veulent placer chacun un fichier dans la file d'impression et que la valeur de `in` est 7.
– Le processus A place son fichier à la position `in` qui vaut 7. Une interruption d'horloge arrive immédiatement après et le processus A est suspendu pour laisser place au processus B.
– Ce dernier place également son fichier à la position `in` qui vaut toujours 7 et met à jour `in` qui prend la valeur 8. Il efface ainsi le nom du fichier placé par le processus A.
– Lorsque le processus A sera relancé, il met à jour la valeur de `in` qui prend la valeur 9.

Sous ces conditions, deux problèmes peuvent se présenter :

– Le fichier du processus A ne sera jamais imprimé;
– **in** ne pointe plus sur le prochain emplacement libre.

Comment éviter ces problèmes ? le problème provient du fait que le processus B a utilisé une variable partagée avant que A ait fini de s'en servir. Si l'on arrive à bien synchroniser les processus, le problème peut être résolu.

Lorsqu'un processus accède à la variable `in`, les autres processus ne doivent ni la lire, ni la modifier jusqu'à ce que le processus ait terminé de la mettre à jour. D'une manière générale, il faut empêcher les autres processus d'accéder à un objet partagé si cet objet est en train d'être utilisé par un processus, ce qu'on appelle d'assurer l'**exclusion mutuelle**. Les situations de ce type, où deux ou plusieurs processus lisent ou écrivent des données partagées et où le résultat dépend de l'ordonnancement des processus, sont qualifiées d'**accès concurrents**.

6.2 Objets et sections critiques

Nous allons utiliser les définitions suivantes :

Objet critique : Objet qui ne peut être accédé simultanément. Comme par exemple, les imprimantes, la mémoire, les fichiers, les variables etc. L'objet critique de l'exemple du *spool* d'impression précédent est la variable partagée `in`.

Section critique : Ensemble de suites d'instructions qui opèrent sur un ou plusieurs objets critiques et qui peuvent produire des résultats imprévisibles lorsqu'elles sont exécutées simultanément par des processus différents.

Ainsi, la section critique de l'exemple du *spool* d'impression est :

- Lecture de la variable `in`.
- L'insertion d'un `nom de fichier` dans le répertoire.
- La mise à jour de `in`.

Imaginez maintenant qu'on désire effectuer une somme des N premiers nombres en utilisant des processus légers. Une vision simpliste pour résoudre ce problème, pourrait suggérer de diviser le travail en deux processus : le premier qui va calculer `S1=1+2+...+M` avec `M=N % 2` et un deuxième qui additionne `S2=M+(M+1)+...+N`, et finalement la variable `somme_totale=S1 + S2` fournira la réponse. La figure 6.2 illustre cette situation. Un code possible pour résoudre ce problème est :

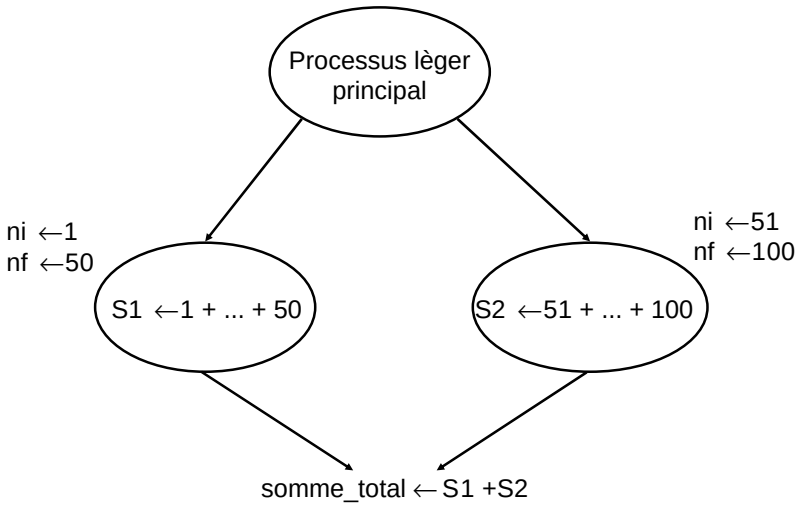


FIG. 6.2 – *Somme de N nombres avec threads.*

```
int somme_totale = 0;
void somme_partielle(int ni, int nf)
   {
       int j = 0;
       int somme_partielle = 0;
       for (j = ni; j <= nf; j++)
           somme_partielle += j;
       somme_totale += somme_partielle;
       pthread_exit(0);
   }
```

Mais, avec cet algorithme, si plusieurs processus exécutent concurremment ce code, on peut obtenir un résultat incorrect. La solution correcte nécessite l'utilisation des **sections critiques**.

Le problème de conflit d'accès serait résolu, si on pouvait assurer que deux processus ne soient jamais en section critique en même temps au moyen de l'**exclusion mutuelle**). En général, les processus qui exécutent des sections critiques sont structurés comme suit :

- Section non critique.
- Demande d'entrée en section critique.
- Section critique.
- Demande de sortie de la section critique.
- Section non critique.

Quatre conditions sont nécessaires pour réaliser correctement une exclusion mutuelle :

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

Comment assurer alors une coopération correcte et efficace des processus ?

6.3 Masquage des interruptions

Avant d'entrer dans une section critique, le processus **masque les interruptions**. Il les restaure à la fin de la section critique. Il ne peut être alors suspendu durant l'exécution de la section critique.

Cependant cette solution est dangereuse, car si le processus, pour une raison ou pour une autre, ne restaure pas les interruptions à la sortie de la section critique, ce serait la fin du système. La solution n'assure pas l'exclusion mutuelle, si le système n'est pas monoprocesseur car le masquage des interruptions concernera uniquement le processeur qui a demandé l'interdiction. Les autres processus exécutés par un autre processeur pourront donc accéder aux objets partagés.

En revanche, cette technique est parfois utilisée par le système d'exploitation pour mettre à jour des variables ou des listes partagées par ses processus, par exemple la liste des processus prêts.

6.4 Exclusion mutuelle par attente active

6.4.1 Les variables de verrouillage

Une autre tentative pour assurer l'exclusion mutuelle est d'utiliser une variable de verrouillage partagée `verrou`, unique, initialisée à 0. Pour rentrer en section critique (voir algorithme de verrouillage), un processus doit tester la valeur du `verrou`. Si elle est égale à 0, le processus modifie la valeur du `verrou` $\leftarrow 1$ et exécute sa section critique. À la fin de la section critique, il remet le `verrou` à 0. Sinon, il attend (par une attente active) que le `verrou` devienne égal à 0, c'est-à-dire : `while(verrou !=0) ;`

Algorithme verrouillage

while verrou $\neq 0$ **do**
  ; / / Attente active
end while
verrou $\leftarrow 1$
Section_critique();
verrou $\leftarrow 0$

Problèmes Cette méthode n'assure pas l'exclusion mutuelle : Supposons qu'un processus est suspendu juste après avoir lu la valeur du verrou qui est égal à 0. Ensuite, un autre processus est élu. Ce dernier teste le verrou qui est toujours égal à 0, met verrou $\leftarrow 1$ et entre dans sa section critique. Ce processus est suspendu avant de quitter la section critique. Le premier processus est alors réactivé, il entre dans sa section critique et met le verrou $\leftarrow 1$. Les deux processus sont en même temps en section critique.

Ainsi cet algorithme *n'est pas correct.* On voit ainsi qu'il est indispensable dans l'énoncé du problème de préciser quelles sont les **actions atomiques**, c'est-à-dire celles qui sont non divisibles. Si on possède sous forme hardware une instruction **test and set** qui sans être interrompue lit une variable, vérifie si elle est égale à 0 et dans ce cas transforme sa valeur en 1,

alors il est facile de résoudre le problème de l'exclusion mutuelle en utilisant un verrou. Toutefois on n'a pas toujours disponible ce type d'instruction, et il faut alors trouver une solution logicielle.

6.4.2 L'alternance

Une autre proposition consiste à utiliser une variable `tour` qui mémorise le tour du processus qui doit entrer en section critique. `tour` est initialisée à `0`.

```
// Processus P1
while (1)
{ // attente active
     while (tour !=0) ;
     Section_critique() ;
     tour = 1 ;
     Section_noncritique() ;
     ...
}
```

```
// Processus P2
while (1)
{ // attente active
     while (tour !=1) ;
     Section_critique() ;
     tour = 0 ;
     Section_noncritique() ;
     ...
}
```

Supposons maintenant que le processus P1 lit la valeur de `tour` qui vaut 0 et entre dans sa section critique. Il est suspendu et P2 est exécuté. P2 teste la valeur de `tour` qui est toujours égale à 0. Il entre donc dans une boucle en attendant que `tour` prenne la valeur 1. Il est suspendu et P1 est élu de nouveau. `P1` quitte sa section critique, met `tour` à `1` et entame sa section non critique. Il est suspendu et P2 est exécuté. P2 exécute rapidement sa section critique, `tour = 0` et sa section non critique. Il teste `tour` qui vaut 0. Il attend que `tour` prenne la valeur 1.

Problème : On peut vérifier assez facilement que deux processus ne peuvent entrer en section critique en même temps, toutefois le problème n'est pas vraiment résolu car il est possible qu'un des deux processus ait plus souvent besoin d'entrer en section critique que l'autre; l'algorithme lui fera attendre son tour bien que la section critique ne soit

pas utilisée. Un processus peut être bloqué par un processus qui n'est pas en section critique.

6.4.3 Solution de Peterson

La solution de Peterson se base sur l'utilisation de deux fonctions : `entrer_region()`; `quitter_region()`;
Chaque processus doit, avant d'entrer dans sa section critique appeler la fonction `entrer_region()` en lui fournissant en paramètre son numéro de processus. Cet appel le fera attendre si nécessaire jusqu'à ce qu'il n'y ait plus de risque. A la fin de la section critique, le processus doit appeler `quitter_region()` pour indiquer qu'il quitte sa section critique et pour autoriser l'accès aux autres processus, comme le montre le code 6.1.

Listing 6.1: *peterson.c*

```
#define FALSE 0
#define TRUE 1
#define N      2        // Nb. processus

int  tour;              // Le tour du processus
int  interesse[N];      // 0   initiale

void entrer_region(int processus)    // processus 0 ou 1
{
    int autre ;         // Autre processus
    autre = 1 - processus ;
    interesse[processus] = TRUE ; // On est interessé
    tour = processus ;  // Positioner le drapeau
                        // Attendre
    while (tour == processus && interesse[autre] ==TRUE) ;
}

void quitter_region (int processus)
{
    // Processus quite region critique
    interesse[processus] = FALSE;
}
```

Prouver la correction de cet algorithme est équivalent à montrer le fait que deux processus ne peuvent pas entrer en section critique en même

temps. Pour y arriver, nous devons réécrire le code, comme montré dans `peterson2.c`:

Listing 6.2: *peterson2.c*

```
#define FALSE 0
#define TRUE 1
#define N     2       // Nb. processus

int  tour;            // Le tour du processus
int  interesse[N];    // 0  initiale

void EntrerSortirRegion(processus)
 {
10   autre = 1 - processus;
   interesse[processus] = TRUE;
   tour = processus;
   while ((tour == i) && interesse[autre ]) ;
   Section_critique ();
   interesse[processus] = FALSE;
 }
```

Pour vérifier que l'algorithme fonctionne correctement et donne accès aux deux processus à la région critique, il faut décomposer cet algorithme en actions atomiques. Voici celles pour le processus `P0`:

```
1. Écrire (true, interesse[0]);
2. Écrire (0, tour);
3. a0 = Lire (interesse[1]);
4. Si (!a0) aller à 7;
5. t0 = Lire(tour);
6. Si (t0 == 0) aller à 3;
7. Section_critique();
8. Écrire (false, interesse[0]);
9. Aller à 1;
```

Les actions atomiques de `P1` sont tout à fait semblables:

```
1'. Écrire (true, interesse[1]);
2'. Écrire (1, tour);
3'. a1 = Lire (interesse[0]);
4'. Si (!a1) aller à 7';
5'. t1 = Lire(tour);
6'. Si (t1 == 1) aller à 3';
```

```
7'. Section_critique();
8'. Écrire (false, interesse[1]);
9'. Aller à 1';
```

Il faut examiner tous les enchevêtrements possibles des deux suites d'actions atomiques. On peut pour des raisons de symétrie supposer que le processus P0 exécute sa première instruction avant P1. Il faut toutefois examiner plusieurs cas par la suite :

Cas 1 : P0 exécute 3 avant que P1 n'exécute 1' les valeurs obtenues par les deux processus sont alors : `a0 = false, t0 = 0, a1 = true, t1 = 1;` Dans ce cas on vérifie que P0 entre en section critique, P1 reste bloqué.

Cas 2 : P1 exécute 1' avant que P0 n'exécute 3, et 2' après que P0 ait exécuté 5

`a0 = true, t0 = 0, a1 = true , t1 = 1;`

On constate qu'aucun des deux processus ne rentre dans la section critique, toutefois immédiatement après le processus P0 obtient :

`a0 = true, t0 = 1`, ainsi P0 entre en section critique, P1 bloqué

Cas 3 : P1 exécute 1' avant que P0 n'exécute 3, et 2' entre 2 et 5. On a dans ce cas : `a0 = true, t0 = 1, a1 = true , t1 = 1;` ainsi P0 entre en section critique, P1 reste bloqué

Cas 4 : P1 exécute 1' avant que P0 n'exécute 2, et 2' avant 2, donc `a0 = true, t0 = 0, a1 = true , t1 = 0;`

P1 entre en section critique et P0 reste bloqué

Ayant démontré la correction de l'algorithme, il faut aussi se poser la question de savoir si deux processus ne peuvent rester bloqués indéfiniment en attendant la section critique, on dirait alors qu'il y a interblocage. Des raisonnements analogues à ceux faits pour la correction montrent qu'il n'en est rien. Cependant, l'algorithme marche seulement pour deux processus.

6.4.4 L'instruction TSL

Les ordinateurs multiprocesseurs ont une instruction atomique indivisible appelée **TSL** (*Test and Set Lock*).

```
tsl registre, verrou
```

L'instruction **TSL** exécute en un seul cycle, de manière indivisible, le chargement d'un mot mémoire dans un registre et le rangement d'une valeur non nulle à l'adresse du mot chargé. Lorsqu'un processeur exécute l'instruction **TSL**, il verrouille le bus de données pour empêcher les autres processeurs d'accéder à la mémoire pendant la durée de l'opération. Cette instruction peut être utilisée pour établir et supprimer des verrous (assurer l'exclusion mutuelle).

```
entrer_region
  Label :
    tsl registre, verrou # Copier verrou dans reg.
                         # verrou = 1
    cmp registre, $0     # verrou = 0 ?
    jnz Label            # si different on boucle (verrou)
    ret                  # ret. entrer section critique

quitter_region
     mov verrou, $0      # mettre 0 dans verrou
     ret                 # quitter section critique
```

6.4.5 Commentaires sur l'attente active

L'attente active se traduit dans une consommation du temps UCT, et un processus en attente active peut rester en permanence dans une boucle infinie. Par exemple, supposons que :

- Deux processus H et B, tels que H est plus prioritaire que B partagent un objet commun. Les règles d'ordonnancement font que H sera exécuté dès qu'il se trouve dans l'état prêt.
- Pendant que H était en E/S, le processus B est entré dans sa section critique.
- Le processus H est redevenu prêt avant que B ait quitté sa section critique.
- Le processus B est ensuite suspendu et le processus H est élu. H effectue alors une attente active et B ne peut être sélectionné puisque H est plus prioritaire.

Conclusion ? le processus B ne peut plus sortir de sa section critique et H boucle indéfiniment.

Une dernière question est de vérifier qu'il y a un minimum d'équité entre les deux processus, c'est-à-dire que l'un d'entre eux ne peut pas attendre indéfiniment son tour d'accès à la section critique alors que l'autre y

accède un nombre infini de fois. Cette question est plus difficile, de même que la généralisation de la solution à un nombre quelconque de processus.

6.5 Primitives SLEEP et WAKEUP

`SLEEP()` est un appel système qui suspend l'appelant en attendant qu'un autre le réveille. `WAKEUP(processus)` est un appel système qui réveille un`processus`. Par exemple, un processus H qui veut entrer dans sa section critique est suspendu si un autre processus B est déjà dans sa section critique. Le processus H sera réveillé par le processus B, lorsqu'il quitte la section critique.

Problème : Si les tests et les mises à jour des conditions d'entrée en section critique ne sont pas exécutés en exclusion mutuelle , des problèmes peuvent survenir. Si le signal émis par `WAKEUP()` arrive avant que le destinataire ne soit endormi, le processus peut dormir pour toujours.

Exemples

```
// Processus P1
 {
    if(cond ==0 ) SLEEP() ;
    cond = 0 ;
    section_critique ( ) ;
    cond = 1 ;
    WAKEUP(P2) ;
    ...
 }

// Processus P2
{
    if (cond ==0 ) SLEEP() ;
    cond = 0 ;
    section_critique ( ) ;
    cond = 1 ;
    WAKEUP(P1) ;
    ...
}
```

Cas 1 : Supposons que :

- P1 est exécuté et il est suspendu juste après avoir lu la valeur de `cond` qui est égale à `1`.

- P2 est élu et exécuté. Il entre dans sa section critique et est suspendu avant de la quitter.
- P1 est ensuite exécuté. Il entre dans sa section critique.
- Les deux processus sont dans leurs sections critiques.

Cas 2 : Supposons que :

- P1 est exécuté en premier. Il entre dans sa section critique. Il est suspendu avant de la quitter ;
- P2 est élu. Il lit `cond` qui est égale à `0` et est suspendu avant de la tester.
- P1 est élu de nouveau. Il quitte la section critique, modifie la valeur de `cond` et envoie un signal au processus P2.
- Lorsque P2 est élu de nouveau, comme il n'est pas endormi, il ignorera le signal et va s'endormir pour toujours.

Dans le système Unix, les appels système qui assurent à peu près, les mêmes fonctions que les primitives SLEEP et WAKEUP sont :

- `pause( )` : suspend le processus appelant jusqu'à réception d'un signal.
- `kill(pid, SIGCONT)` : envoie le signal `SIGCONT` au processus `pid`.

L'émission d'un signal est possible si l'une des deux conditions est vérifiée :

1. L'émetteur et le destinataire du signal appartiennent au même propriétaire
2. L'émetteur du signal est le super-utilisateur.

6.6 Problème du producteur et du consommateur

Dans le problème du **producteur** et du **consommateur** deux processus partagent une mémoire tampon de taille fixe, comme montré à la figure 6.3. L'un d'entre eux, le **producteur**, met des informations dans la mémoire tampon, et l'autre, les retire. Le **producteur** peut produire uniquement si le tampon n'est pas plein. Le **producteur** doit être bloqué tant que le tampon est plein. Le **consommateur** peut retirer un objet du tampon uniquement si le tampon n'est pas vide. Le **consommateur** doit être bloqué tant que le tampon est vide. Les deux processus ne doivent pas accéder à la fois au tampon.

▶ **Exemple 1.** Une solution classique au problème du producteur et du consommateur [Tan89] est montré sur le programme `schema-prod-cons.c` avec l'utilisation des primitives SLEEP et WAKEUP.

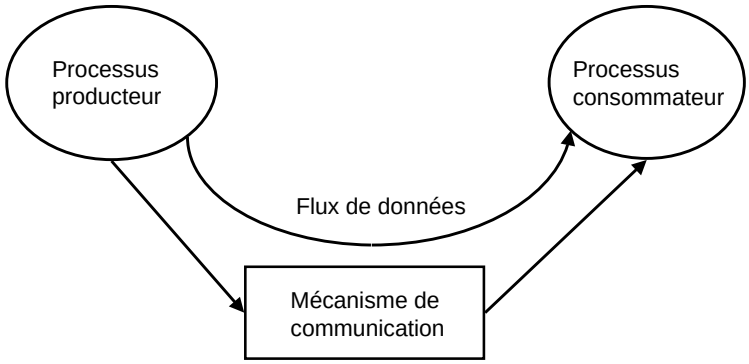


FIG. 6.3 – *Problème du producteur/consommateur.*

Listing 6.3: *schema-prod-cons.c*

```
#define TRUE 1
#define N 50            // Nb. de places dans le tampon
int compteur = 0;       // Nb. objets dans le tampon

void producteur( )
{
 while(TRUE)
    {
        produire_objet();
        if (compteur == N)
            SLEEP();                // Tampon plein
        mettre_objet ();            // dans tampon
        compteur++;
        if (compteur == 1)          // Tampon vide
            WAKEUP(consomateur);
    }
}

void consomateur( )
{
 while(TRUE)
    {
        if (!compteur)
            SLEEP();                // tampon vide
```

```
        retirer_objet ();          // du tampon
        compteur--;
        if (compteur == N-1)       // réveiller producteur
            WAKEUP(producteur);
        consommer_objet();
    }
}
```

6.6.1 Critique des solutions précédentes

La généralisation de toutes ces solutions aux cas de plusieurs processus est bien complexe. Le mélange d'exclusion mutuelle et de suspension est toujours délicat à réaliser et à implanter.

6.7 Sémaphores

Pour contrôler les accès à un objet partagé, E. W. Dijkstra suggéra en 1965 l'emploi d'un nouveau type de variables appelées les **sémaphores**. Un sémaphore est un compteur entier qui désigne le nombre d'autorisations d'accès à une section critique. Il a donc un nom et une valeur initiale, par exemple `semaphore S = 10`.

Les sémaphores sont manipulés au moyen des opérations `P` ou wait et `V` ou signal. L'opération `P(S)` décrémente la valeur du sémaphore `S` si cette dernière est supérieure à `0`. Sinon le processus appelant est mis en attente. Le test du sémaphore, le changement de sa valeur et la mise en attente éventuelle sont effectués en une seule opération atomique indivisible. L'opération `V(S)` incrémente la valeur du sémaphore `S`, si aucun processus n'est bloqué par l'opération `P(S)`. Sinon, l'un d'entre eux sera choisi et redeviendra prêt. `V` est aussi une opération indivisible.

Les sémaphores permettent de réaliser des **exclusions mutuelles**, comme illustré à la figure 6.4.

semaphore mutex=1;		
processus P1: P(mutex) section critique de P1; V(mutex);		processus P2; P(mutex) section critique de P2; V(mutex);

FIG. 6.4 – *Exclusion mutuelle avec des sémaphores.*

La figure 6.5 montre l'exclusion mutuelle d'une petite section critique `a++`.

int a; semaphore mutex=1;		
processus P1: P(mutex) a++; V(mutex);		processus P2; P(mutex) a++; V(mutex);

FIG. 6.5 – *Exclusion mutuelle d'une section critique composée d'une variable a.*

Sur la figure 6.6 on montre la valeur que les sémaphores prennent lors des appels des `P (wait)` et `V (signal)`, pour trois processus P_0, P_1 et P_2.

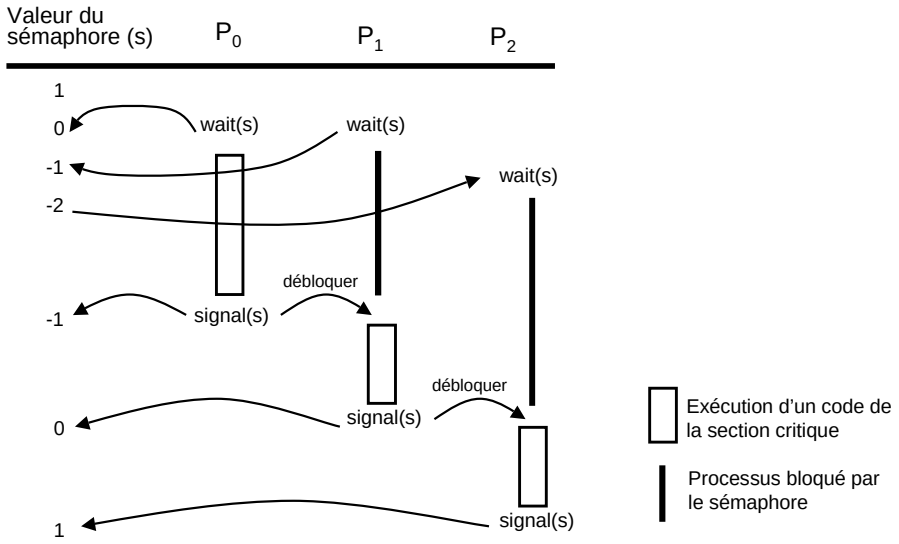


FIG. 6.6 – *Valeurs de sémaphores lors des appels V (signal) et P (wait).*

6.8 Services Posix sur les sémaphores

Le système d'exploitation Linux permet de créer et d'utiliser les **sémaphores** définis par le standard Posix. Les services Posix de manipulation

des sémaphores se trouvent dans la librairie `<semaphore.h>`. Le type sémaphore est désigné par le type `sem_t`.

- `int sem_init(sem_t *sem, int pshared, unsigned int valeur)`
  Initialisation d'un sémaphore. `sem` est un pointeur sur le sémaphore à initialiser; `value` est la valeur initiale du sémaphore; `pshared` indique si le sémaphore est local au processus `pshared=0` ou partagé entre le père et le fils `pshared` $\neq$ `0`. Actuellement Linux ne supporte pas les sémaphores partagés.
- `int sem_wait(sem_t *sem)` est équivalente à l'opération `P` : retourne toujours `0`.
- `int sem_post(sem_t *sem)` est équivalente à l'opération `V` : retourne `0` en cas de succès ou `-1` autrement.
- `int sem_trywait(sem_t *sem)` décrémente la valeur du sémaphore `sem` si sa valeur est supérieure à `0` ; sinon elle retourne une erreur. C'est une opération atomique.
- `int sem_getvalue (sem_t* nom, int * sval);` récupérer la valeur d'un sémaphore : il retourne toujours `0`.
- `int sem_destroy (sem_t* nom);` détruire un sémaphore. Retourne `-1` s'il y a encore des `waits`.

Les sémaphores Posix servent donc à synchroniser les *communications entre threads*. Les sémaphores d'Unix **System V** sont un autre type de mécanisme qui permet la synchronisation entre processus. Sa gestion étant un peu plus complexe, on verra ce mécanisme dans le Chapitre 7, *Communications IPC System V*.

▶ **Exemple 2.** Le programme 6.4 `sem-posix.c` montre l'utilisation de sémaphores Posix.

Listing 6.4: *sem-posix.c*

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

sem_t my_sem;
```

```
#define SEMINITVALUE 1
void * critical_section (void *);
#define MAXTHREADS 100

int main(int argc, char *argv[])
{
    int n, i, error;
    pthread_t threadid[MAXTHREADS];
    if ((argc != 2) || ((n = atoi(argv[1])) <= 0))
    {
        printf("Usage: %s nombre de threads\n", argv[0]);
        exit (1);
    }
    // initialiser le sémaphore
    sem_init(&my_sem, 0, SEMINITVALUE);
    // création des threads
    for (i = 0; i < n; ++i)
      pthread_create(&threadid[i], NULL,critical_section, (void *)i );
    // attendre la fin des threads
    for (i = 0; i < n; ++i)
      pthread_join(threadid[i], NULL);
  exit (0);
}

void * critical_section (void *arg)
{
    int myindex, i, n;
    // extraire l'index des threads
    myindex = (int)arg;
    sem_wait(&my_sem);
    // section critique
    for (i=1; i < 5; i++)
    {
        printf("%d : %d\n", myindex, i);
        if ((n=sleep (1)) != 0) // dormir 1 seconde
        printf("interrupted, no of secs left %d\n",n);
    }
    // quiter section critique
    sem_post(&my_sem);
    pthread_exit(NULL);
```

```
}
```

Sortie du programme `posix-sem.c`:

```
leibnitz> gcc -o posix-sem posix-sem.c -lpthread
leibnitz> posix-sem 2
0 : 1
0 : 2
0 : 3
0 : 4
1 : 1
1 : 2
1 : 3
1 : 4
leibnitz>
```

▶ **Exemple 3.** Producteur / consommateur avec sémaphores. La solution du problème au moyen des sémaphores Posix utilise trois sémaphores dans le programme 6.5. Le premier sémaphore, nommé `plein`, compte le nombre d'emplacements occupés. Il est initialisé à `0`. Le second, nommé `vide`, compte le nombre d'emplacements libres. Il est initialisé à `N` (la taille du tampon). Le dernier, nommé `mutex`, assure l'exclusion mutuelle pour l'accès au tampon (lorsqu'un processus utilise le tampon, les autres ne peuvent pas y accéder).

Listing 6.5: *prod-cons.cpp*

```
#include <semaphore.h>
#include <unistd.h> // sleep
#include <pthread.h>
#include <stdio.h>

#define taille 3
sem_t plein, vide, mutex;
int tampon[taille];
pthread_t cons,prod;
void* consommateur(void *);
void* producteur(void *);

int main()
{
    // initiliser les sémaphores
```

```
    sem_init(&plein,0,0);
    sem_init(&vide,0, taille );
    sem_init(&mutex,0, 1);
    // créer les threads
    pthread_create(&cons,NULL,consommateur,NULL);
    pthread_create(&prod,NULL,producer,NULL);
    // attendre la fin des threads
    pthread_join(prod,NULL);
    pthread_join(cons,NULL);
    printf("fin des thread \n");
    return 0;
}

void* consommateur(void *)
{
    int ic=0, nbcons = 0, objet;
    do {
        sem_wait(&plein);
        sem_wait(&mutex);
        // consommer
        objet = tampon[ic];
        sem_post(&mutex);
        sem_post(&vide);
        printf("ici cons.: tampon[%d]= %d \n", ic, objet);
        ic=(ic+1)%taille;
        nbcons++;
        sleep(2);
    } while ( nbcons<=5 );
    return (NULL);
}

void* producteur(void *)
{
   int ip=0, nbprod=0, objet=0;
   do {
        sem_wait(&vide);
        sem_wait(&mutex);
        // produire
        tampon[ip]=objet;
        sem_post(&mutex);
```

```
            sem_post(&plein);
            printf("ici  prod.: tampon[%d]= %d\n", ip,objet);
            objet++;
            nbprod++;
            ip=(ip+1)%taille;
        } while ( nbprod<=5 );
        return NULL;
}
```

Sortie du programme `prod-cons.cpp`:

```
leibnitz> g++ -o prod-cons prod-cons.cpp -lpthread
leibnitz> prod-cons
ici cons. :tampon[0]= 0
ici prod. : tampon[0]= 0
ici prod. : tampon[1]= 1
ici prod. : tampon[2]= 2
ici prod. : tampon[0]= 3
ici cons. :tampon[1]= 1
ici prod. : tampon[1]= 4
ici cons. :tampon[2]= 2
ici prod. : tampon[2]= 5
ici cons. :tampon[0]= 3
ici cons. :tampon[1]= 4
ici cons. :tampon[2]= 5
fin des  thread
leibnitz>
```

6.8.1 Problème des philosophes

Il s'agit d'un problème théorique très interessant propossé et résolu aussi par Dijkstra. Cinq philosophes sont assis autour d'une table. Sur la table, il y a alternativement cinq plats de spaghettis et cinq fourchettes (figure 6.7). Un philosophe passe son temps à manger et à penser. Pour manger son plat de spaghettis, un philosophe a besoin de deux fourchettes qui sont de part et d'autre de son plat. Écrire un programme qui permette à chaque philosophe de se livrer à ses activités (penser et manger) sans jamais être bloqué.

Il est important de signaler que si tous les philosophes prennent *en même temps* chacun une fourchette, aucun d'entre eux ne pourra prendre l'autre fourchette (situation d'interblocage). Pour éviter cette situation, un philosophe ne prend jamais une seule fourchette. Les fourchettes sont les

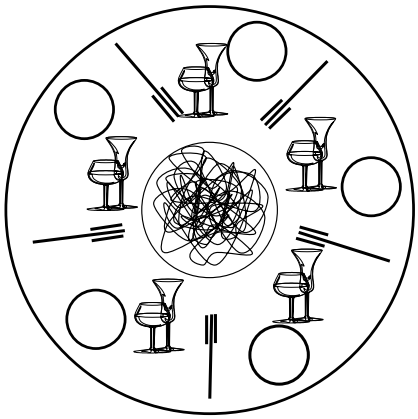


FIG. 6.7 – *Problème des philosophes.*

objets partagés. L'accès et l'utilisation d'une fourchette doit se faire en exclusion mutuelle. On utilisera le sémaphore `mutex` pour réaliser l'exclusion mutuelle.

▶ **Exemple 4.** Une solution au problème des philosophes.

Listing 6.6: *philosophe.c*

```
#include <stdio.h>
#include <unistd.h>     // pour sleep
#include <pthread.h>    // pour les threads
#include <semaphore.h> // pour les sémaphores

#define N 5                 // nombre de philosophes
#define G ((N+i-1)%N) // philosophe de gauche de i
#define D (i)               // philosophe de droite de i
#define libre 1
#define occupe 0
int fourch[N] = {libre, libre, libre, libre, libre};
sem_t mutex;
void * philosophe(void *);

int main()
{
    int NumPhi[N] = {0,1,2,3,4};
    int i;
```

```
    pthread_t ph[N];
    sem_init(&mutex,0, 1) ;
    // création des N philosophes
    for(i=0; i<N; i++)
        pthread_create(&ph[i],NULL,philosophe,&(NumPhi[i]));
    // attendre la fin des threads
    i=0;
    while(i<N && (pthread_join(ph[i++],NULL)==0));
    printf("fin des threads \n");
    return 0;
}

// La fonction de chaque philosophe
void * philosophe(void *num)
{
    int i =*(int *) num;
    int nb = 2;
    while (nb)
    {
        // penser
        sleep (1);
        // essayer de prendre les fourchettes pour manger
        sem_wait(&mutex) ;
        if(fourch[G] && fourch[i])
        {
            fourch[G ] = 0 ;
            fourch[i ] =0 ;
            printf("philosophe [%d] mange \n", i);
            sem_post(&mutex) ;
            nb--;
            // manger
            sleep (1) ;
            // libérer les fourchettes
            sem_wait(&mutex) ;
            fourch[G ] = 1 ;
            fourch[i ] = 1 ;
            printf("philosophe[%d] a fini de manger\n",i);
            sem_post(&mutex) ;
        }
        else sem_post(&mutex);
```

```
    }
}
```

Problème : Cette solution résout le problème d'interblocage. Mais, un philosophe peut mourir de faim car il ne pourra jamais obtenir les fourchettes nécessaires pour manger (problème de famine et d'équité).

Pour éviter le problème de famine il faut garantir que si un processus demande d'entrer en section critique, il obtient satisfaction au bout d'un temps fini. Dans le cas des philosophes, le problème de famine peut être évité, en ajoutant `N` sémaphores (un sémaphore pour chaque philosophe). Les sémaphores sont initialisés à `0`. Lorsqu'un philosophe `i` ne parvient pas à prendre les fourchettes, il se met en attente (`P(S[i])`). Lorsqu'un philosophe termine de manger, il vérifie si ses voisins sont en attente. Si c'est le cas, il réveille les voisins qui peuvent manger en appelant l'opération `V`. On distingue trois états pour les philosophes : penser, manger et faim.

▶ **Exemple 5.** Une meilleure solution au problème des philosophes.

Listing 6.7: *philosophe2.c*

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define N 5
#define G ((N+i-1)%N)  // philosophe de gauche de i
#define D (i)          // philosophe de droite de i
enum etat { penser, faim, manger};
etat Etat[N] = {penser, penser, penser, penser, penser};
sem_t S[N];
sem_t mutex;
void * philosophe(void *);
void Test(int i);

int main()
{
  int i, NumPhi[N] = {0,1,2,3,4};
  pthread_t ph[N];
  sem_init(&mutex, 0, 1);
```

```
  for(i=0; i<N; i++)
    sem_init(&S[i],0,0);
  // création des N philosophes
  for(i=0; i<N; i++)
    pthread_create(&ph[i],NULL,philosophe,&(NumPhi[i]));
  // attendre la fin des threads
  for(i=0; (i<N && pthread_join(ph[i],NULL)==0); i++);
  printf("fin des threads\n");
  return 0;
}

void * philosophe(void * num)
{
  int i=*(int *) num;
  int nb=2;
  while(nb)
  {
    nb--;
    // penser
    sleep(1);
    // tenter de manger
    sem_wait(&mutex);
    Etat[i]=faim;
    Test(i);
    sem_post(&mutex);
    sem_wait(&S[i]);
    // manger
    printf("philosophe[%d] mange\n", i);
    sleep(1);   // manger
    printf("philosophe[%d] a fini de manger\n", i);
    // libérer les fourchettes
    sem_wait(&mutex);
    Etat[i] = penser;
    // vérifier si ses voisins peuvent manger
    Test(G);
    Test(D);
    sem_post(&mutex);
  }
}
// procedure qui vérifie si le philosophe i peut manger
```

```
void Test(int i)
{
  if ( ( Etat[i] == faim) && ( Etat[G] != manger)
       && (Etat[D] !=manger) )
  {
       Etat[i] = manger ;
       sem_post(&S[i]) ;
  }
}
```

Exécution de philosophe2.c

```
leibnitz> g++ -o philosophe2 philosophe2.c -lpthread
leibnitz> philosophe2
philosophe[0] mange
philosophe[2] mange
philosophe[4] mange
philosophe[0] a fini de manger
philosophe[2] a fini de manger
philosophe[1] mange
philosophe[4] a fini de manger
philosophe[3] mange
philosophe[0] mange
philosophe[1] a fini de manger
philosophe[2] mange
philosophe[3] a fini de manger
philosophe[4] mange
philosophe[0] a fini de manger
philosophe[2] a fini de manger
philosophe[1] mange
philosophe[3] mange
philosophe[4] a fini de manger
philosophe[1] a fini de manger
philosophe[3] a fini de manger
fin des  threads
leibnitz>
```

6.8.2 Problème des rédacteurs et des lecteurs

Ce problème modélise les accès à une base de données. Un ensemble de processus tente constamment d'accéder à la base de données soit pour écrire, soit pour lire des informations. Pour assurer une certaine cohérence des données de la base, il faut interdire l'accès (en lecture et en écriture) à tous les processus, si un processus est en train de modifier la base (accède à

la base en mode écriture). Par contre, plusieurs processus peuvent accéder à la base, en même temps, en mode lecture. Les rédacteurs représentent les processus qui demandent des accès en écriture à la base de données. Les lecteurs représentent les processus qui demandent des accès en lecture à la base de données (figure 6.8).

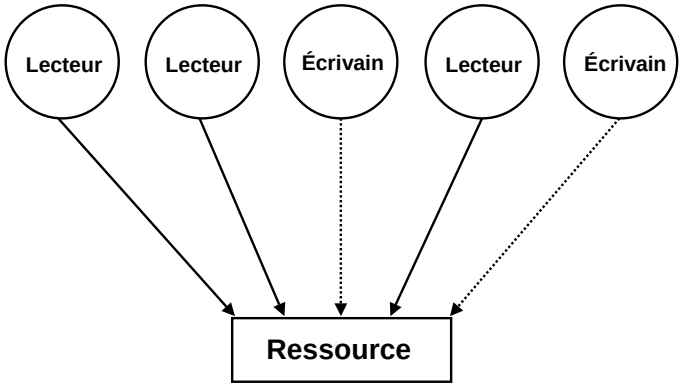


FIG. 6.8 – *Problème des lecteurs et écrivains.*

Pour contrôler les accès à la base, on a besoin de connaître le nombre de lecteurs (`NbL`) qui sont en train de lire. Le compteur `NbL` est un objet partagé par tous les lecteurs. L'accès à ce compteur doit être exclusif (sémaphore mutex). Un lecteur peut accéder à la base, s'il y a déjà un lecteur qui accède à la base (`NbL>0`) ou aucun rédacteur n'est en train d'utiliser la base. Un rédacteur peut accéder à la base, si elle n'est pas utilisée par les autres (un accès exclusif à la base). Pour assurer cet accès exclusif, on utilise un autre sémaphore : `Redact`. Des algorithmes adéquats pour le `lecteur()` et le `redacteur()` sont les suivants :

```
NbL ← 0 / / Nb Lecteurs
semaphore Redact / / ecrivain
semaphore Mutex / / acces a la base
lecteur:
repeat
   P(Mutex)
   if (NbL == 0) then
      P(Redact) / / Premier lecteur empêche écriture
   end if
```

```
    NbL ← NbL + 1
    V(Mutex)
    / / lecture de la base
    / / fin de l'accès à la base
    P(Mutex)
    NbL ← NbL - 1
    if (NbL == 0) then
      V(Redact) / / Dernier lecteur habilite l'écriture
    end if
    V(Mutex)
  until TRUE

  redacteur:
  repeat
    P(Redact)
    / / modifier les données de la base
    V(Redact)
  until TRUE
```

▶ **Exemple 6.** Le programme `red-lec.c` fait l'implantation de l'algorithme des rédacteurs et lecteurs. Il fixe le nombre de lecteurs et rédacteurs à `N=3`, ainsi que le nombre d'accès (afin d'éviter une boucle infinie) à `ACCES=4`. On a simulé des opérations de lecture de la base, des modifications des données et d'écriture dans la base avec des `sleep()` d'une seconde.

Listing 6.8: *red-lec.c*

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define N 3
#define ACCES 4
int NbL = 0;
sem_t Redact;
sem_t mutex ;
void * lecteur(void *);
void * redacteur(void *);
```

```
int main()
{
    int i;
    int numred[N] = {0,1,2};
    int numlec[N] = {0,1,2};
    pthread_t red[N];
    pthread_t lec[N];
    sem_init(&mutex, 0,1);
    sem_init(&Redact,0,1);
    for( i=0; i<N; i++)
     {
    // création des redacteurs
        pthread_create(&red[i],NULL,lecteur,&(numred[i]));
    // création des lecteurs
        pthread_create(&lec[i], NULL,redacteur,&(numlec[i]));
     }
    // attendre la fin des threads
    for(i=0; i<N; i++)
    {
        pthread_join(red[i], NULL);
        pthread_join(lec[i], NULL);
    }
    printf("fin des threads\n");
    return 0;
}

void * lecteur(void * num)
{
  int i =*(int *) num;
  int x=ACCES;
  do
  {
       printf("Lecteur %d\n",i);
       sem_wait(&mutex);
       if (!NbL)
            sem_wait(&Redact);
        NbL++;
        sem_post(&mutex);
        // lecture de la base
```

```
        sleep (1);
        // fin de l'accès à la base
        sem_wait(&mutex);
        NbL--;
        if (!NbL)
            sem_post(&Redact);
        sem_post(&mutex);
60      // traitement des données lues
        sleep (1);
    } while(--x);
}

void * redacteur(void * num)
{
    int i =*(int *) num;
    int x=ACCES;
    do
70  {
        printf("Redacteur %d\n",i);
        sem_wait(&Redact);
        // modifier les données de la base
        sleep (1);
        sem_post(&Redact);
    } while(--x);
}
```

Nous montrons ensuite une exécution de `red-lec.c`:

```
leibnitz> gcc -o red-lec red-lec.c -lpthread ; red-lec
Lecteur 0
Redacteur 0
Lecteur 1
Redacteur 1
Lecteur 2
Redacteur 2
Redacteur 0
Lecteur 0
Lecteur 1
Lecteur 2
Redacteur 1
Redacteur 2
Redacteur 0
```

```
Lecteur 0
Lecteur 1
Redacteur 1
Lecteur 2
Redacteur 2
Redacteur 0
Lecteur 0
Lecteur 1
Redacteur 1
Lecteur 2
Redacteur 2
fin des threads
leibnitz>
```

6.9 Compteurs d'événements

Un **compteur d'événements** est un compteur associé à un événement. Sa valeur indique le nombre d'occurrences de l'événement associé. Il peut être utilisé pour synchroniser des processus. Par exemple, pour réaliser un traitement un processus attend jusqu'à ce qu'un compteur d'événement atteigne une valeur. Trois opérations sont définies pour un compteur d'événement `E` :

`Read(E)` : donne la valeur de `E`
`Advance(E)` : incrémente `E` de `1` de manière atomique
`Await(E,v)` : attend que `E` atteigne ou dépasse la valeur `v`

6.9.1 Producteur/consommateur

Pour le problème du producteur et du consommateur, nous pouvons utiliser deux compteurs d'événements `in` et `out`. Le premier compte le nombre d'insertions dans le tampon alors que le second compte le nombre de retraits du tampon depuis le début.

`in` : est incrémentée de `1` par le producteur après chaque insertion dans le tampon.
`out` : est incrémentée de `1` par le consommateur après chaque retrait du tampon.

Le producteur doit s'arrêter de produire lorsque le tampon est plein. La condition à satisfaire pour pouvoir produire est : `N > in - out`. Posons `sequence = in + 1`. La condition devient alors :

out ≥ sequence - N

De même, le consommateur peut consommer uniquement si le tampon n'est pas vide. La condition à satisfaire pour pouvoir consommer est : in - out > 0. Posons sequence = out + 1. La condition devient alors :

in ≥ sequence.

```
// producteur consommateur

 #define N 100 // taille du tampon
 int in = 0, out = 0 ;

 void producteur (void)
 {
    int objet , sequence =0, pos =0 ;
    while (1)
    {
        objet = objet+1 ;
        sequence = sequence + 1 ;
        // attendre que le tampon devienne non plein
        Await(out, sequence-N) ;
        tampon[pos]= objet ;
        pos = (pos+1)%N ;
        // incrémenter le nombre de dépôts
        Advance(&int) ;
    }
 }

 void consommateur (void)
 {
    int objet , sequence =0, pos =0 ;
    while (1)
    {
        sequence = sequence + 1 ;
        // attendre que le tampon devienne non vide
        Await(in,sequence) ;
        objet = tampon[pos] ;
        // incrémenter le compteur de retraits
        Advance(&out) ;
        printf(" objet consommé [%d]:%d ", pos, objet) ;
        pos = (pos+1)%N ;
    }
 }
```

6.10 Moniteurs

L'idée des moniteurs est de regrouper dans un module spécial, appelé **moniteur**, toutes les sections critiques d'un même problème. Un moniteur est un ensemble de procédures, de variables et de structures de données.

Les processus peuvent appeler les procédures du moniteur mais ils ne peuvent pas accéder à la structure interne du moniteur à partir des procédures externes. Pour assurer l'exclusion mutuelle, il suffit qu'il y ait à tout moment au plus un processus actif dans le moniteur. C'est le compilateur qui se charge de cette tâche. Pour ce faire, il rajoute, au début de chaque procédure du moniteur un code qui réalise ce qui suit : s'il y a processus actif dans le moniteur, alors le processus appelant est suspendu jusqu'à ce que le processus actif dans le moniteur se bloque en exécutant un `wait` sur une variable conditionnelle (`wait(c)`) ou quitte le moniteur. Le processus bloqué est réveillé par un autre processus en lui envoyant un signal sur la variable conditionnelle (`signal(c)`).

Cette solution est plus simple que les sémaphores et les compteurs d'événements, puisque le programmeur n'a pas à se préoccuper de contrôler les accès aux sections critiques. Mais, malheureusement, à l'exception de JAVA, la majorité des compilateurs utilisés actuellement ne supportent pas les moniteurs.

6.10.1 Producteur/consommateur

Les sections critiques du **problème du producteur et du consommateur** sont les opérations de dépôt et de retrait dans le tampon partagé. Le dépôt est possible si le tampon n'est pas plein. Le retrait est possible si le tampon n'est pas vide.

```
Moniteur ProducteurConsommateur
{
 //variable conditionnelle pour non plein et non vide
 bool nplein, nvide ;
 int compteur =0, ic=0, ip=0 ;

 // section critique pour le dépôt
 void mettre (int objet)
 {
   if (compteur==N) wait(nplein) ;
   //attendre jusqu'à ce que le tampon devienne non plein
   // déposer un objet dans le tampon
   tampon[ip] = objet ;
```

```
    ip = (ip+1)%N ;
    compteur++ ;
    // si le tampon était vide avant le dépôt,
    // envoyer un signal pour réveiller le consommateur.
    if (compteur==1) signal(nvide) ;
  }

  // section critique pour le retrait
  void retirer (int* objet)
  {
    if (compteur ==0) wait(nvide) ;
    objet = tampon[ic] ;
    compteur -- ;
    if(compteur==N-1) signal(nplein) ;
  }
 }
```

6.11 Échanges de messages

La synchronisation entre processus peut être réalisée au moyen d'**échange de messages**, dans Unix System V. Un processus peut attendre qu'un autre lui envoie un message.

L'avantage principale est qu'il n'y a pas de partage d'espace de données, et donc il n'y aura pas de conflit d'accès. Comme problèmes, on peut signaler que les messages envoyés peuvent se perdre. Lorsqu'un émetteur envoie un message, il attend pendant un certain délai la confirmation de réception du message par le récepteur (un acquittement). Si, au bout de ce délai, il ne reçoit pas l'acquittement, il renvoie le message. Il peut y avoir des messages en double en cas de perte du message d'acquittement. Pour éviter cela, on associe à chaque message une identification unique. L'authentification des messages se fait alors par l'utilisation des clés.

Le mecanisme detaillé d'échange de messages sera étudié dans le Chapitre 7, *Communications IPC System V*.

6.12 Producteur/consommateur par tubes

Le producteur et le consommateur communiquent au moyen d'un tube de communication nommé. Le producteur dépose ses objets un à un dans le tube. Il se bloque lorsqu'il tente de déposer un objet et le tube n'a pas suffisamment de place. Il est réveillé quand le tube a suffisamment d'espace libre. De son côté, le consommateur récupère un à un des objets du tube.

Il se bloque lorsqu'il tente de récupérer un objet à partir d'un tube vide. Il est réveillé lorsque le tube devient non vide. Le schéma de programme `p-c-pipe.c` montre l'utilisation de cette technique.

Listing 6.9: *p-c-pipe.c*

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
 int fildes [2];     // pipe pour synchroniser
 char c;             // caractère pour synchroniser

 pipe( fildes );
 write( fildes [1], &c , 1);    // nécessaire pour entrer dans la
                                // section critique la première fois
  if (fork () == 0)             // processus fils
  {
    for (;;)
    {
          read( fildes [0], &c , 1);    // entrée section critique
          // < Section critique >
          write( fildes [1], &c , 1);    // quiter section critique
    }
  }
  else   // processus père
  {
    for (;;)
    {
          read( fildes [0], &c , 1);     // entrée section critique
            // < section critique >
          write( fildes [1], &c , 1);    // quitter section critique
    }
  }
 return 0;
}
```

6.13 Exclusion mutuelle (mutex) de threads

La correcte synchronisation entre plusieurs fils d'exécution est très importante. Pour accéder à des données globales partagées, il est indispensable de mettre en œuvre un mechanisme permettant de protèger une variable partagée par plusieurs threads. Ce mécanisme est appelé `mutex` (*MUTual EXclusion*) et repose sur l'utilisation du type `pthread_mutex_t`. Ainsi une variable sert de verrou à une zone de mémoire globale particulière. Les appels d'exclusion mutuelle de threads consistent notamment en l'initialisation, le verrouillage, le déverrouillage et la destruction.

