

Informations pratiques

Le cours

ii Systèmes d'exploitation ii

Étude d'ensemble des techniques matérielles et logicielles utilisées pour construire un système d'exploitation.

L'intervenant

François PELLEGRINI

Maître de Conférences à l'ENSEIRB

Tél 05 56 84 69 35 LaBRI
05 56 84 23 56 ENSEIRB

pelegrin@labri.fr
pelegrin@enseirb.fr

Déroulement

- Le cours prendra une heure et demie par semaine, le lundi matin.
- Les travaux dirigés prendront deux heures par semaine, par groupes (tiers de promotion) :
 - 4 TD avec Michel Pallard,
 - 7 TD avec François Pellegrini,
 - 3 TD avec Frédéric Goudal.
- Vous serez évalués par :
 - un partiel,
 - un examen,
 - un projet commun système-réseaux.

Questions

Posez-en !

Systèmes d'exploitation

François PELLEGRINI
David SHERMAN

6 septembre 2001

Ce document est copiable et distribuable librement et gratuitement à la condition expresse que son contenu ne soit modifié en aucune façon, et en particulier que le nom de ses auteurs et de son institution d'origine continuent à y figurer, de même que le présent texte.

Avant-propos

L'étude des systèmes d'exploitation est transversale à de nombreuses disciplines (électronique, algorithmique, statistique, ...), auxquelles elle emprunte les outils qui lui sont nécessaires. En soi, elle ne constitue pas une discipline fondamentale, mais plutôt une des nombreuses applications de l'informatique, comme par exemple l'informatique de gestion.

Ce qui en fait la spécificité est l'omniprésence des systèmes d'exploitation dans l'informatique actuelle, et la pression continue d'utilisateurs toujours plus nombreux et exigeants, qui imposent aux concepteurs de ces systèmes d'étendre continuellement leurs fonctionnalités tout en gardant un souci constant d'efficacité maximale.

L'esprit des systèmes d'exploitation consiste en la recherche du meilleur compromis entre fonctionnalité, performance, et maintenabilité. Il demande de la rigueur dans l'analyse des problèmes, mais aussi de l'inventivité et de l'astuce dans l'implémentation, sans pour autant rendre le code produit illisible et immaintenable.

Bien que le contenu de ce polycopié soit très technique, et basé sur une énumération d'exemples, l'objectif de ce cours n'est pas (seulement) de vous fournir un compendium d'algorithmes et de recettes à bachotter.

Il est de vous donner une perspective historique sur l'évolution des concepts dans ce domaine en vous montrant comment, en fonction des contraintes technologiques de chaque époque, les ingénieurs ont été amenés à définir des solutions efficaces, et comment l'évolution des technologies a pu conduire ces mêmes ingénieurs à proposer de nouvelles solutions radicalement différentes.

Il s'agit donc pour vous, à travers ces exemples, d'affûter votre sens pratique, en recherchant à chaque fois les contraintes critiques par rapport au problème posé, en imaginant par vous-même les solutions adaptées, et en percevant comment les modifications de ces contraintes pourraient modifier vos solutions. C'est la prise en compte des évolutions possibles qui rendra votre démarche d'ingénieur réellement efficace et pérenne, en ce qu'elle permettra une maintenance et une évolution plus facile de vos programmes. L'étude des systèmes d'exploitation est en cela une voie de l'efficacité de la programmation.

Ouvrages de référence

- *Systèmes d'Exploitation*, A. Tanenbaum, InterÉditions.
- *Principes des Systèmes d'Exploitation*, A. Silberschatz et P. B. Galvin, Addison Wesley.
- *Fundamentals of Operating Systems*, A. M. Lister et R. D. Eager, Springer-Verlag.
- *Operating Systems*, H. M. Dietel, Addison-Wesley.
- *The Design of the Unix Operating System*, M. J. Bach, Prentice Hall.
- *Linux Source Navigator*, B. Walter, <http://metalab.unc.edu/linux-source/>.

Le *Tanenbaum* constitue, même s'il date un peu, une excellente introduction à la problématique des systèmes d'exploitation. Tellement excellente, en fait, qu'il est difficile de s'en écarter lorsque se posent les problèmes de la structuration d'un cours général sur les systèmes d'exploitation et de son contenu.

De fait, le présent polycopié pourrait sembler une pâle copie de ce glorieux original. C'en est plutôt une synthèse, reprenant également des informations plus techniques reprises dans d'autres ouvrages, ainsi que quelques anecdotes personnelles, mais qui n'a aucunement la prétention de le remplacer. Les lecteurs désireux d'en savoir plus sont donc chaudement encouragés à prendre le temps de la lecture d'un ou plusieurs de ces ouvrages.

Chapitre 1

Introduction

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

Répondre complètement à cette question n'est pas simple. De manière pratique, le système d'exploitation est le logiciel le plus important de la machine, puisqu'il fournit :

- une gestion des ressources de celle-ci : processeurs, mémoires, disques, horloges, périphériques, communication inter-processus et inter-machines ;
- une base pour le développement et l'exécution de programmes d'application.

Pourquoi étudier les systèmes d'exploitation ?

- Tout programme est concerné : il est important d'appréhender la façon dont fonctionne un système d'exploitation pour améliorer l'efficacité de ses propres programmes ;
- Tout programmeur est susceptible de rencontrer les mêmes problèmes de mise en œuvre dans son propre domaine : pas la peine de réinventer la roue.
- C'est un sujet intéressant en soi, dont l'objectif est la recherche de l'efficacité, nécessitant une étude théorique approfondie mais dont l'objectif est la fourniture de solutions réalisables en pratique : c'est une excellente approche du métier d'ingénieur !

1.2 Problématique

Pour que les programmes puissent s'exécuter de façon portable et efficace, il faut pouvoir gérer simultanément :

- la multiplicité des différentes ressources ;
- la complexité des composants de chacune d'elles, qui requiert la prise en compte de nombreux détails embêtants, sources de bogues.

On peut appréhender ces problèmes à travers deux exemples.

1.2.1 Utilisation du contrôleur de lecteur de disquettes NEC PD765

Ce contrôleur de lecteur de disquettes possède 16 commandes, lancées par l'écriture de 1 à 9 octets dans les registres du contrôleur, et qui permettent d'effectuer des opérations telles que :

- la lecture ou l'écriture d'un secteur ;
- le déplacement du bras de lecture ;
- le formatage d'une piste ;
- l'initialisation du contrôleur, la calibration des têtes de lecture, des tests internes...

Ainsi, les commandes de lecture et d'écriture prennent 13 paramètres codés sur 9 octets, codant entre autres :

- le nombre de secteurs par piste ;
- la distance entre deux secteurs ;
- le numéro du secteur sur la piste ;
- le mode d'enregistrement ;
- la méthode de traitement des marques d'effaçage...

Le contrôleur retourne 23 champs d'état et d'erreur codés sur 7 octets. Il faut gérer soi-même le démarrage et la mise en veille du moteur, en choisissant le meilleur compromis entre le surcoût en temps du démarrage et l'usure des disquettes.

Personne ne souhaite directement gérer cela dans ses programmes applicatifs. Pour s'en convaincre, il suffit de consulter le fichier `/usr/src/linux/drivers/block/floppy.c` des sources de Linux [?], où la structure ci-dessus est définie sous le nom de `old_floppy_raw_cmd`.

1.2.2 Partage d'imprimante

Une machine multi-utilisateurs fournit un service d'impression, qui peut être utilisé par n'importe quel programme s'exécutant sur la machine. Pour cela, il faut :

- pouvoir verrouiller l'accès à l'imprimante, afin que les flots de caractères produits par les programmes désirant imprimer ne s'entrelacent pas sur le papier ;
- gérer des tampons d'impression, afin que les programmes puissent reprendre leur travail sans devoir attendre la fin effective de l'impression.

Le problème est ici de gérer l'accès à une ressource coûteuse (à la fois en argent et en temps). À tout instant, il faut :

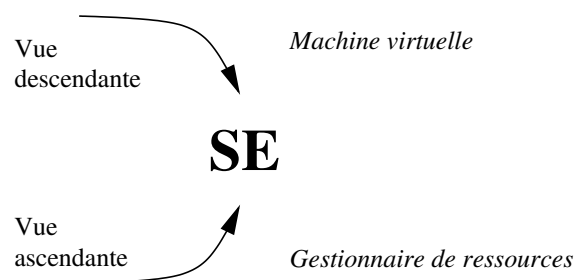
- connaître l'utilisateur d'une ressource donnée (pour éventuellement la facturer) ;
- gérer l'accès concurrent(iel) à cette ressource ;
- pouvoir accorder l'usage (exclusif) à cette ressource ;
- éviter les conflits entre les programmes ou entre les usagers.

1.3 Fonctionnalités d'un système d'exploitation

Un système d'exploitation a pour but :

- de décharger le programmeur d'une tâche de programmation énorme et fastidieuse, et de lui permettre de se concentrer sur l'écriture de son application ;
- de protéger le système et ses usagers de fausses manipulations ;
- d'offrir une vue simple, uniforme, et cohérente de la machine et de ses ressources.

On peut considérer un système d'exploitation de deux points de vue, représentés par le schéma ci-dessous.



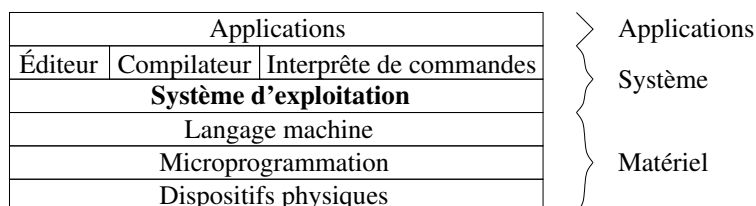
La machine virtuelle fournit à l'utilisateur :

- une vue uniforme des entrées/sorties ;
- une mémoire virtuelle et partageable ;
- la gestion de fichiers et répertoires ;
- la gestion de droits d'accès, sécurité, et du traitement des erreurs ;
- la gestion de processus ;
- la gestion des communications inter-processus.

En tant que gestionnaire de ressources, le système d'exploitation doit permettre :

- d'assurer le bon fonctionnement des ressources et le respect des délais ;
- l'identification de l'usager d'une ressource ;
- le contrôle des accès aux ressources ;
- l'interruption d'une utilisation de ressource ;
- la gestion des erreurs ;
- l'évitement des conflits.

1.4 Place du système d'exploitation dans l'ordinateur



Ne sont pas des systèmes d'exploitation :

- l'interprête de commandes ;
- le système de fenêtrage ;
- les utilitaires (`cp`, `chmod`, `uptime`, ...);
- le compilateur (ni sa bibliothèque);
- l'éditeur...

En fait, tous ces programmes s'exécutent dans un mode non privilégié, car ils n'ont pas besoin d'un accès privilégié au matériel. En revanche, le système d'exploitation fonctionne typiquement en mode privilégié, pour pouvoir accéder à toutes les fonctionnalités du processeur. Ainsi, le système d'exploitation est protégé par le matériel contre les erreurs de manipulation (mais il existe des systèmes d'exploitation s'exécutant sur du matériel non protégé, comme par exemple le DOS sur les anciens IBM PC).

1.5 Principes des systèmes d'exploitation

1.5.1 Appels système

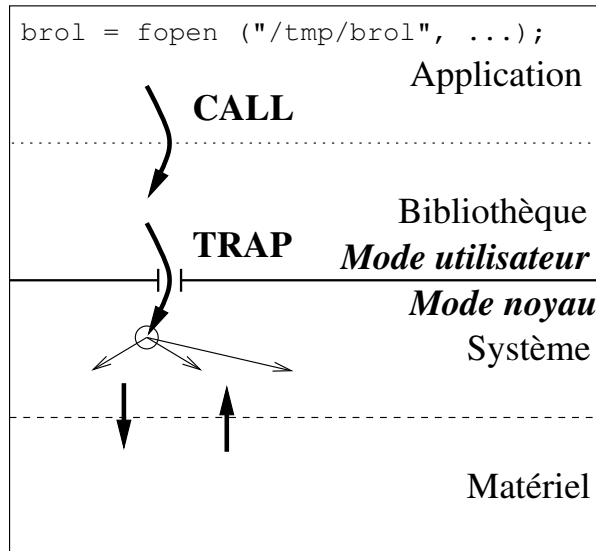
Ils constituent l'interface entre le système d'exploitation et les programmes d'application (ou leurs bibliothèques) qui s'exécutent en mode non privilégié (aussi appelé parfois mode utilisateur). Ils sont réalisés au moyen d'instructions spécifiques (les `;; traps ii`, ou interruptions logicielles), qui permettent le passage en mode privilégié (aussi appelé mode noyau, ou `;; kernel ii`), lorsqu'il existe sur le processeur.

Au niveau du processeur, le mode noyau se différencie habituellement du mode utilisateur par les fonctionnalités suivantes :

- le code et les données utilisés par le système d'exploitation ne sont accessibles qu'en mode noyau. Ceci se fait en n'incluant les segments mémoires correspondants que lors du passage en mode noyau ;
- les instructions de modification de la table des segments mémoire ne sont permises qu'en mode noyau. Ainsi, un programme utilisateur ne pourra modifier ses droits d'accès à la mémoire ;
- les instructions de lecture et d'écriture sur les ports d'entrée/sortie du matériel ne sont permises qu'en mode noyau. Un programme d'application ne peut donc accéder directement au matériel sans passer par le système d'exploitation.

La différence entre mode noyau (privilégié) et mode utilisateur (non privilégié) est gérée directement au niveau du processeur. Elle n'a rien à voir avec la notion de super-utilisateur mise en œuvre sur certains systèmes d'exploitation, qui est gérée au niveau logiciel dans le code du système d'exploitation. En fait, même le super-utilisateur d'un système passe le plus clair de son temps

CPU en mode non-privilégié (utilisateur) du processeur!



Le mécanisme des traps permet à tout programme de dérouter son exécution pour exécuter le code du système d'exploitation. Ce mécanisme doit être implémenté de façon matérielle au niveau du processeur, car les programmes ne peuvent connaître lors de leur compilation l'adresse à laquelle ils devront se brancher pour appeler les fonctionnalités du système d'exploitation ; cela n'est d'ailleurs pas souhaitable, pour des raisons de maintenabilité du système d'exploitation et de portabilité entre versions différentes, car cette adresse peut changer d'une version du système à l'autre.

L'adresse de destination du trap est stockée dans une portion de la mémoire physique de la machine qui n'est accessible qu'en mode noyau, appelée le vecteur d'interruptions (voir section 2.2.2, page 24), qui est initialisée au démarrage du système d'exploitation.

1.5.2 Processus

Typiquement, un processus est une instance d'un programme en train de s'exécuter. Il est représenté au niveau du système d'exploitation par son code (*text*), ses données, sa pile d'exécution, les valeurs courantes des registres du processeur, ainsi que par d'autres données relatives à l'état courant du système : état du processus, liste des fichiers ouverts, etc.

Un processus est créé par d'autres processus (sauf le premier, bien sûr!). Il est susceptible d'être suspendu, redémarré, et de recevoir des événements (signaux) traitables de façon asynchrone.

Dans les systèmes récents, on a deux niveaux d'exécution :

- les processus classiques, ou *processus lourds*, possédant chacun leurs données propres ;
- les tâches légères, ou *threads*, qui peuvent exister au sein de chaque processus lourd (dans ce cas, il y en a au moins une par processus, qui

exécute le code de la fonction `main`), qui ont leur pile propre mais partagent toutes leurs données.

1.5.3 Système de fichiers

Le système de fichiers fournit un modèle commode d'organisation des informations persistantes (c'est à dire dont la durée de vie est supérieure à celle des processus), avec une gestion indépendante du support matériel. Le plus souvent, l'organisation des fichiers se fait de façon arborescente.

