Cours Système

D.Revuz

17 février 2005

Table des matières

1 Introduction **1**

1.1 Unix . . . 1

1.1.1 Pourquoi unix ? . . . 1

1.1.2 le succès d'Unix et de linux . . . 1

1.1.3 Des points forts . . . 1

1.1.4 Des points faibles . . . 2

1.2 Structure générale des systèmes d'exploitation . . . 2

1.2.1 Les couches fonctionnelles . . . 3

1.2.2 L'architecture du système . . . 3

1.2.3 L'architecture du noyau . . . 5

1.3 historique . . . 5

2 Système de Gestion de Fichiers **7**

2.1 Le concept de fichier . . . 7

2.2 Fichiers ordinaires / Fichiers spéciaux. . . . 8

2.3 Organisation utilisateur des Disques . . . 8

2.4 Les inodes . . . 9

2.5 Organisation des disques System V . . . 10

2.6 Adressage des blocs dans les inodes . . . 11

2.7 Allocation des inodes d'un disque . . . 11

2.8 Allocation des blocs-disque . . . 11

3 Le Buffer Cache **17**

3.1 Introduction au buffer cache . . . 17

3.1.1 Avantages et désavantages du buffer cache . . . 17

3.2 Le buffer cache, structures de données. . . . 17

3.2.1 La liste doublement chaînée des blocs libres . . . 18

3.3 L'algorithme de la primitive `getblk` . . . 18

4 La bibliothèque standard **23**

4.1 Les descripteurs de fichiers. . . . 23

4.1.1 Ouverture d'un fichier . . . 24

4.1.2 Redirection d'un descripteur : `freopen` . . . 24

4.1.3 Création de fichiers temporaires . . . 25

4.1.4 Ecriture non formatée . . . 25

4.1.5 Accès séquentiel . . . 26

4.1.6 Manipulation du pointeur de fichier . . . 26

4.1.7 Un exemple d'accès direct sur un fichier d'entiers. . . . 26

4.1.8 Les autres fonctions de déplacement du pointeur de fichier. . . . 26

4.2 Les tampons de fichiers de stdlib. . . . 27

4.2.1 Les modes de bufferisation par défaut. . . . 27

4.2.2 Manipulation des tampons de la bibliothèque standard. . . . 27

4.3 Manipulation des liens d'un fichier . . . 29
4.4 Lancement d'une commande shell . . . 29
4.5 Terminaison d'un processus . . . 30
4.6 Gestion des erreurs . . . 31
4.7 Création et destruction de répertoires . . . 31

5 Appels système du Système de Gestion de Fichier 33
5.1 `open` . . . 33
5.1.1 Déroulement interne d'un appel de `open` . . . 35
5.2 `creat` . . . 36
5.3 `read` . . . 36
5.4 `write` . . . 36
5.5 `lseek` . . . 37
5.6 `dup` et `dup2` . . . 37
5.7 `close` . . . 38
5.8 `fcntl` . . . 38

6 Les processus 41
6.1 Introduction aux processus . . . 41
6.1.1 Création d'un processus - `fork()` . . . 41
6.2 Format d'un fichier exécutable . . . 42
6.3 Chargement/changement d'un exécutable . . . 42
6.4 zone u et table des processus . . . 43
6.5 `fork` et `exec` (revisités) . . . 43
6.6 Le contexte d'un processus . . . 45
6.7 Commutation de mot d'état et interruptions. . . . 45
6.8 Les interruptions . . . 46
6.9 Le problème des cascades d'interruptions . . . 47
6.9.1 Etats et transitions d'un processus . . . 47
6.9.2 Listes des états d'un processus . . . 47
6.10 Lecture du diagramme d'état. . . . 48
6.11 Un exemple d'exécution . . . 49
6.12 La table des processus . . . 49
6.13 La zone u . . . 50
6.14 Accès aux structures `proc` et `user` du processus courant . . . 50
6.14.1 Les informations temporelles. . . . 50
6.14.2 Changement du répertoire racine pour un processus. . . . 51
6.14.3 Récupération du PID d'un processus . . . 51
6.14.4 Positionement de l'euid, ruid et suid . . . 51
6.15 Tailles limites d'un processus . . . 52
6.15.1 Manipulation de la taille d'un processus. . . . 52
6.15.2 Manipulation de la valeur nice . . . 52
6.15.3 Manipulation de la valeur umask . . . 52
6.16 L'appel système `fork` . . . 53
6.17 L'appel système `exec` . . . 53

7 L'ordonnancement des processus 55
7.1 Le partage de l'unité centrale . . . 55
7.1.1 Famine . . . 56
7.1.2 Stratégie globale . . . 56
7.1.3 Critères de performance . . . 57
7.2 Ordonnancement sans préemption. . . . 57
7.3 Les algorithmes préemptifs . . . 57
7.3.1 Round Robin (tourniquet) . . . 58

7.3.2 Les algorithmes à queues multiples . . . 58
7.4 Multi-level-feedback round robin Queues . . . 58
7.4.1 Les niveaux de priorité . . . 58
7.4.2 Evolution de la priorité . . . 59
7.4.3 Les classes de priorité . . . 60

8 La mémoire 61
8.0.4 les mémoires . . . 61
8.0.5 La mémoire centrale . . . 61
8.1 Allocation contiguë . . . 63
8.1.1 Pas de gestion de la mémoire . . . 63
8.1.2 Le moniteur résidant . . . 63
8.1.3 Le registre barrière . . . 63
8.1.4 Le registre base . . . 63
8.1.5 Le swap . . . 65
8.1.6 Le coût du swap . . . 65
8.1.7 Utilisation de la taille des processus . . . 65
8.1.8 Swap et exécutions concurrentes . . . 66
8.1.9 Contraintes . . . 66
8.1.10 Deux solutions existent . . . 66
8.1.11 Les problèmes de protection . . . 66
8.1.12 Les registres doubles . . . 66
8.2 Ordonnancement en mémoire des processus . . . 67
8.3 Allocation non-contiguë . . . 68
8.3.1 Les pages et la pagination . . . 68
8.3.2 Ordonnancement des processus dans une mémoire paginée . . . 68
8.3.3 Comment protéger la mémoire paginée . . . 69
8.3.4 La mémoire segmentée . . . 69

9 La mémoire virtuelle 71
9.0.5 Les overlays . . . 71
9.0.6 Le chargement dynamique . . . 72
9.1 Demand Paging . . . 72
9.1.1 Efficacité . . . 73
9.2 Les algorithmes de remplacement de page . . . 75
9.2.1 Le remplacement optimal . . . 75
9.2.2 Le remplacement peps (FIFO) . . . 75
9.2.3 Moins récemment utilisée LRU. . . . 76
9.2.4 L'algorithme de la deuxième chance . . . 76
9.2.5 Plus fréquemment utilisé MFU . . . 76
9.2.6 Le bit de saleté (Dirty Bit) . . . 77
9.3 Allocation de pages aux processus . . . 77
9.4 L'appel fork et la mémoire virtuelle . . . 78
9.5 Projection de fichiers en mémoire . . . 78
9.6 Les conseils et politiques de chargement des zones mmappées . . . 80
9.7 Chargement dynamique . . . 81

10 Tubes et Tubes Nommés 83
10.1 Les tubes ordinaires (*pipe*) . . . 83
10.2 Création de tubes ordinaires . . . 83
10.3 Lecture dans un tube . . . 85
10.4 Ecriture dans un tube . . . 86
10.5 Interblocage avec des tubes . . . 86
10.6 Les tubes nommés . . . 86

10.6.1 Ouverture et synchronisation des ouvertures de tubes nommés 87
10.6.2 Suppression d'un tube nommé 87
10.6.3 les appels popen et pclose 87

11 Les signaux 89
11.0.4 Provenance des signaux 89
11.0.5 Gestion interne des signaux 89
11.0.6 L'envoi de signaux : la primitive kill 90
11.1 La gestion simplifiée avec la fonction `signal` 91
11.1.1 Un exemple 91
11.2 Problèmes de la gestion de signaux ATT 91
11.2.1 Le signal SIGCHLD 93
11.3 Manipulation de la pile d'exécution 94
11.4 Quelques exemples d'utilisation 95
11.4.1 L'appel pause 95
11.5 La norme POSIX 96
11.5.1 Le blocage des signaux 96
11.5.2 sigaction 97
11.5.3 L'attente d'un signal 97

12 Les verrous de fichiers 99
12.1 Caractéristiques d'un verrou 99
12.2 Le mode opératoire des verrous 99
12.3 Manipulation des verrous 100
12.4 Utilisation de `fcntl` pour manipuler les verrous 101

13 Algorithmes Distribués & Interblocages 103
13.1 exemples 103
13.1.1 Les méfaits des accès concurrents 103
13.1.2 Exclusion mutuelle 104
13.2 Mode d'utilisation des ressources par un processus. 105
13.3 Définition de l'interblocage (deadlock) 105
13.4 Quatre conditions nécessaires à l'interblocage. 105
13.5 Les graphes d'allocation de ressources 105

14 Sécurité et Sûreté de fonctionnement 107
14.1 Protection des systèmes d'exploitation 107
14.2 Généralités sur le contrôle d'accès 108
14.2.1 Domaines de protection et matrices d'accès 109
14.2.2 Domaines de protection restreints 109
14.2.3 Avantages des domaines de protections restreints 110
14.3 Le cheval de Troie 110
14.4 Le confinement 110
14.5 les mécanismes de contrôle 110
14.5.1 Application des capacités au domaines de protection restreints 112
14.6 Les ACL 115
14.6.1 Appels systemes `setacl` et `getacl` 115
14.6.2 Autres pistes sur la sécurité 116

15 Multiplexer des entrées-sorties 119
15.1 Gerer plusieurs cannaux d'entrée sortie 119
15.1.1 Solution avec le mode non bloquant 119
15.1.2 Utiliser les mécanismes asynchrones 119
15.2 Les outils de sélection 119

15.2.1 La primitive `select` . . . 119
15.2.2 La primitive `poll` . . . 121
15.2.3 Le périphérique poll . . . 122
15.2.4 Les extensions de `read` et `write` . . . 123
15.3 une solution multi-activités . . . 123

16 Les threads POSIX **125**
16.0.1 Description . . . 125
16.0.2 `fork` et `exec` . . . 125
16.0.3 clone . . . 127
16.0.4 Les noms de fonctions . . . 127
16.0.5 les noms de types . . . 127
16.0.6 Attributs d'une activité . . . 128
16.0.7 Création et terminaison des activités . . . 128
16.1 Synchronisation . . . 128
16.1.1 Le modèle fork/join (Paterson) . . . 129
16.1.2 Le problème de l'exclusion mutuelle sur les variables gérées par le noyau . . 129
16.1.3 Les sémaphores d'exclusion mutuelle . . . 129
16.1.4 Utilisation des sémaphores . . . 130
16.1.5 Les conditions (évènements) . . . 130
16.2 Ordonnancement des activités . . . 132
16.2.1 L'ordonnancement POSIX des activités . . . 132
16.3 Les variables spécifiques à une thread . . . 133
16.3.1 Principe général des données spécifiques, POSIX . . . 134
16.3.2 Création de clés . . . 134
16.3.3 Lecture/écriture d'une variable spécifique . . . 134
16.4 Les fonctions standardes utilisant des zones statiques . . . 134

17 Clustering **135**
17.1 Le clustering sous linux . . . 135

18 Bibliographie **137**
18.1 Webographie . . . 137

Cours de conception de systèmes et d'utilisation d'UNIX
Ce poly est à l'usage des étudiants de la filière Informatique et Réseaux de l'école d'ingénieurs Ingénieurs 2000 UMLV et de la troisème année de lience d'informatique de Marne la Vallée comme support du cours SYSTÈMES d'EXPLOITATION.

Cette version du , apporte de nombreuse corrections de typo et autre, je remercie David Lecorfec pour sa lecture attentive, et les remarques sur le fond seront prises en compte dans la prochaine version.

Ce poly a une version HTML disponible sur le Web a l'adresse suivante :

`http://www-igm.univ-mlv.fr/~dr/NCS/`

Ce document a de nombreux défauts en particulier son manque d'homogénéité, et une absence d'explications dans certaines parties (explication données en général oralement en cours).

Au menu l'essentiel d'UNIX : SGF, processus, signaux, mémoire, mémoire virtuelle, manipulation des terminaux, tubes, IPC. Quelques détours : micro-noyaux, sécurité. Un chapitre important mais un peut court : les problèmes de programmation distribué et des interblocages (améliorations en cours fin 2004).

Prérequis : pour la partie conceptuelle des notions de programmation et d'algorithmique sont nécessaire pour profiter pleinement du cours, pour la partie technique une compétance raisonable en C est nécessaire, en particulier les notions de pointeurs d'allocation dynamique doivent être maitrisées, les 4 méthodes d'allocation principale du C doivent être maitrisées ! (text,static,auto,heap).

Évolutions futures : dr@univ-mlv.fr (j'attend vos remarques), uniformisation de la présentation, nettoyage des points obscurs, corrections orthographiques, complement sur fcntl, ioctl, plus d'exemples, des sujets de projets , des sujets d'examen.

Chapitre 1

Introduction

Ceci est un polycopié de cours de licence informatique sur les systèmes d'exploitations en général et plus spécialement sur la famille Unix.
Ce poly est le support utilisé pour les licences et pour les Apprentis ingénieurs de la filière Informatique et Réseau.

1.1 Unix

1.1.1 Pourquoi unix ?

Pourquoi ce choix d'unix comme sujet d'étude pour le cours ?

- LE PRIX
- la disponibilité des sources
- L'intélligence des solutions mise en oeuvre
- de grande ressource bibliographique
- il faut mieux apprendre les conceptes fondamentaux dans un système simple et ouvert puis passer a des systèmes propriétaires et fermés que l'inverse.
- parceque je ne vais pas changer mon cours tout de suite

1.1.2 le succès d'Unix et de linux

Le succès d'UNIX sans doute parce que :
- Ecrit dans un langage de haut niveau : C (C++, Objective C) ;
- une interface simple et puissante : les shells, qui fournissent des services de haut niveau ;
- Des primitives puissantes qui permettent de simplifier l'écriture des programmes ;
- Un système de fichier hiérarchique qui permet une maintenance simple et une implémentation efficace ;
- Un format générique pour les fichiers, le flot d'octets qui simplifie l'écriture des programmes ;
- Il fournit une interface simple aux périphériques ;
- Il est multi-utilisateurs et multi-tâches ;
- Il cache complètement l'architecture de la machine à l'utilisateur.

1.1.3 Des points forts

- Système né dans le monde de la recherche
  intégration de concepts avancés
- Diffusion ouverte
  accès aux sources

- Langage (de haut niveau)
  compilation séparée, conditionnelle, paramétrage, précompilation
- Enrichissement constant
- Ouverture (paramétrabilité du poste de travail)
- Souplesse des entrées/sorties
  uniformité
- Facilités de communication inter-systèmes
- Communautés d'utilisateurs (/etc/groups)
- Langages de commandes (flexibles et puissants)
- Aspect multi-utilisateurs
  connections de tout type de terminal, bibliothèques, etc
- Parallélisme
  multi-tâches : "scheduling" par tâche
  communication entre tâches
  multiprocesseurs
- Interface système/applications
  appels système, bibliothèque
- le système de gestion de fichiers
  hiérarchie
- Interfaces graphiques normées : X11.
- Profusion d'interfaces graphiques sous linux Gnome et KDE en particulier

1.1.4 Des points faibles

- Fragilité du S.G.F.
  pertes de fichiers possible en cas de crash
  réglé avec les SGF journalisés
- Gestion et rattrapage des interruptions inadapté au temps réel
  Des évolutions avec RLlinux et OS9.
- Mécanisme de création de processus lourd
  de nombreuses améliorations en particulier les threads.
- Une édition de liens statique
  Amélioration avec les librairies partagées.
  des Modules noyau chargeables/déchargeables dynamiquement
- Rattrapage d'erreur du compilateur C standard peu aisé !
  Ces bugs sont corrigées
- Coût en ressources
- reste globalement efficasse
- Gestion

1.2 Structure générale des systèmes d'exploitation

Un système d'exploitation est un programme qui sert d'interface entre un utilisateur et un ordinateur.

Un système d'exploitation est un ensemble de procédures manuelles et automatiques qui permet

à un groupe d'utilisateurs de partager efficacement un ordinateur. *Brinch Hansen.*

Il est plus facile de définir un système d'exploitation par ce qu'il fait que par ce qu'il est. *J.L. Peterson.*

Un système d'exploitation est un ensemble de procédures cohérentes qui a pour but de gérer la pénurie de ressources. *J-l. Stehlé P. Hochard.*

Quelques systèmes :

le batch Le traitement par lot (disparus).

interactifs Pour les utilisateurs (ce cher UNIX).

temps réels Pour manipuler des situations physiques par des périphériques (OS9 un petit frère futé d'UNIX). L'idée est de gérer le temps vrai.En particulier de gérer des évènement aléatoires qui neccessite l'exécution d'une action en proirité absolue.

distribués UNIX ?, les micros noyaux ? l'avenir ?

moniteurs transactionnels Ce sont des applications qui manipulent des objets à tâches multiples comme les comptes dans une banque, des réservations, etc. L'idée est de décomposer l'activité en actions, chacune indépendantes des autres,pour ce faire elle sont écrites pour avoir un comportement dit "atomique" ainsi il n'y a pas de programme mais des évènement et des actions associées. Il n'y pas dans de changement de context pour traiter une action, c'est le système adapté pour traité de gros volumes de petites opérations.

SE orientés objets Micro Noyaux.

1.2.1 Les couches fonctionnelles

Couches fonctionnelles :

- Programmes utilisateurs
- Programmes d'application éditeurs/tableurs/BD/CAO
- Programmes système assembleurs/compilateurs/éditeurs de liens/chargeurs
- système d'exploitation
- langage machine
- microprogramme
- machines physiques

1.2.2 L'architecture du système

L'architecture globale d'UNIX est une architecture par couches (coquilles) successsives comme le montre la figure 1.2. Les utilisateurs ordinaire communiquent avec la couche la plus évoluée celle des applications (en générale aujourd'hui associé avec une interface graphique). Le programmeur lui va en fonction de ses besoins utiliser des couches de plus en plus profondes, plus précises mais plus difficiles a utiliser.
Chaque couche est construite pour pouvoir être utilisée sans connaitre les couches inférieures (ni leur fonctionnement, ni leur interface).
Cette hiérarchie d'encapsulation permet d'écrire des applications plus portables. En effet si elles sont écrites dans les couches hautes, le travaille de portage est fait par le portage des couches inférieures. Pour des applications où le temps de calcul prime devant la portabilité, les couches basses seront utilisées.

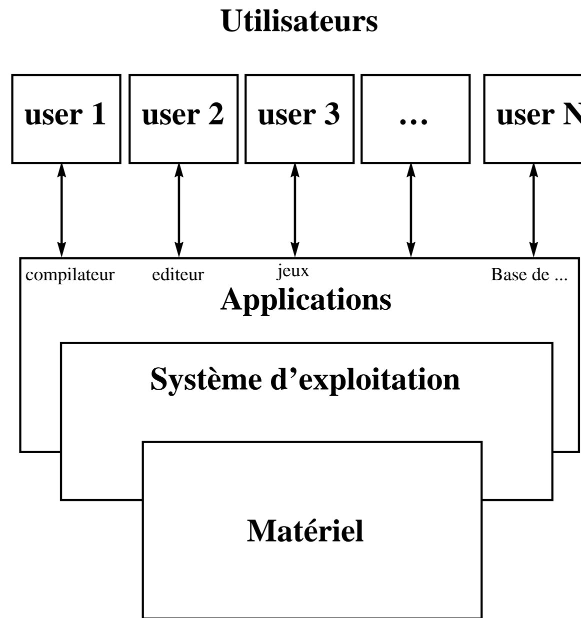

Fig. 1.1 – Vue générale du système

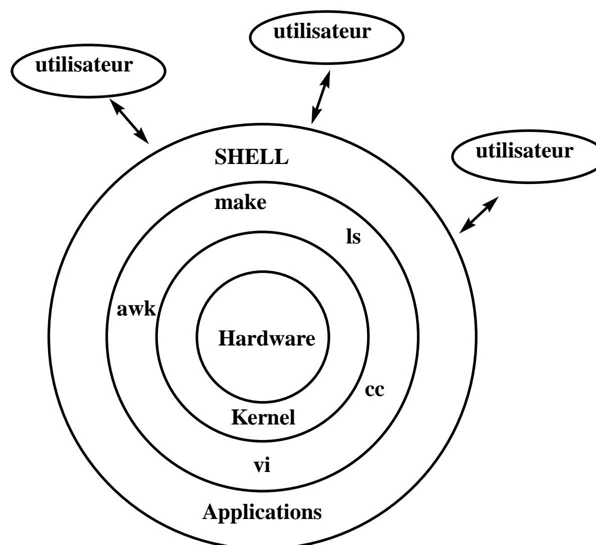

Fig. 1.2 – Point de vue utilisateur

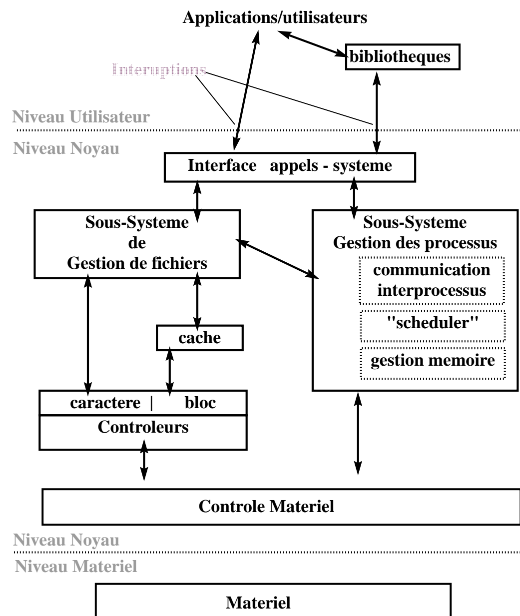

Fig. 1.3 – Architecture du noyau

1.2.3 L'architecture du noyau

L'autre approche architecturale est l'architecture interne du Noyau (kernel). C'est une architecture logicielle elle permet aux développeur de structurer le travail de développement. Le but ici est de simplifier la compréhension et la fabrication du système. Nous cherchons donc ici à décomposer le noyau en parties disjointes (qui sont concevables et programmables de façons disjointes). La Figure 1.3 donne une idée de ce que peut être l'architecture interne d'un noyau UNIX. Noter bien la position extérieure des bibliothèques .

1.3 historique

Il existe un site très agréable sur l'histoire des ordinateurs :

`http://www.computerhistory.org/`

.

Chapitre 2

Système de Gestion de Fichiers

Le système de gestion de fichiers est un outil de manipulation des fichiers et de la structure d'arborescence des fichiers sur disque et a aussi le rôle sous UNIX de conserver toutes les informations dont la pérennité est importante pour le système (et pour les utilisateurs biensur). Ainsi tous les objets importants du système sont référencés dans le système de fichiers (mémoire, terminaux, périphériques variés, etc).

Le système de gestion de fichier permet une manipulation simple des fichiers et gère de façon transparente les différents problèmes d'accès aux supports de masse :
- partage : utilisation d'un même fichier/disque par plusieurs utilisateurs
- efficacité : utilisation de cache, uniformisation des accès
- droits : protection des éléments important du système et protection interutilisateurs
- alignement : transtypage entre la mémoire et les supports magnétiques

2.1 Le concept de fichier

L'unité logique de base de l'interface du Système de Gestion de Fichiers : `le fichier`.

Un fichier Unix est une suite finie de bytes (octets) Matérialisée par des blocs disques, et une inode qui contient les propriétés du fichier (mais pas son nom).
Le contenu est entièrement défini par le créateur, la gestion de l'allocation des ressources nécessaires est a la seule responsabilité du système.

Sur Unix les fichiers ne sont pas typés du point de vue utilisateur, le concept de fichier permet de proposer un type générique (polymorphe) aux programmeurs le système gérant la multiplicité des formats effectifs (différenttes marques et conceptions de disques dur par exemple).

L'inode définit le fichier, soit principalement les informations :
- la localisation sur disque,
- le propriétaire et le groupe propriétaire,
- les droits d'accès des différents utilisateurs,
- la taille,
- la date de création.

On trouvera sur d'autre systèmes d'autres structures d'information pour décrire les fichiers, par exemple NT utilise des "objets files records".

Un nom est lié à un fichier (une référence indique un fichier) mais un fichier n'est pas lié à une référence, un fichier peut exister sans avoir de nom dans l'arborescence.

2.2 Fichiers ordinaires / Fichiers spéciaux.

Le système est un utilisateur du système de gestion de fichier et en temps que créateur il définit quelques contenus structurés ces fichiers auront de ce fait des accès règlementés.

Pour le système les fichiers sont donc organisés en deux grandes familles :

les fichiers standards que sont par exemple les fichiers texte, les exécutables, etc. C'est-à-dire tout ce qui est manipulé et structuré par les utilisateurs.

Les fichiers spéciaux périphériques, mémoire, et autre fichiers "physiques" ou logique. Ces fichiers ont une structure interne définie (par les développeurs du système) qui doit être respecté c'est pourquoi leur manipulation n'est possible que par par l'intermédiaire du système (encore un bon exemple d'encapsulation).

Les catalogues sont des fichiers spéciaux, il faut pour les manipuler physiquement faire appel au système ce qui en protège la structure[1].

Les fichiers physiques dans le répertoire /dev (dev comme devices dispositifs matériels, les périphériques et quelques dispositifs logiques)

- Character devices (périphériques ou la communication ce fait octets par octets)
  les terminaux (claviers, écrans)
  les imprimantes
  la mémoire
  etc
- Block devices (périphériques ou la communication ce fait par groupe d'octet appelés blocs)
  les disques
  les bandes magnétiques
  etc

Les fichiers à usages logiques et non physiques

- liens symboliques
- pseudo-terminaux
- sockets
- tubes nommés
  Ce dernier type de fichiers spéciaux est utilisé pour servir d'interface entre disques, entre machines et simuler : des terminaux, des lignes de communication, etc.

Cette distinction entre fichier ordinaire et spéciaux et tout simplement le fait que le système est un utilisateur comme les autres des fichiers. Pour certains fichier le système utilise une structure interne spéciale (d'ou le nom) qui ne doit pas être modifier sous peine de comportement indéfini. Pour se protéger le système ne permet pas l'accès direct aux informations c'est lui qui fait toutes les entrées sortie sur les fichiers spéciaux de façon a en assurer l'intégrité. Ceci est indépendant du système de droits d'accès, la structure du code du noyau ne permet pas d'autres accès que les accès "spéciaux" [2].

2.3 Organisation utilisateur des Disques

Comment permettre aux utilisateurs d'identifier les données sur les supports de masse ?
Le système le plus répendu aujourd'hui est un système arborescent avec des fichiers utilisés comme

[1] les répertoires sont resté accessible longtemps en lecture comme des fichiers ordinaires mais l'accès en écriture était contraint, pour assurer la structure arborescente acyclique. Aujourd'hui tout les accès au répertoires ont contraint et on a un ensemble d'appels système spécifiques pour réaliser des entrés sortie dans les repertoires. `opendir(3), closedir(3), dirfd(3), readdir(3), rewinddir(3), scandir(3),seekdir(3), telldir(3)` approche qui permet d'être effectivement plus indépendant sur la structure interne des répertoires, avec des système plus efficaces que les listes utilisées dans les première implémentations. Voire Reiser fs par exemple.

[2] Pour plsu d'information sur le sujet aller voire dans les sources les structures de sgf et d'inode TODO : nom de fichiers concernés .

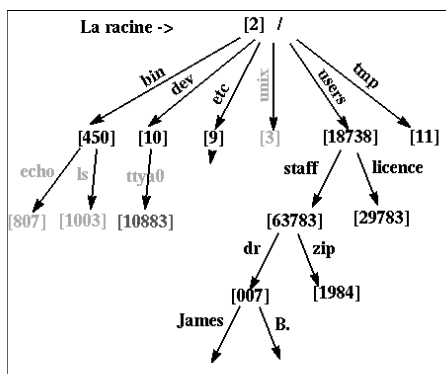

Fig. 2.1 – l'arborescence MULTICS

noeud de l'arbre qui permette de lister les fichiers et les sous arbres qu'il contienti, d'autres organisations "plates" existe ou l'on organise les fichiers en utilisans des types et des extentions de nom de fichier pour "organiser".

Les arborescences de fichiers et de catalogues, organisées comme un graphe acyclique [3], apparaissent avec le projet MULTICS.
Cette organisation logique du disque a les avantages suivants :

Une racine, un accès absolu aisé (à la différence de certains système qui ont de nombreuses "racines").
Une structure dynamique.
Une grande puissance d'expression.
Un graphe acyclique.

L'organisation est arborescente avec quelques connections supplémentaires (liens multiples sur un même fichier) qui en font un graphe. Mais ce graphe doit rester acyclique, pour les raisons suivantes :

L'ensemble des algorithmes simples utilisables sur des graphe acycliques comme le parcours, la vérification des fichiers libres, etc. deviennent beaucoup plus difficiles à écrire pour des graphes admettant des cycles.

Des algorithmes de ramasse-miettes doivent être utilisés pour savoir si certains objets sont utilisés on non et pour récuperer les inodes ou blocs perdus après un crash.

Tous les algorithmes de détection dans un graphe quelconque ont une complexité beaucoup plus grande que ceux qui peuvent profiter de l'acyclicité du graphe.

Sous Unix nous sommes assurés que le graphe est acyclique car il est interdit d'avoir plusieurs références pour un même catalogue (sauf la référence spéciale "..").

Sous UNIX c'est un graphe acyclique !

2.4 Les inodes

L'inode est le passage obligé de tous les échanges entre le système de fichier et la mémoire. L'inode est la structure qui contient toutes les informations sur un fichier donné à l'exception de

[3]Ce n'est pas un arbre car un fichier peut avoir plusieurs références

sa référence dans l'arborescence (son nom), l'arborescence n'étant qu'un outil de référencement des fichiers.

Les informations stockées dans une inode disque sont :
- utilisateur propriétaire
- groupe propriétaire
- type de fichier
  - \- fichiers ordinaires
  - d répertoire (dyrectory)
  - b mode bloc
  - c mode caractère
  - l lien symbolique
  - p pour une fifo (named pipe)
  - s pour une socket
- droits d'accès (ugo*rwx)
- date de dernier accès
- date de dernière modification
- date de dernière modification de l'inode
- taille du fichier
- adresses des blocs-disque contenant le fichier.

Dans une inode en mémoire (fichier en cours d'utilisation par un processus) on trouve d'autres informations supplémentaires :

le statut de l'inode

{ locked,
waiting P
inode à écrire,
fichier à écrire,
le fichier est un point de montage
}

Et deux valeurs qui permettent de localiser l'inode sur un des disques logiques :

Numéro du disque logique
Numéro de l'inode dans le disque

cette information est inutile sur le disque (on a une bijection entre la position de l'inode sur disque et le numéro d'inode).
On trouve aussi d'autres types d'informations comme l'accès à la table des verrous ou bien des informations sur les disques à distance dans les points de montage.

2.5 Organisation des disques System V

L'organisation disque décrite sur la figure 2.2 est la plus simple que l'on peut trouver de nos jours sous UNIX, il en existe d'autres où l'on peut en particulier placer un même disque logique sur plusieurs disques physiques (dangereux), certaines où les blocs sont fragmentables, etc.

Boot bloc utilisé au chargement du système.

Super Bloc il contient toutes les informations générales sur le disque logique.

Inode list Table des inodes.

blocs les blocs de données chainés à la création du disque (`mkfs`).

Les blocs de données ne sont pas fragmentables sous Système V.

Structure du système de fichier sur un disque logique

Plusieurs disques logiques sur un disque physique

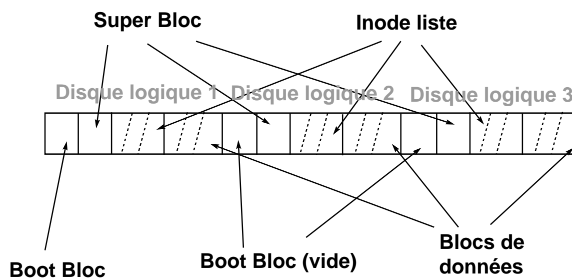

Fig. 2.2 – Organisation des blocs et des inodes (SYS V)

2.6 Adressage des blocs dans les inodes

Le système d'adressage des blocs dans les inodes (système V) consiste en 13 adresses de blocs. Les dix premières adresses sont des adresses qui pointent directement sur les blocs de données du fichier. Les autres sont des adresses indirectes vers des blocs de données contenant des adresses. La figure 2.3 nous montre les trois niveaux d'indirection. L'intérêt de cette représentation est d'économiser sur la taille des inodes tout en permettant un accès rapide au petits fichiers (la majorité des fichiers sont petits). Mais en laissant la possibilité de créer de très gros fichiers :

$$10 + 256 + (256 \times 256) + (256 \times 256 \times 256)$$

blocs disques.

2.7 Allocation des inodes d'un disque

L'allocation des inodes est réalisée en recherchant dans la zone des inodes du disque une inode libre. Pour accélérer cette recherche : un tampon d'inodes libres est géré dans le SuperBloc, de plus l'indice de la première inode libre est gardé en référence dans le SuperBloc afin de redémarrer la recherche qu'à partir de la première inode réellement libre.

Mais ce système a une faille qu'il faut prévoir dans l'écriture dans l'algorithme ialloc d'allocation d'inode, cette faille est décrite dans la Figure 2.10

2.8 Allocation des blocs-disque

L'algorithme utilisé pour gérer l'allocation des inodes s'appuie sur le fait que l'on peut tester si une inode est libre ou non en regardant son contenu. Ceci n'est plus vrai pour les blocs. La solution est de chaîner les blocs. Ce chaînage est réalisé par blocs d'adresses pour accélérer les accès et profiter au maximum du buffer cache. Il existe donc un bloc d'adresses dans le super bloc qui sert de zone de travail pour l'allocateur de blocs. L'utilisation de ce bloc et le mécanisme d'allocation sont décrits dans les Figures 2.11 à 2.16

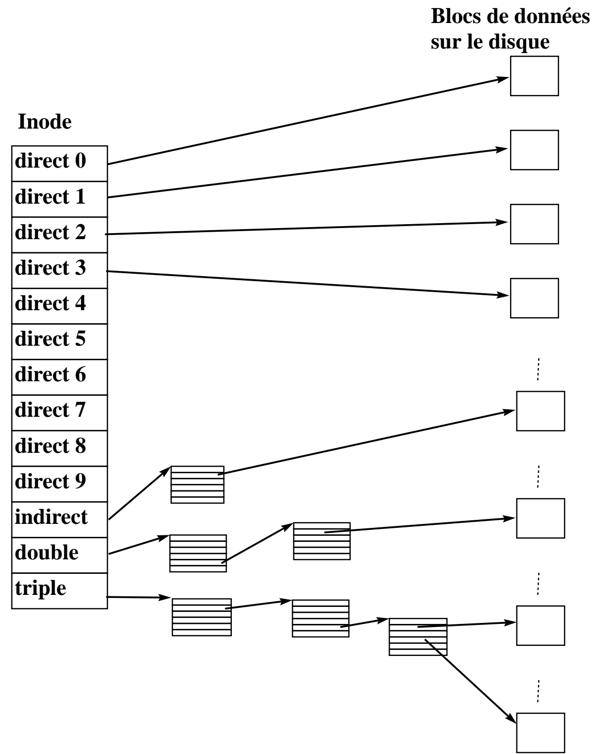

FIG. 2.3 – Adressage direct et indirect des inode UNIX

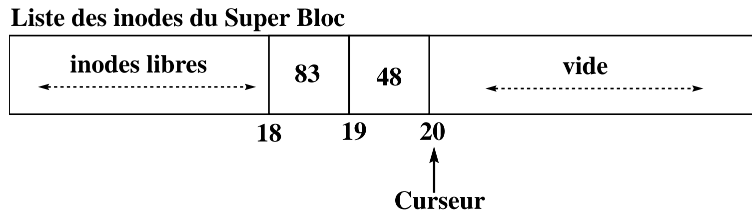

FIG. 2.4 – Inodes libres dans le SuperBloc.

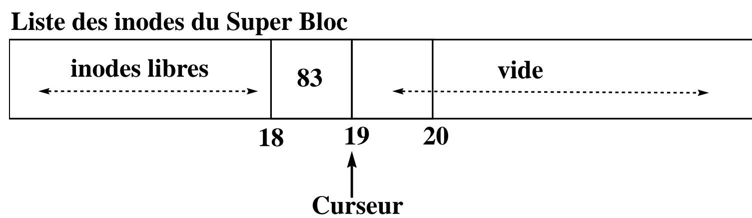

FIG. 2.5 – Allocation d'une inode.