Initialisation

On peut initialiser un mutex statique ou dynamiquement. Ainsi on peut initialiser un mutex statiquement comme :

```
pthread_mutex_t mutex = PHTREAD_MUTEX_INITIALIZER;
```

ou en utilisant l'initialisation dynamique avec l'appel système :

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *attributs);
```

Avec les déclarations :

```
pthread_mutex_t mutex;
pthread_mutexattr_t mutexattr;
...
phtread_mutex_init(&mutex, &mutexattr);
...
```

Verrouillage

L'appel système de verrouillage des threads est :

```
phtread_mutex_lock(phtread_mutex_t *mutex);
```

Déverrouillage

L'appel système de déverrouillage des threads est :

```
phtread_mutex_unlock(phtread_mutex_t *mutex);
```

▶ **Exemple 7.** Le programme 6.10 `sum.c` somme séquentiellement `N` nombres.

Listing 6.10: *sum.c*

```
#include <stdio.h>

main(int argc, char **argv)
{
  double sum;
  int N, i;

  N = atoi(argv[1]);    // Nombre
  sum = 0;              // somme
  for (i=1; i<=N; ++i)
  {
    sum += i;
  }
  printf("Somme = %10.0f\n", sum);
  return(0);
}
```

Le programme `sum-thread.c` somme les mêmes `N` nombres, mais en forme parallèle avec deux threads qui utilisent mutex.

Listing 6.11: *sum-thread.c*

```
#include <stdio.h>
#include <pthread.h>

volatile double somme = 0.0; // somme (shared)
pthread_mutex_t somme_lock; // Lock
volatile double N;           // N

void *processus(void *arg)
{
  double localsum;
  int i;
  int iproc = (*((char *) arg) - '0');
  localsum = 0;              // somme
  for (i=iproc; i<=N; i+=2)
  {
```

```
    localsum += i;
  }
  // Lock de somme, calculer et débloquer
  pthread_mutex_lock(&somme_lock);
  somme += localsum;
  pthread_mutex_unlock(&somme_lock);
  return(NULL);
}

int main(int argc, char **argv)
{
  pthread_t thread0, thread1;
  void *retval;

  N = atoi(argv[1]);        // Nombre
  // Initialiser le lock de la somme
  pthread_mutex_init(&somme_lock, NULL);
  // 2 threads
  if (pthread_create(&thread0, NULL, processus, "0") ||
      pthread_create(&thread1, NULL, processus, "1"))
  {
    printf("%s: erreur thread\n", argv[0]);
    exit (1);
  }
  // Join les deux threads
  if (pthread_join(thread0, &retval) ||
      pthread_join(thread1, &retval))
    {
      printf("%s: erreur join\n", argv[0]);
      exit (1);
    }
  printf("Somme (partagée) = %10.0f\n", somme);
  return 0;
}
```

▶ **Exemple 8.** Soit un tableau M de N éléments rempli par un thread lent et lu par un autre plus rapide. Le thread de lecture doit attendre la fin du remplissage du tableau avant d'afficher sont contenu. Les mutex peuvent

protèger le tableau pendant le temps de son remplissage.

Listing 6.12: *mutex-threads2.c*

```
#include <stdio.h>
#include <pthread.h>

#define N 5
static pthread_mutex_t mutex;
int M[N];

void *lecture(void * arg)
{
  int i;

  pthread_mutex_lock(&mutex);
  for (i=0; i < N; i++)
    printf("Lécture: M[%d] = %d\n", i, M[i]);
  pthread_mutex_unlock(&mutex);
  pthread_exit(0);
}

void *ecriture(void * arg)
{
  int i;

  pthread_mutex_lock(&mutex);
  for (i=0; i < N; i++)
  {
    M[i] = i;
    printf("Écriture : M[%d] = %d\n", i, M[i]);
    sleep (2); // Ralentir le thread d'ecriture
  }
  pthread_mutex_unlock(&mutex);
  pthread_exit(0);
}

int main(void)
{
  pthread_t th1, th2;
  void *ret;
```

```
  pthread_mutex_init(&mutex, NULL);
40
  pthread_create(&th1, NULL, ecriture, NULL);
  pthread_create(&th2, NULL, lecture, NULL);

  pthread_join(th1, &ret );
  pthread_join(th2, &ret );
  return 0;
}
```

Nous montrons ensuite une exécution de `mutex-threads2.c`:

```
leibnitz> gcc -o mt2 mutex-threads2.c -lpthread
leibnitz> mt2
Écriture: M[0] = 0
Écriture: M[1] = 1
Écriture: M[2] = 2
Écriture: M[3] = 3
Écriture: M[4] = 4
Lécture: M[0] = 0
Lécture: M[1] = 1
Lécture: M[2] = 2
Lécture: M[3] = 3
Lécture: M[4] = 4
leibnitz>
```

Sans l'utilisation des mutex, probablement on obtiendrait des sorties érronées comme la suivante:

```
Écriture: M[0] = 0
Lécture: M[0] = 0
Lécture: M[1] = 0
Lécture: M[2] = 0
Lécture: M[3] = 0
Lécture: M[4] = 0
Écriture: M[1] = 1
Écriture: M[2] = 2
Écriture: M[3] = 3
Écriture: M[4] = 4
```

6.14 Sémaphores avec System V (Solaris)

Le Unix System V sur Solaris permet de créer et d'utiliser les sémaphores. Les fonctions de manipulation des sémaphores sont dans la librairie `<synch.h>`. Le type sémaphore est désigné par le mot `sema_t`. L'initialisation d'un sémaphore est réalisée par l'appel système :

```
#include <synch.h>
int sema_init(sema_t*sp, unsigned int count, int type,
              NULL);
```

où `sp` est un pointeur sur le sémaphore à initialiser; `count` est la valeur initiale du sémaphore; `type` indique si le sémaphore est utilisé pour synchroniser des processus légers (threads) ou des processus. Le type peut être `USYNC_PROCESS` ou bien `USYNC_THREAD`. Par défaut, le type sera `USYNC_THREAD`.

- `int sema_wait (sema_t *sp)` est équivalente à l'opération `P`.
- `int sema_post(sema_t *sp)` est équivalente à l'opération `V`.
- `int sema_trywait(sema_t *sp)` décrémente la valeur du sémaphore `sp` si sa valeur est supérieure à `0`; sinon elle retourne une erreur. C'est une opération atomique.

▶ **Exemple 9.** Usage de sémaphores System V.

Listing 6.13: *v-semaphore.c*

```
#include <synch.h> // sémaphores
#include <thread.h> // thr_create et thr_join
#include <stdio.h> // printf
#define val 1

sema_t mutex; // sémaphore
int var_glob=0;
void* increment(void *);
void* decrement(void *);
10

int main()
{
    // initialiser mutex
    sema_init(&mutex,val,USYNC_THREAD,NULL);
    printf("ici main : var_glob = %d\n", var_glob);
```

```
    // création d'un thread pour increment
    thr_create(NULL,0,increment,NULL,0,NULL);
    // création d'un thread pour decrement
    thr_create(NULL,0,decrement,NULL,0,NULL);
    // attendre la fin des threads
    while(thr_join(NULL,NULL,NULL)==0);
    printf("ici main, fin threads : var_glob =%d \n",var_glob);
    return 0;
}

void* decrement(void *)
{
    int nb = 3;
    // attendre l'autorisation d'accès
    sema_wait(&mutex);
    while(nb--)
    {
      var_glob -= 1;
      printf("ici sc de decrement : var_glob=%d \n",
             var_glob);
    }
    sema_post(&mutex);
    return (NULL);
}

void* increment (void *)
{
    int nb=3;
    sema_wait(&mutex);
    while(nb--)
    {
      var_glob += 1;
      printf("ici sc de increment : var_glob = %d \n",
             var_glob);
    }
    sema_post(&mutex);
    return (NULL);
}
```

Exécution du programme `v-semaphore.c` sur la machine Unix nommé

```
jupiter:
 jupiter% gcc v-semaphore.c -lthread -o mutex
 jupiter% mutex
 ici main : var_glob = 0
 ici sc de increment : var_glob = 1
 ici sc de increment : var_glob = 2
 ici sc de increment : var_glob = 3
 ici sc de decrement : var_glob = 2
 ici sc de decrement : var_glob = 1
 ici sc de decrement : var_glob = 0
 ici main, fin threads : var_glob = 0
```

▶ **Exemple 10.** Problème du Producteur/consommateur.

Listing 6.14: *v-prod-cons.c*

```
#include <synch.h>
#include <unistd.h> // sleep
#include <thread.h>
#include <stdio.h>
#define taille 3

sema_t plein, vide,mutex;
int tampon[taille];
void* consommateur(void *);
void* producer(void *);

int main()
{
    // initiliser les sémaphores
    sema_init(&plein,0,USYNC_THREAD,NULL);
    sema_init(&vide,taille,USYNC_THREAD,NULL);
    sema_init(&mutex,1,USYNC_THREAD,NULL);
    // créer les threads
    thr_create(NULL,0,consommateur,NULL,0,NULL);
    thr_create(NULL,0,producer,NULL,0,NULL);
    // attendre la fin des threads
    while((thr_join(NULL,NULL,NULL)==0));
    printf("fin des thread \n");
    return 0;
}
```

```
void* consommateur(void *retrait)
{
    int ic=0, nbcons = 0, objet;
    do
        {
            sema_wait(&plein);
            sema_wait(&mutex);
            // consommer
            objet = tampon[ic];
            sema_post(&mutex);
            sema_post(&vide);
            printf("\n ici cons.: tampon[%d]= %d\n", ic,
                    objet);
            ic=(ic+1)%taille;
            nbcons++;
            sleep(2);
        } while ( nbcons<=5 );
    return (NULL);
}

void* producer(void *)
{
    int ip=0, nbprod=0, objet=0;
    do
        {
            sema_wait(&vide);
            sema_wait(&mutex);
            // produire
            tampon[ip]=objet;
            sema_post(&mutex);
            sema_post(&plein);
            printf("\n ici prod.: tampon[%d]= %d\n", ip,
                    objet);
            objet++;
            nbprod++;
            ip=(ip+1)%taille;
        } while ( nbprod<=5 );
    return NULL;
}
```

Exécution de producteurs et consommateurs :

```
jupiter% gcc prod_cons.c -lthread o- prod_cons
jupiter% prod_cons
ici prod. : tampon[0]= 0
ici prod. : tampon[1]= 1
ici prod. : tampon[2]= 2
ici cons. :tampon[0]= 0
ici prod. : tampon[0]= 3
ici cons. :tampon[1]= 1
ici prod. : tampon[1]= 4
ici cons. :tampon[2]= 2
ici prod. : tampon[2]= 5
ici cons. :tampon[0]= 3
ici cons. :tampon[1]= 4
ici cons. :tampon[2]= 5
fin des thread
jupiter%
```

▶ **Exemple 11.** Problème des philosophes.

Listing 6.15: *v-philosophe.c*

```
#include <stdio.h>      // printf
#include <unistd.h>     // sleep
#include <thread.h>     // threads
#include <synch.h>      // sémaphores

#define N 5             // nombre de philosophes
#define G (N+i-1)%N     // philosophe à gauche de i
#define D (i+1)%N       // philosophe à droite de i
#define libre 1
#define occupe 0
int fourch[N] = {libre , libre , libre , libre , libre };
sema_t mutex;
void * philosophe(void * );

int main( )
{
  int NumPhi[N] = {0,1,2,3,4};
  sema_init(&mutex, 1, NULL, NULL);
  // création des N philosophes
  for(int i=0; i<N; i++)
```

```
        thr_create(NULL,0,philosophe,&(NumPhi[i]),0,NULL);
    // attendre la fin des threads
    while((thr_join(NULL,NULL,NULL)==0));
    printf("fin des threads\n");
    return 0;
}

// La fonction de chaque philosophe
 void * philosophe( void * num)
 {
    int i =* (int *) num;
    int nb = 2;
    while (nb)
    {
            // penser
            sleep (1);
            // essayer de prendre les fourchettes pour manger
            sema_wait(&mutex);
            if(fourch[G] && fourch[i])
            {
                fourch[G] = 0;
                fourch[i] =0;
                sema_post(&mutex);
                nb--;
                // manger
                printf ("philosophe[%d] mange\n", i);
                sleep (1); // manger
                printf("philosophe[%d] a fini de manger\n", i);
                // libérer les fourchettes
                sema_wait(&mutex);
                fourch[G] = 1;
                fourch[i] = 1;
                sema_post(&mutex);
            }
            else sema_post(&mutex);
    }
 }
```

Exécution de la première version des philosophes :

```
jupiter% v-philosophe.c -lthread -o philosophe
```

```
jupiter% philosophe
philosophe[0] mange
philosophe[2] mange
philosophe[0] a fini de manger
philosophe[2] a fini de manger
philosophe[4] mange
philosophe[1] mange
philosophe[4] a fini de manger
philosophe[4] mange
philosophe[1] a fini de manger
philosophe[1] mange
philosophe[4] a fini de manger
philosophe[3] mange
philosophe[1] a fini de manger
philosophe[3] a fini de manger
philosophe[2] mange
philosophe[0] mange
philosophe[0] a fini de manger
philosophe[2] a fini de manger
philosophe[3] mange
philosophe[3] a fini de manger
fin des threads
jupiter%
```

▶ **Exemple 12.** Solution améliorée des Philosophes (Solaris) version 2

Listing 6.16: *v-philosophe2.c*

```
#include <stdio.h>
#include <unistd.h>
#include <thread.h>
#include <synch.h>
#define N 5
#define G (N+i-1)%N  // philosophe à gauche de i
#define D (i+1)%N    // philosophe à droite de i

enum etat { penser, faim, manger};
int fourch[N] = {libre , libre , libre , libre , libre };
etat Etat [N] = {penser, penser, penser, penser, penser};
sema_t S[N]; sema_t mutex;
void * philosophe(void *);

int main()
```

```
{
    int i , NumPhi[N] = {0,1,2,3,4};
    sema_init(&mutex, 1, NULL, NULL) ;
    for ( i =0 ; i<N ; i++)
        sema_int(&S[i],0,NULL,NULL) ;
    // création des N philosophes
    for( i =0; i<N; i++)
    thr_create(NULL,0,philosophe,&(NumPhi[i]),0,NULL);
    // attendre la fin des threads
    while((thr_join(NULL,NULL,NULL)==0));
    printf("fin des threads \n");
    return 0;
}

void * philosophe( void * num)
{
    int i =*( int *) num ;
    int nb = 2 ;
    while (nb)
    {
        nb--;
        // penser
        sleep ( 1) ;
        // tenter de manger
        sema_wait(&mutex) ;
        Etat[i]=faim ;
        test(i ) ;
        sema_post(&mutex) ;
        sema_wait(&S[i]) ;
        // manger
        printf ("philosophe[%d] mange \n", i) ;
        sleep (1) ; // manger
        printf("philosophe[%d] a fini de manger \n", i) ;
        // libérer les fourchettes
        sema_wait(&mutex) ;
        Etat[i ] = penser ;
        // vérifier si ses voisins peuvent manger
        Test(G) ;
        Test(D) ;
        sema_post(&mutex) ;
```

```
    }
}

// procédure qui vérifie si le philosophe i peut manger
void Test(int i)
{
  if ( ( Etat[i] == faim) && ( Etat[G] != manger)
                          && (Etat[D] !=manger) )
        {
            Etat[i] = manger;
            sema_post(&S[i]);
        }
}
```

Exécution de la deuxième version des philosophes :

```
jupiter% v-philosophe2.c -lthread -o philosophe2
jupiter% philosophe2
philosophe[0] mange
philosophe[2] mange
philosophe[0] a fini de manger
philosophe[2] a fini de manger
philosophe[4] mange
philosophe[1] mange
philosophe[4] a fini de manger
philosophe[3] mange
philosophe[1] a fini de manger
philosophe[0] mange
philosophe[0] a fini de manger
philosophe[1] mange
philosophe[3] a fini de manger
philosophe[4] mange
philosophe[4] a fini de manger
philosophe[3] mange
philosophe[1] a fini de manger
philosophe[3] a fini de manger
philosophe[2] mange
philosophe[2] a fini de manger
fin des threads
jupiter%
```

6.15 Exercises

1. Pourquoi le partage de données pose des problèmes dans un système multiprogrammé en temps partagé?
2. Le système Unix permet-il de contrôler les accès aux données partagées?
3. Qu'est-ce qu'une section critique?
4. Expliquez la raison pour laquelle il est nécessaire de synchroniser les processus. Est-ce que tous les types de SE utilisent la synchronisation de processus?
5. Identifiez et expliquez un problème typique qui nécessiterait la synchronisation de processus.
6. Identifiez le point (i.e. les énoncés) d'un processus qui représente sa section critique.
7. Identifiez le problème principal qui peut survenir lorsque l'accès à la section critique d'un processus est contrôlé par une variable booléenne de type verrou. Donnez une solution à ce problème.
8. Quel est le problème principal que résout un sémaphore en ce qui concerne l'accès à une section critique?
9. Quand doit-on utiliser des sémaphores binaire et général? Peut-on utiliser ces deux types de sémaphores dans un même processus et pour un même problème de synchronisation? Expliquez.
10. Pourquoi un tube Unix utilise généralement deux canaux de communication entre deux processus, soit un canal d'entrée et un canal de sortie? Existe-il des alternatives à ce mode de communication utilisant les tubes Unix?
11. Considérez l'application suivante qui implante un système interactif qui lit dans une boucle des caractères entrés via le clavier et qui les affiche à l'écran. Le traitement se termine lorsque l'usager presse la touche `<Ctrl-d>` pour insérer le caractère de fin de fichier (`EOF`) dans le tampon.

```
// Processus parent
char *tampon;   // tampon partagé par les processus
int i=0, j=0;
// positions de lecture et d'écriture dans le tampon
// Processus lecture
   ...
   while (read(stdin, tampon, 1) != NULL)
   ...
```

```
// Processus écriture
   ...
   while (write(stdout, tampon, 1) != NULL)
   ...
```

(a) Synchronisez la communication entre les processus lecture et écriture à l'aide de tubes et d'appels système `fork`.

(b) Synchronisez la communication entre les threads lecture et écriture à l'aide de sémaphores.

(c) Considérez la solution suivante au problème d'accès à des ressources partagées en mode de lecture et d'écriture.

- Les processus en lecture et en écriture sont en compétition pour l'accès à une ressource partagée.
- Un seul processus à la fois peut être dans sa section critique, ceci est valable autant pour les processus en lecture et en écriture.
- Le système ne fait pas de différence entre les processus en lecture et en écriture, et chaque processus peut postuler une requête d'accès à la ressource qui lui sera accordée dès qu'un processus a fini de lire ou d'écrire sur la ressource.
- Cependant, le système définit des processus prioritaires pour l'écriture de la ressource, et les requêtes de ces processus doivent être honorées avant celles des processus normaux en lecture et en écriture dès que la section critique se libère.

Montrez la synchronisation des processus en lecture et en écriture, en utilisant des sémaphores.

12. Considérez le buffer circulaire montré sur la figure 6.9, qui répresente une variation du problème du producteur/consommateur. Résoudre le problème avec des sémaphores.

13. Le code suivant contient une solution proposée au problème d'accès à une section critique. Expliquez pourquoi cette solution serait correcte ou incorrecte, selon qu'elle respecte les 4 principes fondamentaux régissant l'accès à une section critique.

```
/* Variables partagées par les processus P0 et P1. */
 int flag[2];    // Tableau de booléens stockant 0 ou 1
           // Variable indiquant le processus actif,
           // i.e. 0 pour P0 ou 1 pour P1
 int turn;
```

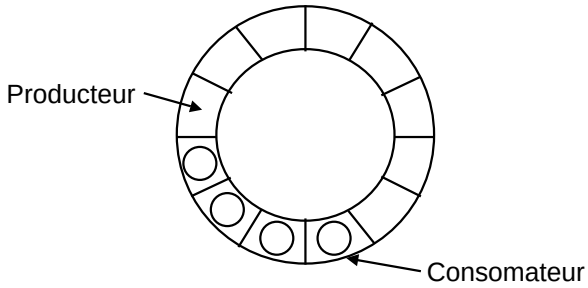


FIG. 6.9 – *Buffer circulaire.*

```
// Code du processus P0
do
{
   flag[0] = 1;       // Mettre le flag de P0 à 1
   while (turn != 0)  // Si P0 est inactif alors, ...
   {
   while (flag[1]) // Attendre l'inactivité de P1
       {   /* skip */ }
       turn = 0;   // Indique que P0 est maintenant actif
   }
   ...
   <Section critique pour le processus P0>
   ...
   // Réinitialiser le flag de P0 afin que P1 puisse
   // s'activer dans sa section critique
   flag[0] = 0;
   ...
   <Autres énoncés pour P0>
   ...
} while (1);

/* Code du processus P1 */
do
{
   flag[1] = 1;    /* Mettre le flag de P1 à 1 */
   while (turn != 1)   /* Si P1 est inactif alors, ... */
   {
       while (flag[0]) /* Attendre l'inactivité de P0 */
       {   /* skip */ }
       turn = 1;   /* Indiquez que P1 est maintenant actif */
```

```
    }
    ...
    <Section critique pour le processus P1>
    ...
    // Réinitialiser le flag de P1 afin que P0 puisse
    // s'activer dans sa section critique
    flag[1] = 0;
    ...
    <Autres énoncés pour P1>
    ...
 } while (1);
```

14. Dijkstra a proposé les 3 solutions suivantes comme étant des solutions logicielles au problème d'accès à une section critique. Par contre, aux vues de Dijkstra ces solutions n'étaient pas tout à fait correctes. Alors, expliquez pourquoi chacune des solutions est effectivement incorrecte.

a)
```
int turn;
    proc(int i)
    {   while (TRUE)
        {   compute;
            while (turn != i);
            <section critique>
            turn = (i+1) mod 2;
        }
    }
    turn = 1;
    fork(proc, 1, 0);
    fork(proc, 1, 1);
```

b)
```
boolean flag[2];
    proc(int i)
    {   while (TRUE)
        {   compute;
            while (flag[(i+1) mod 2]);
            flag[i] = TRUE;
            <section critique>
            flag[i] = FALSE;
        }
    }
    flag[0] = flag[1] = FALSE;
    fork(proc, 1, 0);
    fork(proc, 1, 1);
```

c)
```
boolean flag[2];
    proc(int i)
```

```
{   while (TRUE)
    {   compute;
        flag[i] = TRUE;
        while (flag[(i+1) mod 2]);
        <section critique>
        flag[i] = FALSE;
    }
}
flag[0] = flag[1] = FALSE;
fork(proc, 1, 0);
fork(proc, 1, 1);
```

15. Considérez une route à deux voies (une voie par direction) et orientée selon un axe nord-sud. Cette route traverse un tunnel qui ne possède qu'une seule voie, et par conséquent son accès doit être partagé en temps par les voitures provenant de chacune des deux directions (nord ou sud). Afin de prévenir les collisions, des feux de circulation ont été installés à chacune des extrémités du tunnel. Un feu vert indique qu'il n'y a aucune voiture dans le tunnel provenant de la direction opposée, et un feu rouge indique qu'il y a une voiture à contresens dans le tunnel. Le trafic dans le tunnel est contrôlé par un ordinateur central qui est prévenu de l'arrivée de véhicules par des senseurs posés dans chaque direction. Lorsqu'un véhicule approche le tunnel en direction nord ou sud, le senseur externe exécute la fonction arrivée(direction) pour prévenir l'ordinateur central. Similairement, lorsqu'une voiture sort du tunnel le senseur interne exécute la fonction départ (direction). Concevez une esquisse de code pour synchroniser l'accès au tunnel afin que les feux de circulation fonctionnent correctement dans les deux directions.

16. On a 3 processus P_1, P_2 et P_3 concurrents. Le code *indépendant* de chaque processus $P_i (i = 1,2,3)$ est le suivant :

```
Pi()     // Processus Pi avec i=1,2,3
{
   while(TRUE)
       printf("je suis le processus" i );
}
```

On veut montrer à l'écran la sortie suivante :

```
je suis le processus 1
je suis le processus 2
je suis le processus 3
je suis le processus 1
```

```
je suis le processus 2
je suis le processus 3
...
```

Synchroniser les 3 processus à l'aide des sémaphores binaires.

17. Le programme suivant permet de calculer une estimation de la valeur de π avec un flot unique d'exécution. Parallélisez ce programme à l'aide d'exclusion mutuelle de threads et en respectant l'esprit de la boucle `for (i=0; i<intervals; ++i){...}`.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{
 double width, sum;
 int intervals, i;

 // nombre d'intervalles
 intervals = atoi(argv[1]);
 width = 1.0 / intervals;

 // calcul de pi
 sum = 0;
 for (i=0; i<intervals; ++i)
 {
   double x = (i + 0.5) * width;
   sum += 4.0 / (1.0 + x * x);
 }
 sum *= width;
 printf("Estimation de pi = %10.9f\n", sum);
 return(0);
}
```

7
Communications IPC System V

Les paquets de communication entre processus (*Inter Process Communication* IPC) qu'offre le System V (Unix, Linux, etc.) sont composés de trois mécanismes fondamentaux : les sémaphores, la mémoire partagée et les files de messages.

- Les sémaphores permettent la synchronisation des processus.
- La mémoire partagée permet que les processus partagent son espace des adresses virtuelles.
- Les files de messages permettent l'échange des données avec un format déterminé.

Bien que la communication IPC ne soit pas fondée sur les descripteurs de fichiers, il existe de nombreuses applications qu'en utilisent ces techniques.

7.1 Files de messages

Une file de messages est une structure de données gérée par le noyau, où les processus peuvent introduire et extraire de l'information. La responsabilité d'éviter qu'un conflit se produise entre les processus à l'entrée de la file est uniquement du noyau du système d'exploitation.

La gestion de la file suit le comportement d'un mécanisme FIFO, c'est à dire que les messages se placent dans l'ordre d'arrivée et ils sont livrés dans le même ordre. Pour améliorer le système de gestion, les communications IPC permettent de spécifier un type, ce qui génère des files en fonction du type. Ainsi, on peut demander des message par leur type. Le système retournera le message correspondant du sous-groupe de messages spécifiés.

Un aspect à considérer quand on parle des files de messages, c'est que

leurs éléments (figure 7.1) ne sont pas des messages, mais des têtes de messages dans lesquelles se trouvent les pointeurs aux adresses où les messages sont stockés.

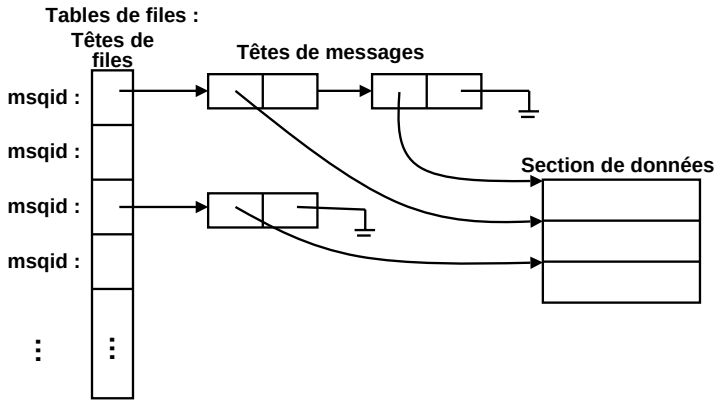


FIG. 7.1 – *Files de messages.*

7.1.1 Génération des clés

Chaque file de message est marquée par un identificateur, c'est-à dire un numéro qui la distingue du reste des files de messages qui sont actives dans le système. Pour créer cet identificateur on a besoin d'une clé (*key*), ainsi tous les mécanismes créés auront un identificateur unique. Cette clé est une variable ou une constante du type `key_t`, dont le type est défini dans `<types.h>`. Il existe plusieurs façons de créer les clés :

1. Le serveur crée une nouvelle structure en spécifiant la clé du type `IPC_PRIVATE` définie dans `<sys/types.h>`.
2. Une autre forme plus simple est celle où le serveur et le client se mettent d'accord pour la clé à utiliser. Ce mécanisme est très rudimentaire, et on a le risque de dupliquer un identificateur.
3. Finalement on a l'option d'utiliser la fonction `ftok()`, fournie par la bibliothèque standard de *C* pour la création des clés. Cette dernière méthode est la plus sûre et la plus utilisée.

La déclaration de `ftok()` est la suivante :

```
#include <types.h>
#include <sys/ipc.h>
key_t ftok(char *fichier, char proj);
```

Cette fonction permet la création d'une clé fondée sur les valeurs de `fichier`, le nom d'un fichier accessible au processus qui demande la création de la clé, et de `proj`, l'identificateur du projet. Pour un même fichier, différents identificateurs de projet vont générer des clés différentes. Si le fichier n'est pas accessible par le projet, —il n'existe pas ou il n'y a pas la permission— `ftok()` retournera la valeur `-1` (du type `key_t`), qui indique erreur dans la génération :

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
int main()
{
  key_t llave;
  if ((llave=ftok(".",'e')) == (key_t) -1)
  {
   printf("Erreur dans la création de la clé");
   exit(1);
  }
  ...
}
```

7.1.2 Fonctions de manipulation de files

`msgget()`

Pour créer une nouvelle file de messages ou accéder à une qui existe déjà, on utilise l'appel système `msgget()`. Cette fonction retourne l'identificateur de la file de messages existant et si la file n'existe pas, elle permet de la créer. Dans le cas qu'un erreur se produise la fonction retournera la valeur `-1`. Le prototype est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget (key_t cle,int attr);
```

Le premier paramètre est la valeur clé. Le paramètre `attr` est un bitmap, semblable aux deux derniers arguments d'`open()`, et où on peut insérer des permissions d'accès :

- `IPC_CREAT | 0XYZ` : Pour créer une file avec permissions, s'il n'y en a aucune associée.

– `IPC_EXCL` : Créera toujours une file.

Note : Si en plus de `IPC_CREAT`, on utilise `IPC_EXCL` le système retournera une erreur dans le cas d'une file déjà existante.

`0XYZ` sont les permis d'accès à la file. On compte les neuf bits les moins significatifs, groupés de trois en trois avec la signification :

– `X` : Permission pour le processus qui exécute l'appel.
– `Y` : Permission pour le groupe de processus auquel appartient le processus qui exécute l'appel.
– `Z` : Permission pour le reste des processus du système.

Dans chaque groupe (`X`, `Y`, ou `Z`) les bits ont la signification suivante : le premier habilite / inhabilite la lecture ou l'écriture. Le troisième n'a pas une signification pour ce cas. Si l'on veut créer une file avec les permissions de lecture et d'écriture pour le processus qui la génère et des permission seulement de lecture pour les autres qui appartient à son groupe, alors `0XYZ = 0640`.

▶ **Exemple 1.** On génère une clé du type `key_t`, avec `ftok`. Le programme génère la file et l'assigne un identificateur.

Listing 7.1: *cle.c*

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main()
{
 key_t cle;
 int idmsg;

 if ((cle = ftok("." , 'e')) == (key_t) -1)
 {
     printf("Erreur de création de clé\n");
     exit (1);
 }

 if ((idmsg = msgget(cle, IPC_CREAT | 0666)) == -1)
```

```
 {
20      printf("Erreur de génération de la   file \n");
        exit (1);
 }
 return 0;
}
```

msgctl()

La fonction `msgctl()` permet de modifier l'information de contrôle que le *kernel* associe à chaque file identifiée par son `idmsg`. Si un erreur se produit, la fonction retournera la valeur `-1`. Le prototype est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl (int msqid, int cmd, struct msqid_ds * buf);
```

Le paramètre `msqid` indique la file à contrôler. Le deuxième paramètre `cmd` permet sélectionner le type d'opération à réaliser sur la file, elle peut prendre les valeurs suivantes :

- `IPC_STAT` retourne l'état de la structure de contrôle associée à la file dans la zone de la mémoire pointée par `buf`
- `IPC_SET` initialise les champs de la structure de contrôle de la file selon le contenu de la zone de mémoire pointée par `buf`
- `IPC_RMID` élimine la file de messages identifiée par `msqid`. La file n'est pas effacée si existe un processus qui l'utilise.

Le paramètre `buf` est un pointeur à une zone de mémoire.

Les objets IPC ne sont pas effacés s'ils ne sont pas éliminés convenablement, ou bien, il faut attendre que le système soit reinitialisé. L'utilisation de `msgctl()` permet d'éliminer une file de messages :

```
if (msgctl(msqid, IPC_RMID, NULL) == -1 )
  {    /* idem mettre 0 que NULL*/
   printf("Erreur à la fermeture de la file\n");
   exit(1);
}
```

`msgsnd()` et `msgrcv()`

L'écriture et la lecture dans une file est réalisée par des unités indivisibles appelées messages qui peuvent être de types différents (pour pouvoir distinguer ces types on inclut un champ `long` dans la structure des messages). Les deux fonctions retournent la valeur `-1` en cas de problème. Leurs prototypes sont :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, int msgsz,
           int attr);
int msgrcv(int msqid, struct msgbuf *msgp, int msgsz,
           long  msgtyp, int attr);
```

Dans les deux cas, `msqid` est l'identificateur de la file. Le paramètre `msgsz` établit la taille du message à lire ou écrire. Le pointeur `msgp` indique la zone de mémoire où se trouvent les données à envoyer ou la place où vont se stocker celles qui seront lues. Le paramètre `msgtyp` seulement apparaît dans l'appel qui permet la lecture de la file. Il spécifie le message qu'on veut lire :

- `msgtyp = 0` accès au premier message qui se trouve dans la file, indépendamment de son type.
- `msgtyp > 0` accès au premier message du type `msgtyp` qui se trouve stocké dans la file.
- `msgtyp < 0` accès au premier message dont le type soit inférieur ou égal à la valeur absolue de `msgtyp`, et en même temps soit le plus petit de tous.

`attr` a une signification en fonction de `msgsnd()` ou `msgrcv()` :

1. écriture (`msgsnd`)
   - `IPC_NOWAIT` : l'appel retourne immédiatement le contrôle et reprend -1. C'est à dire, si la file est pleine et on ne peut pas écrire, on retourne -1 et l'exécution du processus continue.
2. lecture (`msgrcv`)
   - `IPC_NOWAIT` : si il n'y a pas aucun message à lire il retourne `-1` et il continue avec l'exécution.

▶ **Exemple 2.** Schéma `msgsnd()` et `msgrcv()` :

Listing 7.2: *schema.c*

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct message
{
  long tipo;
  int dato_1;
  int dato_2;
};

void main()
{
  key_t llave;
  int idmsg;
  int longitud;
  struct message monmessage;

  // ... initialiser la file (msgget)
  monmessage.tipo = 1;
  monmessage.dato_1 = 24;
  monmessage.dato_2 = 4;
  longitud = sizeof(struct message) - sizeof(long);
  if (msgsnd(idmsg, &monmessage, longitud, 0) == -1)
  {
      printf("Erreur à l'écriture du message\n");
      exit (1);
  }
  if (msgrcv(idmsg, &monmessage, longitud, 1, 0) == -1)
  {
      printf("Erreur à l'écriture du message\n");
      exit (1);
  }
   //... traiter le message et élimination de la file
  return 0;
}
```

7.1.3 Monitorisation

La commande `ipcs` permet de connaître tous les mécanismes de communication IPC entre les processus vivants dans l'ordinateur. Son format est `ipcs [options]`, avec les options :

- Si le champ option est vide, il montrera un résumé de l'information associée aux files de message, sémaphores et mémoire partagée.
- `-u` : On montre l'information détaillée de tous les mécanismes IPC.
- `-q` : On montre l'information relative aux files de messages actives.
- `-m` : On montre l'information relative aux segments de mémoire partagée actifs.
- `-s` : On montre l'information relative aux sémaphores actifs.

```
leibnitz> ipcs
------ Shared Memory Segments --------
key        shmid     owner    perms  bytes    nattch  status
0x00000000 488833024 root    777    1073536  2       dest
0x00000000 488865793 root    644    110592   4       dest
0x00000000 488898562 root    644    110592   2       dest
0x00000000 401113097 langue  777    1000000  0
0x00000000 403243022 langue  777    880000   0

------ Semaphore Arrays --------
key        semid     owner    perms    nsems      status

------ Message Queues --------
key        msqid     owner    perms    used-bytes   messages
```

Dans le cas d'avoir une file de messages active, le système répondra à l'ordre `ipcs -q` comme :

```
leibnitz> ipcs -q
------ Message Queues --------
key         msqid   owner   perms   used-bytes  messages
0xffffffff  385     rebeca  666     0           0
```

Et si l'on a un sémaphore :

```
leibnitz> ipcs -s
------ Semaphore Arrays --------
key        semid   owner      perms      nsems      status
0x00000000 65536   jmtorres   600        1
```

Autre commande utile est `ipcrm`, qui permet d'éliminer un mécanisme de communication et de libérer les ressources qu'il consomme. Son format est : `ipcrm [options]`. Où :

- `-q idmsg` élimine la file de messages avec l'identificateur de file `idmsg`.

- `-Q msgkey` élimine la clé de une file de messages.
- `-m idshm` élimine la zone de mémoire partagée dont l'identificateur est `idshm`.
- `-M shmkey` élimine la clé de mémoire partagée.
- `-s idsem` élimine le sémaphore dont l'identificateur est `idsem`.
- `-S semkey` élimine la clé d'un sémaphore.

Exemple : Si l'on veut éliminer la file de l'exemple précédent —encore existante par n'importe quelle raison— on tape la commande : `ipcrm -m 385`

7.2 Structure de messages

La réussite pour mieux comprendre la communication IPC est dans la compréhension des structures de données qui sont à l'intérieur du noyau et les structures de données que les processus vont échanger.

7.2.1 Buffer de message

Le *buffer* de message est la structure la plus importante. On la déclare en `<sys/msg.h>`

```
#ifdef __USE_GNU
  /*buffer de message pour appels msgsend et msgrcv*/
struct msgbuf
{
  long int mtype; //type de message envoyé : numéro > 0
  char mtext[1];  //texte du message
};
#endif
```

La structure des messages échangés est composée de deux parties : son type et son contenu. Le message doit contenir un champ de type `long` —où on spécifie le type du message— et le champ de données. L'habilité d'associer à un message un type, confère la capacité de multiplexer les messages dans une seule file. On peut assigner à tous les messages la même caractéristique (le type 1 par exemple, et 2 aux autres), et de cette manière, quand on voudra voir seulement ceux avec un type précis avec les appels `msgsnd()` et `msgrcv()`, on demandera ce type souhaité, et les autres seront invisibles. Le champ `mtext` peut contenir non seulement des

chaînes de caractères, mais n'importe quel type de données et dans n'importe quelle forme. On peut ainsi personnaliser les messages. Par exemple :

```
struct mymsg {
  long type;       /* type */

  int n_mensaje;   /* données */
  char indent;
  double x;
  double y;
  time_t temps;
}
```

Le premier élément de la structure indique toujours le type de message et doit être un numéro entier positif. Ce type est utilisé par le processus récepteur pour la sélection de messages.

Une file de message n'assume aucune structure concrète de la partie de données, cependant elle a la limitation de la taille du message à envoyer. En Linux cette taille est défini dans `<sys/msg.h>` comme :

```
#define MSGMAX 4056 //Taille maximale du message en
                    //bytes
```

le message ne peut pas donc être plus grand que 4056 bytes en total, incluant l'élément `mtype` qui a une longueur de 4 bytes (`long`).

7.2.2 `msg` du noyau

Chacun des trois types d'objets IPC a une structure de données interne qui est maintenue par le noyau ; pour les files de message ceci correspond à `msqid_ds`. Le noyau crée, stocke et maintient une structure par chaque file de message. La structure est définie dans `<sys/msg.h>` :

```
// une structure par file de message
struct msqid_ds {
  struct ipc_perm msg_perm;
   ...
   __kernel_time_t msg_stime;    /* last msgsnd time */
   __kernel_time_t msg_rtime;    /* last msgrcv time */
   __kernel_time_t msg_ctime;    /* last change time */
   ...
   unsigned short msg_cbytes;    /* current number of bytes on queue */
   unsigned short msg_qnum;      /* number of messages in queue */
   unsigned short msg_qbytes;    /* max number of bytes on queue */
```

```
    __kernel_ipc_pid_t msg_lspid; /* pid of last msgsnd */
    __kernel_ipc_pid_t msg_lrpid; /* last receive pid */
};
```

- `msg_perm`: Information sur les permissions pour la file de message.
- `msg_stime`: Instant (`time_t`) du dernier message qui a été envoyé.
- `msg_rtime`: Instant du dernier message récupéré de la file.
- `msg_ctime`: Instant du dernier changement fait à la file.
- `msg_cbytes`: Nombre total d'octets qu'il y a dans la file (somme des tailles de tous les messages).
- `msg_qbytes`: Maximum nombre d'octets dans la file.
- `msg_qnum`: Nombre de messages actuellement dans la file.
- `msg_lspid`: Le PID du processus qui a envoyé le dernier message.
- `msg_lrpid`: Le PID du processus qui a récupéré le dernier message.

7.2.3 `ipc_perm` du noyau

Le noyau garde l'information des permissions des objets IPC dans une structure de type `ipc_perm`. Par exemple, dans la structure interne pour la file de message décrite précédemment, l'élément de `msg_perm` est de cet type.

```
struct ipc_perm {
   key_t key;
   ushort uid;   /* euid et egid du propriétaire */
   ushort gid;
   ushort cuid;  /* euid et egid du créateur */
   ushort cgid;
   ushort mode;  /* modes d'accès */
   ushort seq;   /* numéro de séquence du slot */
};
```

Files de messages et sections critiques

Il est possible d'utiliser les files de messages pour contrôler l'accès aux sections critiques, en manipulant un *mutex*. Le programme 7.3 fait appel à cette philosophie.

Listing 7.3: *messages.c*

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/msg.h>

struct mymsgbuf
{
  long type;
  char mtext[1]; // taille du message
};

int main(void)
{
  int x=0;
  int mutex;            // mutex de la file de messages pour
                        // syncrhoniser l'acces a la section critique
  struct mymsgbuf c; // buffer de la file
  mutex = msgget((key_t) "MUTEX", IPC_CREAT | IPC_EXCL | 0777);
  msgsnd(mutex, &c, 1, 0); // necessaire pour entrer en section critique
                           // la premiere fois
  if (fork() == 0)
  { // processus fils
    for(;x<10;)
    {
        msgrcv(mutex, &c , 1, 0, 0); // entrée section critique
        // section critique
        printf("Fils:\n"); x++;
        msgsnd(mutex, &c, 1, 0);     // sortie section critique
    }
    printf(" fils x=%d\n",x);
  }
  else
   { // processus pere
    for(;x<10;)
    {
        msgrcv(mutex, &c , 1, 0, 0); // entrée section critique
        // section critique
        printf("Pere:\n");
        sleep(3); x++;
        msgsnd(mutex, &c, 1, 0);     // sortie section critique
    }
    printf("pere x=%d\n",x);
   }
```

```
  return 0;
}
```

Écrivains et lecteurs

▶ **Exemple 3.** Le programme 7.4 montre l'utilisation de files de messages pour résoudre le problème des écrivains et des lecteurs :

Listing 7.4: *file-w.c*

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSQKEY 34856
#define MSQSIZE 32

struct mymsgbuf
{
  long mtype;
  char mtext[MSQSIZE];
};

int main(void)
{
  key_t key;
  int n, msqid;
  struct mymsgbuf mb;

  // Genère la file de messages sans utiliser ftok
  // ceci a le risque de dupliquer une file avec la même clé

  key = MSQKEY;
  if ((msqid = msgget(key, 0666)) < 0) {
    perror("msgget"); exit (1);
  }

  // Lire de stdin et les envoyer
  // à la file de messages comme appartenant au type 1
  mb.mtype = 1;
  n = read(0, mb.mtext, MSQSIZE);
```

```
    if (msgsnd(msqid, &mb, n, 0) < 0) {
        perror("msgsnd"); exit(1);
    }

    // Envoyer un message de type 2
    mb.mtype = 2;
    n = read(0, mb.mtext, MSQSIZE);
    if (msgsnd(msqid, &mb, n, 0) < 0) {
        perror("msgsnd"); exit(1);
    }
    exit(0);
}
```

Listing 7.5: *file-r.c*

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSQKEY 34856
#define MSQSIZE 32

struct mymsgbuf {
  long mtype;
  char mtext[MSQSIZE];
};

int main(void)
{
  key_t key;
  int n, msqid;
  struct mymsgbuf mb;

  // Crée une nouvelle file de messages
  // on assure qu'elle n'existait pas avec IPC_EXCL
  key = MSQKEY;
  if ((msqid = msgget(key, IPC_CREAT | IPC_EXCL | 0666)) < 0)
  {
      perror("msgget"); exit(1);
  }
  // Recevoir messages. On utilise la même clé que l'autre
  // processus
```

```
    while ((n = msgrcv(msqid, &mb, MSQSIZE, 0, 0)) > 0)
    {
30      switch (mb.mtype)
        {
        case 1:
            write(1, mb.mtext, n);
            break;
        case 2:
            goto out;
        }
    }
    out:
40  // Effacer la file de messages du système
    if (msgctl(msqid, IPC_RMID, (struct msqid_ds *) 0) < 0)
    {
        perror("msgctl"); exit (1);
    }
    exit (0);
}
```

Communication unidirectionnelle entre processus

▶ **Exemple 4.** Cet exemple illustre la communication unidirectionnelle du Capitaine kirk (`kirk.c`) et de M. Spock (`spock.c`) au moyen de files de messages IPC :

Listing 7.6: *kirk.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
10  char mtext[200];
```

```
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }

    printf("Introduzca líneas de texto, ^D para salir:\n");

    buf.mtype = 1;
    while(gets(buf.mtext), !feof(stdin)) {
        if (msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0) == -1)
            perror("msgsnd");
        }

    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(1);
    }

    return 0;
}
```

Listing 7.7: *spock.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) { /* same key as kirk.c */
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue */
        perror("msgget");
        exit(1);
    }

    printf("spock:Listo para recibir mensajes Capitán n");

    for(;;) { /* Spock never quits ! */
        if (msgrcv(msqid, (struct msgbuf *)&buf, sizeof(buf), 0, 0) == -1) {
            perror("msgrcv");
            exit(1);
        }
        printf("spock: \"%s\"\n", buf.mtext);
    }

    return 0;
}
```

7.3 Mémoire partagée

Ces appels système complètent ceux de POSIX pour la projection de fichiers en mémoire (voir Chapitre 9 *Mémoire virtuelle*, Section 9.9 *Services POSIX de projection de fichiers*). L'utilisation de la mémoire partagée nécessite l'entête `<sys/shm.h>`. Les appels système sont :

```
void shmget(key_t key, int taille, int attr);
char *shmat(int identifiant, char *adresse, int attr);
int shmdt(char *adresse);
int shmctl(int identifiant, int commande,
    struct shmid_ds*attributs);
```

`shmget()` permet à partir d'une clé d'obtenir l'identifiant d'un segment mémoire partagée, `shmat()` sert à attacher le segment, `shmdt()` à détacher un segment et `shmctl()` à paramétrer ou supprimer un segment.

Pour effacer un segment de mémoire partagée on utilisera `shmctl()` :

```
semctl(shm, IPC_RMID, NULL);
```

▶ **Exemple 5.** Le programme `shm.c` montre les fonctions pour attacher et partager une zone mémoire :

Listing 7.8: *shm.c*

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/shm.h>

int main ()
{
    int segment_id;
    char *shared_memory;
    struct shmid_ds shmbuffer;
    int segment_size;
    const int shared_segment_size = 0x6400;
    // Allouer mémoire
    segment_id = shmget(IPC_PRIVATE, shared_segment_size,
            IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
    // Attacher mémoire
    shared_memory = (char*) shmat (segment_id, 0, 0);
    printf ("Mémoire attachée à l'adresse %p\n", shared_memory);
```

```
    // Déterminer la taille du segment
    shmctl (segment_id, IPC_STAT, &shmbuffer);
    segment_size = shmbuffer.shm_segsz;
    printf ("Taille du segment : %d\n", segment_size);
    // Écrire une chîne dans le segment partagé
    sprintf(shared_memory, "hello, world.");
    // Détacher le segment
    shmdt(shared_memory);
    // rattacher le segment à une autre adresse
    shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
    printf("Mémoire attachée à l'adresse %p\n", shared_memory);
    printf ("%s\n", shared_memory);
    // Détacher le segment
    shmdt (shared_memory);
    // Deallouer le segment
    shmctl (segment_id, IPC_RMID, 0);
    return 0;
}
```

Exécution de `shm.c` :

```
leibnitz> shm
Mémoire attachée à l'adresse 0x40139000
Taille du segment : 25600
Mémoire attachée à l'adresse 0x5000000
hello, world.
```

▶ **Exemple 6.** Le programme *C++* `exemem.cc` montre l'utilisation des variables partagées. L'obtention de la clé avec `ftok()` est dans ce cas, facultative. Les déclarations et les classes se font dans l'entête `mem.h` :

Listing 7.9: *mem.h*

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

class TMemoria
{
 public:
        TMemoria(int tamanyo);
```

```
        int *Asignar(void);
        // char *Asignar(void );
        void Liberer(int *variable );
        void Liberer(char *variable );
        ~TMemoria();
 private:
        int shmid;
 };

TMemoria::TMemoria(int tamanyo)
{
 key_t llave;
 // llave=ftok (".",' A');
 // if ( llave==-1) { perror("Erreur cle \n"); exit (1); }
 // shmid=shmget(llave,10,IPC_CREAT|0600);
 shmid=shmget(IPC_PRIVATE,tamanyo,IPC_CREAT|0600);
}

int *TMemoria::Asignar(void)
 {
  return ((int *)shmat(shmid,0,0));
 }
/*
char *TMemoria::Asignar(void)
{
 return shmat(shmid,0,0);
}
*/
void TMemoria::Liberer(char *variable)
{
 shmdt(variable);
}

void TMemoria::Liberer(int *variable)
{
  shmdt((char *)variable);
}

TMemoria::~TMemoria()
{
```

```
   shmctl(shmid,IPC_RMID,0);
}
```

Voici un exemple de programme qui utilise `smem.h`. La variable mémoire partagée est `mem`. Le père attends et le fils met `mem = 5`. Puis le père se réveille et `mem = 5` :

Listing 7.10: *exemem.cc*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "mem.h"

int main()
{
 TMemoria mem(sizeof(int));
 int pid;
 int *var;
 int locale=2;

 var=&locale;
 printf("\nvar partagee=%d,var local=%d\n",*var,locale);
 var=mem.Asignar();
 pid=fork();
 if (pid==0)
  {
     var[0]=5;
     locale=5;
     printf("\nFils: var partagée=%d var locale=%d\n",var[0],locale);
     mem.Liberer(var);
     sleep(10);
     exit(0);
   }
  else
   {
     sleep(10);
     printf("\nPère: var partagée=%d var locale=%d\n",var[0],locale);
     exit(0);
   }
 return 0;
```

```
} // main
```

Exécution de `exemem.cc`:

```
leibnitz> g++ -o exemem exemem.cc
leibnitz> exemem
var partagee=2,var local=2
Fils : var partagée=5 var local=5
Père : var partagée=5 var local=2
```

7.4 Sémaphores

Dans System V les sémaphores sont un peu plus complexes que la définition de POSIX. Ici les sémaphores servent à contrôler l'accès à plusieurs exemplaires de la même ressource. Les appels système sont:

```
#include <sem.h>
int semget(key_t key, int nombre, int attr);
int semop(int indentifiant, struct sembuf *operation,
          unsigned nombre);
int semctl(int indentifiant, int numero,
           int commande, union semnum);
```

`semget()` est similaire à `msgget()` et `shmget()`, avec un `nombre` de sémaphores en deuxième argument. Les attributs `attr` peuvent contenir `IPC_CREAT`, `IPC_EXCL` et les permissions.
`semopt()` sert à implanter les opérations P et V (voir Section 6.7 du Chapitre 6 *Synchronisation des processus*) sur des multiples sémaphores. Les opérations sont décrites dans une structure `sembuf` de `<sys/sem.h>`:

- `short int sem_num`: Numéro de sémaphore: 0,1,2,...
- `short int sem_op`: Opération à réaliser.
  - `sembuf.sem_op` = n, avec $n > 0$, alors $V_n()$
  - `sembuf.sem_op` = n, avec $n < 0$, alors $P_n()$
  - `sembuf.sem_op` = 0, le kernel endort le processus jusqu'à ce que le compteur associé au sémaphore soit nul, puis il continue l'exécution.
- `int attr`: Attributs.

Les opérations de Dijkstra pour les sémaphores sont définies donc (la constante symbolique `IPC_UNDO = 0`):

```
int P(int identifiant)
{
   struct sembuf buffer;
   buffer.sem_num = 0;      // buffer = {0, -1, 0}
   buffer.sem_op  = -1;
   buffer.sem_flg = IPC_UNDO;
   return (semop (identifiant, &buffer, 1));
}

int V(int identifiant)
{
   struct sembuf buffer;
   buffer.sem_num = 0;     // buffer = {0, 1, 0}
   buffer.sem_op  = 1;
   buffer.sem_flg = IPC_UNDO;
   return (semop (identifiant, &buffer, 1));
}
```

Dans des opérations réelles on devrait utiliser un seul sémaphore à la fois : `P()` et `V()`.

La fonction `semctl()` modifie le parametrage d'un jeu de sémaphores. En réalité le prototype est :

```
int semctl(int identifiant, int numero, int commande, ...)
```

où les ...indiquent la présence éventuelle d'un argument supplémentaire d'un type non défini. Ainsi, on peut utiliser en dernier argument l'`union`, définie manuellement comme :

```
union semun
{
    int                 value;
    struct semid_ds    *buffer;
    unsigned short int *table;
}
```

En fonction de la `commande`, le `numéro` en deuxième position et l'union en dernière auront des rôles distincts. En particulier nous allons utiliser :

– `commande=SETALL` : Fixer les compteurs de tous les sémaphores avec les valeurs contenus dans `table`. L'argument `numero` est ignoré.

– `commande=IPC_RMID`: Supprimer l'ensemble des sémaphores. Tous les processus en attente sont réveilles. L'argument `numero` et le quatrième argument sont ignorés.

Pour effacer un sémaphore on utilisera `semctl()`:

```
semctl(sem, IPC_RMID, 0);
```

Le schéma d'utilisation des sémaphores est alors:

```
// Création d'un sémaphore
if((sem = semget(cle,1,IPC_CREAT|IPC_EXCL|0600)) == -1)
   {
       perror ("semget");
       exit (1);
   }
 // Initialisation
 table_sem [0]= N;
 semun.table = table_sem;
 if (semctl (sem, 0, SETALL, semun) < 0)
      perror("semctl");

 // Création des processus
 ... fork()...

 // Point de rendez-vous
 // Opération P()
 sembuf.sem_num = 0;
 sembuf.sem_op  = -1;
 sembuf.sem_flg = 0;
 semop (sem, & sembuf, 1);

 // Attente de N processus
 sembuf.sem_op  = 0;
 semop (sem, & sembuf, 1);

 // Opération V()
 sembuf.sem_num = 0;
 sembuf.sem_op = 1;
 sembuf.sem_flg = 0;
 semop (sem, & sembuf, 1);
 ...
```

▶ **Exemple 7.** Le programme suivant reprend l'exemple de la page 180, (Chapitre 6) où on avait écrit le programme `posix-sem.c` pour des séma-

phores POSIX. Maintenant nous allons utiliser la compilation conditionnelle :

```
#define SYSV_SEMAPHORE
```

pour montrer l'utilisation des sémaphores System V. Observez les différences avec POSIX, qui sont bien documentées.

Listing 7.11: *posix-v-sem.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>

/* # define  POSIX_SEMAPHORE */
#define SYSV_SEMAPHORE

#ifdef  POSIX_SEMAPHORE
  #include <semaphore.h> sem_t my_sem;
  #define SEMINITVALUE 1
#endif

#ifdef  SYSV_SEMAPHORE
  #include <sys/ipc.h>
  #include <sys/sem.h>
  int  my_sem, semop_ret;
  #define PERMS S_IRUSR | S_IWUSR
  #define SEM_SET_SIZE 1
  struct  sembuf semwait  = { 0, -1, 0 };    // P
  struct  sembuf semsignal  = { 0, 1, 0 };   // V
#endif

void * critical_section (void *);
#define MAXTHREADS 100

int  main(int argc, char *argv[])
 {
```

```
    int n, i, error;
    pthread_t threadid[MAXTHREADS];
    /* check the number of arguments */
    if ((argc != 2) || ((n = atoi(argv[1])) <= 0))
    {
        printf("Usage: %s number_of_threads\n", argv[0]);
        exit(1);
    }
    // initialiser les sémaphores
#ifdef POSIX_SEMAPHORE
    sem_init(&my_sem, 0, SEMINITVALUE);
#endif

#ifdef SYSV_SEMAPHORE
    my_sem = semget(IPC_PRIVATE, SEM_SET_SIZE, PERMS);
    // initialiser les semaphores à 1
    semop(my_sem, &semsignal, 1);
#endif

    // creation des threads
    for(i = 0; i < n; ++i)
        pthread_create(&threadid[i], NULL,critical_section,
                       (void *)i );

    // attendre les threads
    for(i = 0; i < n; ++i)
        pthread_join(threadid[i], NULL);

 void * critical_section (void *arg)
 {
    int myindex, i, n;
    // extraire index du thread
    myindex = (int)arg;

#ifdef POSIX_SEMAPHORE
    sem_wait(&my_sem);
#endif

#ifdef SYSV_SEMAPHORE
    while ((semop_ret=semop(my_sem, &semwait, 1)==-1) &&
```

```
    (errno == EINTR));
    if (semop_ret == -1)
    {
        printf("Semaphore decrement failed: %s\n",
        perror(errno));
    }
#endif

   // section critique
    for (i=1; i < 5; i++)
    {
        printf("%d : %d\n", myindex, i);
        if ((n=sleep (1)) != 0)    // dormir 1 seconde
        printf("interrupted, no of secs left %d\n",n);
    }
    // quitter
#ifdef POSIX_SEMAPHORE
    sem_post(&my_sem);
#endif

#ifdef SYSV_SEMAPHORE
    while ((semop_ret=semop(my_sem, &semsignal, 1)==-1) &&
    (errno == EINTR));
    if (semop_ret == -1)
    {
        printf("erreur dans incrément du sémaphore :%s\n",
        perror(errno));
    }
#endif
    pthread_exit(NULL);
}
```

▶ **Exemple 8.** Lecteurs et écrivains pris de [Bla02]. Comme d'habitude, ni `ecrivain.c` ni `lecteur.c` testent les conditions d'erreur suite à la manipulation de mémoire partagée ou des sémaphores, mais il faut souligner que dans les communications interprocessus *il faut toujours* tester ces conditions avant de continuer avec la gestion de ces structures.

Listing 7.12: *ecrivain.c*

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#define LG_CHAINE 256

typedef union semun {
        int                     val;
        struct    semid_ds * buffer;
        unsigned short int * table;
    } semun_t;

int  main (int argc, char * argv []);
{
    key_t           key;
    int             sem;
    int             shm;
    struct  sembuf sembuf;
    semun_t         u_semun;
    char *          chaine = NULL;
    unsigned short table [1];

    if  (argc  != 2) {
        fprintf  (stderr, "Syntaxe : %s  fichier_clé  \n", argv  [0]);
        exit  (1);
    }
    if  ((key = ftok  (argv  [1], 0)) == -1) {
        perror ("ftok" );  exit  (1);
    }
    if ((shm = shmget (key, LG_CHAINE, IPC_CREAT | 0600))==-1)
    {
        perror ("shmget"); exit  (1);
    }
    if  ((chaine = shmat (shm, NULL, 0)) == NULL) {
        perror ("shmat"); exit  (1);
    }
    if ((sem = semget (key , 1, 0)) == -1) {
      if ((sem = semget(key, 1, IPC_CREAT | IPC_EXCL | 0600))==-1)
      {
```

```
            perror ("semget"); exit (1);
        }
        chaine [0] = '\0';
        table [0] = 1;
        u_semun . table = table;
        if (semctl (sem, 0, SETALL, u_semun) < 0)
            perror ("semctl");
    }
    sembuf . sem_num = 0;
    sembuf . sem_op = -1;
    sembuf . sem_flg = SEM_UNDO;
    if (semop (sem, & sembuf, 1) < 0)
    {
        perror ("semop"); exit (1);
    }
    fprintf (stdout, "> ");
    fgets (chaine, LG_CHAINE, stdin);

    sembuf . sem_op = 1;
    if (semop (sem, & sembuf, 1) < 0) {
        perror ("semop"); exit (1);
    }
    return (0);
}
```

Listing 7.13: *lecteur.c*

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

int main (int argc, char * argv [])
{
    key_t     key;
    int       sem;
    int       shm;
    struct sembuf sembuf;
    char    *chaine = NULL;
```

```
    if (argc != 2) {
        fprintf (stderr, "Syntaxe: %s fichier_clé \n", argv [0]);
        exit (1);
    }
// Clé
    if ((key = ftok (argv [1], 0)) == -1) {
        perror ("ftok"); exit (1);
    }
// Mémoire partagée et semaphore
    if (((sem = semget (key, 1, 0)) == -1)
      || ((shm = shmget (key, 0, 0)) == -1)) {
        perror ("semget/shmget"); exit (1);
    }
    if ((chaine = shmat (shm, NULL, SHM_RDONLY)) == NULL)
    {
        perror ("shmat"); exit (1);
    }
// P
    sembuf.sem_num = 0;
    sembuf.sem_op = -1;
    sembuf.sem_flg = 0;
    if (semop (sem, &sembuf, 1) < 0) {
        perror ("semop"); exit (1);
    }
// Section critique
    fprintf (stdout, "%s\n", chaine);
// V
    sembuf.sem_op = 1;
    if (semop (sem, &sembuf, 1) < 0)
    {
        perror ("semop"); exit (1);
    }
    return (0);
}
```

7.5 Exercices

1. Utiliser la commande `ipcs` pour vérifier les mécanismes IPC qui sont actifs dans le système.
2. Localiser le champ KEY et identifiant des sémaphores.
3. Écrire un programme *C* qui créera deux sémaphores sous un seul identifiant. Les initialiser avec des valeurs différentes. Ces sémaphores permettront des accès en lecture et écriture uniquement pour le propriétaire. Exécuter le programme et vérifier avec `ipcs` quelle est la clé du groupe de sémaphores (`key`) et quel est l'identifiant (`semid`).
4. Écrire un autre programma en *C* pour retourner la valeur d'un sémaphore. La valeur de l'identifiant et le numéro du sémaphore doivent être introduits depuis la ligne de commande.
5. Écrire un autre programme qui fera une opération d'incrément ou decrément de n unités de la valeur d'un sémaphore. La valeur de l'identifiant et le numéro du sémaphore doivent être introduits depuis la ligne de commande. Utiliser ce programme pour changer les valeurs des sémaphores précédents. Vérifier comment le shell se bloque quand on tente de decrémenter un sémaphore dont la valeur est 0. Débloquer la fenêtre du shell précèdent en incrémentant la valeur du sémaphore depuis un autre shell.
6. Modifier le programme de `kirk.c` et `spock.c` (page 233) pour rendre la communication bi-directionnelle entre les deux processus.
7. Écrire une solution au problème des N philosophes en utilisant les sémaphores IPC pour synchroniser les processus.
8. Écrire deux processus *independants* pour partager una zone de mémoire commune de taille 256 octets. Le premier processus va écrire dans la zone commune le message "Bonjour ! Ici le message !" et restera en attente. Utilisez la commande `ipcs` pour vérifier la mémoire partagée. L'autre processus devra lire la zone commune, et puis afficher le message lu. Utilisez la fonction `ftok()` pour la génération des clés.
9. Exécuter le programma suivant. Le processus père montre à l'écran des lignes avec caractères 10 '+' et le fils des lignes avec 10 caractères 'o'. À cause d'un manque de synchronisation entre les processus, les sorties son mélangées. Utiliser des sémaphores pour résoudre le problème de façon à ce que le fils ne rentre pas en section critique tant que le père y soit, et viceversa. Les sections critiques doivent s'exécuter de façon alternée. Suggestion : utiliser deux sémaphores, un pour

le père et un autre pour le fils.