Le système de fichiers fournit les services classiques sur les fichiers : création, suppression, ouverture, fermeture, positionnement, lecture, écriture.

Afin d'uniformiser l'ensemble des ressources d'entrées/sorties en un modèle cohérent, la plupart des systèmes de fichiers possèdent la notion de fichier spécial : terminaux, souris, disques, mémoire, ... De même, c'est le système de fichiers qui supporte la communication inter-processus, avec les fichiers spéciaux de type pipe, pipe nommé, ou socket.

Sous Unix, la plupart des fichiers spéciaux liés aux périphériques sont situés dans la sous-arborescence du répertoire `/dev`. Sous Linux, les processus eux-mêmes sont représentés comme des sous-répertoires du répertoire `/proc`.

1.6 Historique des systèmes d'exploitation

1.6.1 Première génération (1936-1955)

C'est l'apparition des premiers ordinateurs, à relais et à tubes à vide, programmés par tableaux de connecteurs, puis par cartes perforées au début des années 50. C'est aussi l'apparition du terme *bug*.

1.6.2 Deuxième génération (1955-1965)

L'apparition du transistor rendit les ordinateurs plus fiables. Ils pouvaient maintenant être vendus, et l'on vit apparaître pour la première fois la distinction entre constructeur, opérateur, programmeur, et utilisateur.

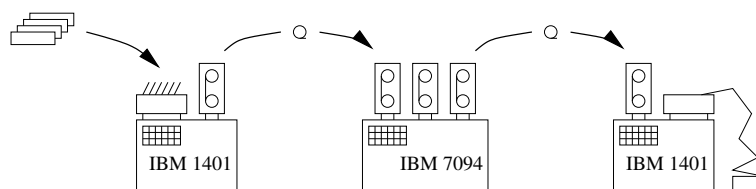
Les programmes s'exécutaient par lots (*batch*). Un interprète de commandes sommaire permettait le chargement et l'exécution des programmes dans l'ordinateur. Ainsi, avec le système FMS (*Fortran Monitor System*), on soumettait les travaux de la manière suivante :

```
$JOB
$FORTRAN
...Programme...
$LOAD
$RUN
...Données...
$END
```

Les gros ordinateurs disposaient typiquement de trois dérouleurs de bande : un pour conserver la bande du système d'exploitation, un pour le programme à exécuter et ses données, et le dernier pour recueillir les données en sortie.

En amont et en aval se trouvaient deux calculateurs plus petits, chargés l'un de transcrire sur bande les cartes perforées apportées par le programmeur, et l'autre d'imprimer sur papier les résultats contenus sur les bandes de sortie de données.

Ces ensembles étaient servis par des opérateurs, dont le rôle était d'alimenter les ordinateurs en bandes, cartes, et papier.



Comme la majeure partie du temps du ordinateur principal était perdue lors des déplacements des opérateurs, un système de traitement par lots (*batch*) fut mis en place : plusieurs travaux (*jobs*) étaient collectés sur une même bande d'entrée, qui était changée une fois par heure et apportée au ordinateur principal. Celui-ci lit le premier travail de la bande, et à la fin de chaque travail lit automatiquement le suivant, jusqu'à la fin de la bande.

1.6.3 Troisième génération (1965-1980)

L'avancée technologique majeure de cette génération est l'apparition des circuits intégrés, qui ont permis de diminuer le rapport coût/performance de plusieurs ordres de grandeur.

La standardisation apportée par les circuits intégrés s'est également appliquée aux machines, avec l'apparition de familles de machines, qui partagent le même langage machine et le même système d'exploitation, pour des puissances et des applications très différentes.

Le problème majeur de cette approche était de pouvoir disposer d'un système d'exploitation efficace sur toutes les machines de la gamme, permettant d'utiliser toute la puissance des gros calculateurs, mais aussi capable de tenir dans la mémoire des plus petits...

Afin de rentabiliser l'utilisation des machines les travaux purent être stockés sur le disque de l'ordinateur dès leur arrivée en salle machine, sans passer par des ordinateurs annexes et des manipulations de bandes. Cette technique s'appella le spouleur (francisation de *spool*, pour *Simultaneous Peripheral Operation On Line*).

De même, s'est développé l'usage de la multiprogrammation : la mémoire physique était partitionnée en segments de tailles fixées à l'avance, dont chacune pouvait accueillir un programme différent. Ainsi, lorsqu'une tâche attendait la fin d'une entrée/sortie, le processeur pouvait basculer sur une autre tâche. C'est à ce moment qu'apparurent les premiers mécanismes matériels de contrôle des accès mémoire, pour protéger mutuellement les programmes contre les accès invalides.

L'inconvénient majeur des systèmes de traitement par lots était que le programmeur ne pouvait plus déboguer son programme en temps réel, comme

c'était le cas au tout début de l'informatique, lorsque chaque programmeur se voyait attribuer la machine pendant plusieurs heures d'affilée. Pour retrouver cette interactivité ont donc été développés les systèmes à temps partagé. Le premier, CTSS (ij *Concurrent Time-Sharing System* ij), crée au MIT en 1962, a été suivi par Multics (ij *MULTiplexed Information and Computing Service* ij), développé aux laboratoires Bell, puis par son descendant Unix, premier système d'exploitation écrit dans un langage de haut niveau, le C.

1.6.4 Quatrième génération (1980-2000)

Le développement des circuits LSI (ij *Large Scale Integration* ij), puis VLSI, a permis l'avènement des ordinateurs personnels, qui ne diffèrent que très peu, du point de vue architectural, des mini-ordinateurs de la génération précédente.

Le développement des réseaux de communication et la baisse de leurs coûts d'accès a permis la création de réseaux locaux de machines, et conduit à l'explosion d'Internet. Ils sont pris en compte par les systèmes d'exploitation en réseau, et complètement intégrés dans les systèmes d'exploitation distribués, qui fournissent une image unifiée de l'ensemble des machines.

1.6.5 Et après ?

Au taux actuel de croissance de l'intégration des composants, la barrière atomique devrait être atteinte vers les années 2010-2015, et la vitesse de la lumière constitue toujours une limite infranchissable. Au niveau fondamental, les recherches s'orientent autour des composants à effets quantiques et des systèmes réversibles, mais seuls des transistors uniques fonctionnent à ce jour en laboratoire. Les composants analogiques optiques lasers sont également étudiés.

Au niveau des systèmes d'exploitation, l'interface homme-machine et les interconnexions entre programmes devraient être les grands bénéficiaires des efforts actuels.

1.7 Types de systèmes d'exploitation

Il n'existe pas de système d'exploitation efficace dans tous les contextes d'utilisation. On définit donc des familles de systèmes, en fonction des contraintes qui pèsent sur eux.

1.7.1 Mono-utilisateur

Ces systèmes d'exploitation n'acceptent qu'un seul utilisateur à un moment donné. Ils sont construits autour d'une machine virtuelle simple, et facilitent l'utilisation des différents périphériques. Ils peuvent être multi-tâches, mais n'implémentent pas les notions d'utilisateur et de protection.

1.7.2 Contrôle de processus

Ces systèmes sont utilisés principalement en milieu industriel, pour le contrôle de machines-outils ou de dispositifs complexes et critiques comme les usines chi-

miques et les centrales nucléaires.

L'objectif du système est de permettre de réagir en un temps garanti à des événements issus de capteurs, pour maintenir la stabilité du processus industriel ou répondre à des alarmes (ii *feed-back* ii).

Ces systèmes sont donc fortement orientés temps-réel, et doivent être fiables et tolérants aux pannes. On sacrifie donc la généralité et la convivialité du système au profit de l'efficacité.

1.7.3 Serveurs de fichiers

Ces systèmes contrôlent de gros ensemble d'informations, interrogeables à distance. On a donc besoin d'un temps de réponse court, avec possibilité de mise à jour à la volée. L'utilisateur n'a pas besoin de savoir comment la base de fichiers est organisée de façon interne, le système réalisant le masquage de cette structure.

Des systèmes de ce type ont été conçus pour les serveurs de fichiers de type RAID (ii Redundant Arrays of Inexpensive Disks ii). Cependant, à l'heure actuelle, vu la puissance des processeurs actuels, il est courant de construire de tels systèmes comme des surcouches de systèmes généraux.

1.7.4 Transactionnel

Ces systèmes contrôlent de grandes bases de données modifiées de façon très fréquente, et doivent garantir un temps de réponse très court, la cohérence constante de la base de données, et la résolution des conflits. Ceci passe par la définition de transactions atomiques et de points de reprise.

Transaction 1 :

p = "Produit1"
 q = q₁
 c = "Dupont"

 stock (p) -= q
 livraison (c, p) += q

Transaction 2 :

p = "Produit2"
 q = q₂
 c = "Durand"

 stock (p) -= q
 livraison (c, p) += q

$$\forall p \in \{\text{Produits}\}, \left(\text{stock}(p) + \sum_{c \in \{\text{Clients}\}} \text{livraison}(c,p) \right) = \text{cste} .$$

Ainsi, dans l'exemple ci-dessus, chacune des transactions doit conserver le nombre de produits actuellement disponibles. Si un processus de modification échoue après la décrémentation de la variable *stock*, le système doit restaurer l'état initial des tables. De même, si les deux transactions s'exécutent en parallèle, le système doit garantir que les modifications concurrentes maintiendront la cohérence de la base, et doit sérialiser les deux transactions si elles s'effectuent sur le même produit, afin que la deuxième ne soit possible que s'il reste assez

de produit en stock après la première.

Ici encore, à l'heure actuelle, les systèmes transactionnels sont le plus souvent construits au dessus de systèmes généraux.

1.7.5 Général

Les systèmes généraux sont caractérisés par la capacité à accueillir simultanément de nombreux utilisateurs effectuant chacun des tâches différentes. Ils sont donc multi-utilisateurs et multi-tâches. Ils doivent disposer de systèmes de fichiers sophistiqués, de systèmes d'entrées/sorties variés, et de nombreux logiciels couvrant un grand nombre de domaines d'application : éditeurs, compilateurs, tableurs, logiciels graphiques, etc.

Ces systèmes peuvent disposer de capacités interactives (les utilisateurs partagent les capacités de la machine, même s'ils ont l'impression de disposer simultanément de toutes, et peuvent lancer simultanément plusieurs processus, qu'ils contrôlent depuis leur terminal), et de traitement par lots (l'utilisateur n'a pas de contact avec son programme entre son lancement et sa terminaison).

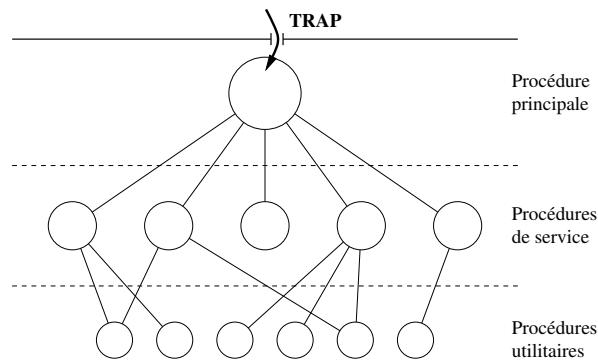
1.8 Structure interne des systèmes d'exploitation généraux

1.8.1 Systèmes monolithiques

Il s'agit de l'organisation la plus répandue. Elle est caractérisée par son absence de structure interne. Le système est une collection de procédures, chacune visible de toutes les autres, et pouvant appeler toute autre procédure qui lui est utile. La seule barrière de protection est entre le mode utilisateur et le mode noyau.

Afin de rationaliser l'écriture de tels systèmes, ceux-ci adoptent naturellement une structuration interne à trois niveaux, mais celle-ci n'est d'aucune protection contre les erreurs.

- La procédure principale est exécutée lors de chaque appel système ; c'est la destination du trap correspondant. C'est elle qui appelle la procédure de service correspondant à l'appel système demandé.
- Les procédures de service sont dédiées au traitement de chaque appel système. Pour ce faire, elles peuvent s'appuyer sur des procédures utilitaires.
- Les procédures utilitaires assistent les procédures de service.



1.8.2 Systèmes en couches

Une formalisation possible de la structure ci-dessus consiste à structurer le système en plusieurs couches, dont chacune s'appuie sur la couche immédiatement inférieure.

Le premier système à être conçu explicitement en couches a été le système THE développé au Technische Hogeschool d'Eindhoven en 1966, qui était un simple système de traitement par lots, décomposé en six couches.

- La couche 0 se chargeait de l'allocation du processeur, en commutant entre les processus à la suite d'interruptions ou d'expirations de délais. Au-dessus de la couche 0, le système était constitué de processus séquentiels, qui pouvaient être programmés indépendamment du fait que chacun partageait le processeur.
- La couche 1 s'occupait de la gestion de la mémoire et du va-et-vient (ii *swap* *ii*). Au-dessus de la couche 1, les processus n'avaient pas à se soucier de savoir s'ils se trouvaient dans la mémoire principale ou sur disque, le logiciel de la couche 1 se chargeant de ramener les pages disque dans la mémoire dès qu'un processus en avait besoin.
- La couche 2 était responsable de la communication entre les processus et la console de l'opérateur. Au-dessus de cette couche, chaque processus avait sa propre console.
- La couche 3 gérait les périphériques d'entrées/sorties, et plaçait les informations échangées dans des mémoires tampons. Chaque processus dialoguait, dès lors, avec des périphériques abstraits.
- La couche 4 contenait les programmes des utilisateurs.
- La couche 5 contenait le processus de l'opérateur système.

On peut deviner les limites de ce formalisme en constatant que la gestion des entrées/sorties, normalement très proche de la gestion matérielle, se trouve ici au-dessus de la communication processus/console.

Le système Multics était une généralisation du concept de couches. Il était organisé en une série d'anneaux concentriques, telle que chaque anneau disposait de plus de droits que les anneaux qui lui étaient extérieurs. Toute procédure d'un anneau désirant appeler une procédure d'un anneau inférieur devait exécuter un appel système, sous forme de trap, dont les paramètres étaient vérifiés avant d'autoriser l'accès. L'avantage de ce mécanisme, supporté par le

matériel, était de pouvoir permettre l'écriture de sous-systèmes protégés pour des utilisateurs ou des groupes d'utilisateurs.

1.8.3 Machines virtuelles

La notion de machine virtuelle est basée sur la constatation qu'un système à temps partagé offre, à la fois, la possibilité de partager le processeur, et une machine virtuelle dotée d'une interface plus pratique que la programmation directe du matériel. L'idée des machines virtuelles consiste à séparer complètement ces deux fonctions.

Ainsi, sur l'IBM 370, le système VM/370 était-il constitué de deux couches.

Application	Application	Application
CMS	CMS	CMS
VM/370		
Matériel du 370		

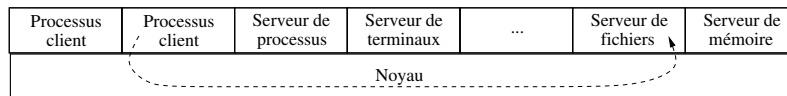
Le cœur du système, appelé moniteur de machine virtuelle, s'exécute juste au dessus du matériel et se charge de la multiprogrammation en fournissant à la couche supérieure plusieurs machines virtuelles. Cependant, contrairement aux autres systèmes d'exploitation, ces machines virtuelles ne sont pas des machines abstraites disposant de fonctionnalités étendues, mais des copies conformes du matériel sous-jacent, et notamment des modes noyau et utilisateur, et de la gestion des interruptions.

Chaque machine virtuelle, étant identique à la machine physique, peut faire tourner tout système d'exploitation qui s'exécuterait directement sur le matériel. En fait, les différentes machines virtuelles peuvent chacune disposer de systèmes d'exploitation différents. Ainsi, sous VM/370, la répartition typique était de faire tourner quelques machines virtuelles pour le traitement par lots, et d'autres sous un système mono-utilisateur interactif simple, appelé CMS (pour *Conversational Monitor System* \grave{a}).