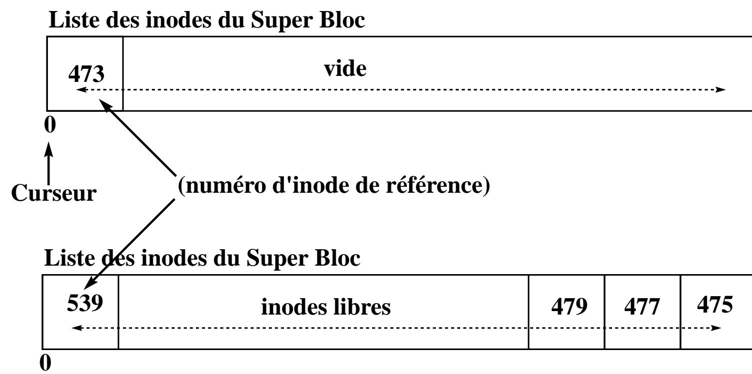

FIG. 2.6 – Si le SuperBloc est vide.

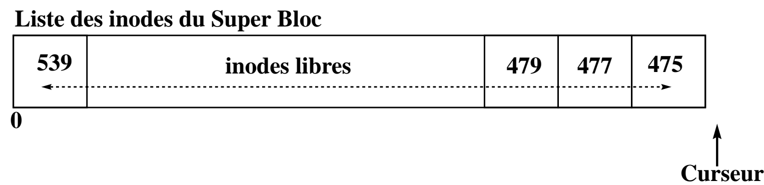

FIG. 2.7 – Libération d'une inode avec le SuperBloc plein.

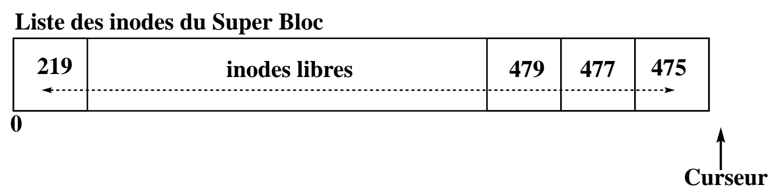

FIG. 2.8 – Le numéro d'inode inférieur au numéro de référence.

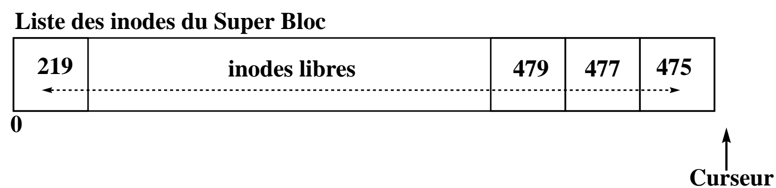

FIG. 2.9 – Le numéro d'inode supérieur au numéro de référence.

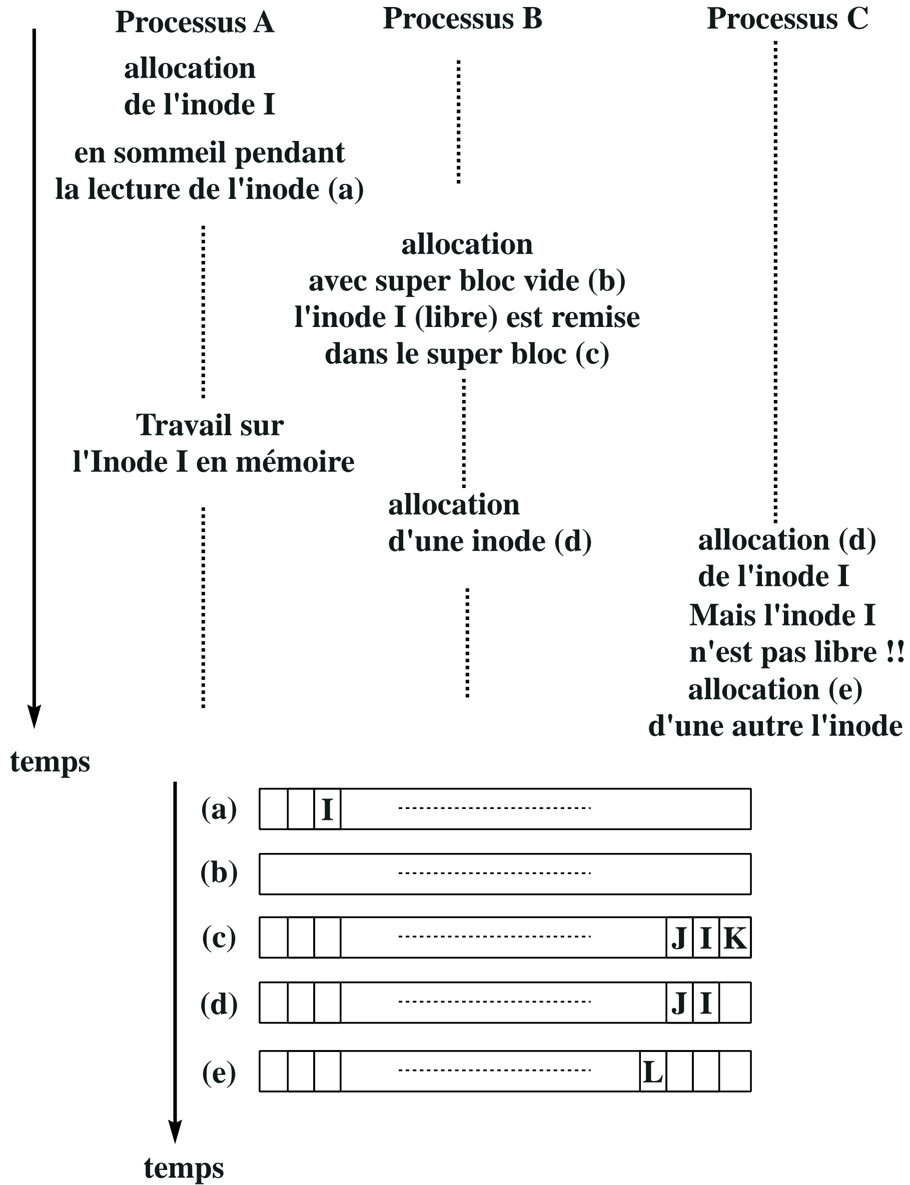

Fig. 2.10 – Faille de l'algorithme d'allocation.

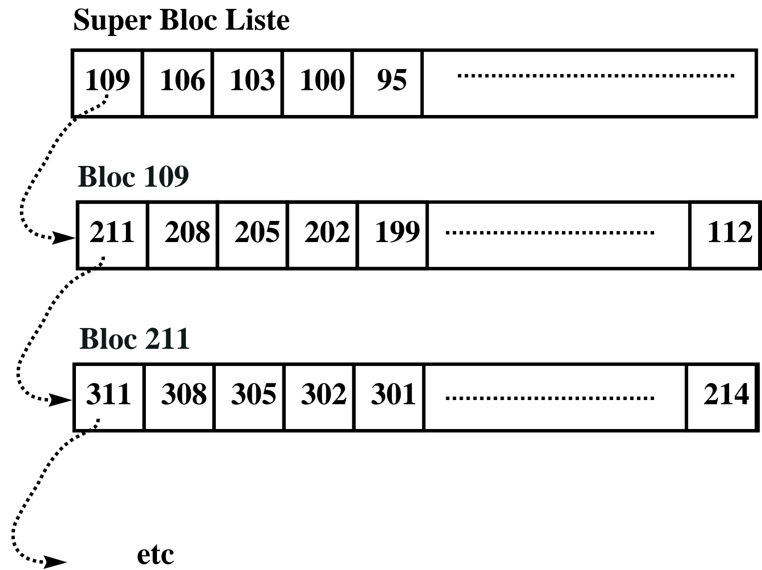

FIG. 2.11 – Liste chainée de blocs.

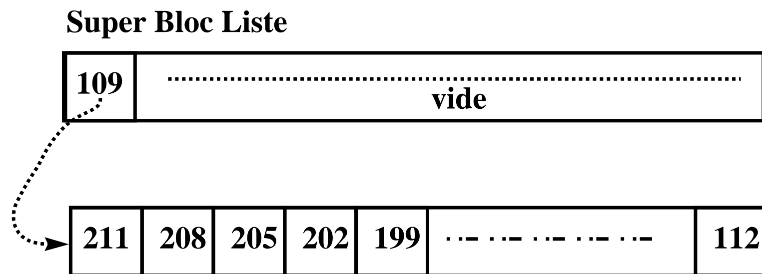

FIG. 2.12 – Etat initial du SuperBloc.

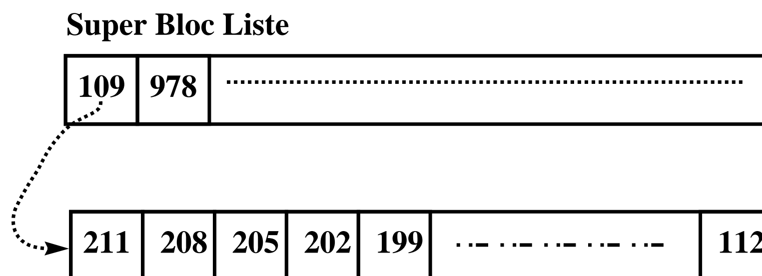

FIG. 2.13 – Libération du bloc 978.

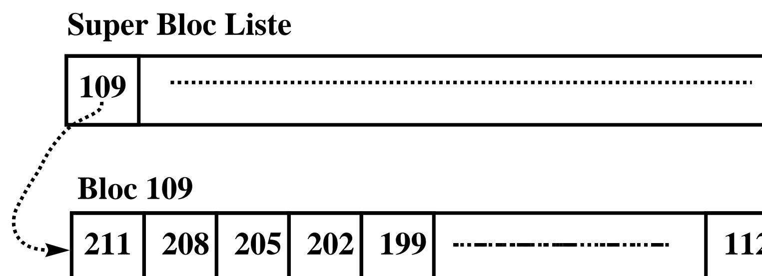

FIG. 2.14 – Allocation du bloc 978.

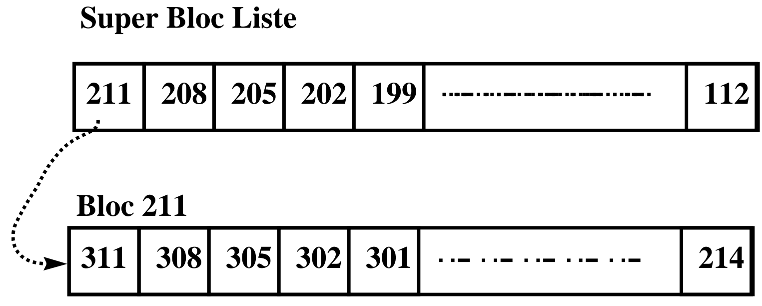

Fig. 2.15 – Allocation du bloc 109.

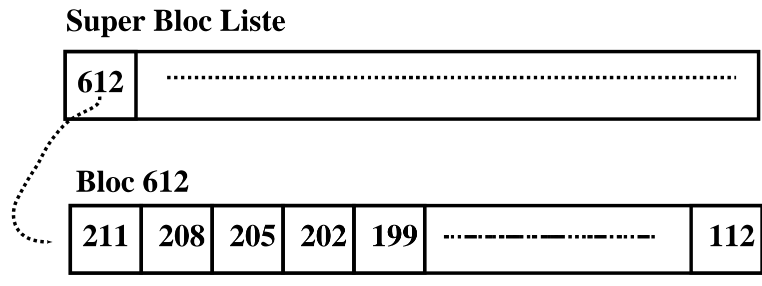

Fig. 2.16 – Libération du bloc 612.

Chapitre 3

Le Buffer Cache

3.1 Introduction au buffer cache

Le buffer cache est un ensemble de structures de données et d'algorithmes qui permettent de minimiser le nombre des accès disque.

Ce qui est très important car les disques sont très lents relativement au CPU et un noyau qui se chargerait de toutes les entrées/sorties serait d'une grande lenteur et l'unité de traitement ne serait effectivement utilisée qu'à un faible pourcentage (voir Historique).
Deux idées pour réduire le nombre des accès disques :

1. bufferiser les différentes commandes d'écriture et de lecture de façon à faire un accès disque uniquement pour une quantité de données de taille raisonnable (un bloc disque).
2. Eviter des écritures inutiles quand les données peuvent encore être changées (écriture différées).

3.1.1 Avantages et désavantages du buffer cache

– Un accès uniforme au disque. Le noyau n'a pas à connaître la raison de l'entrée-sortie. Il copie les données depuis et vers des tampons (que ce soient des données, des inodes ou le superbloc). Ce mécanisme est modulaire et s'intègre facilement à l'ensemble du système qu'il rend plus facile à écrire.
– Rend l'utilisation des entrées-sorties plus simple pour l'utilisateur qui n'a pas à se soucier des problèmes d'alignement, il rend les programmes portables sur d'autres UNIX [1].
– Il réduit le trafic disque et de ce fait augmente la capacité du système. Attention : le nombre de tampons ne doit pas trop réduire la mémoire centrale utilisable.
– L'implémentation du buffer cache protège contre certaines écritures "concurrentes"
– L'écriture différée pose un problème dans le cas d'un crash du système. En effet si votre machine s'arrête (coupure de courant) et que un (ou plusieurs) blocs sont marqués "à écrire" ils n'ont donc pas étés sauvegardés physiquement. L'intégrité des données n'est donc pas assurée en cas de crash.
– Le buffer cache nécessite que l'on effectue une recopie (interne à la mémoire, de la zone utilisateur au cache ou inversement) pour toute entrée-sortie. Dans le cas de transferts nombreux ceci ralentit les entrées-sorties .

3.2 Le buffer cache, structures de données.

Le statut d'un bloc cache est une combinaison des états suivants :

verrouillé l'accès est reservé à un processus.

[1] Les problèmes d'alignement existent toujours quand on transfère des données, cf. protocoles XDR,RPC

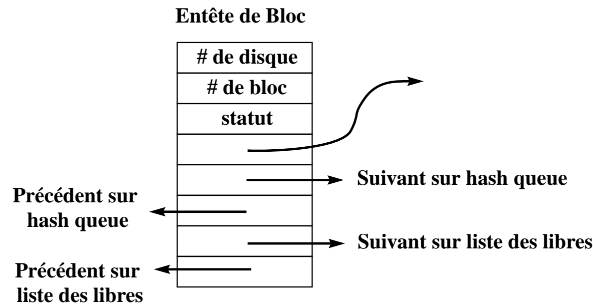

Fig. 3.1 – Structure des entêtes de Bloc du Buffer Cache

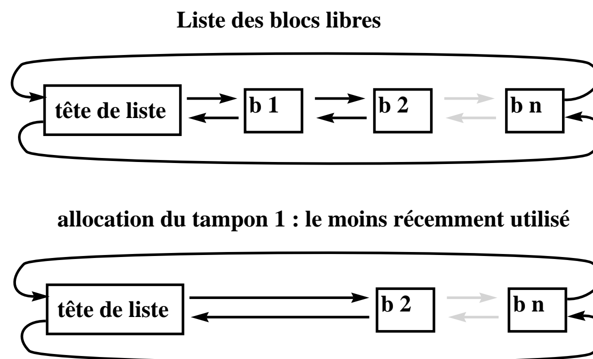

Fig. 3.2 – La liste des tampons libres.

valide (les données contenues dans le bloc sont valides).

"à écrire" les données du bloc doivent être écrites sur disque avant de réallouer le bloc (c'est de l'écriture retardée).

actif le noyau est en train d'écrire/lire le bloc sur le disque.

attendu un processus attend la libération du bloc.

3.2.1 La liste doublement chaînée des blocs libres

Les tampons libres appartiennent simultanément à deux listes doublement chaînées : la liste des blocs libres et la hash-liste correspondant au dernier bloc ayant été contenu dans ce tampon.

L'insertion dans la liste des tampons libres se fait en fin de liste, la suppression (allocation du tampon à un bloc donné) se fait en début de liste, ainsi le tampon alloué est le plus vieux tampon libéré[2]. Ceci permet une réponse immédiate si le bloc correspondant est réutilisé avant que le tampon ne soit alloué à un autre bloc.

3.3 L'algorithme de la primitive `getblk`

```
Algorithme getblk (allocation d'un tampon)
entree : # disque logique , # de block
sortie : un tampon verrouille utilisable pour manipuler bloc
{
    while (tampon non trouve)
```

[2]ordre fifo : first in first out

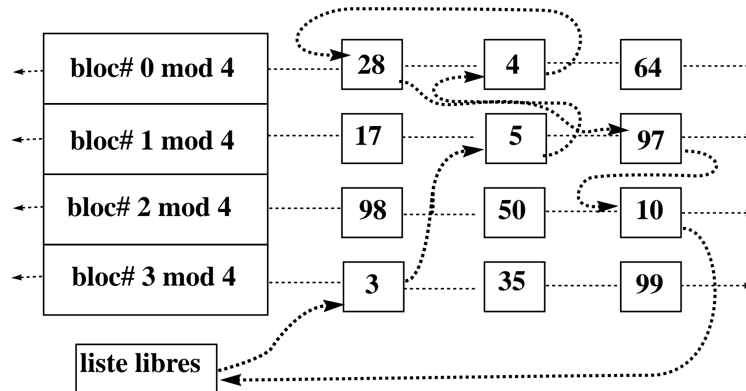

FIG. 3.3 – Etat du buffer cache avant les scénarios 1, 2 et 3.

```
    {
        if (tampon dans sa hash liste)
        {
                if (tampon actif )
                {
       [5]          sleep attente de la liberation du tampon
                    continuer
                }
       [1]      verrouiller le tampon
                retirer le tampon de la liste des tampons libres
                retourner le tampon
        }
        else  /* n'est pas dans la hash liste */
        {
                if (aucun tampon libre )
                {
        [4]         sleep attente de la liberation d'un tampon
                    continuer
                }
                retirer le tampon de la liste libre
        [3]     if (le tampon est a ecrire)
                {
                    lancer la sauvegarde sur disque
                    continuer
                }
        [2]     retirer le buffer de son ancienne liste
                 de hashage, le placer sur la nouvelle
                retourner le tampon
        }
    }
}
```

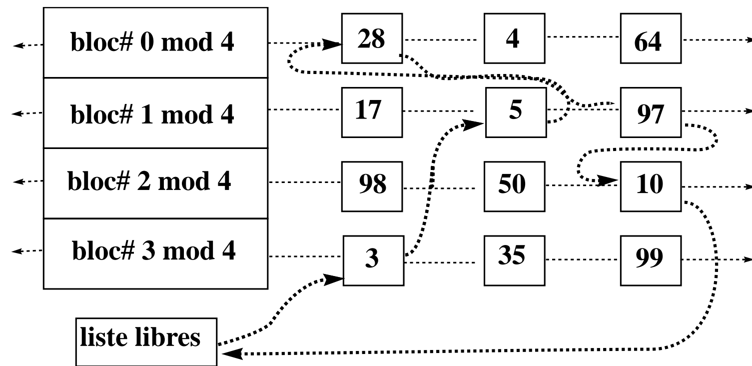

FIG. 3.4 – Scénario 1- Demande d'un tampon pour le bloc-disque 4.

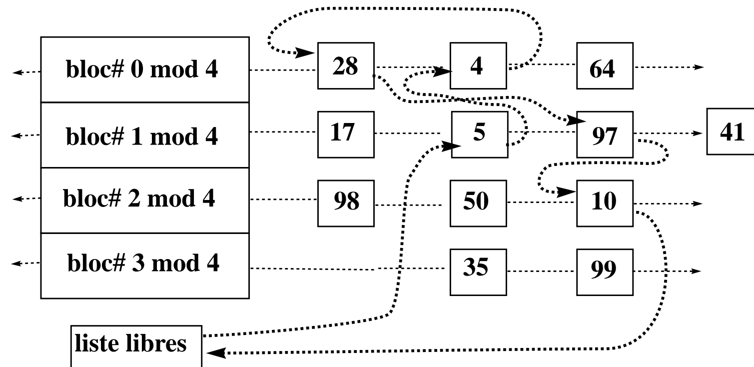

FIG. 3.5 – Scénario 2- Demande d'un tampon pour le bloc-disque 41.

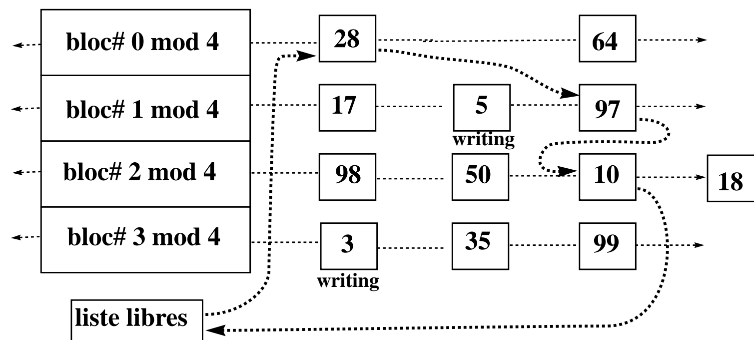

FIG. 3.6 – Scénario 3- Demande pour le bloc 18 (3 & 5 marqués à écrire).

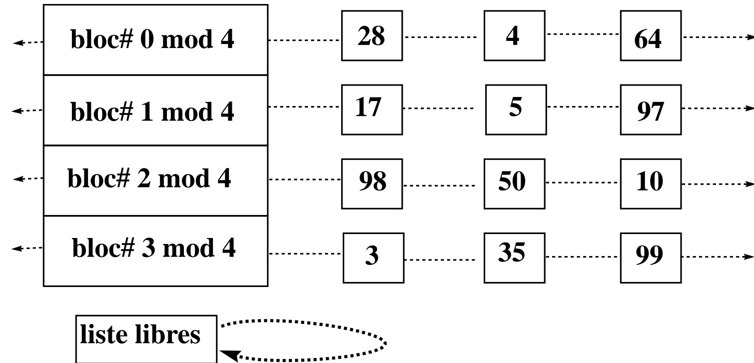

FIG. 3.7 – Scénario 4- Plus de blocs libres.

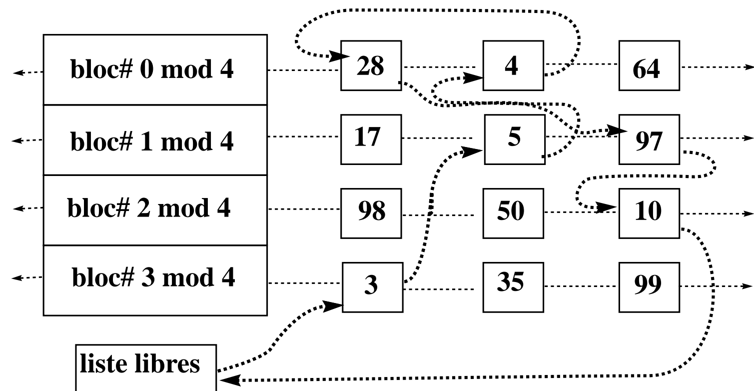

FIG. 3.8 – Scénario 5- Demande pour le bloc 17 qui est déjà utilisé.

Chapitre 4

La bibliothèque standard

4.1 Les descripteurs de fichiers.

Le fichier d'inclusion <stdio.h> contient la définition du type FILE. Ce type est une structure contenant les informations nécessaires au système pour la manipulation d'un fichier ouvert. Le contenu exact de cette structure peut varier d'un système à l'autre (UNIX, VMS, autre).

Toutes les fonctions d'E/S utilisent en premier argument un pointeur sur une telle structure : FILE *. Le rôle de cet argument est d'indiquer le flux sur lequel on doit effectuer l'opération d'écriture ou de lecture.

Pour pouvoir utiliser une fonction d'entrée-sortie il faut donc avoir une valeur pour ce premier argument, c'est le rôle de la fonction fopen de nous fournir ce pointeur en "ouvrant" le fichier. Les deux fonctions printf et scanf sont des synonymes de

```
fprintf(stdout, format, ...)
```

et

```
fscanf(stdin, format, ...)
```

où stdout et stdin sont des expressions de type FILE * définies sous forme de macro-définitions dans le fichier <stdio.h> . Avec POSIX ce sont effectivement des fonctions.

Sur les système de la famille UNIX les fichiers ouverts par un processus le restent dans ses fils. Par exemple le shell a en général trois flux standards :

stdin le terminal ouvert en lecture.

stdout le terminal ouvert en écriture.

stderr le terminal ouvert en écriture, et en mode non bufferisé.

ainsi si l'exécution d'un programme C est réalisée à partir du shell le programme C a déjà ces trois descripteurs de fichiers utilisables. C'est pourquoi il est en général possible d'utiliser printf et scanf sans ouvrir préalablement de fichiers. Mais si l'entrée standard n'est pas ouverte, scanf échoue :

```
#include <stdio.h>
main()
{
    int i;

    if (scanf("%d", &i) == EOF)
    {
        printf("l\'entree standard est fermee\n");
    }
    else
    {
        printf("l\'entree standard est ouverte\n");
```

```
    }
}
```

Compilé,(`a.out`), cela donne les deux sorties suivantes :

```
$ a.out
l'entree standard est ouverte
$ a.out <&- # fermeture de l'entree standard en ksh
l'entree standard est fermee
```

De même `printf` échoue si la sortie standard est fermée.

4.1.1 Ouverture d'un fichier

La fonction de la bibliothèque standard `fopen` permet d'ouvrir un fichier ou de le créer.

```
#include <stdio.h>
FILE  *fopen(const char *filename,
             const char *type);
```

filename est une référence absolue ou relative du fichier à ouvrir ; si le fichier n'existe pas alors il est créé *si et seulement si* l'utilisateur du processus a l'autorisation d'écrire dans le répertoire.

type est une des chaînes suivantes :

"r" ouverture en lecture au début du fichier

"w" ouverture en écriture au début du fichier avec écrasement du fichier si il existe (le fichier est vidé de son contenu à l'ouverture).

"a" ouverture en écriture à la fin du fichier (mode append).

"r+","w+","a+" ouverture en lecture écriture respectivement au début du fichier, au début du fichier avec écrasement, à la fin du fichier.

```
FILE *f;
...
if ((f = fopen("toto", "r")) == NULL)
{
    fprintf(stderr, "impossible d'ouvrir toto\n");
    exit(1);
}
...
```

La fonction retourne un pointeur sur un descripteur du fichier ouvert ou `NULL` en cas d'échec, (accès interdit, création impossible, etc).

4.1.2 Redirection d'un descripteur : `freopen`

Permet d'associer un descripteur déjà utilisé à une autre ouverture de fichier. Ceci permet de réaliser facilement les redirections du shell.

```
FILE *freopen(const char *ref,
              const char *mode,
              FILE *f)
```

Par exemple les redirections de la ligne shell :

```
com <ref1 >>ref2
```

peuvent être réalisées avec

```
if (!freopen("ref1", "r", stdin) || !freopen("ref2", "a", stdout))
{
    fprintf(stderr, "erreur sur une redirection\n");
    exit(1);
}
execl("./com", "com", NULL);
```

4.1.3 Création de fichiers temporaires

La fonction

```
#include <stdio.h>
FILE *tmpfile(void);
```

crée et ouvre en écriture un nouveau fichier temporaire, qui sera détruit (un `unlink` est réalisé immédiatement) à la fin de l'exécution du processus, attention le descripteur est la aussi hérité par les fils, et il faut en gérer le partage. Cette fonction utilise la fonction

```
int mkstemp(char *patron);
```

Les 6 dernier caractère du chemin patron doivent être "XXXXXX" il seront remplacé par une chaine rendant le nom unique, ce chemin sera utilisé pour ouvrir un fichier temporaire avec l'option création et echec sur création avec les droit 0600 ce qui permet d'éviter des troux de sécurité. La fonction retourne le descripteur. Attention `mkstemp` n'assure pas que le fichier sera détruit après utilisation comme c'etait le cas avec `tmpfile` , par contre il devient très difficile de réaliser une attaque sur les fichiers temporaire créer par mkstemp.

4.1.4 Ecriture non formatée

Les deux fonctions suivantes permettent d'écrire et de lire des zones mémoire, le contenu de la mémoire est directement écrit sur disque sans transformation, et réciproquement le contenu du disque est placé tel quel en mémoire. L'intérêt de ces fonctions est d'obtenir des entrées sorties plus rapides et des sauvegardes disque plus compactes mais malheureusement illisibles (binaire). D'autre part les fonction de lecture et d'ecriture sont exactement symétrique ce qui n'est pas le cas de `scanf` et `printf`

```
#include <stdio.h>
int fwrite(void *add, size_t ta, size_t nbobjets, FILE *f);
```

Ecrit `nbobjets` de taille `ta` qui se trouvent à l'adresse `add` dans le fichier de descripteur `f`.

```
#include <stdio.h>
int fread(void *add, size_t ta, size_t nbobjets, FILE *f);
```

Lit `nbobjets` de taille `ta` dans le fichier de descripteur `f` et les place à partir de l'adresse `add` en mémoire.

Attention : La fonction `fread` retourne 0 si l'on essaye de lire au delà du fichier. Pour écrire une boucle de lecture propre on utilise la fonction `feof(FILE *)` :

```
int n[2];
while (fread(n, sizeof(int), 2, f), !feof(f))
      printf("%d %d \n", n[0], n[1]);
```

4.1.5 Accès séquentiel

On distingue deux techniques d'accès aux supports magnétiques :

- L'accès séquentiel qui consiste à traiter les informations dans l'ordre où elle apparaissent sur le support (bandes). Le lecteur physique avance avec la lecture, et se positionne sur le début de l'enregistrement suivant.
- L'accès direct qui consiste à se placer directement sur l'information sans parcourir celles qui la précèdent (disques). Le lecteur physique reste sur le même enregistrement après une lecture.

En langage C l'accès est séquentiel mais il est possible de déplacer le "pointeur de fichier" c'est à dire sélectionner l'indice du prochain octet à lire ou écrire.

Comme nous venons de le voir dans les modes d'ouverture, le pointeur de fichier peut être initialement placé en début ou fin de fichier.

Les quatre fonctions d'entrée-sortie (`fgetc, fputc, fscanf, fprintf`) travaillent séquentiellement à partir de cette origine fixée par `fopen`, et modifiable par `fseek`.

4.1.6 Manipulation du pointeur de fichier

Le pointeur de fichier est un entier `long` qui indique à partir de quel octet du fichier la prochaine fonction d'entrée-sortie doit s'effectuer.

En début de fichier cet entier est nul.

```
#include <stdio.h>
int fseek(FILE *f, long pos, int direction);
```

`f` le descripteur du fichier dans lequel ont déplace le pointeur.

direction est une des trois constantes entières suivantes :

SEEK_SET positionnement sur l'octet pos du fichier

SEEK_CUR positionnement sur le `pos`-ième octet après la position courante du pointeur de fichier. (équivalent à SEEK_SET courant+pos).

SEEK_END positionnement sur le `pos`-ième octet après la fin du fichier.

Remarquer que `pos` est un entier signé : il est possible se placer sur le 4ième octet avant la fin du fichier :

```
fseek(f, -4L, SEEK_END);
```

4.1.7 Un exemple d'accès direct sur un fichier d'entiers.

La fonction suivante lit le n-ième entier d'un fichier d'entiers préalablement écrit grâce à `fwrite` :

```
int lirenieme(int n, FILE *f)
{
    int buf;

    fseek(f, sizeof(int)*(n-1), SEEK_SET);
    fread(&buf, sizeof(int), 1, f);
    return buf;
}  \istd{fseek}\istd{fread}
```

4.1.8 Les autres fonctions de déplacement du pointeur de fichier.

La fonction `ftell`

```
long int ftell(FILE *);
```

retourne la position courante du pointeur.
La fonction `rewind`

```
void rewind(FILE *f);
```

équivalent à : `(void) fseek (f, 0L, 0)`

4.2 Les tampons de fichiers de stdlib.

La bibliothèque standard utilise des tampons pour minimiser le nombre d'appels système. Il est possible de tester l'efficacité de cette bufferisation en comparant la vitesse de recopie d'un même fichier avec un tampon de taille 1 octet et un tampon adapté à la machine, la différence devient vite très importante. Une façon simple de le percevoir est d'écrire un programme `com` qui réalise des écritures sur la sortie standard ligne par ligne, de regarder sa vitesse puis de comparer avec la commande suivantes :`com | cat` la bibliothèque standard utilisant des buffer différents dans les deux cas une différence de vitese d'exécution est perceptible (sur une machine lente la différence de vitesse est évidente, mais elle existe aussi sur une rapide...).

4.2.1 Les modes de bufferisation par défaut.

Le mode de bufferisation des fichiers ouverts par la bibliothèque standard dépend du type de périphérique.

- Si le fichier est un **terminal** la bufferisation est faite ligne à ligne.
  En *écriture* le tampon est vidé à chaque écriture d'un `'\n'` , ou quand il est plein (première des deux occurences).
  En *lecture* le tampon est rempli après chaque validation (RC), si l'on tape trop de caractères le terminal proteste (beep) le buffer clavier étant plein.
- Si le fichier est sur un **disque magnétique**
  En *écriture* le tampon est vidé avant de déborder.
  En *lecture* le tampon est rempli quand il est vide.

Le shell de login change le mode de bufferisation de **stderr** qui est un fichier terminal à non bufferisé.

Nous avons donc à notre disposition trois modes de bufferisation standards :
- Non bufferisé (sortie erreur standard),
- Bufferisé par ligne (lecture/écriture sur terminal),
- Bufferisé par blocs (taille des tampons du buffer cache).

Un exemple de réouverture de la sortie standard, avec perte du mode de bufferisation :

```
#include <stdio.h>
main()
{
    freopen("/dev/tty", "w", stderr);
    fprintf(stderr, "texte non termine par un newline ");
    sleep(12);
    exit(0); /* realise fclose(stderr) qui realise fflush(stderr) */
}
```

Il faut attendre 12 secondes l'affichage.

4.2.2 Manipulation des tampons de la bibliothèque standard.

Un tampon alloué automatiquement (malloc) est associé à chaque ouverture de fichier par `fopen` au moment de la première entrée-sortie sur le fichier.

La manipulation des tampons de la bibliothèque standard comporte deux aspects :

1. Manipulation de la bufferisation de façon ponctuelle (vidange).
2. Positionnement du mode de bufferisation.

Manipulations ponctuelles

La fonction suivante permet de vider le tampon associé au FILE * f :

```
#include <stdio.h>
fflush(FILE *f);
```

En écriture force la copie du tampon associé à la structure f dans le tampon système (ne garantit pas l'écriture en cas d'interruption du système !).
En lecture détruit le contenu du tampon, si l'on est en mode ligne uniquement jusqu'au premier caractère `'\n'`.

La fonction fclose() réalise un fflush() avant de fermer le fichier.
La fonction exit() appel fclose() sur tous les fichiers ouvert par fopen (freopen,tmpfile,...) avant de terminer le processus.

Manipulations du mode de bufferisation et de la taille du tampon.

La primitive

```
int setvbuf(FILE *f,
      char *adresse,
      int mode,
      size_t taille);
```

permet un changement du mode de bufferisation du fichier *f* avec un tampon de taille *taille* fourni par l'utilisateur à l'adresse *adresse* si elle est non nulle, avec le *mode* défini par les macro-définitions suivantes (`<stdio.h>`) :

```
_IOFBF        bufferise
_IONBF        Non bufferise
_IOMYBUF      Mon buffer
_IOLBF        bufferise par ligne (ex: les terminaux)
```

Attention : Il ne faut pas appeler cette fonction après l'allocation automatique réalisée par la bibliothèque standard après le premier appel à une fonction d'entrée-sortie sur le fichier.
Il est fortement conseillé que la zone mémoire pointée par *adresse* soit au moins d'une taille égale à *taille*.

Seul un passage au mode bufferisé en ligne ou non bufferisé peut être réalisé après l'allocation automatique du tampon, au risque de perdre ce tampon (absence d 'appel de `free`). Ce qui permet par exemple de changer le mode de bufferisation de la sortie standard après un `fork`. Attention ce peut être dangereux, pour le contenu courant du tampon comme le montre l'exemple suivant.

Avant cette fonction de norme POSIX on utilisait trois fonctions :

```
void setbuf(FILE *f, char *buf);
void setbuffer(FILE *f,char *adresse,size_t t);
void setlignebuf(FILE *f);
```

```
#include <stdio.h>
main()
{
  printf("BonJour ");
  switch(fork())
  {
    case -1 :
       exit(1);
    case  0 :
       printf("je suis le fils");
/* version 1 sans la ligne suivante version 2 avec  */
       setbuffer(stdout, NULL, 0);
       sleep(1);
       printf("Encore le fils");
       break;
    default :
       printf("je suis le pere");
       sleep(2);
  }
  printf("\n");
}
version 1
fork_stdlib
BonJour je suis le fils Encore le fils
BonJour je suis le pere
version 2
Encore le fils
BonJour je suis le pere
```

4.3 Manipulation des liens d'un fichier

Changer le nom d'un fichier :

```
int rename(const char *de,const char *vers);
```

permet de renommer un fichier (ou un répertoire). Il faut que les deux références soient de même type (fichier ou répertoire) dans le même système de fichiers.
Rappel : ceci n'a d'effet que sur l'arborescence de fichiers.

Détruire une référence :

```
int remove(const char *filename);
```

Détruit le lien donné en argument, le système récupère l'inode et les blocs associés au fichier si c'était le dernier lien.