```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
  int i=0, j, pid; // processus fils

  pid=fork(); // Tester erreur en fork()

  if (pid==-1)
     {
       perror("fork");
       exit(-1);
     }
  if (pid==0)
     { // fils
       while(i<3)
       { // Section critique
         for (j=0;j<10;j++)
         {
           write(1,"o",1);
           sleep(1);
         } // Fin de la section critique
         i++;
       }
    }
  else
    { // Père
     while (i<3)
     { // Section critique
         for (j=0;j<10;j++)
         {
           write(1,"+",1);
           sleep(1);
         } // Fin de la section critique
         i++;
     }
    }
} // main
```

8
Interblocages

8.1 Les processus et les ressources

L'exécution d'un processus nécessite un ensemble de ressources (mémoire principale, disques, fichiers, périphériques, etc.) qui lui sont attribuées par le système d'exploitation. L'utilisation d'une ressource passe par les étapes suivantes :

- **Demande de la ressource :** Si l'on ne peut pas satisfaire la demande il faut attendre. La demande sera mise dans une table d'attente des ressources.
- **Utilisation de la ressource :** Le processus peut utiliser la ressource.
- **Libération de la ressource :** Le processus libère la ressource demandée et allouée.

Lorsqu'un processus demande un accès exclusif à une ressource déjà allouée à un autre processus, le système d'exploitation décide de le mettre en attente jusqu'à ce que la ressource demandée devienne disponible ou lui retourner un message indiquant que la ressource n'est pas disponible : réessayer plus tard.

Les ressources peuvent être de plusieurs types :

- **Reutilisables ou disponibles** Existent-elles après son utilisation ?
  - Reutilisables :
    - Toutes les ressources physiques.
    - Quelques uns logiques (fichiers, mutex, verrous, etc.).
  - Disponibles :
    - Un processus engendre une ressource et un autre l'utilise.

  - Ressources associées à la communication et à la synchronisation comme les messages, les signaux, les sémaphores, etc.
- **D'usage partagé :** Est-ce que la ressource peut être utilisée par plusieurs processus en même temps ? Les ressources partagées n'affectent pas les interblocages.
- **Avec un ou multiples exemplaires :** Existent-ils des multiples exemplaires d'une même ressource ?
- **Preémptible ou non preémptible :** Est-ce qu'on a le droit de retirer une ressource quand elle est utilisée par un processus ?

La différence principale entre ressources préemptibles et non préemptibles c'est que les premières peuvent être retirées sans risque au processus qui les détient, tandis que les deuxièmes ne peuvent être retirée sans provoquer des problèmes. Comme exemples de ressources preémptibles nous avons le processeur (où le changement de contexte correspond à l'expropriation et l'état du processeur est copié à la **Table de Contrôle de Processus** (**BCP**)) et la mémoire virtuelle (le remplacement est une expropriation et le contenu de la page doit être copié au *swap*). Sont des exemples de ressources non preémptibles, les imprimantes et les scanners. Pour étudier le problème des interblocages, nous allons considérer uniquement les ressources non préemptibles.

8.2 Définition d'un interblocage

Des problèmes peuvent survenir, lorsque les processus obtiennent des accès exclusifs aux ressources. Par exemple, un processus P_1 détient une ressource R_1 et attend une autre ressource R_2 qui est utilisée par un autre processus P_2 ; le processus P_2 détient la ressource R_2 et attend la ressource R_1. On a une situation d'**interblocage** (*deadlock* en anglais) car P_1 attend P_2 et P_2 attend P_1. Les deux processus vont attendre indéfiniment comme montré sur la figure 8.1.

En général, un ensemble de processus est en interblocage si chaque processus attend la libération d'une ressource qui est allouée à un autre processus de l'ensemble. Comme tous les processus sont en attente, aucun ne pourra s'exécuter et donc libérer les ressources demandées par les autres. Ils attendront tous indéfiniment.

▶ **Exemple 1.** Interblocages.

Accès aux périphériques. Supposons que deux processus **A** et **B** veulent imprimer, en utilisant la même imprimante, un fichier stocké sur une

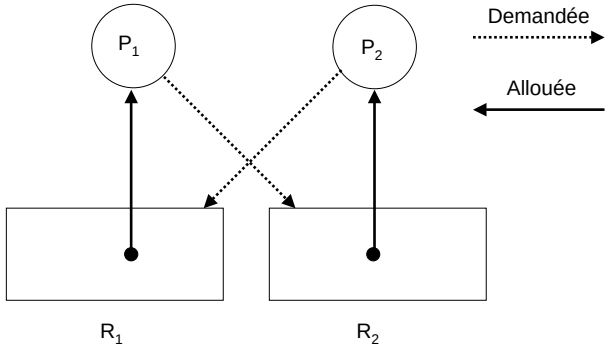


FIG. 8.1 – *Situation d'interblocage de deux processus.*

bande magnétique. La taille de ce fichier est supérieure à la capacité du disque. Chaque processus a besoin d'un accès exclusif au dérouleur et à l'imprimante simultanément. On a une situation d'interblocage si :

- Le processus **A** utilise l'imprimante et demande l'accès au dérouleur.
- Le processus **B** détient le dérouleur de bande et demande l'imprimante.

Accès à une base de données. Supposons deux processus **A** et **B** qui demandent des accès exclusifs aux enregistrements d'une base de données. On arrive à une situation d'interblocage si :

- Le processus **A** a verrouillé l'enregistrement R_1 et demande l'accès à l'enregistrement R_2.
- Le processus **B** a verrouillé l'enregistrement R_2 et demande l'accès à l'enregistrement R_1.

Circulation routière. Considérons deux routes à double sens qui se croisent comme dans la figure 8.2, où la circulation est impossible. Un problème d'interblocage y est présent.

La résolution des interblocages constitue un point théorique très étudié. Mais, pour l'instant, il n'y a pas de solution complètement satisfaisante et la plupart des systèmes —notamment Unix— ne cherchent 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. D'un autre côté, certaines tendances font l'étude des interblocages un domaine très important : plus la technologie progresse, plus il y a

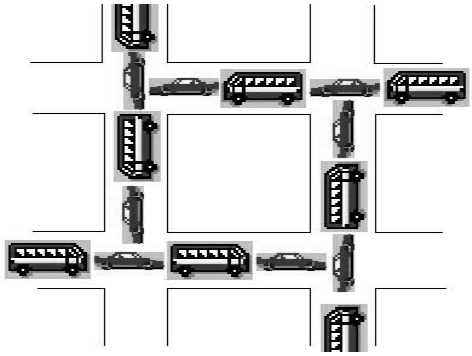


FIG. 8.2 – *Problème de circulation routière.*

de ressources, et un nombre grandissant de processus cohabitent dans un système.

8.2.1 Conditions nécessaires pour l'interblocage

Pour qu'une situation d'interblocage ait lieu, les quatre conditions suivantes doivent être remplies (Conditions de Coffman) :

- **L'exclusion mutuelle.** A un instant précis, une ressource est allouée à un seul processus.
- **La détention et l'attente.** Les processus qui détiennent des ressources peuvent en demander d'autres.
- **Pas de préemption.** Les ressources allouées à un processus sont libérées uniquement par le processus.
- **L'attente circulaire.** Il existe une chaîne de deux ou plus processus de telle maniére que chaque processus dans la chaîne requiert une ressource allouée au processus suivant dans la chaîne.

Par exemple, dans le problème de circulation de la figure 8.2 le trafic est impossible. On observe que les quatre conditions d'interblocage sont bien remplies :

- **Exclusion mutuelle** : Seulement une voiture occupe un endroit particulier de la route à un instant donné.
- **Détention et attente** : Aucun voiture ne peut faire marche arrière.
- **Pas de préemption** : On ne permet pas à une voiture de pousser une autre voiture en dehors de la route.

– **Attente circulaire** : Chaque coin de la rue contient des voitures dont leur mouvement depend des voitures qui bloquent la prochaine intersection.

8.3 Graphe d'allocation des ressources

Le **graphe d'allocation des ressources** est un graphe biparti composé de deux types de nœuds et d'un ensemble d'arcs :

– Les **processus** qui sont représentés par des cercles.
– Les **ressources** qui sont représentées par des rectangles. Chaque rectangle contient autant de points qu'il y a d'exemplaires de la ressource représentée.
– Un arc orienté d'une ressource vers un processus signifie que la ressource est allouée au processus.
– Un arc orienté d'un processus vers une ressource signifie que le processus est bloqué en attente de la ressource.

Ce graphe indique pour chaque processus les ressources qu'il détient ainsi que celles qu'il demande.

▶ **Exemple 2.** Soient trois processus **A**, **B** et **C** qui utilisent trois ressources **R**, **S** et **T** comme illustré à la figure 8.3 :

A	B	C
Demande R	Demande S	Demande T
Demande S	Demande T	Demande R
Libère R	Libère S	Libère T
Libère S	Libère T	Libère R

FIG. 8.3 – *Besoins de trois processus.*

Si les processus sont exécutés de façon séquentielle : **A** suivi de **B** suivi **C**, il n'y pas d'interblocage. Supposons maintenant que l'exécution des processus est gérée par un ordonnanceur du type circulaire. Si les instructions sont exécutées dans l'ordre :

1. **A** demande **R**
2. **B** demande **S**
3. **C** demande **T**

4. **A** demande **S**
5. **B** demande **T**
6. **C** demande **R**

on atteint une situation d'interblocage, fait qui est montré sur la figure 8.4.

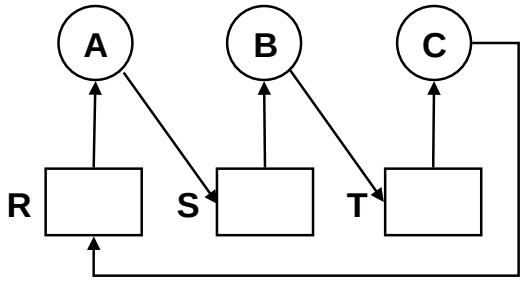


FIG. 8.4 – *Situation d'interblocage de trois processus.*

8.3.1 Réduction du graphe d'allocation des ressources

Un graphe réduit peut-être utilisé pour déterminer s'il existe ou non un interblocage. Pour la **réduction d'un graphe d'allocation des ressources**, les flèches associées à chaque processus et à chaque ressource doivent être vérifiées.

- Si une ressource possède seulement des flèches qui sortent (il n'y a pas des requêtes), on les efface.
- Si un processus possède seulement des flèches qui pointent vers lui, on les efface.
- Si une ressource a des flèches qui sortent, mais pour chaque flèche de requête il y a une ressource disponible dans le bloc de ressources où la flèche pointe, il faut les effacer.

▶ **Exemple 3.** (Pris de [Har02]) Considérez quatre processus $P_1, \cdots, P_4$ qui utilisent des ressources du type R_1, R_2 et R_3. Le tableau 8.1 montre l'allocation courante et le nombre maximaux d'unités de ressources nécessaires pour l'exécution des processus. Le nombre de ressources disponibles est $A = [0,0,0]$. Le graphe d'allocation des ressources pour l'état courant est montré sur la figure 8.5.

Processus	R_1	R_2	R_3	R_1	R_2	R_3
P_1	3	0	0	0	0	0
P_2	1	1	0	1	0	0
P_3	0	2	0	1	0	1
P_4	1	0	1	0	2	0

TAB. 8.1 – *Besoins de quatre processus.*

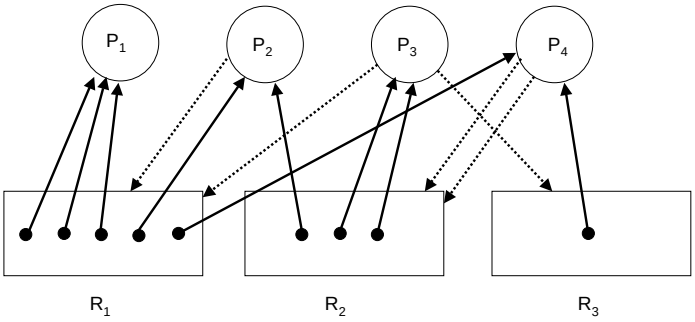


FIG. 8.5 – *Graphe d'allocation des ressources.*

P_1 possède seulement des flèches qui pointent vers lui, alors on les efface. Les requêtes de R_1 effectuées par P_2 peuvent donc être allouées, et P_2 aura seulement des flèches qui pointent vers lui. On les efface aussi. On efface la flèche de R_1 vers P_3 (qui a changé de direction) car R_1 n'a que des flèches sortant. Le graphe réduit final est montré sur la figure 8.6.

8.4 Traitement des interblocages

Comme nous l'avons mentionné, les situations d'interblocage peuvent se produire dans un système. La question qui se pose est donc : doit-il prendre en compte ce problème ou l'ignorer ?

Ignorer complètement les problèmes. On peut faire l'autruche et ignorer les possibilités d'interblocages. Cette « stratégie » est celle de la plupart des systèmes d'exploitation courants car le prix à payer pour les éviter est élevé.

Les détecter et y remédier. On tente de traiter les interblocages, en détectant les processus interbloqués et en les éliminant.

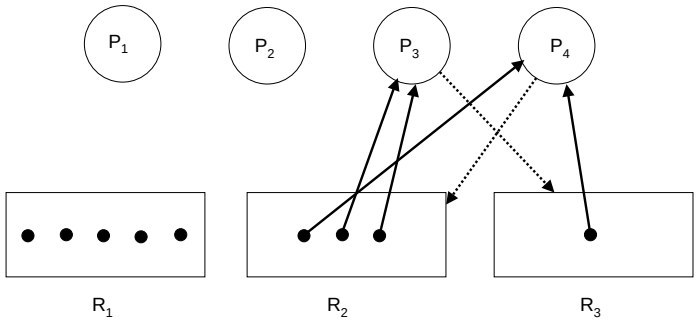


FIG. 8.6 – *Graphe réduit d'allocation.*

Les éviter. En allouant dynamiquement les ressources avec précaution. Le système d'exploitation peut suspendre le processus qui demande une allocation de ressource s'il constate que cette allocation peut conduire à un interblocage. Il lui attribuera la ressource lorsqu'il n'y aura plus de risque.

Les prévenir. En empêchant l'apparition de l'une des quatre conditions de leur existence.

8.5 La détection et la reprise

Dans ce cas, le système ne cherche pas à empêcher les interblocages. Il tente de les détecter et d'y remédier. Pour détecter les interblocages, il construit dynamiquement le graphe d'allocation des ressources du système qui indique les attributions et les demandes de ressources. Dans le cas des ressources à exemplaire unique, il existe un interblocage si le graphe contient au moins un cycle. Dans le cas des ressources à exemplaires multiples, il existe un interblocage si le graphe contient au moins un cycle terminal (aucun arc ne permet de le quitter).

Le système vérifie s'il y a des interblocages :

- A chaque modification du graphe suite à une demande d'une ressource (coûteuse en termes de temps processeur).
- Périodiquement ou lorsque l'utilisation du processeur est inférieure à un certain seuil (la détection peut être tardive).

8.5.1 Algorithme de détection des interblocages

Soient n le nombre de processus $P[1]$,$P[2]$,...,$P[n]$ et m le nombre de types de ressources $E[1]$,$E[2]$,...,$E[m]$ qui existent dans un système. L'**algorithme de détection des interblocages** utilise les matrices et vecteurs suivants :

- Matrice C des allocations courantes d'ordre (n x m). L'élément $C[i,j]$ désigne le nombre de ressources de type $E[j]$ détenues par le processus $P[i]$.
- Matrice R des demandes de ressources d'ordre (n x m). L'élément $R[i,j]$ est le nombre de ressources de type $E[j]$ qui manquent au processus $P[i]$ pour pouvoir poursuivre son exécution.
- Vecteur A d'ordre m. L'élément $A[j]$ est le nombre de ressources de type $E[j]$ disponibles (non attribuées).
- Vecteur E d'ordre m. L'élément $E[j]$ est le nombre total de ressources de type j existantes dans le système.

1. Rechercher un processus $P[i]$ non marqué dont la rangée $R[i]$ est inférieure à A
2. Si ce processus existe, ajouter la rangée $C[i]$ à A, marquer le processus et revenir à l'étape 1
3. Si ce processus n'existe pas, les processus non marqués sont en interblocage. L'algorithme se termine.

On peut démontrer mathématiquement que s'il existe au moins une séquence sûre, alors il existe un nombre infini de séquences sûres. Dans les états sûrs, le système d'exploitation possède le contrôle sur les processus. Dans un état non sûr le contrôle dépends du comportement des processus.

▶ **Exemple 4.** (Pris de [Tan01]). Considérons un système où nous avons trois processus en cours et qui dispose de 4 types de ressources : 4 dérouleurs de bandes, 2 scanners, 3 imprimantes et un lecteur de CD ROM. Les ressources détenues et demandées par les processus sont indiquées par les matrices C et R.

$E = [4,2,3,1]$; $A = [2,1,0,0]$

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} ; R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Seul le processus P3 peut obtenir les ressources dont il a besoin. Il s'exécute jusqu'à la fin puis libère ses ressources, ce qui donne : $A = [2,2,2,0]$. Ensuite P2 peut obtenir les ressources dont il a besoin. Il s'exécute jusqu'à la fin et libère ses ressources, ce qui donne : $A = [4,2,2,1]$. Enfin, P1 peut obtenir les ressources dont il a besoin. Il n'y a pas d'interblocage.

▶ **Exemple 5.** Considérons un système ayant sept processus, A à G, et six ressources R à W. L'attribution des ressources est la suivante :

- A détient R et demande S ;
- B demande T ;
- C demande S ;
- D détient U et demande S et T ;
- E détient T et demande V ;
- F détient W et demande S ;
- G détient V et demande U.

Construire le graphe d'allocation des ressources ? Y a-t-il un interblocage ? Si oui, quels sont les processus concernés ?

▶ **Exemple 6.**

Considérons un système ayant quatre processus, A, B. C et D, et trois types ressources R, S et T, à plusieurs exemplaires, avec 3R, 2S et 2T. L'attribution des ressources est la suivante :

- A détient une ressource de type R et demande une ressource de type S
- B détient 2 ressources de type S et demande une ressource de type R et une ressource de type T
- C détient 1 ressource de type R et demande une ressource de type S
- D détient 2 ressources de type T et demande une ressource de type R

Construire le graphe d'allocation des ressources. Y a-t-il un interblocage ? Si oui, quels sont les processus concernés ?

8.5.2 La reprise des interblocages

Lorsque le système détecte un interblocage, il doit le supprimer, ce qui se traduit généralement par la réalisation de l'une des opérations suivantes :

- Retirer temporairement une ressource à un processus pour l'attribuer à un autre.

– Restaurer un état antérieur (retour en arrière) et éviter de retomber dans la même situation.
– Supprimer un ou plusieurs processus.

8.6 L'évitement des interblocages

Dans ce cas, lorsqu'un processus demande une ressource, le système doit déterminer si l'attribution de la ressource est sûre. Si c'est le cas, il lui attribue la ressource. Sinon, la ressource n'est pas accordée. Un état est sûr si tous les processus peuvent terminer leur exécution (il existe une séquence d'allocations de ressources qui permet à tous les processus de se terminer). Un état est non sûr si on ne peut garantir que les processus pourraient terminer leurs exécutions.

Mais comment déterminer si un état est sûr ou non sûr ? Djikstra a proposé en 1965 un algorithme d'ordonnancement, appelé l'**Algorithme du banquier** qui permet de répondre à cette question. Cet algorithme, utilise les mêmes informations que celles de l'algorithme de détection précédent : matrices A,E,C et R. Un état est caractérisé par ces quatre tableaux.

8.6.1 Algorithme du banquier

Cet algorithme (Dijistra, 1965[1], Habermann 1969[2]) consiste à examiner chaque nouvelle requête pour voir si elle conduit à un état sûr. Si c'est le cas, la ressource est allouée, sinon la requête est mise en attente. L'algorithme détermine donc si un état est ou non sûr :

1. Trouver un processus P_i non marqué dont la rangée i de R est inférieure à A.
2. Si un tel processus n'existe pas alors l'état est non sûr (il y a interblocage). L'algorithme se termine.
3. Sinon, ajouter la rangée i de C à A, et marquer le processus.
4. Si tous les processus sont marqués alors l'état est sûr et l'algorithme se termine, sinon aller à l'étape 1.

L'algorithme du banquier permet bien d'éviter les interblocages mais il est peu utilisé en pratique car on ne connaît pas toujours à l'avance les besoins en ressources des processus.

1. Dijkstra, E.W., 1965. Solution of a Problem in Concurrent Programming Control. Communications of the ACM, 8(9):569, 1965.
2. Habermann, A.N. Prevention of System Deadlocks. CACM, Vol. 12, No. 7, July 1969, pp. 373-385.

▶ **Exemple 7.** On considère quatre processus $P_1, \cdots, P_4$ qui utilisent des ressources du type R_1, R_2 et R_3. Le tableau 8.2 montre l'allocation courante et le nombre maximaux d'unités de ressources nécessaires pour l'exécution des processus. Le nombre de ressources disponibles est $A = [0,0,0]$

1. a) Construire le graphe d'allocation des ressources pour l'état courant.
2. b) L'état courant est-il sûr ?
3. c) Correspond-il à un interblocage ?

Processus	R_1	R_2	R_3	R_1	R_2	R_3
P_1	2	0	0	1	1	0
P_2	3	1	0	0	0	0
P_3	1	3	0	0	0	1
P_4	0	1	1	0	1	0

TAB. 8.2 – *Besoins de quatre processus.*

8.7 La prévention des interblocages

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

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

La Détention et l'attente. Pour ce qui concerne la deuxième condition, elle pourrait être évitée si les processus demandaient leurs 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.

Pas de préemption. 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.

L'attente circulaire. 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 ou en accordant aux processus une seule ressource à la fois (s'il a besoin d'une autre ressource, il doit libérer la première). Par exemple :

- *F(CD-ROM)=1*
- *F(imprimante)=2*
- *F(plotter)=3*
- *F(rubban)=4*

Ainsi on garantit qu'il n'aura pas de cycles dans le graphe des ressources. On peut exiger seulement que aucun processus ne demande une ressource dont le numéro est inférieur aux ressources déjà allouées. Mais ceci n'est pas non plus la panacée, car, en général, le nombre potentiel de ressources est si important qu'il est très difficile de trouver la bonne fonction F pour les numéroter.

8.8 Exercices

1. Citez deux exemples de ressources réutilisables et non-réutilisables. Par quelles entités du système sont gérées les ressources réutilisables et non-réutilisables?
2. Donnez un exemple de programmation qui pourrait mener à l'interblocage entre un groupe de processus.
3. Expliquez brièvement les quatre approches que le SE pourrait utiliser pour manipuler des situations d'interblocage.
4. Étant donné le graphe d'allocation des ressources sur la figure 8.7, complétez le tableau d'allocation des ressources 8.3 et déterminez le vecteur $A = (\bullet, \bullet, \bullet)$.

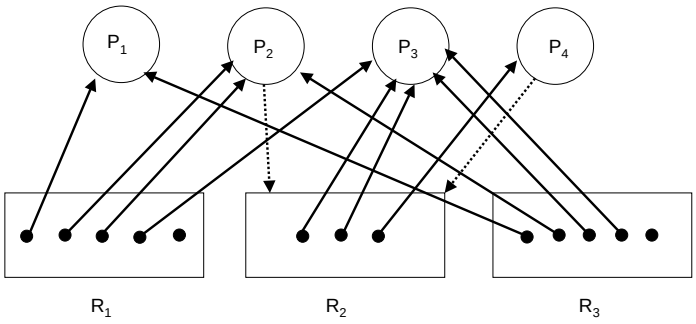


FIG. 8.7 – *Graphe d'allocation des ressources.*

Processus	R_1	R_2	R_3	R_1	R_2	R_3
P_1						
P_2						
P_3						
P_4						

TAB. 8.3 – *Table de processus.*

5. Considérez les tables 8.4, 8.5, et 8.6 qui indiquent les ressources disponibles d'un système, les réclamations maximales faites par des processus, et l'état d'allocation courant de ressources aux processus. Appliquez l'algorithme du banquier pour déterminer si l'état courant d'allocation est sûr.
6. Considérez un système composé de 4 processus (i.e. P_0, P_1, P_2, P_3), et de 3 types de ressources sériellement réutilisables (i.e. R_0, R_1, R_2).

Type de ressources (R_j)	Unités disponibles (C_j)
R_0	5
R_1	6
R_2	9
R_3	4

TAB. 8.4 – *Ressources disponibles.*

Processus (P_i)	R_0	R_1	R_2	R_3
P_0	3	1	3	2
P_1	0	4	5	1
P_2	2	0	2	3
P_3	2	3	4	1
P_4	1	2	3	0

TAB. 8.5 – *Réclamations maximales.*

Le vecteur de ressources disponibles est $A = [3,2,2]$. Les conditions d'utilisation de ressources sont les suivantes :

- Processus P_0 détient 1 unité de R_0, et requiert 1 unité de R_1.
- Processus P_1 détient 2 unités de R_1, et requiert 1 unité de R_0 et R_2.
- Processus P_2 détient 1 unité de R_0, et requiert 1 unité de R_1.
- Processus P_3 détient 2 unité de R_2, et requiert 1 unité de R_0.

Montrez le graphe d'allocation des ressources pour le système décrit ci-haut. Réduisez le graphe et indiquez s'il y a des processus en état d'interblocage.

7. Considérez un système composé de 4 processus (i.e. P_0, P_1, P_2, P_3), et de 3 types de ressources non-réutilisables (i.e. R_0, R_1, R_2). 1 unité chacune de R_0 et de R_2 est disponible. Les conditions d'utilisation de ressources sont les suivantes :
   - Processus P_0 requiert 1 unité de R_0 et 1 unité de R_2.
   - Processus P_1 produit R_0 et R_2, et requiert 1 unité de R_1.
   - Processus P_2 requiert 1 unité chacune de R_0 et de R_2.
   - Processus P_3 produit R_1, et requiert 1 unité de R_2.

   Montrez le graphe d'allocation des ressources pour le système décrit ci-haut. Réduisez le graphe et indiquez s'il y a des processus en état d'interblocage.

Processus (P_i)	R_0	R_1	R_2	R_3
P_0	2	0	1	1
P_1	0	2	3	0
P_2	1	0	2	1
P_3	0	2	1	1
P_4	1	1	2	0
$\sum$	4	5	9	3

TAB. 8.6 – *État d'allocation courant.*

8. Étant donnée les tableaux 8.7, 8.8 et le vecteur $A = (2,1)$, le système est dans un état sûr ou non sûr?

Processus	R_1	R_2
P_0	7	2
P_1	1	3
P_2	1	1
P_3	3	0

TAB. 8.7 – *État courant d'allocation.*

Processus	R_1	R_2
P_0	9	5
P_1	2	6
P_2	2	2
P_3	5	0

TAB. 8.8 – *État des demandes d'allocation.*

9. Étant donné le graphe d'allocation des ressources de la figure 8.8.
   Il y a une réponse correcte parmi les 5 suivantes. Choisissez et justifiez :
   (a) Le graphe a un cycle et donc on peut assurer qu'il n'y a pas d'interblocage.
   (b) Le graphe a un un cycle et donc on peut assurer qu'il y a un interblocage.
   (c) Il y a une séquence de terminaison des processus qui ne produit pas d'interblocage.
   (d) Il y a un interblocage entre les processus P_1, P_2, P_5 P_6.

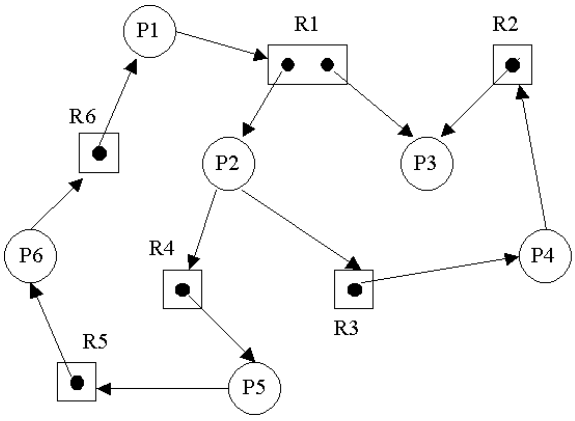


FIG. 8.8 – *Graphe d'allocation des ressources.*

(e) Aucune des réponses antérieures n'est correcte.

9
Gestion de la mémoire

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. La nécessité de gérer la mémoire 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. Nous verrons qu'il y a plusieurs schémas pour la gestion de la mémoire, avec leurs avantages et inconvénients.

9.1 Introduction

9.1.1 Rappels sur la compilation

La production d'un programme passe par plusieurs phases, comme illustré à la figure 9.1 :

- Écriture en langage source (C / C++, Pascal,...).
- Compilation en module objet (langage machine).
  - Peuvent inclure des bibliothèques.
  - Appel aux bibliothèques : point de branchement.
- Assemblés en image binaire.
  - Édition de liens statiques.
  - Édition de liens dynamiques.
- Chargé en mémoire pour exécution.

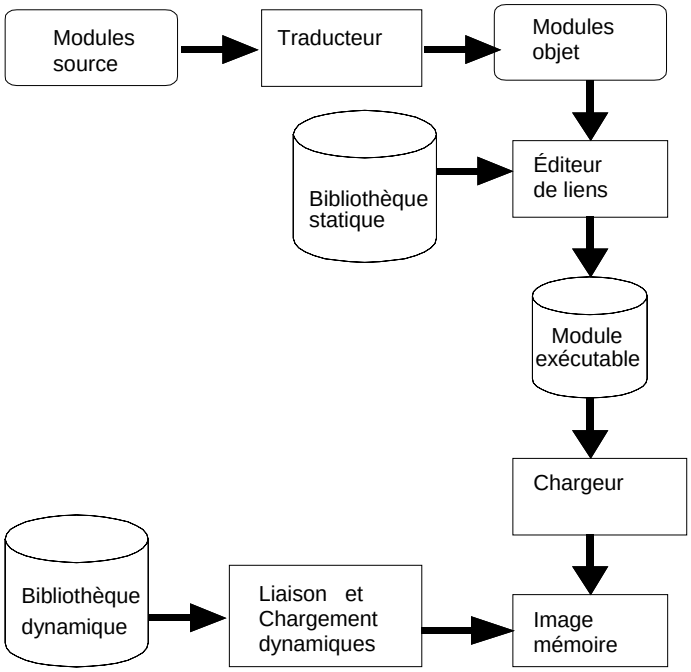


FIG. 9.1 – *Phases de la compilation.*

Représentation des adresses mémoire

- Code source : **adresses symboliques**. Par exemple : `int compteur`
- Module objet : **adresses traduites**. Par exemple le 50ème mot depuis le début d'espace mémoire.
- Module exécutable, chargement : **adresses absolues**. Par exemple l'emplacement mémoire situé à l'adresse FFF7

Liaison d'adresses

Le programme réside sur disque comme *Fichier exécutable*. Il est ensuite chargé en mémoire pour être exécuté. Il peut résider, le plus souvent, dans une partie quelconque de la mémoire. On les appelle *Programmes translatables*[1]. Une exception sont les programmes `.com` de MS/DOS car la compilation est statique.

1. **Translation** sera synonime de **traduction** dans notre contexte, et pourront être utilisés indistinctement.

Liaison des instructions à des adresses mémoire

- Pendant la compilation. Si l'emplacement du processus en mémoire est connu, le compilateur génère des adresses absolues.
- Pendant le chargement. Code translatable si l'emplacement du processus en mémoire n'est pas connu.
- Pendant l'exécution : déplacement dynamique du processus possible. Utilise des fonctions du matériel.

Espace d'adressage logique ou physique

L'unité centrale manipule des **adresses logiques** (emplacement relatif). Les programmes ne connaissent que des adresses logiques, ou virtuelles. L'*espace d'adressage logique* (virtuel) est donc un ensemble d'adresses pouvant être générées par un programme.

L'unité mémoire manipule des **adresses physiques** (emplacement mémoire). Elles ne sont jamais vues par les programmes utilisateurs. L'*espace d'adressage physique* est un ensemble d'adresses physiques correspondant à un espace d'adresses logiques.

9.1.2 La gestion de la mémoire

La mémoire centrale peut avoir une représentation Linéaire et uniforme ou Différenciée :

- **Linéaire et uniforme** : Mémoire non différenciée, avec un espace d'adressage linéaire ou plat (*One level flat address space*).
- **Différenciée** : Avec des **segments linéaires spécifiques et disjoints** pour les procédures, la table de symboles, les programmes, etc. ou avec un **adressage segmenté** à l'aide de deux index, un qui contrôle le numéro de segment et un autre qui contrôle l'adresse dans le segment.

La gestion de la mémoire a deux objectifs principaux : d'abord le partage de mémoire physique entre les programmes et les données des processus prêts, et ensuite la mise en place des paramètres de calcul d'adresses, permettant de transformer une adresse virtuelle en adresse physique. Pour ce faire le gestionnaire de la mémoire doit remplir plusieurs tâches :

- Connaître l'état de la mémoire (les parties libres et occupées de la mémoire).
- Allouer de la mémoire à un processus avant son exécution.

– Récupérer l'espace alloué à un processus lorsque celui-ci se termine.
– Traiter le va-et-vient (*swapping*) entre le disque et la mémoire principale lorsque cette dernière ne peut pas contenir tous les processus.
– Posséder un mécanisme de calcul de l'adresse physique (absolue) car, en général, les adresses générées par les compilateurs et les éditeurs de liens sont des adresses relatives ou virtuelles.

En général, pour ne pas ralentir l'accès à la mémoire le calcul des adresses est réalisé par le matériel. Comme nous allons le voir, les techniques de gestion de la mémoire sont très conditionnées par les caractéristiques du système (monoprogrammé ou multiprogrammé) et du matériel (registres de base et limite, MMU). Nous passerons en revue un certain nombre de méthodes de gestion de la mémoire.

9.2 Système monoprogrammé

9.2.1 Machine nue

Il s'agit d'un des schémas de gestion de mémoire les plus simples. L'utilisateur a tout le contrôle sur l'espace de mémoire, comme le montre la figure 9.2. Cette gestion a comme avantages la flexibilité : l'utilisateur peut contrôler —à sa guise— l'usage de la mémoire et un coût minime, car il n'y a pas besoin de matériel spécifique pour l'implanter. Parmi ses limitations, on peut citer l'inexistance des services d'entrées-sorties, des périphériques externes et de gestion d'erreurs. Ce schéma est utilisé en systèmes dédiés à des tâches spécifiques d'une grande simplicité.

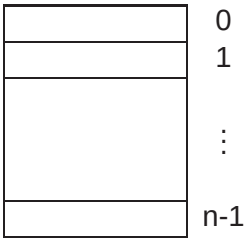


FIG. 9.2 – *Machine nue.*

9.2.2 Moniteur résident

Le schéma suivant est celui qui divise la mémoire en deux sections, une pour les programmes des utilisateurs et une autre pour le **moniteur ré-**

sident du système. Le moniteur résident se retrouve typiquement dans les premières adresses du système, comme illustré à la figure 9.3. Les composantes principales d'un système d'exploitation présentes dans la section du moniteur sont le **vecteur d'interruptions** qui donne accès aux services d'E/S et les **tables PCB** (Bloc de Contrôle de Processus).

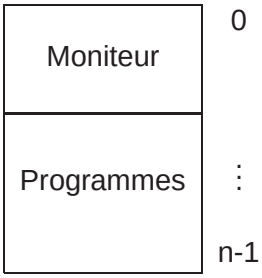


FIG. 9.3 – *Moniteur résident.*

Un exemple de tel type de moniteur est celui de MS-DOS. Un seul processus utilisateur est en mémoire à un instant donné (cette technique n'est plus utilisée de nos jours). On trouve en mémoire, par exemple, le système d'exploitation en mémoire basse, les pilotes de périphériques en mémoire haute (dans une zone allant de 640 Ko à 1 Mo) et un programme d'utilisateur entre les deux (voir figure 9.4).

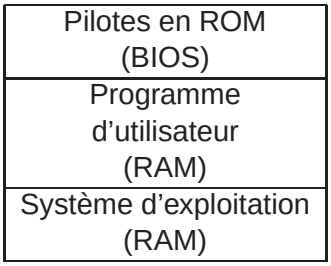


FIG. 9.4 – *Organisation de la mémoire du PC sous MS-DOS.*

Lorsque la mémoire est organisée de cette manière, il ne peut y avoir qu'un seul processus qui s'exécute à un instant donné. L'utilisateur entre une commande sur un terminal et le système d'exploitation charge le programme demandé en mémoire puis l'exécute. Lorsqu'il se termine, le système d'exploitation affiche une invitation sur le terminal et attend la commande suivante pour charger un nouveau processus qui remplace le précédent.

Pour protéger le code et les données du système d'exploitation des

changements que peuvent provoquer les programmes des utilisateurs, il y a un **registre limite** (montré à la figure 9.5) qui contient l'adresse à partir de laquelle commencent les instructions et données des programmes des utilisateurs. Chaque adresse (instruction ou donnée) générée par un programme est comparée avec le registre limite. Si l'adresse générée est supérieure ou égale à la valeur du registre, on l'accepte, sinon on refuse l'accès et on interrompt le processus avec un message d'accès illégal de mémoire.

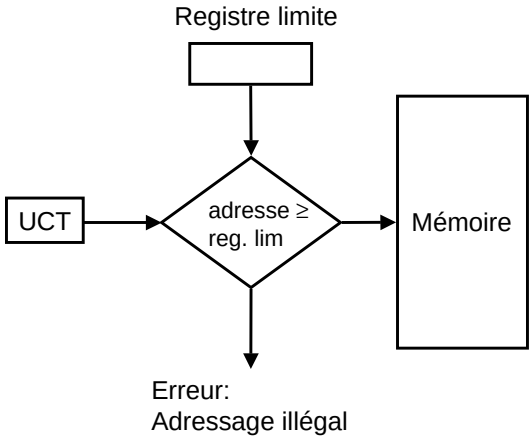


FIG. 9.5 – *Registre limite.*

Toute référence à la mémoire doit être vérifiée, ce qui implique un ralentissement du système.

Il y a plusieurs stratégies pour implanter le **registre limite**. L'une d'entre elles consiste à utiliser une valeur fixe, enregistrée au niveau matériel ou au début du démarrage du système. Cependant si la valeur choisie est trop petite et le système d'exploitation grandisse, il se retrouvera dans une zone non protégée, ou au contraire, l'espace pour les programmes serait diminué. Une solution est l'existence d'un registre variable pour permettre d'initialiser différentes valeurs d'adresses au registre limite. Celui-ci peut être chargé par le système d'exploitation qui utilise des privilèges spéciaux, c'est à dire, on exécute le système d'exploitation en **mode moniteur**.

9.3 Système multiprogrammé

La multiprogrammation permet l'exécution de plusieurs processus à la fois. Elle permet d'optimiser le taux d'utilisation du processeur en rédui-

sant notamment les attentes sur des entrées-sorties. Cette technique nécessite la présence de plusieurs processus en mémoire. La mémoire est donc partagée entre le système d'exploitation et plusieurs processus. Il se pose cependant le problème suivant : *comment organiser la mémoire de manière à faire cohabiter efficacement plusieurs processus tout en assurant la protection des processus ?* Deux cas sont alors à distinguer :

Multiprogrammation sans va-et-vient. Dans ce cas, un processus chargé en mémoire y séjournera jusqu'à ce qu'il se termine (pas de va-et-vient entre la mémoire et le disque).

Multiprogrammation avec va-et-vient. Dans ce cas, un processus peut déplacer temporairement sur le disque (mémoire de réserve : *swap area* ou *backing store*) pour permettre le chargement et donc l'exécution d'autres processus. Le processus déplacé sur le disque sera ultérieurement rechargé en mémoire pour lui permettre de poursuivre son exécution.

Ce schéma pose d'autres difficultés : étant donné que les processus sont chargés à différentes adresses, les adresses relatives générées par les compilateurs et les éditeurs de liens sont différentes des adresses physiques, alors comment réaliser la translation d'adresses ? D'un autre coté, la cohabitation de plusieurs processus en mémoire doit assurer leur protection : il ne faut pas permettre à un processus d'un utilisateur d'adresser la mémoire des processus des autres utilisateurs. Quelle est la meilleur manière d'assurer la protection des processus contre les intrusions des autres ?

9.3.1 Translation des adresses lors du chargement et protection

Utiliser un registre limite a des conséquences. D'abord les programmes doivent posséder un mécanisme pour leur permettre de déterminer si un changement dans la valeur du registre limite sera ou non illégal. Pour éviter cela, la solution triviale consiste en recompiler le programme. Mais ceci n'est pas optimal.

La solution optimale est la génération de code relocalisable, c'est à dire un code où les instructions et les données vont se lier à leurs adresses définitives en mémoire en *temps de chargement* et pas en temps de compilation. Cependant, le registre limite ne peut pas être modifié pendant l'exécution d'un programme : le système d'exploitation doit attendre qu'aucun programme ne s'exécute avant de le modifier. Il existe alors deux manières pour que le système d'exploitation puisse changer la taille du moniteur : la première où les programmes des utilisateurs sont chargés depuis des

hautes positions de mémoire vers le moniteur (voir la figure 9.6). La deuxième consiste à effectuer les liens des adresses dans le temps d'exécution, et ajouter à chaque adresse générée par le programme la valeur du registre limite. De cette façon les programmes vont manipuler des adresses logiques et le matériel de gestion de mémoire les convertira en adresses physiques, comme montre la figure 9.7.

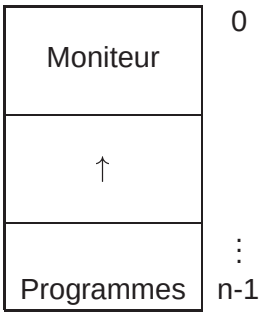


FIG. 9.6 – *Chargement des programmes depuis les hautes positions.*

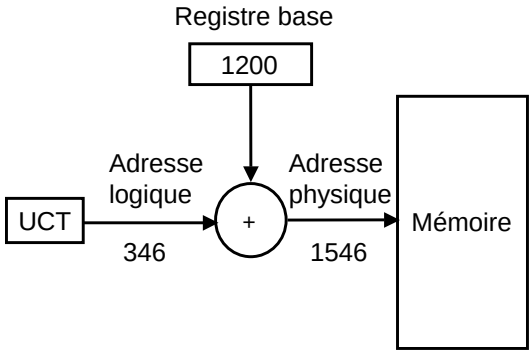


FIG. 9.7 – *Registres limite.*

9.4 Swapping

Le schéma de moniteur résident a été utilisé depuis l'introduction des premiers systèmes d'exploitation multiprogrammés. On utilise une partie de la mémoire pour le moniteur résident et le reste pour les programmes des utilisateurs. Quand le système d'exploitation rend le contrôle au programme suivant, le contenu de la mémoire est copié à un stockage tempo-

raire, et les données et les instructions d'un nouveau programme sont récupérés depuis le stockage secondaire vers la mémoire principale. Ce schéma est connu comme *swapping*.

Le stockage temporaire peut être un disque rapide, ayant l'espace nécessaire pour contenir les images mémoire de tous les programmes en exécution (résidents dans la file de prêts). Évidemment le temps de *swapping* est trop grand comparé au temps de l'UCT, c'est pourquoi il est préférable que chaque processus qui obtient le contrôle de l'UCT s'exécute pour un temps supérieur au temps de *swap* (voir figure 9.8).

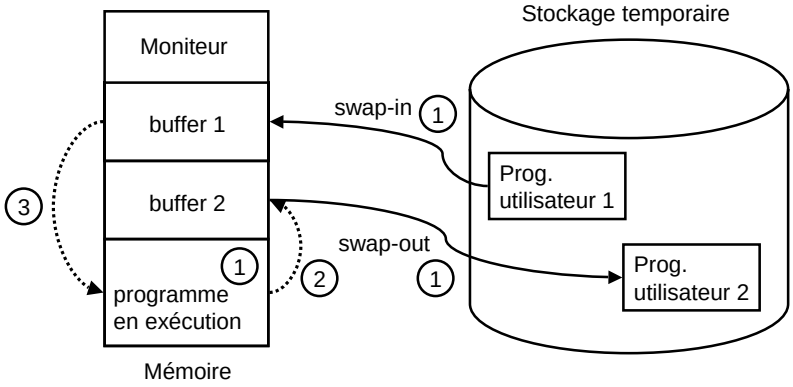


FIG. 9.8 – *(1) Tant que le programme s'exécute, le système d'exploitation fait un swapping de 2 processus, l'un qui sort (swap out) et l'autre qui rentre (swap-in). (2) Quand le processus en exécution termine, ses données et instructions passent au buffer 2 pour ensuite faire swap-out de ce processus. (3) Depuis le buffer 1 on extrait les données et les instructions d'un nouveau processus à s'exécuter.*

9.5 Partitions fixes sans va-et-vient

Dans ce schéma, plusieurs programmes peuvent partager la mémoire. Pour chaque utilisateur existe une partition. La méthode la plus simple consiste à diviser la mémoire en n partitions qui peuvent être de tailles égales ou inégales. On appelle ceci *Mutiprogramming with a Fixed Number of Tasks* (**MFT**). [2]

Avec **MFT**, le nombre et la taille des partitions sont fixés à l'avance. Par

2. Ce partitionnement peut être fait par l'opérateur au démarrage du système, comme c'est le cas du OS / MFT 360 de IBM.

exemple, avec une mémoire de 32 Ko, et un moniteur utilisant 10 Ko, le reste de 22 Ko peut être divisé en 3 partitions fixes de 4Ko, 6 Ko et 12 Ko. La décision sur le nombre et la taille des partitions dépend du niveau de multiprogrammation voulu. En tout cas, le programme avec le plus grande besoin de mémoire doit avoir au moins, une partition pour y mettre ses données et ses instructions.

MFT doit implanter des mécanismes de protection pour permettre à plusieurs programmes de s'exécuter sans envahir l'espace d'adresses d'autre programme. Pour cela il existe deux registres qui indiquent la plage des adresses disponibles pour chaque programme. Les deux interprétations courantes sont les **registres limite** et les **registres base-limite**.

9.5.1 Registres limite

Dans ce cas, les valeurs des registres indiquent la valeur de l'adresse de mémoire la plus petite et la plus grande qu'un programme puisse atteindre. Il y a une paire de valeurs de ces **registres limite** pour chaque programme en exécution. Chaque accès en mémoire entraîne deux vérifications : la première pour ne pas aller en dessous de la limite inférieure, et la seconde pour ne pas aller en dessus de la limite supérieure de mémoire, comme illustré sur les figures 9.9 et 9.10.

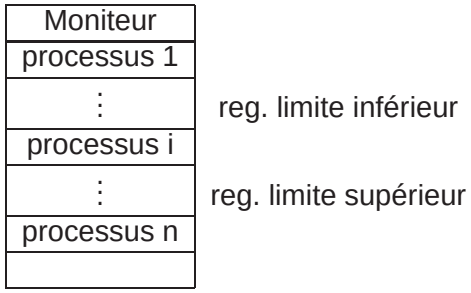


FIG. 9.9 – *Registres limite et mémoire.*

9.5.2 Registres de base et limite

Pour éviter les problèmes de translation d'adresse au cours du chargement d'un processus et de protection, une autre solution consiste à doter la machine de deux registres matériels spéciaux : le **registre de base** et le **registre limite**, comme montré sur la figure 9.11.

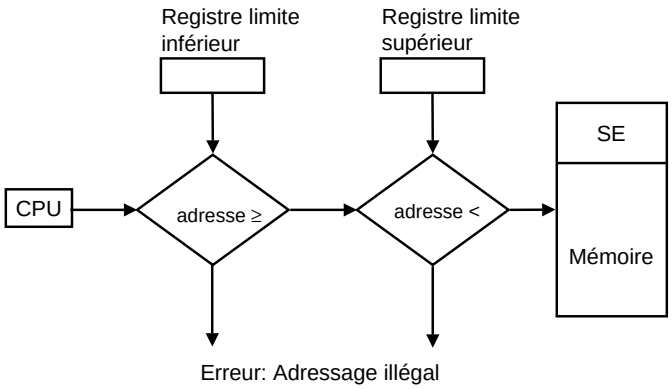


FIG. 9.10 – *Registres limite.*

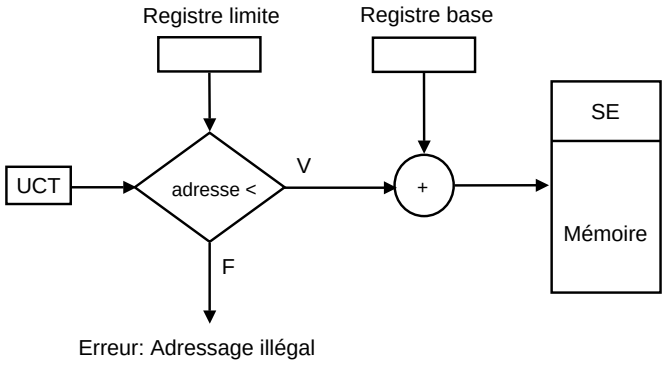


FIG. 9.11 – *Registres de base et limite.*

Quand on lance un processus, on charge dans le registre de base la limite inférieure de la partition qui lui est attribuée (adresse absolue du début de la partition) et dans le registre limite la longueur de la partition. Pendant l'exécution d'une instruction, les adresses relatives référencées par l'instruction sont translatées en adresses physiques en ajoutant le contenu du registre de base à chaque adresse mémoire référencée, on vérifie ensuite si les adresses ne dépassent pas le registre imite afin d'interdire tout accès en dehors de la partition courante [3].

Un important avantage du registre de base est que les programmes peuvent être déplacés en mémoire *après* le début de leur exécution. Il suffit

3. Le PC sous MS-DOS utilise une version simplifiée de ce mécanisme. Voir la note à la fin de ce chapitre.

pour cela de changer la valeur du registre de base, et éventuellement celle du registre limite.

9.5.3 Fragmentation

Avec le schéma **MFT** on fait face au problème de **fragmentation**, puisqu'on génère deux types d'espaces de mémoire non utilisés : la partie d'une partition non utilisée par un processus est nommée **fragmentation interne**. Ceci est dû au fait qu'il est très difficile que tous les processus rentrent exactement dans la taille de la partition. Les partitions qui ne sont pas utilisées, engendrent la **fragmentation externe**. Le phénomène de fragmentation est montré sur la figure 9.12.

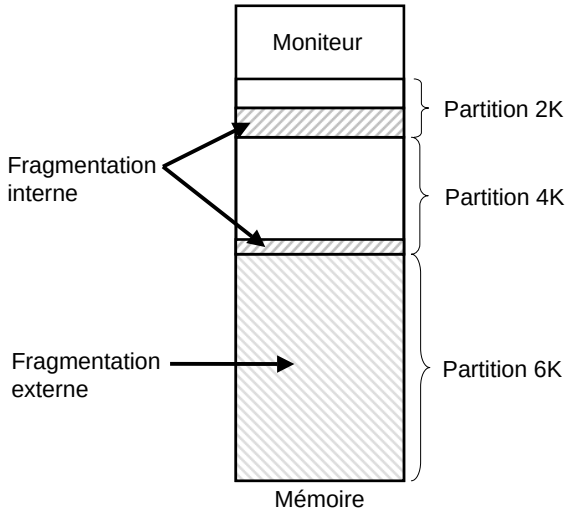


FIG. 9.12 – *Fragmentation interne et externe.*

9.5.4 Files d'entrée

Un autre aspect important est celui de la gestion des requêtes de la mémoire. Il peut y avoir une **file unique** qui stocke toutes les requêtes pour toutes les partitions ou bien une file pour chaque partition. La figure 9.13 à gauche montre une configuration à **files d'entrée multiples**, et à droite, une file unique pour les requêtes.

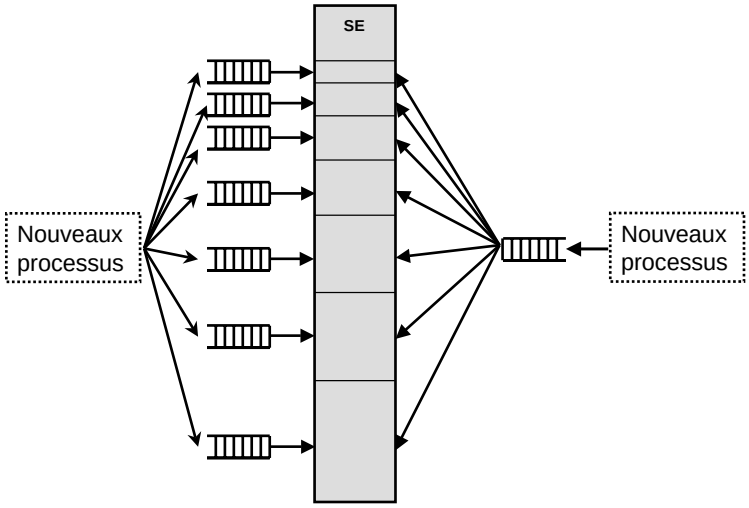


FIG. 9.13 – *File d'entrée multiple à gauche, et file unique à droite.*

Files d'entrée multiples

Chaque nouveau processus est placé dans la file d'attente de la plus petite partition qui peut le contenir. L'espace inutilisé dans une partition allouée à un processus est perdu à cause de la fragmentation interne. La situation suivante peut se produire : des processus en attente alors qu'il y a des partitions vides. La solution à ce problème consiste en l'utilisation d'une seule file d'attente.

File d'entrée unique

Les processus sont placés dans une seule file d'attente jusqu'à ce qu'une partition se libère. Lorsqu'une partition se libère, il faut choisir dans la file d'entrée le prochain processus à charger dans la partition. Le processus chargé résidera en mémoire jusqu'à ce qu'il se termine (pas de va-et-vient). Il y a cependant des inconvénients comme la fragmentation interne et le fait que le nombre de processus en exécution est fixe.

▶ **Exemple 1.** Considérez la configuration montrée sur la figure 9.14 pour une file d'entrée unique, on voit que le processus qui a besoin de 5 Ko entre dans la partition de 6 Ko, le processus de 2 Ko dans la partition de 2 Ko. Le troisième processus de la file reste en attente, bloquant le quatrième processus, qui pourrait s'exécuter dans la partition de 4K, qui est libre, mais

aussi le processus de 1 Ko pourrait être exécuté dans cette partition. Cette décision montre l'alternative entre le concept *best-fit-only* (c'est à dire, exécuter le processus le plus adéquat en termes d'espace) et *first-fit only*, où on exécute le processus avec une priorité plus haute, qui est en tête de la liste.

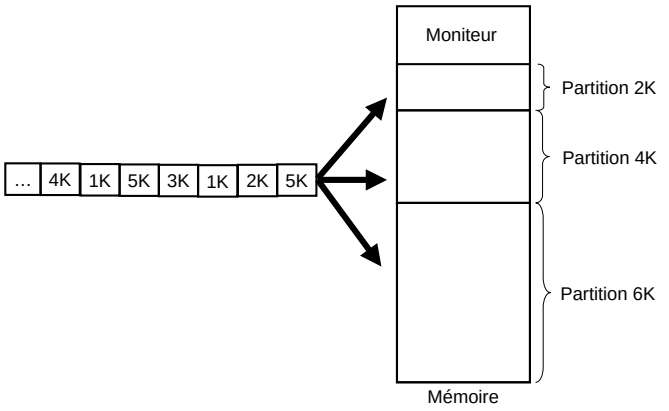


FIG. 9.14 – *Configuration de file d'entrée unique.*

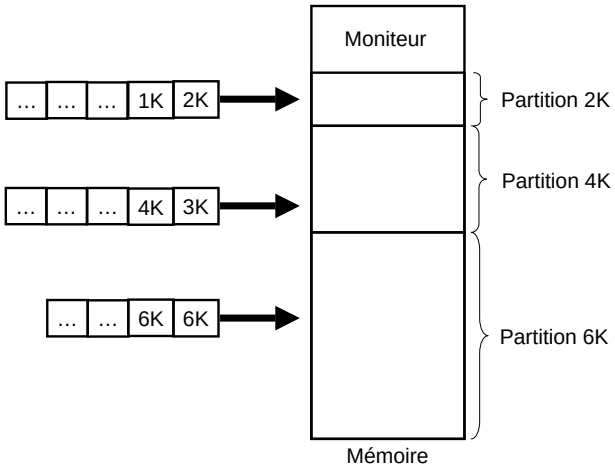


FIG. 9.15 – *Configuration de files multiples.*

Pour assurer la protection des processus dans le système IBM 360, IBM

fut obligée à:

- Diviser la mémoire en blocs de 2Ko et assigner 4 bits de protection à chaque bloc.
- Le mot d'état ou PSW (*program status word*) contenait une clé de 4 bits.
- Le 360 déroutait toutes tentatives d'accès à la mémoire dont les codes de protection différaient de la clé du PSW.
- Ce mécanisme interdisait les interactions entre les processus ou entre les processus et le système d'exploitation.

L'organisation de la mémoire en partitions fixes sans va-et-vient est une méthode facile à mettre en œuvre pour les systèmes à traitement par lots. Mais avec les systèmes à temps partagé, la situation est différente. Puisque la mémoire ne peut pas contenir les processus de tous les utilisateurs, il faut placer quelques uns sur le disque, en utilisant le va-et-vient. Il faut, bien sûr, ramener ces processus en mémoire principale avant de les exécuter. Le mouvement des processus entre la mémoire principale et le disque est appelé va-et vient (*swapping* en anglais). Un autre inconvénient de la partition fixe est la perte de place à cause des programmes qui sont plus petits que les partitions.

9.6 Partitions variables et va-et-vient

Le problème principal avec **MFT** est la détermination optimale des tailles des partitions pour minimiser la fragmentation externe et interne. En systèmes hautement dynamiques, où on ignore le nombre et la taille des processus cela est impossible. Même dans des systèmes avec une charge connue à l'avance, ceci peut devenir problématique.

Par exemple, imaginez un système simple avec une mémoire de 120 Ko, tous les processus utilisent moins de 20 Ko, sauf un qui utilise 80 Ko et qui s'exécute une fois par jour. Évidement pour permettre à tous les processus de s'exécuter le système doit posséder au moins une partition de 80 Ko, et les autres partitions de 20 Ko. Ainsi le système utilisera seulement une fois par jour les 80 Ko, et le reste du temps elle sera inutilisée, avec une fragmentation externe de 80 Ko ; ou bien elle sera occupée partiellement par un processus de 20 Ko avec une fragmentation interne de 60 Ko. Dans les deux cas on n'utilise que la moitié de la mémoire du système.

La solution consiste à faire que les zones de mémoire varient dynamiquement. Cette technique s'appelle assignation de **multiples partitions de mémoire à taille variable** (**MVT**). Le système d'exploitation a une liste de

toutes les zones de mémoire non utilisées et de toutes celles utilisées, ainsi quand un nouveau processus doit s'exécuter, on trouve un espace de mémoire assez grande pour lui contenir et on met toutes ses données et instructions dans cet espace pour l'exécuter.

Avec des partitions variables le nombre et la taille des processus en mémoire varient au cours du temps. L'allocation de la mémoire varie en fonction de l'arrivée et du départ des processus et de la mémoire principale. Cette forme d'allocation conduit éventuellement à l'apparition des trous.

▶ **Exemple 2.** Considérez les caracteristiques suivantes d'un ensemble de processus :

Processus	Besoins de mémoire	Temps
1	60K	10
2	100K	5
3	30K	20
4	70K	8
5	50K	15

TAB. 9.1 – *Leur ordonancement et besoins mémoire sont montrés sur les figures 9.16 et 9.17.*

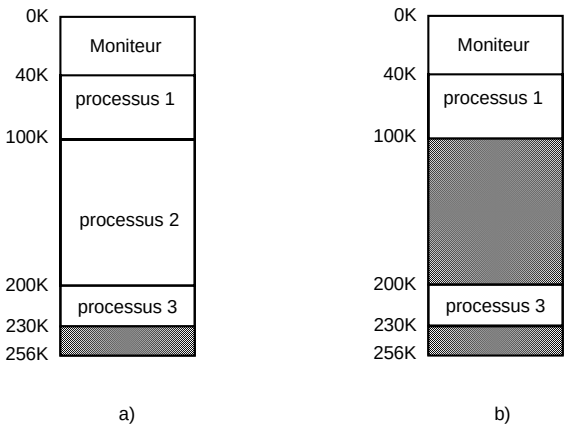


FIG. 9.16 – *Sous un ordonnanceur robin-round (a) le premier qui termine et libère de la mémoire est le processus 2.*

On peut réunir les espaces inutilisés en une seule partition en déplaçant tous les processus vers le bas, comme le fait le **compactage de mémoire**

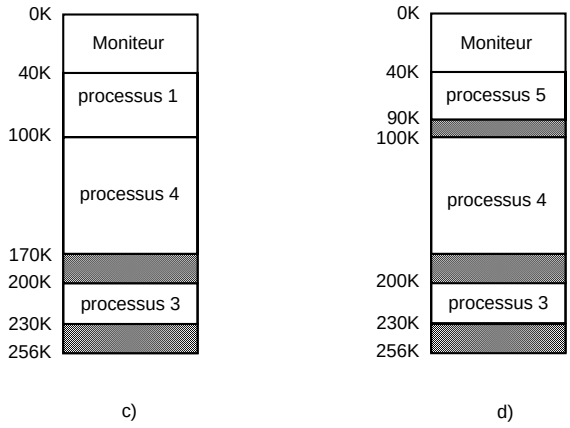


FIG. 9.17 – *Selon FCFS (b) le processus 4 entre. Après le processus 1 termine (c) et le processus 5 entre en exécution (d).*

montré sur la figure 9.18. Cette méthode n'est généralement pas utilisée car elle requiert beaucoup de temps processeur. En revanche, elle est utilisée sur les grands ordinateurs car ils possèdent des circuits spécialisés [4].

L'allocation est simple si les processus ont une taille fixe qu'ils ne modifient jamais. Si par contre, les tailles des segments de données des processus peuvent varier (par exemple par l'utilisation d'allocation dynamique), on a des difficultés dès qu'un processus demande de la mémoire. S'il existe un espace libre adjacent au processus, ce dernier pourra occuper cet espace mémoire. Si cela n'est pas le cas, il faut soit le déplacer dans un emplacement plus grand qui peut le contenir, soit déplacer un ou plusieurs processus sur le disque pour créer un emplacement de taille suffisante. Si la mémoire est saturée et si l'espace de va-et-vient sur le disque ne peut plus contenir de nouveaux processus, il faut tuer le processus qui a demandé de la mémoire.

La taille des processus augmente, en général, au cours de leur exécution. On peut donc allouer à chaque processus déplacé ou recopié à partir du disque un peu plus de mémoire qu'il en demande afin de réduire le temps système perdu pour cette gestion. Il faut cependant recopier sur disque uniquement la mémoire réellement utilisée par le processus. Pour gérer l'allocation et la libération de l'espace mémoire, le gestionnaire doit connaître l'état de la mémoire. Il existe trois manières de mémoriser l'occu-

4. Il faut 1 seconde pour compacter la mémoire de 1 Mo sur un PC. La CDC Cyber a de circuits qui compactent la mémoire à la vitesse de 40 Mo/s.

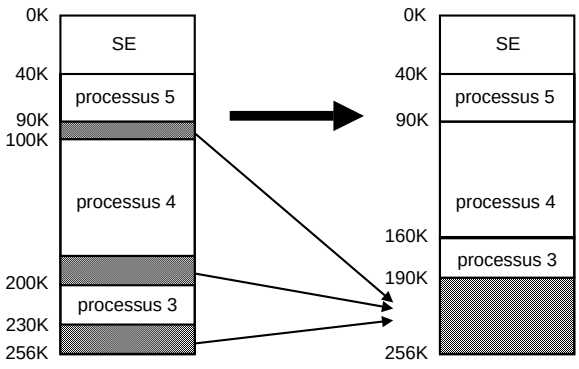


FIG. 9.18 – *Compactage de mémoire.*

pation de la mémoire :

1. Les tables de bits ou *bitmaps*.
2. Les listes chaînées.
3. Les subdivisions.

9.6.1 Gestion de la mémoire par table de bits

La mémoire est divisée en unités d'allocation dont la taille peut varier de quelques mots à plusieurs Ko. A chaque unité, on fait correspondre un bit dans une **Table de bits** ou *bitmap* qui est mis à `0` si l'unité est libre et à `1` si elle est occupée (ou vice versa), comme illustré à la figure 9.19. Plus l'unité d'allocation est faible plus la table de bits est importante. En augmentant la taille de l'unité d'allocation, on réduit la taille de la table de bits mais on perd beaucoup de place mémoire (fragmentation interne).

0	0	1	1	0	0	0	1	1	1	···

FIG. 9.19 – *Table de bits.*

Inconvénient : Lorsqu'on doit ramener un processus de k unités, le gestionnaire de la mémoire doit alors parcourir la table de bits à la recherche de k zéros consécutifs. Cette technique est rarement utilisée car la méthode de recherche est lente.

La figure 9.20 montre l'état de la mémoire avec trois processus qui tournent et la table de bits correspondante.

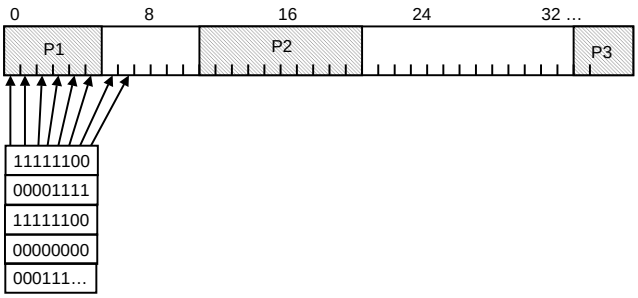


FIG. 9.20 – *État de mémoire avec trois processus et son bitmap.*

9.6.2 Gestion de la mémoire par liste chaînée

Pour mémoriser l'occupation de la mémoire, on utilise une liste chaînée des segments libres et occupés. Un segment est un ensemble d'unités d'allocations consécutives. Un élément de la liste est composé de quatre champs qui indiquent :

- L'état libre ou occupé du segment.
- L'adresse de début du segment.
- La longueur du segment.
- Le pointeur sur l'élément suivant de la liste.

Les listes peuvent être organisées en ordre :

- de libération des zones,
- croissant des tailles,
- décroissant des tailles,
- des adresses de zones.

Par exemple, pour une mémoire ayant l'état d'occupation montré sur la figure 9.21a) on aurait la liste 9.21b).

Libération d'espace

Cet exemple, pris de [Tan01], montre que quatre cas peuvent se présenter lors de la libération de l'espace occupé par un processus X.

- **1er cas.** Modifier l'indicateur de l'état du segment (il passe à libre).

A	X	B	→	A		B

- **2ème cas.** Fusionner les deux segments (il passe à libre).

A	X		→	A	

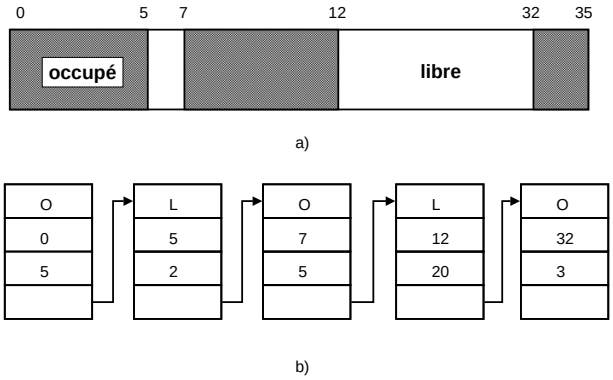
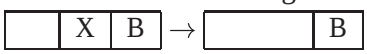
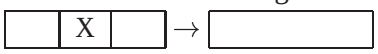


FIG. 9.21 – *a) État de mémoire. b) Liste chaînée.*

– **3ème cas.** Modifier l'indicateur de l'état du segment (il passe à libre). Fusionner les deux segments.



– **4ème cas.** Modifier l'indicateur de l'état du segment (il passe à libre). Fusionner les trois segments.



Allocation d'espace

L'allocation d'un espace libre pour un processus peut se faire suivant quatre stratégies principales : le « premier ajustement » ou *first fit*, la « zone libre suivante », le « meilleur ajustement » ou *best fit* et le « pire ajustement » ou *worst fit*.

1ère zone libre (*first fit*) Dans le cas du « premier ajustement », on recherche le premier bloc libre de la liste qui peut contenir le processus qu'on désire charger. La zone est scindée en deux parties : la première contient le processus et la seconde est l'espace inutilisé (s'il n'occupe pas toute la zone).

Inconvénient : Fragmentation externe (Segments libres entre processus).

Zone libre suivante Il est identique au précédent mais il mémorise en plus la position de l'espace libre trouvé. La recherche suivante commencera à partir de cette position et non à partir du début.

Meilleur ajustement (*best fit*) On cherche dans toute la liste la plus petite zone libre qui convient. On évite de fractionner une grande zone dont on pourrait avoir besoin ultérieurement. Cet algorithme est plus lent que le premier. Il a tendance à remplir la mémoire avec de petites zones libres inutilisables.

Plus grand résidu ou pire ajustement (*worst fit*) Il consiste à prendre toujours la plus grande zone libre pour que la zone libre restante soit la plus grande possible. Les simulations ont montré que cette stratégie ne donne pas de bons résultats.

9.6.3 Algorithme utilisant deux listes séparées

La complexité de la gestion est accrue car cela nécessite des déplacements d'éléments d'une liste à une autre. L'algorithme de placement rapide utilise des listes séparées pour les tailles les plus courantes (la liste des zones de 4 Ko, la liste des zones de 8 Ko, et ainsi de suite).

La recherche est plus rapide, mais lors de la libération il faut examiner les voisins afin de voir si une fusion est possible, sinon on aura une fragmentation en petites zones inutilisables.

9.6.4 Gestion de la mémoire par subdivision

Dans le schéma de **gestion par subdivision** ou *Buddy system*), le gestionnaire de la mémoire mémorise une liste des blocs libres dont les tailles sont de 1, 2, 4, $\ldots 2^n$ octets jusqu'à la taille maximale de la mémoire.

Initialement, on a un seul bloc libre. Lorsqu'il n'y a plus de bloc de taille S_i, on subdivise un bloc de taille S_{i+1} en deux blocs de taille S_i. S'il n'y a plus de bloc de taille S_{i+1}, on casse un bloc de taille S_{i+2}, et ainsi de suite. Lors d'une libération de bloc de taille S_i, si le bloc compagnon est libre, il y a reconstitution d'un bloc de taille S_{i+1}. Pour un système binaire[5] la relation de récurrence sur les tailles est donc :

$$S_{i+1} = 2 \cdot S_i$$

Par exemple, supposons que la taille maximale de mémoire principale est de 1 Mo et que un processus de 70 Ko demande à être chargé. Le gestionnaire détermine d'abord la taille du bloc qui lui sera alloué : qui est égale à la plus petite puissance de 2 qui lui est supérieure, soit 128 Ko. Comme

5. Il peut y avoir d'autres systèmes, comme par exemple celui de Fibonnaci, où la relation de récurrence est : $S_{i+1} = S_i + S_{i-1}; S_0 = S_1 = 1$

il n'y a pas de blocs libres de taille 128 Ko, 256 Ko ou 512 Ko, la mémoire de 1 Mo est divisée en deux blocs de 512 Ko. Le premier bloc est divisé en deux blocs de 256 Ko. Enfin, le premier des deux blocs nouvellement créés est divisé en deux blocs de 128 Ko. L'espace alloué au processus est situé entre l'adresse 0 et 128 Ko.

L'allocation par subdivision est rapide mais elle est assez inefficace en ce qui concerne l'utilisation de la mémoire, car arrondir les tailles à une puissance de 2 implique une certaine fragmentation interne. Linux utilise cet algorithme pour l'assignation de mémoire [BC01].

9.7 Allocation d'espace de 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 déplacer temporairement certains d'entre eux sur une mémoire provisoire, en général une partie réservée du disque (appelée mémoire de réserve, *swap area* ou encore *backing store*), comme montré à la figure 9.22.

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. 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 une 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.

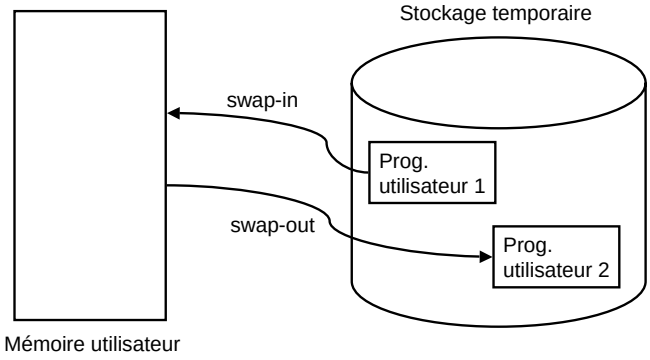


FIG. 9.22 – *Espace de va-et-vient.*

Le système de va-et-vient, s'il permet de pallier le manque de mémoire nécessaire à plusieurs processus, n'autorise cependant pas l'exécution de

programmes de taille supérieure à celle de la mémoire centrale. Une solution consiste à permettre à l'utilisateur de diviser son programme en segments (**segments de recouvrement** ou *overlays* en anglais) et à charger un segment en mémoire. Le segment `0` s'exécute en premier. Lorsqu'il se termine, il appelle un autre segment de recouvrement.

Cette solution n'est pas intéressante, car c'est le programmeur qui doit se charger du découpage de son programme en segments de recouvrement. Une autre solution consiste à décharger le programmeur de cette tâche. Le système d'exploitation charge en mémoire les parties du programme qui sont utilisées. Le reste du programme est stocké sur le disque. On peut ainsi exécuter des programmes dont la taille dépasse celle de la mémoire. Ces techniques appelées mémoire virtuelle seront l'objet d'étude du Chapitre 10, *Mémoire virtuelle*.

9.8 Projection de fichiers en mémoire

On peut utiliser une technique pour **projeter un fichier** dans la mémoire. Si un programme accède à une adresse mémoire qui appartient à la zone associée au fichier, il sera en correspondance directe au fichier. Le programme n'aura plus besoin des services du système d'exploitation pour lire (`read`) et / ou écrire (`write`) dans le fichier.

Il y a quelques avantages de cette technique par rapport à l'accès conventionnel des fichiers par des appels au système.

- Diminution du nombre d'appels au système pour accéder au fichier.
- On évite les copies intermédiaires, car le système d'exploitation transfère directement l'information entre la zone de mémoire et le fichier.
- Plus de facilités de programmation des accès au fichier. Une fois que le fichier est projeté, on peut l'accéder de la même manière que n'importe quelle structure de données du programme.

9.9 Services Posix de projection de fichiers

Les tâches de gestion de mémoire faites par le système d'exploitation sont plutôt internes. C'est pourquoi il n'y a pas beaucoup de services Posix dédiés aux programmes d'utilisateur [CGdMP01][6]. Les services principaux

6. D'autres techniques de partage de variables et zones mémoire seront vues au Chapitre 7 *Communications IPC*, dans la section 7.3 *Mémoire partagée*.

sont en rapport avec la projection de fichiers.

Projection d'un fichier Ce service permet d'inclure dans la mémoire d'un processus un fichier au complet ou une partie. Cette opération crée une zone en mémoire associée à l'objet stocké dans le fichier. Il est possible de spécifier quelques propriétés de la zone mémoire, par exemple le type de protection, ou si la zone est partagée ou privée.

Libération de la projection Ce service élimine une projection entière ou une partie.

9.9.1 Projection d'un fichier

L'appel au système `mmap()` a le prototype :