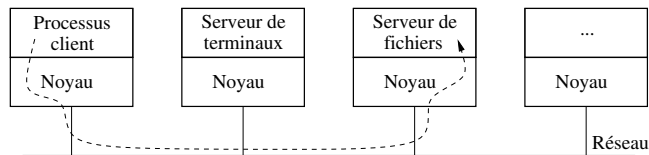
Lorsqu'un programme exécute un appel système sous CMS, l'appel est limité au système d'exploitation de sa propre machine virtuelle. CMS, traitant l'appel, exécute alors les instructions permettant d'effectuer l'entrée/sortie sur le matériel. Ces instructions sont dérivées par VM/370, qui les exécute dans sa simulation du matériel. Il faut cependant remarquer que la simulation efficace d'une machine réelle n'est pas simple du tout.

1.8.4 Systèmes client-serveur

Les systèmes client-serveurs sont basés sur une approche *horizontale* plutôt que *verticale*. Ils sont constitués d'un (micro-)noyau minimum, servant principalement à la communication, permettant aux processus clients d'effectuer des requêtes auprès des différents serveurs de ressources du système d'exploitation.



Le modèle client-serveur se prête extrêmement bien aux systèmes distribués : chaque machine du système exécute un exemplaire du micro-noyau, qui redirige les requêtes vers les serveurs concernés.



En pratique cependant, les latences induites par le réseau rendent les requêtes de base (processus, mémoire) trop coûteuses, et génèrent des goulots d'étranglement. On préfère donc adopter une approche hybride, dans laquelle ces services de base sont gérés par des serveurs locaux (bien que coopérant entre eux pour offrir un service et une vue globale), seuls les services déjà coûteux (fichiers, terminaux) pouvant être totalement distants.

Un autre problème inhérent à cette architecture est la définition de la frontière entre le mode noyau et le mode utilisateur. En effet, comme il est impossible d'autoriser un processus s'exécutant en mode utilisateur à accéder au matériel, les serveurs critiques doivent s'exécuter en mode noyau. Cependant, pour des raisons de sécurité et de maintenabilité, il est préférable que le moins de code possible s'exécute en mode noyau, et donc que les serveurs soient des processus indépendants s'exécutant en mode utilisateur.

La solution la plus couramment retenue est de n'implanter dans les noyaux que les *mécanismes* réalisant les opérations de bas niveau, en laissant aux serveurs la mise en œuvre des *politiques* de gestion des services.

Chapitre 2

Processus

2.1 Modèle

2.1.1 Notion de processus

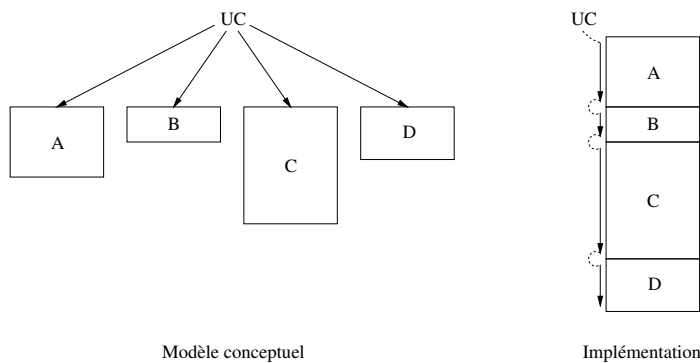
La notion de processus constitue un modèle simple et efficace pour représenter l'exécution concurrente de tâches au sein d'un système d'exploitation multi-tâches.

Le terme de *processus* a été utilisé pour la première fois au sein du système Multics, et popularisé depuis. Tout le travail effectué par l'ordinateur est réalisé par des processus séquentiels, qui représentent chacun une instance d'un programme séquentiel en train de s'exécuter.

Conceptuellement, un processus modélise l'exécution d'un programme sur un processeur virtuel disposant :

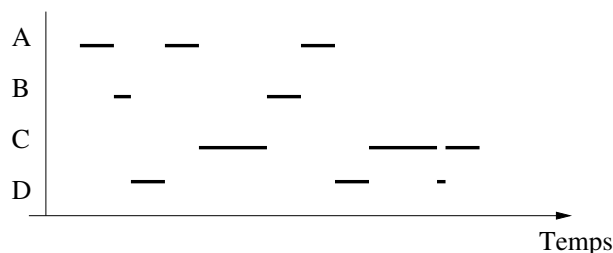
- d'un compteur ordinal ;
- d'un jeu de registres ;
- de mémoire (virtuelle), etc.

Le processeur physique commute entre les processus, sous la direction d'un ordonnanceur.



Dans le cas de systèmes à temps partagé, tous les processus progressent dans le

temps, mais un seul s'exécute à la fois.



La vitesse d'exécution d'un processus donné ne serait pas nécessairement la même si l'ensemble des processus est exécuté à nouveau, car l'état du système au lancement (position des têtes des disques, occupation des caches, ...) ainsi que le traitement d'événements asynchrones (trames réseau) peut conduire à des ordonnancements très différents.

De fait, il n'est pas possible dans un tel système de faire d'hypothèses sur le facteur temps, ce qui représente une complication pour le contrôle des périphériques d'entrées/sorties, qui nécessitent des traitements spécifiques au moyen d'interruptions matérielles.

2.1.2 Relations entre processus

Un processus ne pouvant conceptuellement être créé qu'au moyen d'un appel système, c'est-à-dire par un autre processus ayant réalisé cet appel, on voit apparaître naturellement la notion d'arborescence des processus, à partir d'un ancêtre commun à tous, créé au démarrage du système (processus `init` sous Unix).

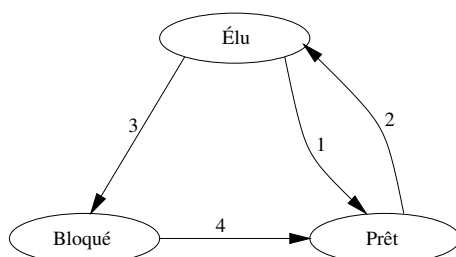
La notion de groupes de processus (*process groups*), définis comme des sous-arborescences, est également utilisée par certains systèmes pour limiter la portée des commandes de contrôle de processus (par exemple la commande `kill` lancée à partir de l'interpréteur de commandes, sous Unix).

2.1.3 États d'un processus

Les processus passent par des états discrets différents. On dit qu'un processus est :

- *élu* s'il est en cours d'exécution sur le processeur. Dans le cas d'une machine multi-processeurs, plusieurs processus peuvent être élus en même temps, mais il y a toujours au plus autant de processus élus que de processeurs (on peut le voir en tapant la commande `ps` sur une machine Unix multi-processeurs, et en regardant le nombre de processus indiqués comme *running* au même moment) ;
- *prêt* s'il est suspendu en faveur d'un autre. Un processus est prêt s'il ne lui manque que la ressource processeur pour s'exécuter ;
- *bloqué* s'il est en attente d'un événement externe (bloc disque, frappe clavier, ...).

Les transitions entre ces trois états sont matérialisées par l'automate suivant (voir [?, page 148] pour un automate complet).



La signification des quatre transitions légales est la suivante.

1. Le processus a épuisé le quantum de temps qui lui a été attribué. L'ordonnanceur, appelé de façon asynchrone par interruption lorsque le temps imparti est écoulé, élit un autre processus parmi les processus prêts.
2. L'ordonnanceur élit ce processus parmi les processus prêts.
3. Le processus s'endort en attente d'un événement externe (décompte d'horloge, attente de données, ...). L'ordonnanceur, appelé de façon explicite par le processus courant lorsque celui-ci ne peut progresser dans le traitement d'un appel système (attente d'événement asynchrone), élit un autre processus parmi les processus prêts.
4. L'événement attendu par le processus se produit. C'est le système d'exploitation qui gère son traitement (et pas le processus, puisqu'il est bloqué), de façon asynchrone, en interrompant temporairement le déroulement du processus actuellement élu pour traiter les données reçues, et faire passer le processus en attente de l'état *bloqué* à l'état *prêt*.

Avec ce modèle, l'exécution d'un processus peut progresser de deux façons :

- de façon synchrone, lorsque le processus est élu et dispose de la ressource processeur ;
- de façon asynchrone, lorsqu'un autre processus est élu, mais que celui-ci est interrompu par le système pour effectuer des traitements au profit du premier processus.

Dans le deuxième cas, le processus ne progresse pas lui-même, puisque son compteur ordinal reste figé là où le processus a été mis dans l'état *bloqué*, mais du travail est réalisé dans son intérêt.

2.2 Mise en œuvre

La gestion des processus et des interruptions représente toujours la couche la plus basse d'un système d'exploitation (même client-serveur!), car elle doit préexister pour qu'on conçoive le reste du système comme un ensemble de traitements séquentiels et concurrents.

2.2.1 Gestion des processus

Pour mettre en œuvre pratiquement le modèle des processus, le système dispose d'une table, appelée *ij* table des processus *i*, jamais swappée, dont chaque

entrée correspond à un processus particulier et contenant des informations telles que :

- les valeurs de son compteur ordinal, de son pointeur de pile, et des autres registres du processeur ;
- son numéro de processus, son état, sa priorité, son vecteur d'interruptions, et les autres informations nécessaires à la gestion du processus par le système ;
- son occupation mémoire ;
- la liste des fichiers ouverts par lui ;
- . . . , en fait, tout ce qui doit être sauvé par/pour lui lorsqu'il passe de l'état *élu* à l'état *prêt*.

Afin que cette table ne soit pas trop énorme, la plupart des systèmes disposent de deux zones par processus. La première, jamais swappée, contient les informations critiques dont le système a toujours besoin. La deuxième, contenue dans l'espace de chaque processus et donc swappable, contient l'information dont le processus n'aura besoin que lorsqu'il passera effectivement de l'état *prêt* à l'état *élu* (et ces données seront alors bien disponibles en mémoire à ce moment). Sous Unix, ces deux structures s'appellent `proc` (pour `process i`) et `u` (pour `user i`).

2.2.2 Gestion des interruptions

Le traitement des interruptions par le système s'effectue en appelant une routine de traitement associée à chaque type d'interruption, dont l'adresse est stockée dans les cases d'un tableau indexé par le type d'interruption, appelé `ii` vecteur d'interruptions `ii`. Par exemple, sous Unix, on distingue les types d'interruptions suivants [?, figure 6.9, page 163] :

Numéro d'interruption	Gestionnaire
0	Horloge
1	Disque
2	(Pseudo-)terminaux (<code>tty</code>)
3	Autres périphériques
4	Logiciel (<code>trap</code>)
5	Autres

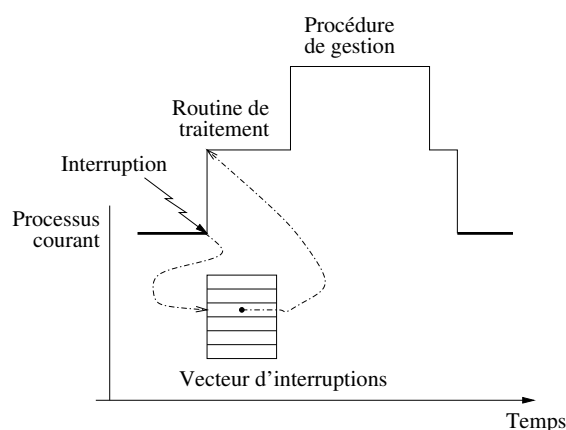
Comme on le voit ici, le `trap` logiciel est en fait une interruption classique, à la différence qu'elle n'est pas prise en compte par le processeur à partir de signaux externes provenant d'autres composants (comme les contrôleurs disques, par exemple), mais induite par l'exécution d'une instruction spécifique du langage machine (l'instruction `INT 21H` pour le DOS).

Des numéros d'interruptions différents sont couramment affectés aux différents périphériques du système, qui permettent d'accéder aux routines de traitement appropriées à travers le vecteur d'interruptions. Ces numéros, parfois paramétrables au niveau du matériel, sont appelés numéros `IRQ` (pour `Interrupt ReQuest i`).

Le traitement d'une interruption par le système s'effectue de la façon suivante :

- le processeur sauvegarde la valeur de son compteur ordinal dans la pile courante, détermine le type de l'interruption, passe en mode noyau, et

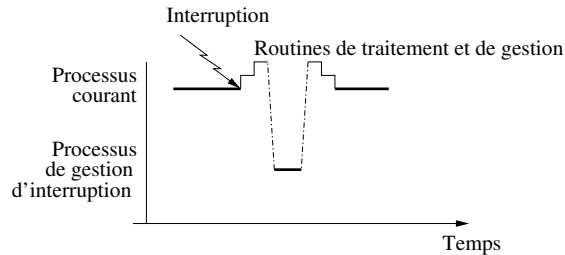
- charge la nouvelle valeur du compteur ordinal à partir de la valeur contenue dans le vecteur d'interruptions pour le type considéré ;
- la routine de traitement de l'interruption, écrite en langage machine, sauvegarde les registres du processeur dans la pile courante, ou bien construit une autre pile dédiée au traitement de l'interruption (voir [?, pages 161–167]), puis appelle la procédure principale de gestion de l'interruption, écrite en langage C ;
- au retour de la procédure de gestion, la routine de traitement restaure les registres du processeur, puis exécute une instruction de retour de procédure (RET ou IRET) pour recharger le compteur ordinal du processeur avec la valeur sauvegardée, qui est dépilée.



Sous Minix, le traitement des interruptions est basé sur un mécanisme client-serveur de type RPC (pour *Remote Procedure Call*), afin que le traitement effectif des interruptions soit effectué par des processus distincts du noyau [?, page 61] :

- comme ci-dessus, le processeur sauvegarde le compteur ordinal, puis charge sa nouvelle valeur à partir du vecteur d'interruptions ;
- la routine de traitement de l'interruption sauvegarde les registres du processeur et construit une nouvelle pile ;
- la procédure de gestion de l'interruption passe un message au processus de gestion de l'interruption indiquant l'arrivée d'une interruption, puis modifie l'état de ce processus en *prêt*, et appelle l'ordonnanceur ;
- l'ordonnanceur choisit le prochain procesus de traitement à exécuter. Les processus de gestion des interruptions sont toujours de plus haute priorité que les processus classiques. Cependant, si le processus interrompu est celui de plus haute priorité (c'est à dire lui-même un processus de gestion d'interruptions), il sera relancé d'abord ;
- le processus de gestion de l'interruption traite celle-ci, puis se rebloque en appelant l'ordonnanceur ;
- lorsque le processus interrompu est réélu, la routine de gestion de l'interruption termine et rend la main à la routine en langage machine de

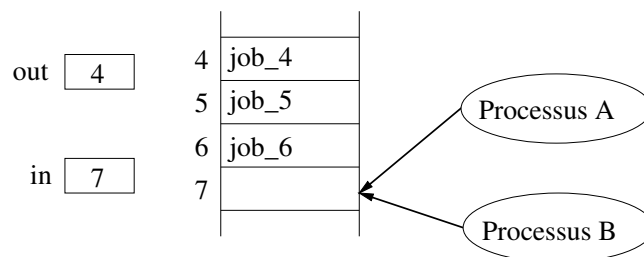
- traitement de l'interruption ;
- la routine de traitement de l'interruption restaure les registres du processeur, puis exécute une instruction de retour de procédure (RET ou IRET) pour recharger le compteur ordinal du processeur avec la valeur sauvegardée, qui est dépilée.