4.4 Lancement d'une commande shell

```
#include <stdlib.h>
int system(const char *chaine_de_commande);
```

Crée un processus "/bin/posix/sh" qui exécute la commande ; il y a attente de la fin du shell, (la commande peut elle être lancée en mode détaché ce qui fait que le shell retourne immédiatement

sans faire un `wait`). Ce mécanisme est très coûteux. Attention la commande `system` bloque les signaux SIGINT et SIGQUIT, il faut analyser la valeur de retour de `system` de la même façons que celle de `wait`. Il est conseillé de bloquer ces deux signaux avant l'appel de `system` .

4.5 Terminaison d'un processus

_exit La primitive de terminaison de processus de bas niveau :

```
#include <stdlib.h>
void _exit(int valeur);
```

La primitive `_exit` est la fonction de terminaison "bas niveau"

- elle ferme les descripteurs ouverts par *open, opendir* ou hérités du processus père.
- la `valeur` est fournie au processus père qui la récupère par l'appel système *wait*. Cette valeur est le code de retour de processus en shell.

Cette primitive est automatiquement appelée à la fin de la fonction `main` (sauf en cas d'appels récursifs de `main`).

exit La fonction de terminaison de processus de stdlib :

```
#include <stdlib.h>
void exit(int valeur);
```

la fonction *exit* :

- lance les fonctions définies par *atexit*.
- ferme l'ensemble des descripteurs ouverts grâce à la bibliothèque standard (*fopen*).
- détruit les fichiers fabriqués par la primitive *tmpfile*
- appelle *_exit* avec `valeur`.

atexit La primitive `atexit` permet de spécifier des fonctions à appeler en fin d'exécution, elle sont lancées par `exit` dans l'ordre inverse de leur positionnement par `atexit`.

```
#include <stdlib.h>
int atexit(void (*fonction) (void ));
```

Exemple :

```
void bob(void) {printf("coucou\n");}
void bib(void) {printf("cuicui ");}

main(int argc)
{
  atexit(bob);
  atexit(bib);
  if (argc - 1)
    exit(0);
  else
    _exit(0);
}
$ make atexit
cc   atexit.c  -o atexit
$ atexit
$ atexit unargument
cuicui coucou
$
```

4.6 Gestion des erreurs

Les fonctions de la bibliothèque standard positionnent deux indicateurs d'erreur, la fonction suivante les repositionne :

```
void clearerr(FILE *);
```

La fonction `int feof(FILE *)` est vraie si la fin de fichier est atteinte sur ce canal, `int ferror(FILE *)` est vraie si une erreur a eu lieu pendant la dernière tentative de lecture ou d'écriture sur ce canal.
Une description en langue naturelle de la dernière erreur peut être obtenue grace à

```
void perror(const char *message);
```

l'affichage se fait sur la sortie erreur standard (`stderr`).

4.7 Création et destruction de répertoires

Création d'un répertoire vide (même syntaxe que `creat`) :

```
#include <unistd.h>
int mkdir(char *ref, mode_t  mode);
```

Destruction :

```
int rmdir(char *ref);
```

avec les mêmes restrictions que pour les shells sur le contenu du répertoire (impossible de détruire un répertoire non vide).

Chapitre 5

Appels système du Système de Gestion de Fichier

Les appels système d'entrées-sorties ou entrées-sorties de bas niveau sont rudimentaires mais polymorphes, en effet c'est eux qui permettent d'écrire des programmes indépendamment des supports physiques sur lesquels se font les entrées/sorties et de pouvoir facilement changer les supports physiques associés a une entrée-sortie.

Les appels système du système de gestion de fichier sont :

`open/creat` ouverture/création d'un fichier

`read/write` lecture/ecriture sur un fichier ouvert

`lseek` déplacement du pointeur de fichier

`dup,dup2` copie d'ouverture de fichier

`close` fermeture d'un fichier

`mount` chargement d'un disque

`mknode` création d'un inode de fichier spécial

`pipe` création d'un tube

`fcntl` manipulation des caractéristiques des ouvertures de fichiers

Les appels système sont réalisés par le noyau et retournent -1 en cas d'erreur.

5.1 open

```
#include <fcntl.h>
int open(char *ref, int mode, int perm);
```

Ouverture du fichier de référence (absolue ou relative à ".") *ref*.
Le *mode* d'ouverture est une conjonction des masques suivants :

```
O_RDONLY  /* open for reading */
O_WRONLY  /* open for writing */
O_RDWR    /* open for read & write */
O_NDELAY  /* non-blocking open */
O_APPEND  /* append on each write */
O_CREAT   /* open with file create */
O_TRUNC   /* open with truncation */
O_EXCL    /* error on create if file exists*/
```

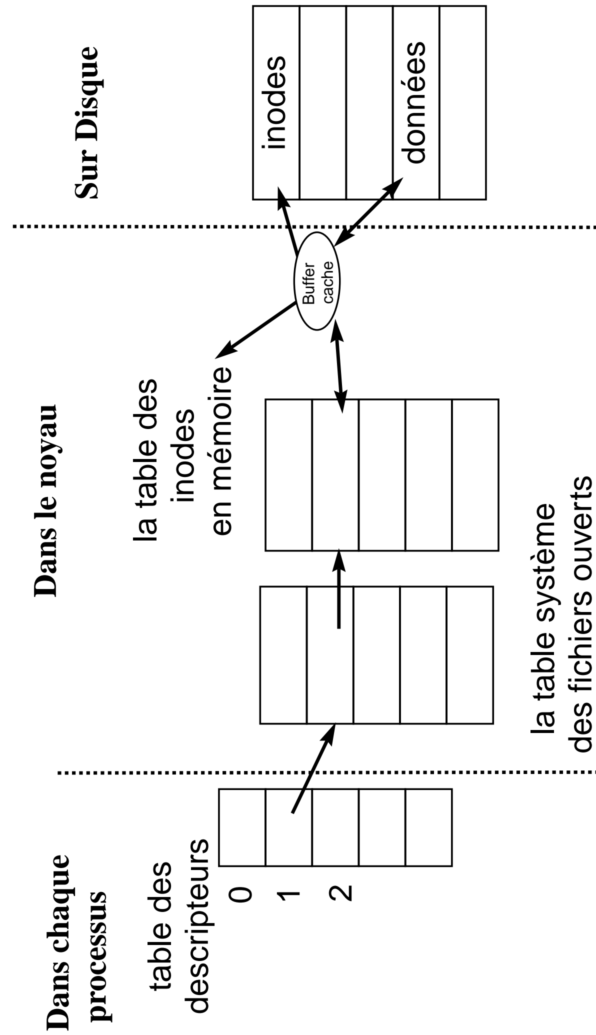

Fig. 5.1 – Tables du système de fichiers.

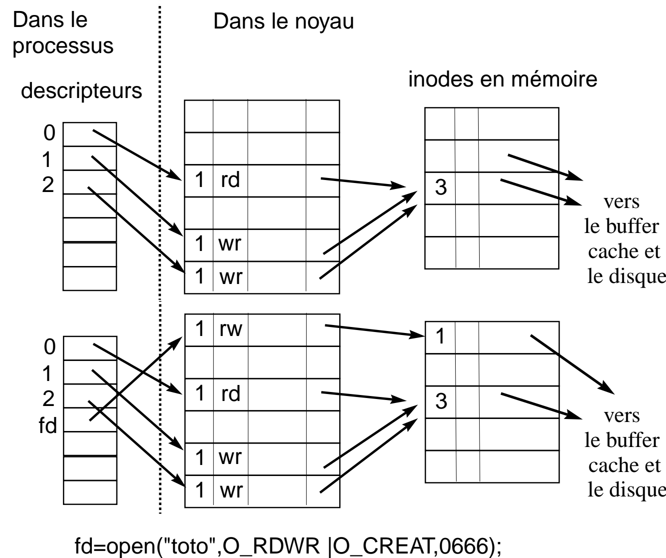

Fig. 5.2 – Avant l'ouverture, descripteurs standard ouverts, puis après l'ouverture de "toto".

Le paramètre *perm*ission n'a de sens qu'à la création du fichier, il permet de positionner les valeurs du champ `mode` de l'inode. Les droits effectivement positionnés dépendent de la valeur de *umask*, grace à la formule `droits = perm & ~ umask`. La valeur par défaut de `umask` est 066 (valeur octale).
La valeur de retour de `open` est le numéro dans la table de descripteurs du processus qui a été utilisé par `open`. Ce numéro est appelé descripteur de l'ouverture. Ce descripteur est utilisé dans les autres appels système pour spécifier l'ouverture de fichier sur laquelle on veut travailler[1], et -1 en cas d'échec de l'ouverture.

5.1.1 Déroulement interne d'un appel de `open`

1. Le système détermine l'inode du fichier référence (*namei*).
2. – Soit l'inode est dans la table des inodes en mémoire.
   – Soit il alloue une entrée et recopie l'inode du disque (*iget*).
3. Le système vérifie les droits d'accès dans le mode demandé.
4. Il alloue une entrée dans la table des fichiers ouverts du système, et positionne le curseur de lecture écriture dans le fichier (offset = 0, sauf dans le cas du mode `O_APPEND` offset=taille du fichier).
5. Le système alloue une place dans la table des descripteurs `_iob` du fichier.
6. Il renvoie au processus le numéro de descripteur, c'est à dire le numéro de l'entrée qu'il vient d'allouer dans le tableau `_iob`.

Si l'opération a échoué dans une des étapes le système renvoie `-1`.

[1] Un même fichier peut être ouvert plusieurs fois.

5.2 `creat`

Création d'un fichier et ouverture en écriture.

```
int creat(char *reference, int permissions);
```

1. Le système détermine l'inode du catalogue où l'on demande la création du fichier.
   (a) Si il existe déjà une inode pour le fichier
      - Le noyau lit l'inode en question (allocation dans la table des inodes en mémoire), vérifie que c'est un fichier ordinaire autorisé en écriture par le propriétaire effectif du processus, sinon échec.
      - Le système libère les blocs de données et réduit la taille du fichier à zéro, il ne modifie pas les droits qu'avait le fichier antérieurement.

   (b) Si n'existait pas d'inode pour le fichier
      - Le système teste les droits en écriture sur le catalogue
      - Il alloue une nouvelle inode (*ialloc*)
      - Il alloue une nouvelle entrée dans la table des inodes en mémoire.

Même suite que pour `open`.

5.3 `read`

```
int nbcharlus = read(int d, char *tampon, int nbalire)
```

***d*escripteur** entrée de la table des descripteurs correspondante au fichier dans lequel doit être effectuée la lecture (fourni par `open`).

nbalire nombre de caractères à lire dans le fichier.

tampon un tableau de caractères alloué par l'utilisateur. Les caractères lus sont placés dans ce tampon.

nbcharlus nombre de caractères effectivement lus, ou -1 en cas d'échec de l'appel système, (droits, ...), la fin de fichier est atteinte quand le nombre de caractères lus est inférieur au nombre de caractères demandés.

Déroulement :

1. Vérification du descripteur $\longrightarrow$ accès aux tables système.
2. Droits (mode adéquat)
3. Grâce à l'inode le système obtient les adresses du (des) bloc(s) contenant les données à lire. Le système effectue la lecture de ces blocs.
4. Le système recopie les données du buffer cache vers le tampon de l'utilisateur.
5. Le curseur dans le fichier est remit à jour dans l'entrée de la table des fichiers ouverts.
6. Le système renvoie le nombre de caractères effectivement lus.

5.4 `write`

```
int nbcecrits = write(int desc, char *tampon, int nbaecrire);
```

Même déroulement que `read` mais avec une allocation éventuelle de bloc-disque dans le cas d'un ajout au-delà de la fin du fichier.
Dans le cas où l'appel concerne un périphérique en mode caractère : le système active la fonction `write` (réciproquement `read` pour une lecture) du périphérique qui utilise directement l'adresse du tampon utilisateur.

Remarquons ici encore le polymorphisme de ces deux appels système qui permet de lire et d'écrire sur une grande variété de périphériques en utilisant une seule syntaxe. Le code C utilisant l'appel système marchera donc indifféremment sur tous les types de périphériques qui sont définis dans le système de fichier. Par exemple, il existe deux périphériques "logiques" qui sont /dev/null et /dev/zéro (que l'on ne trouve pas sur toutes les machines). Le premier est toujours vide en lecture et les écritures n'ont aucun effet (il est donc possible de déverser n'importe quoi sur ce périphérique). Le deuxième fournit en lecture une infinité de zéro et n'accepte pas l'écriture.

5.5 lseek

```
#include <fcntl.h>
off_t  lseek(int d, off_t offset, int direction)
```

`lseek` permet de déplacer le curseur de fichier dans la **table des fichiers ouverts** du système.
offset un déplacement en octets.
d le descripteur.
direction une des trois macros L_SET, L_INCR, L_XTND.

L_SET la nouvelle position est *offset* sauf si *offset* est supérieur à la taille du fichier, auquel cas la position est égale à la taille du fichier. Si l'offset est négatif, alors la position est zéro.

L_INCR la position courante est incrémentée de *offset* place (même contrainte sur la position maximum et la position minimum).

L_XTND Déplacement par rapport à la fin du fichier, cette option permet d'augmenter la taille du fichier (ne pas créer de fichiers virtuellement gros avec ce mécanisme, ils posent des problèmes de sauvegarde).

La valeur de retour de lseek est la nouvelle position du curseur dans le fichier ou -1 si l'appel a échoué.

5.6 dup et dup2

Les appels dup et dup2 permettent de dupliquer des entrées de la table des descripteurs du processus.

```
int  descripteur2 = dup(int descripteur1);
```

1. vérification que descripteur est le numéro d'une entrée non nulle.
2. recopie dans la **première entrée** libre du tableau des descripteurs l'entrée correspondant à descripteur1.
3. le compteur de descripteurs de l'entrée associée à descripteur1 dans la table des ouvertures de fichiers est incrémenté.
4. renvoi de l'indice dans la table des descripteurs de l'entrée nouvellement allouée.

Redirection temporaire de la sortie standard dans un fichier :

```
tempout = open("sortie_temporaire",1);
oldout = dup(1);
close(1);
newout = dup(tempout); /* renvoie 1  */
write(1,"xxxx",4); /* ecriture dans le fichier temporaire */
```

```
close(tempout);
close(1);
newout = dup(oldout);
close(oldout);
```

Il est aussi possible de choisir le descripteur cible avec

```
int ok = dup2(int source, int destination);
```

Recopie du descripteur `source` dans l'entrée `destination` de la table des descripteurs. Si `destination` désigne le descripteur d'un fichier ouvert, celui-ci est préalablement fermé avant duplication. Si `destination` n'est pas un numéro de descripteur valide, il y a une erreur, retour -1.

5.7 `close`

Fermeture d'un fichier.

```
int ok = close(descripteur);
```

1. si *descripteur* n'est pas un descripteur valide retour -1
2. l'entrée d'indice `descripteur` de la table est libérée.
3. Le compteur de l'entrée de la table des fichiers ouvert associé à `descripteur` est décrémenté.

   Si il passe à Zéro alors
4. l'entrée de la table des fichiers ouverts est libérée et le compteur des ouvertures de l'inode en mémoire est décrémenté.

   Si il passe à Zéro alors
5. l'entrée dans la table des inodes en mémoire est libérée.

   Si de plus le compteur de liens de l'inode est à 0 alors
6. le fichier est libéré : récupération de l'inode et des blocs.

Dans le cas d'une ouverture en écriture : le dernier bloc du buffer cache dans lequel on a écrit est marqué "a écrire".

5.8 `fcntl`

L'appel système `fnctl` permet de manipuler les ouverture de fichier après l'ouverture, bien sur il n'est pas possible de changer le mode d'ouverture (lecture/écriture/lecture-écriture) après l'ouverture.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int desc, int commande);
int fcntl(int desc, int commande, long arg);
int fcntl(int desc, int commande, struct flock *verrou);
```

L'appel système `fnctl` permet de positionner des verrous de fichier voire le chapitre 12. L'appel système `fnctl` permet la manipulation de certains des drapeaux d'ouverture :

O_APPEND

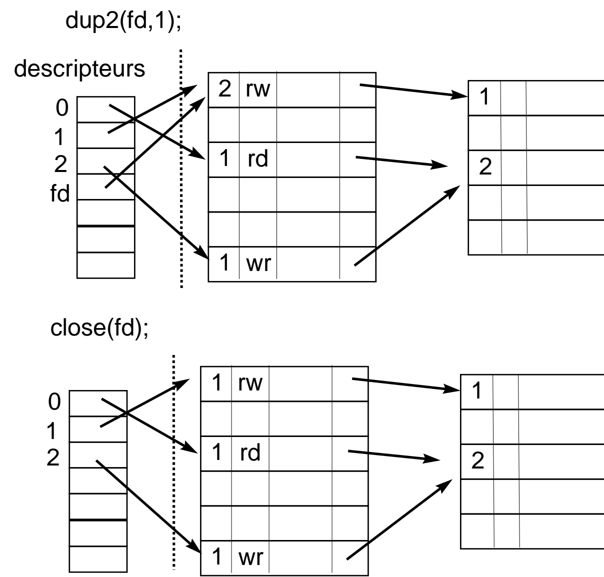

Fig. 5.3 – Redirection de la sortie standard sur "toto".

O_NONBLOCK

O_ASYNC

O_DIRECT

L'appel système `fnctl` permet gerer les signaux associés aux entrée asyncrones.

Chapitre 6

Les processus

6.1 Introduction aux processus

Un processus est un ensemble d'octets (en langage machine) en cours d'exécution, en d'autres termes, c'est l'exécution d'un programme.
Un processus UNIX se décompose en :

1. un espace d'adressage (visible par l'utilisateur/programmeur)
2. Le bloc de contrôle du processus (BCP) lui-même décomposé en :
   - une entrée dans la table des processus du noyau `struct proc` définie dans `<sys/proc.h>`.
   - une structure `struct user` appelée `zone u` définie dans `<sys/user.h>`

Les processus sous Unix apportent :

- La multiplicité des exécutions
  Plusieurs processus peuvent être l'exécution d'un même programme.

- La protection des exécutions
  Un processus ne peut exécuter que ses instructions propres et ce de façon séquentielle ; il ne peut pas exécuter des instructions appartenant à un autre processus.
  Les processus sous UNIX communiquent entre eux et avec le reste du monde grâce aux appels système.

6.1.1 Création d'un processus - `fork()`

Sous UNIX la création de processus est réalisée par l'appel système :

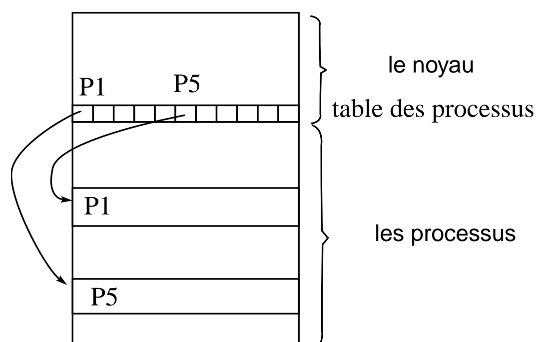

Fig. 6.1 – La table des processus est interne au noyau.

```
int fork(void);
```

Tous les processus sauf le processus d'identification 0, sont créés par un appel à *fork*.
Le processus qui appelle le fork est appelé processus *père*.
Le nouveau processus est appelé processus *fils*.

Tout processus a un seul processus père.
Tout processus peut avoir zéro ou plusieurs processus fils.

Chaque processus est identifié par un numéro unique, son **PID**.

Le processus de PID=0 est créé "manuellement" au démarrage de la machine, ce processus a toujours un rôle spécial[1] pour le système, de plus pour le bon fonctionement des programmes utilisant `fork()` il faut que le PID zéro reste toujours utilisé. Le processus zéro crée, grâce à un appel de fork, le processus init de PID=1.

Le processus de PID=1 de nom *init* est l'ancêtre de tous les autres processus (le processus 0 ne réalisant plus de `fork()`), c'est lui qui accueille tous les processus orphelins de père (ceci a fin de collecter les information à la mort de chaque processus).

6.2 Format d'un fichier exécutable

Les compilateurs nous permettent de créer des fichiers exécutables. Ces fichiers ont le format suivant qui permet au noyau de les transformer en processus :
- Une en-tête qui décrit l'ensemble du fichier, ses attributs et sa carte des sections.
- La taille à allouer pour les variables non initialisées.
- Une section TEXT qui contient le code (en langage machine)
- Une section données (DATA) codée en langage machine qui contient les données initialisées.

- Eventuellement d'autres sections : Table des symboles pour le débugeur, Images, ICONS, Table des chaînes, etc.

Pour plus d'informations se reporter au manuel a.out.h sur la machine.

6.3 Chargement/changement d'un exécutable

L'appel système `execve` change l'exécutable du processus courant en chargeant un nouvel exécutable. Les régions associée au processus sont préalablement libérées :

```
int execve(/* plusieurs formats */);
```

Pour chaque section de l'exécutable une région en mémoire est allouée.
Soit au moins les régions :
- le **code**
- les **données** initialisées

Mais aussi les régions :
- des **piles**
- du **tas**

La région de la **pile** :
C'est une pile de structures de pile qui sont empilées et dépilées lors de l'appel ou le retour de fonction. Le pointeur de pile, un des registres de l'unité centrale, indique la profondeur courante de la pile.

[1] swappeur,gestionnaire de pages

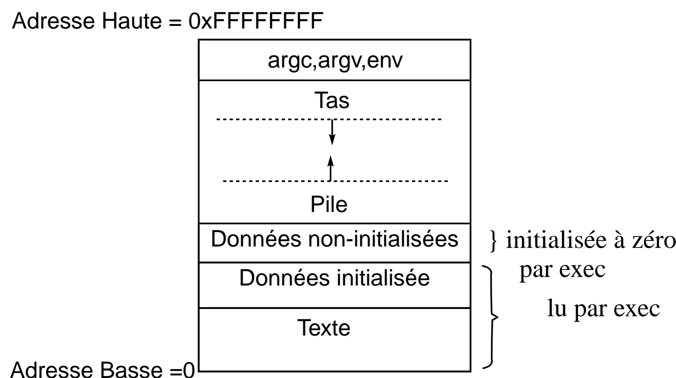

Fig. 6.2 – La structure interne des processus.

Le code du programme gère les extensions de pile (appel ou retour de fonction), c'est le noyau qui alloue l'espace nécessaire à ces extensions. Sur certains systèmes on trouve une fonction `alloca()` qui permet de faire des demandes de mémoire sur la pile.
Un processus UNIX pouvant s'exécuter en deux modes (noyau, utilisateur), une pile privée sera utilisée dans chaque mode.
La pile noyau sera vide quand le processus est en mode utilisateur.

Le **tas** est une zone où est réalisée l'allocation dynamique avec les fonctions `Xalloc()`.

6.4 zone u et table des processus

Tous les processus sont associés à une entrée dans la *table des processus* qui est interne au noyau. De plus, le noyau alloue pour chaque processus une structure appelée *zone u* , qui contient des données privées du processus, uniquement manipulables par le noyau. La *table des processus* nous permet d'accéder à la *table des régions par processus* qui permet d'accéder à la *table des régions*. Ce double niveau d'indirection permet de faire partager des régions.
Dans l'organisation avec une mémoire virtuelle, la table des régions est matérialisée logiquement dans la table de pages.

Les *structures de régions* de la table des régions contiennent des informations sur le type, les droits d'accès et la localisation (adresses en mémoire ou adresses sur disque) de la région.

Seule la **zone u** du processus courant est manipulable par le noyau, les autres sont **inaccessibles.** L'adresse de la zone u est placée dans le mot d'état du processus.

6.5 `fork` et `exec` (revisités)

Quand un processus réalise un `fork`, le contenu de l'entrée de la table des régions est dupliqué, chaque région est ensuite, en fonction de son type, partagée ou copiée.

Quand un processus réalise un `exec`, il y a libération des régions et réallocation de nouvelles régions en fonction des valeurs définies dans le nouvel exécutable.

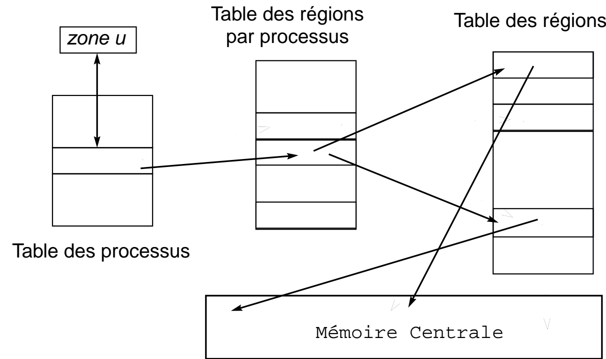

Fig. 6.3 – Table des régions, table des régions par processus

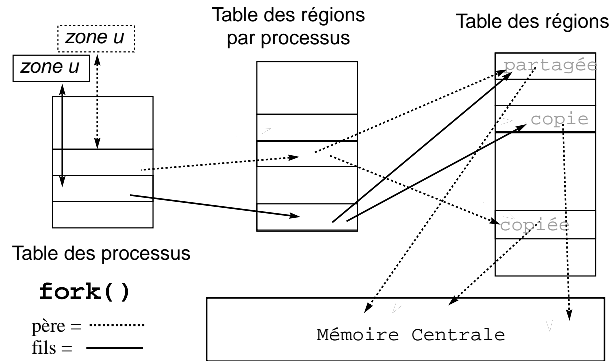

Fig. 6.4 – Changement de régions au cours d'un `fork`.

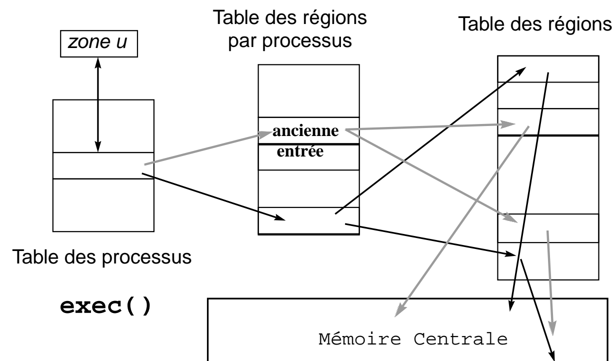

Fig. 6.5 – Changement de régions au cours d'un `exec`.

6.6 Le contexte d'un processus

Le contexte d'un processus est l'ensemble des données qui permettent de reprendre l'exécution d'un processus qui a été interrompu.

Le contexte d'un processus est l'ensemble de

1. son état
2. son mot d'état : en particulier
   – La valeur des registres actifs
   – Le compteur ordinal
3. les valeurs des variables globales statiques ou dynamiques
4. son entrée dans la table des processus
5. sa zone u
6. Les piles user et system
7. les zones de code et de données.

Le noyau et ses variables ne font partie du contexte d'aucun processus !

L'exécution d'un processus se fait dans son contexte.

Quand il y a changement de processus courant, il y a réalisation d'une **commutation de mot d'état** et d'un **changement de contexte**. Le noyau s'exécute alors dans le nouveau contexte.

6.7 Commutation de mot d'état et interruptions.

Ces fonctions de très bas niveau sont fondamentales pour pouvoir programmer un système d'exploitation.

Pour être exécuté et donner naissance à un processus, un programme et ses données doivent être chargés en mémoire centrale. Les instructions du programme sont transférées une à une de la mémoire centrale sur l'unité centrale où elles sont exécutées.

L'unité centrale :
Elle comprend des circuits logiques et arithmétiques qui effectuent les instructions mais aussi des mémoires appelées registres.

Certains de ces registres sont spécialisés directement par les constructeurs de l'unité centrale, d'autres le sont par le programmeur du noyau. Quelques registres spécialisés :

L'accumulateur qui reçoit le résultat d'une instruction ; sur les machines à registres multiples, le jeu d'instructions permet souvent d'utiliser n'importe lequel des registres comme accumulateur.

le registre d'instruction (qui contient l'instruction en cours)

le compteur ordinal (adresse de l'instruction en mémoire) Ce compteur change au cours de la réalisation d'une instruction pour pointer sur la prochaine instruction à exécuter, la majorité des instructions ne font qu'incrémenter ce compteur, les instructions de branchement réalisent des opérations plus complexes sur ce compteur : affectation, incrémentation ou décrémentation plus importantes.

le registre d'adresse

les registres de données qui sont utilisés pour lire ou écrire une donnée à une adresse spécifiée en mémoire.

les registres d'état du processeur : (actif, mode (user/system), retenue, vecteur d'interruptions, etc)

les registres d'état du processus droits, adresses, priorités, etc

Ces registres forment le contexte **d'unité centrale d'un processus**. A tout moment, un processus est caractérisé par ces deux contextes : le contexte d'unité centrale qui est composé des mêmes données pour tous les processus et le contexte qui dépend du code du programme exécuté. Pour

	Nature de l'interruption	fonction de traitement
0	horloge	clockintr
1	disques	diskintr
2	console	ttyintr
3	autres peripheriques	devintr
4	appel system	sottintr
5	autre interruption	otherintr

Fig. 6.6 – Sous UNIX, on trouvera en général 6 niveaux d'interruption

pouvoir exécuter un nouveau processus, il faut pouvoir **sauvegarder** le contexte d'unité centrale du processus courant (mot d'état), puis charger le nouveau mot d'état du processus à exécuter. Cette opération délicate réalisée de façon matérielle est appelée **commutation de mot d'état.** Elle doit se faire de façon non interruptible ! Cette "Super instruction" utilise 2 adresses qui sont respectivement :
l'adresse de sauvegarde du mot d'état
l'adresse de lecture du nouveau mot d'état

Le compteur ordinal faisant partie du mot d'état, ce changement provoque l'exécution dans le nouveau processus.

C'est le nouveau processus qui devra réaliser la sauvegarde du contexte global. En général c'est le noyau qui réalise cette sauvegarde, le noyau n'ayant pas un contexte du même type.

Le processus interrompu pourra ainsi reprendre exactement où il avait abandonné.

Les fonctions `setjmp/longjmp` permettent de sauvegarder et de réinitialiser le contexte d'unité central du processus courant, en particulier le pointeur de pile.

6.8 Les interruptions

Une interruption est une commutation de mot d'état provoquée par un signal produit par le matériel.
Ce signal étant la conséquence d'un événement extérieur ou intérieur, il modifie l'état d'un indicateur qui est régulièrement testé par l'unité centrale.
Une fois que le signal est détecté, il faut déterminer la cause de l'interruption. Pour cela on utilise un indicateur, pour les différentes causes, On parle alors du **vecteur d'interruptions**.
Trois grands types d'interruptions :

- **externes** (indépendantes du processus) interventions de l'opérateur, pannes,etc
- **déroutements** erreur interne du processeur, débordement, division par zéro, page fault etc (causes qui entraine la réalisation d'une sauvegarde sur disque de l'image mémoire "core dumped" en général)
- **appels systèmes** demande d'entrée-sortie par exemple.

Suivant les machines et les systèmes un nombre variable de niveaux d'interruption est utilisé.

Ces différentes interruptions ne réalisent pas nécessairement un changement de contexte complet du processus courant.

Il est possible que plusieurs niveaux d'interruption soient positionnés quand le système les consulte. C'est le niveau des différentes interruptions qui va permettre au système de sélectionner l'interruption à traiter en priorité.

L'horloge est l'interruption la plus prioritaire sur un système Unix.

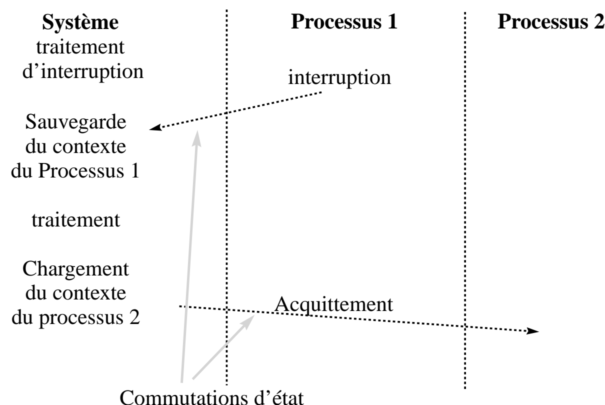

FIG. 6.7 – Le traitement d'une interruption.

6.9 Le problème des cascades d'interruptions

Si pendant le traitement d'une interruption, une autre interruption se produit, et que ceci se répète pendant le traitement de la nouvelle interruption, le système ne fait plus progresser les processus ni les interruptions en cours de traitement ...

Il est donc nécessaire de pouvoir retarder ou annuler la prise en compte d'un ou plusieurs signaux d'interruptions. C'est le rôle des deux mécanismes de **masquage** et de **désarmement** d'un niveau d'interruption. Masquer, c'est ignorer temporairement un niveau d'interruption.

Si ce masquage est fait dans le mot d'état d'un traitement d'interruption, à la nouvelle commutation d'état, le masquage disparaît ; les interruptions peuvent de nouveau être prises en compte. Désarmer, c'est rendre le positionnement de l'interruption caduque. (Il est clair que ceci ne peut s'appliquer aux déroutements).

6.9.1 Etats et transitions d'un processus

Nous nous plaçons dans le cas d'un système qui utilise un mécanisme de swap pour gérer la mémoire ; nous étudierons ensuite le cas des systèmes de gestion paginée de la mémoire (les couples d'états 3,5 et 4,6 y sont fusionnés).

6.9.2 Listes des états d'un processus

1. le processus s'exécute en mode utilisateur
2. le processus s'exécute en mode noyau
3. le processus ne s'exécute pas mais est éligible (prêt à s'exécuter)
4. le processus est endormi en mémoire centrale
5. le processus est prêt mais le swappeur doit le transférer en mémoire centrale pour le rendre éligible. (ce mode est différent dans un système à pagination).
6. le processus est endormi en zone de swap (sur disque par exemple).
7. le processus passe du mode noyau au mode utilisateur mais est préempté[2] et a effectué un changement de contexte pour élire un autre processus.
8. naissance d'un processus, ce processus n'est pas encore prêt et n'est pas endormi, c'est l'état initial de tous processus sauf le *swappeur*.

[2] Bien que le processus soit prêt, il est retiré de l'unité de traitement pour que les autres processus puissent avancer.

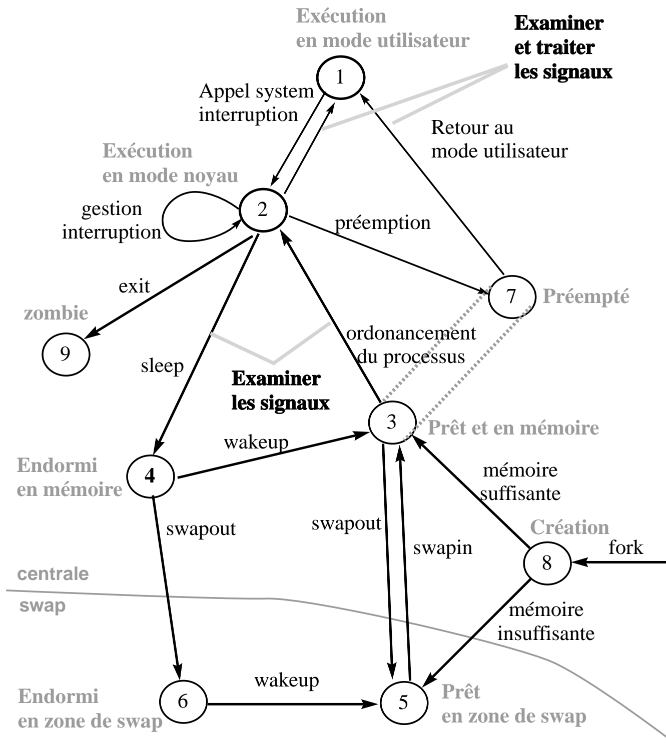

FIG. 6.8 – Diagramme d'état des processus

9. *zombie* le processus vient de réaliser un exit, il apparaît uniquement dans la table des processus où il est conservé le temps pour son processus père de récupèrer le code de retour et d'autres informations de gestion (coût de l'exécution sous forme de temps, et d'utilisation des ressources).

L'état *zombie* est l'état final des processus, les processus restent dans cet état jusqu'à ce que leur père lise leur valeur de retour (exit status).

6.10 Lecture du diagramme d'état.

Le diagramme des transitions d'état permet de décrire l'ensemble des états possibles d'un processus. Il est clair que tout processus ne passera pas nécessairement par tous ces différents états.

La naissance d'un processus a lieu dans l'état 8 après l'appel système `fork` exécuté par un autre processus. Il devient au bout d'un certain temps "prêt à s'exécuter". Il passe alors dans l'état "exécuté en mode noyau" où il termine sa partie de l'appel système `fork`. Puis le processus termine l'appel système et passe dans l'état "exécuté en mode utilisateur". Passé une certaine période de temps (variable d'un système à l'autre), l'horloge peut *interrompre* le processeur. Le processus rentre alors en mode noyau, l'interruption est alors réalisée avec le processus en mode noyau.

Au retour de l'interruption, le processus peut être `préempté` (étant resté tout son quantum de temps sur le cpu), c'est à dire, il reste prêt à s'exécuter mais un autre processus est élu. Cet état 7 est logiquement équivalent à l'état 3, mais il existe pour matérialiser le fait qu'un processus ne peut être préempté qu'au moment où il retourne du mode noyau au mode utilisateur. Quand un processus préempté est réélu, il retourne directement en mode utilisateur.

Un appel système ne peut être préempté. On peut détecter en pratique cette règle, en effet

on constate un ralentissement du débit de la machine pendant la réalisation d'un core de grande taille.

Quand un processus exécute un appel système, il passe du mode utilisateur au mode système. Supposons que l'appel système réalise une entrée-sortie sur le disque et que le processus doive attendre la fin de l'entrée-sortie. Le processus est mis en sommeil (sleep) et passe dans l'état endormi en mémoire. Quand l'entrée-sortie se termine, une interruption a lieu, le traitement de l'interruption consistant à faire passer le processus dans le mode prêt à s'exécuter (en mémoire).

6.11 Un exemple d'exécution

Plaçons-nous dans la situation suivante : l'ensemble de la mémoire est occupé par des processus, mais, le processus le plus prioritaire est un processus dans l'état 5, soit : "prêt à s'exécuter en zone de swap". Pour pouvoir exécuter ce processus, il faut le placer dans l'état 3, soit : "prêt à s'exécuter en mémoire". Pour cela le système doit libérer de la mémoire (faire de la place), en faisant passer des processus des états 3 ou 4 en zone de swap (swapout) donc les faire passer dans les états 5 et 6.
C'est au swappeur de réaliser les deux opérations :

- Sélectionner une victime (le processus le plus approprié), pour un transfert hors mémoire centrale (swapout).
- réaliser ce transfert.
- une fois qu'une place suffisante est libérée, le processus qui a provoqué le swapout est chargé en mémoire (swapin).

Le processus a un contrôle sur un nombre réduit de transitions : il peut faire un appel système, réaliser un `exit`, réaliser un `sleep`, les autres transitions lui sont dictées par les circonstances.

L'appel à `exit()` fait passer dans l'état zombie, il est possible de passer à l'état zombie sans que le processus ait explicitement appelé `exit()` (à la réception de certains signaux par exemple). Toutes les autres transitions d'état sont sélectionnées et réalisées par le noyau selon des règles bien précises. Une de ces règles est par exemple qu'un processus en mode noyau ne peut être préempté[3]. Certaines de ces règles sont définies par l'algorithme d'ordonnancement utilisé.

6.12 La table des processus

La table des processus est dans la mémoire du noyau. C'est un tableau de structure proc (`<sys/proc.h>`). Cette structure contient les informations qui doivent toujours être accessibles par le noyau.

état se reporter au diagramme, ce champ permet au noyau de prendre des décisions sur les changements d'état à effectuer sur le processus.

adresse de la zone u

adresses taille et localisation en mémoire (centrale, secondaire). Ces informations permettent de transférer un processus en ou hors mémoire centrale.

UID propriétaire du processus, permet de savoir si le processus est autorisé à envoyer des signaux et à qui il peut les envoyer.

PID,PPID l'identificateur du processus et de son père. Ces deux valeurs sont initialisées dans l'état 8, création pendant l'appel système `fork`.

évènement un descripteur de l'évènement attendu quand le processus est dans un mode endormi.

Priorités Plusieurs paramètres sont utilisés par l'ordonnanceur pour sélectionner l'élu parmi les processus prêts.

vecteur d'interruption du processus ensemble des signaux reçus par le processus mais pas encore traités.

[3]Exercice : Donner un exemple.

divers des compteurs utilisés pour la comptabilité (pour faire payer le temps CPU utilisé) et que l'on peut manipuler par la commande `alarm`, des données utilisées par l'implémentation effective du système, etc.

6.13 La zone u

La zone u de type `struct user` définie dans `<sys/user.h>` est la zone utilisée quand un processus s'exécute que ce soit en mode noyau ou mode utilisateur. Une unique zone u est accessible à la fois : celle de l'unique processus en cours d'exécution (dans un des états 1 ou 2).

Contenu de la zone u :

pointeur sur la structure de processus de la table des processus.

uid réel et effectif de l'utilisateur qui détermine les divers privilèges donnés au processus, tels que les droits d'accès à un fichier, les changements de priorité, etc.

Compteurs des temps (users et system) consommés par le processus

Masque de signaux Sur système V sous BSD dans la structure proc

Terminal terminal de contrôle du processus si celui-ci existe.

erreur stockage de la dernière erreur rencontrée pendant un appel système.

retour stockage de valeur de retour du dernier appel système.

E/S les structures associées aux entrées-sorties, les paramètres utilisés par la bibliothèque standard, adresses des buffers, tailles et adresses de zones à copier, etc.

"." et "/" le répertoire courant et la racine courante (c.f. chroot())

la table des descripteurs position variable d'un implémentation à l'autre.

limites de la taille des fichiers de la mémoire utilisable etc $\frac{1}{4}$(c.f. *ulimit* en Bourne shell et *limit* en Csh).

umask masque de création de fichiers.

6.14 Accès aux structures `proc` et `user` du processus courant

Les informations de la table des processus peuvent être lues grâce à la commande shell `ps`. Ou par des appels système. Par contre, les informations contenues dans la zone u ne sont accessibles que par une réponse du processus lui-même (en progammation objet, on dit que ce sont des variables d'instances privées), d'où les appels système suivants :
`times, chroot, chdir, fchdir, getuid, getgid, ..., setuid, ..., ulimit, nice, brk, sbrk`.
Qui permettent de lire ou de changer le contenu des deux structures.

6.14.1 Les informations temporelles.