```
#include <sys/mman.h>
caddr_t mmap(caddr_t adresse, size_t longueur, int protection,
             int attribut, int fd, off_t décalage);
```

- `adresse` : indique l'adresse où on projettera le fichier. Normalement on met la valeur `NULL` pour laisser au système le choix.
- `longueur` : longueur de la projection.
- `protection` : mode de protection. `PROT_READ` (lecture), `PROT_WRITE` (écriture) ou `PROT_EXEC` (exécution) ou leurs combinaisons. Le mode de protection doit être compatible avec le mode d'aperture du fichier.
- `fd` : descripteur du fichier.
- `attribut` : permet d'établir des certaines propriétés de la zone.
- `décalage` : correspond à la position dans le fichier de la projection. On n'est pas obligé de projeter tout le fichier.
  - `MAP_SHARED` : Zone partagée. Les modifications vont affecter le fichier. Un processus fils partage cette zone avec le père.
  - `MAP_PRIVATE` : Zone privée. Les modifications n'affectent pas le fichier. Un processus fils ne partage pas cette zone avec son père, il obtient un dupliqué.
  - `MAP_FIXED` : Le fichier va se projeter exactement dans l'adresse spécifiée par le premier paramètre s'il est $\neq$ `0`.

9.9.2 Libération de la projection

L'appel au système `munmap()` avec le prototype :

```
int munmap(void *zone, size_t longueur);
```

libère la projection d'une certaine `longueur` qui se trouve dans la `zone` indiquée. Lorsqu'il réussit il renvoi `0`.

▶ **Exemple 3.** L'exemple suivant, inspiré de [Bla02] montre l'utilisation des appels système[7] `mmap()` et `munmap()`. Il projettera un fichier comme un tableau de caractères, et ensuite manipulera cette projection en mémoire pour changer le premier caractère avec le dernier, le deuxième avec l'avant dernier, etc. Effectuer la même tâche avec des opérations `read()` et `write()` peut s'avérer beaucoup plus compliqué.

Listing 9.1: *mmap1.c*

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main(int argc, char *argv[])
{
char   *projection;
int     fichier;
struct stat etat_fichier;
long    taille_fichier;
int     i;
char    tmp;

if(argc != 2)
  { printf("Syntaxe: %s fichier a inverser\n",argv[0]);
    exit(1);
  }
fichier = open(argv[1], O_RDWR);
stat(argv[1], &etat_fichier);

taille_fichier = etat_fichier.st_size;
projection = (char *) mmap(NULL, taille_fichier,
                           PROT_READ | PROT_WRITE,
                           MAP_SHARED, fichier, 0);
```

7. L'emploi de l'appel système `stat()` sera aussi utilisé pour déterminer la taille du fichier en question. `stat()` fait partie des appels système pour la gestion du *Système de fichiers* (voir chapitre 11).

```
 if(projection == (char *) MAP_FAILED)
    {
      perror("mmap");
      exit (0);
    }
 close( fichier );

 for(i=0; i< taille_fichier /2; i++)
 {
    tmp = projection[i];
    projection[i] = projection[ taille_fichier - i -1];
    projection[ taille_fichier - i -1] = tmp;
 }
 munmap((void *) projection, taille_fichier );
 return 0;
}
```

```
leibnitz> cat > test.txt
No hay nada que canse mas a una persona
que tener que luchar,
no contra su espiritu,
sino contra una abstraccion.
-José Saramago.
^D
leibnitz> mmap1 test.txt
leibnitz> cat test.txt

.ogamaraS ésoJ-
.noiccartsba anu artnoc onis
,utiripse us artnoc on
,rahcul euq renet euq
anosrep anu a sam esnac euq adan yah oNleibnitz>
leibnitz> mmap1 test.txt
leibnitz> cat test.txt
No hay nada que canse mas a una persona
que tener que luchar,
no contra su espiritu,
sino contra una abstraccion.
-José Saramago.
leibnitz>
```

9.10 Note sur MS-DOS

Le système MS-DOS dispose d'un modèle de mémoire nettement plus compliqué que ceux d'Unix et Linux. Il faut interpréter cette complexité comme un défaut : la simplicité d'Unix / Linux 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 80x86— qu'elle exploite jusqu'à la corde. L'architecture originale sur laquelle le MS-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, comme montré sur la figure 9.23.

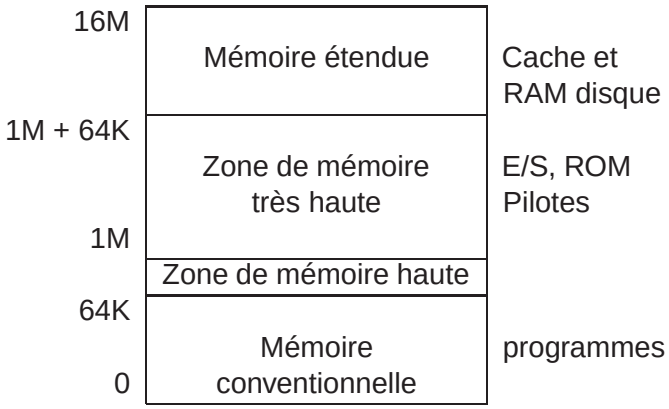


FIG. 9.23 – *Mémoire dans MS-DOS.*

9.11 Exercices

1. Expliquez la fonction principale d'un éditeur de liens dans la génération de l'espace adresse d'un processus. Pourquoi doit-on relocaliser les adresses dans le fichier exécutable qui est généré par l'éditeur de liens? Quels éléments du code doivent être relocalisés? Pourquoi doit-on aussi relocaliser des adresses lors de l'exécution d'un programme?
2. Expliquez l'une des raisons pour laquelle un processus peut être déalloué de la mémoire principale avant la fin de son exécution.
3. Comparez la fragmentation interne à la fragmentation externe pour la mémoire, et expliquez laquelle pénalise le plus la performance du système.
4. Identifiez les zones de mémoire où sont sauvegardés les divers éléments d'un programme en état d'exécution.
5. Donnez un avantage pour l'utilisation de partitions fixes de mémoire, et un avantage pour l'utilisation de partitions variables de mémoire.
6. Laquelle des stratégies d'allocations de partitions (*first fit*, *best fit*, *worst fit*) minimise le plus la fragmentation?
7. Quel avantage procure au système d'exploitation l'utilisation de registres de relocation?
8. Lequel des algorithmes d'allocation de mémoire nous fournirait le meilleur degré de multiprogrammation? Expliquer.
9. Parmi les algorithmes d'ordonnancement de processus, lequel résulterait en un meilleur degré de multiprogrammation? Expliquez.
10. Considérez un système dont l'espace mémoire utilisateur compte 1 Mo. On décide d'effectuer une **partition fixe** de cet espace mémoire en trois partitions de tailles respectives 600 Ko, 300 Ko, 100 Ko. On suppose que les processus arrivent dans le système comme montré sur la table 9.2.

    A(200 Ko, 35) veut dire que le processus A nécessite une partition de 200 Ko et que son temps de séjour en mémoire centrale est 35 unités de temps. Bien entendu, un processus qui ne peut pas être chargé en mémoire est placé sur la file des processus en attente de chargement en mémoire. Un processus chargé en mémoire y séjournera jusqu'à la fin de son exécution. Donnez les états successifs d'occupation de la mémoire si :

    – L'ordonnanceur de haut niveau fonctionne selon le schéma plus

Instant t	Événement
t = 0	A(200 Ko, 35) arrive
t = 10	B(400 Ko, 65) arrive
t = 30	C(400 Ko, 35) arrive
t = 40	D(80 Ko, 25) arrive
t = 50	E(200 Ko, 55) arrive
t = 70	F(300 Ko, 15) arrive

TAB. 9.2 – *Besoins de mémoire et temps d'arrivage.*

court d'abord (SJF) et le mode d'allocation des trous utilise un algorithme du type *best fit*.

- L'ordonnanceur fonctionne selon PAPS (FCFS) et le mode d'allocation des trous utilise un algorithme du type *First Fit*.

11. Considérez un gestionnaire de mémoire utilisant la stratégie de partitions variables. Supposez que la liste de blocs libres contient des blocs de taille de 125, 50, 250, 256, 160, 500, et 75 octets. Le bloc de 125 octets est en tête de liste et le bloc de 75 octets est en fin de liste.
    - Lequel des blocs représente le meilleur espace disponible pour un processus nécessitant une taille de mémoire de 84 octets.
    - Lequel des blocs représente le plus grand espace disponible pour un processus nécessitant une taille de mémoire de 221 octets.
    - Lequel des blocs représente le premier espace disponible pour un processus nécessitant une taille de mémoire de 178 octets.
12. Pour chacune des stratégies d'allocation de blocs de mémoire libre (meilleur, plus grand, ou premier), expliquez de quelle façon la liste de blocs libres devrait être organisée afin d'obtenir une performance optimale.
13. Considérez un système n'utilisant pas de registres de relocation. Par conséquent, c'est le chargeur de programme qui s'occupe de la relocation des instructions dans l'image mémoire à partir de l'exécutable. Serait-il possible de décharger et de recharger l'image mémoire du programme durant le cycle d'exécution? Expliquez comment cela pourrait être réalisable ou pourquoi cela ne pourrait pas être réalisable.

10
Mémoire virtuelle

La **mémoire virtuelle** est une technique qui permet d'exécuter des programmes dont la taille excède la taille de la mémoire réelle. L'**espace d'adressage** d'un processus, généré par les compilateurs et les éditeurs de liens, constitue la **mémoire virtuelle** du processus ou **espace d'adressage logique**, comme le montre la figure 10.1. Avec des techniques adéquates, comme on verra dans le présent chapitre, sa taille peut être très supérieure à celle de la mémoire physique ou réelle.

10.1 Introduction

Il serait trop coûteux d'attribuer à tout processus un espace d'adressage complet, surtout parce que beaucoup n'utilisent qu'une petite partie de son espace d'adressage [JLH00]. En général, la mémoire virtuelle et la mémoire physique sont structurées en unités d'allocations (pages pour la mémoire virtuelle et cadres pour la mémoire physique). La taille d'une page est égale à celle d'un cadre. Lorsqu'un processus est en cours d'exécution, seule une partie de son espace d'adressage est en mémoire.

Les adresses virtuelles référencées par l'instruction en cours doivent être traduites en adresses physiques. Cette conversion d'adresse est effectuée par des circuits matériels de gestion. Si cette adresse correspond à une adresse en mémoire physique, on transmet sur le bus l'adresse réelle, sinon il se produit un **défaut de page**. Par exemple, si la mémoire physique est de 32 Ko et l'adressage est codé sur 16 bits, l'espace d'adressage logique peut atteindre la taille 2^{16} soit 64 Ko.

L'espace d'adressage est structuré en un ensemble d'unités appelées **pages** ou **segments**, qui peuvent être chargées séparément en mémoire. L'ensemble de son espace d'adressage (mémoire virtuelle) est stocké sur

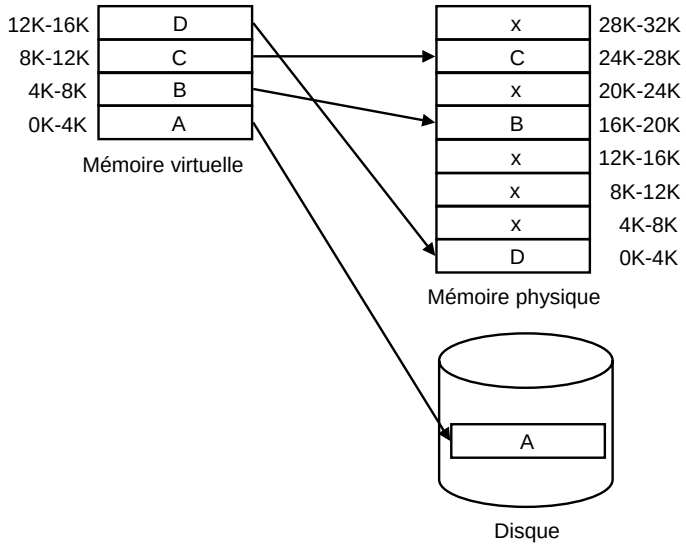


FIG. 10.1 – *Espace des adresses virtuelles et espace physique. Le processus dans son espace virtuel continu comprend quatre pages : A, B, C et D. Trois blocs sont situés dans la mémoire physique et un est sur le disque.*

disque. Pour exécuter un processus, le système d'exploitation charge en mémoire uniquement une ou quelques unités (pages ou segments) y compris celle qui contient le début du programme. Lorsqu'un processus est en cours d'exécution seule une partie de son espace d'adressage est en mémoire principale. Cette partie est dite résidante. Les parties de cet espace sont chargées en mémoire principale à la demande. Elles peuvent être dispersées en mémoire centrale. Lorsqu'un processus référencie une partie qui ne réside pas en mémoire, l'exécution du processus est suspendue, c'est à dire il passe à l'état bloqué en attente d'une E/S. Le système d'exploitation reprend le contrôle, il modifie l'état du processus (état bloqué), lance une demande d'E/S pour ramener la partie référencée, puis exécute d'autres processus.

Lorsque le transfert en mémoire de la partie référencée est terminé (interruption de fin d'E/S), le système d'exploitation reprend le contrôle et modifie l'état du processus bloqué pour passer à l'état prêt.

10.2 Pagination pure

La mémoire virtuelle et la mémoire physique sont structurées en unités d'allocations appelés **pages** pour la mémoire virtuelle et cases ou **cadres** pour la mémoire physique. La taille d'une page est fixe et égale à celle d'un cadre. Elle varie entre 2 Ko et 16 Ko. 4 Ko est une valeur typique assez répandue. Un schéma de la traduction d'adresses lors de la **pagination pure** est montré sur la figure 10.2 Dans la pagination il n'y a pas de fragmenta-

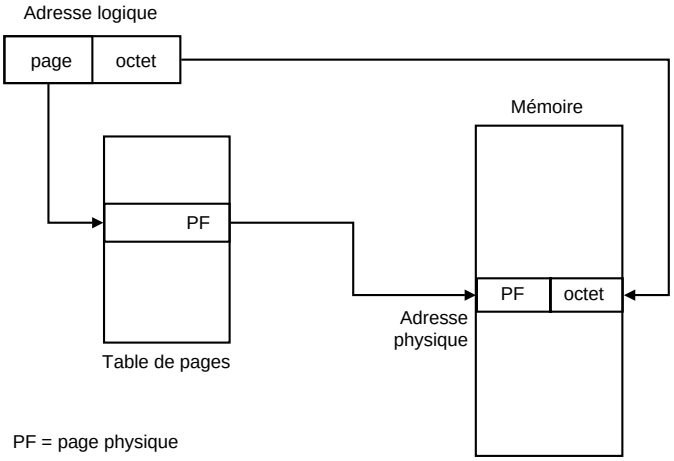


FIG. 10.2 – *Pagination pure.*

tion externe car toutes les pages sont de même taille. Par contre, il peut y avoir une fragmentation interne si la dernière page de l'espace d'adressage logique n'est pas pleine.

▶ **Exemple 1.** Soit un programme de 64 Ko sur une machine 32 Ko de mémoire. Son espace d'adressage logique est sur le disque. Il est composé de 16 pages de 4 Ko. La mémoire physique est structurée en 8 cadres de 4 Ko, comme montré sur la figure 10.3, prise de [Tan01]. Quand le programme tente d'accéder à l'adresse 0 avec une instruction comme `movw $0, %eax`, l'adresse virtuelle 0 est envoyée au MMU (*Memory Management Unit*). Le MMU constate que cette adresse se trouve dans la page 0 (0K - 4K), et la transformation donne la case 2 (8K - 12K) dans la mémoire physique. Cette valeur (8192) est mise dans le bus du système. Ainsi les adresses virtuelles entre 0 et 4095 sont transformées en adresses physiques comprises entre 8192 à 12287.

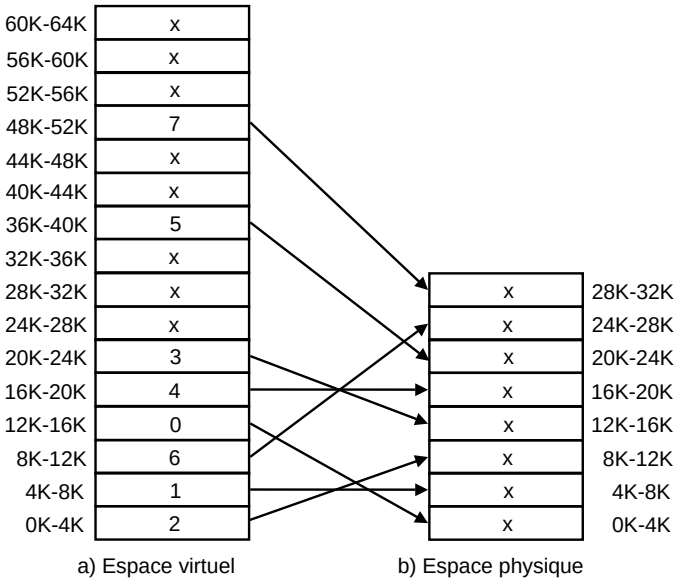


FIG. 10.3 – *Espace virtuel et espace physique.*

10.2.1 Unité de gestion de la mémoire (MMU)

Les adresses virtuelles, générées par les compilateurs et les éditeurs de liens, sont des couples composés d'un numéro de page et d'un déplacement relatif au début de la page. Les adresses virtuelles référencées par l'instruction en cours d'exécution doivent être converties en adresses physiques. Cette conversion d'adresse est effectuée par le MMU, qui sont des circuits matériels de gestion.

Nous allons considérer la page l'unité de transfert. Le MMU vérifie si l'adresse virtuelle reçue correspond à une adresse en mémoire physique, comme montré à la figure 10.4. Si c'est le cas, le MMU transmet sur le bus de la mémoire l'adresse réelle, sinon il se produit un défaut de page. Un défaut de page provoque un déroutement (ou TRAP) dont le rôle est de ramener à partir du disque la page manquante référencée. La correspondance entre les pages et les cases est mémorisée dans une table appelée **Table de pages** (**TP**). Le nombre d'entrées dans cette table est égal au nombre de pages virtuelles. La **Table de pages** d'un processus doit être — en totalité ou en partie — en mémoire centrale lors de l'exécution du processus. Elle est nécessaire pour la conversion des adresses virtuelles en adresses physiques.

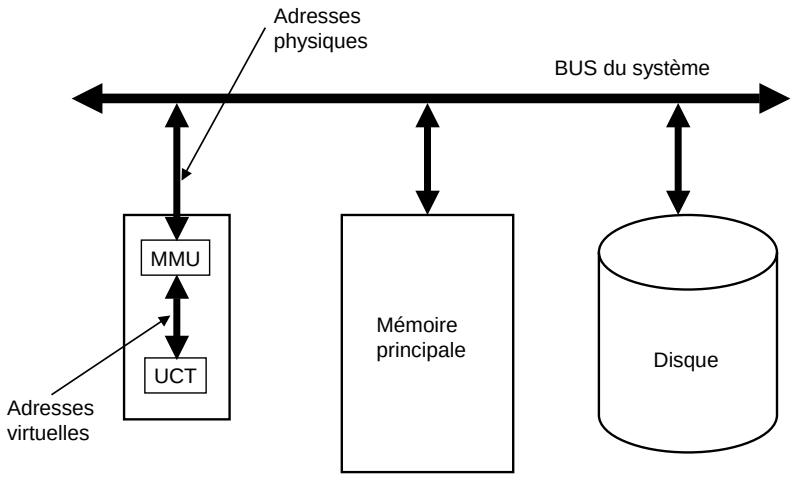


FIG. 10.4 – *Unité de gestion de la mémoire MMU.*

Structure de la Table de pages

Chaque entrée de la **Table de pages** est composée de plusieurs champs, notamment :

1. Le bit de présence.
2. Le bit de référence (R).
3. Les bits de protection.
4. Le bit de modification (M).
5. Le numéro de case correspondant à la page.

La structure d'une table de pages typique, bien que dépendant de la machine, est montrée sur la figure 10.5. La taille varie de machine à machine, mais 32 bits est une taille répandue.

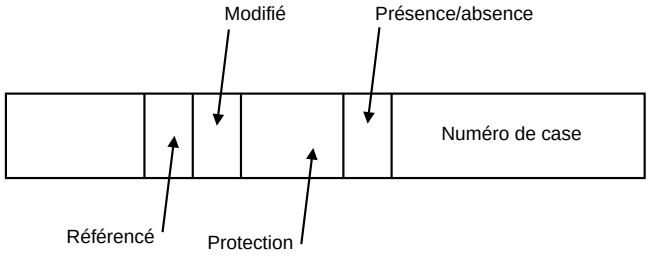


FIG. 10.5 – *Structure d'une entrée typique d'une* ***Table de pages****.*

Fonctionnement d'un MMU

Le MMU reçoit, en entrée une adresse virtuelle et envoie en sortie l'adresse physique ou provoque un déroutement. Dans l'exemple de la figure 10.3, l'adresse virtuelle est codée sur 16 bits. Les 4 bits de poids fort indiquent le numéro de page, comprise entre 0 et 15. Les autres bits donnent le déplacement dans la page, entre `0` et `4095`. Le MMU examine l'entrée dans la **Table de pages** correspondant au numéro de page, dans ce cas `2`. Si le bit de présence est à 0, la page n'est pas en mémoire alors le MMU provoque un déroutement. Sinon, il détermine l'adresse physique en recopiant dans les 3 bits de poids le plus fort le numéro de case `(110)` correspondant au numéro de page (`0010`), et dans les 12 bits de poids le plus faible de l'adresse virtuelle. L'adresse virtuelle `8196` (`0010 0000 0000 0100`) est convertie alors en adresse physique `24580` (`1100 0000 000 0100`) comme le montre la figure 10.6. Le SE maintient une copie de la table

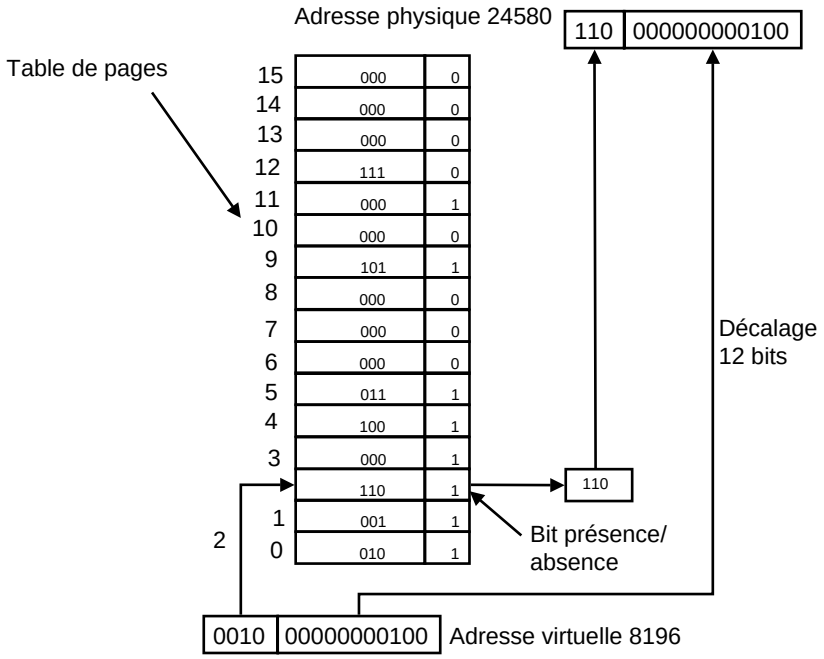


FIG. 10.6 – *Opérations du MMU.*

de pages pour processus, qui permet d'effectuer la traslation des adresses. Cette table permet aussi à l'ordonnanceur d'allouer la UCT à un processus qui devient prêt. La pagination augmente alors le temps de changement de

contexte.

Table de pages à plusieurs niveaux

La taille de la table de pages peut être très grande : par exemple, on aurait plus de 1 million d'entrées (2^{20}) pour un adressage virtuel sur 32 bits et des pages de 4 Ko. Pour éviter d'avoir des tables trop grandes en mémoire, de nombreux ordinateurs utilisent des tables des pages à plusieurs niveaux. Un schéma de table de pages à deux niveaux est montrée sur la figure 10.7.

Par exemple, une table de pages à deux niveaux, pour un adressage sur 32 bits et des pages de 4 Ko, est composée de 1024 tables de 1 Ko. Il est ainsi possible de charger uniquement les tables de 1 Ko nécessaires. Dans ce cas, une adresse virtuelle de 32 bits est composée de trois champs : un pointeur sur la table du 1er niveau, un pointeur sur une table du 2ème niveau et un déplacement dans la page, de 12 bits. Maintenant, si l'on reprend l'exemple

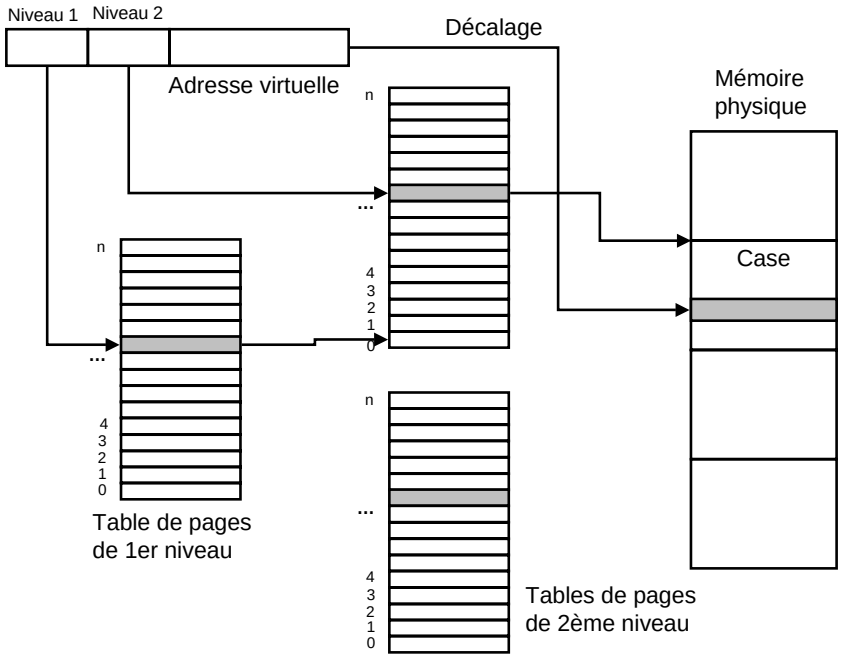


FIG. 10.7 – *Table de pages à deux niveaux.*

cité auparavant —UCT avec des adresses logiques à 32 bits, des pages à 4 Ko avec des entrées à 4 octets— et si le schéma proposé est de deux niveaux

avec 10 bits d'adresses dans chaque niveau. La table de pages de premier niveau aurait une taille de 4 Ko (2^{10} entrées de 4 octets) qui pointeraient vers 1024 tables de pages de deuxième niveau. A leur tour, chaque table de pages de deuxième niveau va utiliser 4 Ko (2^{10} entrées de 4 octets) qui pointeraient vers 1024 cadres. Chaque table de deuxième niveau aura un espace d'adressage de 4 Mo (1024 cadres * 4 Ko).

10.2.2 Accès à la table de pages

L'accès à la table de pages (qui se trouve en mémoire) peut se faire via un registre du MMU qui pointe sur la table de pages. Ceci est une solution lente, car elle nécessite des accès mémoire. Lorsqu'un processus passe à l'état élu, l'adresse de la table de pages est chargée dans le registre. On a constaté que la pagination peut avoir un grand impact sur les performances du système.

MMU avec une mémoire associative

Si la table de pages est grande, la solution précédente n'est pas réalisable. Afin d'accélérer la traduction d'adresses, on peut doter le MMU d'une table de registres machines rapides indexée au moyen du numéro de page virtuelle. Lorsqu'un processus passe à l'état élu, le système d'exploitation charge la **Table de pages** du processus dans la **Table de registres** à partir d'une copie située en mémoire centrale. La solution consiste à doter le MMU d'un dispositif appelé **mémoire associative**, qui est composée d'un petit nombre d'entrées, normalement entre 8 à 64. On a observé que la plupart des programmes ont tendance à faire un grand nombre de références à un petit nombre de pages. La mémoire associative contient des circuits spéciaux pour accéder aux adresses qui sont hautement référencées. La mémoire associative est appelée aussi **TLB** *Translation Lookside Buffer*. Ce composant contient des informations sur les dernières pages référencées. Chaque entrée du **TLB** est composée de :

1. Un bit de validité.
2. Un numéro de page virtuelle.
3. Un bit de modification (M).
4. Deux bits de protection.
5. Un numéro de case.

La traduction d'adresses en utilisant une mémoire associative est montrée à la figure 10.8. Lorsqu'une adresse virtuelle est présentée au MMU,

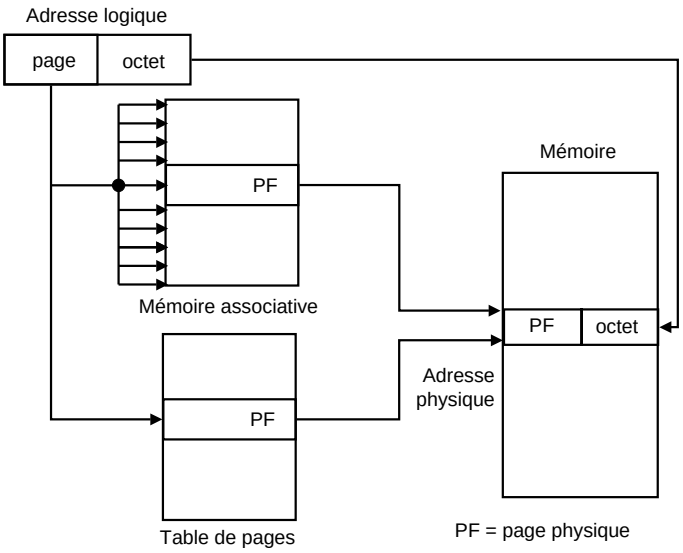


FIG. 10.8 – *Mémoire associative.*

il contrôle d'abord si le numéro de la page virtuelle est présent dans la mémoire associative (figure 10.8), en le comparant simultanément (en parallèle) à toutes les entrées. S'il le trouve et le mode d'accès est conforme aux bits de protection, la case est prise directement de la mémoire associative (sans passer par la Table de pages). Si le numéro de page est présent dans la mémoire associative mais le mode d'accès est non conforme, il se produit un défaut de protection. Si le numéro de page n'est pas dans la mémoire associative, le MMU accède à la Table de pages à l'entrée correspondant au numéro de pages. Si le bit de présence de l'entrée trouvée est à 1, le MMU remplace une des entrées de la mémoire associative par l'entrée trouvée. Sinon, il provoque un défaut de page.

▶ **Exemple 2.** Supposons qu'il faille 100 ns pour accéder à la table de pages et 20ns pour accéder à la mémoire associative. Si la fraction de références mémoire trouvées dans la mémoire associative (taux d'impact) est s, le temps d'accès moyen est alors :

$$\hat{t} = 20s + 100(1 - s)$$

10.3 Algorithmes de remplacement de page

A la suite d'un défaut de page, le système d'exploitation doit ramener en mémoire la page manquante à partir du disque. S'il n'y a pas de cadres libres en mémoire, il doit retirer une page de la mémoire pour la remplacer par celle demandée. Si la page à retirer a été modifiée depuis son chargement en mémoire, il faut la réécrire sur le disque. Quelle est la page à retirer de manière à minimiser le nombre de défauts de page?

Le choix de la page à remplacer peut se limiter aux pages du processus qui ont provoqué le défaut de page (**allocation locale**) ou à l'ensemble des pages en mémoire (**allocation globale**).

En général, l'allocation globale produit de meilleurs résultats que l'allocation locale. Des algorithmes de remplacement de page ont été proposés. Ces algorithmes mémorisent les références passées aux pages. Le choix de la page à retirer dépend des références passées. Il y a plusieurs algorithmes de remplacement de page : l'algorithme de remplacement aléatoire qui choisit au hasard la page victime (qui n'est utilisé que pour des comparaisons), l'algorithme optimal de Belady, l'algorithme de remplacement de la page non récemment utilisée (NRU), l'algorithme FIFO, l'algorithme de l'horloge, et l'algorithme de la page la moins récemment utilisée, parmi d'autres.

10.3.1 Remplacement de page optimal (Algorithme de Belady)

L'algorithme optimal de Belady consiste à retirer la page qui sera référencée le plus tard possible dans le futur. Cette stratégie est impossible à mettre en œuvre car il est difficile de prévoir les références futures d'un programme. Le **remplacement de page optimal** a été cependant utilisé comme base de référence pour les autres stratégies, car il minimise le nombre de défauts de page.

▶ **Exemple 3.** Soit un système avec $m = 3$ cases de mémoire et une suite de références ω={7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1}. À la figure 10.9 on montre une ligne pour chaque case et une colonne pour chaque référence. Chaque élément de ligne i et colonne j montre la page chargée dans la case i après avoir été référencée. Les entrées en **noir** sont des pages chargées après un défaut de page. L'algorithme optimale fait donc 9 défauts de page.

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	2	2	2	2	2	2	2	2	2	2	**7**	7	7
1		**0**	0	0	0	0	0	**4**	4	4	**0**	0	0	0	0	0	0	0	0	0
2			**1**	1	1	**3**	3	3	3	3	3	3	3	**1**	1	1	1	1	1	1

FIG. 10.9 – *Algorithme optimale de Belady.*

10.3.2 Remplacement de la page non récemment utilisée (NRU, *Not Recently Used*)

Cet algorithme utilise les bits **R** et **M** associés à chaque page pour déterminer les pages non récemment utilisées. Le bit **R** est positionné à **1** chaque fois qu'une page est référencée. Le bit **R** est remis à **0** à chaque interruption d'horloge. Le bit **M** est positionné lorsque la page est modifiée (elle n'est plus identique à sa copie sur disque). Lorsqu'un défaut de page se produit, l'algorithme NRU sélectionne la page à retirer en procédant comme suit :

- Il vérifie s'il existe des pages non référencées et non modifiées (**R=0** et **M=0**). Si c'est le cas, il sélectionne une page et la retire.
- Sinon, il vérifie s'il existe des pages non référencées et modifiées (**R=0** et **M=1**). Si c'est le cas, il sélectionne une page et la retire (une sauvegarde sur disque de la page retirée est nécessaire).
- Sinon, il vérifie s'il existe des pages référencées et non modifiées (**R=1** et **M=0**). Si c'est le cas, il sélectionne une page et la retire.
- Sinon, il sélectionne une page référencée et modifiée et la retire (**R=1** et **M=1**). Dans ce cas, une sauvegarde sur disque de la page retirée est nécessaire.

10.3.3 Remplacement de page FIFO

Il mémorise dans une file de discipline FIFO (premier entré, premier sorti) les pages présentes en mémoire. Lorsqu'un défaut de page se produit, il retire la plus ancienne, c'est à dire celle qui se trouve en tête de file.

▶ **Exemple 4.** La suite de références ω={7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1} avec $m = 3$ cases fait 15 défauts de page avec l'algorithme FIFO, comme le montre la figure 10.10.

Cet algorithme ne tient pas compte de l'utilisation de chaque page. Par exemple, à la dixième référence la page `0` est retirée pour être remplacée par

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0	0	0	0	0	**7**	7	7
1		**0**	0	0	0	**3**	3	3	**2**	2	2	2	2	**1**	1	1	1	1	**0**	0
2			**1**	1	1	1	**0**	0	0	**3**	3	3	3	3	**2**	2	2	2	2	**1**

FIG. 10.10 – *Algorithme de remplacement FIFO.*

la page 3 puis tout de suite après la page retirée est rechargée. L'algorithme est rarement utilisé car il y a beaucoup de défauts de page.

10.3.4 Remplacement de page de l'horloge

Dans cet algorithme les pages en mémoire sont mémorisées dans une liste circulaire en forme d'horloge (figure 10.11). Un indicateur pointe sur la page la plus ancienne. Lorsqu'un défaut de page se produit, la page pointée par l'indicateur est examinée. Si le bit R de la page pointée par l'indicateur est à 0, la page est retirée, la nouvelle page est insérée à sa place et l'indicateur avance d'une position. Sinon, il est mis à 0 et l'indicateur avance d'une position. Cette opération est répétée jusqu'à ce qu'une page ayant R égal à 0 soit trouvée.