2.3 Communication inter-processus

Les processus d'un système ne s'exécutent pas tous de manière isolée. Certains processus ont besoin de coopérer, et nécessitent donc des moyens de communication et de synchronisation, et d'autres se trouvent en compétition pour les ressources du système, soit à cause de la nature physique de la ressource (non partageabilité), soit parce que les opérations sur cette ressource peuvent provoquer des incohérences ou des interblocages.

Pour illustrer ces problèmes, prenons comme exemple la sérialisation de travaux d'impression. Les tâches à imprimer sont stockées dans un répertoire de spoule, tel que chaque tâche reçoive à son arrivée un numéro d'ordre d'exécution strictement croissant (on ne se préoccupera pas ici des problèmes de débordement). Pour administrer ce répertoire, deux variables globales partagées *in* et *out* contiennent respectivement le prochain numéro de tâche à attribuer, et le prochain numéro de tâche à imprimer (on a donc toujours $in \geq out$). Supposons que, à un instant donné, on ait $in = 7$ et $out = 4$.



Si, au même moment, deux processus A et B souhaitent lancer l'impression d'un fichier, ils doivent chacun exécuter la séquence suivante :

```
local_in = in;
placer_job (local_in);
in = local_in + 1;
```

Si le premier processus à effectuer cette séquence est interrompu par l'ordonnateur entre la première et la troisième instruction, les deux processus utiliseront le même numéro pour placer leur tâche, et l'un des deux travaux sera perdu. Cependant, à la fin, le contenu du répertoire sera cohérent avec l'état des variables, et le processus d'impression ne pourra détecter aucune anomalie.

Il est donc nécessaire que le système fournisse un support sûr et efficace de la communication inter-processus, permettant tant la synchronisation que l'échange de données. Ce support peut être basé sur les mécanismes du système de fichiers (pipes, pipes nommés), ou bien sur ceux de gestion de la mémoire (mémoire partagée, sémaphores).

En fait, tous les problèmes de synchronisation se ramènent au problème de l'accès concurrent à une *variable partagée* (on parle dans la littérature anglo-saxonne de *race conditions*). La mise au point de programmes contenant de tels accès est difficile, car ces programmes peuvent fonctionner pendant de longues périodes de temps et produire des résultats incohérents de façon imprévisible.

2.4 Sections critiques

La solution conceptuellement naturelle à la résolution des problèmes d'accès concurrents consiste à interdire la modification de données partagées à plus d'un processus à la fois, c'est à dire définir un mécanisme d'*exclusion mutuelle* sur des portions spécifiques du code, appelées *sections critiques*.

On peut formaliser le comportement des sections critiques au moyen des quatre conditions suivantes :

1. deux processus ne peuvent être simultanément dans la même section critique ;
2. aucune hypothèse n'est faite sur les vitesses relatives des processus, ni sur le nombre de processeurs ;
3. aucun processus suspendu en dehors d'une section critique ne peut bloquer les autres ;
4. aucun processus ne doit attendre trop longtemps avant d'entrer en section critique.

La première condition est suffisante à elle seule pour éviter les conflits d'accès. Cependant, elle ne suffit pas à garantir le bon fonctionnement du système d'exploitation, en particulier en ce qui concerne l'égalité d'accès aux sections critiques.

Il existe de nombreux mécanismes pour mettre en œuvre l'exclusion mutuelle, dont chacun a ses avantages et inconvénients.

2.4.1 Masquage des interruptions

Le moyen le plus simple d'éviter les accès concurrents est de masquer les interruptions avant d'entrer dans une section critique, et de les restaurer à la sortie

de celle-ci. Ainsi, le processus en section critique ne pourra pas être suspendu au profit d'un autre processus, puisque l'inhibition des interruptions empêche l'ordonnanceur de s'exécuter.

Cette approche, qui autorise un processus utilisateur à masquer les interruptions du système, est inéquitable, car un processus se maintenant longtemps en section critique peut accaparer le processeur. Elle est de plus dangereuse, car le système peut être bloqué si le processus oublie de les restaurer.

Cependant, elle est couramment utilisée par le système d'exploitation lui-même, pour manipuler de façon sûre ses structures internes. Il faut néanmoins remarquer qu'elle n'est pas efficace pour les systèmes multi-processeurs, puisque les processus s'exécutant sur les autres processeurs peuvent toujours entrer en section critique.

2.4.2 Variables de verrouillage

L'inconvénient du masquage des interruptions est son inefficacité : lorsqu'un processus entre en section critique, tous les autres processus sont bloqués, même si la section critique ne concerne qu'un seul autre processus. Il faut donc pouvoir définir autant de sections critiques indépendantes que nécessaire.

Pour cela, on déclare une variable par section critique, qui joue le rôle de verrou. La variable est mise à 1 par le processus entrant dans la section critique considérée, et remise à 0 par le processus lorsqu'il quitte la section critique. Avant d'entrer en section critique, un processus doit donc tester l'état de la variable, et boucler en dehors de la section critique si elle a déjà été positionnée à 1 par un autre processus, selon l'algorithme suivant.

```
while (verrou == 1) ;      /* Attente      */
verrou = 1;               /* Verrouillage */
section_critique ();
verrou = 0;               /* Déverrouillage */
```

Malheureusement, cette solution présente le même défaut que l'algorithme de la section 2.3 : un processus peut être interrompu entre sa sortie de la boucle `while` et le positionnement du verrou, permettant ainsi à un autre processus de rentrer lui aussi en section critique.

On pourrait imaginer d'interdire les interruptions entre la fin du `while` et le positionnement du verrou. Cependant, pour être sûr d'être le seul à entrer en section critique, il faudrait inhiber les interruptions avant le `while`, ce qui conduirait à un blocage puisque l'ordonnanceur ne pourrait plus donner la main au processus actuellement en section critique pour qu'il sorte de celle-ci.

Une solution possible consiste à effectuer le `while` avec les interruptions activées et, si on sort du `while`, à réaliser un ultime test en inhibant les interruptions, en retournant au `while` si le test est infructueux.

```
loop:                      /* Étiquette de boucle externe */
while (verrou == 1) ;      /* Attente hors inhibition     */
s = splx (0);              /* Inhibition des interruptions */
```

```

if (verrou == 1) {           /* Si le verrou a déjà été remis */
    splx (s);                /* Restauration des interruptions */
    goto loop;              /* Boucle hors inhibition          */
}
verrou = 1;                 /* Verrouillage */
splx (s);                   /* Restauration */
section_critique ();
verrou = 0;                 /* Déverrouillage */

```

Cette méthode est analogue par le principe avec la méthode TSL décrite plus bas, en section 2.4.5. Elle est valable et efficace, mais uniquement dans le cas des machines mono-processeurs, puisque sur les machines multi-processeurs les processus s'exécutant sur les autres processeurs peuvent ici encore continuer à entrer en section critique.

2.4.3 Variable d'alternance

Le problème dans l'algorithme précédent est que les deux processus modifient la même variable en même temps. Pour éviter cela, on peut mettre en œuvre une variable *tour*, qui définit quel processus a le droit d'entrer en section critique.

```

int          tour;          /* Variable de tour */

while (1) {
    while (tour != p) ;    /* Attendre son tour */
    section_critique ();
    tour = 1 - p;         /* 0 --> 1, 1 --> 0 */
    section_non_critique ();
}

```

Cette solution, si elle est valide du point de vue de l'exclusion mutuelle, ne respecte pas les deuxième et troisième règles des sections critiques. En effet, lorsqu'un processus quitte la section critique, il s'interdit d'y revenir avant que l'autre processus n'y soit entré à son tour. Si les vitesses des deux processus sont très différentes, le plus rapide attendra toujours le plus lent.

2.4.4 Variables d'alternances de Peterson

La solution imaginée par Peterson améliore l'approche ci-dessus, en permettant au processus dont ce n'est pas le tour de rentrer quand-même en section critique, lorsque l'autre processus n'est pas encore prêt à le faire. Elle est basée sur deux procédures, *entrer_région* et *quitter_région*, qui encadrent la section critique.

```

int          tour;          /* Variable de tour */
int          intéressé[2]; /* Drapeaux          */

entrer_région (p)

```

```

{
    intéressé[p] = VRAI;      /* Veut rentrer      */
    tour        = p;        /* Est passé en dernier */
    while ((tour == p) &&    /* Attendre son tour   */
          (intéressé[1-p] == VRAI)) ;
}

quitter_région (p)
{
    intéressé[p] = FAUX;
}

```

Si l'un des deux processus désire entrer et que l'autre n'est pas intéressé à entrer, le premier processus entre immédiatement en section critique.

Tant que l'un des processus est en section critique, l'autre ne peut y entrer.

Si les deux processus sont en compétition pour entrer en section critique, le dernier à vouloir entrer écrase la valeur de la variable `tour` positionnée par le premier, et n'entre donc pas en section critique. En revanche, le premier processus, qui pouvait boucler en attente si le deuxième venait de positionner sa variable `intéressé`, entre effectivement en section critique.

Cette solution est valable et efficace.

2.4.5 Instruction TSL

Cette solution au problème de l'exclusion mutuelle est basée sur un support matériel, à savoir une instruction spécifique du langage machine. L'instruction `Test and Set, Lock` charge le contenu d'un mot mémoire dans un registre (elle est de ce point de vue analogue à l'instruction `LOAD`), puis met une valeur non-nulle à cet emplacement mémoire, ces deux opérations étant garanties comme indivisibles. Pour assurer l'atomicité de cette instruction, le processeur qui l'exécute verrouille le bus de données (d'où le `lock`), pour empêcher les autres processeurs d'accéder à la mémoire pendant la durée de l'opération.

En se basant sur cette instruction, on peut mettre en œuvre une exclusion mutuelle, en se servant d'une variable mémoire `verrou` initialement positionnée à 0, comme illustré ci-dessous dans un pseudo-langage machine.

```

INT    verrou = 0        // Verrou

entrer_région:
    TSL    reg, $verrou    // Lecture et positionnement du verrou
    CMP    reg, 0         // Test si verrou non déjà positionné
    JNZ    entrer_région  // Boucle si verrou déjà positionné
    RET

quitter_région:
    MOV    $verrou, 0     // Libération du verrou
    RET

```


Cette solution est valable et efficace, même dans le cas des systèmes multi-processeurs.

Le problème de toutes les approches décrites ci-dessus est qu'elles mettent en œuvre des mécanismes d'attente active (*spin lock*) : les processus désirant entrer en section critique consomment énormément de ressource processeur, et font un usage intensif du bus mémoire (ceci n'étant vraiment pénalisant que dans le cas d'un système multi-processeur).

De plus, dans le cas d'un système orienté temps-réel où les priorités relatives des processus sont strictement respectées, si un processus H de haute priorité désire entrer en section critique alors qu'un processus B de basse priorité s'y trouve déjà, H aura toute la ressource processeur pour boucler et B ne pourra jamais quitter la section critique, d'où blocage des deux processus.

Il faut donc mettre en œuvre des mécanismes d'exclusion mutuelle qui bloquent les processus en attente ; ceci ne peut se faire qu'au niveau du système d'exploitation, puisqu'il faut contrôler l'ordonnanceur de manière à ne pas accorder la ressource processeur aux processus en attente (par leur mise en dehors de la liste des processus prêts). Plusieurs formalismes ont été définis dans ce but.

2.4.6 Primitives sleep et wakeup

Les primitives `sleep` et `wakeup` sont deux appels système. La primitive `sleep` endort le processus qui l'appelle, en rendant la main à l'ordonnanceur, alors que `wakeup` permet à un processus de réveiller un processus endormi dont l'identifiant lui est passé en paramètre.

Ces appels système peuvent servir à mettre en œuvre des mécanismes de type producteur-consommateur, comme illustré ci-dessous.

```
#define N      100          /* Taille de la file      */
int           compteur = 0; /* Nombre courant d'objets */

producteur () {           /* Processus producteur   */
    while (1) {           /* Boucle infinie        */
        produire_objet (o); /* Produire un objet     */
        if (compteur == N) /* Si la file est pleine */
            sleep ();      /* Attente de retrait    */
        mettre_objet (o); /* Mettre l'objet en file */
        compteur ++;      /* Un de plus dans la file */
        if (compteur == 1) /* Si la file était vide */
            wakeup (consommateur); /* Réveiller le consommateur */
    }
}

consommateur () {        /* Processus consommateur */
    while (1) {          /* Boucle infinie        */
        if (compteur == 0) /* Si la file est vide   */
            sleep ();      /* Attente de dépôt     */
        retirer_objet (o); /* Retirer l'objet de la file */
        compteur --;      /* Un de moins dans la file */
    }
}
```

```

    if (compteur == (N-1))    /* Si la file était pleine */
        wakeup (producteur); /* Réveiller le producteur */
        consommer_objet (o); /* Consommer l'objet courant */
    }
}

```

Cette solution est élégante, mais conduit aux mêmes conflits d'accès que pour l'exemple du spoule décrit en section 2.3, du fait de l'accès concurrent à la variable `compteur`. Considérons en effet la situation suivante : la file est vide, et le consommateur vient de lire la valeur 0 dans la variable `compteur` ; l'ordonnanceur l'interrompt et donne la main au producteur, qui insère un objet dans la file, incrémente le compteur, et appelle `wakeup` pour réveiller le consommateur ; comme celui-ci n'était pas endormi, l'appel `wakeup` est sans effet ; l'ordonnanceur donne la main au consommateur, qui appelle la routine `sleep`.

Comme le producteur n'appellera plus la routine `wakeup`, et que le consommateur est endormi, le producteur remplira la file jusqu'à ce qu'elle soit pleine, puis s'endormira à son tour, et donc les deux processus seront bloqués (il y a *deadlock* `;;`, ou *ij* étreinte fatale `;;`).

Ici encore, les problèmes apparaissent du fait de la non-atomicité du test et de l'action qui lui est associée.

Une première solution consiste à inhiber les interruptions avant les tests et à ne les restaurer qu'après. Cette solution est techniquement efficace, car le code de la fonction `sleep` est petit (il n'est en effet pas nécessaire ici de protéger les tests sur `wakeup`). Cependant, comme dit dans la section 2.4.1, il n'est pas possible de laisser le contrôle des interruptions à l'utilisateur, et cette solution doit être réservée au noyau (qui l'utilise effectivement lorsque nécessaire).

Une deuxième solution consisterait à stocker un bit de réveil en attente lorsqu'un processus fait l'objet d'un `wakeup` alors qu'il n'est pas endormi, et à n'endormir un processus que si son bit de réveil est à zéro (sinon, le bit est remis à zéro et le processus ne s'endort pas). Cependant, on peut exhiber des situations à producteurs et/ou consommateurs multiples, pour lesquelles autant de bits que de processus seraient nécessaires.

2.4.7 Sémaphores

L'idée des sémaphores est d'utiliser une variable entière pour compter le nombre de réveils en attente, et d'encapsuler cette variable dans un objet système manipulable seulement à partir d'appels systèmes spécifiques.

Un sémaphore possède la valeur 0 si aucun réveil n'a été mémorisé, et une valeur positive s'il y a un ou plusieurs réveils en attente. L'appel système `down` décrémente la valeur du sémaphore lorsque celle-ci est supérieure à 0, ou endort le processus appelant si elle était déjà à 0. L'appel système `up` incrémente la valeur du sémaphore si aucun processus n'est endormi du fait d'un appel `down`, ou réveille l'un de ces processus (choisi au hasard) sinon.

La manipulation des sémaphores à travers des appels systèmes permet d'inhiber de façon sûre les interruptions pour garantir leur atomicité, puisque cette inhibition se fait sous le contrôle du système et non de l'utilisateur.