```
#include <sys/times.h>
clock_t times(struct tms *buffer);
```

times remplit la structure pointée par `buffer` avec des informations sur le temps machine utilisé dans les état 1 et 2.
La structure**?** :

```
struct tms {
       clock_t    tms_utime;      /* user time */
       clock_t    tms_stime;      /* system time */
```

```
    clock_t    tms_cutime;     /* user time, children */
    clock_t    tms_cstime;     /* system time, children */
  };
```

contient des temps indiqués en microsecondes 10-6 secondes, la précision de l'horloge est par defaut sur les HP9000 700/800 de 10 microsecondes.

6.14.2 Changement du répertoire racine pour un processus.

```
#include <unistd.h>
int chroot(const char *path);
```

permet de définir un nouveau point de départ pour les références absolues (commençant par /). La référence .. de ce répertoire racine est associée à lui-même, il n'est donc pas possible de sortir du sous-arbre défini par `chroot`. Cet appel est utilisé pour rsh et ftp, et les comptes pour invités.

Les appels suivants permettent de changer le répertoire de travail de référence "." et donc l'interprétation des références relatives :

```
int chdir(char *ref);
int fchdir(int descripteur);
```

6.14.3 Récupération du PID d'un processus

```
#include <unistd.h>
pid_t   getpid(void);
pid_t   getpgrp(void);
pid_t   getppid(void);
pid_t   getpgrp2(pid_t pid);
```

L'appel `getpid()` retourne le PID du processus courant, `getppid` le PID du processus père, `getpgrp` le PID du groupe du processus courant, `getpgrp2` le PID du groupe du processus pid (si pid=0 alors équivalent à `getpgrp`).

6.14.4 Positionement de l'euid, ruid et suid

L'uid d'un processus est l'identification de l'utilisateur exécutant le processus. Le système utilise trois uid qui sont :

euid uid effective utilisé pour les tests d'accès.

ruid uid réelle, uid à qui est facturé le temps de calcul.

suid uid sauvegardée, pour pouvoir revenir en arrière après un setuid.

```
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

Fonctionnement :
si euid == 0 (euid de root) les trois uid sont positionnés à la valeur de uid
sinon si uid est égal à ruid ou suid alors euid devient uid. ruid et suid ne changent pas. sinon rien ! pas de changements.
Syntaxe identique pour setgid et gid.

La commande *setreuid()* permet de changer le propiétaire réel du processus, elle est utilisé pendant le login, seul le super utilisateur peut l'exécuter avec succès.

6.15 Tailles limites d'un processus

```
#include <ulimit.h>
long ulimit(int cmd,...);
```

La commande *cmd* est

UL_GETFSIZE retourne le taille maximum des fichiers en blocs.

UL_SETFSIZE positionne cette valeur avec le deuxième argument.

UL_GETMAXBRK valeur maximale pour l'appel d'allocation dynamique de mémoire : brk.

Ces valeurs sont héritées du processus père.

La valeur FSIZE (taille maximum des fichiers sur disques en blocs) peut être changée en ksh avec ulimit [n].

6.15.1 Manipulation de la taille d'un processus.

```
#include <unistd.h>
int brk(const void *endds);
void *sbrk(int incr);
```

Les deux appels permettent de changer la taille du processus. L'adresse manipulée par les deux appels est la première adresse qui est en dehors du processus. Ainsi on réalise des augmentations de la taille du processus avec des appels à `sbrk` et on utilise les adresses retournées par `sbrk` pour les appels à `brk` pour réduire la taille du processus. On utilisera de préférence pour les appels à `sbrk` des valeurs de `incr` qui sont des multiples de la taille de page. Le système réalisant des déplacement du point de rupture par nombre entier de pages (ce qui est logique dans un système de mémoire paginé). A ne pas utiliser en conjonction avec les fonctions d'allocation standard `malloc, calloc, realloc, free`.

6.15.2 Manipulation de la valeur nice

Permet de changer la valeur de *nice* utilisée par le processus. Si l'on a des droits privilégiés la valeur peut être négative. La valeur de nice est toujours comprise entre -20 et +20 sous linux. Seul le super utilisateur pouvant utiliser une valeur négative.

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

La commande shell `renice priorite -p pid -g pgrp -u user` permet de changer le nice d'un processus actif.

6.15.3 Manipulation de la valeur umask

L'appel `umask` permet de spécifier quels droits doivent être interdits en cas de création de fichier. cf. 5.1

```
#include <sys/stat.h>
mode_t umask(mode_t mask);
```

la valeur retournée est l'ancienne valeur.

6.16 L'appel système `fork`

l'appel système `fork` permet le création d'un processus clône du processus courrant.

```
pid_t  fork(void);
```

DEUX valeurs de retour en cas de succès :
- Dans le processus père valeur de retour = le PID du fils,
- Dans le processus fils valeur de retour = zéro.

Sinon
- Dans le processus père valeur de retour = -1.

Les `PID` et `PPID` sont les seules informations différentes entre les deux processus.

6.17 L'appel système `exec`

```
#include <unistd.h>
extern char **environ;

int execl( const char *path, const char *arg0, ...,NULL);
int execv(const char *path, char * const argv[]);
int execle( const char *path, const char *arg0, ...,NULL,  char * const envp[]);

int execve(const char *file, char * const argv[], char * const envp[]);
int execlp( const char *file,const char *arg0, ... , NULL );
int execvp(const char *file, char * const argv[]);
```

Informations conservées par le processus : PID PPID PGID ruid suid (pour l'euid cf le `setuidbit` de `chmod`), nice, groupe d'accès, catalogue courant, catalogue "/", terminal de contrôle, utilisation et limites des ressources (temps machine, mémoire, etc), umask, masques des signaux, signaux en attente, table des descripteurs de fichiers, verrous, session.

Quand le processus exécute dans le nouvel exécutable la fonction :

```
 main(int argc, char **argv,char **envp)
```

argv et *env* sont ceux qui ont été utilisés dans l'appel de `execve`.

Les différents noms des fonction `exec` sont des mnémoniques :

`l` liste d'arguments

`v` arguments sont forme d'un vecteur.

`p` recherche du fichier avec la variable d'environnement `PATH`.

`e` transmission d'un environnement en dernier paramètre, en remplacement de l'environnement courant.

Chapitre 7

L'ordonnancement des processus

La sélection dans le temps des processus pouvant accèder à une ressource est un problème dit d'ordonnancement. Nous présentons ici :

- le cas général
- les besoins et les problèmes

et nous décrirons des solutions que l'on trouve sous UNIX pour différents problèmes d'ordonnancement.

Les algorithmes d'ordonnancement réalisent la sélection parmi les processus actifs de celui qui va obtenir l'utilisation d'une ressource, que ce soit l'unité centrale, ou bien un périphérique d'entrée-sortie.

Pour l'unité centrale notre but est de maximiser débit et taux utile de l'unité centrale :

le débit est le nombre moyen de processus exécutés en un temps donné.

le taux utile est la proportion de temps réellement utilisée pour exécuter des processus utilisateurs.

Un exemple :
Soient 2 processus A et B de même comportement 30 périodes de deux seconde :
1 seconde d'activité
1 seconde d'inactivité

```
AIAIAIAIAIAIAIAIAIAIAIAIAIAIAIAIAIAI
```

Si l'on exécute les deux processus consécutivement on obtient un débit de 1 processus par minute, et un taux utile de 50%. Si l'on entrelace les périodes actives et inactives des deux processus on obtient un débit de 2 processus par minute et un taux d'utilisation de 100%.

Pour une autre ressource d'autres critères seront utilisés.

7.1 Le partage de l'unité centrale

Ce partage doit être fait non seulement entre les processus utilisateurs mais aussi entre les différentes tâches du système, scheduler, entrées-sorties, gestion des interruptions, etc.

Nous demandons de plus à l'algorithme d'ordonnancement de nous assurer **l'exclusion mutuelle** et **l'absence de famine**, qui sont les points-clefs de la plupart des problèmes d'ordonnancement. L'invention d'un algorithme d'ordonnancement se base en générale sur des remarques statistique sur le comportement des processus :

- Le couple UC/ES (cpu/io), les processus ont tendance à basculer constamment entre des phases d'entrées-sorties et des phases de calcul sur l'unité centrale.
- Les processus consommant de longues périodes d'U.C. sont proportionnellement rares.

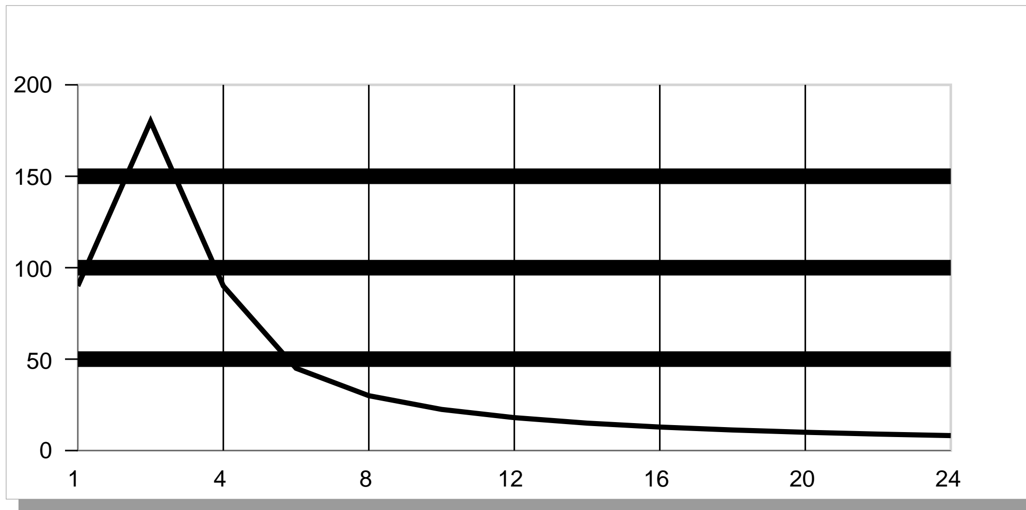

FIG. 7.1 – Histogramme de répartition de la durée de la période d'utilisation de l'unité centrale

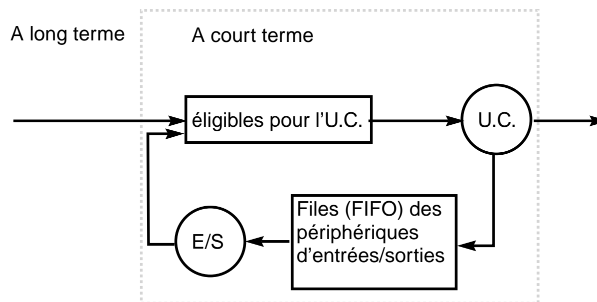

FIG. 7.2 – Stratégie globale d'ordonnancement.

7.1.1 Famine

Notre première tâche est d'affecter une ressource (l'UC par exemple) à un unique processus à la fois (exclusion mutuelle) et s'assurer de l'absence de famine.
famine : un processus peut se voir refuser l'accès à une ressource pendant un temps indéterminé, il est dit alors que le processus est en famine.

Un système qui ne crée pas de cas de famine : fournira toujours la ressource demandée par un processus, au bout d'un temps fini.

Si on prend le cas des périphériques (tels que les disques) l'ordonnancement peut se faire de façon simple avec par exemple une file d'attente (FIFO).

Pour l'unité centrale on va devoir utiliser des structures de données plus complexes car nous allons avoir besoin de gérer des priorités. C'est par exemple, autoriser l'existence de processus qui évitent la file d'attente. La structure de données utilisée peut parfaitement être une file, une liste, un arbre ou un tas, ceci en fonction de l'élément-clef de notre algorithme de sélection (âge, priorité simple, priorité à plusieurs niveaux, etc).

Cette structure de données doit nous permettre d'accéder à tous les processus prêts (éligibles).

7.1.2 Stratégie globale

On peut représenter l'ordonnancement global avec le schéma 7.2

Les ordonnancements à court terme doivent être très rapides, en effet le processus élu ne va utiliser l'unité centrale que pendant un très court laps de temps (10 milli-secondes par exemple).

Si on utilise trop de temps (1 milli-seconde) pour sélectionner cet élu, le taux utile décroît très rapidement (ici on perd 9% du temps d'unité centrale).

Par contre l'ordonnancement à long terme peut être plus long car il a lieu moins souvent (toutes les secondes par exemple). La conception de l'ordonnanceur à long terme est faite dans l'optique d'obtenir un ordonnanceur à court terme rapide.

7.1.3 Critères de performance

Les critères de performance des algorithmes d'ordonnancement

- Taux d'utilisation de l'unité centrale
- Débit
- Temps réel d'exécution
- Temps d'attente
- Temps de réponse

Ces cinq critères sont plus ou moins mutuellement exclusifs.

Les comparaisons des différents algorithmes se fait donc sur une sélection de ces critères.

7.2 Ordonnancement sans préemption.

- FCFS : First Come First served
  Facile à écrire et à comprendre, peu efficace ...
- SJF : Shortest Job First
  le plus petit en premier.
  Optimal pour le temps d'attente moyen ...
- A priorité :
  L'utilisateur donne des priorités aux différents processus et ils sont activés en fonction de cette priorité.

  problème ⟶ famine possible des processus peu prioritaires

  Solution ⟶ faire augmenter la priorité avec le temps d'attente :
  plus un processus attend, plus sa priorité augmente ainsi au bout d'un certain temps le processus devient nécessairement le plus prioritaire.
  re-problème ⟶ si le processus en question (le très vieux très gros) est exécuté alors que de nombreux utilisateurs sont en mode interactif chute catastrophique du temps de réponse et du débit

  solution ⟶ préemption.

La **préemption** est la possibilité qu'a le système de reprendre une ressource à un processus sans que celui-ci ait libéré cette ressource.
Ceci est impossible sur bon nombre de ressources. Lesquelles ?

7.3 Les algorithmes préemptifs

FCFS ne peut être préemptif ...
SJF peut être préemptif : si un processus plus court que le processus actif arrive dans la queue, le processus actif est préempté.

Dans des systèmes interactifs en temps partagé un des critères est le temps de réponse, c'est à dire que chaque utilisateur dispose de l'unité centrale régulièrement. Heureusement, les processus interactifs utilisent l'UC pendant de très courts intervalles à chaque fois.

7.3.1 Round Robin (tourniquet)

Cet algorithme est spécialement adapté aux systèmes en temps partagé.
On définit un **quantum de temps** (time quantum) d'utilisation de l'unité centrale.
La file d'attente des processus éligibles est vue comme une queue circulaire (fifo circulaire).
Tout nouveau processus est placé à la fin de la liste.

De deux choses l'une, soit le processus actif rend l'Unité Centrale avant la fin de sa tranche de temps (pour cause d'entrée/sortie) soit il est préempté, et dans les deux cas placé en fin de liste.

Un processus obtiendra le processeur au bout de (n -1)*q secondes au plus (n nombre de processus et q longueur du quantum de temps), la famine est donc assurément évitée.

Remarquons que si le quantum de temps est trop grand, round-robin devient équivalent à FCFS. De l'autre coté si le quantum de temps est très court, nous avons théoriquement un processeur n fois moins rapide pour chaque processus (n nombre de processus).

Malheureusement si le quantum de temps est court, le nombre de changements de contexte dûs à la préemption grandit, d'où une diminution du taux utile, d'où un processeur virtuel très lent.

Une règle empirique est d'utiliser un quantum de temps tel que 80 pourcent des processus interrompent naturellement leur utilisation de l'unité centrale avant l'expiration du quantum de temps.

7.3.2 Les algorithmes à queues multiples

Nous supposons que nous avons un moyen de différencier facilement les processus en plusieurs classes de priorité différentes (c'est le cas sous UNIX où nous allons différencier les tâches système, comme le swappeur, des autres tâches).

Pour sélectionner un processus, le scheduler parcourt successivement les queues dans l'ordre décroissant des priorités.

Un exemple de queues organisées en fonction du contenu des processus :
- les processus systèmes
- les processus interactifs
- les processus édition
- les processus gros calcul
- les processus des étudiants

pour qu'un processus étudiant soit exécuté il faut que toutes les autres files d'attente soient vides ...

Une autre possibilité est de partager les quantums de temps sur les différentes queues.

Il est aussi possible de réaliser différents algorithmes de scheduling sur les différentes queues :
- Round Robin sur les processus interactifs
- FCFS sur les gros calculs en tâche de fond.

7.4 Multi-level-feedback round robin Queues

Le système d'ordonnancement des processus sous UNIX (BSD 4.3 et system V4) utilise plusieurs files d'attente qui vont matérialiser des niveaux de priorité différents et à l'intérieur de ces différents niveaux de priorité, un système de tourniquet.

7.4.1 Les niveaux de priorité

Le scheduler parcourt les listes une par une de haut en bas jusqu'à trouver une liste contenant un processus éligible. Ainsi tant qu'il y a des processus de catégorie supérieure à exécuter les autres processus sont en attente de l'unité centrale.

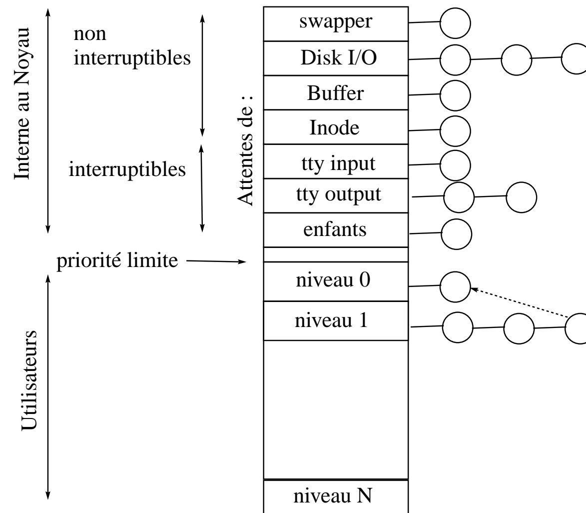

FIG. 7.3 – Les queues multiples en tourniquet

Dans les listes internes au noyau, de simples files d'attente sont utilisées avec la possibilité de doubler les processus endormis de la même liste (en effet seul le processus réveillé par la fin de son entrée/sortie est éligible).

Pour les processus utilisateurs, la même règle est utilisée mais avec préemption et la règle du tourniquet.

C'est à dire, on calcul une priorité de base qui est utilisée pour placer le processus dans la bonne file d'attente.

Un processus qui utilise l'unité centrale voit augmenter sa priorité.
Un processus qui libère l'unité centrale pour demander une entrée/sortie ne voit pas sa priorité changer.
Un processus qui utilise tout sont quantum de temps est préempté et placé dans une nouvelle file d'attente.

Attention : plus la priorité est grande moins le processus est prioritaire.

7.4.2 Evolution de la priorité

Regardons la priorité et l'évolution de la priorité d'un processus utilisateur au cours du temps. Les fonctions suivantes sont utilisées dans une implémentation BSD.

Pour calculer la priorité d'un processus utilisateur, le scheduler utilise l'équation suivante qui est calculée tous les 4 clicks horloge (valeur pratique empirique) :

$$P_{\text{usrpri}} = \text{PUSER} + \frac{P_{\text{cpu}}}{4} + 2 \times P_{\text{nice}}$$

cette valeur est tronquée à l'intervalle PUSER..127. En fonction de cette valeur le processus est placé dans une des listes correspondant à son niveau courant de priorité.

Ceci nous donne une priorité qui diminue linéairement en fonction de l'utilisation de l'unité centrale (il advient donc un moment où le processus devient le processus le plus prioritaire !).

P_{nice} est une valeur spécifiée par le programmeur grâce à l'appel système `nice`. Elle varie entre -20 et +20 et seul le super utilisateur peut spécifier une valeur négative.

P_{cpu} donne une estimation du temps passé par un processus sur l'unité centrale. A chaque click d'horloge, la variable `p_cpu` du processus actif est incrémentée. Ce qui permet de matérialiser la

consommation d'unité central du processus. Pour que cette valeur ne devienne pas trop pénalisante sur le long terme (comme pour un shell) elle est atténuée toute les secondes grâce à la formule suivante :

$$P_{\text{cpu}} = \frac{2 \times \text{load}}{2 \times \text{load} + 1} \times P_{\text{cpu}} + P_{\text{nice}}$$

la valeur de `load` (la charge) est calculée sur une moyenne du nombre de processus actifs pendant une minute.

Pour ne pas utiliser trop de ressources, les processus qui sont en sommeil (sleep) voient leur P_{cpu} recalculé uniquement à la fin de leur période de sommeil grâce à la formule :

$$P_{\text{cpu}} = \left(\frac{2 \times \text{load}}{2 \times \text{load} + 1}\right)^{\text{sleep_time}} \times P_{\text{cpu}}$$

la variable `sleep_time` étant initialisée à zéro puis incrémentée une fois par seconde.

7.4.3 Les classes de priorité

La priorité des processus en mode système dépend de l'action à réaliser.

PSWAP 0 priorité en cours de swap
PINOD 10 priorité en attendant une lecture d'information sur le système de fichiers
PRIBIO 20 priorité en attente d'une lecture/écriture sur disque
PZERO 25 priorité limite
PWAIT 30 priorité d'attente de base
PLOCK 35 priorité d'attente sur un verrou
PSLEP 40 priorité d'attente d'un évènement
PUSER 50 priorité de base pour les processus en mode utilisateur

Le choix de l'ordre de ces priorités est très important, en effet un mauvais choix peut entraîner une diminution importante des performances du système.

Il vaut mieux que les processus en attente d'un disque soient plus prioritaires que les processus en attente d'un buffer, car les premiers risquent fort de libérer un buffer après leur accès disque (de plus il est possible que ce soit exactement le buffer attendu par le deuxième processus). Si la priorité était inverse, il deviendrait possible d'avoir un interblocage ou une attente très longue si le système est bloqué par ailleurs.

De la même façons, le swappeur doit être le plus prioritaire et non interruptible $\longrightarrow$ Si un processus est plus prioritaire que le swappeur et qu'il doit être swappé en mémoire ...
En Demand-Paging le swappeur est aussi le processus qui réalise les chargements de page, ce processus doit être le plus prioritaire.

Chapitre 8

La mémoire

8.0.4 les mémoires

La mémoire d'un ordinateur se décompose en plusieurs éléments, dont le prix et le temps d'accès sont très variables, cf figure 8.1. Nous développerons dans ce chapitre et le suivant les questions et solutions relatives à la mémoire centrale.

L'importance de la gestion de la mémoire centrale vient de son coût et du coût relatif des autres formes de stockage, la figure 8.2 donne une idée des caractéristiques relatives des différents types de stockage.

8.0.5 La mémoire centrale

La mémoire est un tableau à une dimension de mots machines (ou d'octets), chacun ayant une adresse propre. Les échanges avec l'extérieur se font en général par des lectures ou des écritures à des adresses spécifiques.

Le système Unix est multi-tâche,ceci pour maximiser l'utilisation du cpu. Cette technique pose comme condition obligatoire que la mémoire centrale soit utilisée et/ou partagée entre les différentes tâches.

Les solutions de gestion de la mémoire sont très dépendantes du matériel et ont mis longtemps à évoluer vers les solutions actuelles. Nous allons voir plusieurs approches qui peuvent servir dans des situations particulières .

La mémoire est le point central dans un système d'exploitation, c'est à travers elle que l'unité centrale communique avec l'extérieur.

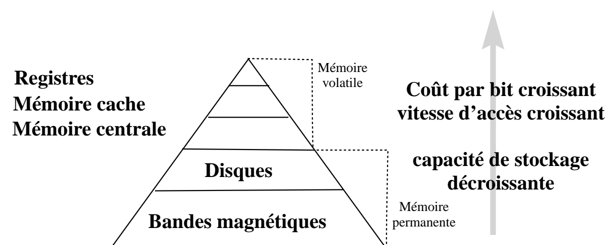

Fig. 8.1 – Hiérarchie de mémoires

CARACTERISTIQUES DES TYPES DE MEMOIRES

TYPE DE MEMOIRE	TAILLE (Octets)	TEMPS D'ACCES (secondes)	COUT RELATIF PAR BIT
CACHE	10^{3}-10^{4}	10^{-8}	10
MEMOIRE CENTRALE	10^{6}-10^{7}	10^{-7}	1
DISQUE	10^{8}-10^{9}	10^{-3}-10^{-2}	10^{-2}-10^{-3}
BANDE	10^{8}-10^{9}	10-10^{2}	10^{-4}

Fig. 8.2 – Caractéristiques relatives des mémoires.

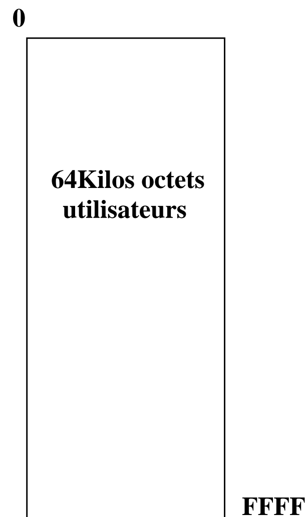

Fig. 8.3 – Une mémoire de 64 Kilo Octets.

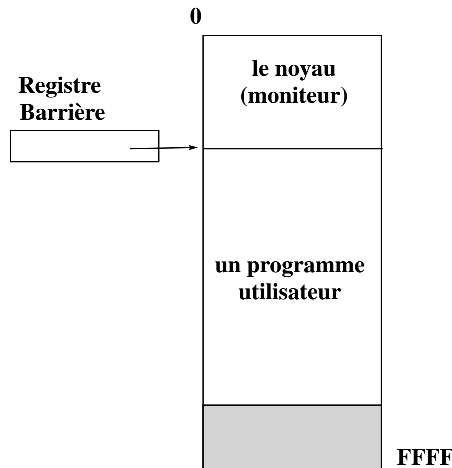

Fig. 8.4 – Protection du moniteur par un registre barrière.

8.1 Allocation contiguë

8.1.1 Pas de gestion de la mémoire

Pas de gestion de la mémoire ! Cette méthode, qui a l'avantage de la simplicité et de la rapidité, permet toute liberté quand à l'utilisation de la mémoire. En effet, toute adresse est accessible, et peut être utilisée pour n'importe quelle tâche. Le désavantage : aucune fonctionnalité, tout doit être reprogrammé, typiquement il n'y pas de système d'exploitation !

8.1.2 Le moniteur résidant

On cherche à protéger le noyau des interférences possibles de la part des utilisateurs. Pour cela, toute adresse d'instruction ou de donnée manipulée par un programme utilisateur est comparée à un registre barrière (fence register).

Tant que l'adresse est supérieure à la barrière, l'adresse est légale, sinon l'adresse est une référence illégale au moniteur et une interruption est émise (invalid adress).

Cette méthode demande que pour tout accès à la mémoire une vérification de la validité de l'adresse soit réalisée. Ceci ralentit toute exécution d'un accès mémoire. (Paterson donne comme exemple de ralentissement des temps de 980 nanosecondes sans vérification et 995 nanosecondes avec vérification). Globalement ce temps supplémentaire peut être oublié.

8.1.3 Le registre barrière

L'implémentation d'un tel mécanisme doit être réalisée de façon matérielle.

La valeur du registre barrière est parfois réalisée de façon fixe sur une machine, ce qui pose des problèmes dès que l'on veut changer le noyau et/ou protéger plus de mémoire (voir DOS).

8.1.4 Le registre base

Le mécanisme suivant est une notion plus utile et plus ergonomique pour décrire la zone d'adressage d'un programme, et utile pour résoudre le problème de déplacement des programmes en mémoire (relocation).

En effet, du fait que l'on utilise un registre barrière, les adresses utilisables de la mémoire ne commencent plus à 0000, alors que l'utilisateur veut continuer à utiliser des adresses logiques qui commencent à 0000.

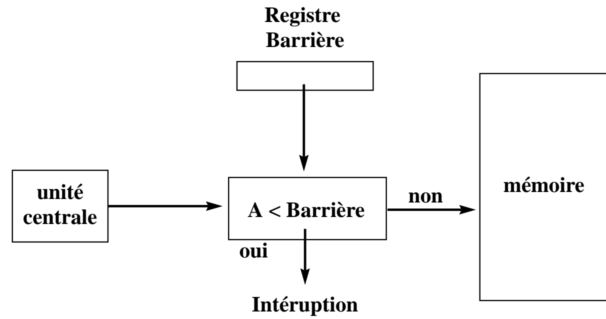

Fig. 8.5 – Implémentation du registre Barrière.

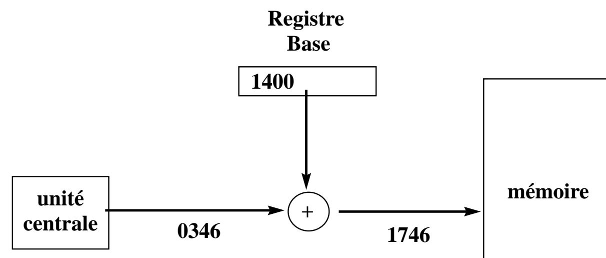

Fig. 8.6 – Implémentation du registre de Base.

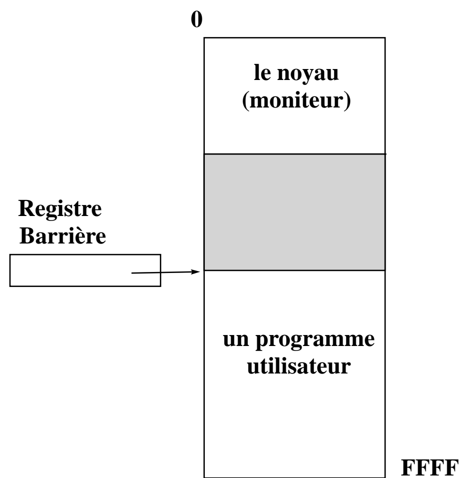

Fig. 8.7 – Positionnement d'un processus par un registre de Base.

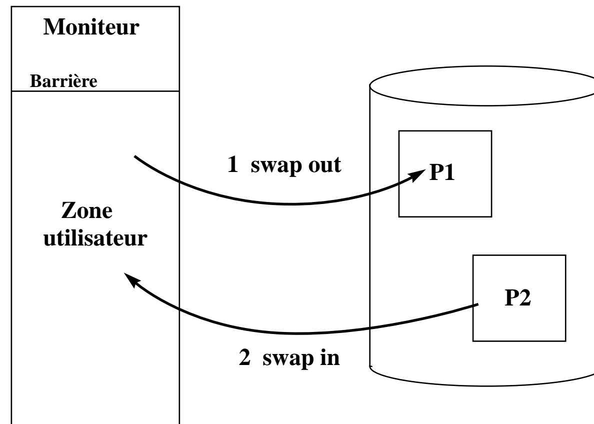

Fig. 8.8 – Un système de swap utilisant uniquement un registre barrière.

Pour continuer à fournir cette possibilité le registre barrière est transformé en registre de base (relocation) . A chaque utilisation d'une adresse logique du programme, on ajoute à cette adresse la valeur du registre de base pour trouver l'adresse physique. L'utilisateur ne connaît plus les adresses physiques. Il travaille uniquement avec des adresses logiques (xdb).

Le moniteur a évidemment une valeur nulle pour son registre de base et donc peut adresser toute la mémoire. Le changement de la valeur du registre de base se fait de façon protégée en mode moniteur.

Ces deux systèmes de protection de la mémoire sont clairement mono-processus. Seul le moniteur peut être protégé par ces mécanismes, il n'est pas possible de protéger les processus entre eux.

8.1.5 Le swap

Il est possible avec les registres barrière ou les registres de base d'écrire des systèmes temps partagé, en utilisant le mécanisme de swap (échange).

Swapper, c'est échanger le contenu de la mémoire centrale avec le contenu d'une mémoire secondaire. Par extension swapper devient l'action de déplacer une zone mémoire de la mémoire vers le support de swap (en général un disque) ou réciproquement du périphérique de swap vers la mémoire.