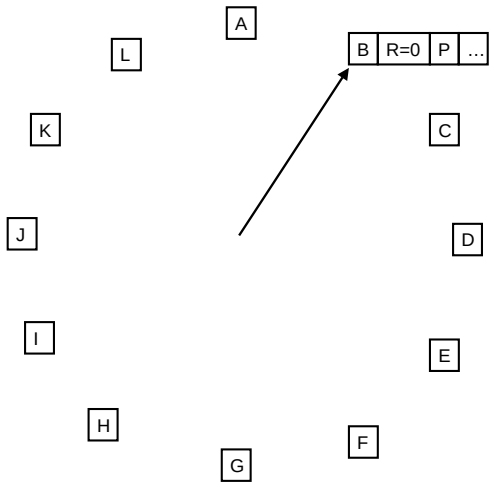


FIG. 10.11 – *Algorithme de l'horloge. Si la valeur de $R = 0$ on tue la page et une nouvelle page est insérée, sinon on met $R = 0$ et on avance un tour.*

10.3.5 Remplacement de la page la moins récemment utilisée (*LRU Last Recently Used*)

L'algorithme LRU mémorise dans une liste chaînée toutes les pages en mémoire. La page la plus utilisée est en tête de liste et la moins utilisée est en queue. Lorsqu'un défaut de page se produit, la page la moins utilisée est retirée. Pour minimiser la recherche et la modification de la liste chaînée, ces opérations peuvent être réalisées par le matériel. Cet algorithme est coûteux.

▶ **Exemple 5.** La suite de références ω={7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1} avec $m = 3$ cases fait 12 défauts de page avec l'algorithme de remplacement de la page la moins récemment utilisée, comme le montre la figure 10.12.

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0	1	1	1	1	**1**	1	1
1		**0**	0	0	0	0	0	**0**	0	**3**	3	3	3	3	3	**0**	0	0	0	0
2			**1**	1	1	**3**	3	3	**2**	2	2	2	2	2	2	2	2	**7**	7	7

FIG. 10.12 – *Algorithme de remplacement de la page la moins récemment utilisée.*

Parmi tous les algorithmes présentés, les simulations ont montré que l'algorithme de l'horloge figure parmi les meilleurs.

10.4 Nombre de cases allouées à un processus

On peut allouer un même nombre de cases mémoire à 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 cases proportionnellement aux tailles des programmes. Si un processus est deux fois plus grand qu'un autre, il recevra le double de cases. L'allocation des cases peut se faire lors du chargement ou à la demande au cours de l'exécution.

10.4.1 Anomalie de Belady

Quand on compare la performance des algorithmes particulièrs, on peut penser à faire varier le nombre de pages allouées au processus. On pourrait ainsi espérer que si l'on incrément le nombre de pages allouées alors

le nombre de défauts va diminuer. Mais sous certaines conditions une situation appelée **Anomalie de Belady** peut se produire et l'ajout de plus de pages se traduit par un incrément du nombre de défauts. L'algorithme FIFO est spécialement touché par cette anomalie.

10.5 Écroulement du système

Si le système passe plus de temps à traiter les défauts de page qu'à exécuter des processus on peut avoir des problème de l'écroulement du système. Si le nombre de processus est trop grand et l'espace propre à chacun sera insuffisant, ils passeront leur temps à gérer des défauts de pages. C'est ce qu'on appelle l'**écroulement du système**. On peut limiter le risque d'écroulement en surveillant le nombre de défauts de page provoqués par un processus. 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. S'il n'y a plus de pages disponibles et trop de défauts de pages, on devra suspendre un des processus.

10.6 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 pour recommencer son exécution. Ce retour sur instruction n'est possible qu'avec l'aide du matériel.

10.7 Segmentation

Dans un système paginé, l'espace d'adressage virtuel d'un processus est à une dimension. Or en général, un processus est composé d'un ensemble d'unités logiques :

- **Les différents codes :** le programme principal, les procédures, les fonctions bibliothèques.
- **Les données initialisées**.
- **Les données non initialisées**.
- **Les piles d'exécution**.

L'idée de la technique de **segmentation** est d'avoir un espace d'adressage à deux dimensions. On peut associer à chaque unité logique un espace d'adressage appelé **segment**. L'espace d'adressage d'un processus est composé d'un ensemble de segments. Ces segments sont de tailles différentes (fragmentation externe). Un schéma de traduction d'adresses par segmentation est montré sur la figure 10.13. La segmentation facilite l'édition de

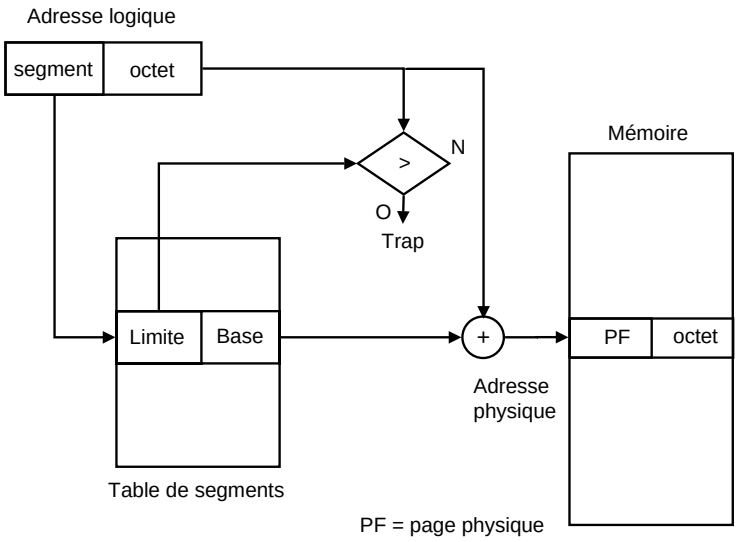


FIG. 10.13 – *Segmentation simple.*

liens, ainsi que le partage entre processus de segments de données ou de codes.

10.8 Segmentation paginée

La segmentation peut être combinée avec la pagination. Chaque segment est composé d'un ensemble de pages. Les adresses générées par les compilateurs et les éditeurs de liens, dans ce cas, sont alors des triplets :

```
<numéro du segment, numéro de page, déplacement dans la page>
```

Le schéma de traduction d'adresses par segmentation paginée est montré sur la figure 10.14.

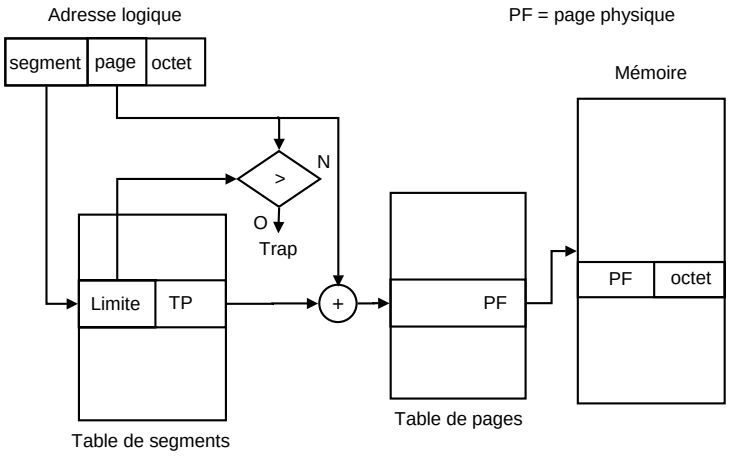


FIG. 10.14 – *Segmentation paginée.*

10.9 Mémoire cache

La mémoire cache est une mémoire à temps d'accès très court ($\leq$ 10 ns). Elle coûte plus chère. Le temps d'accès à la mémoire principale est 100 ns. La mémoire cache est placée entre le processeur et la mémoire centrale, comme montré sur la figure de la hiérarchie de la mémoire 10.15. Dans un système à pagination, lorsqu'une adresse virtuelle est référencée, le système examine si la page est présente dans le cache. Si c'est le cas, l'adresse virtuelle est convertie en adresse physique. Sinon, le système localise la page puis la recopie dans le cache. Le but du cache est de minimiser le temps d'accès moyen $\hat{t}$:

$\hat{t}$ = temps d'accès + taux d'échec · temps de traitement de l'échec

10.10 Cas d'étude

10.10.1 Unix

Les premiers systèmes Unix utilisaient la technique de va-et-vient. Avant l'exécution, un processus est entièrement chargé en mémoire. S'il n'y a pas assez de place, un ou plusieurs processus sont transférés sur le disque.

Le choix du processus à transférer sur le disque dépend essentiellement de trois facteurs : l'état bloqué ou prêt, la priorité et le temps de résidence en mémoire. Toutes les quelques secondes, le permuteur (*swapper* ou **chargeur**) processus de `pid 0` examine l'état des processus transférés sur le

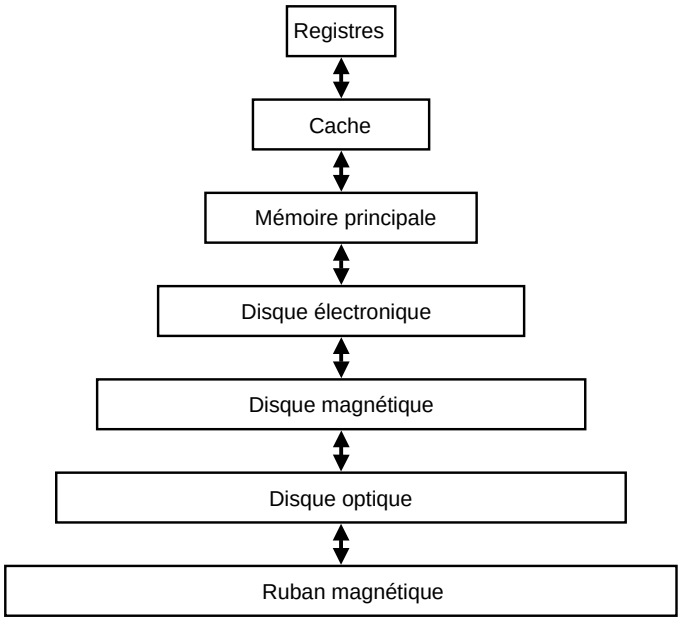


FIG. 10.15 – *Hiérarchie de la mémoire.*

disque pour voir si certains d'entre eux sont devenus prêts. Si c'est le cas, il choisit celui qui est resté trop longtemps sur le disque pour le transférer en mémoire centrale. Cette procédure est répétée jusqu'à ce qu'il n'y ait plus de processus prêts sur le disque, ou tous les processus soient nouvellement chargés et il n'y ait pas assez d'espace mémoire.

Le permuteur mémorise, dans deux listes chaînées, les espaces libres en mémoire et sur le disque. La stratégie d'allocation utilisée est le premier ajustement (première zone libre suffisamment grande). La pagination à la demande a été introduite dans les systèmes Unix par Berkeley (depuis BSD 4.3 et **System V**). Seules les pages nécessaires à l'exécution sont ramenées en mémoire centrale.

Le **voleur de pages** est le processus qui exécute l'algorithme de remplacement de pages. Il est réveillé périodiquement (toutes les secondes) pour voir si le nombre de cases libres en mémoire est au moins égal à un seuil `min`. Si le nombre de cases libres en mémoire est inférieur à `min`, le voleur de pages transfère des pages sur le disque jusqu'à ce que `max` cadres soient disponibles. Sinon, il se remet au sommeil.

Pour le choix des pages à transférer, le voleur de page utilise une version améliorée de l'algorithme de l'horloge. Chaque case est dotée d'un bit

de référence **R** qui est mis à `1` à chaque référence. Il parcourt les pages en mémoire et teste leurs bits de référence. S'il est à `0` alors le voleur de pages incrémente une variable "`âge`" associée à la page, sinon le bit de référence est remis à zéro. Lorsque `âge` dépasse une valeur donnée, la page est placée dans l'état disponible pour le déchargement.

10.10.2 Linux

Sous Linux le format des fichiers exécutables est divisé en régions ou zones, comme suit :

- Type du fichier (les deux premiers octets).
- Zone de code `.text`
- Zone des données `.data`
- Zone des données initialisées `.bss`
- Zone de la pile.

La taille d'un fichier exécutable peut être determinée par la commande `size` :

```
leibnitz> size /vmunix
text data bss dec hex
3243344 494336 856864 4594544 461b70
```

La zone de la pile ou *stack* est visible uniquement à l'exéecution, et elle contient les variables locales et les arguments des fonctions, enre autres.

Chaque processus sur une machine `80x86` de 32-bits a 3 Go d'espace d'adressage virtuel, le Go restant est réservé aux Tables de pages et certaines données du Kernel du système d'exploitation (voir figure 10.16). Les programmes s'exécutent dans l'espace d'utilisateur et les accès à la mémoire se font dans des adresses $\leq$ `0xbfffffff`. Des mécanismes de protection empêchent les programmes d'accéder à l'espace du Kernel, mais on permet de basculer l'exécution vers cet espace lors des appels au système et l'accès aux Entrées / Sorties. C'est le SE qui est chargé d'initialiser correctement l'espace utilisateur lors du chargement d'un programme. Toutes les informations sont contenues dans l'en-tête du fichier exécutable. Un format couramment utilisé sous Linux est le format **ELF** (*Executable and Linkable Format*), qui contient de l'information sur les segments, les librairies dynamiques, l'adresse de la première instruction à exécuter, entre autres. D'autres détails sur le contenu des registres du processeur lors du chargement d'un exécutable sous format **ELF** se trouvent dans [BG02].

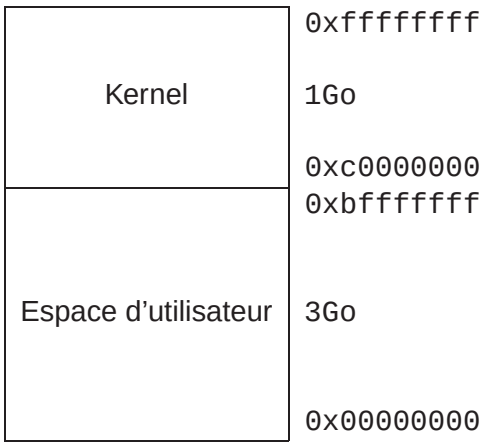


FIG. 10.16 – *Espace mémoire sous Linux.*

L'espace d'adressage est composé d'un ensemble de régions. Chaque région est un ensemble de pages contiguës de 4 Ko. Par exemple, le segment de code est une région [1].

Segmentation paginée dans le 80x86

Le processeur 80x86 utilise la segmentation paginée pour gérer la mémoire. La mémoire physique est divisée en pages de 4Ko. Le nombre maximum de segments par processus est de 16K, et la taille de chaque segment peut aller jusqu'à 4Go. L'idée est d'effectuer une traslation d'adresses logiques en adresses linéaires puis en adresses physiques (figure 10.17).

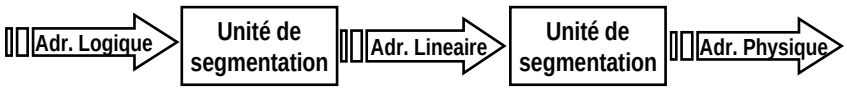


FIG. 10.17 – *Segmentation paginée du 80x86 sous Linux (1).*

Une **adresse logique** est une paire (**sélecteur**, **déplacement**). Le sélecteur est codé sur 16 bits, et le déplacement sur 32 bits (figure 10.18).

Une **adresse linéaire** de 32 bits est divisée en 3 parties (2 niveaux) :

– Pointeur de répertoire de pages sur 10 bits.

– Pointeur de table de pages sur 10 bits.

1. Pour l'assembleur GNU `as` le segment de code est nommé `.text`, le segment des données `.data`, le segment des données initialisées à zéro `.bss`, etc.

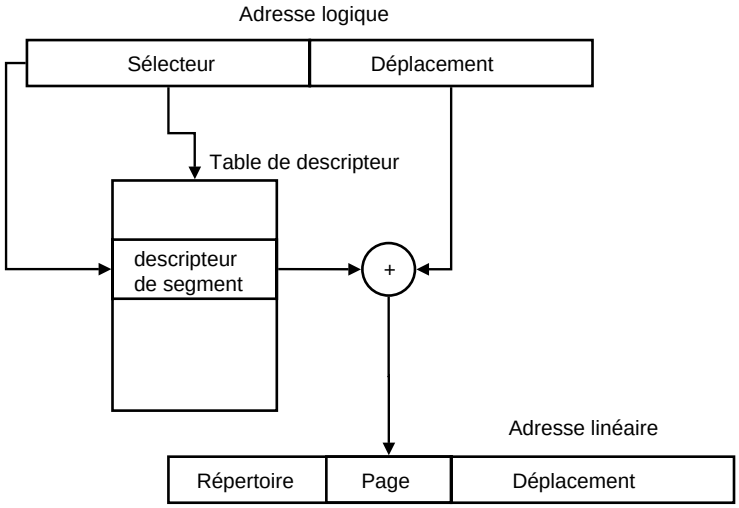


FIG. 10.18 – *Traslation d'adresse logique → adresse linéaire.*

– Déplacement dans la page sur 12 bits.

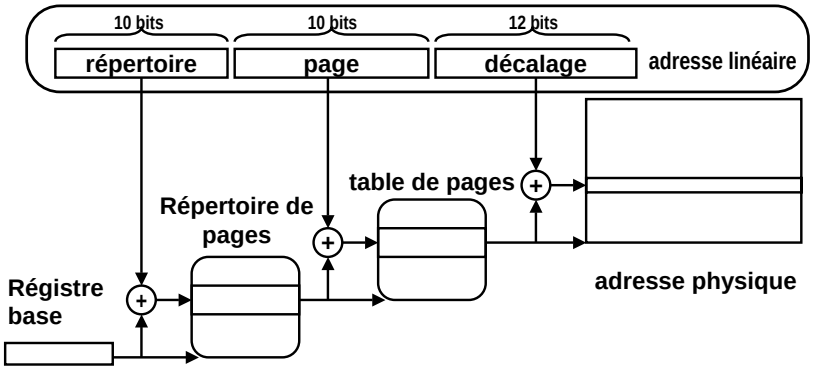


FIG. 10.19 – *Segmentation paginée du 80x86 sous Linux (2).*

Le **gestionnaire de mémoire** est un système de pagination à la demande (voir figure 10.19). La mémoire centrale est gérée comme suit : initialement, la mémoire est composée d'une seule zone libre. Lorsqu'une demande d'allocation arrive, la taille de l'espace demandé est arrondie à une puissance de 2. La zone libre initiale est divisée en deux. Si la première est trop grande, elle est, à son tour, divisée en deux et ainsi de suite. Sinon, elle est allouée au demandeur. Le gestionnaire de la mémoire utilise un tableau qui contient

des têtes de listes. Le premier élément du tableau contient la tête de la liste des zones de taille 1. Le deuxième élément contient la tête de la liste des zones de taille 2 et ainsi de suite. Cet algorithme *Buddy* conduit vers une importante fragmentation interne. Les espaces non utilisés (de la fragmentation interne) sont récupérés et gérés différemment. Lors de la libération de l'espace, les zones contiguës de même taille sont regroupées en une seule zone.

Comme dans Unix, un démon se charge de maintenir au moins un certain nombre de pages libres en mémoire. Il vérifie périodiquement (toutes les secondes) ou après une forte allocation d'espace, l'espace disponible. Si l'espace disponible devient insuffisant, il libère certaines cases (pages) en prenant soin de recopier sur disque celles qui ont été modifiées, en utilisant l'algorithme de l'horloge.

Commande `vmstat`

L'appel système `vmstat` fourni des renseignements et des statistiques sur l'usage de la mémoire virtuelle du système :

```
leibnitz> vmstat
   procs                  memory   swap     io    system         cpu
 r  b  w  swpd  free  buff cache si  so bi  bo  in    cs  us sy  id
 1  0  0  1480 79980 20192 99232  0   0  3   4 226    90  88  1  11
```

où (d'après `man vmstat`) :

`procs`

- `r` : Processus en attente d'UCT.
- `b` : Nombre de processus en *sleep* non interrompible.
- `w` : Nombre de processus *swapped out* mais prêts.

`memory`

- `swpd` : Quantité de mémoire virtuelle utilisée (Ko).
- `free` : Quantité de mémoire libre (Ko).
- `buff` : Quantité de mémoire utilisée comme buffers (Ko).

`swap`

- `si` : Quantité de mémoire *swapped in* depuis le disque (Ko / s).
- `so` : Quantité de mémoire *swapped* au disque (Ko / s).

`io`

- `bi` : Blocs envoyés (blocks / s).
- `bo` : Blocs reçus (blocks / s).

`system`

- `in` : Interruptions à la seconde, incluant l'horloge.
- `cs` : Changements de contexte à la seconde.

`cpu` (Pourcentages de l'utilisation de l'UCT)

- `us` : Temps utilisateur.
- `sy` : Temps du système.
- `id` : Temps non utilisé.

10.10.3 MS-DOS

MS-DOS ne fait ni de va-et-vient ni de pagination à la demande. Plusieurs programmes peuvent être en même temps en mémoire. La mémoire utilisateur est partagée en zones contiguës (arenas). Chaque zone commence par une en-tête. Ce dernier contient notamment la taille de la zone et un pointeur sur le descripteur du processus qui occupe la zone ou 0 si la zone est libre. Lorsque MS-DOS doit allouer de la mémoire, il parcourt la chaîne des zones jusqu'à trouver une zone libre assez grande (premier ajustement). Si la zone trouvée est très grande, elle est scindée en deux parties. Une partie est allouée et l'autre est libre. S'il n'y a pas assez d'espace, il retourne un message d'erreur à la demande d'allocation. La taille des programmes est limitée aux 640 Ko. Des techniques de segments de recouvrement et de mémoire paginée permettent de s'affranchir de cette limite, mais il faut pour cela installer des pilotes de mémoire étendue.

10.11 Exercises

1. Expliquez un avantage majeur que procure l'utilisation de la mémoire virtuelle sur la performance du système.
2. Quelle est la différence principale entre un algorithme de remplacement statique de pages et un algorithme de remplacement dynamique de pages?
3. Considérez une mémoire virtuelle avec une taille de mémoire physique (principale) de 1 Mo, et supportant des blocs de 128 octets. Aussi, supposez un processus occupant un espace d'adresse logique de 22 Ko.
   - Calculez le nombre de cadres dans l'espace d'adresse physique et le nombre de pages dans l'espace d'adresse logique.
   - Montrez les formats des adresses physique et logique, soit le nombre de bits pour les blocs, les cadres et les pages.
   - Déterminez l'adresse physique dans laquelle se situe l'adresse logique 10237, si l'on suppose que la page contenant l'adresse 10237 se trouve dans le cadre 1839.
4. Considérez la séquence de références de pages ω=0, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9 faite par un processus. Montrez la séquence d'allocation des pages en mémoire pour chacun des algorithmes énumérés ci-dessous. Aussi, calculez le taux de faute de pages produit par chacun des algorithmes de remplacement de pages de mémoire.
   (a) L'algorithme optimal de Belady.
   (b) L'algorithme du moins récemment utilisé.
   (c) L'algorithme du moins fréquemment utilisé.
   (d) L'algorithme du premier-arrivé-premier-servi (PAPS).
   (e) L'algorithme de l'espace vital avec une taille de fenêtre V=3. Variez la valeur de V afin de déterminer le point d'allocation optimal. Que survient-il si l'on alloue plus de cadres au processus que son niveau optimal?
5. On considère un ordinateur dont le système de mémoire virtuelle dispose de 4 cases (frames ou cadres) de mémoire physique pour un espace virtuel de 8 pages. On suppose que les quatre cases sont initialement vides et que les pages sont appelées dans l'ordre suivant au cours de l'exécution d'un processus par le processeur : 1, 2, 3, 1, 7, 4, 1, 8, 2, 7, 8, 4, 3, 8, 1, 1. Indiquez tout au long de la séquence d'exécution quelles pages sont présentes dans les cases de la mémoire physique

et le nombre de défauts de page selon que l'algorithme de remplacement de pages est :

(a) PAPS.

(b) L'algorithme de remplacement optimal.

6. Supposez un système de mémoire paginée ayant 2g+h adresses virtuelles et 2h+k adresses en mémoire principale, d'où g, h, et k sont des nombres entiers. Spécifiez la taille des blocs et le nombre de bits pour les adresses virtuelle et physique.
7. Supposez une taille de blocs de 1 Ko, déterminez le numéro de page et le numéro de ligne pour chacune des adresses virtuelles suivantes.

(a) 950

(b) 1851

(c) 25354

(d) 11842

11
Système de fichiers

UN fichier désigne un ensemble d'informations stockées sur le disque. Le **système de fichiers** est la partie du système d'exploitation qui se charge de gérer les fichiers. La gestion consiste en la création (identification, allocation d'espace sur disque), la suppression, les accès en lecture et en écriture, le partage de fichiers et leur protection en contrôlant les accès.

11.1 Introduction

11.1.1 Qu'est ce qu'un fichier?

Pour le système d'exploitation, un fichier est une suite d'octets. Par contre, les utilisateurs peuvent donner des significations différentes au contenu d'un fichier (suites d'octets, suite d'enregistrements, arbre, etc.). Chaque fichier est identifié par un **nom** auquel on associe un emplacement sur le disque (une référence) et possède un ensemble de propriétés : ses **attributs**.

Le nom est en général, composé de deux parties séparées par un point. La partie qui suit le point est appelée **extension** (`prog.c, entete.h, fich.doc, archivo.pdf`, etc.). L'extension peut être de taille fixe, comme dans MS-DOS ou variable comme c'est le cas d'Unix / Linux ; obligatoire ou non. L'extension est nécessaire dans certains cas. Par exemple, le compilateur *C* rejettera le fichier `prog.txt` même si son contenu est un programme *C*. Le nom d'un fichier peut être sensible à la typographie : ainsi en Unix / Linux `Archivo` $\neq$ `archivo`.

11.1.2 Cycle de vie d'un fichier

Les fichiers — comme bien d'autres composants — ont un cycle de vie. Ils sont crées (ou ouverts), on les fait modifier (écrire), on lit à partir d'eux, et finalement, peut être, ils meurent (sont effacés). Ceci est illustré par la figure 11.1, avec des appels système qu'on étudiera dans le présent Chapitre.

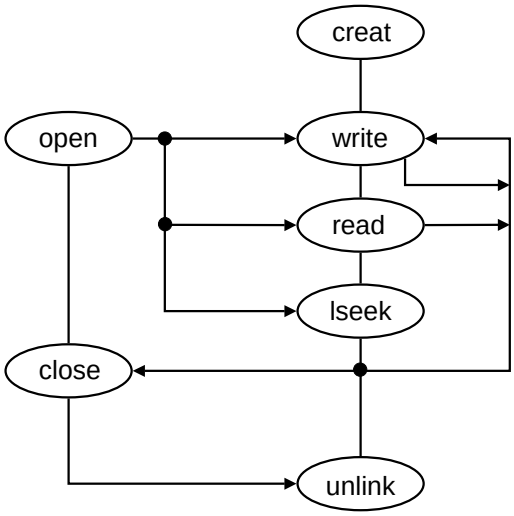


FIG. 11.1 – *Le cycle de vie d'un fichier.*

11.1.3 Types de fichiers

Dans un système, il existe plusieurs types de fichiers. Unix/Linux et MS-DOS ont des fichiers ordinaires, des répertoires, des fichiers spéciaux caractère et des fichiers spéciaux bloc.

Les **fichiers ordinaires** contiennent les informations des utilisateurs. Ils sont en général des fichiers ASCII ou binaires. Les **répertoires** sont des fichiers système qui maintiennent la structure du système de fichiers. Les **fichiers spéciaux caractère** sont liés aux Entrées/Sorties et permettent de modéliser les périphériques d'E/S série tels que les terminaux, les imprimantes et les réseaux. Les **fichiers spéciaux bloc** modélisent les disques.

Dans le système Unix/Linux, `"-"` désigne les fichiers ordinaires, `"d"` les répertoires, `"c"` les périphériques à caractères, `"b"` les périphériques à blocs, et `"p"` les tubes avec nom (*named pipe*). Les périphériques sont des fichiers désignés par des références.

Les **fichiers spéciaux bloc** et **caractère** vont identifier les dispositifs physiques du matériel : les disques, les rubans magnétiques, les terminaux, les réseaux, etc. Chaque type de dispositif a un contrôleur responsable de sa communication. Dans le système d'exploitation il y a une table qui pointe vers les différents contrôleurs de dispositifs. Tous les dispositifs seront alors traités comme de fichiers.

Dans Unix / Linux les périphériques comme les terminaux sont des fichiers spéciaux qui se trouvent sous le répertoire `/dev`. On copie par exemple le texte : `"abcd"` tapé au clavier, dans un fichier par la commande :

```
leibnitz> cat > fichier
 abcd
 ^D
```

et sur un terminal, par la commande :

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

11.2 Accès aux fichiers

Pour accéder à un fichier il faut fournir au système de fichiers les informations nécessaires pour le localiser sur le disque, c'est-à-dire lui fournir un **chemin d'accès**. Les systèmes modernes permettent aux utilisateurs d'accéder directement à une donnée d'un fichier, sans le parcourir depuis le début du chemin.

11.2.1 Attributs des fichiers

Les attributs des fichiers diffèrent d'un système à un autre. Cependant ils peuvent être regroupés en deux catégories :

1. Les attributs qui servent à **contrôler les accès** comme le code de protection, le mot de passe, le propriétaire, etc.
2. Les attributs qui définissent le **type** et l'**état courant** du fichier : indicateur du type ASCII / binaire, taille courante, date de création, date de la dernière modification, etc.

11.2.2 i-nœuds

Dans le système Unix/Linux toutes les informations des fichiers sont rassemblées dans une structure associée au fichier, appelée **nœud d'information**, **i-nœud** ou **i-node**. L'i-nœud contient les informations suivantes : le type du fichier (régulier, répertoire, caractère, bloc ou tube) ; le code de protection sur 9 bits ; l'identificateur et groupe du propriétaire ; les dates de création, du dernier accès et de la dernière modification ; le compteur de références ; la taille et finalement la table d'index composée de 13 numéros de blocs et de pointeurs. La figure 11.2 montre la structure typique d'un i-nœud.

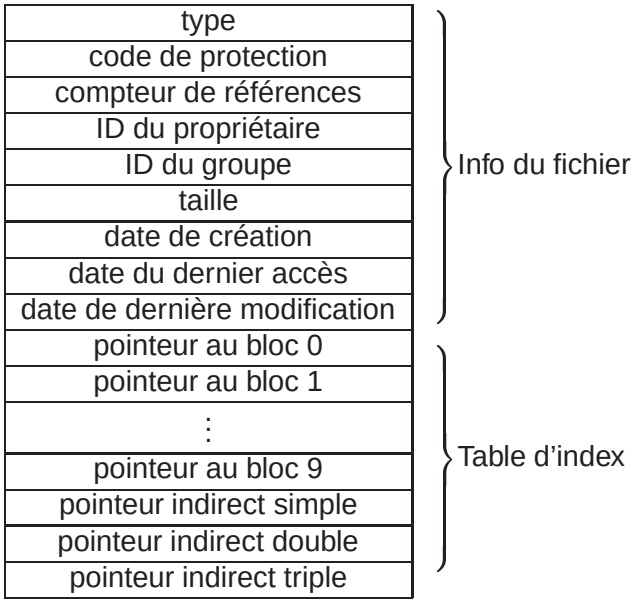


FIG. 11.2 – *Structure d'un i-nœud.*

11.3 Services Posix sur les fichiers

Les fichiers permettent de stocker des informations et de les rechercher ultérieurement. Les systèmes fournissent un ensemble d'appels système relatifs aux fichiers. Dans le cas d'Unix/Linux, les principaux appels système Posix relatifs aux fichiers sont :

– `open()` et `creat()` pour l'ouverture d'un fichier.
– `close()` pour la fermeture d'un fichier.

- `read()` pour la lecture d'un fichier.
- `write()` pour l'écriture dans un fichier.
- `lseek()` pour déplacer le pointeur de fichier.
- `stat()` pour récupérer des informations d'un fichier.
- `link()` pour créer un lien entre deux fichiers.
- `unlink()` pour supprimer un lien ou un fichier.

Ouverture d'un fichier

```
#include <unistd.h>
#include <fcntl.h>
int open(char * filename, int mode);
int open(char *filename, int mode, int permission);
int creat(char * filename, int permission);
```

En cas de succès, `open()` retourne un descripteur du fichier. En cas d'échec, il retourne `0`. Chaque fichier ouvert a un pointeur qui pointe sur un élément du fichier. Ce pointeur sert à parcourir le fichier. A l'ouverture du fichier, le pointeur de fichier pointe sur le premier élément du fichier, sauf si l'option `O_APPEND` a été spécifiée. Les lectures et les écritures se font à partir de la position courante du pointeur. Chaque lecture et chaque écriture entraînent la modification de la position du pointeur.

L'appel système `creat()`[1] est de moins en moins utilisée. Son usage est équivalent à celui de :
`open(char *filename, O_WRONLY|O_CREAT|O_TRUNC, int mode)`
Le `mode` est une combinaison de plusieurs élements assemblés par un `OU` logique. Il faut utiliser de façon obligée l'une de trois constantes :

- `O_RDONLY` : fichier ouvert en lecture exclusive.
- `O_WRONLY` : fichier ouvert en écriture exclusive.
- `O_RDWR` : fichier ouvert en lecture et en écriture.

On peut ensuite utiliser d'autres constantes qui permettent de mieux préciser l'usage :

- `O_CREAT` : créer le fichier ouvert même en lecture exclusive.
- `O_EXCL` : employé conjointement avec `O_CREAT`, cette constante garantit qu'on n'écrasera pas un fichier déjà existent. L'ouverture échoue si le fichier existe déjà.

1. Bien qu'en anglais le mot correct soit *create*, Ken Thompson a décidé de la nommer `creat()`, à cause d'une raison inconnue.

– `O_TRUNC` : utilisée avec `O_WRONLY` ou avec `O_RDWR`, permet de ramener la taille d'un fichier existent à zéro.

Les **permissions** sont utilisées lors de la création d'un fichier. Elles servent à signaler les autorisations d'accès au nouveau fichier crée. On peut les fournir en représentation octale directement (précédées d'un `0`) ou bien on peut utiliser les constantes symboliques les plus fréquentes de la table 11.1.

Constante symbolique	Valeur octale	Signification de l'autorisation
`S_IRUSR`	`00400`	Lecture pour le propriétaire
`S_IWUSR`	`00040`	Écriture pour le propriétaire
`S_IXUSR`	`00100`	Exécution pour le propriétaire
`S_IRWXU`	`00700`	Lecture + écriture + exécution pour le propriétaire
`S_IRGRP`	`00020`	Lecture pour le groupe
`S_IROTH`	`00004`	Lecture pour tout le monde
`S_IWOTH`	`00002`	Écriture pour tout le monde
`S_IXOTH`	`00001`	Exécution pour tout le monde

TAB. 11.1 – *Constantes symboliques de permission.*

Ainsi si l'on utilise la combinaison :

`"S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH" = 0644 = rw-r-r-`

on donne les droits de lecture à tous et les droits de lecture et écriture seulement au propriétaire du fichier.

Exemple : Cet appel système crée, puis ouvre un fichier nommé `archivo` pour un accès en lecture / écriture. Le descripteur de fichier est `fd`. Le code de protection du fichier est `0600` ce qui correspond à `(wr-----)`.

```
int fd;
fd = open("archivo", O_CREAT | O_RDWR, 0600);
```

Exemple : Cet appel système ouvre le fichier `archivo` en mode écriture et positionne le pointeur à la fin du fichier. Les écritures dans le fichier ajouteront donc des données à la fin de celui-ci.

```
int fd;
fd = open("archivo", O_WRONLY | O_APPEND);
```

Exemple : Les appels suivants sont équivalents :

```
fd = creat("datos.txt", 0751);
fd = open("datos.txt", O_WRONLY | O_CREAT | O_TRUNC, 0751);
```

Fermeture des fichiers

L'appel système `close()` permet de libérer un descripteur de fichier `fd`. Le fichier pointé par le descripteur libéré est fermé s'il n'a plus aucun descripteur associé au fichier.

```
int close (int fd);
```

Lecture d'un fichier

```
#include <unistd.h>
int read(int fd, char* buf, int count);
```

L'appel système `read()` lit, à partir de la position courante du pointeur, `count` octets au maximum et les copie dans le tampon `buf`. En cas de succès, il retourne le nombre d'octets réellement lus. L'appel système `read()` retourne `0` si la fin de fichier est atteinte. Il retourne `-1`, en cas d'erreur.

Écriture dans un fichier

```
#include <unistd.h>
int write(int fd, void* buf, int count);
```

L'appel système `write()` copie `count` octets du tampon `buf` vers le fichier `fd`, à partir de la position pointée par le pointeur de fichier. En cas de succès, il retourne le nombre d'octets réellement écrits. Sinon, il renvoie `-1`.

Il faut retenir que `read()` et `write()` manipulent des octets, et que le programmeur a la responsabilité de les interpréter d'une ou autre forme. Par exemple, regardez les lectures de types différents de données à partir d'un descripteur du fichier `fd` (les écritures seraient tout à fait équivalentes):

```
char c, s[N];
int i, v[N];

// 1 char :
   read(fd, &c, sizeof(char));
// N char's:
   read(fd, s, N*sizeof(char));
// 1 int :
   read(fd, &i, sizeof(int));
// N int's :
   read(fd, v, N*sizeof(int));
```

▶ **Exemple 1.** Le programme `copie-std.c` crée un fichier appelé "`fichier`" dans lequel il copie les données lues à partir du clavier. Les Entrées/Sorties sont effectuées avec des appels système (de bas niveau) sans utiliser la bibliothèque standard de *C* "`stdio.h`".

Listing 11.1: *copie-std.c*

```
#include <unistd.h>
#include <fcntl.h>

#define taille 80

int main()
{
    int fd , nbcar;
    char buf[ taille ];

    // créer un fichier
    fd = open("fichier", O_CREAT| O_WRONLY);
    if(fd==-1)
    {
        write(2,"Erreur d'ouverture\n", 25);
        return -1;
    }
    write(1, "Ouverture avec succès\n" , 30);
    // copier les données introduites à partir
    // du clavier dans le fichier
    while ((nbcar = read(0, buf, taille )) > 0)
        if ( write(fd , buf,nbcar) == -1) return -1;
    return 0;
}
```

▶ **Exemple 2.** Le programme `copie.c`, copie un fichier dans un autre.

Listing 11.2: *copie.c*

```
#include <sys/wait.h> // wait
#include <fcntl.h>    // les modes
#include <unistd.h>   // les appels système
```

```
#define TAILLE 4096
int main (int argc, char*argv[])
{
   int status, src, dst, in, out;
   char buf[TAILLE];
   if ( fork () == 0)
   {
      if (argc !=3) exit (1);
      src = open(argv[1], O_RDONLY);
      if (src < 0) exit (2);
      dst = open( argv[2], O_RDWR | O_CREAT, 0666);
      if(dst<0) exit (3);
      while (1)
      {
         in = read(src, buf, TAILLE);
         if (in<0) exit (4);
         if(in==0) break;
         out = write(dst, buf, in);
         if ( out<=0) exit (5);
      }
      close (src);
      close(dst);
      exit (0);
   }
   else
   {
      if (wait(&status)>0 && (status»8)==0)
         write(1,"done\n",5);
      else write(2, "error\n",6);
      return 0;
   }
}
```

L'exécution de `copie.c` montre qu'on peut copier n'importe quel type de fichiers, même des fichiers exécutables. Cependant cette version du programme a écrasé le mode (`rwxr-xr-x` a été transformé en `rw-r-r-`):

```
leibnitz> ls -l stat*
-rwxr-xr-x    1 jmtorres prof    15926 Nov 15 14:53 stat*
leibnitz> copie stat stat2
done
```

```
leibnitz> ls -l stat*
-rwxr-xr-x    1 jmtorres prof    15926 Nov 15 14:53 stat*
-rw-r--r--    1 jmtorres prof    15926 Nov 15 15:12 stat2
leibnitz>
```

Accès direct à une position dans le fichier

```
int lseek(int fd, int offset, int origine);
```

L'appel système `lseek()` permet de modifier la position courante du pointeur de parcours du fichier. Il retourne la nouvelle position du pointeur ou `-1` en cas d'erreur. Le positionnement dans un descripteur est mémorisé dans la Table de fichiers et non dans la Table de descripteurs. Le deuxième argument `offset` fournit la nouvelle valeur de la position. Le troisième argument `origine` peut prendre comme valeurs :

- `SEEK_SET` : Position = `offset`.
- `SEEK_CUR` : Position courante + `offset`.
- `SEEK_END` : Taille du fichier + `offset`.

Observez le diagramme de la figure 11.3.

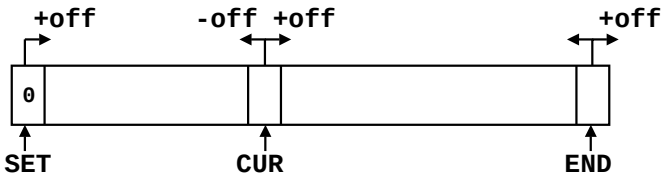


FIG. 11.3 – *Pointeur de fichier : `lseek()`.*

Ainsi, pour connaître la position courante on peut utiliser simplement : `lseek(fd,0,SEEK_CUR)`.

▶ **Exemple 3.** Soit le fichier `archivo` contenant la chaîne : "`le printemps`". A l'ouverture de ce fichier, le pointeur de fichier est positionné sur le 1er caractère "`l`". Sa valeur est `0`.

```
char buf ;
 //déplacer le pointeur du fichier à la position 3 du fichier
 lseek(fd, 3,SEEK_SET);

 // la valeur du pointeur est 3.
 // lire un caractère dans buf (buf reçoit p)
```

```
read(fd, buf, 1);

// positionner le pointeur deux caractères plus loin
lseek(fd,2,SEEK_CUR);

// la valeur du pointeur est 4+2.
// lire un caractère dans buf (buf reçoit n)
read(fd,buf,1);

//positionner le pointeur à la fin du fichier
lseek(fd,0,SEEK_END);
read(fd,buf,1) ; // retournera 0
```

▶ **Exemple 4.** Accès directs aux éléments d'un fichier avec `seek.c` :

Listing 11.3: *seek.c*

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main (int argc, char* argv [])
{
    int PLine[200];
    int fd, ncar, i;
    int  nline=0, pfich=0;
    char buffer[4096];

    // ouverture du fichier en lecture seulement
    fd = open (argv[1], O_RDONLY);
    if ( fd==-1) exit (1); // erreur d'ouverture
    PLine[0]=0; // position de la ligne 0
    while (1)
    {   // lecture du fichier
        ncar = read(fd, buffer, 4096);
        if (ncar == 0) break; // fin de fichier
        if (ncar ==-1) exit (1); // erreur de lecture
        for ( i=0; i<ncar; i++)
        {
            pfich++;
            // fin de ligne rencontrée
```

```
            if(buffer[i]=='\n') PLine[++nline]=pfich;
          }
    }
    PLine[nline+1]=pfich;
    for(i=0; i<=nline; i++)
30        printf(" PLine[%d] = %d \n",i,PLine[i]);
    // accès à la première ligne en utilisant lseek.
    lseek(fd, PLine[0], SEEK_SET);
    ncar = read (fd, buffer, PLine[1]-PLine[0]);
    // afficher à l'écran le contenu du buffer
    write(1, buffer, ncar);
    return 0;
}
```

Exécution de `seek.c` :

```
leibnitz> cat >> exemple
nom
prenom
code
leibnitz> cat exemple
nom
prenom
code
leibnitz> seek exemple
PLine[0] = 0
PLine[1] = 4
PLine[2] = 11
PLine[3] = 16
nom
leibnitz>
```

Duplication des descripteurs de fichier

Chaque processus a une table de descripteurs de fichier. Cette table contient les descripteurs de fichier du processus. Chaque descripteur pointe vers un fichier ouvert pour le processus. Lorsqu'un processus est créé, le système lui ouvre automatiquement trois fichiers :

- L'entrée standard auquel il associe le descripteur `0`.
- La sortie standard auquel il associe le descripteur `1`.
- La sortie standard auquel il associe le descripteur `2`.

Si le processus ouvre explicitement un fichier, le descripteur associé au fichier serait 3 et ainsi de suite.

```
int dup (int oldfd);
int dup2(int oldfd, int newfd);
```

L'appel système `dup()` permet d'associer à un même fichier plusieurs descripteurs. `dup()` associe le plus petit descripteur libre au fichier pointé par le descripteur `oldfd`. L'appel système `dup2()` associe le descripteur `newfd` au fichier pointé par `oldfd`.

▶ **Exemple 5.** Duplication de descripteurs de fichier :

Listing 11.4: *dup.c*

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
   int fd1,
       fd2,
       fd3;

   fd1 = open ("test", O_CREAT | O_RDWR, 0644);
   if(fd1==-1)
     {
       perror("open");
       exit (1);
     }
   printf("fd1 = %d\n", fd1);
   write(fd1, "hello ",6);
   fd2 = dup(fd1); // fd2 et fd1 sont associés au fichier test
   printf("fd2 = %d\n", fd2);
   write(fd2, "wor",3);
   close (0);
   fd3 = dup(fd1); // fd3 et fd1 sont associés au fichier test
   printf("fd3 = %d\n", fd3);
   write (0, "ld",2);
   dup2 (3,2);
   write (2, "!\n", 2);
```

```
    return 0;
}
```

Exécution de `dup.c` :

```
leibnitz> gcc -o dup dup.c
leibnitz> dup
fd1 = 3
fd2 = 4
fd3 = 0
leibnitz> ls -l test
-rw-r--r--    1 jmtorres prof   13   Nov 15 16:03 test
leibnitz> more test
hello world!
leibnitz>
```

Il faut retenir que si deux descripteurs de fichiers sont associés à la même entrée de la table, le déplacement du pointeur avec `lseek()` (ou bien avec `read/write`) affectera l'autre.

▶ **Exemple 6.** Observez le petit programme `lseek.c` :

Listing 11.5: *lseek.c*

```
#include <unistd.h>

int main(void)
{
    int fd;

    if ((fd = open("file", O_RDWR)) < 0)
        exit (1);
    if (lseek(fd, 10, SEEK_SET) < 0)
        exit (1);
    if (write(fd, "123", 3) < 0)
        exit (1);
    return 0;
}
```

Sans surprise, l'exécution sur un fichier `file` :

```
pascal> cat file
ABCDEFGHIJKLMNOPQRSTUVWXYZ
pascal> lseek
```

```
pascal> cat file
ABCDEFGHIJ123NOPQRSTUVWXYZ
pascal>
```

▶ **Exemple 7.** Mais dans cet autre programme `dup-lseek.c`, deux descripteurs différents avec la même entrée de la table de fichiers, vont modifier le pointeur du parcours du fichier :

Listing 11.6: *dup-lseek.c*

```
#include <unistd.h>

int main(void)
{
  int fd;

  if ((fd = open("file", O_RDWR)) < 0)
      exit (1);
  dup2(fd, 1);
  if (lseek(fd, 10, SEEK_SET) < 0)
      exit (1);
  if (lseek (1, -5, SEEK_CUR) < 0)
      exit (1);
  if (write(fd,"123", 3) < 0)
      exit (1);
  return 0;
}
```

Exécution de `dup-lseek.c` sur un fichier `file` :

```
pascal> cat file
ABCDEFGHIJKLMNOPQRSTUVWXYZ
pascal> dup-lseek
pascal> cat file
ABCDE123IJKLMNOPQRSTUVWXYZ
pascal>
```

Le premier `lseek` met le pointeur à la position absolue `10`, mais le deuxième `lseek` (malgré l'utilisation d'autre `fd`, utilise la même entrée de la table) le fait reculer `5` octets.

▶ **Exemple 8.** Finalement, regardez l'exemple `dup3.c` suivant, qui ouvre le même fichier `file` deux fois :

Listing 11.7: *dup3.c*

```
#include <unistd.h>

void main(void)
{
 int fd1, fd2;

 if ((fd1 = open("file", O_RDWR)) < 0)
        exit (1);
 if ((fd2 = open("file", O_WRONLY)) < 0)
        exit (1);
 if (lseek(fd1 , 10, SEEK_SET) < 0)
        exit (1);
 if (write(fd1, "456" , 3) < 0)
        exit (1);
 if (lseek(fd2 , 5, SEEK_CUR) < 0)
        exit (1);
 if (write(fd2, "123" , 3) < 0)
        exit (1);
 return 0;
}
```

et sa sortie :

```
pascal> cat file
ABCDEFGHIJKLMNOPQRSTUVWXYZ
pascal> dup3
pascal> cat file
ABCDE123IJ456NOPQRSTUVWXYZ
pascal>
```

Pourquoi ce comportement ? Souvenez vous que deux ouvertures *du même fichier* fournissent *deux descripteurs de fichiers*, avec des entrées indépendantes dans la Table de fichiers.

Obtention des informations d'un fichier

```
int stat(char* filename, struct stat * buf);
```

L'appel système `stat()` permet de récupérer des informations sur le fichier `filename`. En cas de succès, l'appel système `stat()` retourne `0` et les informations sur le fichier sont récupérées dans `buf`. En cas d'erreur, il retourne `-1`. `stat()` stocke l'information recuperée dans une structure du type `stat` :

```
struct stat {
   mode_t  st_mode;  /* le code de protection */
   ino_t   st_ino;   /* numéro du i-noeud du fichier */
   dev_t   st_dev;   /* dispositif */
   nlink_t st_nlink; /* nombre de liens physiques */
   uid_t   st_uid;   /* UID du propriétaire */
   gid_t   st_gid;   /* GID du groupe */
   off_t   st_size;  /* la taille du fichier (octets) */
   time_t  st_atime; /* dernier accès */
   time_t  st_mtime; /* la date de la dernière modification */
   time_t  st_ctime; /* dernière modification de données */
};
```

Ainsi `buf.st_ino` indique par exemple le numéro du i-nœud du fichier stocké dans `buf`. Il y a des macros prédéfinis qui prennent `buf.st_mode` comme argument et retournent `1` ou `0` :

- `S_ISDIR(buf.st_mode)` : vrai si le fichier est un répertoire.
- `S_ISCHAR(buf.st_mode)` : vrai si c'est un fichier spécial caractère.
- `S_ISBLK(buf.st_mode)` : vrai si c'est un fichier spécial bloc.
- `S_ISREG(buf.st_mode)` : vrai si c'est un fichier régulier.
- `S_ISFIFO(buf.st_mode)` : vrai si c'est un `pipe` ou `FIFO`.

▶ **Exemple 9.** Récupérer des informations à partir d'un fichier.

Listing 11.8: *stat.c*

```
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char* argv[])
{
   struct stat buf;
   mode_t mode;
   int result;
```

```
    result = stat(argv[1], &buf);
    if (result == -1)
        printf("Infos sur %s non disponibles\n", argv[1]);
    else
    {
        mode = buf.st_mode;
        if (S_ISDIR(mode))
            printf("%s est un répertoire", argv[1]);
        else
        if (S_ISREG(mode))
            printf("%s est un fichier ordinaire", argv[1]);
        else
        if (S_ISFIFO(mode))
            printf("%s est un pipe", argv[1]);
        else
            printf("%s est un fichier special ");
        printf("\nTaille du fichier : %d\n", buf.st_size);
    }
    return 0;
}
```

Exécution de `stat.c` :

```
leibnitz> stat
Infos sur (null) non disponibles
leibnitz> stat stat.c
stat.c est un fichier ordinaire
Taille du fichier : 775
leibnitz>
```

Le programme suivant `estado.c` montre toutes les caractéristiques d'une liste de fichiers. Observez l'utilisation des fonctions `getpwuid()` et `getgrgid()` pour l'obtention de l'`UID` et du groupe de l'utilisateur.

▶ **Exemple 10.** Caractéristiques d'une liste de fichiers :

Listing 11.9: *estado.c*

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<pwd.h>
```

```
#include<grp.h>

char permisos[]={'x','w','r'};
void estado (char *archivo);

int main (int argc, char **argv)
{
  int i;
  if(argc != 2)
  {
      printf("Usage : %s nombre_archivo\n", argv[0]);
      exit (1);
  }
  for (i=1; i<argc; i++)
     estado(argv[i]);
  return 0;
}

void estado (char *archivo)
{
  struct stat buf;
  struct passwd *pw;
  struct group *gr;
  int i;

  stat(archivo,&buf);
  printf("Fichier              : %s\n",archivo);
  printf("Dispositivo : %d, %d\n",
        (buf.st_dev & 0xff00 ) » 8,  buf.st_dev & 0x00ff);
  printf("i-noeud              : %d\n", buf.st_ino);
  printf("Type                 : " );
  switch(buf.st_mode & S_IFMT)
  {
     case S_IFREG : printf("régulier\n");
        break;
     case S_IFDIR : printf("répertoire\n");
        break;
     case S_IFCHR : printf("spécial caractere\n");
        break;
     case S_IFBLK : printf("spécial bloc\n");
```

```
            break;
        case S_IFIFO : printf("FIFO\n");
            break;
    }
    printf("Permissions       : 0%o," ,buf.st_mode & 0777);
    for(i=0; i<9; i++)
        if(buf.st_mode & (0400 >> i))
            printf("%c", permisos[(8-i)%3]);
        else printf("-");
    printf("\n");
    printf("Liens             : %d\n",buf.st_nlink);
    printf("UID               : %d\n", buf.st_uid);
    printf("Nom               : " );
    pw = getpwuid(buf.st_uid);
    printf("%s\n", pw->pw_name);
    printf("GID               : %d\n", buf.st_gid);
    printf("Nom               : " );
    gr = getgrgid(buf.st_gid);
    printf("%s\n", gr->gr_name);
    switch(buf.st_mode & S_IFMT)
    {
        case S_IFCHR :
        case S_IFBLK : printf("Dispositif    : %d, %d\n",
                (buf.st_rdev & 0xff00 ) >> 8,  buf.st_rdev & 0x00ff);
    }
    printf("Longueur          : %d bytes.\n",buf.st_size );
    printf("Dernier accès     : %s",
        asctime (localtime(&buf.st_atime)));
    printf("Dernière modif.   : %s",
        asctime (localtime(&buf.st_mtime)));
    printf("Dernier ch. d'état: %s",
        asctime (localtime(&buf.st_atime)));
}
```

Exécution de `estado.c` :

```
leibnitz> gcc -o estado estado.c
leibnitz> estado estado
Fichier           : estado
Dispositivo       : 0, 0
i-noeud           : 1854250
```

```
Type              : régulier
Permissions       : 0755, rwxr-xr-x
Liens             : 1
UID               : 11047
Nom               : jmtorres
GID               : 100
Nom               : prof
Longueur          : 17555 bytes.
Dernier accFs     : Tue Dec  3 15:03:24 2002
Dernière modif.   : Tue Dec  3 15:03:24 2002
Dernier ch. d'état: Tue Dec  3 15:03:24 2002
leibnitz>
```

Effacer un fichier

```
#include <unistd.h>
int unlink(char* nom_fichier);
```

L'appel système `unlink()` permet d'effacer un fichier. Il supprime le lien entre `nom_fichier` et l'i-nœud correspondant. Il retourne `1` si le fichier a été bien effacé ou `-1` s'il y a eu une erreur.

L'exemple suivant permet de constater que même si l'on efface un fichier, son contenu est encore accessible tant qu'il ne soit pas fermé.

▶ **Exemple 11.** Effacer (et encore re-utiliser) un fichier.

Listing 11.10: *unlink-cs.c*

```
#include <unistd.h>
#include <fcntl.h>
#define N 16

int main (void)
 {
  char chaine[N];
  int fp;

  write(1, "Création fichier \n",17);
  fp = open("test", O_RDWR | O_CREAT);
  if (fp<0)
        exit (0);
  write (fp, "0123456789abcdef", N);
```

```
  system ("ls -l test");
  write (1, "Effacement fichier\n",19);
  if (unlink("test") == -1)
  {
      perror ("unlink");
      exit (1);
  }
  system ("ls -l test");
  write (1, "Relecture du contenu du fichier\n",32);
  lseek(fp,0,SEEK_SET);
  read(fp, chaine, N);
  write(1, chaine, N);
  write(1, "Fermeture fichier\n",18);
  close (fp);
  return 0;
}
```

Exécution de `unlink-cs.c`:

```
leibnitz> gcc -o unlink unlink-cs.c
leibnitz> unlink
Création fichier
-rw-r--r--    1 jmtorres prof 26 Nov  1 17:17 test
Effacement fichier
ls: test: No such file or directory
Relecture du contenu du fichier
Lu : ABCDEFGHIJKLMNOPQRSTUVWXYZ
Fermeture fichier
leibnitz>
```

11.3.1 Transfert rapide de données

La forme classique de copier un fichier dans un autre, consiste en allouer un *buffer* de taille fixe, copier ensuite quelques données dans le *buffer*, puis écrire le *buffer* dans un autre descripteur et répéter jusqu'à ce que toutes les données soient copiées. Ceci n'est pas efficace ni en temps ni en espace car on utilise une mémoire additionnelle (le *buffer*) et on exécute une copie additionnelle des données. L'appel système `sendfile()` fournit un mécanisme efficient pour copier fichiers où le *buffer* est éliminé.