Dans le cas d'un système multi-processeurs, il faudra protéger chaque sémaphore par une variable verrou manipulée au moyen de l'instruction TSL, pour s'assurer qu'un seul processeur à la fois accède à un sémaphore. Dans ce cas précis, l'attente active par TSL est préférable de loin à une attente passive, car les manipulations des variables sémaphores sont bien moins coûteuses qu'un changement de contexte, et car un verrou positionné dans ce cadre est sûr d'être déverrouillé rapidement puisque le processus qui a effectué le verrouillage est en mode noyau sur un processeur dont les interruptions sont inhibées.

```

#define N      100          /* Taille de la file          */
semaphore     mutex;      /* Sémaphore d'exclusion mutuelle */
semaphore     vide;       /* Nombre de places vides      */
semaphore     plein;      /* Nombre de places occupées   */

sem_init (mutex, 1);      /* Un seul processus en section critique */
sem_init (vide, N);      /* La file est toute vide          */
sem_init (plein, 0);     /* Aucun emplacement occupé       */

producteur () {          /* Processus producteur        */
    while (1) {          /* Boucle infinie              */
        produire_objet (o); /* Produire un objet          */
        sem_down (vide);  /* On veut une place vide     */
        sem_down (mutex); /* On bloque la file         */
        mettre_objet (o); /* Mettre l'objet en file     */
        sem_up (mutex);   /* Libération de la file     */
        sem_up (plein);   /* Un objet est à prendre    */
    }
}

consommateur () {       /* Processus consommateur      */
    while (1) {          /* Boucle infinie              */
        sem_down (plein); /* Attente d'un objet         */
        sem_down (mutex); /* On bloque la file         */
        retirer_objet (o); /* Consommer l'objet courant */
        sem_up (mutex);   /* Libération de la file     */
        sem_up (vide);    /* Une place est à prendre   */
        consommer_objet (o); /* Consommer l'objet courant */
    }
}

```

L'utilisation des sémaphores, bien qu'amenant une simplification conceptuelle de la gestion des exclusions mutuelles, est encore fastidieuse et propice aux erreurs. Ainsi, si l'on inverse par mégarde les deux `down` dans le code du consommateur, et si la file est vide, le consommateur se bloque sur le sémaphore `plein` après avoir positionné le sémaphore `mutex`, et le producteur se bloque sur son appel à `mutex` sans pouvoir débloquent le sémaphore `plein`, réalisant ainsi un interblocage.

2.4.8 Moniteurs

L'idée des moniteurs est d'offrir le support des exclusions mutuelles au niveau du langage. Un moniteur est constitué d'un ensemble de données et de procédures regroupées dans un module ayant comme propriété que seul un processus peut être actif dans le moniteur à un instant donné.

```
moniteur file {
    int      compteur = 0;
    ...
    procédure mettre (Objet o) {
    }
    ...
    procédure retirer (Objet o) {
    }
}
```

C'est le compilateur qui gère l'exclusion mutuelle au sein du moniteur, typiquement en ajoutant un sémaphore verrou aux données du moniteur et en insérant des instructions de test de ce verrou au début de chaque procédure du moniteur, afin d'endormir les processus appelants lorsqu'un processus est déjà entré dans le moniteur. L'exclusion mutuelle étant assurée par le compilateur et non par le programmeur, le risque d'erreur est bien plus faible.

Les moniteurs garantissent l'exclusion mutuelle, mais un autre mécanisme est nécessaire pour bloquer un processus entré dans un moniteur et devant attendre une ressource, afin qu'un autre processus puisse entrer à son tour dans le moniteur¹ pour lui apporter la ressource dont il a besoin.

Cette fonctionnalité est assurée par ce qu'on appelle des variables de condition, qui sont en fait des sortes de sémaphores, et auxquelles sont associées les primitives `wait` et `signal`². La primitive `wait` bloque le processus appelant sur la variable de condition passée en paramètre, alors que la primitive `signal` permet à l'un des processus bloqués sur la variable de se réveiller. Cependant, le processus réveillant et le processus réveillé ne pouvant être actifs en même temps dans le moniteur, l'ordonnanceur doit en choisir un (ce sera le plus souvent le processus appelant, puisqu'il dispose déjà de la ressource processeur), qui devra sortir du moniteur avant que l'autre puisse poursuivre son exécution.

Les primitives `wait` et `signal` ressemblent beaucoup aux primitives `sleep` et `wakeup`. Elles n'en ont cependant pas les défauts, car leur utilisation exclusive à l'intérieur d'un moniteur garantit l'exclusion mutuelle, et annule donc le risque d'exécution concurrente que l'on avait avec `sleep` et `wakeup`.

```
moniteur file {
    condition plein, vide;
    int      compteur = 0;
    ...
}
```

1. Remarquez bien que plusieurs processus peuvent se trouver simultanément dans un moniteur; la seule contrainte des moniteurs est qu'un seul processus à la fois peut y être *actif*.

2. Ces primitives n'ont rien à voir avec les appels système `wait()` et `signal()` d'Unix; ce sont des faux amis.

```

procédure mettre (Objet o) {
  if (compteur == N)
    wait (plein);
  mettre_objet (o);
  compteur ++;
  if (compteur == 1)
    signal (vide);
}
procédure retirer (Objet o) {
  if (compteur == 0)
    wait (vide);
  retirer_objet (o);
  compteur --;
  if (compteur == (N - 1))
    signal (plein);
}
}

```

Une implémentation effective des moniteurs a été réalisée dans le langage Java, au moyen du qualificatif/mot-clé `synchronized`, et des méthodes `wait` et `notify` [?]. Cette filiation est explicite, car le fait d'appeler les méthodes `wait` ou `notify` depuis un bloc de code non synchronisé entraîne la levée d'une exception de type `IllegalMonitorStateException`. Cependant, il n'existe pas en Java de variables de condition.

2.4.9 Échange de messages

Le problème de synchronisation sur une ressource partagée peut être reformulé en terme de communication inter-processus, et implémenté sous la forme de deux primitives (appels systèmes) `send` et `receive`. La primitive `send` envoie un message à un processus donné, alors que la primitive `receive` permet à un processus de lire un message provenant d'un processus, donné ou quelconque (au choix du récepteur); si aucun message n'est disponible, `receive` bloque le processus appelant.

Cette solution, qui permet l'échange d'informations entre processus d'une architecture à mémoire distribuée, est beaucoup plus complexe à mettre en œuvre que les précédentes, car elle nécessite de résoudre plusieurs problèmes majeurs :

- la perte possible de messages ;
- le nommage des processus ;
- l'authentification des messages ;
- le surcoût dû à la latence des communications ;
- la tolérance aux pannes.

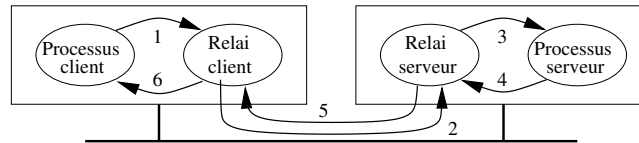
La perte possible de messages se traite en utilisant un protocole d'acquiescement des messages émis : tout message reçu doit être suivi de l'envoi d'un message d'acquiescement informant l'émetteur que son message a bien été reçu et compris. Pour éviter les doublons résultant de la perte d'un acquiescement, les

messages sont numérotés, afin qu'un récepteur recevant plusieurs fois le même message puisse ignorer les messages retransmis. Pour des raisons d'efficacité, les acquittements peuvent être groupés, et même être inclus dans les messages émis par le récepteur en direction de l'émetteur. Un bon exemple d'un tel protocole est le protocole de transmission à fenêtre glissante HDLC.

Le nommage des processus est un problème difficile. Chaque processus peut facilement disposer d'un nom unique sur chaque machine, attribué par le système d'exploitation (de type PID), mais offrir un service de nommage global permettant la communication entre processus non apparentés est un réel problème. Tout d'abord, cela nécessite l'existence d'un espace de nommage commun à l'ensemble des processus, avec tous les problèmes de goulot d'étranglement posés par l'accès à cette ressource critique. Qui plus est, il est théoriquement impossible d'empêcher deux processus différents de vouloir porter le même nom, et donc d'empêcher les conflits entre applications différentes (ou instances de la même application) si par malheur leurs processus se déclarent avec des noms identiques; seule la détection de tels conflits est possible.

L'authentification des messages peut être réalisée au moyen de crypto-systèmes à clés publiques et privées, mais cela nécessite ici encore le recours à un serveur de clés globalement accessible qui constitue encore un goulot d'étranglement.

Le surcoût constitue un réel problème du fait des intermédiaires mis en jeu, comme par exemple lorsqu'on implémente un appel de procédure à distance (ii *Remote Procedure Call* $\llcorner\llcorner$, ou RPC) sous forme d'échange de messages.



Ce type d'appel interdit naturellement le passage par référence, et nécessite donc des recopies mémoire supplémentaires, qui doivent prendre en compte l'hétérogénéité des architectures.

Les appels de procédure à distance induisent également un problème de tolérance aux pannes: lorsque la réponse ne parvient pas à l'appelant, il est impossible de savoir si l'opération demandée a effectivement été réalisée ou non. Les systèmes ii au moins une fois $\llcorner\llcorner$ (ii *at least once* $\llcorner\llcorner$) garantissent que l'appel sera retransmis, au moins une fois, jusqu'à obtention d'un acquittement; les systèmes ii au plus une fois $\llcorner\llcorner$ (ii *at most once* $\llcorner\llcorner$) assurent qu'un appel ne sera transmis qu'au plus une fois; les systèmes ii peut-être $\llcorner\llcorner$ (ii *maybe* $\llcorner\llcorner$) ne garantissent rien, mais sont les plus simples à réaliser.

La réalisation d'un système de type producteur-consommateur par envois de messages peut se réaliser assez simplement si l'on suppose que le système assure lui-même le stockage temporaire (ii *bufferisation* $\llcorner\llcorner$) des messages en attente de lecture.

```
#define N      100                /* Taille de la file */

producteur () {                  /* Processus producteur */
    while (1) {                  /* Boucle infinie      */
        produire_objet (o);      /* Produire un objet   */
    }
}
```

```

    recevoir (consommateur, ""); /* Consommer un jeton */
    envoyer (consommateur, o); /* Envoyer un objet */
  }
}

consommateur () {
    /* Processus consommateur */
    for (i = 0; i < N-1; i ++) /* Envoyer N jetons */
        envoyer (producteur, ""); /* (messages vides) */
    while (1) { /* Boucle infinie */
        envoyer (producteur, ""); /* Générer un nouveau jeton */
        recevoir (producteur, o); /* Recevoir un objet */
        consommer_objet (o); /* Consommer l'objet courant */
    }
}

```

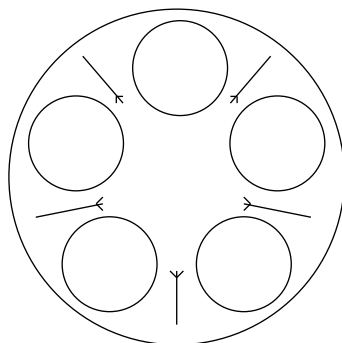
Lorsque le système n'assure pas la bufférisation des messages en attente, on obtient en prenant $N = 1$ un mécanisme synchrone de type `ii` rendez-vous `ii`.

2.5 Problèmes classiques de communication inter-processus

2.5.1 Le problème des philosophes

Le problème dit `ii` des philosophes `ii` [Dijkstra, 1965] est un problème classique d'accès concurrent qui a servi de référence pour évaluer l'efficacité des différents modèles de primitives de synchronisation proposées au fil du temps.

Cinq philosophes sont assis autour d'une table ronde, et ont chacun devant eux une assiette pleine de spaghetti tellement glissants qu'il faut deux fourchettes pour les manger. Une fourchette est disposée entre deux assiettes consécutives.



Un philosophe passe son temps à penser et à manger. Lorsqu'un philosophe a faim, il tente de s'emparer des deux fourchettes qui sont immédiatement à sa gauche et à sa droite, l'une après l'autre, l'ordre n'importe pas. S'il obtient les deux fourchettes, il mange pendant un certain temps, puis repose les fourchettes et se remet à penser. Le problème consiste à fournir un algorithme permettant à chaque philosophe de se livrer à ses activités sans jamais être bloqué de façon irrémédiable.

Une solution fautive est la suivante.

```
#define N      5                /* Nombre de philosophes */

philosophe (i) {                /* Numéro de 0 à (N-1) */
  while (1) {                    /* Boucle infinie      */
    penser ();
    prendre_fourchette (i); /* Bloque tant que pas obtenue */
    prendre_fourchette ((i + 1) % N); /* Bloque pareillement */
    manger ();
    poser_fourchette (i);
    poser_fourchette ((i + 1) % N);
  }
}
```

Ce programme peut conduire à des interblocages si tous les philosophes décident de manger en même temps. On peut modifier le programme précédent en libérant la fourchette gauche si la droite est prise, mais alors il peut arriver que les philosophes ne mangent jamais s'ils sont synchronisés (on parle de *ij* famine \llcorner , ou *ij* *starvation* \llcorner). On peut rendre le programme statistiquement correct en introduisant des délais aléatoires, mais une solution systématique et déterministe est largement préférable si elle existe.

Une solution déterministe correcte et efficace est la suivante.

```
#define N      5                /* Nombre de philosophes */
#define GAUCHE ((i+N-1) % N) /* Numéro voisin gauche  */
#define DROITE ((i+1) % N)   /* Numéro voisin droit   */

typedef enum {
  PENSE,
  A_FAIM,
  MANGE
} Philosophe;

semaphore mutex;                /* Exclusion mutuelle      */
semaphore s[N];                 /* Un sémaphore par philosophe */
Philosophe p[N];                /* État des philosophes    */

philosophe (i) {                /* Numéro de 0 à (N-1) */
  while (1) {                    /* Boucle infinie      */
    penser ();
    prendre_fourchettes (i); /* Prend les deux fourchettes ou bloque */
    manger ();
    poser_fourchettes (i);
  }
}

test (i) {
  if ((p[i] == A_FAIM) && /* Si i demande les fourchettes */
      (p[GAUCHE] != MANGE) && /* Et qu'elles sont toutes libres */
      (p[DROITE] != MANGE)) {
    p[i] = MANGE;            /* Récupère (implicitement) les fourchettes */
  }
}
```



```

    up (s[i]);          /* Réveille le philosophe i s'il dormait */
  }
}

prendre_fourchettes (i) {
  down (mutex);        /* Section critique */
  p[i] = A_FAIM;       /* Demande les fourchettes */
  test (i);            /* Essaie de les prendre */
  up (mutex);          /* Fin section critique */
  down (s[i]);         /* S'endort si pas prises */
}

poser_fourchettes (i) {
  down (mutex);        /* Section critique */
  p[i] = PENSE;        /* Ne veut plus manger */
  test (GAUCHE);      /* Propose au philosophe de gauche */
  test (DROITE);       /* Propose au philosophe de droite */
  up (mutex);          /* Fin section critique */
}

```

2.5.2 Le problème des lecteurs et des rédacteurs

Le problème des lecteurs et des rédacteurs est un autre problème classique, qui modélise l'accès à une base de données, où plusieurs processus peuvent lire simultanément la base mais un seul processus peut y écrire à la fois.

Une solution déterministe correcte et efficace est la suivante.

```

semaphore mutex;      /* Exclusion mutuelle */
semaphore verrou;     /* Contrôle d'accès à la base */
int    lecteurs;      /* Nombre de lecteurs potentiels */

lecteur () {
  down (mutex);        /* Section critique */
  lecteurs ++;        /* Veut lire */
  if (lecteurs == 1)  /* Si premier lecteur */
    down (verrou);    /* Verrouille la base */
  up (mutex);          /* Fin section critique */
  lecture ();
  down (mutex);        /* Section critique */
  lecteurs --;        /* Ne veut plus lire */
  if (lecteurs == 0) /* Si dernier lecteur */
    up (verrou);      /* Déverrouille la base */
  up (mutex);          /* Fin section critique */
}

ecrivain () {
  down (verrou);
  écriture ();
  up (verrou);
}

```

2.6 Ordonnancement des processus

Dans un système d'exploitation, il est courant que plusieurs processus soient simultanément prêts à s'exécuter. Il faut donc réaliser un choix pour ordonner dans le temps les processus prêts sur le processeur, qui est dévolu à un ordonnanceur.