Le système va réaliser cet échange à chaque changement de contexte. Les systèmes de swap utilisent une mémoire secondaire qui est en général un disque mais on peut utiliser d'autre supports secondaires plus lents ou plus rapides comme des bandes ou mémoires secondaires (non accessibles par l'unité de traitement).

8.1.6 Le coût du swap

Sur un tel système, le temps de commutation de tâches est très important. Il est donc nécessaire que chaque processus reste possesseur de l'unité de traitement un temps suffisamment long pour que le ralentissement dû au swap ne soit pas trop sensible. Que ce passe-t-il sinon ? Le système utilise la majeure partie de ses ressources à déplacer des processus en et hors mémoire centrale. L'unité de traitement n'est plus utilisée au maximum ...

8.1.7 Utilisation de la taille des processus

Pour améliorer les mécanismes de swap, on remarque que le temps de swap est proportionnel à la taille des données à déplacer. Pour améliorer les performances, il faut donc introduire la notion

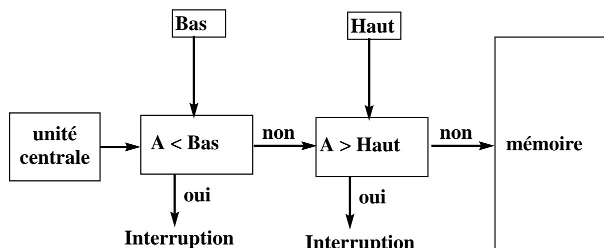

FIG. 8.9 – Double registre barrière.

de taille effective d'un processus, ce qui permet d'améliorer le débit mais cela impose que toutes les augmentations ou réductions de taille d'un processus utilisateur soient réalisée par un appel système (`sbrk`) afin que le noyau connaisse à tout moment la taille réelle de chaque processus.

8.1.8 Swap et exécutions concurrentes

Une autre approche très efficace est de réaliser le swap pendant l'exécution d'un autre processus. Mais avec le système de registres de relocation c'est dangereux. En effet nous ne pouvons pas assurer qu'un processus utilisateur donné ne va pas écrire dans les adresses réservées à un autre processus.

8.1.9 Contraintes

Le swap introduit d'autres contraintes : un processus doit être en préempté actif pour être swappé, c'est à dire n'être en attente d'aucune entrée-sortie. En effet, si P1 demande une E/S et pendant cette demande il y a échange de P1 et P2, alors la lecture demandée par P1 a lieu dans les données de P2.

8.1.10 Deux solutions existent

Soit ne jamais swapper de processus en attente d'entrées-sorties. Soit réaliser toutes les entrées-sorties dans des buffers internes au noyau (solution UNIX), ce qui a pour coût une recopie mémoire à mémoire supplémentaire par E/S. Les transferts entre le noyau et le processus ayant lieu uniquement quand le processus est en mémoire.

8.1.11 Les problèmes de protection

Nous venons d'apercevoir des problèmes de protection entre un processus et le noyau. Si l'on autorise plusieurs processus à résider en mémoire en même temps, il nous faut un mécanisme de protection inter-processus.
Deux méthodes sont couramment utilisées : les extensions du registre barrière et du registre de base (relocation).

8.1.12 Les registres doubles

Deux registres Barrière Bas et Haut
Si Adresse $<$ Bas $\longrightarrow$ lever une exception erreur d'adresse
Si Adresse $>=$ Haut $\longrightarrow$ lever une exception erreur d'adresse
Sinon adresse correcte.

Deux registres de relocation *Base* et *Limit*, on travaille avec des adresses logiques *Limit* donne la valeur maximale d'une adresse logique et *Base* donne la position en mémoire de l'adresse logique

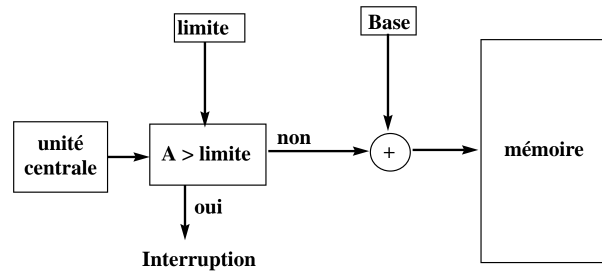

FIG. 8.10 – Base et Limite.

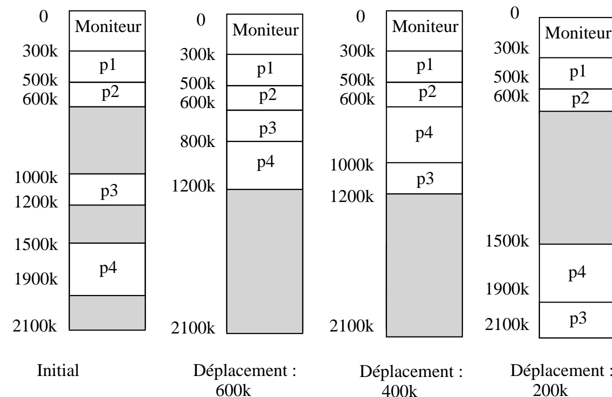

FIG. 8.11 – Une situation d'ordonnancement de processus en mémoire.

zéro.

Si Adresse >= *Limit* ⟶ lever une exception erreur d'adresse
sinon utiliser l'adresse physique Adresse+*Base*.

8.2 Ordonnancement en mémoire des processus

Les choix de l'implémentation des mécanismes d'adressage influence énormément l'ordonnancement des processus.

Nous travaillons dans le cas d'un système de traitement par lots c'est à dire en temps partagé mais les processus restent en mémoire tout le temps de leur exécution. S'il n'y a plus de place le processus est mis en attente (i.e. non chargé en mémoire).

Nous devons résoudre le problème suivant : il nous faut un algorithme pour choisir dynamiquement, parmi les blocs libres de la mémoire centrale, celui qui va recevoir le nouveau processus (algorithme d'allocation de mémoire à un processus). On reconnaît en général trois méthodes :

First-fit Le premier bloc suffisamment grand pour contenir notre processus est choisi.

Best-fit Le plus petit bloc suffisamment grand pour contenir notre processus est choisi.

Worst-fit Le bloc qui nous laisse le plus grand morceau de mémoire libre est choisi (le plus grand bloc).

De nombreuse expériences pratiques et des simulations ont montré que le meilleur est first-fit puis best-fit et que ces deux algorithmes sont beaucoup plus efficaces que worst-fit. **Compactage** On cherche à améliorer ces mécanismes en défragmentant la mémoire c'est à dire en déplaçant les processus en mémoire de façon à rendre contiguës les zones de mémoire libre de façon à pouvoir les utiliser.

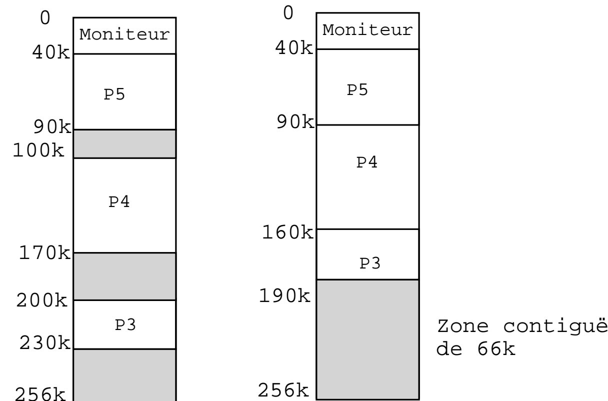

Fig. 8.12 – Compactage

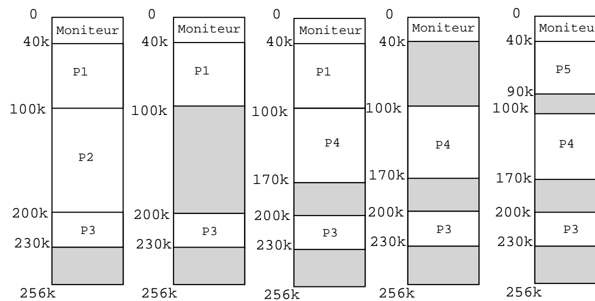

Fig. 8.13 – Plusieurs déplacements possibles.

8.3 Allocation non-contiguë

8.3.1 Les pages et la pagination

Pour accélérer ces mécanismes d'allocation, la notion de page a été introduite.

On va découper la mémoire et les processus en pages. Grâce à ce système, il ne sera plus nécessaire de placer les processus dans une zone contigüe de la mémoire. Il devient possible d'allouer de la mémoire à un processus sans avoir à réaliser de compactage !

Ce principe des page nécessite de nouvelles possibilités matérielles. Toute adresse est maintenant considérée comme un couple

(Numéro de page, Position dans la page)

A : adresse logique, P : taille de page
Numéro de page = A div P
Position = A modulo P

8.3.2 Ordonnancement des processus dans une mémoire paginée

Le choix de l'organisation mémoire a une influence prépondérante sur l'ordonnancement des processus, qui devient beaucoup plus indépendant de la mémoire quand celle-ci est paginée.
Le désavantage de la méthode de gestion de mémoire par un mécanisme de page est le phénomène de **fragmentation interne**. On alloue une page entière alors que le processus ne l'utilise qu'en partie. Mais la taille des mémoires et des processus deviennent tels par rapport aux tailles de page que cette perte devient minime.

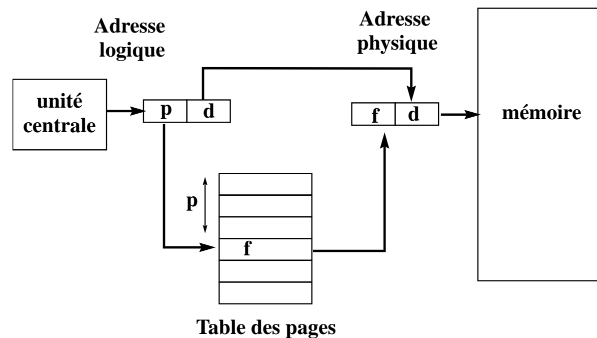

FIG. 8.14 – Calcul d'une adresse avec la table des pages

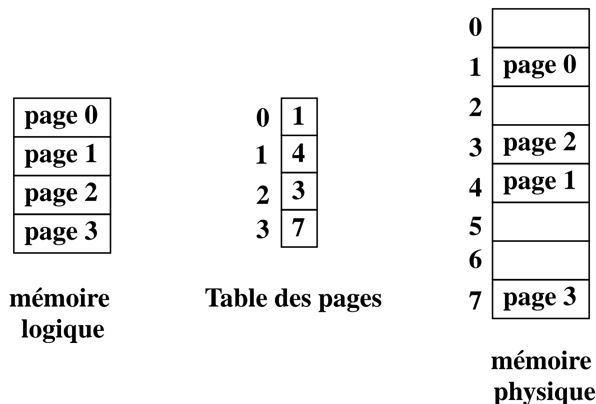

FIG. 8.15 – La mémoire logique et la Table des pages.

Un avantage des pages est une plus grande simplicité du partage de la mémoire entre différents processus. En particulier quand plusieurs processus partagent le même code. La page qui contient du code utilisé par les processus sera partageable et protégée en écriture.

Sous Unix le compilateur produit automatiquement des programmes dont la partie code est partageable.

8.3.3 Comment protéger la mémoire paginée

Les protections d'accès sont faites au niveau de la table des pages.

On a une table des pages globale. C'est donc le système qui alloue les pages à un processus, qui par construction (du système de pagination) ne peut pas écrire en dehors de ses propres pages. De plus, dans la table des pages d'un processus, des drapeaux indiquent le type de page (droits d'accès en lecture/écriture/exécution).

8.3.4 La mémoire segmentée

Nous venons de voir que les adresses logiques utilisées par le programmeur sont différentes des adresses physiques.

La mémoire segmentée est une organisation de la mémoire qui respecte le comportement usuel des programmeurs, qui généralement voient la mémoire comme un ensemble de tableaux distincts contenant des informations de types différents. Un segment pour chaque type : données, code, table des symboles, librairies etc. Ces différentes zones ayant des tailles variées, et parfois variables au cours du temps (le tas par exemple).

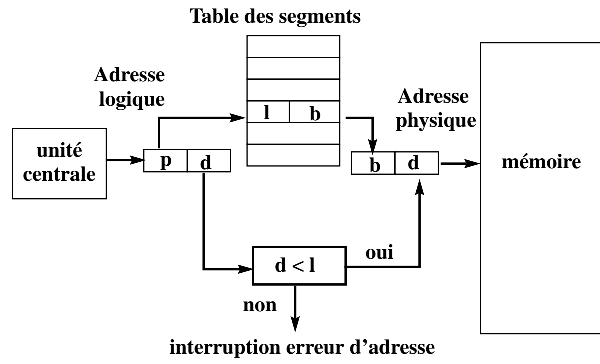

Fig. 8.16 – Mémoire segmentée

La mémoire segmentée non paginée pose des problèmes de compactage (défragmentation). La stratégie idéale est : la mémoire en segments paginés.

Chapitre 9

La mémoire virtuelle

Les méthodes de gestion mémoire que nous venons de voir ont toutes un défaut majeur qui est de garder l'ensemble du processus en mémoire, ce qui donne :
- un coût en swap important
- Impossibilité de créer de très gros processus.

Les méthodes de mémoire virtuelle permettent d'exécuter un programme qui ne tient pas entièrement en mémoire centrale !

Nous avons commencé par présenter des algorithmes de gestion de la mémoire qui utilisent le concept de base suivant :
l'ensemble de l'espace logique adressable d'un processus doit être en mémoire pour pouvoir exécuter le processus.

Cette restriction semble à la fois raisonnable et nécessaire, mais aussi très dommageable car cela limite la taille des processus à la taille de la mémoire physique.

Or si l'on regarde des programmes très standards, on voit que :
- il y des portions de code qui gèrent des cas très inhabituels qui ont lieu très rarement (si ils ont lieu)
- les tableaux, les listes et autres tables sont en général initialisés à des tailles beaucoup plus grandes que ce qui est réellement utile
- Certaines options d'application sont très rarement utilisées

Même dans le cas où le programme en entier doit résider en mémoire, tout n'est peut-être pas absolument nécessaire en même temps.

Avec la mémoire virtuelle, la mémoire logique devient beaucoup plus grande que la mémoire physique.

De nombreux avantages :
Comme les utilisateurs consomment individuellement moins de mémoire, plus d'utilisateurs peuvent travailler en même temps. Avec l'augmentation de l'utilisation du CPU et de débit que cela implique (mais pas d'augmentation de la vitesse).

Moins d'entrées-sorties sont effectuées pour l'exécution d'un processus, ce qui fait que le processus s'exécute (temps réel) plus rapidement.

9.0.5 Les overlays

Une des premières versions d'exécutables partiellement en mémoire est celle des "overlay" qui est l'idée de charger successivement des portions disjointes et différentes de code en mémoire, exécutées l'une après l'autre.

Les différentes passes d'un compilateur sont souvent réalisées en utilisant un overlay (préprocesseurs, pass1, pass2, pour les compilateurs C).

Les overlay nécessitent quelques adaptations de l'éditeur de liens et des mécanismes de relocation.

9.0.6 Le chargement dynamique

Un autre système couramment utilisé dans les logiciels du marché des micros est le chargement dynamique. Avec le chargement dynamique, une fonction n'est chargée en mémoire qu'au moment de son appel. Le chargement dynamique demande que toutes les fonctions soient repositionnables en mémoire de façon indépendante.

A chaque appel de fonction on regarde si la fonction est en mémoire sinon un éditeur de liens dynamique est appelé pour la charger.

Dans les deux cas (overlay et chargement dynamique), le système joue un rôle très restreint, il suffit en effet d'avoir un bon système de gestion de fichiers.

Malheureusement, *le travail* que doit réaliser le programmeur pour choisir les overlays et/ou installer un mécanisme de chargement dynamique efficace *est non trivial* et requiert que le programmeur ait une *parfaite connaissance* du programme.

Ceci nous amène aux *techniques automatiques.*

9.1 Demand Paging

La méthode de **Demand Paging** est la plus répandue des implémentations de mémoire virtuelle, elle demande de nombreuse capacités matérielles.

Nous partons d'un système de swap où la mémoire est découpée en pages. Comme pour le swap, quand un programme doit être exécuté nous le chargeons en mémoire (swap in) mais au lieu de faire un swap complet, on utilise un "swappeur paresseux" (lazy swapper).

Un swappeur paresseux charge une page **uniquement si** elle est nécessaire.

Que ce passe-t-il quand le programme essaie d'accéder à une page qui est hors mémoire ?

- le matériel va traduire l'adresse logique en une adresse physique grâce à la table des pages.
- tant que les pages demandées sont en mémoire, le programme tourne normalement, sinon si la page est contenue dans l'espace des adresses logiques mais n'est pas chargée, il y a une **page fault**.

En général, une erreur d'adresse est dûe à une tentative d'accès à une adresse extérieure (invalide). Dans ce cas, le programme doit être interrompu, c'est le comportement normal d'un système de swap.

Mais il est possible avec un swappeur paresseux que la page existe mais ne soit pas en mémoire centrale, d'où les étapes suivantes dans ce cas :

On peut faire démarrer un processus sans aucune page en mémoire. La première **Page Fault** aurait lieu à la lecture de la première instruction (l'instruction n'étant pas en mémoire).

Il faut réaliser une forme spéciale de sauvegarde de contexte, il faut garder une image de l'état du processus qui vient d'effectuer une **Page Fault** mais de plus il faudra redémarrer (réexécuter) l'instruction qui a placé le processus dans cet état, en effet il est possible que l'instruction ne se soit pas terminé par manque de données.

Le système d'exploitation a ici un rôle important, c'est lui qui va réaliser le chargement de la page manquante puis relancer le processus et l'instruction.

Les circuits nécessaires à la méthode de Demande Paging sont les mêmes que ceux que l'on utilise pour un système de swap paginé, c'est-à-dire une mémoire secondaire et un gestionnaire de pages (table des pages).

Par contre, la partie logicielle est beaucoup plus importante.

Enfin il faut que les **instructions soient interruptibles**, ce qui n'est pas toujours le cas sur tous les processeurs et ce qui est fondamental, comme nous allons le voir sur des exemples :

add A,B in C

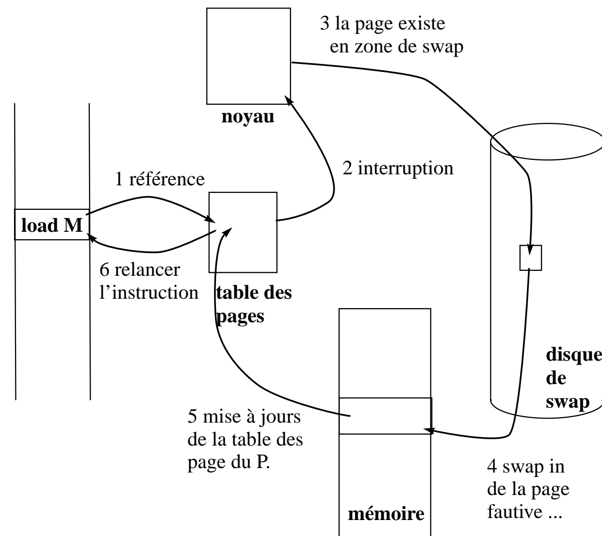

FIG. 9.1 – Etapes de la gestion d'une erreur de page

1. chercher et décoder l'instruction add
2. charger le contenu de l'adresse A
3. charger le contenu de l'adresse B
4. sommer et sauvegarder dans C

Si l'erreur de page a lieu dans le 4ième accès à la mémoire (C), il faudra de nouveau recommencer les 3 accès mémoire de l'instruction, c'est-à-dire lire l'instruction, etc.

Un autre type de problème vient d'instructions comme la suivante que l'on trouve sur PDP-11 :

MOV (R2)++,–(R3)

cette instruction déplace l'objet pointé par le registre R2 dans l'adresse pointé par R3, R2 est incrémenté après le transfert et R3 avant.

Que se passe-t-il si l'on a une erreur de page en cherchant à accéder à la page pointé par R3 ?

9.1.1 Efficacité

Efficacité des performances de Demand Paging :

Soit ma = 500 nanosecondes, le temps moyen d'accès a une mémoire.
le temps effectif d'accès avec le Demand Paging est

temps effectif = (1-p)*ma + p * "temps de gestion de l'erreur de page"
où p est la probabilité d'occurrence d'une erreur de page (page fault).

Une erreur de page nécessite de réaliser les opérations suivantes

1. lever une interruption pour le système
2. sauvegarder le contexte du processus
3. déterminer que l'interruption est une erreur de page
4. vérifier que la page en question est une page légale de l'espace logique, déterminer où se trouve la page dans la mémoire secondaire.
5. exécuter une lecture de la page sur une page mémoire libre (libérer éventuellement une page cf. algorithme de remplacement de page)
   – attendre que le périphérique soit libre

   - temps de latence du périphérique
   - commencer le transfert
6. allouer pendant ce temps-là le cpu à un autre utilisateur
7. interruption du périphérique
8. sauvegarde du contexte du processus courant
9. déterminer que l'interruption était la bonne interruption (venant du périphérique)
10. mise à jour de la table des pages et d'autres pages pour indiquer que la page demandée est en mémoire maintenant.
11. attendre que le processus soit sélectionné de nouveau pour utiliser l'unité centrale (cpu)
12. charger le contexte du processus !

Toutes ces instructions ne sont pas toujours réalisées (on peut en particulier supposer que l'on ne peut pas préempter l'unité centrale, mais alors quelle perte de temps pour l'ensemble du système).

Dans tous les cas, nous devons au moins réaliser les 3 actions suivantes :

- gérer l'interruption
- swapper la page demandée
- relancer le processus

Ce qui coûte le plus cher est la recherche de la page sur le disque et son transfert en mémoire, ce qui prend de l'ordre de 1 à 10 millisecondes.

Ce qui nous donne en prenant une vitesse d'accès mémoire de 1 microseconde et un temps de gestion de page de 5 millisecondes un

$$\text{temps effectif} = (1 - p) + p \times 5000 \quad \text{microsecondes}$$

Une erreur de page toutes les mille pages nous donne un temps effectif onze fois plus long que l'accès standard.

Il faut réduire à moins d'une erreur de page tout les 100000 accès pour obtenir une dégradation inférieure à 10

On comprend bien que les choix à faire sur des pages qu'il faut placer en mémoire sont donc très importants.

Ces choix deviennent encore plus importants quand l'on a de nombreux utilisateurs et qu'il y a sur-allocation de la mémoire, exécution concurrente de 6 processus de la taille supérieure ou égale à la mémoire physique !

Si l'on suppose de plus que nos 6 programmes utilisent dans une petite séquence d'instructions toutes les pages de leur mémoire logique, nous nous trouvons alors dans une situation de pénurie de pages libres.

Le système d'exploitation peut avoir recoure à plusieurs solution dans ce cas-là

1. tuer le processus fautif ...
2. utiliser un algorithme de remplacement de page

Cet algorithme de remplacement est introduit dans notre séquence de gestion d'erreur de page là où l'on s'attribuait une page libre de la mémoire centrale.

Maintenant il nous faut sélectionner une victime, c'est-à-dire, une des pages occupées de la mémoire centrale qui sera swappée sur disque et remplacée par la page demandée.

Remarquons que dans ce cas-là notre temps de transfert est doublé, comme il faut à la fois lire une page et sauvegarder une page sur disque (le temps de transfert disque est ce qui est le plus coûteux dans la gestion d'une erreur de page).

Il est possible de réaliser des systèmes de **demand segments**, mais le lecteur avisé remarquera rapidement les problèmes posés par la taille variable des segments.

9.2 Les algorithmes de remplacement de page

Un algorithme de remplacement de page doit minimiser le nombre de Page Faults.

On recherche l'algorithme qui réduit au mieux la probabilité d'occurrence d'une erreur de page. Un algorithme est évalué en prenant une chaîne de numéros de page et en comptant le nombre de fautes de page qui ont lieu au cours de cette suite d'accès, et cela en fonction du nombre de pages de mémoire centrale dont il dispose.

Pour illustrer les algorithmes de remplacement, nous utiliserons la suite de pages suivante :
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
et 3 pages en mémoire centrale.

9.2.1 Le remplacement optimal

Utiliser comme victime la page qui ne sera pas utilisée pendant le plus longtemps.
Soit pour notre suite :

```
7xx 70x 701 201 - 203 - 243 - -203 - - 201 - - - 701 - -
```

soit seulement 9 fautes de page.
Mais cet "algorithme" n'est valable que dans un cas où l'on connaît à l'avance les besoins, ce qui n'est généralement pas le cas.

9.2.2 Le remplacement peps (FIFO)

L'algorithme le plus simple est Premier Entré Premier Sorti (First-In-First-Out).

Quand une victime doit être sélectionnée c'est la page la plus ancienne qui est sélectionnée.

Soit pour la liste

```
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
```

et trois page de mémoire centrale :

```
7XX/70X/701/201-201/231/230/430/420/423/
023-023-023/013/012-012-012/712/702/701
```

soit **Quinze** Page Faults.
Ce mécanisme rapide et simple à programmer n'est malheureusement pas très efficace. Il existe des suites de pages pour lesquelles cet algorithme fait plus de page faults avec quatre pages mémoire qu'avec trois ! (par exemple : 1,2,3,4,1,2,5,1,2,3,4,5).

9.2.3 Moins récemment utilisée LRU.

LRU (Least Recently Used page).
Nous utilisons ici le vieillissement d'une page et non plus l'ordre de création de la page. On fait le pari que les pages qui ont été récemment utilisées le seront dans un proche avenir, alors que les pages qui n'ont pas été utilisées depuis longtemps ne sont plus utiles.

Soit pour notre suite :

```
7xx 70x 701 201 - 203 - 403 402 432 032 - - 132 - 102 - 107 -
```

soit **Douze** Page Faults.

L'algorithme LRU est un bon algorithme mais il pose de nombreux problèmes d'implémentation et peut demander de substantiels outils matériels.

Des solutions logicielles :

Des compteurs à chaque entrée de la table des pages, on ajoute un compteur de temps qui est mis à jour à chaque accès à la page. Il faut rechercher sur l'ensemble de la table la victime. De plus, ces temps doivent être mis à jour quand on change de table de page (celle d'un autre processus ...). On ne peut utiliser le temps réel ...

Une pile à chaque fois que l'on accède à une page, la page est placée en sommet de pile. Le dessus est toujours la page la plus récemment utilisée et le fond de la pile la moins récemment utilisée.

Des masques On utilise un octet associé à chaque page. Le système positionne à 1 le bit de poids fort à chaque accès à la page. Toutes les N millisecondes (click d'horloge, cf clock, N = 100 sur fillmore) le système fait un décalage à droite de l'octet associé à chaque page. On obtient ainsi un historique de l'utilisation de la page. L'octet à 00000000 indique que la page n'a pas été utilisée depuis 8 cycles, 11111111 indique que la page a été utilisée pendant les 8 cycles. La page de masque 11000100 à été utilisée plus récemment que 01110111. Si l'on interprète ces octets comme des entiers non-signés, c'est la page ayant le plus petit octet qui a été utilisée le moins récemment (l'unicité des numéros n'étant pas assurée, la sélection entre numéros identiques se fait avec l'ordre FIFO).

9.2.4 L'algorithme de la deuxième chance

Un bit associé à chaque page est positionné à 1 à chaque fois qu'une page est utilisée par un processus. Avant de retirer une page de la mémoire, on va essayer de lui donner une deuxième chance. On utilise un algorithme FIFO plus la deuxième chance :
Si le bit d'utilisation est à 0, la page est swappée hors mémoire (elle n'a pas été utilisée depuis la dernière demande de page).
Si le bit est à 1, il est positionné a zéro et l'on cherche une autre victime. Ainsi cette page ne sera swappée hors mémoire que si toutes les autres pages ont été utilisées, et utilisent aussi leur deuxième chance.

On peut voir ceci comme une queue circulaire, où l'on avance sur les pages qui ont le bit à 1 (en le positionnant à zéro) jusqu'à ce que l'on trouve une page avec le bit d'utilisation à zéro.

9.2.5 Plus fréquemment utilisé MFU

Plus fréquemment Utilisée :
Comme son nom l'indique, c'est la fréquence d'utilisation qui joue au lieu de l'ancienneté, mais c'est le même mécanisme que LRU. Ces deux algorithmes de LRU et MFU sont rarement utilisés car trop gourmands en temps de calcul et difficiles à implémenter, mais ils sont assez efficaces.

9.2.6 Le bit de saleté (Dirty Bit)

Remarquons que si il existe une copie identique sur disque (zone de swap) d'une page de mémoire, il n'est pas nécessaire dans le cas d'un swapout de sauvegarder la page sur disque, il suffit de la libérer.
Le bit de saleté permet d'indiquer qu'une page est (ou n'est plus) conforme à la page en zone de swap.

Ce bit de propreté est utilisé dans les autres algorithmes, on choisit entre deux victimes possibles la plus propre, c'est-à-dire celle qui ne nécessite pas de swapout.

9.3 Allocation de pages aux processus

Comment répartir les pages sur les différents processus et le système ?

remplacement local le processus se voit affecté un certain nombre de pages qu'il va utiliser de façon autonome, son temps d'exécution ne dépend que de son propre comportement.

remplacement global le comportement d'allocation de pages aux processus dépend de la charge du système et du comportement des différents processus.

Le remplacement local demande que l'on réalise un partage entre les différents processus.

Le partage "équitable" : m pages de mémoire physique, n processus, m/n pages par processus ! On retrouve ici un problème proche de la fragmentation interne, un grand nombre de pages est donné à un processus qui en utilise effectivement peu.

On fait un peu mieux en utilisant : $S = \Sigma\ s_i$ où s_i est le nombre de pages de la mémoire logique du Processus i. Chaque processus se voit attribué $(s_i/S)m$ pages. On améliore en faisant varier ce rapport en fonction de la priorité de chaque processus.

Problèmes d'écroulement Si le nombre de pages allouées à un processus non-prioritaire tombe en dessous de son minimum vital, ce processus est constamment en erreur de page : il passe tout son temps à réaliser des demandes de pages. Ce processus doit être alors éjecté entièrement en zone de swap et reviendra plus prioritaire quand il y aura de la place.
Un exemple de bonne et mauvaise utilisation des pages (rappel les compilateurs c allouent les tableaux sur des plages d'adresse croissante contigües int m[A][B] est un tableau de A tableaux de B entiers) :

```
/* bonne initialisation */
int m[2048][2048];
main()
{int i,j;
for(i=0;i<2048;i++)
        for(j=0;j<2048;j++)
        m[i][j] = 1;
}
```

ce processus accède a une nouvelle page toute les 2048 affectation.

```
/* mauvaise initialisation */
int m[2048][2048];
main()
{int i,j;
for(i=0;i<2048;i++)
        for(j=0;j<2048;j++)
        m[j][i] = 1;
}
```

ce processus accède a une nouvelle page toute les affectations !
Attention : En fortran l'allocation des tableaux se fait dans l'autre sens par colones ...

Si la mémoire est libre et assez grande, les deux processus sont grossièrement aussi rapides, par contre si on lance dix exemplaires du premier, le temps d'attente est juste multiplié par 10. Pour le deuxième, le temps d'attente est au moins multiplié par 100 (je n'ai pas attendu la fin de l'exécution).

9.4 L'appel fork et la mémoire virtuelle

Nous avons vu que la primitive fork() réalise une copie de l'image mémoire du processus père pour créer le processus fils. Cette copie n'est pas intégrale car les deux processus peuvent partager des pages marquées en lecture seule, en particulier le segment du code est partagé par les deux processus (réentrance standard des processus unix).

Mais avec le système de demand-paging, on peut introduire une nouvelle notion qui est la "copie sur écriture" (copy on write). On ajoute à la structure de page de la table des pages des indicateurs de "copie sur écriture". L'idée est de réaliser la copie de la page uniquement dans le cas où l'un des processus qui peuvent y accèder réalise une écriture. Dans ce cas-là, la page est recopiée avant l'écriture et le processus écrivain possède alors sa propre page.

L'intérêt de ce mécanisme est surtout visible dans le cas très fréquent où le fork est immédiatement suivi par un exec. En effet, ce dernier va réaliser une libération de toutes les pages, il est donc inutile de les recopier juste avant cette libération.

Le système BSD a introduit la première version de cette idée en partant de l'appel système vfork() qui lui permet le partage totale de toutes les pages entre le processus père et le processus fils sans aucune copie. L'intérêt est de pouvoir réaliser rapidement un execve sans avoir à recopier l'espace d'adressage du processus père.

9.5 Projection de fichiers en mémoire

La fonction mmap permet la projection de fichiers en mémoire. Le segment du fichier indiqué est placé en mémoire à partir de l'adresse indiquée. Le segment de fichier peut ainsi être parcouru par des accès par adresse sans utiliser de commande de lecture ou d'écriture.

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

void *mmap(void  *adr, int len,
           int   prot, int options,
           int   desc, int offset);

int munmap(void *adr, int len);
```

L'adresse adr indique où doit être placé le fichier, cette adresse doit être une adresse de début de page (un multiple de sysconf(_SC_PAGE_SIZE)), si le paramètre est NULL alors le système sélectionne l'adresse de placement qui est retournée par la fonction. L'intervalle de position

[offset, offset+len]

du fichier desc est placé en mémoire.
prot indique les protections d'accès sous HP-UX les protections suivantes sont disponible :

```
---     PROT_NONE
r--     PROT_READ
r-x     PROT_READ|PROT_EXECUTE
rw      PROT_READ|PROT_WRITE
rwx     PROT_READ|PROT_WRITE|PROT_EXECUTE
```

`options` indique si l'on veut que les écritures réalisées dans les pages contenant la projection soient partagées (MAP_SHARED), ou au contraire qu'une copie sur écriture soit réalisée (MAP_PRIVATE).

La fonction `munmap` permet de libérer la zone mémoire d'adresse `adr` et de longueur `len`.

Pour une autre forme de mémoire partagée, voir le chapitre sur les IPC (sur le web).

Un exemple d'utilisation de `mmap` pour copier un fichier :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>

int main(int argc,char *argv[])
{
  int fdin,fdout ;
  struct stat statbuf ;
  char *src,*dst ;
  if (argc != 3)
    {
      fprintf(stderr,"usage : %s source destination ",argv[0]) ;
      exit(-1) ;
    }
  if ((fdin = open(argv[1], O_RDONLY)) < 0)
    {
      fprintf(stderr,"impossible d\'ouvrir : %s en lecture ",argv[1]) ;
      exit(-2) ;
    }
  if ((fdout = open(argv[2], O_RDWR|O_CREAT|O_TRUNC,0666)) < 0)
    {
      fprintf(stderr,"impossible d\'ouvrir : %s en ecriture ",argv[2]) ;
      exit(-3) ;
    }
  if (fstat(fdin,&statbuf) < 0 )
    {
      fprintf(stderr,"impossible de faire stat sur  %s ",argv[1]) ;
      exit(-4) ;
    }
  if (lseek(fdout, statbuf.st_size -1 , SEEK_SET) == -1 )
    {
      fprintf(stderr,"impossible de lseek  %s ",argv[2]) ;
      exit(-5) ;
    }
  if (write(fdout,"",1) != 1)
    {
      fprintf(stderr,"impossible d\'ecrire sur   %s ",argv[2]) ;
      exit(-6) ;
    }
```

```
  if ((src = mmap (0,statbuf.st_size, PROT_READ,
                MAP_FILE | MAP_SHARED, fdin,0)) == (caddr_t) -1 )
    {
      fprintf(stderr,"impossible de mapper   %s ",argv[1]) ;
      exit(-7) ;
    }
  if ((dst = mmap (0,statbuf.st_size, PROT_READ | PROT_WRITE,
                MAP_FILE | MAP_SHARED, fdout,0)) == (caddr_t) -1 )
    {
      fprintf(stderr,"impossible de mapper   %s ",argv[2]) ;
      exit(-8) ;
    }
  memcpy(dst,src,statbuf.st_size) ; /* copie */

  exit(0) ;

}
```

Attention, quand vous utilisez `mmap` c'est de la mémoire, mais c'est a vous de gérer l'allignement. Exemple :

```
char *p = map(...);
int *q = p+1; // warning
 *q = 1 ; // problème d'allignement
```

9.6 Les conseils et politiques de chargement des zones mmappées

Une foit que l'on décider de faire des projection en mémoire avec `mmap` il peut être oportin de faire appel à la fonction `madvise` qui permet de donner un conseil au système en le prévenant par avance de la façon dont vous aller utiliser le segement de mémoire. En particulier allez vous lire le fichier sequentiellement ou de façon alléatoire. Avez vous encore besoin du fichier après lecture etc. Biensur la fonction `madvise` ne se limite pas aux pages mappés mais c'est sur celle ci qu'il est le plus facile de prendre des décisions, les autres pages étant gérer dans la pile le tas et le code zone plus délicates et moins bien cartographiées en générale (sic).

```
#include <sys/mman.h>
int madvise(void *start, size_t length, int advice);
```

La valeur du conseil `advice` :

MADV_NORMAL Comportement par défaut.