```
#include <sys/sendfile.h>
sendfile(write_fd, read_fd, &offset, stat_buf.st_size);
```

Les descripteurs peuvent être des fichiers, des **sockets** [2] ou d'autres dispositifs. Les paramètres sont `write_fd` le descripteur du fichier à écrire, `read_fd` le descripteur de lecture, `offset` un pointeur vers un offset dynamique et le nombre d'octets à copier `st_size`. On peut utiliser `fstat()` pour déterminer cette taille. `offset` contient le décalage (`0` indique le début du fichier). La valeur retournée est le nombre d'octets copiés.

▶ **Exemple 12.** Le programme `fcopie.c` montre cette technique efficace :

Listing 11.11: *fcopie.c*

```
#include <fcntl.h>
#include <sys/sendfile.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
 int fd_r;
 int fd_w;
 struct stat buf;
 off_t offset = 0;

 // fd source
 fd_r = open(argv[1], O_RDONLY);
 // obtenir sa taille
 fstat (fd_r, &buf);
 // fd de sortie avec les mêmes permissions du fd source
 fd_w = open(argv[2], O_WRONLY | O_CREAT, buf.st_mode);
 // Copier
 sendfile(fd_w, read_fd, &offset, buf.st_size );
 close(fd_r);
 close(fd_w);
 return 0;
}
```

2. Voir Chapitre 14 *Introduction aux systèmes distribués*, Section 14.5 *Sockets*.

11.4 Répertoires

Les systèmes d'exploitation modernes adoptent une **structure arborescente** pour représenter le système de fichiers. Les nœuds de l'arbre sont des répertoires et les feuilles sont des fichiers. Un répertoire est composé de fichiers et de sous répertoires. Pour accéder à un fichier, il suffit de spécifier les répertoires du chemin qui mène de la racine de l'arborescence au fichier (chemin d'accès). Dans le système Unix / Linux, chaque répertoire contient aussi sa propre référence `"."` et celle du répertoire immédiatement supérieur `".."`. Dans la figure 11.2 on montre un répertoire d'Unix.

i-nœud	Nom du fichier
`20`	`.`
`3`	`..`
`100`	`chap1`
`2378`	`chap2`
`125`	`scheduler`
⋮	⋮

TAB. 11.2 – *Répertoire Unix.*

Un répertoire est aussi considéré comme un fichier particulier composé de fichiers [3].

11.5 Services Posix sur les répertoires

Les répertoires d'Unix / Linux possèdent une entrée par fichier. Chaque entrée possède le nom du fichier et le numéro de l'i-nœud. Le système

3. Les répertoires de MS-DOS possèdent une entrée par fichier. Chaque entrée a la structure suivante :

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

Unix / Linux offre des appels système Posix pour manipuler les répertoires :

- `mkdir()` créer un répertoire.
- `rmdir()` supprimer un répertoire vide.
- `opendir()` ouvrir un répertoire.
- `closedir()` fermer un répertoire.
- `readdir()` lire les entrées d'un répertoire.
- `rewindir()` placer le pointeur d'un répertoire.
- `link()` créer une entrée dans un répertoire.
- `unlink()` effacer une entrée d'un répertoire.
- `chdir()` changer de répertoire de travail.
- `rename()` renommer un répertoire.
- `getcwd()` obtenir le nom du répertoire actuel.

Création d'un répertoire

```
#include <sys/types.h>
#include <dirent.h>
int mkdir(const char *name, mode_t mode);
```

L'appel système `mkdir()` crée un répertoire avec le `UID` du propriétaire = `UID` effectif et le `GID` du propriétaire = `GID` effectif. Comme arguments il reçoit `name` qui est le nom du répertoire, et `mode` qui sont les bits de protection. Il retourne `0` ou `-1` en cas d'erreur.

Effacer un répertoire

```
#include <sys/types.h>
int rmdir(const char *nom);
```

`rmdir()` efface le répertoire s'il est vide. Si le répertoire n'est pas vide, on ne l'efface pas. Arguments : `nom` correspond au nom du répertoire. Il retourne `0` ou `-1` en cas d'erreur.

Ouverture d'un répertoire

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(char *nom);
```

`opendir()` ouvre un répertoire comme une séquence d'entrées. Il met le pointeur au premier élément. `opendir()` reçoit comme argument `nom` qui est le nom du répertoire. Il retourne un pointeur du type `DIR` pour l'utiliser en `readdir()` ou `closedir()` ou bien `NULL` s'il a eu une erreur.

Fermer un répertoire

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

L'appel système `closedir()` ferme le lien entre `dirp` et la séquence d'entrées du répertoire. Arguments : `dirp` pointeur retourné par `opendir()`. Il retourne `0` ou `-1` en cas d'erreur.

Lecture des entrées du répertoire

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

L'appel système `readdir()` reçoit comme arguments `dirp` qui est le pointeur retourné par `opendir()`. Il retourne un pointeur à un objet du type `struct dirent` qui représente une entrée de répertoire ou `NULL` s'il y a eu erreur. Il retourne l'entrée suivante du répertoire associé à `dirp` et avance le pointeur à l'entrée suivante. La structure est dépendante de l'implantation. Pour simplification, on peut assumer qu'on obtient un membre `char *d_name`.

Changer du répertoire

```
int chdir(char *name);
```

`chdir()` modifie le répertoire actuel, à partir duquel se forment les noms relatifs. Arguments : `name` est le nom d'un répertoire. Il retourne `0` ou `-1` en cas d'erreur.

Renommer un répertoire

```
#include <unistd.h>
int rename(char *old, char *new);
```

`rename()` change le nom du répertoire `old`. Le nom nouveau est `new`. Arguments : `old` nom d'un répertoire existant, et `new` le nom nouveau du répertoire. Il retourne `0` ou `-1` en cas d'erreur.

▶ **Exemple 13.** Parcours d'un répertoire avec `liste-dir.c` :

Listing 11.12: *liste-dir.c*

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

#define MAX_BUF 256

int main(int argc, char **argv)
{
    DIR *dirp;
    struct dirent *dp;
    char buf[MAX_BUF];

    // montrer le répertoire actuel
    getcwd(buf, MAX_BUF);
    printf("Répertoire actuel : %s\n", buf);
    // ouvre le répertoire passé comme argument
    dirp = opendir(argv[1]);
    if (dirp == NULL)
    {
        printf("Ne peut pas l'ouvrir %s\n", argv[1]);
    }
    else
    {
        // lit entrée à entrée
        while ((dp = readdir(dirp)) != NULL)
            printf("%s\n", dp->d_name);
        closedir(dirp);
    }
    exit (0);
}
```

Exécution de `liste-dir.c`:

```
leibnitz> pwd
/home/ada/users/jmtorres/inf3600/logiciel
leibnitz> ls
bin/    chap2/  chap4/  chap7/  examen/    scheduler/  tmp/
chap1/  chap3/  chap5/  chap9/  parallel/  sockets/
leibnitz> chap9/liste-dir ../logiciel
Répertoire actuel : /home/ada/users/jmtorres/inf3600/logiciel
```

```
.
..
chap1
chap2
chap3
scheduler
bin
tmp
parallel
chap4
sockets
chap5
chap7
examen
chap9
leibnitz>
```

Le programme précèdent ne montre pas le contenu des sous-répertoires qu'il trouve. Pour cela il nous faut des appels récursifs du même programme. Ceci nous le laissons comme exercice.

11.6 Répertoires sur Unix System V

En Unix System V l'appel système `getdents()` permet de lire une entrée du répertoire courant et insère dans `buf` les informations concernant cette entrée.

```
#include <sys/dirent.h>
int getdents (int fd, struct dirent* buf, int size)
```

La structure `dirent` est composée des champs suivants :

- `buf.d_ino` c'est le numéro du i-nœud.
- `buf.d_off` c'est la position relative de la prochaine entrée du répertoire.
- `buf.d_reclen` c'est la longueur de la structure d'une entrée du répertoire.
- `d_name` c'est le nom du fichier.

▶ **Exemple 14.** Parcours d'un répertoire avec `getd.c` :

Listing 11.13: *getd.c*

```
#include <stdio.h>       // printf
#include <string.h>      // strcmp
#include <fcntl.h>       // O_RDONLY
#include <dirent.h>      // getdents
#include <sys/types.h>   // mode_t
#include <sys/stat.h>    // les macros S_ISDIR
#include <unistd.h>      // open ...

int main(int argc, char* argv[])
{
int fd, nbcar, result;
struct stat buf;
mode_t mode;
struct dirent entree;
result = stat(argv[1], &buf);
if (result == -1)
{
    printf("les infos sur %s non disponibles \n", argv[1]);
    exit(1);
}
mode = buf.st_mode;
if (S_ISDIR(mode))
    printf("%s est un répertoire\n", argv[1]);
else
{
    printf("%s n'est pas un répertoire\n", argv[1]);
    exit(1);
}
fd = open(argv[1], O_RDONLY);
if (fd == -1)
{
    printf("erreur d'ouverture du rép. %s\n", argv[1]);
    exit(1);
}
lseek(fd,0,SEEK_SET);
while (1)
{
    nbcar = getdents(fd, &entree, 40);
```

```
    if (nbcar == -1)
    {
        printf("Erreur de lecture du rép. %s\n",argv[1]);
        exit (1);
    }
    if(nbcar==0)
    {
        printf("Fin de lecture du rép. %s\n",argv[1]);
        break;
    }
    if (strcmp(entree.d_name, ".") !=0 &&
        strcmp(entree.d_name, ".." ) !=0 )
        printf("%s\n", entree.d_name);
    lseek(fd,entree.d_off, SEEK_SET);
} // fin du while
close(fd);
printf("\nChangement de répertoire de travail\n");
chdir(argv[1]);
system("pwd");
printf("Contenu du nouveau répertoire de travail : \n");
system("ls");
exit (0);
}
```

Exécution de `getd.c` :

```
jupiter% gcc getd.c -o getd
jupiter% getd nouvrep
nouvrep est un répertoire
exemple
test
Fin de lecture du rép. nouvrep
Changement de répertoire de travail
/jupiter/home/bouchene/gestfichier/nouvrep
Contenu du nouveau répertoire de travail :
exemple test
```

11.7 Exercises

1. Expliquez la raison principale pour laquelle le système alloue une table de descripteur de fichier à chaque processus mais maintient qu'une seule table d'état de fichier pour tous les processus. Quelle information contient la table d'état de fichier?
2. Expliquez la raison pour laquelle tout nouveau fichier créé est affecté par un numéro de descripteur à partir de `3` et non `0`.
3. Expliquez la raison pour laquelle les descripteurs de fichier physique (sur disque) et logique (en mémoire) ne contiennent pas nécessairement la même information.
4. Expliquez le type d'information que contient le champ de compte de références pour un descripteur de fichier physique et aussi pour un descripteur de fichier logique.
5. Expliquez les étapes suivies et les structures de données utilisées lors d'une opération d'ouverture de fichier sous Unix/Linux.
6. Expliquez un avantage et un désavantage pour chacune des techniques d'allocation de blocs suivante :
   (a) Allocation contiguë de blocs.
   (b) Allocation de blocs linéairement chaînés.
   (c) Allocation de blocs par indexage.
7. Considérez un système d'allocation de blocs sous Unix/Linux ayant les caractéristiques suivantes.
   (a) Taille du bloc de 16 Ko.
   (b) Taille de pointeurs de blocs de 32 bits.

   Calculez la taille maximale d'un fichier. En pratique, il n'existe pas de fichiers d'application qui nécessiteraient la capacité calculée ci-dessus. Alors, donnez une raison pour laquelle le système maintient telle capacité maximale.
8. On considère un système disposant d'un système de fichiers similaire à celui d'Unix avec une taille de blocs de données de 4K (4096 octets) et des pointeurs (numéros de blocs) définis sur 4 octets. On supposera que le i-nœud de chaque fichier compte 12 pointeurs directs, 1 pointeur indirect simple, 1 pointeur indirect double et 1 pointeur indirect triple. On désire créer un fichier contenant un total de 20.000.000 (vingt millions) de caractères (caractères de fin de ligne et de fin de fichier compris). Quelle est la fragmentation interne totale sur le disque résultant de la création de ce fichier?

9. Réécrire le code du programme `unlink-cs.c` en utilisant un processus fils et `exec()` en lieu de la fonction `system()`.
10. Écrire un programme `rec-dir.c` afin de parcourir l'arbre des sous-répertoires de façon récursive.

12
Stockage et partage de fichiers

Les fichiers de données sur les disques se répartissent dans des blocs de taille fixe. La lecture ou l'écriture d'un élément d'un fichier impliquera donc le transfert du bloc entier qui contient cet élément. 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. Il faut implanter alors des techniques adéquates de stockage de fichiers. Il y a trois manières de faire l'allocation des fichiers : l'**allocation contiguë**, la **liste chaînée de blocs** et l'**indexation par i-nœuds**.

12.1 Stockage des fichiers

Des études sur de nombreux systèmes ont montré que la taille moyenne d'un fichier est de 1 Ko. En général, les tailles de blocs fréquemment utilisées sont de 512, 1024, ou 2048 octets. Chaque disque conserve, dans un ou plusieurs blocs spécifiques, un certain nombre d'informations telles que le nombre de ses blocs, leur taille, leurs états, entre autres. À chaque fichier correspond une liste de blocs contenant ses données. L'allocation est en général, 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** par i-nœuds.

12.1.1 Allocation contiguë

La table d'allocation de fichiers contient seulement une entrée par fichier, avec le bloc de début et la taille du fichier. L'**allocation contiguë** est

simple et performante, mais a le grave défaut de remplir rapidement l'espace avec des zones inutilisables (à cause de la fragmentation externe), et dans ce cas la compactation n'est pas envisageable avec des temps raisonnables. En plus il faudrait déclarer la taille du fichier au moment de sa création.

12.1.2 Liste chaînée

Les blocs d'un même fichier sont chaînés sous une représentation de **liste chaînée** montrée à la figure 12.1. Chaque bloc contiendra des données ainsi que l'adresse du bloc suivant. Le fichier doit mémoriser 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.

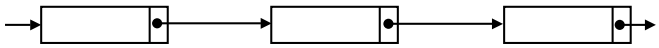


FIG. 12.1 – *Liste chaînée.*

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 une version améliorée 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**)[1]. Chaque disque dispose d'une table **FAT** et cette dernière possède autant d'entrées qu'il y a de blocs sur le disque. Chaque entrée de la **FAT** contient le numéro du bloc suivant. Par exemple, dans la table suivante :

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	E	EOF	EOF	L	...

"**x**" indique la taille du disque, "**L**" désigne un bloc libre et "**E**" un bloc endommagé. Le fichier commençant au bloc 6, sera constitué des blocs : $6 \rightarrow 8 \rightarrow 4 \rightarrow 2$. Le parcours de la FAT est nettement plus rapide que la chaîne de blocs. Cependant elle doit constamment être tout entière en mémoire, afin d'éviter les accès au disque pour localiser un bloc.

1. Il existe de **FAT**s de 16 et de 32 bits.

12.1.3 Table d'i-nœuds

Le système Unix associe à chaque fichier un numéro unique d'identification. A chaque numéro est associé un ensemble d'informations appelé **nœud d'information** ou **i-nœud**. Parmi les champs de l'i-nœud, la **table d'index** indique l'emplacement physique du fichier. Elle est composée de 13 entrées. Les 10 premières contiennent les numéros des 10 premiers blocs composant le fichier. Pour les fichiers de plus de 10 blocs, on a recours à des indirections. 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 s'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 un pointeur à double indirection, et le bloc n° 13 un pointeur à triple indirection. Sur la figure 12.2 nous montrons l'usage des indirections simples, doubles et triples d'un i-nœud.

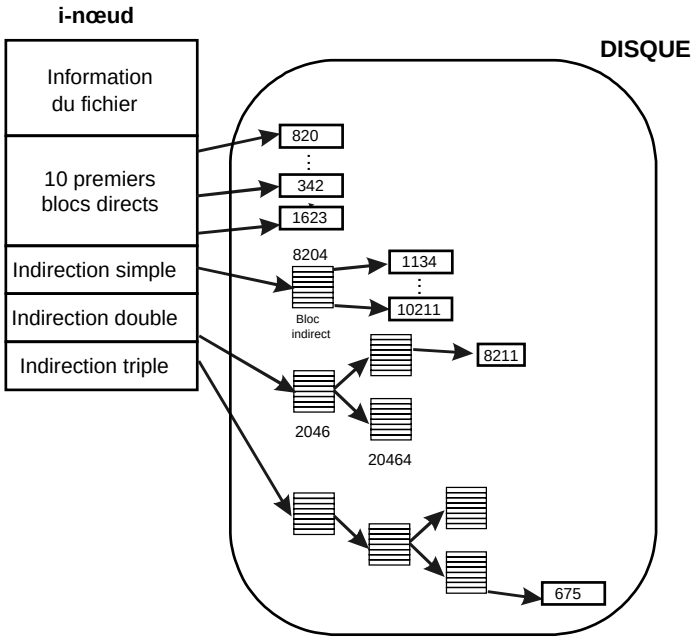


FIG. 12.2 – *Indirections d'un i-nœud.*

Un fichier peut avoir une taille maximale de 16 Go quand il utilise des pointeurs à triple indirection. La table d'index comporte jusqu'à 4 niveaux d'indexation, mais seul le premier niveau est présent en mémoire lorsque

le fichier correspondant est ouvert.

Dans Linux, la table d'i-nœuds diffère un peu de celle utilisée dans Unix **System V**. Sous Linux la table est composée de 12 entrées pour les blocs et de 3 entrées pour les indirections.

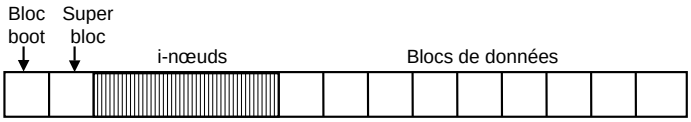


FIG. 12.3 – *i-nœud et super-bloc.*

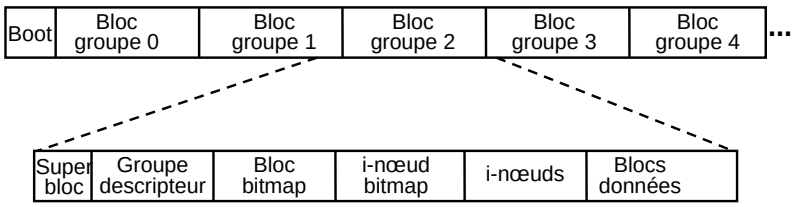


FIG. 12.4 – *Partition EXT2 (en haut) et un groupe de blocs EXT2 (en bas).*

12.2 Le système de fichiers /proc

Linux comporte le pseudo-système de fichiers /proc. Ceci est en réalité un répertoire qui contient des entrées numérotées par les processus en cours. Les entrées possèdent de l'information sous un format lisible par les personnes.

Statistiques du système

L'entrée de /proc/uptime contient de l'information en secondes du temps de démarrage du système et du temps mort :

```
leibnitz> cat /proc/uptime
547217.94 288458.71
```

Variable d'environnement

L'entrée de /proc/environ contient de l'information de l'environnement. Le programme suivant montre la variable d'environnement obtenue

à partir de l'entrée `/proc`

Listing 12.1: *env.c*

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
  int pid = atoi(argv[1]);
  int fd;
  char filename[24];
  char environment[8192];
  size_t length;
  char* next_var;

  // générer le nom d'environnement pour le processus
  snprintf(filename, sizeof (filename), "/proc/%d/environ",
          (int) pid);
  // lire le fichier
  fd = open(filename, O_RDONLY);
  length = read(fd, environment, sizeof(environment));
  close(fd);
  environment[length] = '\0';
  // cycle sur les variables
  next_var = environment;
  while (next_var < environment + length)
  {
    printf ("%s\n", next_var);
    next_var += strlen (next_var) + 1;
  }
  return 0;
}
```

Par exemple, dans une fenêtre on lance la commande `cat` qui reste bloqué en attente d'un argument de la part de l'utilisateur :

```
leibnitz> cat
```

```
 ...
```

et dans un autre terminal, on vérifie les processus et on lance le programme `env.c` avec le numéro de processus de `cat` :

```
leibnitz> ps -u jmtorres
 PID TTY          TIME CMD
1956 pts/4    00:00:00 tcsh
5764 pts/4    00:00:00 cat
5768 pts/1    00:00:00 tcsh
5819 pts/1    00:00:00 ps
leibnitz> gcc -o env env.c
leibnitz> env 5764
REMOTEHOST=torres.dgi.polymtl.ca
TERM=ansi
HZ=100
HOME=/home/ada/users/jmtorres
SHELL=/bin/tcsh
PATH=/home/ada/users/jmtorres/bin:/home/ada/users/jmtorres/
home/ada/users/jmtorres/office52:/home/ada/users/jmtorres/
bin/X11:/usr/local/bin:/usr/local/sbin:/usr/local/qt/bin:
/usr/openwin/bin:/usr/X11/bin:/usr/share/texmf/bin:/usr/bin:/
usr/local/java/j2sdk1.4.0_01/bin:/bin:/usr/bin:/sbin:/usr/games:/
usr/local/Simscript/bin:/usr/local/:/usr/local:/usr/local/Eclips
e/bin/i386_linux
...
HOSTNAME=leibnitz
...
EDITOR=emacs
...
CONSOLE=OFF
leibnitz>
```

Fichier de descripteurs de processus

L'entrée `fd` est un sous-répertoire qui contient des entrées pour les fichiers descripteurs ouverts par un processus. Chaque entrée est un lien symbolique au fichier ou au 'dispositif ouvert dans ce fichier descripteur. On peut écrire ou lire de ces liens symboliques ; écrire vers ou lire du fichier correspondant ou du dispositif ouvert par le processus. Les entrées dans le sous-répertoire `fd` sont nommées par les numéros du fichier descripteur.

Nous présentons une stratégie élégante qu'on peut essayer avec les entrées `fd` en `/proc`. Ouvrir une fenêtre shell, et trouver le processus ID du

processus `shell` en exécutant `ps` :

```
bash-2.05a$ ps
 PID TTY          TIME CMD
1956 pts/4    00:00:00 tcsh
2413 pts/4    00:00:00 bash
2414 pts/4    00:00:00 ps
bash-2.05a$
```

Dans ce cas, le shell (`bash`) s'exécute dans le processus `2413`. Maintenant, ouvrir une deuxième fenêtre et voir le contenu du sous-répertoire `fd` pour ce processus :

```
bash-2.05a$ ls -l /proc/2413/fd
total 0
lrwx------ 1 jmtorres prof  64 Nov 11 15:53 0 -> /dev/pts/4
lrwx------ 1 jmtorres prof  64 Nov 11 15:53 1 -> /dev/pts/4
lrwx------ 1 jmtorres prof  64 Nov 11 15:53 2 -> /dev/pts/4
lrwx------ 1 jmtorres prof  64 Nov 11 15:53 255 -> /dev/pts/4
bash-2.05a$
```

Les descripteurs de fichier `0`, `1`, et `2` sont initialisés à l'entrée, la sortie et l'erreur standard respectivement. Ainsi, en écrivant à `/proc/2413/fd/1`, on écrit au dispositif attaché à `stdout` par le processus shell — dans ce cas là, le pseudo `TTY` de la première fenêtre — et dans la deuxième fenêtre, on écrit un message sur ce fichier :

```
bash-2.05a$ echo "Hola..." >> /proc/2413/fd/1
bash-2.05a$
```

Le texte `"Hola..."` apparaît alors sur la première fenêtre.

12.3 Partage de fichiers

Lorsque plusieurs utilisateurs travaillent sur un même projet, ils doivent souvent **partager des fichiers**. Pour permettre le partage d'un fichier existant, le système Unix/Linux offre la possibilité d'associer plusieurs références à un fichier, en créant des **liens physiques** ou des **liens symboliques**. Par exemple, dans la figure 12.5, le fichier **X** est partagé entre les répertoires `/B/B/X` et `/C/C/C/X`

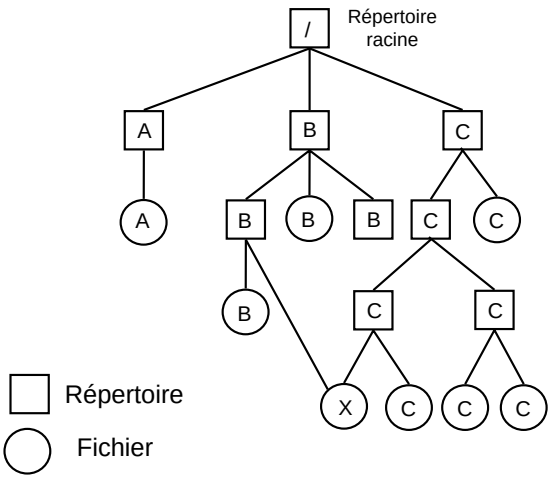


FIG. 12.5 – *Fichier partagé.*

12.3.1 Liens physiques

Pour réaliser le partage d'un fichier se trouvant sur un disque, il suffit d'associer à son numéro d'i-nœud plusieurs chemins d'accès ou références. Ce type de lien est **physique**. Les lien physiques sont possibles seulement si les chemins d'accès font références uniquement aux répertoires gérés par un *même système de fichiers*. Dans d'autres mots, on ne peut pas créer de lien à partir d'un répertoire d'une machine vers un fichier se trouvant sur une autre machine.

L'appel système qui permet de créer un lien physique à un fichier existant est `link()` :

```
int link(char* oldpath, char* newpath);
```

La commande shell équivalente à `link()` est `ln` :

```
ln oldpath newpath
```

L'appel système `link()` associe un autre chemin d'accès `newpath` à un fichier ayant comme chemin d'accès `oldpath`. Les deux chemins désignent le même fichier (`newpath` et `oldpath` référencent le même fichier). Une fois les liens établis, le fichier pourra être désigné par l'un des deux noms (chemins d'accès).

Par exemple, l'appel système `link("origref", "../autreref")` opère comme suit :

- Il recherche dans le répertoire courant l'entrée telle que le nom de fichier soit `"origref"`, puis récupère le numéro de l'i-nœud. Il incrémente le nombre de références de l'i-nœud.
- Il ajoute dans le répertoire père du répertoire courant une entrée ayant comme nom de fichier `"autreref"` et le numéro d'i-nœud du fichier `"origref"`.

Il existe un appel système qui permet de supprimer un lien physique à un fichier. Il s'agit de `unlink` :

```
int unlink(char * path);
```

`unlink()` supprime la référence `path` à un fichier. Si le fichier n'a plus de référence, il est supprimé du disque. Par exemple, `unlink("origref")` opère comme suit :

- Il localise, dans le répertoire courant, l'entrée telle que le nom de fichier soit `"origref"`. Il récupère le numéro de l'i-nœud puis supprime cette entrée du répertoire.
- Il décrémente le nombre de références à l'i-nœud. Si le nombre de références devient égal à `0`, le fichier est supprimé du disque (blocs des données et i-nœud).

La commande `shell` qui permet de supprimer un lien est `rm`. Regardez l'exemple suivant :

```
pascal> cat original
exemple d'un fichier avec plusieurs références.
pascal> ln original autreref
pascal> ls -l original autreref
-rw-r--r--    2 jmtorres prof      48 Nov 18 15:06 autreref
-rw-r--r--    2 jmtorres prof      48 Nov 18 15:06 original
pascal> cat autreref
exemple d'un fichier avec plusieurs références.
pascal> rm original
pascal> ls -l original autreref
/bin/ls: original: No such file or directory
-rw-r--r--    1 jmtorres prof      48 Nov 18 15:06 autreref
pascal> cat autreref
exemple d'un fichier avec plusieurs références.
pascal>
```

12.3.2 Liens symboliques

Les **liens symboliques** permettent de créer des liens vers des fichiers qui ne sont pas forcément gérés par le même système de fichiers. Ils présentent donc l'avantage de pouvoir constituer des liens vers des fichiers situés sur n'importe quel ordinateur à distance. Outre le chemin d'accès sur la machine même, il faut placer l'adresse réseau de la machine dans le chemin d'accès.

Un lien symbolique est un pointeur indirect vers un fichier existant. La destruction d'un lien symbolique vers un fichier n'affecte pas le fichier. La commande `shell` d'Unix qui crée des liens symboliques est `ln` avec l'option `s` pour symbolique. Par exemple, la commande :

```
ln -s /usr/include/stdio.h stdio.h
```

crée une nouvelle entrée dans le répertoire courant pour un nouveau fichier dont le nom est `stdio.h`. Le contenu de ce nouveau fichier est le chemin d'accès :

`/usr/include/stdio.h`.

Le type du fichier créé est un link `(l)`.

Par exemple :

```
pascal> ln -s /usr/include/stdio.h stdio.h
pascal> ls -l stdio.h
lrwxrwxrwx  1 jmtorres prof   20 Nov 18 15:02 stdio.h ->
              /usr/include/stdio.h
pascal> cat stdio.h
...
```

cette commande affiche le contenu du fichier `/usr/include/stdio.h`

Problèmes Les liens permettent aux fichiers d'avoir plusieurs chemins d'accès. Les programmes qui parcourent tous les catalogues pour en traiter les fichiers peuvent rencontrer les fichiers partagés plusieurs fois. Un même fichier peut donc subir le même traitement plusieurs fois (par exemple, ils peuvent être recopiés plusieurs fois). Une fois les liens établis, plusieurs utilisateurs peuvent accéder à un même fichier. Les accès peuvent être concurrents. Le système Unix fournit des appels système qui permettent de contrôler les accès aux données d'un même fichier.

Lecture de liens symboliques

L'appel système `readlink()` récupère la source d'un lien symbolique. Il prend comme arguments le chemin au lien symbolique, un `buffer` pour

recevoir le nom de la source et la longueur de ce `buffer`. `readlink` n'ajoute pas caractère `NUL` à la fin du nom du chemin. Cependant, il retourne le nombre de caractères du chemin. Si le premier argument à `readlink` pointe vers un fichier qui n'est pas un lien symbolique, `readlink` donne `errno = EINVAL` et retourne `-1`.

▶ **Exemple 1.** Le programme suivant imprime le chemin d'un lien symbolique spécifié sur la ligne de commande :

Listing 12.2: *sym.c*

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
    char target_path[256];
    char* link_path = argv[1];

    // Essai de lire le chemin du lien symbolique
    int len = readlink(link_path, target_path, sizeof(target_path));
    if(len == -1)
    {   // L'appel échoué
        if (errno == EINVAL)
            printf("%s n'est pas un lien symbolique\n",
                   link_path);
        else
            // Un autre problème a eu lieu
            perror("readlink");
        return 1;
    }
    else
    {
        // NUL-terminaison du chemin objectif
        target_path[len] = '\0';
        // Imprimer
        printf ("%\n", target_path);
        return 0;
    }
}
```

Regardez l'utilisation de `sym.c` :

```
pascal> ln -s /usr/bin/wc my_link
pascal> sym my_link
/usr/bin/wc
pascal>
```

▶ **Exemple 2.** Voici un autre exemple sur les différences entre liens symboliques et physiques. L'utilisation de l'option `"-li"` (listage des i-nœuds) de `ls` sera utilisée. Montrons d'abord les liens symboliques :

```
pascal> cat > liga.txt
exemple d'un fichier avec plusieurs références. ^D
pascal> ls -l liga*
-rw-r--r--   1 jmtorres prof    48 Nov 18 15:06 liga.txt
pascal> ln liga.txt liga.ln
pascal> ls -li liga*
3052239 -rw-r--r--   2 jmtorres prof    48 Nov 18 15:06 liga.ln
3052239 -rw-r--r--   2 jmtorres prof    48 Nov 18 15:06 liga.txt
pascal> rm liga.txt
pascal> cat liga.ln
exemple d'un fichier avec plusieurs références.
pascal> ls -li liga*
3052239 -rw-r--r--   1 jmtorres prof    48 Nov 18 15:06 liga.ln
pascal>
```

▶ **Exemple 3.** Observez maintenant ce qui se passe avec les liens physiques :

```
pascal> cp liga.ln ligado.txt
pascal> ln -s ligado.txt liga2.ln
pascal> ls -li liga*
3052242 lrwxrwxrwx   1 jmtorres prof    10 Nov 18 15:51 liga2.ln ->
                       ligado.txt
3052240 -rw-r--r--   1 jmtorres prof    48 Nov 18 15:51 ligado.txt
3052239 -rw-r--r--   1 jmtorres prof    48 Nov 18 15:06 liga.ln
pascal> rm ligado.txt
pascal> cat liga2.ln
cat: liga2.ln: No such file or directory
pascal> ls -li liga*
3052242 lrwxrwxrwx   1 jmtorres prof    10 Nov 18 15:51 liga2.ln ->
                       ligado.txt
3052239 -rw-r--r--   1 jmtorres prof    48 Nov 18 15:06 liga.ln
pascal>
```

Est-ce que les liens symboliques marchent avec les répertoires? Si vous en doutez, tapez les commandes suivantes :

```
pascal> ln -s /tmp temporal
pascal> cd temporal
pascal> ls -l
...
```

et maintenant :

```
pascal> cd /tmp
pascal> ls -l
...
```

Y-a-t-il des différences entre les deux sorties?

L'exemple suivant, pris de [CGdMP01], effectue la copie d'un fichier en utilisant des appels système **Posix** et la projection de fichiers en mémoire (Section 9.9 du Chapitre 9) :

▶ **Exemple 4.** Copie avec projection en mémoire :

Listing 12.3: *copie-mem.c*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

void main(int argc, char **argv)
{
10    int i, fdo, fdd;
    char *org, *dst, *p, *q;
    struct stat bstat;

    if (argc!=3)
        {
            fprintf(stderr, "%s orig dest\n", argv[0]);
            exit (1);
        }
    // Ouvrir source en lecture
```

```
    fdo=open(argv[1], O_RDONLY);
    // Créer destination
    fdd=open(argv[2],O_CREAT | O_TRUNC | O_RDWR, 0640);
    // Longueur du fichier source
    if ( fstat (fdo, &bstat)<0)
        {
            perror(" fstat " );
            close(fdo);
            close(fdd);
            unlink(argv[2]);
            exit (1);
        }
    // Longueur de la destination = longueur de la source
    if ( ftruncate(fdd, bstat.st_size)<0)
        {
            perror("ftruncate" );
            close(fdo);
            close(fdd);
            unlink (argv[2]);
            exit (1);
        }
    // Projeter le fichier source
    if ((org=mmap((caddr_t)0, bstat.st_size, PROT_READ,
                    MAP_SHARED,fdo,0)) == MAP_FAILED)
        {
            perror("mmap source");
            close(fdo);
            close(fdd);
            unlink (argv[2]);
            exit (1);
        }
    // Projeter le fichier destination
    if (( dst=mmap((caddr_t)0,bstat.st_size, PROT_WRITE,
            MAP_SHARED,fdd,0)) == MAP_FAILED)
        {
            perror("mmap destination");
            close(fdo);
            close(fdd);
            unlink (argv[2]);
            exit (1);
```

```
60          }
        close(fdo);
        close(fdd);
        // Copier
        p=org;
        q=dst;
        for(i=0; i<bstat.st_size ; i++)
            *q++= *p++;
        // Éliminer projections
        munmap(org, bstat.st_size);
70      munmap(dst, bstat.st_size);
    }
```

12.4 Verrouillage de fichiers

Les processus qui désirent effectuer des accès exclusifs à un fichier peuvent verrouiller, en une seule opération atomique, aussi bien un octet qu'un fichier entier. Deux sortes de **verrouillage de fichiers** sont disponibles : **partagé** et **exclusif**. Si une portion d'un fichier est verrouillée par un **verrou partagé**, une seconde tentative d'y superposer un verrou partagé est autorisée, mais toute tentative d'y poser un verrou exclusif sera rejetée tant que le verrou n'est pas relâché. Si une partie de fichier contient un **verrou exclusif**, toute tentative d'en verrouiller une quelconque portion sera rejetée tant que le verrou n'est pas relâché.

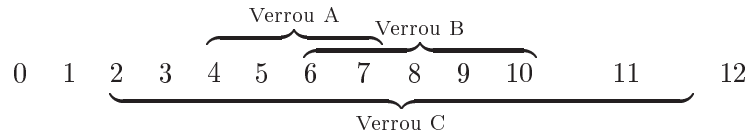
12.4.1 Verrouillage d'une partie de fichier

Lorsqu'un processus demande la pose d'un verrou, il doit spécifier s'il veut être bloqué si le verrou ne peut pas être posé. S'il choisit d'être bloqué, lorsque le verrou existant sera levé, le processus sera débloqué et son verrou pourra être posé. S'il ne veut pas être bloqué, l'appel système se termine immédiatement, avec un code qui indique si le verrouillage a pu être ou non effectué.

Par exemple, soient quatre processus concurrents **A**, **B**, **C** et **D** qui partagent un même fichier. Les processus font dans les l'ordre les demandes de verrouillage suivantes :

– Le processus **A** demande la pose d'un verrou partagé sur les octets 4 à 7 du fichier.

– Le processus **B** demande de mettre un verrou partagé sur les octets 6 à 9.
– Le processus **C** demande la pose d'un verrou partagé sur les octets 2 à 11.
– Enfin, le processus **D** demande un verrouillage exclusif de l'octet 9 avec une requête bloquante en cas d'échec.



Les processus **A**, **B** et **C** arrivent à poser leurs verrous partagés. Par contre, le processus **D**, est bloqué jusqu'à ce que l'octet 9 devienne libre, c'est-à-dire jusqu'à ce que **B** et **C** libèrent leurs verrous.

12.4.2 Services Posix de verrouillage de fichiers

Posix a défini les appel système `lockf()` et `fcntl()` comme standard pour le verrouillage des parties d'un fichier.

`lockf()`

```
int lockf(int fd, int fonction, int size);
```

Le service `lockf()` retourne `0` en cas de succès et `-1` en cas d'échec. `fd` est un descripteur du fichier, le mode d'ouverture doit être `O_WRONLY` ou `O_RDWR` et `fonction` indique l'action à faire. Les valeurs possibles de `fonction` sont définies dans l'entête `<unistd.h>` comme suit :

`F_ULOCK` libère le verrou posé précédemment sur la partie composée de `size` octets contigus et pointée par le pointeur de fichier.

`F_LOCK` pose le verrou sur la partie libre composée de `size` octets contigus et pointée par le pointeur de fichier. Si la partie à verrouiller n'est pas libre, le processus demandeur est mis en attente jusqu'à ce que la partie devienne libre. Le processus n'est pas mis en attente si le système détecte un interblocage.

`F_TLOCK` pose le verrou sur la partie libre composée de `size` octets contigus et pointée par le pointeur de fichier. Si la partie à verrouiller n'est pas libre, l'appel système `lockf()` retourne -1 (échec de tentative de pose de verrou).

`F_TEST` teste si la partie composée de `size` octets contigus et pointée par le pointeur de fichier est libre ou non. Si la partie à verrouiller n'est pas libre, `lockf()` retourne `-1`.

▶ **Exemple 5.** Le programme `partage.c` montre comment effectuer le partage d'un fichier sans verrouillage :

Listing 12.4: *partage.c*

```
#include <unistd.h>

int main(void)
{
   int i,
   fd=1;

   if(fork())
   {
10      for(i=0; i<5; i++)
        {
            write(fd,"pere ecris\n",11);
            sleep(1);
        }
   } else
   {
        for(i=0; i<4; i++)
        {
            write(fd," fils  ecris\n",11);
20          sleep(1);
        }
   }
   return 0;
}
```

Exécution du programme `partage.c` :

```
pascal> gcc -o partage partage.c
pascal> partage
père écris
fils écris
père écris
fils écris
```

```
fils écris
père écris
père écris
fils écris
père écris
pascal>
```

▶ **Exemple 6.** Le programme `verrou.c` montre le partage d'un fichier avec verrouillage :

Listing 12.5: *verrou.c*

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(void)
{
 int i, fd=1;

 if(fork()) // il s'agit du père
        {
            lseek(fd,0,0);
            if( lockf(fd, F_LOCK,1) <0)
            {
                write(fd,"pere lockf failed",18); return (-1);
            }
            for(i=0; i<5; i++)
            {
                write(fd,"pere ecris \n",13);
                sleep(1);
            }
            write(fd,"père va libérer le verrou\n",26);
            lseek(fd,0,0);
            lockf(fd,F_ULOCK,0);
            wait(NULL);
        }
 else // il s'agit du fils
       {
            lseek(fd,0,0); // verrouillé l'octet 0
            if( lockf(fd, F_LOCK,1) <0)
```

```
30              {
                    write(fd," fils  lockf  failed \n",18); return (-1);
                }
                for(i=0; i<4; i++)
                    {
                        write(fd," fils  ecris \n",12);
                        sleep(1);
                    }
                write(fd," fils  va libérer  le  verrou \n",26);
                lseek(fd,0,0);
40              lockf(fd,F_ULOCK,0);
        }
    close(fd);
    return 0;
}
```

Exécution de `verrou.c` :

```
pascal> gcc -o verrou verrou.c
pascal> verrou
père écris
père écris
père écris
père écris
père écris
père va libérer le verrou
fils écris
fils écris
fils écris
fils écris
fils va libérer le verrou
pascal>
```

`fcntl()`

L'appel système `fcntl()` est, en réalité beaucoup plus compliqué car le prototype est : `int fcntl(int fd, int commande, ...);` et sert à plusieurs utilisations. Parmi elles on trouve la duplication de descripteurs, l'accès aux attributs d'un descripteur et le verrouillage des fichiers.

```
#include <fcntl.h>
int fcntl(int fd, int op, struct flock *verrou);
```

`fd` est le descripteur du fichier à verrouiller. `verrou` est une structure avec les champs :

- `l_type` : `int` Type de verrouillage.
- `l_whence` : `int` équivalent à l'appel système `lseek()` avec `SEEK_SET`, `SEEK_CUR` et `SEEK_END`.
- `l_start` : Début de la portion verrouillée du fichier.
- `l_len` : Longueur verrouillée en octets.
- `l_pid` : `PID` du processus. Donné par le système.

Le type de verrouillage peut être :

- `F_RDLCK` : Verrou partagé en lecture.
- `F_WRLCK` : Verrou exclusif en écriture.
- `F_UNLCK` : Absence de verrou.

`op` est l'une des opérations suivantes :

- `F_GETLK` : Accès aux caractéristiques d'un verrou existant.
- `F_SETLK` : Définition ou modification d'un verrou en mode non bloquant.
- `F_SETLKW` : Définition ou modification d'un verrou en mode bloquant.

▶ **Exemple 7.** Le programme 12.6 `lock.c` bloque un fichier jusqu'à ce qu'on tape `<Return>` pour le débloquer.

Listing 12.6: *lock.c*

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char* file = argv[1];
  int fd;
  struct flock lock;

  printf("Ouverture de %s\n",file);
  fd = open(file, O_RDWR);
  printf("Verrouillage\n");
```

```
    memset(&lock, 0, sizeof(lock));
    lock.l_type = F_WRLCK;
    fcntl(fd,F_SETLKW,&lock);
    printf("Verrouillé . Presser ENTER pour debloquer...");
    getchar();
    printf("Deverrouillage\n");
    lock.l_type = F_UNLCK;
    fcntl(fd,F_SETLKW,&lock);
    close(fd);
    return 0;
}
```

Dans un terminal, créer un fichier de TEST, et le verrouiller avec `lock.c` :

```
leibnitz> cat > TEST
HOLA!
^D
leibnitz> lock TEST
Ouverture de TEST
Verrouillage
Verrouillé. Presser ENTER pour débloquer...
```

Dans un autre terminal, essayer de verrouiller le même fichier TEST :

```
leibnitz> lock TEST
Ouverture de TEST
Verrouillage
```

L'exécution reste gelée jusqu'à ce qu'on revienne à la première terminal et on presse <Return> :

```
...
Verrouillé. Presser ENTER pour débloquer...
Déverrouillage
leibnitz>
```

12.4.3 Verrouillage avec `flock()`

Le système Linux et Unix BSD offrent `flock()`, un autre appel système pour le verrouillage . Cet appel n'est pas standardisé Posix, mais il est encore utilisé.

```
#include <sys/file.h>
int flock(int fd, int operation);
```

Où `fd` est le descripteur de fichier, avec des `operation` valables :

- `LOCK_SH` **Verrou partagé**. Plus d'un processus peut verrouiller en même temps.
- `LOCK_EX` **Verrou exclusif**.
- `LOCK_UN` Déverrouiller le `fd`.
- `LOCK_NB` Non bloquer le verrouillage.

Regardez l'exemple suivant :

▶ **Exemple 8.** Utilisation de `flock` dans le programme `flock.c` :

Listing 12.7: *flock.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/file.h>

void display(int fd)
{
  int donnee;

  flow(fd,LOCK_SH);

  while(read(fd, &donnee, sizeof(int)) > 0)
        printf("donnee: %d\n",donnee);
  close(fd);
}

void add(int fd)
{
  int donnee;

  flow(fd,LOCK_EX);
  do
  {
    printf("Donnee? ");
    scanf("%d",&donnee);
    write(fd, &donnee, sizeof(int));
  } while (donnee!=-1);
  close(fd);
}
```

```
int flow(int fd, int op)
{
  printf("lock %s sur fd=%d\n",
         (op & LOCK_SH)?"partage":"exclusif", fd);
  if(flock(fd, op | LOCK_NB) != -1)
    return 0;
  printf("Autre processus a le lock. Atteindre\n");
  return flock(fd,op);
}

int main(int argc, char *argv[])
{
  int fd;
  int opt;

  if(argc==1)
    {
      printf("flock 1=Montrer 2=Ajouter\n");
      exit (1);
    }
  fd = open("file", O_CREAT|O_RDWR, 0640);
  opt = atoi(argv[1]);
  if(opt==1) display(fd);
  if(opt==2) add(fd);
  return 0;
}
```

Dans un écran shell exécuter `flock.c` :

```
pascal> flock
flock 1=Montrer 2=Ajouter
pascal> flock 2
lock exclusif sur fd=3
Donnee? 1
Donnee? 2
Donnee? 3
Donnee? 4
```

Sans terminer, dans un deuxième shell exécuter `flock 1` :

```
pascal> flock 1
lock partagé sur fd=3
Autre processus a le lock. Atteindre
```

Et dans un troisième shell exécuter `flock 2` :

```
pascal> flock 2
lock exclusif sur fd=3
Autre processus a le lock. Atteindre
```

Si l'on revient dans la première session et on termine l'ajout des données :

```
...
Donnee? 4
Donnee? -1
pascal>
```

probablement la troisième (ou deuxième) session va se débloquer, et on aura la demande des données. Si l'on poursuit et on finit la saisie :

```
...
Autre processus a le lock. Atteindre
Donnee? 5
Donnee? 6
Donnee? -1
pascal>
```

C'est alors que la deuxième (ou troisième) session se réveille et montre son exécution. On espère évidement qu'elle montrera la séquence `1, 2, 3, 4, 5, 6, -1`, mais ce qu'elle montre est totalement différent :

```
...
lock partage sur fd=3 Autre processus a le lock. Atteindre
donnee: 5
donnee: 6
donnee: -1
donnee: 4
donnee: -1
pascal>
```

Pourquoi ne montre-t-elle pas la liste attendue? À vous de répondre.

12.5 Antémémoire

Les processus ne peuvent pas manipuler directement les données du disque. Ils doivent être déplacés en mémoire centrale. De nombreux systèmes de fichiers cherchent à réduire le nombre d'accès au disque car le temps d'accès moyen au disque est égal à quelques dizaines de millisecondes.

La technique la plus courante pour réduire les accès au disque, consiste à utiliser une **antémémoire** (**buffer cache** ou **block cache**). L'antémémoire

est un espace situé en mémoire centrale dans laquelle on charge un ensemble de blocs du disque. A chaque demande d'accès à un élément d'un fichier, on examine d'abord si le bloc désiré — celui qui contient l'élément du fichier — se trouve dans l'antémémoire. Si c'est le cas, la demande est satisfaite sans avoir à accéder au disque. Sinon, pour satisfaire la demande, le bloc désiré doit être d'abord chargé à partir du disque dans l'antémémoire. S'il faut charger un bloc et que l'antémémoire est pleine, il faut retirer un des blocs de l'antémémoire pour le remplacer par celui demandé. Le bloc retiré doit être recopié sur le disque s'il a été modifié depuis son chargement. Pour choisir le bloc à retirer, il est possible d'utiliser un des algorithmes de remplacement de pages (FIFO, LRU, etc.).

Problème : Si des blocs placés dans l'antémémoire ont été modifiés et une panne survient avant de les copier sur disque, le système de fichiers passe dans un état incohérent.

Pour minimiser les risques d'incohérence, le système Unix recopie les blocs contenant les i-nœuds et les répertoires immédiatement après leur modification. Les blocs de données ordinaires sont recopiés manuellement s'ils doivent être retirés de l'antémémoire ou automatiquement par un démon toutes les 30 secondes. On peut également déclencher ce démon par l'appel système `sync()`.

Dans MS-DOS, les blocs de l'antémémoire sont recopiés sur disque à chaque modification. L'anténoire n'est pas la seule façon d'améliorer les performances d'un système de fichiers. Une autre technique consiste à réduire le temps de recherche du bloc dans le disque (voir Chapitre 13 *Périphériques d'entrées/sorties*, Section 13.5 *Gestion du bras du disque*).

12.6 Cohérence du système de fichier

La plupart des ordinateurs ont un programme utilitaire qui vérifie la cohérence du système de fichiers. Ce programme est peut être exécuté à chaque démarrage du système surtout à la suite d'un arrêt forcé. La vérification de la cohérence peut se faire à deux niveaux : blocs ou fichiers.

12.6.1 Au niveau des blocs

Au niveau des blocs, le vérificateur construit deux tables. La première indique pour chaque bloc occupé, le nombre de fois où le bloc est référencé dans les i-nœuds des fichiers. La seconde indique pour chaque bloc libre, le

nombre de fois où il figure dans la liste des blocs libres (ou la table de bits des blocs libres).

Si le système de fichiers est cohérent, chaque bloc a un 1 soit dans la première table, soit dans la seconde. Si un bloc n'apparaît ni dans la première table, ni dans la deuxième, le bloc est dit manquant. Le vérificateur ajoute les blocs manquants à la liste des blocs libres. Si le bloc apparaît deux fois dans la liste des blocs libres, le vérificateur reconstruit la liste des blocs libres.

Le pire qui puisse arriver est qu'un bloc appartienne à deux fichiers (ou plus). Si on détruit l'un des deux fichiers, le bloc sera placé dans la liste des blocs libres (il sera alors à la fois libre et utilisé). Si on détruit les deux fichiers, le bloc figurera deux fois dans la liste des blocs libres. Dans ce cas, la meilleure solution consiste à allouer un bloc libre, à y copier le bloc commun et à remplacer le bloc commun dans l'un des deux fichiers par le nouveau. La dernière possibilité d'incohérence dans les deux tables est qu'un bloc soit, à la fois, utilisé et libre. La solution dans ce cas consiste à retirer le bloc de la liste des blocs libres.

12.6.2 Au niveau des fichiers

Le vérificateur vérifie 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. Il commence au catalogue racine et parcourt récursivement toute l'arborescence pour déterminer pour chaque numéro d'i-nœud le nombre de références. Il compare ensuite le nombre de références obtenu pour chaque numéro d'i-nœud avec celui contenu dans l'i-nœud correspondant. Si les deux nombres sont différents, le système est dans un état incohérent. La solution consiste à corriger dans l'i-nœud, la valeur du nombre de références. Dans la plupart des systèmes Unix, le programme `fsck` effectue cette tâche à chaque démarrage, si nécessaire. Voir `man fsck` pour plus de détails.

12.7 Protection

Les systèmes de fichiers contiennent parfois des informations très importantes. Ils doivent donc protéger ces informations contre les accès non autorisés et les pertes. Les causes des pertes sont les catastrophes naturelles, les erreurs matérielles ou logicielles, les erreurs humaines. La plupart de ces problèmes peuvent être résolus si l'on effectue des sauvegardes assez régulièrement.

Le problème des intrus est beaucoup plus complexe. Pour assurer la protection contre les intrus plusieurs mécanismes ont été mis en oeuvre, par exemple l'identification de l'utilisateur (mots de passe ou identification physique) ou les codes de protection des objets, parmi d'autres.

12.8 Exercices

1. On considère un système de fichiers tel que l'information concernant les blocs de données de chaque fichier est donc accessible à partir de l'i-nœud de celui-ci (comme dans Unix). On supposera que :
   - Le système de fichiers utilise des blocs de données de taille fixe 1K (1024 octets).
   - L'i-nœud de chaque fichier (ou répertoire) contient 12 pointeurs directs sur des blocs de données, 1 pointeur indirect simple, 1 pointeur indirect double et 1 pointeur indirect triple.
   - Chaque pointeur (numéro de bloc) est représenté sur 4 octets.

   (a) Quelle est la plus grande taille de fichier que ce système de fichiers peut supporter ?
   (b) On considère un fichier contenant 100,000 octets. Combien de blocs de données sont-ils nécessaires au total pour représenter ce fichier sur disque ?

2. Dans le système Unix, tous les renseignements concernant un fichier sont conservés dans des i-nœuds. Un fichier est identifié par le numéro de son i-nœud à l'intérieur de la table des i-nœuds. Un répertoire est un ensemble de couples composés chacun d'un nom de fichier relatif à ce répertoire et du numéro de l'i-nœud identifiant le fichier. La commande `ln` permet d'associer un autre nom (lien) à un fichier existant. Cela est très utile lorsque l'on veut pouvoir accéder à un fichier depuis plusieurs répertoires. Il est indispensable de connaître à chaque instant le nombre de liens sur un fichier car on ne peut supprimer un fichier que si tous les liens ont été supprimés. Cette information se trouve dans l'i-nœud du fichier.
   (a) Indiquer les opérations à effectuer par le processus (s'exécutant en mode système) lors de l'exécution d'une commande `ln`.
   (b) Expliquez pourquoi est-il nécessaire d'empêcher la manipulation du fichier existant par d'autres processus pendant l'exécution de cette commande ?
   (c) Supposons que le verrouillage du fichier est réalisé à l'aide d'un bit de verrouillage placé dans l'i-nœud du fichier. Peut-on avoir des situations d'interblocage, si le fichier existant est verrouillé pendant l'exécution de `ln` ? Proposer une solution pour traiter ce problème.

3. Soient n processus identiques accédant en parallèle à un fichier contenant un numéro à incrémenter. Écrire le programme de ces processus et utiliser un verrou externe pour synchroniser l'accès. Observez le résultat sans le verrou.

13
Périphériques d'Entrées/Sorties

CHaque ordinateur dispose de **périphériques** d'Entrées/Sorties tels que les terminaux, les disques, les imprimantes, les équipements de communication, auxquels les programmes doivent pouvoir accéder. Le système Unix/Linux considère les périphériques d'Entrées/Sorties comme des **fichiers spéciaux**, et donc il intègre la gestion des périphériques dans le système de fichiers.

13.1 Périphériques

A chaque périphérique d'Entrée/Sortie est attribué un chemin d'accès, le plus souvent à partir du répertoire `/dev`.