Pour les systèmes de traitement par lots, l'algorithme d'ordonnancement est relativement simple, puisqu'il consiste à exécuter le programme suivant de la file dès qu'un emplacement se libère dans la mémoire de l'ordinateur (multi-programmation).

Pour les systèmes multi-utilisateurs, multi-tâches, et multi-processeurs, l'algorithme d'ordonnancement peut devenir très complexe.

Le choix d'un algorithme d'ordonnancement dépend de l'utilisation que l'on souhaite faire de la machine, et s'appuie sur les critères suivants :

- équité : chaque processus doit pouvoir disposer de la ressource processeur ;
- efficacité : l'utilisation du processeur doit être maximale ;
- temps de réponse : il faut minimiser l'impression de temps de réponse pour les utilisateurs interactifs ;
- temps d'exécution : il faut minimiser le temps d'exécution pris par chaque travail exécuté en traitement par lots ;
- rendement : le nombre de travaux réalisés par unité de temps doit être maximal.

En fait, plusieurs de ces critères sont mutuellement contradictoires, et l'on a montré [Kleinock 1975] que tout algorithme d'ordonnancement qui favorise une catégorie de travaux le fait au détriment d'une autre.

Qui plus est, rien ne permet de connaître à l'avance les demandes en ressources de chacun des processus (E/S, mémoire, processeur) au cours de leur exécution, et donc le temps passé entre deux appels système. Pour assurer l'équité entre processus, il est donc nécessaire de mettre en œuvre un mécanisme de temporisation, afin de rendre la main à l'ordonnanceur pour que celui-ci puisse déterminer si le processus courant peut continuer ou doit être suspendu au profit d'un autre. On effectue alors un ordonnancement avec réquisition (*preemptive scheduling*) du processeur, bien plus complexe à réaliser que le simple ordonnancement par exécution jusqu'à achèvement, car il implique la possibilité de conflits d'accès qu'il faut prévenir au moyen de mécanismes délicats (sémaphores ou autres).

2.6.1 Ordonnancement circulaire

Le mécanisme d'ordonnancement circulaire (*round robin*, ou tourniquet) est l'un des plus simples et des plus robustes. Il consiste à attribuer à chaque processus un quantum de temps pendant lequel il a le droit de s'exécuter.

Si un processus s'exécute jusqu'à épuisement de son quantum, le processeur est réquisitionné par l'ordonnanceur et attribué à un autre processus. Si, en revanche, le processus se bloque (attente passive) ou termine avant la fin de son quantum, le processeur est immédiatement attribué à un autre processus.

Cet algorithme se met aisément en œuvre au moyen d'une liste circulaire doublement chaînée.

En fait, la principale question relative à l'algorithme d'ordonnement circulaire concerne le choix de la durée du quantum par rapport à celle d'un changement de contexte. Un quantum trop petit conduit à un gaspillage de la ressource processeur, alors qu'un quantum trop grand réduit fortement l'interactivité du système lorsque de nombreux processus sont en attente d'exécution. En pratique, un quantum d'une centaine de milli-secondes constitue un compromis acceptable.

2.6.2 Ordonnement avec priorité

Dans la plupart des cas, on souhaite mettre en œuvre un mécanisme de priorité, afin de pouvoir favoriser certaines classes de processus par rapport à d'autres. Les critères de priorité peuvent dépendre du type de travail réalisé, de l'utilisateur demandeur du service, ou du prix payé pour celui-ci.

Les priorités peuvent être modifiées dynamiquement par le système afin d'atteindre certains de ses objectifs. Par exemple, certains processus qui réalisent beaucoup d'E/S passent en fait la majeure partie de leur existence dans le système à être bloqués en attente de la fin de leurs E/S. Du point de vue du système, ces processus mobilisent inutilement de la mémoire tant qu'ils sont bloqués, et il est donc souhaitable qu'ils puissent lancer leurs demandes d'E/S le plus vite possible afin que leurs requêtes s'exécutent en parallèle d'un autre processus, pendant qu'ils seront bloqués en attente. Pour cela, un moyen simple consiste à attribuer comme priorité à un processus la valeur de la fraction restante du quantum de temps utilisé par le processus lors de sa dernière élection. On peut moyenner ces valeurs au cours du temps afin de lisser les artefacts

Il est souvent pratique de regrouper les processus par classes de priorité, et d'utiliser l'ordonnement par priorité entre classes, et l'ordonnement circulaire entre les processus d'une même classe. Cependant, si les priorités ne sont pas ajustées dynamiquement, certains processus risquent de ne jamais être exécutés.

On peut remarquer que les processus de plus faible priorité sont les plus gros consommateurs de ressource processeur. Afin d'améliorer l'efficacité du système lorsqu'aucun processus interactif n'est prêt, il est donc possible d'attribuer aux processus de plus basse priorité un quantum de temps plus grand, afin de minimiser le nombre de changements de contexte.

Le problème majeur des approches par classes est la gestion du changement de classe lorsque le processus change de comportement. Ainsi, il faut pouvoir redonner une haute priorité aux processus qui ont calculé longtemps avant de redevenir interactifs. Une solution possible consiste à remonter un processus réalisant une E/S dans la plus haute classe possible, en supposant que ce processus va redevenir interactif. Un des problèmes inhérents à cette approche est relaté sous forme d'anecdote dans [?, page 96] : un utilisateur dont le processus consommait beaucoup de temps CPU remarqua qu'il suffisait de taper un retour chariot toutes les quelques secondes pour diminuer considérablement le temps d'exécution de son programme, et en fit part à tous ses amis...

2.6.3 Ordonnancement du plus court d'abord

Ce type d'ordonnancement s'applique lorsqu'on dispose d'un ensemble de tâches dont on peut connaître la durée à l'avance, comme par exemple dans le cas d'un traitement par lots de transactions journalières bancaires.

Si la file d'attente contient plusieurs tâches de même priorité, on minimise le temps de réponse en effectuant toujours d'abord celle ayant le temps le plus court.

En effet, si l'on dispose de n tâches de durées $t_1 \leq t_2 \leq \dots \leq t_n$, que l'on ordonne dans un certain ordre i, j, \dots, k , le temps de réponse moyen est :

$$T = \frac{t_i + (t_i + t_j) + \dots + (t_i + t_j + \dots + t_k)}{n} = \frac{n t_i + (n-1) t_j + \dots + t_k}{n} .$$

La contribution de plus grand poids étant celle de t_i , on minimisera T en prenant $t_i = t_1$, puis $t_j = t_2$, et ainsi de suite...

Cette technique peut être appliquée aux processus interactifs en considérant que chaque commande tapée par l'utilisateur constitue une tâche indépendante. Le problème est alors de connaître le temps que prendra une commande donnée. Ceci n'est pas possible, mais on peut l'estimer en se basant sur le temps des commandes précédentes.

Une solution possible consiste à estimer le temps de la commande suivante en calculant une moyenne sur les temps des dernières commandes effectuées, pondérée de façon à donner plus d'importance à la commande précédente (on effectue en fait un filtrage passe-bas sur le signal échantillonné des temps). Pratiquement, on utilise un coefficient a , avec $0 \leq a \leq 1$, tel que le temps estimé T_i^e à l'étape i soit égal à $a T_{i-1}^m + (1-a) T_{i-1}^e$, où T_{i-1}^e et T_{i-1}^m sont respectivement les temps estimé et mesuré à l'étape $i-1$. Ainsi, on aura, avec $a = \frac{1}{2}$, :

$$\begin{aligned} T_1^e &= T_0^m , \\ T_2^e &= \frac{1}{2} T_1^m + \frac{1}{2} T_0^m , \\ T_3^e &= \frac{1}{2} T_2^m + \frac{1}{4} T_1^m + \frac{1}{4} T_0^m , \\ T_4^e &= \frac{1}{2} T_3^m + \frac{1}{4} T_2^m + \frac{1}{8} T_1^m + \frac{1}{8} T_0^m , \\ &\dots \end{aligned}$$

Cette méthode de calcul, qui fait décroître le poids des mesures les plus anciennes, est appelée *méthode du vieillissement* (ii *aging* li).

On peut remarquer que la méthode d'ordonnancement du plus court d'abord n'est optimale que lorsque l'ensemble des travaux est initialement disponible.

2.6.4 Ordonnancement dicté par une politique

L'objectif de ce type d'ordonnancement est de garantir à l'utilisateur une performance annoncée.

Un exemple simple d'une telle règle est par exemple que tout utilisateur connecté doit recevoir une fraction équitable de la puissance du processeur. Pour respecter cette règle, le système doit mémoriser le temps processeur consommé par chaque utilisateur depuis la plus récente connexion, et comparer ce temps avec le temps écoulé depuis cette connexion divisé par le nombre d'utilisateurs actuellement connectés. L'algorithme consiste alors à exécuter en priorité les processus ayant le rapport le plus faible.

Un algorithme semblable peut être utilisé pour les systèmes temps-réel, pour lesquels les délais doivent impérativement être respectés. Pour cela, on exécute en priorité les processus les plus proches de leur date d'expiration : un processus qui doit produire un résultat sous dix secondes est prioritaire sur un qui doit avoir terminé sous dix minutes...

2.6.5 Ordonnement à deux niveaux

Tous les mécanismes d'ordonnement considérés jusqu'à présent supposaient que l'ensemble des processus était accessible en mémoire. Cependant, si la mémoire physique est limitée, et que l'on dispose d'un mécanisme de va-et-vient (*swapping*), le temps de commutation vers un processus diffère de plusieurs ordres de grandeur selon que ce processus est déjà situé en mémoire ou déporté sur le disque, et cette latence doit être prise en compte par l'algorithme d'ordonnement.

Pour cela, on met en place un ordonnanceur à deux niveaux :

- un ordonnanceur de bas niveau, semblable à ceux déjà vus, s'occupe des processus présents en mémoire ;
- un ordonnanceur de haut niveau, de temps en temps (quand l'ordonnanceur de bas niveau lui alloue le processeur !), échange les processus entre la mémoire et le disque.

Pour décider de migrer un processus donné depuis le disque vers la mémoire centrale, ou réciproquement, l'ordonnanceur de haut niveau peut se baser sur les critères suivants :

- le temps écoulé depuis le dernier va-et-vient du processus ;
- la quantité de temps processeur récemment consommée par le processus ;
- la taille du processus (les petits sont moins pénalisants pour le va-et-vient) ;
- la priorité du processus.

Sous Unix, l'ordonnanceur de haut niveau est un processus particulier, le *swapper*, qui s'exécute en mode noyau, et accède directement aux structures du système plutôt que d'utiliser des appels système. C'est le seul processus chargé de charger les processus en mémoire depuis les périphériques de va-et-vient [?, page 280 et suivantes]. Il exécute une boucle infinie, et chaque fois qu'il est actif cherche à migrer des processus depuis le disque vers la mémoire, en migrant éventuellement des processus de la mémoire vers le disque s'il lui faut de la place.

En revanche, la migration des processus depuis la mémoire vers les périphériques de va-et-vient est assurée conjointement par le *swapper* et par le noyau, celui-ci se chargeant directement de la migration des processus vers le disque dans l'un des trois cas suivants :

- l'appel système `fork()` doit allouer de la place pour le processus fils ;
- l'appel système `brk()` augmente la taille des données d'un processus ;
- un processus grossit du fait de l'augmentation de la taille de sa pile.

Lorsque le *swapper* se réveille pour charger des processus en mémoire, il parcourt la table des processus en cherchant les processus dans l'état prêt mais swappés sur disque, et choisit celui qui est resté swappé le plus longtemps. S'il y a assez de mémoire libre, le *swapper* charge le processus en mémoire, et libère

l'espace qu'il occupait sur le disque de va-et-vient. Cet algorithme est répété jusqu'à ce que l'une des deux conditions suivantes se produise :

- il ne reste plus de processus prêts sur le disque de va-et-vient. Dans ce cas, le *swapper* s'endort jusqu'à ce qu'un processus swappé soit réveillé ou que le noyau swappe sur disque un processus prêt (voir plus haut) ;
- le *swapper* trouve un processus prêt, mais il n'existe pas assez de mémoire libre pour le contenir. Dans ce cas, le *swapper* essaie d'abord de migrer sur disque un autre processus et, en cas de succès, relance son algorithme de migration, en cherchant à nouveau un processus prêt à charger en mémoire.

Si le *swapper* doit migrer un processus vers le disque, il le choisit parmi les processus en mémoire autres que les zombies (qui ne consomment pas de place mémoire) et les processus actuellement verrouillés sur des opérations mémoire. Les processus endormis sont choisis prioritairement par rapport aux processus prêts et, s'il n'y a pas de processus endormis, le processus prêt à migrer est choisi en fonction de sa valeur de priorité et du temps qu'il a déjà passé en mémoire. Un processus prêt doit avoir résidé en mémoire au moins 2 secondes avant de pouvoir être migré sur le disque ; de même, un processus migré sur disque doit y avoir résidé au moins 2 secondes avant de pouvoir être rechargé en mémoire.

Chapitre 3

Mémoire

Plus que la ressource processeur, la mémoire constitue la ressource la plus critique des systèmes d'exploitation, dont le mésusage peut avoir des effets dramatiques sur les performances globales du système.

Les fonctionnalités attendues d'un gestionnaire efficace de la mémoire sont les suivantes :

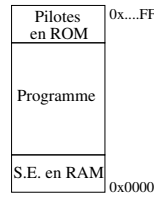
- connaître les parties libres de la mémoire physique ;
- allouer de la mémoire aux processeurs, en évitant autant que possible le gaspillage ;
- récupérer la mémoire libérée par la terminaison d'un processus ;
- offrir aux processus des services de mémoire virtuelle, de taille supérieure à celle de la mémoire physique disponible, au moyen des techniques de va-et-vient et de pagination.

3.1 Mémoire sans va-et-vient ni pagination

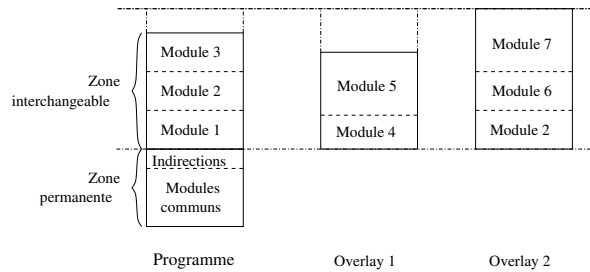
C'est le cas le plus simple (disons même *ii* rustique *ii* !), dans lequel on ne considère que la mémoire physique effectivement disponible, que l'on accède directement au moyen d'adresses physiques (par opposition aux mécanismes d'adressage virtuel que l'on verra plus loin...).

3.1.1 Monoprogrammation

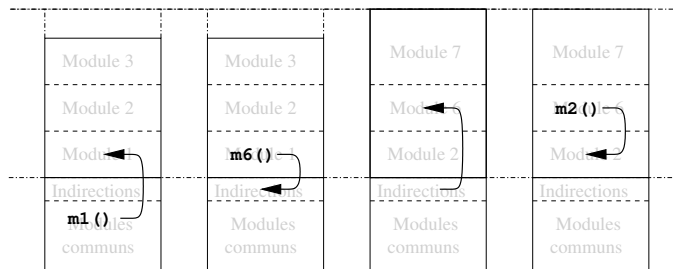
Dans ce cas, on n'a qu'un seul processus en mémoire à un instant donné, qui peut utiliser l'ensemble de la mémoire physique disponible. En pratique, une partie de l'espace d'adressage est dédiée au système d'exploitation, qui se compose des fonctions de démarrage et de gestion de bas niveau de la machine, situées en ROM (on dit *ii* romées *ii*), et le système d'exploitation proprement dit, sujet à modifications, et chargé en RAM à partir du disque lors du démarrage. À la fin de chaque programme, l'interpréteur de commandes, qui fait partie du système d'exploitation, demande à l'utilisateur le prochain programme à lancer. Cette structure est classiquement illustrée par le cas des PC sous DOS.