MADV_RANDOM prevoit des acces aux pages dans un ordre aleatoire.

MADV_SEQUENTIAL prevoit des acces aux pages dans un ordre sequentiel.

MADV_WILLNEED prevoit un acces dans un futur proche.

MADV_DONTNEED Ne prevoit pas d'acces dans un futur proche. Biensur si c'est une mmap vous pouvez aussi utiliser la commande `munmap`.

Biensur ce ne sont que des conseils le système les utilisera si il en a la possibilité, soit parce qu'il y a du temps idle (sans activité) soit parce qu'il profitera des lectures groupées sur disque en réalisant des lectures en avance cas séquentiel. Il peut aussi profiter de l'indication DONTNEED pour prendre des décissions dans le code de remplacement de page.

9.7 Chargement dynamique

Indépendement de l'existance de la mémoire virtuel il est possible de gérer "à la main" le code accessible en utilisant le chargement direct (non automatique) de librairies.

Pour construire une librairie de fichier lib.c :

```
#include <stdio.h>

/* fonction optionelle qui permet d'initialiser la libraire */
void _init()
{
  fprintf(stderr," intialisation \n");
}
/* fonction optionelle qui est appellée avant le déchargement */
void _fini()
{
fprintf(stderr," déchargement exécution de _fini \n");
}
/* des fonctions spécifiques a votre libraire */
int doit(int u)
{
  fprintf(stderr," doit to u= %d " , u );
  return u*u;
}
```

Compilation de la libraire, l'option nostartfile pour que le compilateur ne construise pas d'exécutable.

```
 gcc -shared -nostartfiles -o ./malib lib.c
```

Le fichier main qui va charger la libraire puis appeler une fonction de cette libraire.

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

int main(int argc, char **argv) {
  void *handle;
  int (*doit)(int);
  char *error;

  handle = dlopen ("./malib", RTLD_LAZY);
  if (!handle) {
    fputs (dlerror(), stderr);
    exit(1);
  }

  doit = dlsym(handle, "doit");
  if ((error = dlerror()) != NULL)  {
    fprintf (stderr, "%s\n", error);
    exit(1);
  }

  printf ("%d\n", (*doit)(23));
  handle = dlopen ("./malib", RTLD_LAZY); // nouveau référencement avce le même handle
 dlclose(handle); // décrementation du conteur de référence
```

```
 fprintf(stderr," avant le deuxieme dlclose \n");
  dlclose(handle);
}
```

Chapitre 10

Tubes et Tubes Nommés

Les tubes sont un mécanisme de communication qui permet de réaliser des communications entre processus sous forme d'un flot continu d'octets. Les tubes sont un des éléments de l'agrément d'utilisation d'UNIX. C'est ce mécanisme qui permet l'approche filtre de la conception sous UNIX.

Mécanisme de communication lié au système de gestion de fichier, les tubes nommés ou non sont des paires d'entrées de la table des fichiers ouverts, associées à une inode en mémoire gérée par un driver spécifique. Une entrée est utilisée par les processus qui écrivent dans le tube, une entrée pour les lecteurs du tube.
L'opération de lecture y est destructive !

L'ordre des caractères en entrée est conservé en sortie (premier entré premier sorti).

Un tube a une capacité finie : en général le nombre d'adresses directes des inodes du SGF (ce qui peut varier de 5 à 80 Ko).

10.1 Les tubes ordinaires (*pipe*)

Un tube est matérialisé par deux entrées de la table des ouvertures de fichiers, une de ces entrées est ouverte en écriture (l'entrée du tube), l'autre en lecture (la sortie du tube). Ces deux entrées de la table des fichiers ouverts nous donnent le nombre de descripteurs qui pointent sur elles. Ces valeurs peuvent être traduites comme :

nombre de lecteurs = nombre de descripteurs associés à l'entrée ouverte en lecture. *On ne peut pas écrire dans un tube sans lecteur.*

nombre d'écrivains = nombre de descripteurs associés à l'entrée ouverte en écriture. La nullité de ce nombre définit le comportement de la primitive `read` lorsque le tube est vide.

10.2 Création de tubes ordinaires

Un processus ne peut utiliser que les tubes qu'il a créés lui-même par la primitive `pipe` ou qu'il a hérités de son père grâce à l'héritage des descripteurs à travers `fork` et `exec`.

```
#include <unistd.h>
int pipe(int  p[2]);
```

On ne peut pas manipuler les descripteurs de tubes avec les fonctions et primitives : `lseek, ioctl, tcsetattr` et `tcgetattr`, comme il n'y a pas de périphérique associé au tube (tout est

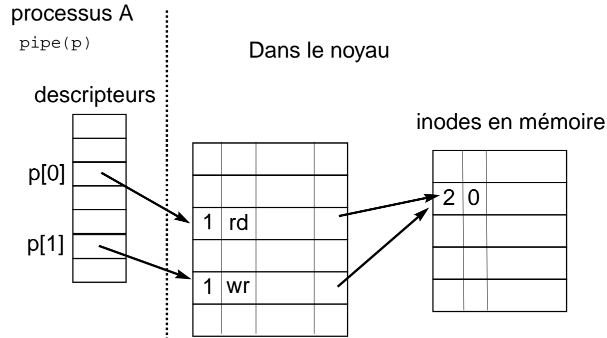

FIG. 10.1 – Ouverture d'un tube

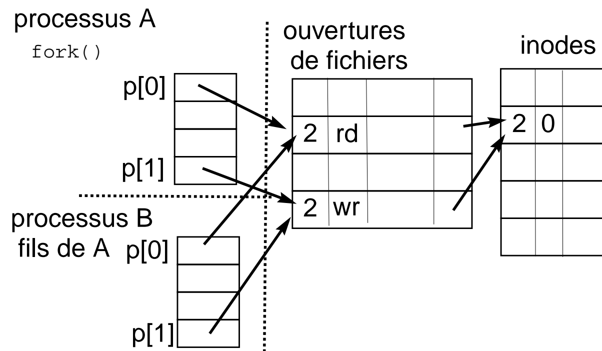

FIG. 10.2 – Héritage d'un tube

fait en mémoire).

Héritage d'un tube dans la figure 10.2 : le processus B hérite des descripteurs ouverts par son père A et donc, ici, du tube.

Dans la Figure 10.3, les descripteurs associés aux tubes sont placés comme descripteurs 0 et 1 des processus A et B, c'est à dire la sortie de A et l'entrée de B. Les autres descripteurs sont fermés pour assurer l'unicité du nombre de lecteurs et d'écrivains dans le tube.

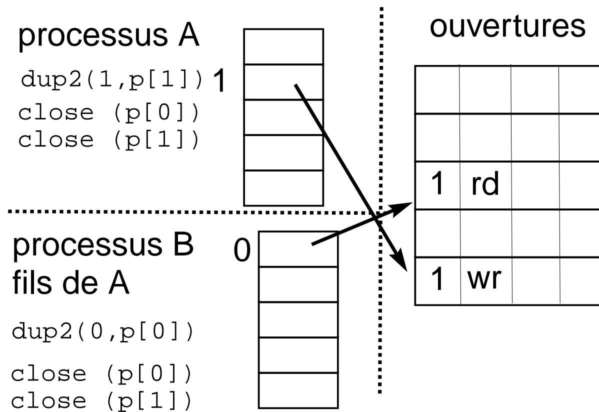

Fig. 10.3 – Redirection de la sortie standard de A dans le tube et de l'entrée standard de B dans le tube, et fermeture des descripteurs inutiles

10.3 Lecture dans un tube

On utilise l'appel système `read`.

```
int nb_lu;
nb_lu = read(p[0], buffer, TAILLE_READ);
```

Remarquer que la lecture se fait dans le descripteur p[0].

Comportement de l'appel :

Si le tube n'est **pas vide** et contient **taille** caractères :
lecture de nb_lu = min(taille, TAILLE_READ) caractères.
Si le tube est **vide**
Si le nombre d'écrivains est *nul*
alors c'est la fin de fichier et nb_lu est nul.
Si le nombre d'écrivains est *non nul*
Si lecture bloquante alors sommeil
Si lecture non bloquante alors en fonction de l'indicateur
`O_NONBLOCK` nb_lu= -1 et errno=EAGAIN.
`O_NDELAY` nb_lu = 0.

10.4 Ecriture dans un tube

```
nb_ecrit = write(p[1], buf, n);
```

L'écriture est atomique si le nombre de caractères à écrire est inférieur à PIPE_BUF, la taille du tube sur le système. (cf `<limits.h>`).

Si le nombre de lecteurs est nul
    envoi du signal SIGPIPE à l'écrivain.
Sinon
    Si l'écriture est bloquante, il n'y a retour que quand
        les n caractères ont été écrits dans le tube.
    Si écriture non bloquante
        Si n > PIPE_BUF, retour avec un nombre inférieur à n
            éventuellement -1 !
        Si n ≤ PIPE_BUF
            et si n emplacements libres, écriture nb_ecrit = n
        sinon retour -1 ou 0.

Comment rendre un read ou write non bloquant ? en utilisant fctnl sur le descripteur du tube. `F_SETFL` fixe de nouveaux attributs pour le descripteur de fichier fd. Les nouveaux attributs sont contenus dans arg. Seuls O_APPEND, O_NONBLOCK et O_ASYNC peuvent être modifiés ainsi, les autres attributs ne sont pas affectés.

10.5 Interblocage avec des tubes

Un même processus a deux accès à un tube, un accès en lecture, un accès en écriture et essaie de lire sur le tube vide en mode bloquant ⟶ le processus est bloqué indéfiniment dans la primitive `read`.

Avec deux processus :
deux tubes entre les deux processus, tous les deux bloqués en lecture ou tous les deux bloqués en écriture, tous les deux en attente d'une action de l'autre processus.

10.6 Les tubes nommés

Les tube nommés sont des tubes (`pipe`) qui existent dans le système de fichiers, et donc peuvent être ouverts grâce à une référence.
Il faut préalablement créer le tube nommé dans le système de fichiers, grâce à la primitive `mknod` (`mkfifo`), avant de pouvoir l'ouvrir avec la primitive `open`.

```
int mknod(reference, mode | S_IFIFO,0);
```

`mode` est construit comme le paramètre de mode de la fonction `open`.

En POSIX, un appel simplifié :

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *ref, mode_t mode);
```

On peut créer des FIFOs à partir du shell grâce à

```
mkfifo [-p] [-m mode] ref ...
```

L'ouverture d'un tube nommé se fait exclusivement soit en mode O_RDONLY soit en mode O_WRONLY, ainsi le nombre de lecteur et d'écrivain peut être comptabilisé.

10.6.1 Ouverture et synchronisation des ouvertures de tubes nommés

Il y a automatiquement synchronisation des processus qui ouvrent en mode bloquant un tube nommé.

L'opération d'ouverture sur un tube nommé est bloquante en lecture.
Le processus attend qu'un autre processus ouvre la fifo en écriture.
L'ouverture en écriture est aussi bloquante, avec attente qu'un autre processus ouvre la fifo en lecture. L'ouverture bloquante se termine de façons synchrone pour les deux processus.

Ainsi un unique processus ne peut ouvrire à la fois en lecture et écriture un tube nommé.

En mode non bloquant (O_NONBLOCK, O_NDELAY), seule l'ouverture en lecture réussit dans tous les cas. L'ouverture en écriture en mode non bloquant d'un tube nommé ne fonctionne que si un autre processus a déjà ouvert en mode non bloquant le tube en lecture, ou bien qu'il est bloqué dans l'appel d'une ouverture en lecture en mode bloquant. Ceci pour éviter que le processus qui vient d'ouvrir le tube nommé, n'écrive dans le tube avant qu'il n'y ait de lecteur (qu'un processus ait ouvert le tube en lecture) et ce qui engendrerait un signal SIGPIPE (tube détruit), ce qui n'est pas vrai car le tube n'a pas encore été utilisé.

10.6.2 Suppression d'un tube nommé

L'utilisation de `rm` ou `unlink` ne fait que détruire la référence, le tube n'est réellement détruit que lorsque son compteur de liens internes et externes est nul.
Une fois que tous les liens par référence sont détruits, le tube nommé devient un tube ordinaire.

10.6.3 les appels popen et pclose

Une interface plus facile pour lancer un coprocessus est proposé avec les primitives popen et pclose.

Chapitre 11

Les signaux

Les signaux sont un mécanisme asynchrone de communication inter-processus. Intuitivement, il sont comparables à des sonneries, les differentes sonneries indiquant des évènements différents. Les signaux sont envoyés à un ou plusieurs processus. Ce signal est en général associé à un évènement.

Peu portables entre BSD et ATT, ils deviennent plus commodes à utiliser et portables avec la norme POSIX qui utilise la notion utile de vecteur de signaux et qui fournit un mécanisme de masquage automatique pendant les procédures de traitement (comme BSD).

Un signal est envoyé à un processus en utilisant l'appel système :

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

`signal` est un numéro compris entre 1 et `NSIG` (défini dans `<signal.h>`) et `pid` le numéro du processus.

Le processus visé reçoit le signal sous forme d'un drapeau positionné dans son bloc de contrôle. Le processus est interrompu et réalise éventuellement un traitement de ce signal.

On peut considérer les signaux comme des interruptions logicielles, ils interrompent le flot normal d'un processus mais ne sont pas traités de façon synchrone comme les interruptions matérielles.

11.0.4 Provenance des signaux

Certains signaux peuvent être lancés à partir d'un terminal grâce aux caractères spéciaux comme `intr`, `quit` dont la frappe est transformée en l'envoi des signaux `SIGINT` et `SIGQUIT`. D'autres sont dûs à des causes internes au processus, par exemple : `SIGSEGV` qui est envoyé en cas d'erreur d'adressage, `SIGFPE` division par zéro (Floating Point Exception).

Enfin certains sont dûs à des évènements comme la déconnection de la ligne (le terminal) utilisé : si le processus leader d'un groupe de processus est déconnecté, il envoie à l'ensemble des processus de son groupe le signal `SIGHUP` (Hangup = raccrocher).

11.0.5 Gestion interne des signaux

C'est dans le bloc de contrôle (BCP) de chaque processus que l'on trouve la table de gestion des signaux (attention, sous System V $<$ V.4, la table de gestion des processus est dans la `zone u`, c'est à dire dans l'espace-mémoire du processus).

Cette table contient, pour chaque signal défini sur la machine, une structure `sigvec` suivante :

```
{
    bit pendant;
```

```
    void (*traitement)(int);
}
```

En BSD et POSIX, on a un champ supplémentaire : `bit masque ;`
Le drapeau `pendant` indique que le processus a reçu un signal, mais n'a pas encore eu l'occasion de prendre en compte ce signal.

Remarque : comme `pendant` est un unique bit, si un processus reçoit plusieurs fois le même signal avant de le prendre en compte, alors il n'y a pas mémorisation des réceptions successives, un seul traitement sera donc réalisé.

Comme nous l'avons vu dans le graphe d'état des processus, la prise en compte des signaux se fait au passage de l'état actif noyau à l'état actif utilisateur. Pourquoi la prise en compte de signaux se fait-elle uniquement à ce moment là ?

Parce que
Une sauvegarde de la pile utilisateur et du contexte a été effectuée quand le processus est passé en mode noyau. Il n'est pas nécessaire de faire un nouveau changement de contexte. Il est facile pour traiter le signal de réaliser immédiatement une nouvelle augmentation de pile pour le traitement du signal, de plus la pile noyau est vide (remarque : en POSIX, il devient possible de créer une pile spéciale pour les fonctions de traitement de signaux).

L'appel à la fonction de traitement est réalisé de façon à ce qu'au retour de la fonction, le processus continue son exécution normalement en poursuivant ce qui était en cours de réalisation avant la réception du signal. Si l'on veut que le processus se poursuive dans un autre contexte (de pile), il doit gérer lui-même la restauration de ce contexte.

La primitive `longjmp` peut permettre de réaliser des changements de contexte interne au processus, grâce à un désempilement brutal.

Pendant ce changement d'état, la table de gestion des signaux du processus est testée pour la présence d'un signal reçu mais non traité (c'est un simple vecteur de bit pour le bit pendant, et donc testable en une seule instruction, ceci doit être fait rapidement comme le test de réception d'un signal est souvent réalisé).
Si un signal a été reçu (et qu'il n'est pas masqué), alors la fonction de traitement associée est réalisée. Le masquage permet au processus de temporiser la mise en øeuvre du traitement.

11.0.6 L'envoi de signaux : la primitive kill

```
kill(int pid, int sig)
```

Il y a NSIG signaux sur une machine, déclarés dans le fichier /usr/include/signal.h.
La valeur de `pid` indique le `PID` du processus auquel le signal est envoyé.

0 Tous les processus du **groupe** du processus réalisant l'appel `kill`

1 En système V.4 tous les processus du système sauf 0 et 1

pid positif le processus du pid indiqué

pid négatif tous les processus du groupe | pid |

le paramètre `sig` est interprété comme un signal si `sig` $\in$ [0-NSIG], ou comme une demande d'information si `sig` = 0 (suis-je autorisé à envoyer un signal à ce(s) processus ?). Comme un paramètre erroné sinon.

La fonction `raise(int signal)` est un raccourci pour `kill(getpid(), signal)`, le processus s'envoie à lui-même un signal.

Remarquez que l'on peut réécrire `kill(0, signal)` par `kill(-getpid(), signal)`. Rappel : les PID sont toujours positifs.

11.1 La gestion simplifiée avec la fonction `signal`

ZZZ : cette section est historique, utiliser la norme POSIX décrite plus loin.

```
ancien C : (*signal(sig, func))()
           int sig;
           int (*func)();

ANSI C :   void (*signal(int sig, void (*action)(int)))(int);
```

La fonction `signal` permet de spécifier ou de connaître le comportement du processus à la réception d'un signal donné, il faut donner en paramètre à la fonction le numéro du signal `sig` que l'on veut détourner et la fonction de traitement `action` à réaliser à la réception du signal.

Trois possibilités pour ce paramètre `action`

SIG_DFL Comportement par défaut, plusieurs possibilités

- **exit** Le processus se termine (avec si possible la réalisation d'un core)
- **ignore** Le processus ignore le signal
- **pause** Suspension du processus
- **continue** Reprise du processus si il était suspendu.

SIG_IGN le signal est ignoré.

Remarque : les signaux SIGKILL, SIGSTOP ne peuvent pas être ignorés.

HANDLER Une fonction de votre cru.

11.1.1 Un exemple

Exemple pour rendre un programme insensible à la frappe du caractère de contrôle intr sur le terminal de contrôle du processus.

```
void got_the_blody_signal(int n) {
    signal(SIGINT, got_the_blody_signal);
    printf(" gotcha!!  your  (%d) signal is useless \n");
}

main() {
    signal(SIGINT, got_the_blody_signal);
    printf(" kill me now !! \n");
    for(;;);
}
```

une version plus élégante et plus fiable :

```
    signal(SIGINT, SIG_IGN);
```

11.2 Problèmes de la gestion de signaux ATT

Les phénomènes suivants sont décrits comme des problèmes mais la norme POSIX permet d'en conserver certains, mais fournit aussi les moyens de les éviter.

1. un signal est repositionné à sa valeur par défaut au début de son traitement (handler).

```
#include <signal.h>

traitement()  {
    printf("PID %d en a capture un \n", getpid());
```

```
->     reception du deuxieme signal, realisation d'un exit
    signal(SIGINT, traitement);
}

main() {
    int ppid;
    signal(SIGINT,traitement);
    if (fork()==0)
    {/* attendre que pere ait realise son nice() */
        sleep(5);
        ppid = getppid(); /* numero de pere */
        for(;;)
            if (kill(ppid,SIGINT) == -1)
                exit();
    }
/* pere ralenti pour un conflit plus sur */
    nice(10);
    for(;;) pause();  <- reception du premier signal
/* pause c'est mieux qu'une attente active */
}
```

Si l'on cherche à corriger ce défaut, on repositionne la fonction `traitement` au début du traitement du signal. Ceci risque de nous placer dans une situation de dépassement de pile : en effet, dans le programme précédent, nous pouvons imaginer que le père peut recevoir un nombre de signaux arbitrairement grand pendant le traitement d'un seul signal, d'où une explosion assurée de la pile (il suffit en effet que chaque empilement de la fonction `traitement` soit interrompu par un signal)

```
traitement(){
    signal(SIGINT,traitement);
->   signal SIGINT
    printf("PID %d en a capture un \n",getpid());
}
```

On peut aussi ignorer les signaux pendant leur traitement, mais cela peut créer des pertes de réception.
Enfin, la solution BSD/POSIX où l'on peut bloquer et débloquer la réception de signaux à l'aide du vecteur de masquage (sans pour autant nous assurer de la réception de tous les signaux!!). De plus, en POSIX, le traitement d'un signal comporte une clause de blocage automatique. On indique quels signaux doivent être bloqués pendant le traitement du signal, grâce à un vecteur de masquage dans la structure `sigaction`.

Ceci est le comportement naturel de gestion des interruptions matérielles : on bloque les interruptions de priorité inférieure pendant le traitement d'un interruption.

2. Seconde anomalie des signaux sous System V $<$ V4 : certains appels systèmes peuvent être interrompus et dans ce cas la valeur de retour de l'appel système est -1 (échec). Il faudrait, pour réaliser correctement le modèle d'une interruption logicielle, relancer l'appel système en fin de traitement du signal. (Sous BSD ou POSIX, il est possible de choisir le comportement en cas d'interruption d'un appel système grâce à la fonction `siginterrupt`, c-a-d relancer ou non l'appel système, un appel à `read`, par exemple, peut facilement être interrompu si il nécessite un accès disque).

3. Troisième anomalie des signaux sous ATT : si un signal est ignoré par un processus endormi, celui-ci sera réveillé par le système uniquement pour apprendre qu'il ignore le signal et doit donc être endormi de nouveau. Cette perte de temps est dûe au fait que le vecteur des signaux est dans la zone u et non pas dans le bloc de contrôle du processus.

11.2.1 Le signal SIGCHLD

Le signal SIGCHLD (anciennement SIGCLD) est un signal utilisé pour réveiller un processus dont un des fils vient de mourir. C'est pourquoi il est traité différemment des autres signaux. La réaction à la réception d'un signal SIGCHLD est de repositionner le bit pendant à zéro, et d'ignorer le signal, mais le processus a quand même été réveillé pour cela. L'effet d'un signal SIGCHLD est donc uniquement de réveiller un processus endormi en priorité interruptible.

Si le processus capture les signaux SIGCHLD, il invoque alors la procédure de traitement définie par l'utilisateur comme il le fait pour les autres signaux, ceci en plus du traitement par défaut.

Le traitement normal est lié à la primitive `wait` qui permet de récupérer la valeur de retour (exit status) d'un processus fils. En effet, la primitive `wait` est bloquante et c'est la réception du signal qui va réveiller le processus, et permettre la fin de l'exécution de la primitive `wait`.

Un des problèmes de la gestion de signaux System V est le fait que le signal SIGCHLD est reçu (raised) au moment de la pose d'une fonction de traitement.

Ces propriétés du signal SIGCHLD peuvent induire un bon nombre d'erreurs.
Par exemple, dans le programme suivant nous positionnons une fonction de traitement dans laquelle nous repositionnons la fonction de traitement. Comme sous System V, le comportement par défaut est repositionné pendant le traitement d'un signal. Or le signal est levé à la pose de la fonction de traitement, d'où une explosion de la pile.

```
#include <stdio.h>
#include <unistd.h> /* ancienne norme */
#include <signal.h>

void hand(int sig) {
    signal(sig, hand);
    printf("message qui n'est pas affiche\n");
}

main() {
    if (fork()) { exit(0); /* creation d'un zombi */ }
    signal(SIGCHLD, hand);
    printf("ce printf n'est pas execute\n");
}
```

Sur les HP, un message d'erreur vous informe que la pile est pleine : `stack growth failure`.
Deuxième exemple :

```
#include <signal.h>
#include <sys/wait.h>

int pid, status;

void hand(int sig) {
    printf(" Entree dans le handler \n");
    system("ps -l");                /* affichage avec etat zombi du fils */
    if ((pid = wait(&status)) == -1) /* suppression du fils zombi */
    {
         perror("wait handler ");
         return ;
```

```
    }
    printf(" wait handler  pid: %d    status %d \n", pid, status);
    return;
}

main() {
    signal(SIGCHLD,hand);  /* installation du handler */
    if (fork() ==  0)
    {   /* dans le fils */
        sleep(5);
        exit(2);
    }
/* dans le pere */
    if ((pid = wait(&status)) == -1) /* attente de terminaison du fils */
    {
        perror("wait main ");
        return ;
    }
    printf(" wait main  pid: %d    status %d \n", pid, status);
}
```

résultat :

```
 Entree dans le handler
F S  UID   PID  PPID  C PRI NI     ADDR   SZ  WCHAN    TTY  TIME COMD
1 S  121  6792  6667  0 158 20  81ac180    6  49f5fc  ttys1 0:00 sigchld
1 S  121  6667  6666  0 168 20  81ac700  128 7ffe6000 ttys1 0:00 tcsh
1 Z  121  6793  6792  0 178 20  81bda80    0          ttys1 0:00 sigchld
1 S  121  6794  6792  0 158 20  81ac140   78  4a4774  ttys1 0:00 sh
1 R  121  6795  6794  4 179 20  81bd000   43          ttys1 0:00 ps
wait handler  pid: 6793    status 512     (2 * 256)
wait main: Interrupted system call
```

A la mort du fils, Le père reçoit le signal SIGCHLD (alors qu'il était dans le `wait` du main), puis le handler est executé, et `ps` affiche bien le fils zombi. Ensuite c'est le `wait` du handler qui prend en compte la terminaison du fils. Au retour du handler, l'appel a `wait` du main retourne -1, puisqu'il avait été interrompu par SIGCHLD.

11.3 Manipulation de la pile d'exécution

La primitive

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int indicateur);
```

sauvegarde un environnement d'exécution, c'est à dire un état de la pile, et si *indicateur* est non nul, sauvegarde le masque de signaux courant. La valeur de retour de cette fonction est zéro quand on fait une sauvegarde, et sinon dépend du paramètre *valeur* de la fonction `siglongjmp`.

```
int siglongjmp(sigjmp_buf env, int valeur);
```

La primitive `siglongjmp` permet de reprendre l'exécution à l'endroit sauvegardé par `sigsetjmp` dans la variable *env*.

Deux remarques : `env` doit avoir été initialisé par `sigsetjmp`, les valeurs de pile placées au-dessus de l'environnement repris sont perdues. L'environnement de pile doit encore exister dans la pile au moment de l'appel, sinon le résultat est indéterminé.

11.4 Quelques exemples d'utilisation

```
/*un exemple de signaux BSD */
#include <stdio.h>
#include <signal.h>
void gots1(int n)   { raise(SIGUSR2);  printf("got  s1(%d) ", n); }
void gots2(int n)   { printf("got  s2(%d) ", n); }

main()
{
    int mask ;
    struct sigvec s1,s2;

    s1.sv_handler = gots1;
    s1.sv_mask = sigmask(SIGUSR1);
    sigvec(SIGUSR1, &s1, NULL);

    s2.sv_handler = gots2;
    s2.sv_mask = sigmask(SIGUSR2);
    sigvec(SIGUSR2, &s2, NULL);

    printf(" sans masquage de SIGUSR2: ")
    raise(SIGUSR1);

    printf(" \n avec masquage de SIGUSR2: " );
    s1.sv_mask = sigmask(SIGUSR2);
    sigvec(SIGUSR1, &s1, NULL);

    raise(SIGUSR1);
}
```

Nous donne les affichages suivant :

```
 sans masquage de SIGUSR2: got  s2(31) got  s1(30)
 avec masquage de SIGUSR2: got  s1(30) got  s2(31)
```

Sous BSD, pas de fonction de manipulation propre des groupes de signaux (on regroupe les signaux par des conjonctions de masques).

Le problème de "l'interruption" des appels système par les signaux est corrigé par la fonction :

```
int  siginterrupt(int sig, int flag);
```

le drapeau `flag` prend comme valeur 0 ou 1, ce qui signifie que les appels systèmes interrompus par un signal seront :
soit relancés avec les mêmes paramètres.
soit retourneront la valeur -1, et dans ce cas la valeur de `errno` est positionnée à `EINTR`.

Certaines fonctions comme `readdir` utilisent des variables statiques, ces fonctions sont dites non réentrantes. Il faut éviter d'appeler ce type de fonctions dans un handler de signal, dans le cas où l'on fait déjà appel à la fonction dans le reste du processus. De la même façon la variable `errno` est unique. Si celle-ci est positionnée dans le `main` mais qu'un signal arrive avant son utilisation, une primitive appelée dans le handler peut en changer la valeur ! (ce problème de réentrance sera vu plus en détail avec les processus multi-activités).

11.4.1 L'appel pause

Fonction de mise en attente de réception d'un signal :

```
pause(void);
```

cette primitive est le standard UNIX d'attente de la réception d'un signal quelconque, BSD propose la primitive suivante :

```
sigpause(int sigmask)
```

qui permet l'attente d'un groupe spécifique de signaux, attention les signaux du masque sont débloqués (c.f. `sigprocmask`).

11.5 La norme POSIX

La norme POSIX ne définit pas le comportement d'interruption des appels systèmes, il faut le spécifier dans la structure de traitement du signal.

Les ensembles de signaux La norme POSIX introduit les ensembles de signaux :
ces ensembles de signaux permettent de dépasser la contrainte classique qui veut que le nombre de signaux soit inférieur ou égal au nombre de bits des entiers de la machine. D'autre part, des fonctions de manipulation de ces ensembles sont fournies et permettent de définir simplement des masques. Ces ensembles de signaux sont du type sigset_t et sont manipulables grâce aux fonctions suivantes :

```
int sigemptyset(sigset_t *ens)          /* raz */
int sigfillset(sigset_t *ens)           /* ens = { 1,2,...,NSIG} */
int sigaddset(sigset_t *ens, int sig)   /* ens = ens + {sig} */
int sigdelset(sigset_t *ens, int sig)    /* ens = ens - {sig } */
```

Ces fonctions retournent -1 en cas d'échec et 0 sinon.

```
int sigismember(sigset_t *ens, int sig);   /* sig appartient à ens ?*/
```

retourne vrai si le signal appartient à l'ensemble.

11.5.1 Le blocage des signaux

La fonction suivante permet de manipuler le masque de signaux du processus :

```
#include <signal.h>
int sigprocmask(int op, const sigset_t  *nouv, sigset_t *anc);
```

L'opération `op` :

SIG_SETMASK affectation du `nouv`eau masque, recupération de la valeur de l'`anc`ien masque.

SIG_BLOCK union des deux ensembles `nouv` et `anc`

SIG_UNBLOCK soustraction `anc - nouv`

On peut savoir si un signal est *pendant* et donc *bloqué* grâce à la fonction :

```
int sigpending(sigset_t *ens);
```

retourne -1 en cas d'échec et 0 sinon et l'ensemble des signaux pendants est stocké à l'adresse `ens`.

11.5.2 sigaction

La structure `sigaction` décrit le comportement utilisé pour le traitement d'un signal :

```
struct sigaction {
        void (*sa_handler) ();
        sigset_t sa_mask;
        int sa_flags;}
```

sa_handler fonction de traitement (ou `SIG_DFL` et `SIG_IGN`)

sa_mask ensemble de signaux supplémentaires à bloquer pendant le traitement

sa_flags différentes options

- **SA_NOCLDSTOP** le signal SIGCHLD n'est pas envoyé à un processus lorsque l'un de ses fils est stoppé.
- **SA_RESETHAND** simulation de l'ancienne méthode de gestion des signaux, pas de blocage du signal pendant le handler et repositionnement du handler par défaut au lancement du handler.
- **SA_RESTART** les appels système interrompus par un signal capté sont relancés au lieu de renvoyer -1. Cet indicateur joue le rôle de l'appel `siginterrupt(sig,0)` des versions BSD.
- **SA_NOCLDWAIT** si le signal est SIGCHLD, ses fils qui se terminent ne deviennent pas zombis. Cet indicateur correspond au comportement des processus pour SIG_IGN dans les versions ATT.

Le positionnement du comportement de reception d'un signal se fait par la primitive `sigaction`. L'installation d'une fonction de traitement du signal SIGCHLD peut avoir pour effet d'envoyer un signal au processus, ceci dans le cas où le processus a des fils zombis, c'est toujours le problème lié à ce signal qui n'a pas le même comportement que les autres signaux.
Un handler positionné par `sigaction` reste jusqu'à ce qu'un autre handler soit positionné, à la différence des versions ATT où le handler par défaut est repositionné automatiquement au début du traitement du signal.

```
#include <signal.h>
int sigaction(int sig,
              const struct sigaction *paction,
              struct sigaction       *paction_precedente);
```

Cette fonction réalise soit une demande d'information. Si le pointeur `paction` est null, on obtient la structure `sigaction` courante. Sinon c'est une demande de modification du comportement.

11.5.3 L'attente d'un signal

En plus de l'appel `pause`, on trouve sous POSIX l'appel `int sigsuspend(const sigset_t *ens)`; qui permet de réaliser de façons `atomique` les actions suivantes :

- l'installation du masque de blocage défini par `ens` (qui sera repositionné à sa valeur d'origine) à la fin de l'appel,
- mise en attente de la réception d'un signal non bloqué.

Chapitre 12

Les verrous de fichiers

Mécanismes de contrôle d'accès concurrents à un fichier, les verrous sont d'une grande utilité dans les applications de gestion et dans l'élaboration de bases de données partagées.
Les verrous sont rattachés aux **inœuds**. Ainsi toutes les ouvertures d'un même fichier, et à fortiori tous les descripteurs sur ces ouvertures, "voient" le verrou.
La protection réalisée par le verrou a donc lieu sur le fichier physique.
Un verrou est la **propriété** d'un seul **processus**, et seul le processus propriétaire du verrou peut le modifier ou l'enlever, attention le verrou ne protège pas contre les accès du processus propriétaire (attention à une situation multi-thread).

12.1 Caractéristiques d'un verrou

Les verrous sont définis par deux caractéristiques :

La portée : Ensemble des positions du fichier auxquelles le verrou s'applique. Cet ensemble est un intervalle, soit une portion du fichier

`[position1, position2]`

soit jusqu'à la fin du fichier

`[position1, fin de fichier[`

dans ce dernier cas si le fichier augmente, le verrou protège les nouvelles positions.

Le type : qui décrit les possibilités de cohabitation des différents verrous.

F_RDLCK partagé, plusieurs verrous de ce type peuvent avoir des portées non disjointes, par exemple les verrous $[80, 150]$ et $[100, 123]$

F_WRLCK exclusif, pas de cohabitation possible avec un autre verrou quelque soit son type.

12.2 Le mode opératoire des verrous

Le mode opératoire joue sur le comportement des primitives `read` et `write`. Les verrous d'un fichier sont soit **consultatifs**, soit **impératifs** (NON-POSIX) [1] .

Dans le premier mode **advisory** (consultatif), la présence d'un verrou n'est testée qu'à la pose d'un verrou, la pose sera refusée s'il existe un verrou de portée non disjointe et que l'un des deux verrous est exclusif.

[1]En effet le mode impératif n'est pas POSIX, et donc par défaut n'est pas mise en oeuvre sur les disque sous linux.

Dans le second mode **mandatory**, la présence de verrous est testée pour la pose mais aussi pour les appels systèmes `read` et `write`.

Dans le mode consultatif, les verrous n'ont d'effet que sur les processus jouant effectivement le jeu, c'est-à-dire, posant des verrous sur les zones du fichiers sur lesquels ils veulent réaliser une lecture (verrou partagé) ou une écriture (verrou exclusif).

Dans le mode impératif, les verrous ont un impact sur les lectures/écritures de tous les processus :
- sur les verrous de type partagé (F_RDLCK), toute tentative d'écriture (appel système write) par un autre processus est bloquée ;
- sur les verrous de type exclusif (F_WRLCK), toute tentative de lecture ou d'écriture (read et write) par un autre processus est bloquée.

Pour rendre l'utilisation impérative il faut sous **linux** monter le disque avec l'option `-o mand`. Puis il faut utiliser la commande chmod pour positionner le SETGID bit, soit chmod g+s fichier en shell soit la même chose en C : si l'on a le descripteur `d` d'une ouverture sur le fichier

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
...
struct stat buf
fstat(d, &buf);
fchmod(d,buf.st_mode |  S_ISGID);
```

même chose avec stat et chmod si l'on la référence du fichier.

12.3 Manipulation des verrous

La structure de verrou `flock` :

```
struct flock  {
      short    l_type;      /* F_RDLCK, F_WRLCK,F_UNLCK */
      short    l_whence;    /* SEEK_SET,SEEK_CUR,SEEK_END */
      off_t    l_start;     /* position relative a l_whence */
      off_t    l_len;       /* longueur de l'intervalle */
      pid_t    l_pid;       /* PID du processus propriétaire */
};
```

le champ `l_type`

F_RDLCK verrou partagé

F_WRLCK verrou exclusif

F_UNLCK déverrouillage

Les manipulations de verrous se font avec la primitive `fcntl`, c'est-à-dire par le biais d'un descripteur. Pour poser un verrou partagé, ce descripteur doit pointer sur une ouverture en lecture. De même, il faut un descripteur sur une ouverture en écriture pour un verrou de type exclusif
.