```
/dev/lp
/dev/tty
/dev/net
/dev/null
```

Ces fichiers spéciaux sont accessibles de la même façon que les autres fichiers, avec des appels système `read()`, `write()`, etc. Les fichiers spéciaux sont en réalité vides, et ont pour mission d'associer les dispositifs et leurs contrôleurs. Le SE utilise deux numéros entiers appelés **numéro majeur** (14 bits) et **numéro mineur** (18 bits). Ils sont stockés dans l'i-nœud du fichier, pour accéder au dispositif associé.

Le **numéro majeur** identifie une classe de dispositifs (en réalité identifie le pilote de la classe, c'est à dire les terminaux, les imprimantes, les disques, etc.) et le **numéro mineur** identifie un élément de cette classe : un terminal spécifique, un disque ou une partition, etc.

Les avantages de gérer les Entrées / Sorties de cette manière sont :

- Les fichiers et les dispositifs sont traités de façon similaire.
- Les noms sont passés aux programmes de la même manière.
- Les fichiers spéciaux sont sous le même type de protection d'accès que les fichiers et les répertoires.

Le logiciel d'Entrées / Sorties peut être structuré en quatre couches, comme montré sur la figure 13.1 :

1. Le traitement des interruptions des E / S.
2. Les pilotes de périphériques qui traitent les requêtes de plus haut niveau qui émanent du logiciel (indépendant du matériel). On a un pilote par type de périphérique.
3. Le logiciel du système d'exploitation indépendant des périphériques effectue les fonctions communes à tous les périphériques tout en fournissant une interface uniforme au logiciel des utilisateurs.
4. Le logiciel accessible au niveau de l'utilisateur (les appels système).

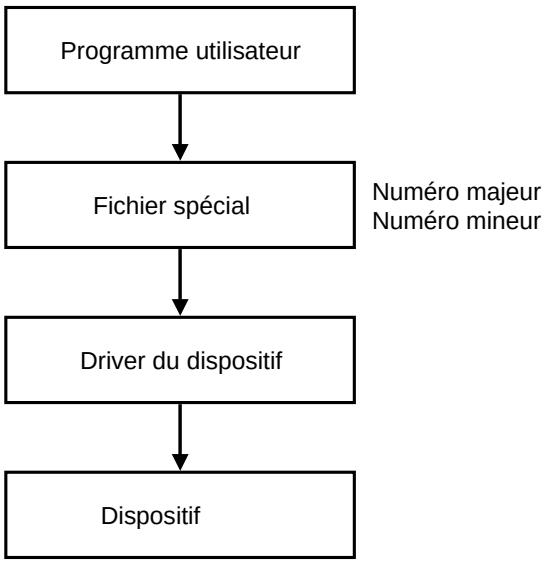


FIG. 13.1 – *Fichiers spéciaux et dispositifs.*

13.2 Le répertoire /dev

Le répertoire /dev du système GNU/Linux contient la liste des périphériques à caractère et à bloc que Linux reconnaît. Les entrées en /dev ont des noms standardisés qui correspondent au numéro de dispositif majeur et mineur.

Par exemple, le périphérique maître /dev/hda attaché au contrôleur IDE primaire, a un numéro de dispositif majeur et mineur de 3 et 0 respectivement. Si le dispositif supporte les partitions, la première partition /dev/hda1, aura son numéro mineur 1, la deuxième /dev/hda2 aura son mineur 2, et ainsi de suite. On l'a vérifié sur leibnitz :

```
leibnitz> ls -l /dev/hda /dev/hda1 /dev/hda2
brw-rw---- 1 root disk     3,   0 Jun  9 15:27 /dev/hda
brw-rw---- 1 root disk     3,   1 Jun  9 15:27 /dev/hda1
brw-rw---- 1 root disk     3,   2 Jun  9 15:27 /dev/hda2
```

De façon similaire, /dev a des entrées pour le périphérique à caractères port parallèle :

```
leibnitz> ls -l /dev/lp0 /dev/lp1
crw-rw----    1 root     lp 6, 0 Apr 27  1995 /dev/lp0
crw-rw----    1 root     lp 6, 1 Apr 27  1995 /dev/lp1
```

Dans la plupart des cas, il n'est pas nécessaire d'utiliser mknod pour créer notre propre périphérique d'entrées. On utilise simplement les entrées en /dev. Les programmes non super-utilisateur n'ont pas de choix sauf pour l'utilisation de dispositifs d'entrées qui existent déjà parce qu'ils ne peuvent pas les créer. Typiquement, seulement les systèmes gérants et les développeurs travaillant avec du matériel spécialisé ont besoin de créer des pilotes d'E/S. La plupart des versions GNU/Linux incluent des ressources pour aider le gestionnaire du système à créer les dispositifs d'entrées standards avec des noms corrects.

Accès aux dispositifs par ouverture de fichiers

Dans le cas de périphériques à caractères, leur utilisation demeure très simple : cela correspond à ouvrir le dispositif comme un fichier normal, puis à lire ou écrire sur lui. On peut même utiliser les commandes d'un fichier normal, telles que cat, ou la **redirection** du shell pour envoyer les données vers ou à partir d'un dispositif. Par exemple, si on a une imprimante branchée au premier port parallèle, on peut imprimer des fichiers en les envoyant directement à /dev/lp0 :

```
cat document.txt >> /dev/lp0
```

Il faut avoir la permission pour écrire sur le périphérique. Dans beaucoup de systèmes GNU/Linux seulement `root` et le démon d'impression (`lpd` ou `lp`) peuvent écrire sur le fichier. Il dépend aussi de comment l'imprimante interprète le contenu des données envoyées. Quelques imprimantes imprimeront les fichiers texte tels quels, mais d'autres vont les interpréter. Les imprimantes PostScript interprètent et impriment les fichiers PostScript.

Pour un programme, envoyer des données aux périphériques est très simple. Par exemple, ce fragment de code utilise des fonctions E/S de bas niveau pour envoyer le contenu de `buf` à `/dev/lp0`:

```
...
int fd = open("/dev/lp0",O_WRONLY);
write(fd, buf, sizeof(buf));
close(fd);
...
```

/dev/null

L'entrée `/dev/null` a deux utilisations :

- Écarter n'importe quelle écriture : `command > /dev/null`, un peu comme un trou noir.
- Quand on copie à partir de `/dev/null` on obtient un fichier de taille zéro : `cp /dev/null fichier_vide`

/dev/zero

Ce dispositif retourne une séquence infinie de `0`s.

/dev/random

Génération de numéros aléatoires. Visualisation avec `od`[1] de `/dev/random`:

```
leibnitz> od /dev/random
 0000000 064252 154513 124132 046024 055630 044173 126252 075347
 0000020 144673 033032 150536 120400 161154 011740 015621 007271
 0000040 104171 051240 124450 024716 146702 071344 076755 064454
 0000060 177260 151770 024642 136707 143407 127412 020273 051672
 ...
```

1. `od` : *dump* octal de fichiers.

13.3 Les contrôleurs

Chaque périphérique a son propre contrôleur qui est commandé par le processeur de l'ordinateur. La communication entre le processus et le contrôleur se fait par l'intermédiaire de quelques registres. Le système d'exploitation (un pilote d'E/S) demande des E/S en écrivant des commandes dans les registres du contrôleur. Ces commandes correspondent directement aux mécanismes physiques du périphérique tels que le déplacement du bras d'un disque[2].

A la suite d'une transmission d'une commande au contrôleur, le processeur peut effectuer un autre travail. Le contrôleur de son côté traite la commande et lorsqu'il termine, il envoie une interruption pour permettre au système d'exploitation de réquisitionner le processeur afin de récupérer les résultats de l'opération. Ces résultats sont dans les registres ou le tampon du contrôleur.

Exemple du son : `ioctl()`

La fonction `ioctl()` est utilisée pour le contrôle des périphériques à très bas niveau. Cette fonction a la syntaxe :

```
int ioctl(int manejador, int operation, int *valor)
```

`ioctl()` est une fonction *C* pour traiter des dispositifs. L'argument `manejador` (handle) est un descripteur du fichier `/dev/dsp` obtenu par `open()`. Pour `operation`, l'entête `linux/soundcard.h` contient les constantes suivantes :

- `SOUND_PCM_WRITE_BITS` : Résolution en bits (8 ou 16).
- `SOUND_PCM_WRITE_CHANNELS` : Nombre de canaux (1 ou 2).
- `SOUND_PCM_WRITE_RATE` : Fréquence d'échantillonnage / reproduction (en hertz).
- `SOUND_PCM_READ_BITS` : Obtenir résolution actuelle.
- `SOUND_PCM_READ_CHANNELS` : Obtenir le nombre actuel de canaux.
- `SOUND_PCM_READ_RATE` : Obtenir la fréquence d'échantillonnage / reproduction actuelle.

2. Par exemple, le contrôleur du lecteur de disquettes de l'IBM PC accepte 15 commandes différentes comme `READ`, `WRITE`, `SEEK` et `FORMAT`.

On passe la valeur par référence en `valor`. `ioctl()` retourne `-1` en cas d'erreur.

▶ **Exemple 1.** Supposons que l'on dispose d'un fichier d'échantillons monophoniques de 8 bits et qu'on veut reproduire à une fréquence d'échantillonnage de 22,000 Hz. Le programme `sound.c` suivant permet de reproduire le fichier `"muestras.raw"`.

Listing 13.1: *sound.c*

```
#include <linux/soundcard.h>
#include <sys/ioctl.h>
#include <fcntl.h>

int fich_dsp;
int fich_muestras;
int valor;
char bufer[8192];
int leidas;

int main()
{
   fich_dsp = open("/dev/dsp",O_WRONLY);
   valor=8; // Résolution
   ioctl(fich_dsp,SOUND_PCM_WRITE_BITS, &valor);
   valor=1; // Mono
   ioctl(fich_dsp,SOUND_PCM_WRITE_CHANNELS, &valor);
   valor=22000; // Fréquence
   ioctl(fich_dsp,SOUND_PCM_WRITE_RATE, &valor);
   fich_muestras=open("muestras.raw",O_RDONLY);
   do
   {
     leidas=read(fich_muestras,bufer,sizeof(bufer));
     write(fich_dsp,bufer,leidas);
   } while(leidas>0);

   close(fich_muestras);
   close(fich_dsp);
  return 0;
}
```

13.4 L'accès direct à la mémoire

De nombreux contrôleurs, en particulier ceux des périphériques bloc, permettent l'accès direct à la mémoire (**DMA**). Le **DMA** a été créé pour libérer le processeur de transférer les données du tampon du contrôleur vers la mémoire centrale. Pour ce faire, le processeur envoie, dans la commande —par exemple de lecture— deux informations supplémentaires : l'**adresse mémoire** où le bloc doit être écrit et le **nombre d'octets** à transférer.

Le contrôleur lit d'abord le bloc du périphérique, place les données dans son tampon et vérifie s'il y a eu des erreurs de lecture. S'il y a eu erreur, il peut selon le cas soit recommencer la lecture ou provoquer une interruption. Autrement il copie le premier octet dans la mémoire principale à l'adresse spécifiée au **DMA**, incrémente cette adresse et décrémente le compteur du **DMA** (qui était initialisé au nombre d'octets à transférer). Cette opération est répétée jusqu'à ce que le compteur du **DMA** atteigne la valeur zéro. Le contrôleur génère alors une interruption.

13.5 Cas du disque

Tous les disques sont composés de **cylindres** qui contiennent autant de **pistes** qu'il y a de têtes placées verticalement. Les pistes se divisent en secteurs, le nombre de secteurs est compris entre 8 et 32. Tous les secteurs contiennent le même nombre d'octets. Chaque disque a un pilote qui reconnaît et exécute les demandes de lecture / écriture. Chaque demande de lecture / écriture nécessite d'abord de positionner le bras du disque pour ramener la tête de lecture sur le secteur concerné.

Structure physique des disques

Les disques sont composés de plusieurs surfaces magnétisées et de têtes de lecture / écriture. Les surfaces se divisent en **cylindres**, avec une piste pour chaque tête et un nombre de secteurs par piste. Les secteurs peuvent être de 512 bytes. Ainsi la capacité du disque est :

```
Capacité=cylindres * pistes * secteurs * taille secteur
```

Les opérations d'E / S se calculent en considérant deux paramètres :

- Le temps de recherche : le temps nécessaire pour aller d'une piste à autre.
- Le temps de latence : le demi-tour que les données prennent pour arriver sur les têtes, une fois sur la piste.

Ainsi le temps d'accès T_{acces} est calculé comme :

$$T_{acces} = n * T_{recherche} + T_{latence} + T_{transfert}$$

13.5.1 Planification de disque

Le système d'exploitation a la responsabilité d'utiliser le matériel de forme efficace. Avec les disques ceci implique des accès rapides. Le temps d'accès a deux composantes principales :

- Recherche : temps que le bras du disque prend pour déplacer les têtes jusqu'au cylindre qui contient le secteur désiré.
- Latence : temps d'attente supplémentaire pour que le disque tourne jusqu'à ce qu'il se place sur le secteur désiré.

L'objectif consiste à minimiser le temps de recherche, qui est proportionnel à la distance de recherche. On peut ainsi augmenter la **bande passante**, qui est le nombre d'octets transférés divisés pas le temps de transfert.

Il y a plusieurs algorithmes de planification du disque, qui sont des variations des algorithmes de gestion de mémoire. Ainsi on trouve dans la littérature les algorithmes **FCFS**, **SCAN**, **C-SCAN**, **SSTF**, parmi d'autres. L'algorithme de planification est fondamental pour optimiser l'accès au disque. Les critères de planification consistent en optimiser le temps de recherche et en donner un service déterministe et fiable.

FCFS

S'il y a plusieurs demandes de lecture / écriture, une première solution consiste à traiter les requêtes dans l'ordre d'arrivée (PAPS ou FCFS). L'exemple dans [Tan01] suppose qu'on a la liste de requêtes *L=11,1,36,16,34,9*. Si l'on utilise **FCFS**, le parcours total de cylindres sera de 111. Cependant, pour traiter un maximum de demandes en un temps plus court, il serait plus intéressant de minimiser les déplacements du bras.

SSTF

De nombreux pilotes utilisent une table indexée par les numéros des cylindres, où chaque entrée pointe sur le premier élément d'une liste chaînée des différentes requêtes qui concernent un cylindre. A la fin d'une requête, le pilote doit choisir la prochaine requête à traiter par exemple celle qui concerne le cylindre le plus proche de la position actuelle du bras, cet algorithme s'appelle le **plus court déplacement d'abord** ou *Shortest Seek*

Time First **SSTF**. Avec la même liste *L=11,1,36,16,34,9* de requêtes, **SSTF** nécessite seulement un parcours de 61 cylindres (voir figure 13.2).

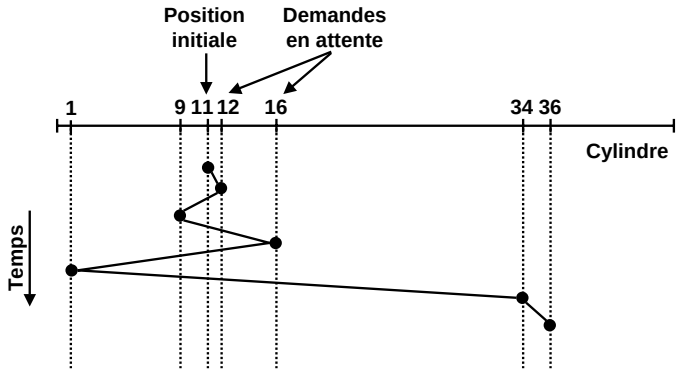


FIG. 13.2 – *SSTF.*

Cet algorithme présente un inconvénient : si de nouvelles requêtes arrivent continuellement concernant des cylindres proches de la position actuelle, le traitement des requêtes qui concernent les cylindres éloignés risquerait d'être retardé indéfiniment. Il n'y a pas donc d'équité.

SCAN

Une autre solution consiste à choisir un sens de parcours (montant ou descendant), puis à servir toutes les requêtes dans cette direction en commençant par les plus proches, jusqu'à épuisement. On inverse alors le sens de parcours et on recommence. On appelle ceci l'**algorithme de l'ascenseur** ou **SCAN** (voir figure 13.3). La liste *L=11,1,36,16,34,9* de requêtes nécessite un parcours de 60 cylindres avec **SCAN**. Le déplacement maximum du bras est égal à deux fois le nombre de cylindres. Une variation de cet algorithme c'est l'algorithme **C-SCAN**, où l'on fait des parcours circulaires.

Critique des algorithmes

Étant donnée la diversité des algorithmes de planification de disque, il est naturel de se poser les questions : lequel choisir ? selon quels critères ? **SSTF** est l'un des plus utilisés. **SCAN** et **C-SCAN** semblent adéquats pour des systèmes très chargés. Par contre en systèmes peu chargés, **FCFS** pourrait suffire. Le type de requêtes a une influence sur la planification du disque : normalement les fichiers sont stockés dans des positions conti-

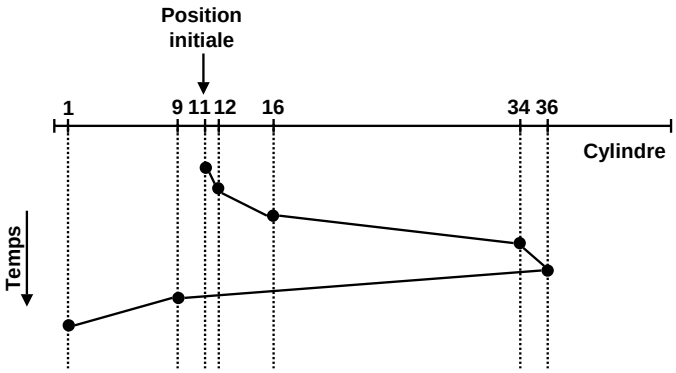


FIG. 13.3 – *SCAN*.

guës, ce qui fait que les requêtes soient proches l'une de l'autre. Si c'est le cas, alors **SSTF** est plus performant.

▶ **Exemple 2.** (Pris de [SGG01]) Soit la séquence de recherche suivante : **98, 183, 37, 122, 14, 124, 65 et 67**. On suppose que la tête se trouve initialement sur la piste **53**, et qu'il existe 200 cylindres. Les résultats sur les algorithmes sont montrés sur les figures 13.4 à 13.7 suivantes.

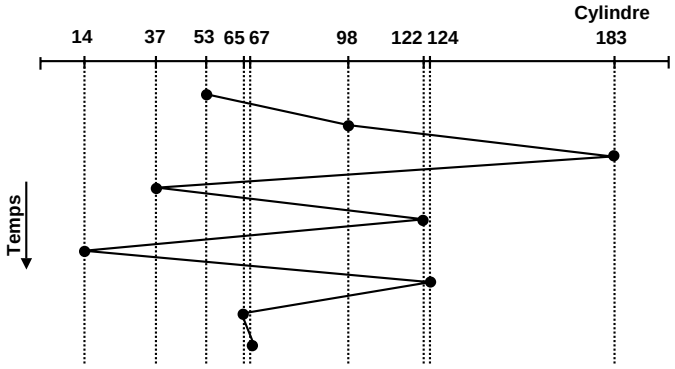


FIG. 13.4 – *Parcours de FCFS = 595 cylindres.*

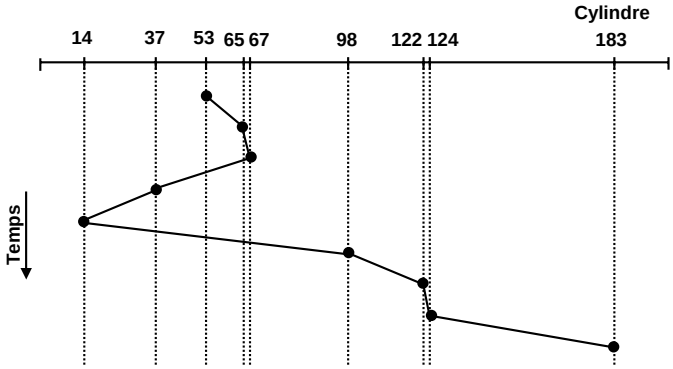


FIG. 13.5 – *Parcours de SSTF = 236 cylindres.*

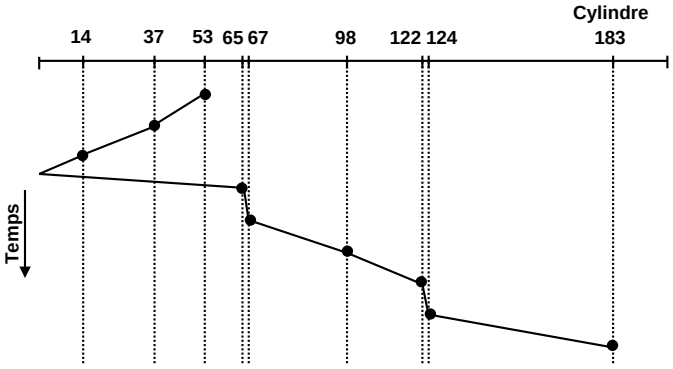


FIG. 13.6 – *Parcours de SCAN = 240 cylindres.*

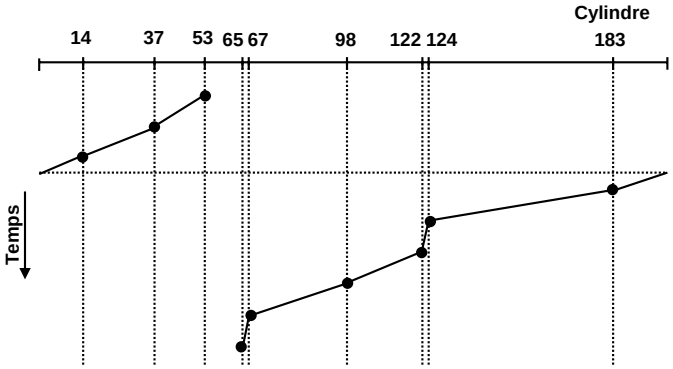


FIG. 13.7 – *Parcours de C-SCAN = 205 cylindres.*

13.6 Exercices

1. Définissez un pilote de périphérique. Pourquoi le pilote est-il implanté dans l'espace système ? Étant dans l'espace système, est-ce que le pilote peut directement accéder à toutes les ressources du noyau ? Expliquez.
2. Dans la technique d'E/S avec interruption, expliquez la façon dont le contrôle d'exécution est retourné au programme d'application qui effectue une opération d'E/S. En d'autres termes, expliquez où est sauvegardée l'adresse de retour de l'appel système.
3. Dans la technique d'E/S avec interruption, le pilote est implanté en deux parties pour effectuer chaque opération d'E/S. La première partie s'occupe d'envoyer la commande au contrôleur, et la deuxième partie gère les signaux d'interruption reçus du contrôleur et aussi retourne le contrôle d'exécution à l'application. Expliquez pourquoi il est nécessaire que le pilote soit implanté en deux parties distinctes pour chaque opération d'E/S.
4. Est-il possible qu'un même processus puisse en même temps compléter une opération d'E/S et exécuter d'autres instructions sur l'UCT ? En d'autres termes, expliquez s'il est possible qu'un processus puisse effectuer une opération d'E/S sans blocage.
5. Est-ce que le fait d'augmenter le nombre de tampons de mémoire dans le contrôleur d'E/S va automatiquement augmenter la performance des opérations d'E/S ? Expliquez.
6. Quel serait un avantage de la technique d'E/S avec accès direct en mémoire sur la performance du système ?
7. Considérons un disque de 40 cylindres. Une demande de lecture du cylindre 11 arrive. Pendant que la recherche est en cours, le pilote reçoit, dans cet ordre, de nouvelles requêtes qui concernent les cylindres 1, 36, 16, 34, 9 et 12. Dans quel ordre seront traitées les requêtes pour les algorithmes suivants :
   (a) Premier arrivé premier servi (FCFS).
   (b) Plus court déplacement d'abord (SSTF).
   (c) Algorithme de l'ascenseur (déplacement montant) (SCAN).
8. On considère un disque présentant 128 pistes numérotées en ordre croissant de l'intérieur vers l'extérieur de 0 à 127. On suppose que la tête de lecture/écriture se trouve présentement placée à la verticale de la piste 15, que sa position précédente était sur la piste 29 et que

des requêtes arrivent pour des accès aux pistes suivantes : 100, 30, 27, 55, 16, 122, 44, 63, 56 (dans cet ordre). Quel serait le déplacement total de la tête de lecture / écriture après que toutes les demandes aient été satisfaites si l'algorithme de planification des déplacements de la tête est :

(a) Premier arrivé premier servi (FCFS)?

(b) Le plus petit déplacement d'abord (SSTF)?

9. Considérez un système de disque magnétique avec les caractéristiques suivantes :
   - Vitesse de rotation du disque de 7200 tours par minute.
   - Disque encodé sur les deux surfaces.
   - Lecteur comportant 5 disques fixes.
   - 256 pistes par surface.
   - Secteurs de 512 octets.
   - 1 Méga-octet par piste.
   - Temps de déplacement de la tête d'une piste à l'autre de 5 μsecs.

   (a) Calculez la capacité totale du système de disque en octets.

   (b) Calculez le nombre de secteurs.

   (c) Calculez la capacité de chaque cylindre du système de disque.

   (d) Calculez le temps nécessaire pour que le disque effectue une seule rotation.

   (e) Calculez le temps nécessaire pour transférer les données d'un secteur.

   (f) Calculez les temps moyens pour se positionner sur une piste et sur un secteur.

10. Considérez des séquences de requêtes pour les pistes 23, 10, 4, 14,27, 11, 5, 17, et supposez que la tête du disque est positionnée initialement sur la piste 8. Pour chacun des algorithmes suivants, calculez le nombre de déplacements moyens de la tête du disque en supposant un disque de 128 pistes dont les numéros croissent de l'extérieur (0) vers l'intérieur (127).

    (a) L'algorithme du premier-arrivé-premier-servi (FCFS).

    (b) L'algorithme du plus petit temps de recherche d'abord (SSTF).

    (c) L'algorithme SCAN.

    (d) L'algorithme C-SCAN.

(e) L'algorithme Look.
(f) L'algorithme Look circulaire.

14
Introduction aux systèmes distribués

Pourquoi les **Systèmes d'exploitation distribués**? Le développement de microprocesseurs puissants et l'invention des réseaux locaux à haut débit (LAN Local Area Networks) ont permis de connecter des dizaines voire des centaines d'ordinateurs pour échanger de petites quantités d'information en une milliseconde (10 Mo/s ou plus). Ces ordinateurs connectés par un réseau de communication constituent un **système distribué**.

14.1 Introduction

Les **Systèmes d'exploitation distribués** permettent le partage des données entre plusieurs utilisateurs, le partage de ressources (périphériques qui sont chers), offrent des facilités de communication entre utilisateurs (membres d'un groupe de travail) et permettent aussi de faire coopérer plusieurs machines à la réalisation d'un grand travail. Cependant les systèmes distribués ont des problèmes particuliers. Par exemple, quelles sortes de systèmes d'exploitation, de langages de programmation et d'application doit-on utiliser? Quel degré de transparence fournir à l'utilisateur? Comment éviter la perte de messages? Il faut considérer aussi des aspects non négligeables de surcharge du réseau et de sécurité.

14.2 Concepts matériels

Si tous les systèmes distribués sont formés de nombreuses UCT, il existe diverses façons pour elles d'être interconnectées et de communiquer.

Multiprocesseurs à bus

Ils sont constitués par un certain nombre de processeurs connectés à un bus commun auquel se rattache également un module de mémoire. Un bus classique comporte 32 fils d'adresse, 32 fils de données et 20 à 30 fils de commande qui travaillent en parallèle. Pour lire un mot en mémoire, l'UCT met l'adresse de ce mot sur les fils d'adresse, puis engendre un signal sur les fils de commande appropriés pour indiquer qu'on veut faire une lecture. La mémoire répond en envoyant la valeur contenue dans le mot mémoire sur les fils de données, ce qui permet à l'UCT de la lire. L'écriture fonctionne de la même façon. Puisqu'il y a une seule mémoire, si l'UCT A écrit un mot mémoire et que l'UCT B lit ce mot une microseconde plus tard, B récupère la valeur qui vient d'être écrite (mémoire cohérente).

Problème : Il y a surcharge du bus dès qu'on atteint 4 ou 5 UCT, ce qui dégrade considérablement les performances.

Solution : L'ajout d'une mémoire cache entre l'UCT et le bus (14.1). Le cache conserve les mots mémoire auxquels on a récemment fait accès. Tous les accès à la mémoire passent par le cache. Si le mot demandé est dans le cache, celui-ci répond lui-même à l'UCT et aucune demande au bus n'est faite.

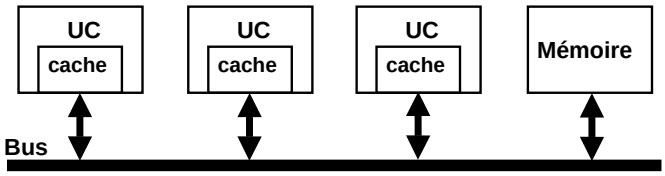


FIG. 14.1 – *Mémoire cache entre processeurs.*

Si le cache est assez grand, la probabilité de succès est élevée et le trafic engendré par chaque UCT sur le bus chute de façon significative. Le nombre d'UCT du système peut alors être augmenté. Avec des caches de 64Ko à 1 Mo, on atteint des taux de succès de 90%.

Problème des caches

Supposons que deux UCT lisent le même mot mémoire dans leur cache. Si l'une modifie la valeur dans le mot (qui se trouve dans le cache), lorsque

l'autre lira de nouveau le mot de son cache, elle récupérera l'ancienne valeur. Ceci est un problème de mémoire incohérente. La solution à ce problème sont les **mémoires caches à écriture immédiate**, qui fonctionnent comme suit : lorsqu'un mot est modifié dans un cache, il est aussitôt modifié en mémoire. De plus, tous les caches doivent constamment surveiller le bus. Chaque fois qu'un cache détecte une écriture à une adresse mémoire dont il possède le contenu, il peut soit invalider l'entrée correspondante soit la mettre à jour avec la nouvelle valeur (cache espion). A peu près, tous les multiprocesseurs à bus utilisent soit cette architecture soit une architecture très similaire. Avec cette méthode, il est possible de mettre 32 à 64 UCT sur un seul bus.

Multiprocesseurs commutés

La mémoire est divisée en modules que l'on relie aux UCT par une matrice de commutation (figure 14.2). Chaque UCT et chaque module mémoire dispose d'une ligne d'interconnexion. A chaque intersection, un élément électronique, le noeud de commutation peut être ouvert ou fermé par matériel. Quand une UCT veut accéder à un certain module de mémoire, on ferme temporairement le nœud de commutation correspondant, ce qui établit le chemin de connexion. L'intérêt de cette matrice de commutation est que plusieurs UCT peuvent accéder en même temps à des modules de mémoire différents.

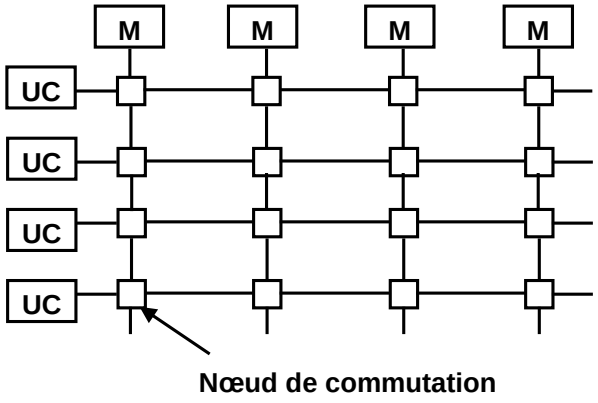


FIG. 14.2 – *Multiprocesseurs commutés.*

Cependant il y a l'inconvénient du nombre de nœuds de commutation, qui est très élevé. Pour n UCT et n modules mémoire, il faut n^2 nœuds de

commutation. On a alors recherché des structures de réseaux d'interconnexion qui nécessitent peu de points de connexion : le **réseau oméga** par exemple, montré à la figure 14.3. Mais il introduit un nouveau problème : le temps de transmission n'est pas satisfaisant.

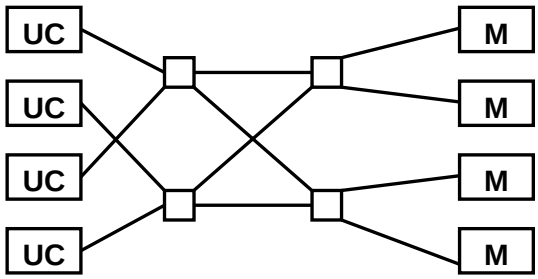


FIG. 14.3 – *Réseau commuté omega.*

Multicalculateurs à bus

Plusieurs calculateurs (stations de travail) sont reliés par un réseau local (LAN) au moyen d'un bus (figure 14.4). Dans ce cas, chaque UCT est directement reliée à sa propre mémoire locale. Le seul problème à régler est la communication des UCT entre elles.

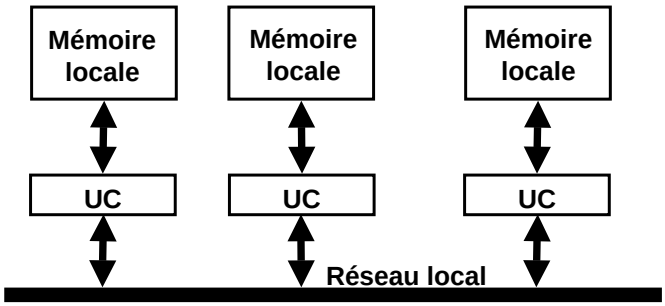


FIG. 14.4 – *Multicalculateurs à bus.*

Multicalculateurs commutés

Les calculateurs sont reliés par un réseau d'interconnexion. Chaque UCT a un accès direct et exclusif à sa mémoire privée. Les UCT communiquent

par envoi de messages qui devront souvent traverser plusieurs nœuds du réseau pour atteindre leur destination.

14.3 Concepts logiciels

Les logiciels qui gèrent des systèmes avec plusieurs UCT peuvent être divisés en trois catégories :

- Les systèmes d'exploitation réseaux.
- Les systèmes d'exploitation distribués.
- Les systèmes d'exploitation multiprocesseurs à mémoire partagée (Remarque : les multiprocesseurs à mémoire partagé ne sont pas des systèmes distribués).

Systèmes d'exploitation réseaux, NFS

Ils tournent sur un type de matériel faiblement couplé (ensemble de machines reliées par un réseau). Chaque utilisateur dispose d'une machine qui lui est réservée et qui a son propre système d'exploitation, c'est ce qu'on appelle une station de travail (figure 14.5).

La première forme de communication utilisée était de transformer la station de travail de l'utilisateur en un terminal distant connecté à la machine distante (il pourra ainsi disposer de toutes les ressources de la machine distante) : `rlogin machine`. Cette forme de communication est utile mais extrêmement rudimentaire. Les concepteurs de systèmes ont très vite recherché un modèle de communication et de partage de données plus élaboré.

Une première amélioration consiste à fournir un système de fichiers globaux et partagés, accessibles à toutes les stations de travail. Le système de fichiers se trouve sur une ou plusieurs machines appelées serveurs de fichiers. Un serveur de fichiers accepte des requêtes de lecture et d'écriture provenant des programmes utilisateurs (les clients). Les systèmes d'exploitation réseaux permettent donc aux utilisateurs de stations de travail indépendantes de communiquer au moyen d'un système de fichiers partagés, tout en étant maître de leur machine. L'un des systèmes d'exploitation réseaux les plus connus est celui de SUN microsystems : *Network File Manager* (**NFS**). Il est utilisé dans Unix / Linux et dans MS-DOS. L'implantation de NFS proposée par SUN est composée de trois couches :

- La couche des appels système qui gère les appels tels que `OPEN`, `READ` et `CLOSE`.

– La couche du système de fichiers virtuels qui gère la table des fichiers ouverts (locaux ou distants).
– La couche noyau et de communication (client NFS et serveur NFS).

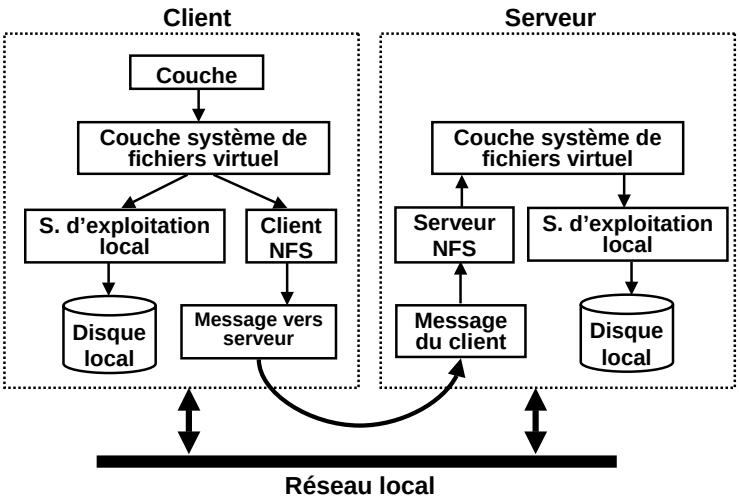


FIG. 14.5 – *Réseau NFS.*

Le problème de l'antémémoire

L'antémémoire risque de rendre les données de fichiers partagés incohérentes. A 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.

Comment NFS traite ce problème ? NFS ne résout pas totalement les incohérences potentielles dues à l'antémémoire ; il tente seulement de les limiter. Pour ce faire, il associe un temporisateur armé, à chaque bloc de l'antémémoire qui provoque à chaque déclenchement, la destruction de l'entrée correspondante (3 s pour les blocs de données et 30 s pour les blocs de répertoires). Les blocs modifiés dans l'antémémoire sont retournés au serveur par un démon s'exécutant toutes les 30 s. Chaque fois qu'un client doit ouvrir un fichier qui est déjà dans l'antémémoire, le système envoie un message au serveur pour récupérer la date de la dernière modification du fichier, puis la compare avec celle de la copie qu'il possède. Dans le cas

où cette dernière date serait plus ancienne, il effectuera une nouvelle lecture des données. Ces précautions n'éliminent cependant pas totalement les risques. Notamment, un fichier peut être créé sur une machine et rester invisible aux autres pendant 30 s. NFS est très largement utilisé dans le monde entier.

Systèmes d'exploitation distribués

Un **Système d'exploitation distribué** est un logiciel fortement couplé qui s'exécute sur un matériel, composé de plusieurs UCT faiblement couplés (plusieurs machines reliées par un réseau). Son but est de donner l'illusion aux utilisateurs que le réseau de calculateurs tout entier est un unique système en temps partagé et non un ensemble de machines distinctes (le vrai système distribué). Pour ce faire, un tel logiciel doit avoir les mécanismes suivants :

- Communication interprocessus : permettre à n'importe quel processus de dialoguer avec n'importe quel autre. Les modèles généralement employés sont les couches de protocoles de l'ISO, le modèle **client/serveur**, les appels de procédures à distance et la communication par groupe.
- Coopération et de synchronisation des processus : pouvoir s'exécuter sur des machines différentes (les sections critiques et l'allocation des ressources).
- Ordonnancement des processus : sans migration ou avec migration.

Systèmes multiprocesseurs en temps partagé

Ce type de systèmes est une combinaison d'un matériel fortement couplé avec un logiciel fortement couplé. Vu de l'extérieur, un multiprocesseur se comporte comme un monoprocesseur. La caractéristique principale de cette classe de systèmes est l'existence d'une file d'attente unique des processus exécutables : une liste de tous les processus existants dans le système qui sont prêts.

Cette file d'attente se trouve dans la mémoire partagée. Les UCT doivent obtenir des accès exclusifs à cette file (par sémaphores). De même, on dispose d'une seule antémémoire pour les Entrées / Sorties qui est considérée comme un objet critique. Lorsqu'on fait un appel système `WRITE`, on verrouille l'antémémoire jusqu'à la fin de l'écriture des données.

14.4 Communication dans les systèmes distribués

Dans un système distribué, toutes les communications se font par échange de messages. Quand un processus A veut communiquer avec un processus B, il commence par élaborer un message dans son propre espace d'adressage. Il exécute ensuite un appel système qui va prendre le message et l'envoyer à B via le réseau. Il faut que A et B s'accordent sur la signification des bits envoyés.

D'une manière générale, le processus émetteur et le récepteur doivent être d'accord sur de nombreux détails, c'est qu'on appelle le **protocole** : Combien de volts faut-il pour indiquer qu'un bit vaut 0 et combien pour 1 ? Comment le destinataire sait-il qu'un bit est le dernier du message ? Comment peut-il détecter si un message est altéré ou perdu ? et que doit-il faire dans ce cas ?, etc. Pour résoudre ces problèmes, l'**ISO** (L'Organisation Internationale de Normalisation) a défini un modèle de référence qui identifie clairement les différents niveaux d'une communication (couches de protocoles). Le **modèle ISO** fait la distinction entre deux types de protocoles :

- Les **protocoles connectés** : ouvrir une connexion, échanger des données, fermer la connexion.
- Les **protocoles sans connexion** : équivalent à mettre un message dans une boîte aux lettres.

Couches de protocoles

Dans le **modèle ISO** (figure 14.6) la communication est partagée en **sept couches**. Chaque couche concerne un aspect spécifique de la communication :

La **couche physique** se charge de la transmission électronique. Combien de volts représentent un 0 et un 1 ? Combien peut-on envoyer de bits par seconde ? La transmission est-elle bidirectionnelle ?

La **couche liaison** se charge de détecter et de corriger les erreurs de transmission. Elle regroupe les bits en forme de **trames** et ajoute une configuration spéciale de bits au début de chaque trame pour la délimiter. Elle additionne selon un algorithme les bits de la trame et met le résultat de cette opération à la fin de la trame. Ce résultat permet au destinataire de vérifier s'il n'y a pas d'erreur dans la trame reçue.

La **couche réseau** se charge d'acheminer les messages à travers le réseau en les faisant passer par un certain nombre de points. Le choix du meilleur

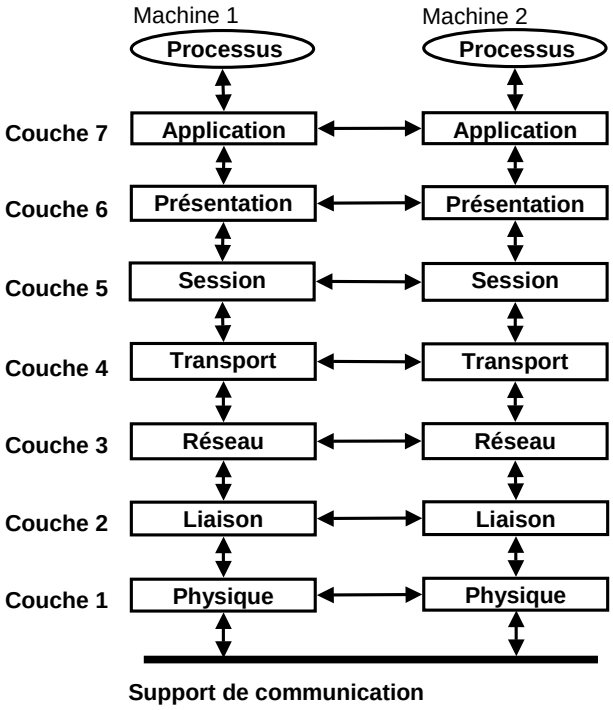


FIG. 14.6 – *Modèle ISO.*

chemin s'appelle le routage. Deux types de protocoles réseaux sont largement utilisés : le protocole orienté connexion `X.25` et le protocole sans connexion `IP` (*Internet Protocol*).

La **couche transport** se charge de veiller à ce que les paquets émis soient reçus. Lorsqu'elle reçoit un message de la couche session, elle le segmente en des entités plus petites (paquets). Elle attribue à chacune un numéro de séquence et les envoie en leur ajoutant une en-tête qui indique quels sont les paquets qui ont été envoyés, ceux qui ont été reçus. Il se charge de détecter et de corriger les déséquencements de paquets. Le protocole `TCP` est l'un des protocoles de transport le plus utilisé surtout en combinaison avec `IP` : `TCP/IP`.

La **couche session** a pour rôle de fournir aux processus de la couche supérieure, les services tels que l'établissement de la connexion de session entre deux applications, l'organisation et la synchronisation de l'échange des messages (lettres).

La **couche présentation** permet de faire communiquer des machines

qui ont des représentations internes différentes. La couche application fournit les services de communication aux utilisateurs (le transfert de fichiers, le courrier électronique).

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 ajoutera des informations qui seront retirées et traitées par la couche correspondante du destinataire. Au niveau le plus bas, il y aura beaucoup plus d'informations échangées, que de données utiles aux utilisateurs.

Parce qu'il y a risque de perte d'efficacité due à la gestion des protocoles, de nombreux systèmes distribués utilisent un sous-ensemble de ces couches, comme montré pour le protocole TCP / IP (figure 14.7).

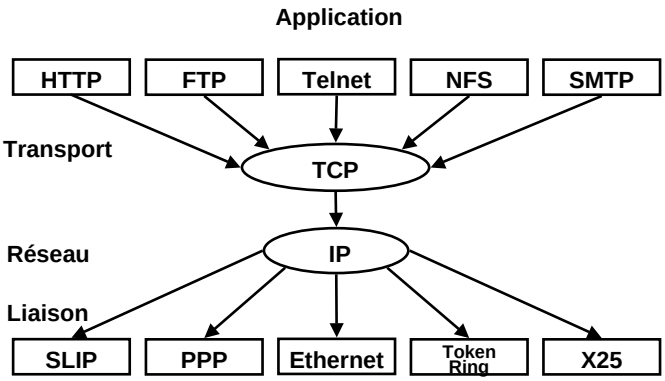


FIG. 14.7 – *Architecture TCP/IP.*

Modèle client/serveur

L'idée du **modèle client/serveur** est de structurer le système d'exploitation en groupes de processus coopérants, appelés **serveurs**, qui offrent des services aux utilisateurs ou **clients**. Les machines serveurs et clients tournent, en général, sous le même micro-kernel. Le client envoie une demande à un serveur qui traite la demande et renvoie une réponse au client. Les services de communication fournis par le micro-kernel peuvent être réduits à deux appels système :

- `send(dest, &buffer)` : envoie le contenu du `buffer` au `dest`.
- `receive(recp, &buffer)` : attend la réception d'un message en provenance de `recp`. Le message est récupéré dans `buffer`.

Pour envoyer un message, l'émetteur (client) doit connaître l'adresse du destinataire (serveur). Cette adresse est par exemple de la forme : `<Numéro de la machine>.<local-id>`. Au moment de son lancement, un processus serveur envoie un appel système au noyau pour lui indiquer le `local-id` sur lequel il se met à l'écoute. Quand un message arrive avec l'adresse `machine.local-id`, le noyau sait donc à quel processus il doit fournir le message. Beaucoup de communications dans **BSD Unix** utilisent cette méthode, avec un nombre de 32 bits pour l'adresse Internet de la machine et un nombre de 16 bits pour le champ `local-id`.

Appels de procédures à distance

Bien que le modèle client / serveur soit un moyen élégant de structurer un système d'exploitation distribué, son problème est qu'il fonde la communication sur les entrées / sorties (`send` et `receive`). Les **appels de procédures à distance** ou **Remote Procedure Calls** permettent d'avoir une meilleure abstraction. Ils imitent les appels de procédures 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*). Les mécanismes de passage de paramètres sont cependant légèrement modifiés. 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'un pointeur sur un tableau, 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[nbr-1], nbr
```

Le passage par paramètres devient un passage par copie / restauration. 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.

Communications de groupes

Les RPC ne traitent que les cas dans lesquels un seul client dialogue avec un seul serveur. Un groupe est un ensemble de processus qui agissent

ensemble d'une façon spécifiée par le système ou l'utilisateur. Un message envoyé à un groupe doit être reçu par tous ses membres (communication de un vers plusieurs). Par exemple, si on dispose d'un groupe constitué de tous les serveurs de fichiers, un client peut émettre sa requête au groupe (tous les serveurs recevront la requête).

14.5 Sockets

Les **sockets** ou **prises** sont des briques de base des mécanismes de communication en systèmes distribués. Elles sont apparues dans Unix 4.2 BSD en 1983. L'étude des sockets est complexe. Nous faisons ici que donner un simple aperçu de leurs capacités.

Les sockets sont manipulées au moyen des descripteurs de fichiers, du même que les fichiers. On peut donc leur appliquer des appels système comme `read()`, `write()`, `close()`, etc. La communication au moyen de **sockets** est basée sur le modèle **client/serveur** et est bidirectionnelle. Le **serveur** est un processus exécutant dans un nœud du réseau. Il gère l'accès à une certaine ressource. Le **client** est un processus qui s'exécute dans le même nœud ou ailleurs. Il effectue des requêtes au serveur. De façon générale, il y a deux types de serveurs : **interactifs** et **concurrents**. Dans les **serveurs interactifs** le serveur prend la requête et lui même fourni la réponse. Dans les **serveurs concurrents** le serveur prend les requêtes, puis il crée d'autres processus qui fourniront les réponses.

Scénario normal

Le scénario normal d'une communication bidirectionnelle client / serveur est montré à la figure 14.8. Les principales actions seront donc les suivantes :

Le serveur :

- Ouvrir le canal de communications puis informer au réseau l'adresse de réponses et sa disposition d'accepter des requêtes.
- Tant qu'il y ait des requêtes de services sur l'adresse publiée.
- Réception d'une requête :
  - S'il s'agit d'un **serveur interactif** il répondra au client.
  - S'il s'agit d'un **serveur concurrent**, il créera un processus fils avec `fork()`. pour répondre au client.

Le client :

– Ouvrir le canal de communications puis se connecter à l'adresse publiée par le serveur. Cette adresse doit être connue par le client.
– Envoyer une requête au client et attendre la réponse.
– Fermer le canal de communications et finir son exécution.

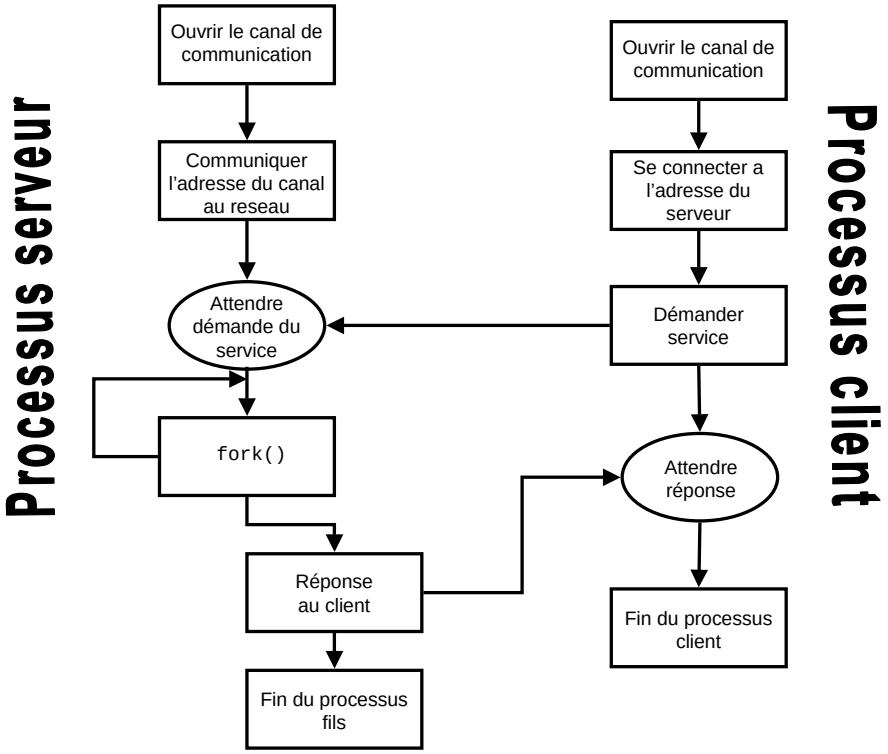


FIG. 14.8 – *Modèle Client-serveur concurrent.*

Schéma de serveur interactif

Listing 14.1: *schema-i.c*

```
int sockfd, newsockfd;

if (( sockfd = socket  ( ... ) < 0 )
    err_sys ("socket error");
if (bind(sockfd  , ... ) < 0 )
```

```
    err_sys ("bind error");
if (listen(sockfd, 5) < 0)
    err_sys ("listen error");
while(1)
{
    newsockfd = accept(sockfd, ...);
    if (newsokfd < 0)
        err_sys ("accept error");
    traiter_requete(newsockfd, ...); // traiter la requête
    close(newsockfd);
} // while
```

Schéma de serveur concurrent

Listing 14.2: *schema-c.c*

```
int sockfd, newsockfd;

if ((sockfd = socket (...) < 0)
    err_sys("socket error");
if (bind(sockfd, ...) < 0)
    err_sys("bind error");
if (listen(sockfd, 5) < 0)
    err_sys ("listen error");
while(1)
{
    newsockfd = accept(sockfd, ...);
    if (newsokfd < 0)
        err_sys("accept error");
    if (!fork())
    {
      close(sockfd); // fils
      traiter_requete(newsockfd, ...); // traiter la requête
      exit (0);
    }
    wait (...)                    // père attendre
    close(newsockfd);
} // while
```

Modes connecté et sans connexion

On peut établir des similitudes entre le mode connecté et les appels téléphoniques : on attends le son de ligne, on digite un numéro (l'adresse), puis on établi un simple protocole ("*allô ?*") et finalement on parle (l'envoi de messages peut être avec renvoi ou correction d'erreurs). On n'a pas besoin de rétablir le protocole ni l'adresse du correspondant à tout moment, tant qu'on est branché. C'est un mode fiable. Le scénario d'une communication **connectée** est illustré à la figure 14.9.

Par contre, le mode non connecté ressemble plutôt au courrier (ou à l'*e-mail*) car il nécessite l'ajout d'une enveloppe avec l'adresse pour chaque lettre (message). Les lettres sont envoyées dans un certain ordre, mais l'ordre d'arrivée ne sera peut être pas le même. Elles peuvent encore se perdre. C'est un mode non fiable. La communication orientée **non connexion** est illustré à la figure 14.10.

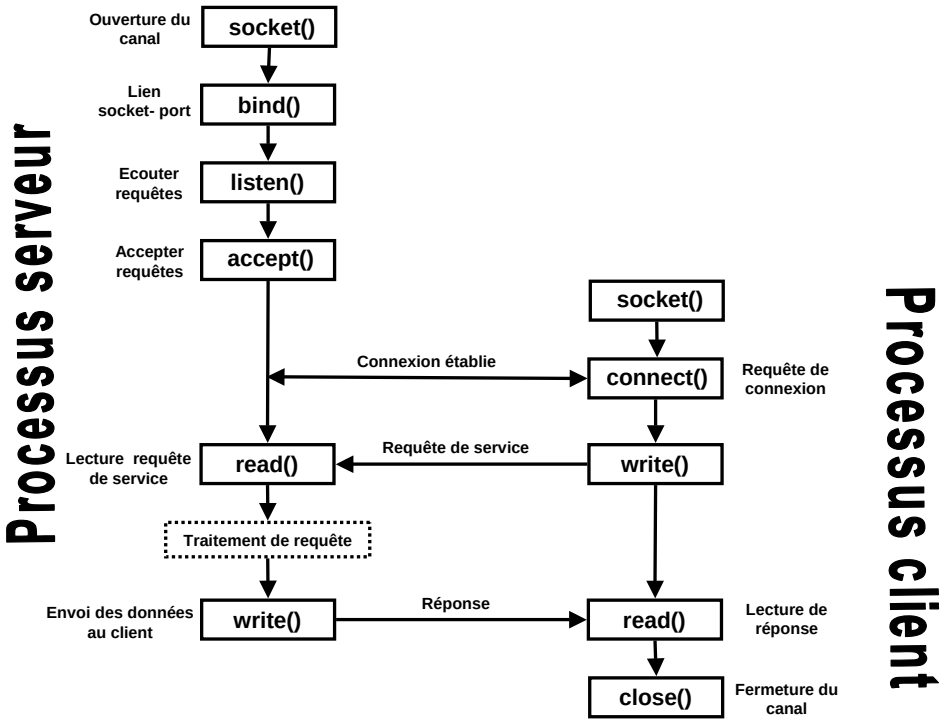


FIG. 14.9 – *Modèle Client-serveur connecté.*

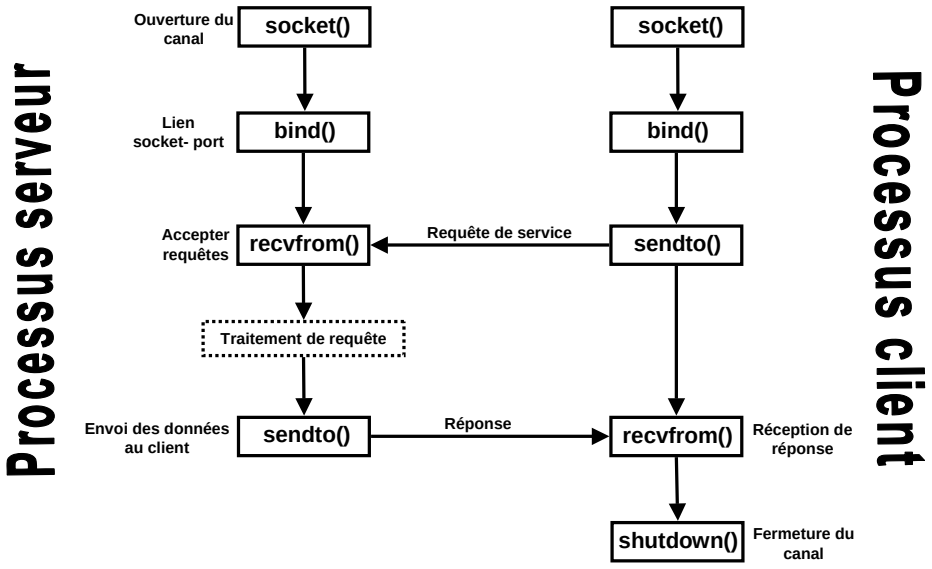


FIG. 14.10 – *Modèle Client-serveur non connecté.*

Adresses du réseau

```
#include <sys/socket.h>
struct sockaddr  {
   unsigned short int sa_family; /* Famille de sockets: AF_xxx */
   unsigned char  sa_data[14];   /* 14 octets avec l'adresse   */
                                 /* spécifique du protocole */
};
```

Familles de protocoles Internet

```
#include <netinet/in.h>
struct in_addr {
   u_long s_addr; /* 32 bits pour identifier réseau et hôte */
}

struct sockaddr_in {
   short int          sin_family; /* AF_INET*/
   unsigned short int  sin_port;   /* numéro de port */
   struct in_addr      sin_addr;   /* adresse IP */
   char sin_zero [8] ;             /* 8 bytes non utilisés */
}
```

Pour stocker l'adresse d'une socket il faut utiliser la structure `in_addr`. Le champ `sin_port` réfère le numéro de port *dans l'ordre des octets* du réseau.