Lorsque la taille des programmes dépasse la taille de la mémoire physique, il est possible d'utiliser des segments de recouvrement (des *overlays* ou des surcouches) : le programme est alors composé d'une zone permanente, servant à la fois aux changements d'*overlay* et à contenir les routines les plus fréquemment utilisées, et d'un ensemble d'*overlays* interchangeables correspondant le plus souvent aux grandes fonctionnalités du programme.



Le changement d'*overlay* s'effectue en passant par la zone permanente, où le chargement du nouvel *overlay* à partir du disque est réalisé. Afin de limiter les remplacements d'*overlays*, il est possible de dupliquer un même module au sein de plusieurs *overlays*. Si les fonctions à appeler sont présentes au sein de l'*overlay*, la copie locale est utilisée, sinon l'appel est effectué dans la zone d'indirections, où une routine adaptée se charge du chargement de l'*overlay* correspondant et de l'appel de la fonction nouvellement chargée.



La résolution des adresses et l'appel aux routines locales plutôt qu'aux routines d'indirection est effectuée statiquement par l'éditeur de liens. C'est à l'utilisateur de définir la structure de ses *overlays* en fonction du comportement attendu de son programme.

Malgré le coût sans cesse décroissant et la capacité sans cesse croissante des mémoires disponibles sur le marché, ce paradigme architectural tend à être de moins en moins utilisé (en dépit des mises en garde de [?, page 213]), du fait de la demande sans cesse croissante de mémoire pour les processus, et de l'économie

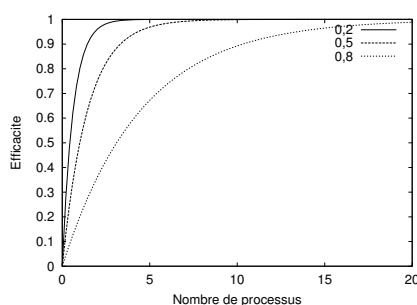
que représente l'utilisation du va-et-vient par rapport à l'achat d'une mémoire physique de capacité équivalente, peu rentabilisée.

3.1.2 Multiprogrammation

La multiprogrammation est peu utilisée sur les petits systèmes dans ce contexte, mais l'est en revanche sur les gros systèmes dédiés au calcul intensif. L'intérêt d'utiliser la multiprogrammation est double. Tout d'abord, elle facilite le développement des programmes en les fractionnant en processus indépendants du point de vue du système. Cependant, son intérêt principal à ce niveau réside dans la maximisation de la ressource processeur : si le programme à exécuter possède une partie interactive, il est aberrant de monopoliser l'ensemble de la machine pour attendre une frappe de l'utilisateur. Ceci reste vrai dans le cas de processus non interactifs mais réalisant beaucoup d'entrées-sorties.

À titre d'exemple, si chacun des processus lancés calcule pendant 20 % de son temps, il faudrait idéalement pouvoir lancer 5 processus pour avoir une chance d'utiliser pleinement le processeur (dans le cas idéal où les 5 processus ne sont jamais simultanément en phase d'entrées/sorties).

On peut modéliser simplement ce comportement de façon probabiliste. Soit p la fraction de temps passée par un processus en attente d'entrées/sorties. Si n processus sont chargés en mémoire, la probabilité que tous soient simultanément en attente est de p^n , et donc le taux d'utilisation du processeur est de $1 - p^n$.



À titre d'exemple, pour un taux d'attente de 80 %, il faut plus de 10 processus simultanément en mémoire pour tenter d'obtenir plus de 90 % d'utilisation du processeur.

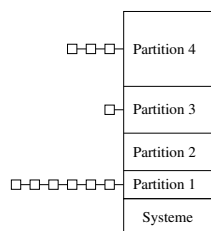
La formule ci-dessus peut également être utilisée pour déterminer la quantité de mémoire à acheter pour atteindre un taux d'efficacité donné, en fonction du type des processus. Ainsi, avec 1 Mo de mémoire, un système d'exploitation de 200 Ko, et des processus de 200 Ko passant 80 % de leur temps en attente, on ne peut loger que quatre processus en mémoire, avec un taux d'utilisation de 60 %. Si l'on ajoute un Mo de plus, on peut loger 9 processus et avoir un taux d'utilisation de 87 %. Avec un troisième Mo supplémentaire, on obtient un taux d'utilisation de 96 %, très satisfaisant.

En réalité, le modèle probabiliste présenté ci-dessus n'est qu'une approximation, car en réalité les processus ne sont pas indépendants, puisqu'un processus prêt doit attendre que le processeur se libère pour pouvoir poursuivre son exécution. Un modèle plus précis est fourni par les files d'attente.

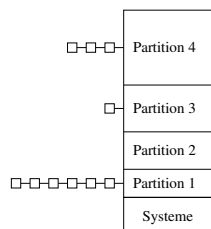
3.1.3 Multiprogrammation avec partitions fixes

La principale question lorsqu'on utilise la multiprogrammation est d'organiser la mémoire de la façon la plus efficace possible. Une méthode possible est de diviser la mémoire en partitions, habituellement de tailles inégales, ce qui est réalisé par l'opérateur lors du démarrage de la machine (le partitionnement de la machine peut être modulé automatiquement en fonction du jour et de l'heure sur la plupart des gros systèmes informatiques).

Chaque nouvelle tâche est alors placée dans la file d'attente de la plus petite partition pouvant la contenir, l'espace restant de la partition étant alors perdu.



L'utilisation de files multiples pose problème lorsque les files des grandes partitions sont vides et que les autres sont pleines. Une alternative consiste à utiliser une file unique : dès qu'une partition se libère, on y place la plus grande tâche de la file d'attente qui peut y tenir ; comme cette stratégie désavantage les tâches de petites tailles, alors qu'il faudrait théoriquement les traiter en priorité, on peut imaginer d'autres algorithmes cherchant la première tâche pouvant tenir dans la partition et pas dans les plus petites, en relâchant cette dernière contrainte si aucune tâche n'est trouvée lors de la première passe.



3.1.4 Translation de code et protection

La multiprogrammation soulève deux problèmes majeurs : la translation d'adresse et la protection contre les tentatives d'accès de la part des autres processus.

Le problème de la translation d'adresse apparaît lorsqu'on ne connaît pas l'adresse de début du programme lors de la phase d'édition de liens. Les adresses générées par le compilateur sont donc fausses, et le programme ne peut être directement exécuté sur la machine.

Une solution possible consiste à conserver, dans une table annexe au programme et générée par l'éditeur de liens, les positions de tous les mots mémoire qui sont des adresses. Lors du chargement du programme dans la partition que le système lui a attribuée, le code binaire du programme est parcouru et l'adresse

de base de la partition est ajoutée à toutes les adresses déclarées. Cette technique fut en particulier utilisée par IBM sur les machines de sa gamme 360.

Cependant, traduire les adresses des programmes lors du chargement de ceux-ci ne résoud pas le problème de la protection. Une solution possible, également adoptée par IBM sur les machines de la gamme 360, fut de diviser la mémoire en blocs de 2 Ko et d'affecter une clé à 4 bits à chacun. Le mot d'état du processeur (PSW, pour *Program Status Word*) contenait lui aussi un champ clé de 4 bits, différent pour chaque processus (seuls 16 processus pouvaient donc s'exécuter simultanément en mémoire). Ainsi, lors d'un accès mémoire, le système interceptait les tentatives d'accès à la mémoire dont les codes de clé différaient du code contenu dans le PSW. Comme les codes des blocs et du PSW ne pouvaient être modifiés que par le système d'exploitation en mode privilégié, ce mécanisme interdisait toute interaction entre processus différents.

Une autre solution, plus générique, aux problèmes de la translation d'adresses et de la protection est de doter le processeur de deux registres spéciaux accessibles uniquement en mode privilégié, appelés registres de base et de limite. Quand on charge un processus en mémoire, le registre de base est initialisé avec la limite inférieure de la partition qui lui est attribuée, et le registre de limite avec sa limite supérieure. La valeur du registre de base est alors implicitement ajoutée à l'exécution à toute adresse générée par le processeur, et comparée avec la valeur du registre de limite, les adresses inférieures à la base et supérieures à la limites déclenchant automatiquement la levée d'une exception.

Le grand avantage de cette solution est que les programmes peuvent être déplacés en mémoire de façon totalement transparente, seul les deux registres de base et de limite devant être modifiés pour tenir compte de cette modification.

3.2 Le va-et-vient

L'organisation de la mémoire en partitions fixes est une méthode simple et efficace pour le traitement par lots : il est inutile de payer le surcoût induit par une plus grande complexité tant qu'il y a suffisamment de tâches pour occuper constamment le processeur.

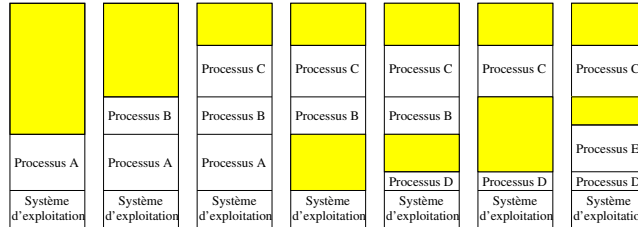
En revanche, dans les systèmes à temps partagé, la mémoire physique ne peut contenir tous les processus des utilisateurs, et il est donc nécessaire de placer sur le disque certains de ces processus, qu'il faudra donc ramener régulièrement en mémoire centrale afin de les exécuter. Le mouvement des processus entre la mémoire centrale et le disque est appelé *va-et-vient* (*swapping*).

3.2.1 Multiprogrammation avec partitions variables

Utiliser le mécanisme de va-et-vient avec une gestion de la mémoire par partitions de taille fixe conduit à un gaspillage de celle-ci, car la taille des programmes peut être bien plus petite que celle des partitions qui les hébergent, ce qui rend nécessaire la mise en œuvre d'un mécanisme de gestion de la mémoire avec des partitions de taille variable.

Avec un mécanisme de partitions à taille variable, le nombre, la position, et la taille des partitions varient au cours du temps, en fonction des processus

présents à un instant donné en mémoire centrale.

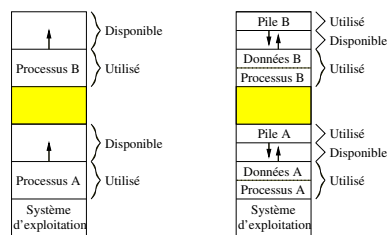


On améliore ainsi grandement l'utilisation de la mémoire, en rendant cependant les politiques et mécanismes d'allocation et de libération plus compliqués.

Lorsque la mémoire devient trop fragmentée, il est possible d'effectuer un compactage de celle-ci, en déplaçant tous les processus vers le bas de la mémoire. Cette solution est coûteuse, et était réalisée sur les gros systèmes par des circuits spécialisés, qui assuraient des débits de transfert suffisants (40 Mo/s sur le Cyber CDC). Le mécanisme de pagination, dont nous parlerons plus loin, rend maintenant cette technique inutile.

Un autre problème important concerne la taille des partitions attribuées à chaque processus, car celle-ci tend à augmenter avec le temps.

Une solution possible consiste à allouer à chaque processus un espace légèrement plus grand que sa taille actuelle, tant au chargement que lors des migrations à partir du disque ; cependant, seule la mémoire effectivement utilisée sera recopiée sur disque lors des migrations. Lorsque les programmes disposent de deux segments dont la taille peut grandir, comme la pile et le tas, on place ceux-ci tête-bêche, et l'espace libre entre les deux peut ainsi être utilisé indifféremment par l'un ou l'autre en fonction de ses besoins.



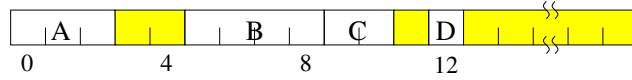
3.2.2 Gestion de la mémoire

Dans tous les cas, il faut disposer d'un mécanisme spécifique pour mémoriser les zones mémoire libres et occupées, minimiser l'espace perdu lors d'une allocation, et réduire autant que possible la fragmentation. Typiquement, ces mécanismes peuvent être de trois sortes.

Gestion par tables de bits

Avec cette technique, la mémoire est subdivisée en unités d'allocation, dont la taille peut varier de quelques mots à quelques kilo-octets. Chacune de ces unités est représentée par un bit dans une table d'allocation possédée par le

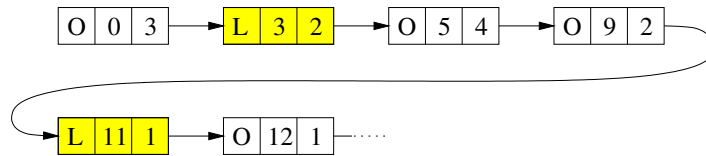
système, celle-ci permettant alors de mémoriser l'occupation de la mémoire.



Cette approche présente en fait deux inconvénients. Le premier est le choix de la taille de l'unité d'allocation. Une taille trop petite imposera une table plus grande, et donc des temps d'opérations plus grands, alors qu'une taille trop grande causera des pertes de place plus importantes dans la dernière unité d'allocation, partiellement remplie. Le deuxième est la recherche de k bits libres dans la table, qui nécessite des manipulations coûteuses sur des masques de bits. Les tables de bits sont donc rarement utilisées.

Gestion par listes chaînées

Cette technique consiste à gérer une liste chaînée des segments libres et occupés, dans laquelle chaque segment est représenté par un bit d'état, son adresse de début, et éventuellement sa taille, et qui est triée par adresses de début croissantes.



Afin de conserver à la liste la plus courte taille possible, et de limiter la fragmentation, un compactage est effectué à la libération de chaque segment : le nouveau bloc libre est fusionné avec son prédécesseur et/ou son successeur si ceux-ci sont également libres. Il ne peut donc y avoir plus de deux éléments consécutifs décrivant un segment libre. Pour des raisons d'efficacité, la liste est doublement chaînée.



Plusieurs algorithmes sont envisageables pour réaliser l'allocation d'un segment :

- *First fit* (le premier qui correspond) : la liste est parcourue jusqu'à ce qu'on trouve un segment libre de taille suffisante pour contenir l'espace mémoire demandé. S'il reste de l'espace libre dans le segment, la cellule correspondante est scindée pour conserver la trace de cet espace ;
- *Best fit* (le mieux qui correspond) : la liste est entièrement parcourue, et on choisit finalement le plus petit segment dont la taille est supérieure à celle de la mémoire demandée. On évite ainsi de fractionner une grande zone dont on pourrait avoir besoin ultérieurement. En fait, des expériences ont montré que cet algorithme fait perdre plus de place que le précédent, car il tend à remplir la mémoire avec des petites zones libres

inutilisables. En moyenne, l'algorithme *First fit* donne des zones plus grandes ;

- *Worst fit* (plus grand résidu) : cet algorithme est similaire au précédent, à la différence qu'on cherche la plus grande zone possible, qui donnera le résidu le plus utilisable possible. Cette stratégie ne donne pas de bons résultats car elle détruit en premier les segments de grande taille, qui ne seront plus utilisables ;
- *Quick fit* : pour accélérer les recherches, on peut scinder en deux les listes de segments libres et occupés, et scinder la liste des segments libres afin de disposer de listes séparées pour les tailles les plus courantes. On simplifie ainsi la recherche, mais on complique la fusion des blocs libérés.