Pour décrire la portée du verrou que l'on veut poser, on utilise la même syntaxe que pour la primitive `lseek`, le début de l'intervalle est **whence+l_start** :
`l_whence` = SEEK_SET $\longrightarrow$ whence = 0
`l_whence` = SEEK_CUR $\longrightarrow$ whence = offset courrant
`l_whence` = SEEK_END $\longrightarrow$ whence = taille du fichier.

La longueur du verrou est définie par le champ `l_len`. Si cette valeur est nulle, le verrou va jusqu'à la fin du fichier (même si le processus change cette fin). Remarque : il est possible de poser un verrou dont la portée est supérieure à la taille du fichier.

Le champ `l_pid` contient le pid du processus propriétaire du verrou, ce champ est rempli par `fcntl` dans le cas d'un appel consultatif (`F_GETLK`).

12.4 Utilisation de `fcntl` pour manipuler les verrous

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int desc, int commande, struct flock *verrou);
```

`fcntl` retourne 0 en cas de succès, ou -1 en cas d'echec.

Trois commandes possibles :

F_SETLK pose non bloquante

si il existe un verrou incompatible, `errno` a pour valeur EAGAIN

si l'on n'a pas les droits d'accès sur le fichier pour le type de verrou demandé, alors `errno` a pour valeur EACCES ;

si la pose du verrou crée une situation d'interblocage, alors `errno` a pour valeur EDEADLK.

F_SETLKW pose bloquante (Wait)

succès immédiat si il n'y a pas de verrou incompatible, ou succès une fois les verrous incompatibles levés.

si l'appel est interrompu, `errno` a pour valeur EINTR

si une situation d'interblocage est détectée, alors `errno` a pour valeur EDEADLK.

F_GETLK Test d'existence d'un verrou incompatible avec le verrou passé en paramètre (retour -1 sur des paramètres incorrects)

si il existe un tel verrou incompatible, alors la structure flock passée en paramètre est remplie avec les valeurs de ce verrou incompatible. Le champ `l_pid` indique alors l'identité du processus propriétaire de ce verrou incompatible.

sinon, la structure flock reste inchangée excepté le champ type qui contient F_UNLCK.

Attention, après un test d'existence qui nous informe de l'absence de verrou incompatible, nous ne sommes pas assuré qu'au prochain appel la pose de ce verrou soit possible, en effet un autre processus a peut-être posé un verrou incompatible entre-temps (cf. interblocages chapitre 13).

Chapitre 13

Algorithmes Distribués & Interblocages

Ce chapitre introduit les problèmes liés à la gestion de processus concurrents. Le problème à resoudre est le partage de ressources entre différents processus asynchrones. Les I.P.C. et les verrous sont deux types d'outils permettant le partage asynchrone de ressources entre processus.

13.1 exemples

13.1.1 Les méfaits des accès concurrents

L'exemple le plus simple est une variable entière partagée par deux processus ou threads, ou bien manipulé par une fonction asyncrone comme un handler de signal. Supposont que l'on définise deux fonctions de manipulation de la variable :

```
int getValue();
void setValue(int );
```

Pour incrémenter la variable, il suffit d'exécuter `setValue(getValue()+1) ;` mais décomposont en `int tmp=getValue() ; setValue(tmp+1) ;` ce qui ne change pas grand chose `tmp` étant s'implement allouée sur la pile dans le premier cas. Regardont l'éxecution suivant du code par deux thread :

```
int tmp1=getValue();
                    | int tmp2= getValue();
setValue(tmp1+1);
                    | setValue(tmp2+1);
```

Que c'est il passé ?
la variable n'a été incrémentée qu'une fois !

Le cas d'un signal le code c de prog

```
#include <stdio.h>
#include <signal.h>
int nbi; // nbre d'incrementation accès atomique
int partage; // nbr d'incrémentation  accès non atomique

int getValue() { return partage; }
```

```
void setValue(int x) { partage = x; }

void handler(int s)
{
int tmp=getValue();
setValue(tmp+1);
nbi++;
}


int
main(int c, char *argv[])
{
struct sigaction sig;
long diff = 0;
sig.sa_handler= handler;
sig.sa_flags = 0;

sigaction(SIGUSR1, &sig, NULL);
fprintf(stderr,"sigusr1= %d\n",SIGUSR1);
fprintf(stderr,"%d %d\n", nbi,partage);

for(;;)
{
        int tmp=getValue();
        setValue(tmp+1);
        nbi++;
if ((partage != nbi) && (diff!= nbi-partage))
        { diff = nbi-partage; fprintf(stderr,"%d\n", nbi-partage); }
}
}
```

13.1.2 Exclusion mutuelle

Problème : il y a une rivière que l'on peut traverser par un gué fait de pierre alignées, où il n'est pas possible de se croiser, et il n'est pas possible de faire demi-tour. Comment doit-t-on organiser le passage ?

Solutions :

1. regarder avant de traverser
2. si deux personnes arrivent en même temps sur chaque rive,
   si elles avancent en même temps $\longrightarrow$ interblocage
   si elles attendent en même temps $\longrightarrow$ interblocage
3. Un remède : un côté prioritaire $\longrightarrow$ famine. En effet si le coté OUEST est prioritaire et qu'un flot continu de personnes arrive de ce côté, les personnes à l'EST sont bloquées indéfiniment.
4. Une solution : alterner les priorités.

Pour des ressources système comme les fichiers, le partage n'est pas géré par le SGF. Il faut donc un mécanisme de partage : les verrous, qui permettent un partage dynamique et partiel (portions de fichiers). Pour un partage entre utilisateurs, on utilise plutôt des outils comme `SCCS, RCS`.

13.2 Mode d'utilisation des ressources par un processus.

Formalisons les opérations réalisables sur une ressource.
– requête : demande bloquante de ressources
– utilisation : lecture/écriture sur la zone verrouillée
– libération : verrou L-type

13.3 Définition de l'interblocage (deadlock)

Un ensemble de processus est en **interblocage** si et seulement si tout processus de l'ensemble est en attente d'un évènement qui ne peut être réalisé que par un autre processus de l'ensemble.

Exemple :

Le processus A possède un verrou de portée [0,400] sur un fichier f, et demande un verrou de portée [800,1000] sur ce même fichier, alors qu'un processus B possède un verrou de portée [600,900] sur le fichier f et demande un verrou de portée [0,33] sur f. Les deux processus sont en interblocage. Dans le cas de la pose de verrous sous UNIX, il y a détection de cet interblocage et la commande `fcntl` échoue.

13.4 Quatre conditions nécessaires à l'interblocage.

Les conditions suivantes sont **nécessaires** pour avoir une possibilité d'interblocage.

Exclusion mutuelle les ressources ne sont pas partageables, un seul processus à la fois peut utiliser la ressource.

Possession & attente il doit exister un processus qui utilise une ressource et qui est en attente sur une requête.

Sans préemption les ressources ne sont pas préemptibles c'est-à-dire que les libérations sont faites volontairement par les processus. On ne peut pas forcer un processus à rendre une ressource. (Contre exemple : le CPU sous Unix est préemptible)

Attente circulaire il doit exister un ensemble de processus P_i tel que P_i attend une ressource possédée par P_{i+1}.

Les quatre conditions sont nécessaires pour qu'une situation d'interblocage ait lieu.
Exercice : montrer que pour les verrous, les quatre conditions tiennent.
Exercice : montrer que si l'une des condition n'est pas vérifiée alors il ne peut y avoir d'interblocage.

13.5 Les graphes d'allocation de ressources

Les graphes d'allocation de ressources permettent de décrire simplement les problèmes d'interblocage.
`G = (N,T) N = P U R`
P : ensemble des processus
R : ensemble des ressources

T est inclus dans RXP U PXR
Soit le couple (x,y) appartenant à T,
si (x,y) appartient à RXP, cela signifie que la ressource x est utilisée par le processus y.
si (x,y) appartient à PXR, cela signifie que le processus x demande la ressource y.

Chapitre 14

Sécurité et Sûreté de fonctionnement

Comme pour une habitation il faut que votre système offre deux chose importante, d'une part que la vie dans l'habitation soit sur, pas de risque pour les utilisateurs ni pour les éléments matériel. Feux , indondation, enfermement, tremblement de terre etc. C'est la sûreté de fonctionnement.
D'autre par l'habitation est protégée ainsi que ses habitants contre des attaques plus ou moins malveillantes. La porte du jardin est fermée pour éviter que des animaux détériore le jardin. L'habitation est munie de systéme de vérouillage pour se proteger contre un cambriollage. C'est la sécurité.

La sûreté de fonctionnement est un élément statégique qui doit être gérer par la direction informatique en fonction de contraintes opérationnelles. La sécurité est le problème de tout le monde. Pour que la sécurité fonctionne, il faut que toutes les personnes ayant un accès à une ressource soient conscient du degré de sécurité associé à la ressource. La stratégie de sécurité doit bien sur être définie par la direction informatique mais c'est un travail collectif (installation d'une sérure n'a pas d'effet si tout le monde laisse la porte ouverte).

14.1 Protection des systèmes d'exploitation

Sécuriser un système, c'est protéger ce système contre un fonctionnement imprévu ou défectueux.

Il peut s'agir :
- d'erreurs de programmation (d'un utilisateur, ou du système lui-même) qui se propagent au système (du fait de contrôles insuffisants ou mal effectués).
- d'un mauvais fonctionnement du matériel.
- enfin, d'un opérateur, concepteur ou réalisateur malveillant ou peu scrupuleux (quand il s'agit d'informations financières !).

Le recensement des opérations frauduleuses aux Etats-Unis au cours d'une année a donné 339 cas de fraude, pour un coût d'un milliard de francs.

La protection des sites a également un coût très important (temps et complexité), d'où des systèmes de protection qui résultaient d'un compromis coût/efficacité.

Le coût en ressources de la protection étant resté stationnaire, les systèmes et les machines actuelles plus rapides ont rendu ce coût moins prohibitif.
L'idée d'un système de protection est de traiter les différents types de problèmes de manière générale et unitaire.

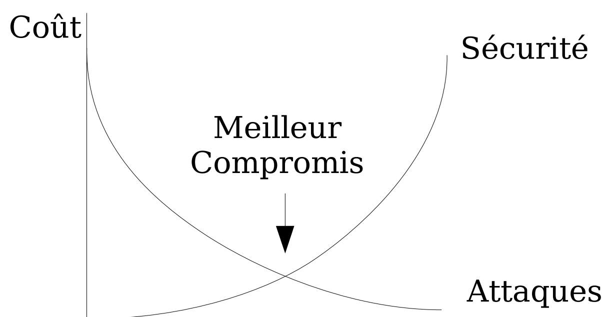

Fig. 14.1 – Un compromis entre le cout d'une attaque et celui de la sécurité

Implantés seuls, les dispositifs de protection coûtent cher.

Heureusement, si ces dispositifs permettent d'augmenter les performances du logiciel, dans des domaines comme celui de la fiabilité ou de la résistance aux erreurs, leur coût relatif diminue. Si, de plus, ces dispositifs permettent une gestion des ressources partagées plus facile et plus sûre, ils peuvent devenir compétitifs d'un point de vue commercial.

Il est difficile de définir précisément ce que l'on entend par protection d'un système d'exploitation (et d'information en général), tant les facteurs qui peuvent influer sur cette notion (humains, sociaux, économiques), sont nombreux. On peut dire cependant que la protection se rapporte à tout ce par quoi l'information peut être modifiée, divulguée ou détruite. Dans certains cas, la gestion du trafic aérien par exemple, elle peut être la garantie des performances du système. La confidentialité d'enregistrements financiers, médicaux ou personnels relève aussi de la protection, comme le fait qu'un processus utilisateur ne puisse être exécuté en mode système. La protection exige enfin la correction des processus système.

- pérennité du système
- confidentialité des données (système, utilisateur, etc.)
- correction du système

A l'opposé, nous ne parlerons pas de :

- protection physique de l'ordinateur (feu, vol, coupures, etc.)
- malveillance ou incompétence de l'opérateur (il est éventuellement possible de limiter soigneusement les privilèges du super-utilisateur afin de préserver le système).

Le degré de protection du système dépend de deux facteurs :

- le degré de protection des informations qu'il manipule
- le degré de confiance en ses logiciels, en particulier le système d'exploitation.

Un logiciel est fiable quand il satisfait correctement ses spécifications et quand, de plus, il est capable de résister à un environnement imprévu (données erronées, pannes, etc.), soit en corrigeant l'anomalie, soit en la signalant, mais en évitant que les erreurs ne se propagent et ne contaminent le système tout entier.

La protection, l'intégrité et l'authenticité des données qui transitent dans un système d'information sont réalisées par les systèmes cryptographiques (ATHENA et Kerberos au MIT).

Le confinement des erreurs est obtenu en contrôlant les accès aux entités du système d'exploitation, par les domaines de protection.

14.2 Généralités sur le contrôle d'accès

Contrôle très précis de l'utilisation des ressources par les processus.

Sujets \ Objets	Fichier 1	Segment 1	Segment 2	Processus 2	Editeur
Processus 1	Lire	Executer	Lire Ecrire		Entrer
Processus 2	Lire Ecrire				Entrer
Processus 3		Lire Ecrire Executer		Entrer	Entrer

Fig. 14.2 – Matrice d'accès

Deux niveaux :
- un niveau logique (soft), celui du modèle de protection, ensemble de règles qui définissent quels accès (aux ressources) sont autorisés et quels accès sont interdits. Ces règles sont définies soit à la conception du système, soit par les utilisateurs.
- un niveau matériel qui permet d'appliquer le modèle réellement. C'est le rôle des mécanismes de protection.

Le premier doit être dynamique. Par contre, le deuxième doit être stable pour faciliter l'implémentation, le contrôle et la fiabillisation.
Les deux doivent de surcroît être indépendants du modèle pour offrir un vaste ensemble de règles possibles.
Un exemple de protection simple est celui des répertoires sous unix, pour éviter qu'ils soit corompus (ou que l'arborescence soit corompue), il sont identifiés comme des fichiers spéciaux et il n'est possible d'y accèder que par le truchement d'appels systèmes spécifiques. Biensur il est toujours possible de les manipuler si l'on peut accèder en mode **raw** au disque dur mais cette option est réservée au super utilisateur.

14.2.1 Domaines de protection et matrices d'accès

On formalise le système comme un ensemble d'entités actives, les sujets, un ensemble d'entités accessibles, les objets. Le modèle de protection définit quels sujets ont accès à quels objets et comment (modalités d'accès).

On parle alors de droit d'accès, définis par le couple (objet, modalités)

Exemple : (fichier, lire)

Le modèle doit fixer à tout instant les droits d'accès dont dispose chaque processus. Cet ensemble de droits est le domaine de protection du processus. Voire un exemple de matrice d'accès dans la figure 14.2

14.2.2 Domaines de protection restreints

Il est souhaitable que la matrice d'accès puisse évoluer dynamiquement. En effet, un même processus peut avoir, au cours de son existence, des besoins variables afin que chaque module

qui compose un processus ne mette pas en danger des ressources non utilisées. Par exemple : un module de lecture de données, un module de calcul, un module d'impression. On va donc exécuter chaque module dans un domaine de protection le plus réduit possible.

C'est le *principe du moindre privilège* : un programme ne peut endommager un objet auquel il n'a pas accès !

Pour mettre en place ces domaines dynamiques, une possibilité est de changer les droits d'accès du processus au cours de son exécution. Une autre possibilité est d'ajouter aux objets le type "domaine" et de contrôler les accès à la matrice. L'édition de cases de la matrice devient une opération protégée.

14.2.3 Avantages des domaines de protections restreints

Avantages de cette souplesse :

- le maillon faible : un système rigide laisse souvent des "poternes" (portes dérobées) pour pouvoir implémenter certaines opérations ;
- si les mesures de protection sont trop pesantes, l'expérience prouve que l'on crée souvent des moyens "exceptionnels" pour les contourner ;
- il est intéressant de faire varier les contrôles suivant les utilisateurs ;
- on peut réaliser des accès à la carte sur certains objets ;
- enfin, certains problèmes de protection nécessitent des mesures souples, ce sont : "le cheval de Troie" et le confinement.

14.3 Le cheval de Troie

Un utilisateur fait souvent appel à un certain nombre de programmes qu'il n'a pas écrit lui-même (heureusement), un éditeur par exemple. Ce programme peut être un cheval de Troie : il va profiter des droits donnés par l'utilisateur pour consulter, copier, modifier ou altérer des données auxquelles il n'est pas censé accéder.

14.4 Le confinement

Le problème ici est tout simplement le fait que le programme ne manipule pas de données de l'utilisateur mais simplement enregistre ses paramètres d'appels (les utilisateurs à qui vous envoyez du courrier par exemple). Le problème du confinement est donc de vous protéger contre ce type d'extraction d'informations (ce qui peut par exemple être utilisé en bourse pour connaitre votre comportement d'achat).

14.5 les mécanismes de contrôle

Accès hiérarchiques

UNIX (4)/ MULTICS (8) / VMS

Listes d'accès

UNIX/MULTICS

Capacités

Les capacités sont des triplets (UTILISATEUR, DROITS, POINTEUR). La manipulation des capacités est réalisée de façon protégée. Le pointeur n'est pas directement utilisable par l'utilisateur de la capacité. La capacité donne le droit d'accès à certains utilisateurs d'une certaine ressource. Pour qu'un autre utilisateur puisse utiliser votre ressource, vous devez lui donner une capacité.

Changer de protection revient à changer de C-liste.

La notion de domaine se matérialise par une simple indirection sur une autre C-liste.

Comme les capacités donnent un accès sans contrôle aux objets, la protection des capacités doit être absolue. Elle est donc réalisée de façon matérielle.

Sujets \ Objets	Fichier 1	Segment 1	Segment 2	Processus 2	Editeur	Domaine 1	Domaine 2
Domaine 1	Lire	Executer	Lire Ecrire		Entrer	Entrer	Entrer
Domaine 2	Lire Ecrire				Entrer		
Domaine 3		Lire Ecrire Executer		Entrer	Entrer	Entrer	

Processus i ⟶ Domaine i

FIG. 14.3 – Matrice d'accès

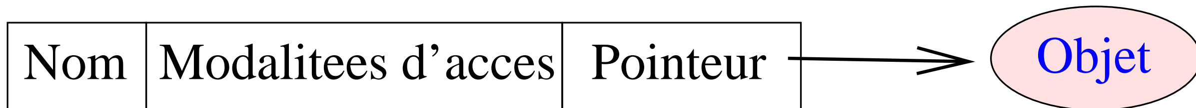

FIG. 14.4 – Une capacité

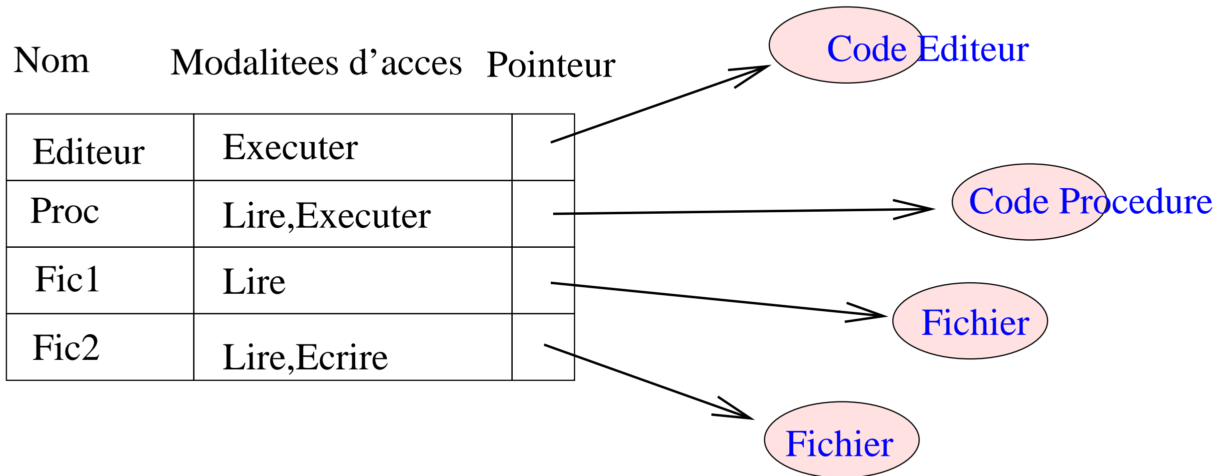

FIG. 14.5 – Une liste de capacités

14.5.1 Application des capacités au domaines de protection restreints

Les C-listes sont des objets d'un type n'ayant qu'un droit d'entrée, la C-liste contenant le droit réel.

Cette technique sur les C-listes permet d'implanter facilement le principe de moindre privilège.

Les mécanismes d'accès mémoire modernes permettent aisément de réaliser les capacités.

Un problème important est la révocation

En effet, une fois que vous avez donné une capacité, l'accès est définitivement donné. Pour régler ce problème, on ne fournira pas la capacité d'accès à un objet mais à un domaine, et on détruira ce domaine si l'on veut de nouveau interdire l'accès à l'objet. On crée deux capacités en chaine et l'on détruit celle que l'on possède quand ont veut retirer l'accès.

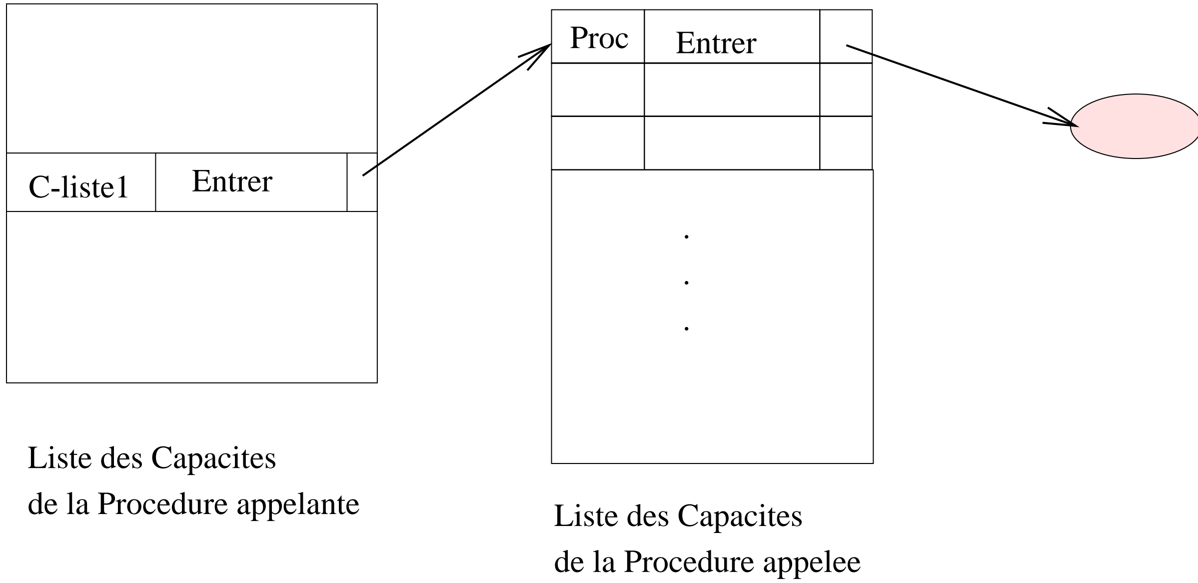

FIG. 14.6 – Changement du domaine de protection

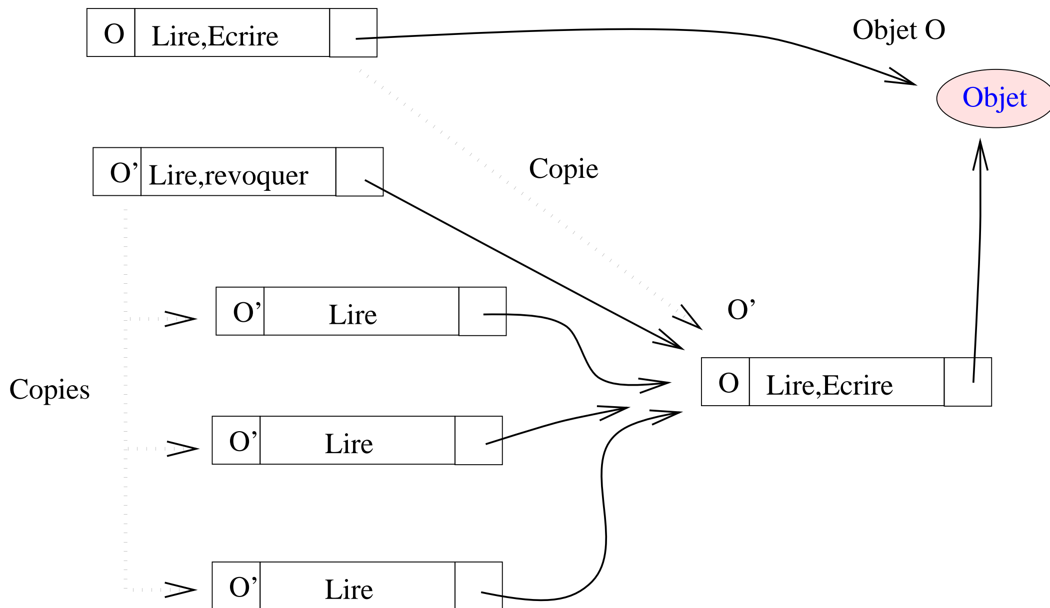

FIG. 14.7 – Transmission d'une capacité

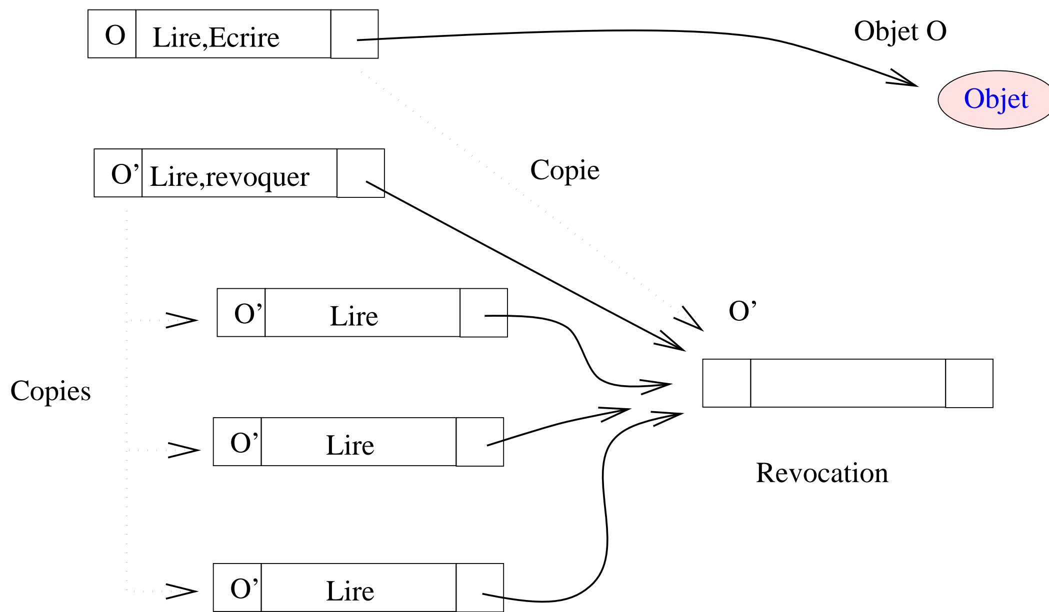

Fig. 14.8 – Révocation d'une capacité

14.6 Les ACL

Les ACL (access control lists) sont une extension des modes de protection standard d'UNIX. Les ACL sont des droits que l'on définit en plus des 9 bits de protection classiques, ils permettent en particulier d'autoriser l'accès ou de le refuser, à un utilisateur donné, ou à un groupe donné.

Deux commandes permettent de manipuler les ACL, ce sont `chacl` et `lsacl`.
La syntaxe de la commande shell `chacl` :

```
chacl '(dr.staff,r-x)(zipstein.%,r-x)(%.licence,---)' proj
```

qui donne sur le fichier `proj` les droits de lecture et d'écriture à l'utilisateur `dr` du groupe `staff` et à l'utilisateur `zipstein` quelque soit son groupe et qui refuse cet accès aux utilisateurs du groupe `licence`.

```
chacl '(binome.%,rwx)(%.@,--x)(%.%,---)' catalogue_projet
```

qui donne le droit d'accès total à l'utilisateur `binome` (quelque soit son groupe), permet le parcours du répertoire aux membres du groupe propriétaire et refuse l'accès à tous les autres utilisateurs.

Deux symboles spéciaux :

`%` pour n'importe qui (utilisateur ou groupe)

`@` pour le propriétaire ou le groupe propriétaire

On retrouve aussi les autres syntaxes de `chmod` par exemple :
`chacl %.%=r fichier`
ou
`chacl @.%=5 fichier`

Attention les acl sont détruits par la commande `chmod` et la commande `chacl` ne permet pas de positioner les autres bits définis dans l'inode ; seuls les 9 bits de protections sont positionnables par `chacl`.

Pour positionner les droits standard et des acl, il faut donc réaliser en succession un `chmod` puis un `chacl`.

On utilisera :
`chacl '(prof.%,rwx)' catalogue_projet`
pour les projets de C ou de système.

La commande `lsacl [fichiers]` permet de connaître les acl associés aux fichiers, remarquer qu'à l'inverse de `/bin/ls` cette commande n'a pas de paramètres par défaut.

14.6.1 Appels systemes `setacl` et `getacl`

On trouvera deux appels systèmes correspondant :

```
#include <sys/acl.h>

int setacl(
     const char *path,
     size_t nentries,
     const struct acl_entry *acl
);

int fsetacl(
     int fildes,
     size_t nentries,
     const struct acl_entry *acl
```

```
);
```

Un bon exercice : récrire `lsacl` de façon qu'il fonctionne d'une manière similaire à `/bin/ls`. Utilisation de la commande script pour montrer le comportement des acl.

```
Script started on Fri May  5 10:33:20 1995
$ lsacl *
(dr.%,rw-)(%.staff,---)(%.%,---) fich
(dr.%,rw-)(%.staff,---)(%.%,---) file
(dr.%,rwx)(%.staff,---)(%.%,---) projet
$ chacl  '(prof.%,rwx)' fich
$ lsacl *
(prof.%,rwx)(dr.%,rw-)(%.staff,---)(%.%,---) fich
(dr.%,rw-)(%.staff,---)(%.%,---) file
(dr.%,rwx)(%.staff,---)(%.%,---) projet
$ chacl '(%.staff,rx)' fich
$ lsacl *
(prof.%,rwx)(dr.%,rw-)(%.staff,r-x)(%.%,---) fich
(dr.%,rw-)(%.staff,---)(%.%,---) file
(dr.%,rwx)(%.staff,---)(%.%,---) projet
$ chacl '(illouz.staff=' fich
$ lsacl fich
(illouz.staff,---)(prof.%,rwx)(dr.%,rw-)(%.staff,r-x)(%.%,---) fich
$ chacl '(prof.%,rx)' . ..
$ su prof
Password:
$ cat fich
$ touch fich
$ chacl '(dr.staff,x)' fich
chacl: file "fich": Not owner (errno = 1)
$ lsacl *
(illouz.staff,---)(prof.%,rwx)(dr.%,rw-)(%.staff,r-x)(%.%,---) fich
(dr.%,rw-)(%.staff,---)(%.%,---) file
(dr.%,rwx)(%.staff,---)(%.%,---) projet
$ exit # du su
$ exit # du script

script done on Fri May  5 10:37:18 1995
```

14.6.2 Autres pistes sur la sécurité

- crack et autres logiciels d'attaque de mots de passe
- root-fix et autres Pots de Miel
- virus et logiciels antivirus (sous linux ? ? ? ?)
- Gestion des mots de passe
- Honey Pot
- Haute disponibilité
  - Redondance
  - RAID (cf chapitre)
  - Distribution de disques
  - SAN
  - Clusters
  - keep alive
- Sauvegardes et systèmes de sauvegarde

- Les attaques par déni de services

Chapitre 15

Multiplexer des entrées-sorties

15.1 Gerer plusieurs cannaux d'entrée sortie

Dans ce chapitre, nous voulons présenter le problème des attentes actives sur plusieurs descripteurs.

L'exemple le plus fréquent est celui d'un serveur web, le serveur doit gérer simultanément un très grand nombre de flux d'entrée et de flux de sortie et de flux de contrôle (les information de contrôle des sockets).

15.1.1 Solution avec le mode non bloquant

Il est possible d'utiliser des entrée-sorties non bloquantes mais c'est loing d'être la solution optimal car notre processus vas réaliser de nombreux appels système inutile d'autant plus si dans le cas d'un serveur avec des comportements de clients très alléatoires. Le coût en ressources de cette attente active est extrêmement cher, et doit être évité dans le cas d'une machine en temps partagé.

15.1.2 Utiliser les mécanismes asynchrones

On peut utiliser des entrées-sorties asynchrones et demander au noyau de nous prévenir par un signal qui informe de l'arrivée de données sur un descripteur. Ce signal est `SIGIO`, mais ce n'est valable que sur les descripteurs qui sont des périphériques. De plus ce mécanisme ne désigne pas le descripteur sur lequel s'est faite l'arrivée de caractères, d'où de nouvelles pertes de temps dûes aux appels réalisés inutilement en mode non bloquant.

15.2 Les outils de sélection

La solution la plus efficase vient de systèmes de sélection qui prend un paramètre un ensemble de descripteurs, et qui permet tester si l'un de ses descripteurs est près à satisfaire un appel système `read` ou `write`. Cet appel est bloquant jusqu'à l'arrivée de caractères sur un des descripteurs de l'ensemble. Ainsi il n'y pas de consomation de ressource processus inutile, le travail est fait à un niveau plus bas (dans le noyau) de façon plus économique en ressources.

15.2.1 La primitive `select`

La première implémentation d'un outil de selection sous Unix est l'appel système select, malheureusement sa syntaxe est devenu inadapté pour cituations ou le nombre de descipteur utilisé par le programme est très grand ce qui peut arriver facilement avec un serveur de fichier. Nous fournissont à la primitive `select` :

- Les descripteurs que nous voulons scruter. (l'indice du plus grand descripteur qui nous intéresse dans la table des descripteurs du processus)
- Les conditions de réveil sur chaque descripteur (en attente de lecture, écriture, évènement ?)
- Combien de temps nous voulons attendre.

La fonction retourne pour chaque descripteur s'il est prêt en lecture, écriture, ou si l'évènement a eu lieu, et aussi le nombre de descripteur prêts. Cette information nous permet ensuite d'appeler `read` ou `write` sur le(s) bon(s) descripteur(s).

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

int select(int maxfd,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds
           struct timeval *delai);
```

Retourne le nombre de descripteurs prêts, 0 en cas d'expiration du délai.

Paramétrage du délai :

```
struct timeval {
    long tv_sec;
    long tv_usec;
};
```

`delai == NULL` Bloquant, attente infinie

`delai->tv_sec == 0 && delai->tv_usec == 0` Non bloquant, retour immédiat.

`delai->tv_sec > 0 && delai->tv_usec >0` Semi bloquant, attente jusqu'à ce qu'un descripteur soit prêt ou que le délai en secondes plus microsecondes soit écoulé.

Les trois pointeurs (`readfds`, `writefds`, et `exceptfds`) sur des ensembles de descripteurs sont utilisés pour indiquer en entrée les situations qui nous intéressent. C'est à priori (cela peut varier avec l'implémentation) des tableaux de bits avec un bit pour chaque descripteur du tableau de descripteurs du processus. L'entier `maxfd` est la position du dernier bit significatif de ce tableau de bits.

Les seules façons de manipuler ces ensembles de descripteurs sont :

- Allocation :`fd_set *fd=(fd_set*)malloc(sizeof(fd_set));`
- Création
- Affectation
- Utilisation d'une des quatre macros suivantes :

  `FD_ZERO(fd_set fdset)` RAZ de l'ensemble.

  `FD_SET(int fd, fd_set *fdset)` Positionne le bit fd a 1.

  `FD_CLR(int fd, fd_set *fdset)` Positionne le bit fd à 0

  `FD_ISSET(int fd, fd_set *fdset)` vrai si le bit fd est à 1 dans l'ensemble.

Un descripteur est considéré comme prêt en lecture si un appel `read` dessus ne sera pas bloquant. De même, un descripteur est considéré comme prêt en écriture si un appel `write` ne sera pas bloquant. Les exceptions / évènements sont définis pour les lignes de communication qui acceptent les *messages hors bande* comme les `sockets` en mode datagramme.

15.2.2 La primitive `poll`

La primitive `poll` fournit un service proche de `select` avec une autre forme d'interface. Cette interface est adaptée quand le nombre de descripteurs ouvert par le processsus est très grand mais que l'on ne s'intérèsse qu'a un petit nombre de ceux-ci.

```
#include <stropts.h>
#include <poll.h>
int poll(struct pollfd fdarray[],
         unsigned long nfds,
         int           timeout
  );

struct pollfd {
        int   fd;
        short events;
        short revents;
};
```

Ici on spécifie la liste de descripteurs (dans un tableau) et ce que l'on veut gèter sur chacun d'eux.