Familles de protocoles Unix

```
#include <sys/un.h>
struct sockaddr {
   unsigned short sun_family; /* AF_UNIX */
   unsigned char sun_path[108]; /* path name*/
};
```

Ordre des octets

Dans le protocole TCP/IP les numéros sont utilisés en format *Big-endian*. Sur des ordinateurs qui n'utilisent pas ce format[1], il est nécessaire d'utiliser des fonctions pour traduire les numéros entre le format qui utilise TCP/IP et celui employé par l'ordinateur même :

```
#include <netinet/in.h>
u_long  htonl(u_long hostlong);
u_long  ntohl(u_long netlong);"
u_short htons(u_short hostshort);
u_short ntohs(u_short netshort);
```

`htonl()` traduit un numéro de 32 bits représenté dans l'ordre du format de l'ordinateur au format du réseau. `ntohl()` traduit un numéro de 32 bits représenté dans l'ordre du format du réseau à celui de l'ordinateur. Les deux autres fonctions font pareil mais avec des entiers courts.

Informations de l'hôte

Les informations concernant une machine **hôte** sont regroupées dans la structure `hostent`, avec les champs :

- `h_name` : `char *` Nom de l'hôte
- `h_aliases` : `char **` Liste d'alias
- `h_addrtype` : Type d'adresse `AF_INET` ou `AF_INET6`
- `h_lenght` : `int` Longueur des adresses

1. Notamment les processeurs Intel 80x86 utilisent le format little endian. Dans [Bla02] et [BG02] le lecteur trouvera plus d'information sur les ordres des octets *Big-endian* et *little-endian*.

- `h_addr_list`: `char **` Liste d'adresses de l'hôte
- `h_addr`: `char * h_addr_list[0]`

14.5.1 Appels système

Les principaux appels système[2] qui permettent de créer, d'établir une connexion et d'échanger des informations au moyen des sockets sont :

- `socket()` : Établir une connexion.
- `listen()` : Écouter.
- `connect()` : Connecter.
- `accept()` : Accepter des données.
- `bind()` : Étiquetter une socket serveur avec une adresse.
- `close()` : Fermer la socket
- `setsockopt()` : Réutilisation de la socket

`socket()`

```
#include <sys/socket.h>
int socket(int domaine, int type, int protocole);
```

Les arguments de `socket()` sont :

- `domaine`. On le fixe à `AF_INET` pour les protocoles Internet ou comme `AF_UNIX` ou `AF_LOCAL`[3] pour des sockets en communication interne. Il y a d'autres types de domaine comme `AF_INET6` (protocole IPng), `AF_IPX` (Novell), etc.
- `type`. Ici on spécifie le type de socket à utiliser : **flux** ou **datagrammes**. Les constantes symboliques sont `SOCK_STREAM` ou `SOCK_DGRAM` respectivement.
- `protocole`. Il s'agit du protocole d'utilisation. On peut consulter le fichier `/etc/protocol` pour vérifier ceux reconnus par le système. Si `protocole = 0` alors le `domaine = AF_INET` et le `type = SOCK_STREAM`, ce qui correspond au protocole TCP / IP.

`socket()` retourne un descripteur de socket, lequel on peut utiliser pour des appels système. Il retourne `-1` s'il y a eu erreur.

2. La norme Posix 1003.1g veut standardiser les appels système des sockets.
3. Posix recommande d'utiliser `AF_LOCAL`.

`listen()`

```
#include <sys/socket.h>
int listen(int sock, int nb_attente);
```

L'appel système `listen()` indique qu'on attend des connexions sur une socket `sock`. le `nb_attente` est une valeur entre `0` et `128` qui spécifie le nombre de connexions en attente. `5` est une valeur largement utilisée.

`connect()`

```
#include <sys/socket.h>
int connect(int sock, struct sockaddr *adresse,
            socklen_t longueur);
```

L'appel système `connect()` sert —du côté du client— à connecter le serveur dont l'`adresse` est passée. Le troisième argument est la `longueur` de la structure en deuxième position.

`accept()`

```
#include <sys/socket.h>
int accept(int sock, struct sockaddr *adresse,
           socklen_t longueur);
```

`listen()` n'est pas bloquant, et informe au noyau qu'on désire établir des connexions. L'appel système `accept()` met le programme effectivement en attente.

`bind()`

```
#include <sys/socket.h>
int bind(int sock, struct sockaddr *adresse,
         socklen_t longueur);
```

L'appel système `bind()` va affecter une identité à la socket `sock` serveur.

`close()`

```
#include <sys/socket.h>
int close(int sock);
```

L'appel système `close()` ferme la socket associée au descripteur `sock`.

`setsockopt()`

```
#include <sys/socket.h>
int i=1;
setsockopt(int sock,SOL_SOCKET,SO_REUSEADDR,&i,sizeof(i));
```

`setsockopt()` sert —entre autres— à la réutilisation des ports. Si on l'utilise tel qu'il est montré, on n'a pas besoin d'attendre un temps pour que le port devienne disponible.

`gethostbyname()`

```
#include <netdb.h>
struct hostent gethostbyname(char *nom);
```

La fonction `gethostbyname()` obtient la structure d'un hôte à partir d'un `nom` de serveur.

`inet_ntoa()`

```
#include <netinet/in.h>
char *inet_ntoa(struct in_addr adresse);
```

La fonction `inet_ntoa()` transforme une adresse `in_addr` en notation pointée.

14.5.2 Sockets ou tubes?

Les sockets se comparent avantageusement par rapport à d'autres mécanismes de communication, comme les tubes ou *pipes*. Le seul inconvénient est le code additionnel pour établir la communication (tout a un prix !), mais une fois que cela est fait, l'utilisation de la socket demeure au même niveau de simplicité que l'utilisation des descripteurs de fichier. La table 14.1 montre cette comparaison.

Sockets	Pipes
référence par descripteurs	référence par array de descripteurs
communication entre processus de machines différentes	communication entre processus de la même machine
communication bidirectionnelle	communication unidirectionnelle
philosophie client-serveur	échange simple d'information

TAB. 14.1 – *Comparaison de sockets vs. pipes.*

14.5.3 Sockets locales

Les **sockets locales** utilisent les synonymes de domaine `PF_LOCAL` ou `PF_UNIX`. Elles sont dans le domaine Unix et résident dans le système de fichiers sous forme d'un fichier de type socket. Son entrée commence par `"s"` sur le listage de `ls`. Le seul protocole admis est `0`.

▶ **Exemple 1.** Client-serveur dans la même machine :

Listing 14.3: *sock-client.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/socket.h>
#include <unistd.h>
#include <sys/un.h>
#include <string.h>

void write_text(int socket_fd, const char *text)
{
    int length = strlen(text)+1;

    write(socket_fd, &length, sizeof(length));
    write(socket_fd, text, length);
}

int main(int argc, char *argv[])
{
 const char* const socket_name = argv[1];
 const char* const message = argv[2];
 int socket_fd;
 struct sockaddr_un name;

 socket_fd = socket(PF_LOCAL, SOCK_STREAM, 0);
 name.sun_family = AF_LOCAL;
 strcpy(name.sun_path, socket_name);
 connect(socket_fd, &name, SUN_LEN(&name));
 write_text(socket_fd, message);
 close(socket_fd);
 return 0;
}
```

Listing 14.4: *sock-server.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/socket.h>
#include <unistd.h>
#include <sys/un.h>
#include <string.h>

int server(int client_socket)
{
  while(1)
  {
    int length;
    char *text;

    if (read(client_socket,&length, sizeof(length))==0) return 0;

    text = (char *) malloc(length);

    read(client_socket, text, length);
    printf("%s\n",text);

    free(text);

    if (!strcmp(text,"quit")) return 1;
  }
}

int main(int argc, char *argv[])
{
  const char* const socket_name = argv[1];
  int socket_fd;
  struct sockaddr_un name;
  int client_sent_quit_message;

  socket_fd = socket(PF_LOCAL, SOCK_STREAM, 0);
  name.sun_family = AF_LOCAL;
  strcpy(name.sun_path, socket_name);
  bind(socket_fd, &name, SUN_LEN(&name));
```

```
  listen (socket_fd ,5);

  do
  {
    struct  sockaddr_un client_name;
    socklen_t client_name_len;
    int  client_socket_fd ;

    client_socket_fd  =  accept(socket_fd, &client_name, &client_name_len);

    client_sent_quit_message = server(client_socket_fd );

    close(client_socket_fd );
  } while (!client_sent_quit_message);

  close(socket_fd);
  unlink(socket_name);
  return 0;
}
```

Exemple d'exécution de `sock-server.c` et `sock-server.c`. Dans une fenêtre compiler et lancer le serveur avec un nom de sockect locale, par exemple `sock` :

```
pascal> gcc -o serveur sock-server.c
pascal> serveur sock
_
```

Dans une autre fenêtre shell, lancer la commande `ls -l sock` pour vérifier que la socket est bien créée. La `"s"` indique socket. Puis, compiler `sock-client.c` et lancer un client en utilisant `sock` et un message :

```
pascal> ls -l sock
srwxr-xr-x    1 jmtorres prof      0 Nov 21 14:44 sock=
pascal> gcc -o client sock-client.c
pascal> client sock Bonjour!
pascal> client sock message
pascal> _
```

Dans le premier écran, les messages apparaissent :

```
pascal> serveur sock
Bonjour!
message
_
```

On peut ouvrir un autre shell dans une troisième fenêtre, et lancer un autre client avec des messages et peut-être la commande `quit` que nous avons définie pour quitter le serveur :

```
pascal> client sock Hola!
pascal> client sock Adios!
pascal> client sock quit
pascal>
```

Du côté du serveur on montre les messages et on finit l'exécution :

```
pascal> serveur sock
Bonjour!
message
Hola!
Adios!
quit
pascal> _
```

14.5.4 Sockets dans le domaine Internet

DNS sont les sigles de *Domain Name Service* qui peut se traduire par « **Service de Noms de Domaine** » et qui est utilisé pour trouver des adresses `IP`. Par exemple, l'adresse `IP` du serveur `www.udg.mx` codé sur 4 octets en **notation pointée** est `148.202.105.112`.

Pour utiliser des **sockets dans le domaine Internet** on met `AF_INET`. Un protocole très utilisé est celui de `TCP/IP`. Les numéros de port sont des nombres entiers sur 16 bits. S'ils sont $\leq$ `255` alors ils appartient à des services bien définis (ainsi `ftp 21`, `mail 25`, `http 80`, etc.). On peut consulter `/etc/services` pour plus d'information. Les numéros de port $\leq$ `1024` appartient à des services avec `UID NULL`, ce qui empêche les intrusions non désirées au système.

▶ **Exemple 2.** Obtenir l'adresse **IP** d'un serveur. Pour ce programme nous aurons besoin de `gethostbyname()` et de `inet_ntoa()` :

Listing 14.5: *gethost.c*

```
#include <stdio.h>
#include <netdb.h>   // gethostbyname()

 int  main(int argc, char *argv[])
 {
```

```
    struct hostent *he;

    if (argc!=2)
    {
        printf("Utilisation : %s <hostname>\n",argv[0]);
        exit(-1);
    }
    // obtenir adresse par le nom
    he = gethostbyname(argv[1]);
    // Nom de l'hôte
    printf("Nom de l'hôte : %s\n",he->h_name);
    // Adresse IP
    printf("Addresse IP : %s\n",
            inet_ntoa(*((struct in_addr *)he->h_addr)));
    return 0;
}
```

Exécution de `gethost.c`:

```
leibnitz> gethost www.polymtl.ca
Nom de l'hôte : www.polymtl.ca
Addressse IP : 132.207.4.56
leibnitz> gethost www.cuentos.da.ru
Nom de l'hôte : www.cuentos.da.ru
Addressse IP : 195.161.113.101
leibnitz> gethost www.udg.mx
Nom de l'hôte : www.udg.mx
Addressse IP : 148.202.105.112
```

▶ **Exemple 3.** L'exemple suivant `sock-inet.c` utilise des sockets sur le domaine Internet. Il se connecte à un serveur `HTTP` et obtient la page web index sur le serveur, puis il l'affiche.

Listing 14.6: *sock-inet.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>
```

```
#define REQUETE "GET / HTTP/1.0\n\n"
#define CAR 10000

void get_home_page(int socket_fd)
{
  int characters_read;
  char buf[CAR];

  write(socket_fd, REQUETE, strlen(REQUETE));

  while(1)
  {
    characters_read = read(socket_fd, buf, CAR);
    if (!characters_read) return;
    write(1, buf, characters_read);
  }
}

int main(int argc, char *argv[])
{
  int socket_fd;
  struct sockaddr_in name;
  struct hostent* hostinfo;

  socket_fd = socket(PF_INET, SOCK_STREAM, 0);
  name.sin_family = AF_INET;
  hostinfo = gethostbyname(argv[1]);
  if (hostinfo == NULL)
    return 1;
  name.sin_addr = *((struct in_addr *) hostinfo->h_addr);
  name.sin_port = htons(80);
  if (connect(socket_fd, &name, sizeof(struct sockaddr_in)) == -1)
  {
    perror("connect");
    return 1;
  }
  get_home_page(socket_fd);
  return 0;
}
```

Un exemple d'exécution de `sock-inet.c` sur le serveur de l'École Polytechnique de Montréal est montré à continuation :

```
leibnitz> sock-inet www.polymtl.ca
HTTP/1.1 200 OK
Date: Thu, 21 Nov 2002 20:50:36 GMT
Server: Apache/1.3.26 (Unix) FrontPage/4.0.4.3
        AuthMySQL/2.20 PHP/4.1.2
Last-Modified: Mon, 14 Jan 2002 20:46:48 GMT
ETag: "8ff12-51a-3c4343b8"
Accept-Ranges: bytes Content-Length: 1306
Connection: close
Content-Type: text/html

<html>
<head> <title>&Eacute;cole Polytechnique de Montr&eacute;al
       </title>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1"> <meta name="descrip
       tion"
content="École Polytechnique aims to provide quality tr
       aining in
engineering at both the undergraduate and graduate levels.
       The Master'

                              ⋮

onLoad="MM_goToURL('parent','fr/fr/index. php');return
document.MM_returnValue">
</body>
</html>
```

D'ailleurs, nous aurons obtenu la même réponse avec une connexion `ftp` sur le port `80` :

```
leibnitz> telnet www.polymtl.ca 80
Trying 132.207.4.56...
Connected to www.polymtl.ca.
Escape character is '^]'. GET /
HTTP/1.0

HTTP/1.1 200 OK
Date: Thu, 21 Nov 2002 20:50:36 GMT
Server: Apache/1.3.26 (Unix) FrontPage/4.0.4.3 AuthMySQL/
```

```
        2.20 PHP/4.1.2
 ...
```

▶ **Exemple 4.** Nous montrons un exemple entre un serveur et un client sur des machines différentes. Le client envoi une requête pour additionner 2 nombres entiers, le serveur les additionne, puis envoi le résultat qui est affiché par le client. Le contrôle d'erreurs n'existe pas afin d'alléger le code.

Listing 14.7: *suma-client.c*

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>          /* socket INET */
//# include <arpa/inet.h>
#include <netdb.h>

int main(int argc, char *argv[])
{
        int sd;
        struct sockaddr_in server_addr;
        struct hostent *hp;
        int num[2], res;

        sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        bzero((char *)&server_addr, sizeof(server_addr));
        hp = gethostbyname ("leibnitz.info.polymtl.ca");
        memcpy (&(server_addr.sin_addr), hp->h_addr, hp->h_length);
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = 4200;
       // s'établit la connexion
        connect(sd, (struct sockaddr *) &server_addr,
        num[0]=atoi(argv[1]);
        num[1]=atoi(argv[2]);;
        write(sd, (char *) num, 2*sizeof(int)); // envoi la demande
        read(sd, &res, sizeof(int));      // recu la réponse
        printf("Resultat est %d \n", res);
        close (sd);
        exit (0);
}
```

Listing 14.8: *suma-server.c*

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

int main(int argc, char *argv[])
{
        struct sockaddr_in server_addr, client_addr;
10      int sd, sc;
        int size, val;
        int num[2], res;

        sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        val = 1;
        setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
                (char *) &val, sizeof(int));

        bzero((char *)&server_addr, sizeof(server_addr));
20      server_addr.sin_family = AF_INET;
        server_addr.sin_addr.s_addr = INADDR_ANY;
        server_addr.sin_port = 4200;
        bind(sd, &server_addr, sizeof(server_addr));
        listen(sd, 5);
        size = sizeof(client_addr);
        while (1)
        {
          write(1,"attente connexion\n",18);
          sc = accept(sd, (struct sockaddr *) &client_addr, &size);
30        read(sc, (char *) num, 2 *sizeof(int));    // reçoit la demande
          res = num[0] + num[1];
          write(sc, &res, sizeof(int)); // envoi le resultat
          close(sc);
        }
        close (sd);
        exit (0);
}
```

L'exécution est montrée sur les machines `leibnitz` et `pascal`, toutes les

deux dans le domaine Internet `info.polymtl.ca`:

```
leibnitz> gcc -o suma-server suma-server.c
leibnitz> suma-server
attente connexion
attente connexion
attente connexion
...

pascal> gcc -o sc suma-client.c
pascal> suma-client 5 -2
Résultat est 3
pascal> suma-client 9 5
Résultat est 14
pascal> suma-client 1 99
Résultat est 100
...
```

▶ **Exemple 5.** Finalement voici un exemple d'une communication non connectée avec le protocole UDP, un serveur de temps `time-server.c`:

Listing 14.9: *time-client.c*

```
#include <stdio.h>
#include <unistd.h>       // info du hôte et fork()
#include <netdb.h>        // info du hôte
#include <sys/param.h>    // MAXHOSTNAMELEN
#include <sys/socket.h>
#include <malloc.h>
#include <netinet/in.h>   // Définition sockaddr_in
#include <sys/types.h>
#include <time.h>         // time
10

#define TAM 50

int main(int argc, char **argv)
{
 int sock;                // socket
                          // structure : puerto, @IP, protocole
 struct sockaddr_in dir_socket_C;
                          // structure : puerto, @IP, protocole
 struct sockaddr_in dir_socket_S;
```

```
20  struct sockaddr_in dir_socket_R; // info réception
    char * name_host_S;   // nom du host Serveur
    char * name_host_C;   // nom du host Client
                          // structure info serveur
    struct hostent *info_host_S;
                         // structure info client
    struct hostent *info_host_C;
    struct in_addr ** addr_host_S;   // IP serveur
    struct in_addr ** addr_host_C;   // IP client
    int size,tmp;
30  char mensaje_C[TAM];
    char * mensaje_tiempo;
    int puertoS;                     //Port du serveur
    int puertoC;                     //Port du client

    // Mémoire pour variables
    mensaje_tiempo= (char *) malloc (sizeof(char)*TAM);
    name_host_S= (char *) malloc (sizeof(char)*50);
    info_host_S= (struct hostent *)
                 malloc (sizeof(struct hostent));
40  addr_host_S= (struct in_addr **)
                 malloc (sizeof(struct in_addr)*10);
    name_host_C= (char *) malloc (sizeof(char)*50);
    info_host_C= (struct hostent *)
                 malloc (sizeof(struct hostent));
    addr_host_C= (struct in_addr **)
                 malloc (sizeof(struct in_addr)*10);
    name_host_S=argv[1];
    puertoS=atoi(argv[2]);
    sock=socket(AF_INET,SOCK_DGRAM,0);
50  // nom de l'hôte local
    gethostname(name_host_C,MAXHOSTNAMELEN);
    // adresse locale de l'hôte
    info_host_C = gethostbyname(name_host_C);
    printf("HOST client:%s\n",name_host_C);

    addr_host_C= (struct in_addr **)info_host_C->h_addr_list;
                                     // adresse local host
    dir_socket_C.sin_addr= ** addr_host_C;
    dir_socket_C.sin_port= 0;        // port
```

```
  dir_socket_C.sin_family= AF_INET; // protocole

  printf(" Num socket : %d \n",sock);
  printf(" Num port  : %d \n",dir_socket_C.sin_port);
  printf(" Addr host  : %s \n",
         (char *)inet_ntoa(dir_socket_C.sin_addr));
  printf(" Famille    : %d \n",dir_socket_C.sin_family);
  tmp=1;
  setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,&tmp,sizeof(tmp));
  bind(sock,(struct sockaddr *) &dir_socket_C,
           sizeof(dir_socket_C));
  printf("RETURN pour demander l'heure au serveur \n");
  fgets(mensaje_C,TAM,stdin);
  // adresse serveur
  info_host_S=gethostbyname(name_host_S);
  addr_host_S= (struct in_addr **)info_host_S->h_addr_list;
  printf("HOST Server= %s PORT=%d \n",info_host_S->h_name,
         puertoS);
  dir_socket_S.sin_addr= ** addr_host_S;  // Adresse serveur
//  dir_socket_S . sin_port = htons(puertoS ); // Port serveur
  dir_socket_S.sin_port= puertoS;          // Port serveur
  dir_socket_S.sin_family= AF_INET;
  size=sizeof(dir_socket_R);
  sendto(sock,mensaje_C,TAM,0,&dir_socket_S,size);
  // Retourner le temps
  size=sizeof(dir_socket_R);
  recvfrom(sock,mensaje_tiempo,TAM,0,&dir_socket_R,&size);
  printf("%s \n",mensaje_tiempo);
  close(sock);
}
```

Listing 14.10: *time-server.c*

```
#include <stdio.h>
#include <unistd.h>       // info du host et fork()
#include <netdb.h>        // info du host
#include <sys/param.h>    // MAXHOSTNAMELEN
#include <sys/socket.h>   // sockets
#include <malloc.h>
#include <netinet/in.h>   // sockaddr_in
#include <sys/types.h>
```

```
#include <time.h>          // time
10
#define TAM 50

int main(int argc, char **argv)
{
 int sock_S;                       // identificateur de socket
                                   // structure : port ,@IP, protocole
 struct sockaddr_in dir_socket_S;
                                   // structure : port ,@IP, protocole
 struct sockaddr_in dir_socket_R;
20                                 // info réception
 char * name_host_S;               // nom host local
 struct hostent *info_host_S; // structure info du host
 struct in_addr **addr_host_S;//IP du host
 int size,tmp;
 char mensaje_C[TAM];
 char *mensaje_tiempo;
 time_t tiempo;
 int puerto=atoi(argv[1]);
 // Mémoire de variables
30 name_host_S= (char *) malloc (sizeof(char)*TAM);
 mensaje_tiempo= (char *) malloc (sizeof(char)*TAM);
 info_host_S= (struct hostent *)
              malloc (sizeof(struct hostent));
 addr_host_S= (struct in_addr **)
              malloc (sizeof(struct in_addr)*10);

 sock_S=socket(AF_INET,SOCK_DGRAM,0); // IPPROT_UDP
 // nom du hôte local
 gethostname(name_host_S, MAXHOSTNAMELEN);
40 // adresse de l'hôte
 info_host_S=gethostbyname(name_host_S);
 printf("HOST:%s\n",name_host_S);
 addr_host_S= (struct in_addr **)
              info_host_S->h_addr_list;
                                   // adresse local host
 dir_socket_S.sin_addr= ** addr_host_S;
//  dir_socket_S.sin_port = htons(puerto ); // port
 dir_socket_S.sin_port= (puerto);  // port
```

```
  dir_socket_S.sin_family= AF_INET; // protocole
50
  printf(" Num socket : %d \n",sock_S);
  printf(" Num port  : %d \n",dir_socket_S.sin_port);
  printf(" Addr host  : %s \n",
         (char *)inet_ntoa(dir_socket_S.sin_addr));
  printf(" Famille    : %d \n",dir_socket_S.sin_family);

  tmp=1;
  setsockopt(sock_S,SOL_SOCKET,
             SO_REUSEADDR,&tmp,sizeof(tmp));
60 bind(sock_S,(struct sockaddr *) &dir_socket_S,
      sizeof(dir_socket_S));
  while (1)
    {
      printf("Serveur> ");
      size=sizeof(dir_socket_R);
      recvfrom(sock_S,mensaje_C,TAM,0,&dir_socket_R,
               &size);
      printf("%s \n",mensaje_C);

70    // obtenir temps
      tiempo=time(NULL);
      mensaje_tiempo=ctime((time_t *)&tiempo);
      printf("Temps envoye=%s \n",mensaje_tiempo);
      size=sizeof(dir_socket_R);
      sendto(sock_S,mensaje_tiempo,TAM,0,&dir_socket_R,
             size );
    }
   close(sock_S);
}
```

Nous allons montrer l'exécution de `time-client.c` et de `time-serveur.c` sur les machines `pascal` et `leibnitz`. D'abord on compile le client sur `leibnitz`:

```
leibnitz> gcc -o tc time-client.c
```

Puis on compile le serveur sur `pascal` et on l'exécute:

```
pascal> gcc -o ts time-seveur.c
```

```
pascal> ts 1234
HOST:pascal
 Num socket  : 3
 Num port    : 1234
 Addr host   : 132.207.12.139
 Famille     : 2
Serveur>
```

Ensuite on exécute le client sur `leibnitz`. On donne le nom de la machine serveur et le numéro de port `1234`, et l'exécution reste gelée jusqu'à ce qu'on tape `<RETURN>` :

```
leibnitz> tc pascal.info.polymtl.ca 1234
HOST client:leibnitz
 Num socket : 3
 Num port   : 0
 Addr host  : 132.207.12.68
 Famille    : 2
RETURN pour demander l'heure au serveur
...
```

Quand ont tape `<RETURN>` sur `leibnitz` :

```
...
HOST Server= pascal.info.polymtl.ca PUERTO=1234
Tue Nov 26 17:01:36 2002
leibnitz> _
```

le serveur sur `pascal` répond avec :

```
 pascal> ts 1234
 HOST:pascal
 ...
 Serveur> _
Temps envoye=Tue Nov 26 16:59:31 2002
Serveur> _
```

Suggestions de lecture

Référence : Silverwschatz A., Galvin P., Gagné G., *Applied Operating System Concepts*, Wiley, 2003.

Chapître 10

A
Opérateurs et variables shell

`test` et `[ ]`

Opérateur	Description
! expression	True si l'expression est fausse. Elle peut être utilisée pour inverser n'importe quel autre test
-b fichier	True si le fichier est un périphérique bloc
-c fichier	True si le fichier est un périphérique caractère
-d fichier	True si le fichier est un répertoire
-e fichier	True si le fichier existe
-f fichier	True si le fichier est un fichier normal
-p fichier	True si le fichier est un FIFO
-L fichier	True si le fichier est un lien symbolique
-S fichier	True si le fichier est un socket Unix
-G fichier	True si le fichier appartient au même groupe que le GID effectif du processus courant
-n chaine	True si la longueur de la chaîne $\neq 0$
-z chaine	True si la longueur de la chaîne = 0
-O fichier	True si le fichier appartient à la même personne que l'UID effectif du processus courant
-g fichier	True si le fichier a le bit setgid armé
-r fichier	True si les permissions de l'utilisateur sont suffisantes pour lire le fichier
-k fichier	True si le fichier a le bit sticky armé
-t [fd]	True si le descripteur de fichier correspond à un terminal réel. fd=1 par défaut (sortie standard)

-u fichier	True si le fichier a le bit setuid armé
-w fichier	True si les permissions de l'utilisateur courant sont suffisantes pour écrire dans le fichier spécifié
-x fichier	True si les permissions de l'utilisateur courant sont suffisantes pour exécuter le fichier spécifié
Expression1 -a expression2	True si les deux expressions spécifiées sont vraies
Expression1 -o expression2	True si au moins une des expressions spécifiées est vraie
fichier1 -ef fichier2	True si les deux fichiers correspondent aut même numéro d'inœud sur le même périphérique
fichier1 -nt fichier2	True si la date de dernière modification du fichier1 est plus récente que celle du fichier2
fichier1 -ot fichier2	True si la date de dernière modification du premier fichier est plus ancienne que celle du second
nombre1 -eq nombre2	True si nombre1 = nombre2
nombre1 -ne nombre2	True si nombre1 $\neq$ nombre2
nombre1 -le nombre2	True si nombre1 $\leq$ nombre2
nombre1 -lt nombre2	True si nombre1 $<$ nombre2
nombre1 -ge nombre2	True si nombre1 $\geq$ nombre2
nombre1 -gt nombre2	True si nombre $>$ nombre2
chaine1 = chaine2	True si les deux chaînes sont égales
chaine1 != chaine2	True si les chaînes ne sont pas égales

Variables bash spéciales

W = écriture ; R = lecture

Variable	Accès	Description
!	R	ID du processus en arrière-plan le plus récent
@	R	Contient tous les paramètres du contexte courant Lorsqu'elle est utilisée entre des délimiteurs doubles, elle est évaluée comme des valeurs délimitées séparées, une pour chaque paramètre passé au contexte courant
#	R	Nombre de paramètres du contexte courant
*	R	Tous les paramètres du contexte courant Si elle est utilisée entre des délimiteurs doubles, le résultat est un unique paramètre contenant tous les paramètres passés, séparés par des espaces
$	R	ID de processus bash actuel
-	R	Liste des flags d'option courants (de la commande), une lettre pour chacun, sans séparation
_	R	Nom de chemin complet du processus courant au cours de l'initialisation. En recherche de mail, contient le nom du fichier de mail courant Dans les autres cas, contient l'argument final de la commande précédente
0	R	Nom du processus ou script en cours
1 à 9	R	9 premiers paramètres du script (ou de la fonction) courant
BASH	R	Chemin complet du shell courant
BASH_VERSION	R	Numéro de version de votre bash
BASH_VERSINFO	R	Informations sur la version actuel de bash
DISPLAY	W	Nom et le numéro de la console de la machine sur laquelle les clients X affichent leurs interfaces
EUID	R	ID utilisateur effectif numérique du processus shell courant
HISTCMD	R	Index numérique de la commande actuelle dans l'historique des commandes

HISTSIZE	W	Taille maximale de l'historique des commandes
HOME	R	Chemin du répertoire de l'utilisateur
HOSTNAME	R	Nom de la machine
HOSTTYPE	R	Architecture de la machine
IFS	W	Valeur du séparateur de champs interne. Cette valeur est utilisée pour découper les commandes en leurs composants
LANG	W	Indique la localisation courante (ou préférée) aux programmes qui supportent l'internationalisation Linux. En tant que telle, elle est souvent utilisée comme variable d'environnement
LD_LIBRARY_PATH	W	Spécifie les emplacements additionnels (séparés par :) dans lesquels doivent être cherchées les librairies partagées pour les exécutables liés dynamiquement
LD_PRELOAD	W	Spécifie une liste de librairies spécifiques (séparées par des espaces) à charger dans les autres, y compris celles spécifiées par le programme lui-même. Pour des raisons de sécurité, cette spécification peut être incompatible avec setuid et setgid
LINENO	R	Lorsqu'elle est utilisée dans un shell ou une fonction, contient l'offset dans les lignes depuis le démarrage de ce shell ou cette fonction
MACHTYPE	R	Identificateur GNU du type de machine
MAIL	W	Vous informe de l'arrivée d'un nouveau mail dans une boîte de style UNIX mbox. Si vous voulez que bash vous informe automatiquement, spécifiez l'emplacement de cette boîte
MAILCHECK	W	Intervalle (en secondes) qui indique la fréquence de contrôle d'arrivée de nouveau mail dans la boîte spécifiée
OLDPWD	R	Nom du répertoire de travail précédent
OSTYPE	R	Nom du système d'exploitation courant
PATH	W	Liste de répertoires séparés par : dans laquelle on cherche les binaires lors de l'exécution de programmes. C'est une variable d'environnement

PPID	R	ID du père du processus en cours
PS1	W	Forme de générer l'invite de bash
PWD	R	Répertoire de travail courant
RANDOM	R	Valeur aléatoire différente. Dépendant de l'implantation
REPLY	R	Valeur des données lues sur l'entrée standard lorsqu'on y accède après la commande read, à moins qu'une variable différente ne soit spécifiée à read
SECONDS	R	Nombre de secondes écoulées depuis le démarrage du processus shell courant
SHELLOPTS	R	Liste des options du shell courant
UID	R	ID utilisateur numérique réel du propriétaire du processus shell courant

B
Appels système

`fork()` : Création de nouveaux processus

Prototype :
`<sys/types.h>`
`<unistd.h>`
`pid_t fork()`
`pid_t` est un type entier défini dans `<sys/types.h>`
Valeurs retournées :
`-1` en cas d'erreur. Nombre de processus usager ou système excédé. La variable error contient le numéro d'erreur `EAGAIN`.
`>0` identifiant le processus fils.
`0` retourné au processus fils.

`execv()` : Charger un programme dans l'espace adresse d'un processus

Prototype :
`<unistd.h>`
`int execv(const char *path, char *const argv[]);`
`path` spécifie le programme à charger.
`argv` spécifie les arguments à passer au programme chargé.
`argv[0]` contient le nom du programme.
`argv[i]` contient les arguments.
Valeurs retournées :
Si `execv` retourne alors l'appel a échoué.
Appel de `perror("execv n'a pas pu exécuter le programme")` pour afficher un message d'erreur.

`execvp()` : Charger un programme dans l'espace adresse d'un

processus

Prototype :
```
<unistd.h>
int execvp(const char *file, char *const argv[]);
```
`file` spécifie le programme à charger.
Répertoire de recherche du programme sauvegardé dans la variable `PATH`
```
%PATH = /bin:/usr/bin:/usr/jmtorres %export PATH
```
`argv[0]` contient le nom du programme.
`argv[i]` contient les arguments.
Valeurs retournées :
Si `execvp` retourne alors l'appel a échoué.
Appel de `perror("execvp n'a pas pu exécuter le programme")` pour afficher un message d'erreur.

`execve()` : Charger un programme dans l'espace adresse d'un processus

Prototype :
```
<unistd.h>
int execve(const char *path, char *const argv[],
char *const envp[]);
```
`path` spécifie le programme à charger.
`argv[0]` contient le nom du programme.
`argv[i]` contient les arguments.
`envp` spécifie un environnement à substituer à l'environnement par défaut `USER`, `PATH`, `HOME`, `LOGNAME`, `SHELL`, `TERM`, etc.
Valeurs retournées :
Si `execve` retourne alors l'appel a échoué.
Appel de `perror("execve n'a pas pu exécuter le programme")` pour afficher un message d'erreur.

`execl()` : Charger un programme dans l'espace adresse d'un processus

Prototype :
```
<unistd.h>
int execl(const char *path, char *const arg*0, ...,
const char *argn, (char *)0);
```
`path` spécifie le programme à charger.
`argi (i >=0)` spécifie une liste d'arguments à passer au programme à charger. Liste d'arguments terminée par `NULL`.
`arg0` contient le nom du programme, `arg1,...,argn` contient les arguments.

Valeurs retournées :
Si `execl` retourne alors l'appel a échoué.
Appel de `perror("execl n'a pas pu exécuter le programme")` pour afficher un message d'erreur.

`execlp()` : Charger un programme dans l'espace adresse d'un processus

Prototype :
```
<unistd.h>
int execlp(const char *file, char *const arg*0, ...,
const char *argn, (char *)0);
```
`file` spécifie le programme à charger.
`argi (i >=0)` spécifie une liste d'arguments à passer au programme à charger. Liste d'arguments terminée par `NULL`.
`arg0` contient le nom du programme, `arg1,...,argn` contient les arguments.
Valeurs retournées :
Si `execlp` retourne alors l'appel a échoué.
Appel de `perror("execlp n'a pas pu exécuter le programme")` pour afficher un message d'erreur.

`execle()` : Charger un programme dans l'espace adresse d'un processus

Prototype :
```
<unistd.h>
int execle(const char *path, char *const arg*0, ...,
const char *argn, (char *)0, char *const envp[]););
```
`path` spécifie le programme à charger.
`argi (i >=0)` spécifie une liste d'arguments à passer au programme à charger. Liste d'arguments terminée par `NULL`.
`arg0` contient le nom du programme, `arg1,...,argn` contient les arguments.
Valeurs retournées :
Si `execle` retourne alors l'appel a échoué.
Appel de `perror("execle n'a pas pu exécuter le programme")` pour afficher un message d'erreur.

`exit()` : Terminaison et nettoyage d'un processus

Prototype :
```
<stdlib.h>
int atexit(void (*func) (void));
```
`atexit` enregistre la foction pointée par `func`.

`func` ne doit pas avoir d'arguments.
Fonctions exécutées en ordre inversé d'enregistrement.
Valeurs retournées :
Indication du succès de l'enregistrement de la fonction.

`atexit()` : Définition par l'utilisateur d'opérations de terminaison

Prototype :
```
<stdlib.h>
int atexit(void (*func) (void));
```
`atexit` enregistre la fonction pointée par `func`.
`func` ne doit pas avoir d'arguments.
Fonctions exécutées en ordre inversé d'enregistrement.
Valeurs retournées :
Indication du succès de l'enregistrement de la fonction.

`wait()` : Synchronisation d'un processus parent avec ses processus fils.

Prototype :
```
<sys/types.h>
<sys/wait.h>
pid_t wait(int *status);
```
`status` contient le code de statut de terminaison du processus fils.
Valeurs retournées :
Numéro de processus (`pid`) du premier processus fils terminé.
-1 si le processus n'a aucun processus fils.
La variable error contient le code d'erreur `ECHILD`.

`waitpid()` : Synchronisation d'un processus parent avec un processus fils spécifique.

Prototype :
```
<sys/types.h>
<sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```
`pid` spécifie le numéro du processus fils.
`status` contient le code de statut de terminaison du processus fils.
`options` est habituellement initialisé à `WNOHANG`.
Processus parent ne bloque pas durant l'exécution du processus fils, et `waitpid` retourne `0`.
Si `pid = -1` et `options = 0` alors `waitpid` a le même comportement que `wait`.
Valeurs retournées :

Numéro de processus fils ayant terminé.
-1 si le processus n'a aucun processus fils.
La variable error contient le code d'erreur `ECHILD`.

`getpid()` : Détermination du numéro d'un processus.

Prototype :
```
<sys/types.h>
<unistd.h>
pid_t getpid(void);
```
Valeurs retournées :
Numéro du processus appelant.

`getppid()` : Détermination du numéro du parent d'un processus.

Prototype :
```
<sys/types.h>
<unistd.h>
pid_t getppid(void);
```
Valeurs retournées :
Numéro du parent du processus appelant.

`pthread_create()` : Création d'un thread dans l'espace adresse d'un processus

Prototype :
```
<pthread.h>
int pthread_create(pthread_t *thread, const pthread_
attr_t *attr,
void *(*start_routine)(void*), void *arg);
```
`thread` pointe vers une adresse contenant le `ID` du nouveau `thread`.
`attr` spécifie les attributs du `thread`. Si `attr` est `NULL` alors les attributs par défaut sont utilisés.
Valeurs retournées :
`0` en cas de succès de création du `thread`.
Un code d'erreur en cas d'échec de création du `thread`. Nombre exédentaire de `threads`, attribut invalide, permissions insuffisantes, etc.

`pthread_exit()` : Terminaison d'un texttttthread.

Prototype :
```
<pthread.h>
void pthread_exit(void *value_ptr);
```
`value_ptr` est disponible à tout `thread` se synchronisant avec le

`thread` terminé.
Valeurs retournées :
Aucune.

`pthread_join()` : Synchronisation du `thread` appelant avec un autre `thread`.

Prototype :
```
<pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
```
`thread` est un pointeur vers l'`ID` du `thread` dont la terminaison est attendue.
`value_ptr` est passé à `pthread_exit()`
Valeurs retournées :
`0` en cas de succès de synchronisation avec un `thread`.
Un code d'erreur en cas d'échec de synchronisation avec un `thread`.
Pas de synchronisation possible avec le `thread` spécifié, ID inexistant du `thread` spécifié, etc.

`pthread_detach()` :
Indication au système que l'espace adresse du thread spécifié peut être déalloué à sa terminaison.

Prototype :
```
<pthread.h>
int pthread_detach(pthread_t thread);
```
`thread` est un pointeur vers l'`ID` du `thread`.
Valeurs retournées :
`0` en cas de succès.
Un code d'erreur en cas d'échec de synchronisation avec un `thread`.
Pas de synchronisation possible avec le `thread` spécifié, ID inexistant du `thread` spécifié, etc.

`pthread_attr_init()` : Initialisation des attributs par défaut d'un `thread`.

Prototype :
```
<pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
```
`attr` est initialisé avec tous les attributs par défaut spécifiés pour un système donné.
Valeurs retournées :
`0` en cas de succès d'initialisation de l'objet `attr`.
Un code d'erreur en cas d'échec d'initialisationde l'objet `attr`.

Insuffisance de mémoire.

`pthread_attr_destroy()` `destroy()` : Effacement des attributs par défaut d'un `thread`.

Prototype :

```
<pthread.h>
int pthread_attr_destroy(pthread_attr_t *attr);
```

`attr` est effacé.

Valeurs retournées :

`0` en cas de succès d'initialisation de l'objet `attr`.

Un code d'erreur en cas d'échec d'initialisationde l'objet `attr`.

`sleep()` : Dormir pendant quelque temps.

Pause de `NUMBER` secondes. `SUFFIX` peut-être en `s` secondes, `m` minutes, `h` heures ou `d` journées.

Prototype :

```
sleep [OPTION]...NUMBER[SUFFIX]
```

C
Signaux

Signaux Linux

Numéro	Nom	Action	Description
1	SIGHUP	exit	Terminaison de la liason avec le terminal
2	SIGINT	exit	Interruption en provenance du terminal (<ctrl C>)
3	SIGQUIT	core	Quitter en provenance du terminal (<ctrl \>)
4	SIGILL	core	Instruction illégale
5	SIGTRAP	core	Trace ou trap de point de débogage
6	SIGIOT		
7	SIGBUS	core	Erreur de bus (adresse mal alignée)
8	SIGFPE	core	Exception d'arithmétique point flottant
9	SIGKILL	exit	Élimination (pas d'attrapage et masquage)
10	SIGUSR1		Signal1 défini par l'usager
11	SIGSEGV	core	Faute de segmentation (référence mémoire)
12	SIGUSR2	exit	Signal défini par l'usager
13	SIGPIPE	exit	Tube sans processus récepteur
14	SIGALRM	exit	Déclenchement de l'alarme (setitimer, alarm)
15	SIGTERM	exit	Terminaison envoyée par la commande kill
17	SIGCHLD	ignore	Chargement d'état du processus enfant
18	SIGCONT		Reprise
19	SIGSTOP	stop	Arrêt (pas d'attrapage et masquage)
20	SIGTSTP	stop	Arrêt (<ctrl Z>)
21	SIGTTIN	stop	Lecture au terminal par processus en arrière plan (arrêt)
22	SIGTTOU	stop	Ecriture au terminal par un processus en

			arrière plan (arrêt)
23	SIGURG	ignore	Données réseau urgentes
24	SIGXCPU	core	Expiration du temps UCT (getrlimit)
25	SIGXFSZ	core	Dépassement de taille de fichier (getrlimit)
26	SIGVTALRM	exit	Déclenchement d'un temporisateur virtuel (setitimer)
27	SIGPROF	exit	Déclenchement d'un temporisateur profilé (setitimer)
28	SIGWINCH	ignore	Changement de taille de la fenêtre
29	SIGIO		
30	SIGPWR	ignore	Panne d'alimentation ou réinitialisation

Signaux Unix

Numéro	Nom	Action	Description
1	SIGHUP	exit	Terminaison de la liason avec le terminal
2	SIGINT	exit	Interruption en provenance du terminal (<ctrl C>)
3	SIGQUIT	core	Quitter en provenance du terminal (<ctrl \>)
4	SIGILL	core	Instruction illégale
5	SIGTRAP	core	Trace ou trap de point de débogage
6	SIGABRT	core	Annulation par un appel à abort (...)
7	SIGEMT	core	Trap d'émulation
8	SIGFPE	core	Exception d'arithmétique point flottant
9	SIGKILL	exit	Élimination (pas d'attrapage et masquage)
10	SIGBUS	core	Erreur de bus (adresse mal alignée)
11	SIGSEGV	core	Faute de segmentation (référence mémoire)
12	SIGSYS	core	Appel système erroné
13	SIGPIPE	exit	Tube sans processus récepteur
14	SIGALRM	exit	Déclenchement de l'alarme (setitimer, alarm)
15	SIGTERM	exit	Terminaison envoyée par la commande kill
16	SIGUSR1	exit	Signal défini par l'usager
17	SIGUSR2	exit	Signal défini par l'usager
18	SIGCHLD	ignore	Chargement d'état du processus enfant
19	SIGPWR	ignore	Panne d'alimentation ou réinitialisation
20	SIGWINCH	ignore	Changement de taille de la fenêtre
21	SIGURG	ignore	Données réseau urgentes

22	SIGPOLL	exit	Evénement sur un périphérique en polling
23	SIGSTOP	stop	Arrêt (pas d'attrapage et masquage)
24	SIGTSTP	stop	Arrêt (<ctrl Z>)
25	SIGCONT	ignore	Continuation (<ctrl Q>
26	SIGTTIN	stop	Lecture au terminal par processus en arrière plan (arrêt)
27	SIGTTOU	stop	Ecriture au terminal par un processus en arrière plan (arrêt)
28	SIGVTALRM	exit	Déclenchement d'un temporisateur virtuel (setitimer)
29	SIGPROF	exit	Déclenchement d'un temporisateur profilé (setitimer)
30	SIGXCPU	core	Expiration du temps UCT (getrlimit)
31	SIGXFSZ	core	Dépassement de taille de fichier (getrlimit)

D
Sites Internet

- Site du cours à l'École Polytechnique de Montréal
  - http://www.cours.polymtl.ca/inf3600/
- Site Internet de GNU
  - http://www.gnu.org
- Sites pour les développeurs Linux
  - http://developer.kde.org
  - http://www.linuxprogramming.com
  - http://www.blackdown.org
  - http://developer.gnome.org
  - http://www.perl.com
  - http://java.sun.com
- Site du serveur Web Apache pour Linux/Unix
  - http://www.apache.org
- Sites des distributions et des noyaux Linux
  - http://www.redhat.com
  - http://www.calderasystems.com
  - http://www.suse.com
  - http://www.infomagic.com
  - http://www.linuxppc.com
  - http://www.slackware.com
  - http://www.linux-mandrake.com
  - http://www.java.sun.com
  - http://www.kernel.org

    - http://www.debian.org
    - http://www.linux.org
- Sites des bureaux et des gestionnaires de fenêtres Linux
    - http://www.gnome.org
    - http://www.kde.org
    - http://www.x11.org
    - http://www.fvwm.org
    - http://www.windowmaker.org
    - http://www.enlightment.org
    - http://www.xfree86.org
    - http://www.themes.org
- Site sur les appels système Unix
    - http://www.opennc.org/onlinepubs/007908799/
- Sites sur des systèmes d'exploitation divers
    - http://www.apple.com/macosx
    - http://www.microsoft.com
    - http://www.qnx.org
    - http://www.freebsd.org
    - http://www.novell.com
    - http://www.palmos.com
    - http://www.sun.com
    - http://www.samba.com
    - http://www.beos.com
    - http://www.ibm.com
    - http://www.free-vms.org
- Sites sur le simulateur Nachos
    - http://www.cs.duke.edu/narten/110/nachos/main/main.html
    - http://www.cs.washington.edu/homes/tom/nachos/

Bibliographie

[BC01] Daniel. P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.

[BG02] Martine Belaïche and Richard Gourdeau. *INF2600 Programmation de systèmes sur microprocesseur*. École Polytechnique de Montréal, 2002.

[Bla02] Christoffe Blaes. *Programmation système en C sous Linux*. Editions Eyrolles, Paris, 2002.

[CGdMP01] J. Carretero, F. García, P. de Miguel, and F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.

[Har02] J. Archer Harris. *Operating Systems*. Schaum's outlines, McGraw-Hill, 2002.

[JLH00] David A. Patterson John L. Hennessy. *Architecture des ordinateurs. Une approche quantitative*. Thomson Publishing et Morgan kauffman Publishing, France, 2000.

[SGG01] A. Silverschatz, P. Gavin, and G. Gagne. *Principles appliqués des systèmes d'exploitation avec Java*. Vuibert, Paris, France, 2001.

[Tan89] Andrew Tanenbaum. *Les systèmes d'exploitation*. InterEditions, Paris, France, 1989.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, New Jersey, 2001.

Index

Symboles

/dev 385, 387
/proc 358
; 52
> 51
$* 55
$0 55
$# 55
& 53

A

accès concurrents 166
adresse
  linéaire 317
  logique 317
AF_INET 416, 422
AF_LOCAL 416
AF_UNIX 416
algorithme
  Buddy 319
  C-SCAN 393
  du banquier 261
  SCAN 393
  SSTF 393
allocation
  contiguë 355
  globale 308
  locale 308
anomalie de Belady 312
antémémoire 378, 404
appel système
  chdir() 347
  close() 326
  closedir() 347
  rename 347
appel système 26, 32, 33, 36
  atexit() 444
  chdir() 34, 348
  chmod() 34
  close() 34, 326, 329
  closedir() 348
  creat() 326, 327
  dup() 335
  dup2() 335
  exec 83
  exec() 75
  execl() 442, 443
  execlp() 443
  execv() 441
  execve() 34, 442
  execvp() 441
  exit() 34, 443
  fcntl() 370
  flock() 375
  fork() 34, 76, 77, 214, 441
  getpid() 65, 445
  getppid() 445
  kill() 34, 138
  link() 34, 327, 362
  lockf() 370
  lseek() 34, 327, 332

mkdir() 34, 347
mknod() 154
mmap() 292
mount() 34
munmap() 292
open() 34, 326, 327
opendir() 347
pause() 141
pipe() 148, 149
pthread_attr_ 447
pthread_attr_init() 446
pthread_create() 445
pthread_detach() 446
pthread_exit() 445
pthread_join() 446
read() 34, 327
readdir() 347, 348
readlink() 364
rmdir() 34, 347
sendfile() 344
sleep() 141, 447
socket() 416
stat() 34, 327, 339
sync() 379
time() 34
umount() 34
unlink() 34, 327, 343, 363
vmstat() 319
wait() 34, 444
waitpid() 34, 444
write() 34, 327, 329

B

bande passante 392
BCP 252
Belady 308
bitmap 286
bits de protection 62
Bloc de Contrôle de Processus 72
boot 66
BSD 43
  verrouillage 375
Buddy system 289

C

case 53
cat 49
cd 49
chargeur 314
chmod 49
client 37, 403, 408, 410
client / serveur 37, 405, 408
communication interprocessus 135
compteur d'événements 194
conditions de Coffman 254
core 137
cp 49
cylindre 391

D

défaut de page 299
date 49
deadlock 252
DMA 391
DNS 422

E

échange de messages 147, 197
ecroulement du système 312
environ 358
espace d'adressage 299
exclusion mutuelle 166, 168, 174, 175, 178

F

FAT 356
FCFS 392
fichiers 34, 61, 62
  attributs 323
  chemin d'accès 34, 61, 325, 362

absolu 35
relatif 35
copier 344
descripteur 62
extension 323
nom 323
ordinaires 324
projection 291
propriétaire 62
spéciaux 325, 385
caractère 324
bloc 324
verrouillage 369
file d'entrée
multiples 280
unique 280
file de messages 219
find 49
for 53
format **ELF** 316
fragmentation 280
externe 280
interne 280
FreeBSD 44
ftok() 220
ftp 422, 425

G

gethostbyname() 418, 422
GNU 48
GNU/Linux 48
graphe d'allocation des ressources 255
réduction 256
grep 49, 53
groupe d'utilisateur 61

H

hôte 415
head 49, 53
hiérarchie de la mémoire 314
HTTP 422
HURD 37

I

i-nœud 326, 339, 355, 357, 362, 366, 380
if 53
inet_ntoa() 418, 422
Inter Process Communication 219
interblocages 252
algorithme de détection 259
prévention 262
Internet 422
interpréteur de commandes 32
interruption logicielle 46, 136
ioctl() 389
IPC 219
ipcrm 226
ipcs 226
ISO 406

K

kernel 46
kill 49

L

lien
physique 362
symbolique 364
Linux 27, 36, 44, 45
ln 49, 362, 364
ls 49

M

mémoire
associative 306
cache 314
compactage 284
partagée 236
physique 301

principale 269
segment 313
virtuelle 291, 299
machine virtuelle 26
man 48
masquage des interruptions 168
MFT 277
Minix 27, 37
mkd 49
mknod 154
MMU 302
moniteur 196
moniteur résident 273
more 53
msgctl() 223
msgget() 221
Multics 43
multiflot 97
multiprogrammation 29, 30
lots 29
partage de temps 30
multithreading 97
mutex 199
mv 49
MVT 283

N

NFS 403
nice 128
notation pointée 422
noyau 35, 46
numéro majeur 385
numéro mineur 385

O

objet critique 166
ordonnancement
circulaire 120
non préemptif 116
plus petit temps de séjour 120
round robin 120
tourniquet 120
ordonnanceur 113, 116
à deux niveux 126
à priorité 124
de threads 126
Linux 129
moyen terme 114
non préemtif 119
préemptif 119
overlays 291

P

périphérique 385
pagination 301
partage de fichiers 361
passwd 49
PCB 72, 273
Peterson 171
PF_LOCAL 419
PF_UNIX 419
philosophes 184
phtread_mutex_lock 199
pipeline 52
pipes 147, 418
named 154
piste 391
POSIX 240
Posix 33, 43, 75, 97, 138, 179, 291, 326, 347
PE commandes 49
prises 158, 410
processus 33, 63, 71, 72, 74
démon 66, 74
exécution
asynchrone 75
séquentielle 75
fils 74–77, 83
légers 34, 96
père 74, 76

synchronisation 165, 194
zombie 88
producteur/consommateur 176, 194, 196, 197
projection de fichier 291, 292
protocole 406
connecté 406
modèle ISO 406
non connecté 406
ps 49, 65
pstree 65
pthread_create() 98
pthread_exit() 99
pthread_join() 98
pthread_mutex_t 199
pwd 49

Q

quantum 120, 121
choix 121
variable 125

R

répertoires 34, 63, 324, 346
read 55
redirection 50, 387
registre
base et limite 278
limite 274, 278
remplacement de page 308
de l'horloge 310
FIFO 309
la moins récemment utilisée 311
non récemment utilisée 309
optimal 308
rm 49
rmdir 49
root 61
RPC 409

S

sémaphore 178
Posix 179
System V 240
SCHED_FIFO 129
sched_getscheduler() 130
SCHED_OTHER 129
SCHED_RR 129
scheduler 113
section critique 166, 168
segmentation 313
paginée 313
segments de recouvrement 291
serveur 38, 408, 410
concurrent 410
HTTP 423
interactif 410
setsockopt() 418
shell 32, 47, 52, 53, 64
arithmétique 59
Bourne 48
Korn 48
script 53
signaux 136
SLEEP 175
sleep 49
socket 147, 158, 410
accept() 417
bind() 417
close() 417
connect() 417
connectée 413
Internet 422
listen() 417
locale 419
non connectée 413
socket() 416
sort 50, 53
spool 29, 165
swapping 277, 283

synchronisation 197, 204
système
  micro-kernel 37
  monolithique 36
système d'exploitation 26
  distribué 32, 399
  mono-utilisateur 31
  multitâche 113
  préemptif 113
système de fichiers 323, 346
  /proc 358
  structure arborescente 346
systèmes multiprocesseurs 32
System V 43, 204, 243
  sémaphores 240
system() 75, 90

T

table
  d'allocation de fichiers 356
  de bits 286
  de pages 302
  index 357
tail 53
talons 409
threads 95
  combinés 97
  gestion Posix 97
  noyau 97
  Solaris 106
  synchronisation 199
  utilisateur 97
trames 406
Translation Lookside Buffer 306
tsl 173
tubes de communication 147
  nommés 154
  sans nom 147

U

umask 62
Unix 33, 43, 45, 61
  BSD 44
  shell 48
uptime 358

V

verrou
  exclusif 369, 376
  partagé 369, 376
verrouillage 169
voleur de pages 315

W

WAKEUP 175
wc 49, 148
while 53
who 49, 148

Z

zombie 88