La valeur de retour est -1 en cas d'erreur, 0 si le temps d'attente `timeout` est écoulé, ou un entier positif indiquant le nombre de descripteurs pour lesquels la valeur du champ `revents` a été modifiée.

Les évènements sont ici :
Pour les évènements de `events` :

POLLIN Données non prioritaire peuvent être lues.

POLLPRI Données prioritaire peuvent être lues.

POLLOUT Données non prioritaire peuvent être écrites, les messages de haute priorité peuvent toujours êtres écrits.

Pour les `revents` (valeurs de retour de la primitive `poll`) :

POLLIN,POLLPRI les données sont là.

POLLOUT l'écriture est possible

POLLERR Une erreur a eu lieu.

POLLHUP La ligne a été coupée.

POLLNVAL Descripteur invalide.

Le mode de blocage de la primitive `poll` dépend du paramètre `timeout`

timeout == INFTIM Bloquant, INFTIM est défini dans `stropts.h`.

timeout == 0 Non bloquant.

timeout > 0 Semi bloquant, attente de `timeout` micro secondes.

Un Exemple Attente de données sur ifd1 et ifd2, de place pour écrire sur ofd, avec un délai maximum de 10 seconds :

```
#include <poll.h>
struct pollfd fds[3] ;
int ifd1, ifd2, ofd, count ;

fds[0].fd = ifd1 ;
fds[0].events = POLLIN ;
fds[1].fd = ifd2 ;
fds[1].events = POLLIN ;
fds[2].fd = ofd ;
```

```
fds[2].events = POLLOUT ;
count = poll(fds, 3, 10000) ;
 if (count == -1) {
        perror("poll failed") ;
         exit(1) ;
 }
 if (count==0)
       printf("Rien \n") ;
 if (fds[0].revents & POLLIN)
       printf("Données a lire sur ifd%d\n", fds[0].fd) ;
 if (fds[1].revents & POLLIN)
       printf("Données a lire sur ifd%d\n", fds[1].fd) ;
 if (fds[2].revents & POLLOUT)
       printf("De la place sur fd%d\n", fds[2].fd) ;
```

15.2.3 Le périphérique poll

Dans le cas de serveur travaillant avec un très grand nombre de descripteurs (plueirus dizaine de milliers de descripteurs) les deux syntaxes `poll` et `select` sont inéfficaces. Soit dans le cas de`select` car le nombre de descripteurs scrutés par le noyau est très grand alors qu'un très faible par d'entre eux sont inutilisé. Soit dans le cas de `poll` car il faut manipuler avant chaque appel un très grand tableau et que le système doit relire ce tableau a chaque appel.

Pour résoudre ce problème une nouvelle interface a été mise au point `/dev/poll`. Cette interface permet de créer un périphérique poll dans lequel il suffit d'écrire pour ajouter un descripteur a la liste des descipteurs que le noyau doit scruter. Et il suffit d'écrire de nouveau pour retirer un descripteur.

Epoll est un patch du noyau ![1]

1. Il faut verifier la présence d'epoll sur votre système, ouvrer le périphérique `/dev/epoll` en mode O_RDWR, sinon retour a `select` et `poll` `kdpfd = open("/dev/epoll",O_RDWR);`
2. Définiser le nombre maximal `maxfd` de descipteurs scrutables `#include <linux/eventpoll.h> \ldots ioctl(epo`
3. Allouer un segment de mémoire partagé avec `char *map = (char *)mmap(NULL, EP_MAP_SIZE(maxfds, PROT_READ | PROT_WRITE, MAP_PRIVATE, epoll_fd, 0))`
4. Maintenant vous pouvez ajouter des descipteurs

```
struct pollfd pfd;
pfd.fd = fd;
pfd.events = POLLIN | POLLOUT | POLLERR | POLLHUP;
pfd.revents = 0;
if (write(kdpfd, &pfd, sizeof(pfd)) != sizeof(pfd)) {
/* gestion d'erreur */
}
```

5. Récupere les évenements

```
struct pollfd *pfds;
struct evpoll evp;

for (;;) {
        evp.ep_timeout = STD_SCHED_TIMEOUT;
        evp.ep_resoff = 0;

        nfds = ioctl(kdpfd, EP_POLL, &evp);
```

[1] Il existe deux version une version /Dev/poll et une version /dev/epoll qui faut utiliser car plus efficasse.

```
                pfds = (struct pollfd *) (map + evp.ep_resoff);
                for (ii = 0; ii < nfds; ii++, pfds++) {
                        traitement(pfds[ii].fd, pfds[ii].revents);
                }
        }
```

6. Retirer des descripteurs

```
pfd.fd = fd;
pfd.events = POLLREMOVE;
pfd.revents = 0;
if (write(kdpfd, &pfd, sizeof(pfd)) != sizeof(pfd)) {
/* gestion d'erreur */
}
```

Le petit détail technique génial de cette interface est le fait que pendant que vous récupere des évenements le système continu a travailler pour vous dans le segment de mémoire fournis par `mmap` ce qui fait que votre programme s'exécute en parallèle de la récupération d'information sur les périphériques et que l'appel `ioctl` est ainsi très rapide.

15.2.4 Les extensions de `read` et `write`

Une extension `readv, writev` de `read` et `write` permet en un seul appel système de réaliser l'écriture de plusieurs zones mémoire non contiguës, ce qui permet d'accélerer certaines entrées-sorties structurées. Mais aussi de mieux organiser les appels système dans notre cas.

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

ssize_t readv(int fd, const struct iovec iov[], int iovl);
ssize_t writev(int fd, const struct iovec iov[], int iovl);

struct iovec {
    void *iov_base ;
    int   iov_len;
};
```

15.3 une solution multi-activités

L'utilisation de plusieurs activités (threads, voir chapitre 16) permet de réaliser plusieurs appels de `read` en simultané, le premier `read` qui se débloque entraine l'exécution de l'activité le réalisant, ainsi le coût d'attente sur les descripteurs est minimal le système signalant imédiatement à la thread l'évenement. Le seul problème est d'avoir a gérer cette multiplicité d'activités, ce qui est dans le cas d'un simple echange bidirectionel est raisonable il suffit de deux activités indépendantes.

Pour un serveur de fichier cette solutions multi-activité peut ne pas supporter la monté en charge le nombre de threads étant limité plus rapidement que celui des descripteurs (a ma connaissance ont peut créer au plus 10000 threads sous linux et xp)

Pour une situation plus complexe comme un serveur de partage de données, des mécanismes d'exclusion mutuelle entre activités devront être mis en oeuvre, ce qui peut compliquer inutilement le problème. La solution avec un seul processus gérant rapidement des requêtes multiples sur plusieurs flux étant plus simple a réaliser.

Chapitre 16

Les threads POSIX

La programmation par thread (actvité) est naturelle pour gérer des phénomènes asyncrones. Les entrées utilisateur dans les interfaces graphiques (souris, clavier) sont plus facile a gérer si l'on peut séparer l'activité principale du logiciel de la gestion des commandes utilisateur. Les entrées sorties multiples voir le chapitre 15 correspondant, sont gérées plus simplement en utilisant des threads.

Les actvitiés sont une nouvelle façon de voire les processus dans un système. L'idée est de séparer en deux le concept de processus. La première partie est l'environement d'exécution, on y retrouve une très grande partie des éléments constitutifs d'un processus en particulier les informations sur le propriétaire, la position dans l'arborescence le masque de création de fichier etc. La deuxième partie est l'`activité`, c'est la partie dynamique, elle contient une pile, un context processeurs (pointeur d'instruction etc), et des données d'ordonancement.

L'idée de ce découpage est de pouvoir associer plusieurs activité au même environement d'exécution. Pour CHORUS l'ensemble des ressources d'un environnement d'exécution est appelé des acteurs, MACH parle de tâches et AMOEBA de process. Mais tous désigne l'unité d'exécution par le terme de thread of control.

Organisation en mémoire pour un processus UNIX avec plusieurs threads : voir figure 16.1.

On peut grace au thread gérer plusieurs phénomènes asyncrone dans le même contexte, c'est à dire, un espace d'adressage commun, ce qui est plus confortable que de la mémoire partagée et moins couteux en ressource que plusieurs processus avec un segment de mémoire partagé.

Un processus correspond à une instance d'un programme en cours d'éxécution. Un thread correspond à l'activité d'un processeur dans le cadre d'un processus. Un thread ne peut pas exister sans processus (la tâche englobante), mais il peut y a voir plusieurs thread par processus, dans le cas de linux il ne peut y a voir de tâche sans au moins une activité.

16.0.1 Description

Un processus est composé des parties suivantes : du code, des données, une pile, des descripteurs de fichiers, des tables de signaux. Du point de vue du noyau, transférer l'exécution à un autre processus revient à rediriger les bons pointeurs et recharger les registres du processeur de la pile. Les divers threads d'un même processus peuvent partager certaines parties : le code, les données, les descripteurs de fichiers, les tables de signaux. En fait, ils ont au minimum leur propre pile, et partagent le reste.

16.0.2 `fork` et `exec`

Après un `fork`, le fils ne contient qu'une seule activité (celle qui a exécuté le `fork`). Attention aux variables d'exclusion mutuelle (qui font partie de l'espace d'adressage partagé) qui sont

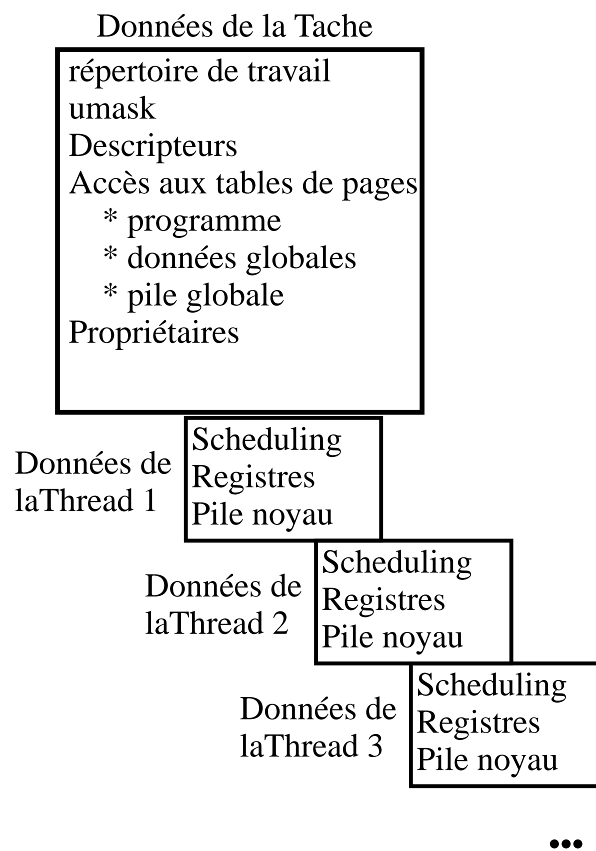

FIG. 16.1 – Organisation mémoire, partage des fonctions entre le processus et les activités

conservées après le `fork()` et dont le contenu ne varie pas. Ainsi si une activité a pris le sémaphore avant le `fork()`, si l'activité principale cherche à prendre ce sémaphore après le `fork()` elle sera indéfiniment bloquée.

Après un `exec`, le processus ne contient plus que la thread qui a exécuté l'une des six commandes `exec`. Pas de problème avec les sémaphores comme l'espace d'adressage a changé.

16.0.3 clone

Sous linux (et rarement sous les systèmes Unix) il existe un appel système un peut spécial. Cet appel système réalise un dédoublement du processus comme `fork` d'ou son nom de `clone`. Cet appel système permet de préciser exactement ce que l'on entend partager entre le processus père et le processus fils.
Eléments partageables :

ppid Création d'un frère au lieux d'un fils.

FS Partage de la structure d'information liée au système de fichier (".", "/", umask),

FILES Partage de la table des descripteurs,

SIGHAND Partage de la table des gestionnaires de Signaux, mais pas des masques de signaux,

PTRACE Partage du "crochet" (hook) de debug voire l'appel ptrace.

VFORK Partage du processeur ! le processus père est bloqué tantque le fils n'a pas exécuté soit `_exit` soit `execve`, c'est à dire qu'il s'est détaché de tout les élément partageable du processus père (sauf les FILEs),

VM Partage de la mémoire virtuelle, en particulier les allocations et désallocations par `mmap` et `munmap` sont visibles par les deux proccessus.

pid Les deux processus ont le même numéro.

THREAD Partage du groupe de thread, les deux processus sont ou ne sont pas dans le même groupe de threads.

16.0.4 Les noms de fonctions

```
pthread[_objet]_operation[_np]
```

où

objet désigne si il est présent le type de l'objet auquel la fonction s'applique. Les valeurs possibles de objet peuvent être

`cond` pour une variable de condition

`mutex` pour un sémaphore d'exclusion mutuelle

opération désigne l'opération a réaliser, par exemple `create`, `exit` ou `init`

le suffixe `np` indique, si il est présent, qu'il s'agit d'une fontion non portable, c'est-à-dire Hors Norme.

16.0.5 les noms de types

```
pthread[_objet]_t
```

avec `objet` prenant comme valeur `cond`, `mutex` ou rien pour une thread.

16.0.6 Attributs d'une activité

Identification d'une pthread : le TID de type pthread_t obtenu par un appel à la primitive :

```
pthread_t pthread_self(void);
```

pour le processus propriétaire

```
pid_t getpid(void);
```

En POSIX, le fait de tuer la thread de numéro 1 a pour effet de tuer le processus ainsi que toutes les autres threads éventuelles du processus.

Pour tester l'égalité de deux pthreads on utilise

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

16.0.7 Création et terminaison des activités

Création

```
int pthread_create (pthread_t      *p_tid,
                    pthread_attr_t attr,
                    void           *(*fonction) (void *arg),
                    void           *arg
                   );
```

La création et l'activation d'une activité retourne -1 en cas d'echec, 0 sinon.

- le tid de la nouvelle thread est placé à l'adresse `p_tid`
- `attr` attribut de l'activité (ordonnancement), utiliser `pthread_attr_default`
- la paramètre `fonction` correspond à la fonction exécutée par l'activité après sa création : il s'agit donc de son point d'entrée (comme la fonction `main` pour les processus). Un retour de cette fonction correspondra à la terminaison de cette activité.
- le paramètre `arg` est transmis à la fonction au lancement de l'activité.

Terminaison

a) les appels UNIX `_exit` et donc `exit` terminent toutes les threads du processus.
b) Terminaison d'une thread

```
 int pthread_exit (int *p_status);
```

`p_status` code retour de la thread, comme dans les processus UNIX la thread est zombifiée pour attendre la lecture du code de retour par une autre thread. A l'inverse des processus, comme il peut y avoir plusieurs threads qui attendent, la thread zombie n'est pas libérée par la lecture du p_status, il faut pour cela utiliser une commande spéciale qui permettra de libérer effectivement l'espace mémoire utilisé par la thread.

Cette destruction est explicitement demandée par la commande

```
int pthread_detach (pthread_t *p_tid);
```

Si un tel appel a lieu alors que l'activité est en cours d'exécution, cela indique seulement qu'à l'exécution de `pthread_exit` les ressources seront restituées.

16.1 Synchronisation

Trois mécanismes de synchronisation inter-activités :

- la primitive `join`
- les sémaphores d'exclusion mutuelle
- les conditions (évènements)

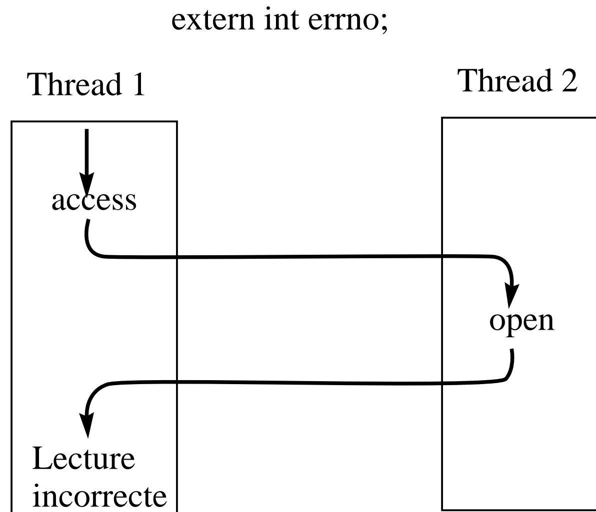

FIG. 16.2 – Changement de la valeur `errno` par une autre thread

16.1.1 Le modèle fork/join (Paterson)

Les rendez-vous : join
La primitive

```
int pthread_join (pthread_t tid, int **status);
```

permet de suspendre l'exécution de l'activité courante jusqu'à ce que l'activité `tid` exécute un appel (implicite ou explicite) à `pthread_exit`. Si l'activité `tid` est déjà terminée, le retour est immédiat, et le code de retour de l'activité visée est égal à **status (double indirection).
La primitive retourne :
0 en cas de succès
-1 en cas d'erreur

EINVAL si le `tid` est incorrect

ESRCH activité inexistante

EDEADLOCK l'attente de l'activité spécifiée conduit à un interblocage.

16.1.2 Le problème de l'exclusion mutuelle sur les variables gérées par le noyau

Il est nécessaire d'avoir plusieurs variables `errno`, une par activité. En effet cette variable globale pourrait être changée par une autre activité. Voir plus loin comment définir des variables globales locales à chaque activité.

16.1.3 Les sémaphores d'exclusion mutuelle

Ces sémaphores binaires permettent d'assurer l'exclusion mutuelle.

- Il faut définir un objet de type `pthread_mutex_t` qui correspond à un ensemble d'attributs de type `pthread_mutexattr_t`
  (on utilisera en général la constante `pthread_mutexattr_default`).
- Initialiser la variable par un appel à la fonction
  ```
  int pthread_mutex_init(pthread_mutex_t     *p_mutex,
                         pthread_mutexattr_t attr);
  ```
- On pourra détruire le sémaphore par un appel à la fonction
  ```
  int pthread_mutex_destroy(pthread_mutex_t *p_mutex);
  ```

16.1.4 Utilisation des sémaphores

Opération P :
Un appel à la fonction

```
pthread_mutex_lock (pthread_mutex_t *pmutex);
```

permet à une activité de réaliser une opération P sur le sémaphore. Si le sémaphore est déjà utilisé, l'activité est bloquée jusqu'à la réalisation de l'opération V (par une autre activité) qui libèrera le sémaphore.

Opération P non bloquante :

```
pthread_mutex_trylock (pthread_mutex_t *pmutex);
```

renvoie 1 si le sémaphore est libre
0 si le sémaphore est occupé par une autre activité
-1 en cas d'erreur.

Opération V :
Un appel à la fonction

```
pthread_mutex_unlock(pthread_mutex_t *pmutex);
```

réalise la libération du sémaphore désigné.

16.1.5 Les conditions (évènements)

Les conditions permettent de bloquer une activité sur une attente d'évènement. Pour cela l'activité doit posséder un sémaphore, l'activité peut alors libérer le sémaphore sur l'évènement, c'est-à-dire : elle libère le sémaphore, se bloque en attente de l'évènement, à la réception de l'évènement elle reprend le sémaphore.

Initialisation d'une variable de type `pthread_cond_t`

```
int pthread_cond_init (pthread_cond_t *p_cond, pthread_condattr_t attr);
```

L'attente sur une condition

```
int pthread_cond_wait (pthread_cond_t *p_cond, pthread_mutex_t *p_mutex);
```

Trois étapes

1. libération sur sémaphore *p_mutex
2. activité mise en sommeil sur l'évènement
3. réception de l'évènement, récupération du sémaphore

La condition est indépendante de l'événement et n'est pas nécessairement valide à la réception (cf. exemple).

Exemple, le programme suivant :

```
pthread_mutex_t m;
pthread_cond_t  cond;
int             condition = 0;

void *ecoute(void *beurk)
{
    pthread_mutex_lock(m);
    sleep(5);
    while (!condition)
        pthread_cond_wait(cond, m);
```

```
    pthread_mutex_unlock(m);

    pthread_mutex_lock(print);
    printf(" Condition realisee\n");
    pthread_mutex_unlock(print);
}

main()
{
    pthread_t lathread;

    pthread_create(lathread, pthread_attr_default, ecoute, NULL);
    sleep(1);
    pthread_mutex_lock(m);
    condition = 1;
    pthread_mutex_unlock(m);
    pthread_cond_signal(cond);
}
```

Un autre exemple d'utilisation de condition avec deux threads qui utilisent deux tampons pour réaliser la commande cp, avec une activité responsable de la lecture et l'autre de l'écriture. Les conditions permettent de synchroniser les deux threads. Ici nous utilisons la syntaxe NeXT/MACH.

```
#include <sdtio.h>
#include <fcntl.h>
#import  <mach/cthreads.h>

enum { BUFFER_A_LIRE = 1, BUFFER_A_ECRIRE = -1 };

mutex_t     lock1;  /* variables de protection et d'exclusion */
condition_t cond1;

char buff1[BUFSIZ];
int  nb_lu1;
int  etat1 = BUFFER_A_LIRE;
int  ds, dd;        /* descripteurs source et destination */

lire()         /* activite lecture */
{
    for(;;)  { /* lecture dans le buffer 1 */
        mutex_lock(lock1);
        while (etat1 == BUFFER_A_ECRIRE)
            condition_wait(cond1, lock1);
        nb_lu1 = read(ds, buff1, BUFSIZ);
        if (nb_lu1 == 0)
        {
            etat1 = BUFFER_A_ECRIRE;
            condition_signal(cond1);
            mutex_unlock(lock1);
            break;
        }
        etat1 = BUFFER_A_ECRIRE;
        condition_signal(cond1);
        mutex_unlock(lock1);
```

```
    }
}

ecrire()
{
    for(;;)
    { /* ecriture du buffer 1 */
        mutex_lock(lock1);
        while (etat1 == BUFFER_A_LIRE)
            condition_wait(cond1, lock1);
        if (nb_lu1 == 0)
        {
            mutex_unlock(lock1);
            exit(0);
        }
        write(dd, buff1, nb_lu1);
        mutex_unlock(lock1);
        etat1 = BUFFER_A_LIRE;
        condition_signal(cond1);
    }
}

main()
{
    ds    = open(argv[1], O_RDONLY);
    dd    = open(argv[2], O_WRONLY|O_TRUNC|O_CREAT, 0666);
    lock1 = mutex_alloc();
    cond1 = condition_alloc();

    cthread_fork((cthread_fn_t)lire, (any_t)0);
    ecrire(); /* la thread principale realise les ecritures */
}
```

16.2 Ordonnancement des activités

16.2.1 L'ordonnancement POSIX des activités

L'ordonnancement des activités DCE basé sur POSIX est très similaire à l'ordonnancement des activités sous MACH. Deux valeurs permettent de définir le mode d'ordonnancement d'une activité :
la politique et la priorité.
Pour manipuler ces deux valeurs, il vous faut créer un objet attribut d'activité (`pthread_attr`) en appelant `pthread_attr_create()`, puis changer les valeurs par défaut avec les fonctions décrites plus loin et créer la pthread avec cet objet `pthread_attr`. Ou bien la pthread peut elle-même changer ses deux valeurs, priorité et politique.

Les fonctions sont :

```
#include <pthread.h>
pthread_attr_setsched(pthread_attr_t *attr, int politique);
```

Les différentes politiques possibles sont :

SCHED_FIFO La thread la plus prioritaire s'exécute jusqu'à ce qu'elle bloque. Si il y a plus d'une pthread de priorité maximum, la première qui obtient le cpu s'exécute jusqu'à ce qu'elle bloque.

SCHED_RR Round Robin. La thread la plus prioritaire s'exécute jusqu'à ce qu'elle bloque. Les threads de même priorité maximum sont organisées avec le principe du tourniquet, c'est-à-dire qu'il existe un quantum de temps au bout duquel le cpu est préempté pour une autre thread (voire Chapitre 6 sur les Processus).

SCHED_OTHER Comportement par défaut. Tous les threads sont dans le même touniquet, il n'y a pas de niveau de priorité, ceci permet l'absence de famine. Mais les threads avec une politique `SCHED_FIFO` ou `SCHED_RR` peuvent placer les threads `SCHED_OTHER` en situation de famine.

SCHED_FG_NP (option DCE non portable) Même politique que `SCHED_OTHER` mais l'ordonnanceur peut faire évoluer les priorités des threads pour assurer l'équité.

SCHED_BG_NP (option DCE non portable) Même politique que `SCHED_FG_NP`, mais les threads avec une politique `SCHED_FIFO` ou `SCHED_RR` peuvent placer les threads `SCHED_BG_NP` en situation de famine.

```
pthread_attr_setprio(pthread_attr_t *attr, int prio);
```

La priorité varie dans un intervalle défini par la politique :
PRI_OTHER_MIN <= prio <= PRI_OTHER_MAX
PRI_FIFO_MIN <= prio <= PRI_FIFO_MAX
PRI_RR_MIN <= prio <= PRI_RR_MAX
PRI_FG_MIN_NP <= prio <= PRI_FG_MAX_NP
PRI_BG_MIN_NP <= prio <= PRI_BG_MAX_NP

Ces deux fonctions retournent 0 en cas de succès et -1 sinon. La valeur de **errno** indiquant si l'erreur est une question de paramètres ou de permission.

Les deux fonctions que l'on peut appeler sur une pthread pour changer sa priorité ou sa politique sont :

```
pthread_setprio(pthread_t *unepthread, int prio);
pthread_setsched(pthread_t *unepthread, int politique, int prio);
```

Il est possible de connaître la priorité ou la politique d'une pthread ou d'un objet pthread_attr avec :

```
pthread_attr_getprio(pthread_attr_t *attr,int prio);
pthread_attr_getsched(pthread_attr_t *attr,int politique);
pthread_getprio(pthread_t *unepthread, int prio);
pthread_getsched(pthread_t *unepthread, int politique);
```

16.3 Les variables spécifiques à une thread

Avec un processus multi-threads, nous sommes dans une situation de partage de données. Toutes les données du processus sont à priori manipulables par toutes les threads. Or certaines données sont critiques et difficilement partageables. Premièrement ce sont les données de la bibliothèque standard. Pour les fonctions de la bibliothèque standard, on peut résoudre le problème en utilisant un sémaphore d'exclusion mutuelle `pthread_mutex_t` pour POSIX.

Mais certaines variables ne peuvent être protégées. C'est le cas de la variables `errno`, comme nous l'avons vu précédemment. Pour cette variable, la solution est d'avoir une variable par thread. Ainsi le fichier `<errno.h>` est modifié et contient :

```
extern int *_errno();
#define errno (*_errno())
```

La valeur `errno` est obtenue par une fonction qui retourne la valeur de `errno` associée à la thread qui fait l'appel à `_errno` .

16.3.1 Principe général des données spécifiques, POSIX

L'idée des données spécifique est de créer un vecteur pour chaque donnée spécifique. Ainsi pour des données spécifique statiques, chaque thread possède son propre exemplaire. Les données spécifiques sont identifiées par des clés de type `pthread_key_t`.

16.3.2 Création de clés

La création d'une clé est liée à la création d'un tableau statique (variable globale), initialisé à NULL à la création. La fonction

```
#include <pthread.h>
int pthread_keycreate (pthread_key_t *p_cle,
                       void          (*destructeur)(void *valeur));
```

permet la création du tableau, 0 succès et -1 echec. La structure pointée par `p_cle` nous permettra d'accèder aux valeurs stockées, la clé est évidemment la même pour toutes les threads. Le paramètre `destructeur` de type pointeur sur fonction prenant un pointeur sur void en paramètre et renvoyant void, donne l'adresse d'une fonction qui est exécutée à la terminaison de la thread (ce qui permet de faire le ménage). Si ce pointeur est nul, l'information n'est pas détruite à la terminaison de l'activité.

16.3.3 Lecture/écriture d'une variable spécifique

La fonction

```
#include <pthread.h>
int pthread_getspecific (pthread_key_t *p_clé, void **pvaleur);
```

permet la lecture de la valeur qui est copié à l'adresse `pvaleur` retourne 0 ou -1 selon que l'appel à réussi ou non. La fonction

```
#include <pthread.h>
int pthread_setspecific (pthread_key_t *p_clé, void *valeur);
```

permet l'écriture à l'emplacement spécifié de `valeur` retourne 0 ou -1 selon que l'appel a réussit ou non.

16.4 Les fonctions standardes utilisant des zones statiques

Certaines fonctions standardes comme `ttyname()` ou `readdir()` retourne l'adresse d'une zone statique. Plusieurs threads en concurrence peuvent donc nous amener à des situations incohérentes. La solution des sémaphores d'exclusion étant coûteuse, ces fonctions sont réécrites pour la bibliothèque de thread de façon à être réentrantes.

Attention les problèmes de réentrance peuvent avoir lieu en utilisant des appels systèmes non réentrant dans les handlers de signaux ! Ceci sans utiliser de threads !

Chapitre 17

Clustering

Le clustering sont des techniques liés a l'utilisation de grappes d'ordinateurs utilisé comme un super ordinateur avec plusieurs processeurs. L'objectif est le RAIP : Redundant Array of Inexpensive PROCESSOR.

la station de trvail individuelle vielli a grande vitesse, une facon de recycler ceux qui sont en peu trop vieux (mais pas encore trops vieux) est de les rassembler dans votre premier cluster. De ces PC nous allons tirer un super calculateur, bien sur il est toujours plus rapide d'achetter un G5 si on en a les moyens, pour les mainframes (je l'affirme vous n'en avez pas les moyens ou alors vous en avez deja plusieurs...)[1].

Il existe différentes techniques de clustering pour différent objectifs :

Tolérence au pannes Google, le cluster est organisé pour assurer la redondance des unité de calcul pour assurer la continuité de service.

Super calculateur Earth Simulator,Plusieurs processeurs travaillant en même temps permet d'optenir un super calculateur à peu de frais, comme les processeurs qui compose chaque noeud sont bon marché. IL faut que le problème s'y prète c'est le cas des calculs météorologiques et des calculs de simulation à grande échelle.

Monté en charge Google, Le problème pour une application n'est pas toujours un problème de puissance de calcul parfois ce qui posse problème c'est la quantité d'entrées sorties qui faut assurer. Vous pouvez d'ailleurs facilement tester cette propriété en réalisant un petit programme qui sans saturer l'unité central sature complètement les entrées sorties `while :;do; { cp grosfichier /tmp/$PID } &; done`

Ainsi le clustering a essentiellement pour objectif d'utiliser le dicton "l'union fait la force" pour résoudre une difficulté de calcul.

17.1 Le clustering sous linux

Pour plus d'information le vous conseil le site suivant qui vous donnera de bonnes références. `http://www-igm.univ-mlv.fr/~dr/Xpose2001/vayssade/`

[1]C'est evident si vous avez les moyens d'acheter un mainframe vous avez les moyens d'achetter un super-cluster haut de gamme.

Chapitre 18

Bibliographie

J.-M. Rifflet. *La programation sous* UNIX. Ediscience, 1993. Le manuel de référence.

A.Tanenbaum. *Systèmes d'exploitation, sysytèmes centralisés, systèmes distribués.* Inter-Editions, 1994. Cours général sur les sytèmes d'exploitation.

M. Bach. *The design of the UNIX operating system.* 1986. Prentice-Hall, Englewood Cliffs, N.J. ISBN 0-13-201757-1

J. Beauquier & B. Bérard Systèmes d'exploitation concepts et algorithmes. 1991. McGraw-Hill. ISBN 2-7042-1221-X

W.R. Stevens, *UNIX Network Programming.* 1990 Prentice-Hall, Englewood Cliffs, N.J.

W.R. Stevens, *Advanced Programming in the UNIX Environnement* Addison-Wesley ISBN 0-201-56317-7

18.1 Webographie

Vous trouverez a l'url suivant une webographie : www-igm.univ-mlv.fr/ dr/Cours.html

Index

/dev/epoll, 156
/dev/null, 105
/dev/poll, 156
/dev/pty, 110
/dev/tty, 104

accès direct, 34
accès séquentiel, 34
Allocation contiguë, 77
appels systèmes
- _exit, 38
- accept, 178
- brk, 60
- cfgetispeed, 110
- cfgetospeed, 110
- cfsetispeed, 110
- cfsetospeed, 110
- chdir, 59
- chroot, 59
- close, 46
- connect, 177
- creat, 44
- dup, 45
- dup2, 46
- exec, 51
- execle, 61
- execlp, 61
- execv, 61
- execve, 50, 61
- execvp, 61
- exit, 57
- fchdir, 59
- fcntl, 46
- fork, 50, 51, 56, 61
- getgrp2, 105
- getmsg, 173
- getpeername, 179
- getpgrp, 59, 105
- getpgrp2, 59
- getpid, 59
- getppid, 59
- getsid, 106
- getsockname, 179
- getsockopt, 179
- htonl, 179
- introduction, 41
- ioctl, 111
- isatty, 103
- kill, 113, 114
- listen, 177
- mkfifo, 100
- mknod, 100
- mmap, 92
- munmap, 92
- nice, 60
- open, 41
- pause, 120, 121
- pipe, 97
- poll, 155
- putmsg, 173
- putpmsg, 173
- read, 44, 99
- recv, 178
- recvfrom, 178
- recvmsg, 178
- sbrk, 60
- select, 153
- send, 178
- sendmsg, 178
- sendto, 178
- setgid, 59
- setpgid, 105
- setsid, 104
- setsockopt, 179
- setuid, 59
- sigaction, 121
- siginterrupt, 119
- siglongjmp, 118
- signal, 115
- sigpause, 120
- sigprocmask, 120
- sigsetjmp, 118
- sleep, 57
- socket, 175
- socketpair, 176
- tcdrain, 110
- tcflush, 110
- tcgetattr, 109
- tcgetgrp, 106
- tcgetsid, 106

  `tcsetattr`, 109
  `tcsetpgrp`, 106
  `times`, 58
  `ttyname`, 104
  `ulimit`, 60
  `umask`, 60
  `write`, 44, 100
arrière plan, 105

Best-fit, 81
bibliothèques, 5
boot bloc, 10
buffer cache, 25
bufferisation, 35
bufferiser, 25

chargement dynamique, 86
compactage, 81

désarmer, 55
Demand Paging, 86
Demand-Paging, 68
droits, 7

exclusion mutuelle, 63

famine, 63
FCFS, 65
ffs, 17
fichier, 7
  inodes, 9
  ordinaires, 8
  physiques, 8
  spéciaux, 8
fifo, 100
`FILE`, 31
First-fit, 81

groupe, 7, 10

handler, 114
HotSwap, 182

inodes, 9, 11
interruption, 54
intr, 104, 113

lazy swapper, 86
load, 68
longjmp, 114

Métadisque, 182
masquer, 55
mkfifo, 100

noyau, 5

ordonnancement, 63, 82
overlays, 85

page fault, 86
pages, 82
pendant, 114
physique, 103
pile, 50
pointeur de fichier, 34
préemption, 65
premier plan, 105
priorité, 67
proc, 22
processus, 49
  $DATA$, 50
  $TEXT$, 50
  états, 55, 57
  accumulateur, 53
  changement de contexte, 53
  commutation de mot d'état, 53, 54
  compteur ordinal, 53
  context, 53
  création, 49
  decomposition, 49
  format de fichier, 50
  mode d'un, 55
  mot d'état, 53
  niveau d'interruption, 54
  `struct proc`, 49
  recouvrement, 50
  swapin, 57
  swapout, 57
  table des processus, 51
  table des régions par processus, 51
  `struct user`, 49
  zone u, 51, 58
propriétaire, 7, 10
`protocole`, 175
pseudo-terminaux, 103, 110

quit, 104, 113

référence, 7
RAID, 181
redirection, 32
registre barrière, 77
registre base, 77
Round Robin, 66

SIGHUP, 104, 113
SIGINT, 113
signaux, 113
  kill, 113
SJF, 65

static, 104
`stdio.h`, 31
stdlib
    `atexit`, 38
    `clearerr`, 39
    `exit`, 38
    `fclose`, 36
    `feof`, 33, 39
    `fflush`, 36
    `fopen`, 32
    `fread`, 33
    `freopen`, 32, 35
    `fseek`, 34
    `ftell`, 35
    `fwrite`, 33
    `mkdir`, 39
    `perror`, 39
    `printf`, 31
    `remove`, 37
    `rename`, 37
    `rewind`, 35
    `rmdir`, 39
    `scanf`, 31
    `setbuf`, 37
    `setbuffer`, 37
    `setlignebuf`, 37
    `setvbuf`, 36
    `stderr`, 31
    `stdin`, 31
    `stdout`, 31
    `system`, 38
    `tmpfile`, 33
    `tmpnam`, 33
    `Xalloc`, 51
super bloc, 10
susp, 104, 113
swap, 79, 85
synchronisation, 101
Système de Gestion de Fichiers, 7

tas, 50
terminal de contrôle, 104
termios, 106
tubes, 97
tubes nommés, 100

Worst-fit, 81