Gestion par subdivisions

La gestion des listes chaînées en fonction de leur taille accélère l'allocation, mais rend la libération plus lente car il faut rechercher les voisins du segment libéré. L'allocation par subdivision (*buddy system*, ou allocation des copains) s'appuie sur la représentation binaire des adresses pour simplifier ces processus.

Le gestionnaire de mémoire ne manipule que des blocs dont la taille est égale à une puissance de deux, et les chaîne au moyen d'un vecteur de listes de telle sorte que la liste d'indice k contienne les blocs de taille 2^k .

Initialement, toutes les listes sont vides, sauf celle dont la taille de blocs est égale à la taille de la mémoire, qui contient une unique entrée pointant sur l'intégralité de la mémoire libre.

Lorsqu'on doit allouer une zone mémoire, on recherche un bloc libre dans la liste correspondant à la plus petite puissance de deux supérieure ou égale à la taille demandée. Si la liste est vide, on sélectionne un bloc dans la liste de puissance immédiatement supérieure, et on le divise en deux moitiés (deux blocs copains) dont l'une est affectée à la zone mémoire allouée, et l'autre est chaînée dans la liste de taille correspondante. Si la liste de puissance immédiatement supérieure est elle-même vide, on remonte le tableau de listes jusqu'à trouver un bloc libre, que l'on subdivise récursivement autant de fois que nécessaire pour obtenir un bloc de taille voulue.

Lorsqu'on libère un bloc et que son copain est lui-même déjà libre, les deux copains sont fusionnés pour redonner un unique bloc de taille supérieure, et ce processus se poursuit récursivement jusqu'à ce que plus aucune fusion ne soit possible. Dans ce cas, le bloc résultat est placé dans la liste correspondante.

État initial	1024			
Alloue A=70	A	128	256	512
Alloue B=35	A	B	64	256
Alloue C=200	A	B	64	C
Libère A	128	B	64	C
Alloue D=60	128	B	D	C
Libère B	128	64	D	C
Libère D	256		C	512
Libère C	1024			

Cet algorithme est rapide: lorsqu'un bloc de taille 2^k est libéré, seule la liste des blocs de même taille doit être explorée pour rechercher si son copain ne s'y trouve pas déjà. En revanche, il est extrêmement inefficace, puisqu'en moyenne le quart de la mémoire est gaspillé. On parle alors de problème de fragmentation interne, car l'espace perdu fait partie des segments alloués, par opposition à la fragmentation externe, qui se produit avec la gestion de la mémoire par listes chaînées (phénomène du damier, ou *checkboarding* ;).

Allocation de l'espace de va-et-vient

Les algorithmes présentés ci-dessus mémorisent l'occupation de la mémoire centrale pour que le système puisse trouver de la place lorsqu'un processus y est ramené. Ils peuvent également être utilisés par le système de va-et-vient pour allouer la place sur le disque. La seule différence avec une allocation mémoire classique est que l'espace de va-et-vient est exprimé comme un multiple de la taille des blocs disque.

Certains systèmes allouent la place sur disque à la création de chaque processus, de sorte qu'un processus donné soit toujours swappé au même endroit, alors que d'autres attribuent et libèrent l'espace de swap lors de chaque va-et-vient.

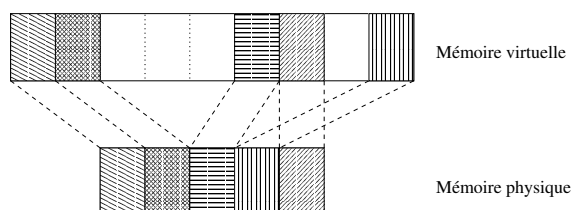
3.3 La mémoire virtuelle

Dans le cas où les programmes à exécuter sont trop importants pour tenir entièrement en mémoire, une solution possible consiste à utiliser des overlays (voir page 46). Cependant, cette solution est lourde, tant pour le système qui doit migrer l'intégralité de l'overlay lorsqu'une unique fonction externe doit être appelée, que pour le programmeur qui doit penser et optimiser à la main le découpage en overlays de son application.

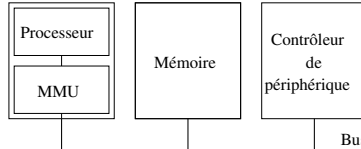
Une autre solution possible est d'utiliser un mécanisme de mémoire virtuelle. Dans ce cas, la taille du programme et de ses données peut dépasser la taille de la mémoire physique, et c'est au système d'exploitation de ne conserver en mémoire physique que les parties utiles à l'avancement immédiat du processus.

3.3.1 Pagination

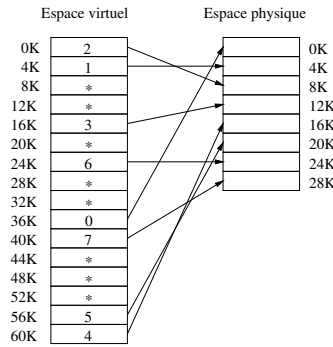
Le principal problème de la mémoire virtuelle est que les parties immédiatement utiles à l'avancement d'un processus sont presque toujours disjointes en mémoire centrale (le code et la pile, par exemple). Un mécanisme efficace de mémoire virtuelle doit donc pouvoir effectuer la traduction des adresses virtuelles manipulées par le processus en adresses lécales dans la mémoire physique, en gérant les discontinuités.



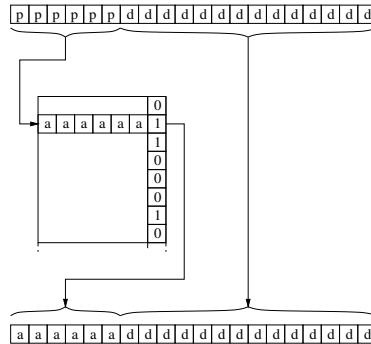
Pour cela, on a recours à la pagination. Les adresses virtuelles utilisées par le processeur pour accéder à l'ensemble de l'espace d'adressage virtuel sont traduites en adresses physiques par un circuit spécial situé entre le processeur et le bus mémoire, et appelé MMU (pour *ij Memory Management Unit ij*). Comme la MMU est utilisée pour chaque accès au bus, il est crucial que le mécanisme de traduction utilisé soit simple et efficace.



L'espace d'adressage virtuel est divisé en pages, dont la taille est une puissance de deux, habituellement comprise entre 512 et 4096 octets. Les unités correspondantes dans la mémoire physique sont les pages mémoire physiques (*ij page frames ij*). La MMU dispose d'une table d'indirection des pages pour convertir les adresses virtuelles en adresses physiques, qui donne pour chaque page virtuelle le numéro de page physique sur laquelle elle est placée (*ij mapping ij*). Les pages virtuelles non actuellement mappées en mémoire centrale sont identifiées par un bit de présence.



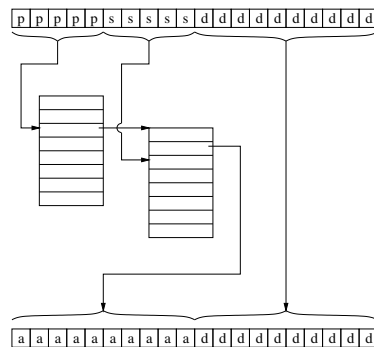
L'indexation de la table des pages s'effectue en décomposant l'adresse virtuelle en un numéro de page (*ij page index ij*) et un déplacement (*ij offset ij*) dans la page. L'index de page sert à adresser la table des pages, qui fournit le numéro de page physique remplaçant l'index de page dans l'adresse physique.



Ce mécanisme est très rapide, mais la table des pages utilisée, constituée principalement de références vides, consomme inutilement beaucoup trop de mémoire.

Ainsi, avec un espace d'adressage de 2^{32} octets, et des pages de 2^{12} octets, c'est-à-dire de 4 ko, on doit avoir 2^{20} entrées, c'est-à-dire que la table doit faire 1 méga-mots.

Pour réduire la mémoire utilisée par la table des pages, et éviter de coder des espaces vides, on lui applique son propre principe, en créant une table à deux niveaux d'indirections : la partie haute de l'adresse permet de déterminer la table des pages devant être utilisée, qui est elle-même accédée au moyen de la partie médiane de l'adresse afin de déterminer le numéro de la page physique utilisée.



Ainsi, avec un premier niveau indexé sur 10 bits, et un deuxième niveau également sur 10 bits, on réduit considérablement la mémoire occupée par la table des pages pour de petits programmes, tout en ne générant qu'un surcoût marginal pour les gros.

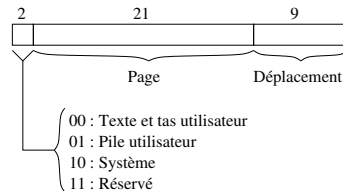
Les drapeaux d'état d'une page comprennent habituellement les informations suivantes :

- antémémoire : ce bit sert à indiquer si les données de la page peuvent être conservées dans les caches du ou des processeurs. Cette information est critique pour pouvoir piloter des périphériques mappés en mémoire centrale ;
- référencée : ce bit indique que la page existe ;
- modifiée : ce bit indique que la page a été modifiée par rapport à sa version sur disque ;
- protections : ces bits définissent le niveau de protection de la page, en particulier si elle est partagée et quels sont ses droits d'accès ;
- présence : ce bit est positionné si la page est présente en mémoire physique ;
- numéro : ce champ donne le numéro de page physique de la page, si elle est marquée comme présente.

La liste ci-dessous présente quelques architectures de pagination mémoire pour des processeurs célèbres.

- DEC PDP-11 : cette machine, basée sur un processeur 16 bits, disposait également d'adresses physiques sur 16 bits, et accédait à une mémoire virtuelle d'au plus 4 Mo par pages de 8 Ko. La table des pages, à un niveau, ne contenait que 8 entrées ;

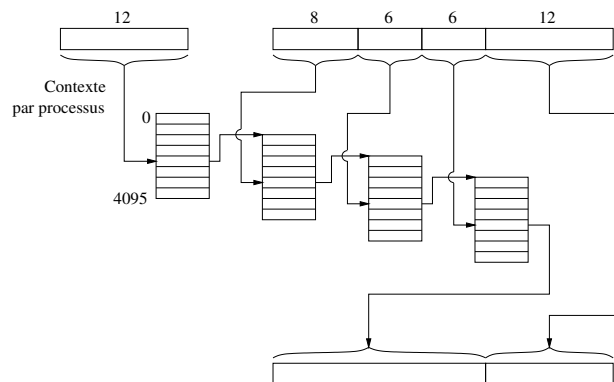
- DEC VAX : cette machine, basée sur un processeur 32 bits, accédait à une mémoire vive d'au plus 4 Go par pages de 512 octets. Les adresses virtuelles étaient structurées selon le schéma suivant :



L'espace entre 0 et 1 Go contient le programme et les données de l'utilisateur, l'espace entre 1 et 2 Go sa pile, et l'espace entre 2 et 3 Go le système, qui est partagé entre tous les processus.

Comme les éléments de la table des pages étaient codés sur 4 octets, la table des pages atteignait 8 Mo par processus. Pour ne pas l'allouer en entier, on simulait un deuxième niveau en plaçant la table non pas dans une mémoire spécifique à la MMU, mais dans l'espace virtuel dédié au système d'exploitation, où elle pouvait elle-même faire l'objet de pagination ;

- Sparc : la MMU est organisée avec une table à trois niveaux, dont la première table est indexée par une valeur de contexte dépendant du processus ;



- Motorola 68030 : ce processeur permet de définir jusqu'à quatre niveaux de tables de pages. Le nombre de tables ainsi que le nombre de bits par niveau sont programmables au moyen d'un registre spécifique appelé TCR (*Table Configuration Register*).

L'existence de plusieurs niveaux de tables, si elle permet une économie substantielle en terme de mémoire, pose un problème majeur de performance, puisqu'un accès mémoire à une page de données peut nécessiter jusqu'à quatre accès de tables, et donc peut quintupler le coût de l'accès mémoire.

Pour éviter ces accès multiples, les MMU modernes sont dotées d'une mémoire cache associative, de petite taille, qui établit une correspondance directe entre les numéros de pages virtuelles les plus récemment utilisées et les numéros de pages physiques correspondantes. Ainsi, si le numéro de la page virtuelle demandée est présent dans la mémoire cache, le numéro de page physique sera

immédiatement fourni. Sinon, les tables devront être parcourues, et le résultat du parcours sera stocké dans la mémoire associative, à la place de l'entrée la moins récemment utilisée, afin d'accélérer un accès futur à la même page (en vertu des principes de localité). Cette mémoire cache associative incorporée à la MMU est appelée TLB, pour *Translation Lookaside Buffer* [1].

3.3.2 Algorithmes de remplacement de pages

Lorsque le processeur demande une page qui n'est pas présente en mémoire, la MMU lève une exception, qui se traduit par l'arrivée d'une interruption au niveau du processeur. Ensuite, le système passe par les étapes suivantes :

1. le déroutement provoque la sauvegarde du pointeur d'instruction, et l'appel à la routine d'interruption associée ;
2. la routine d'interruption en assembleur sauvegarde les registres et appelle la routine de traitement du système d'exploitation ;
3. le système analyse l'adresse ayant provoqué le défaut de page et détermine la page manquante. Dans le cas où l'accès à la page manquante provoquerait une violation de protection, un signal est envoyé au processus. Sinon, s'il n'existe pas de page mémoire libre, l'algorithme de remplacement sélectionne une page. Si celle-ci a été modifiée, le système démarre sa copie sur disque et endort le processus courant en attente de la fin de l'écriture ;
4. s'il existait une page mémoire libre, ou lorsque le processus a été réveillé par la terminaison de l'écriture de la page à remplacer, le système démarre le chargement de la nouvelle page et endort le processus courant en attente de la fin de la lecture ;
5. lorsque le processus est réveillé par la fin de la lecture, le système met à jour la table des pages du processus ;
6. la routine d'interruption remet l'instruction fautive dans son état original, positionne la valeur de retour du pointeur d'instruction de façon à la réexécuter, restaure les valeurs des registres, et retourne.

Le principal problème de la pagination est le choix des pages à supprimer de la mémoire centrale pour faire place aux nouvelles pages demandées par le processeur. Il semble préférable à première vue de supprimer une des pages les moins utilisées, mais choisir une page déjà modifiée nécessite sa copie préalable sur disque, ce qui peut être coûteux. Plusieurs algorithmes ont été proposés pour augmenter l'efficacité du remplacement.

Algorithme de remplacement optimal

Cet algorithme, qui est le meilleur possible, est impossible à implémenter. Il reviendrait, chaque fois qu'une page est accédée, à la marquer avec un compteur indiquant à quel moment cette page sera réaccédée de nouveau. Ainsi, lorsqu'un remplacement de page serait nécessaire, on pourrait toujours enlever la page dont on aurait le moins besoin.

Bien évidemment, sauf dans de rares cas très particuliers, il est impossible de prévoir à l'avance les pages dont un programme pourrait avoir besoin, en fonction du jeu de données avec lequel il est lancé. On peut cependant obtenir cette

information en traçant la première exécution du programme, afin d'optimiser les accès mémoire lors des exécutions suivantes.

Cette technique n'a pas d'intérêt pratique dans un contexte d'exécution réel. En revanche, elle permet d'obtenir une mesure de performance idéale, qui sert de référence pour juger de l'efficacité d'autres algorithmes.

Algorithme du peu fréquemment utilisé

On associe à chaque page deux bits d'information : le bit R, ou bit de référence, est positionné chaque fois que la page est accédée, et le bit M est positionné chaque fois que la page est modifiée par le processus.

Pour des raisons d'efficacité, il est essentiel que ces bits soient mis à jour directement par le matériel de la MMU. Si celle-ci ne le permet pas, on peut cependant émuler ce comportement de façon logicielle. Pour ce faire, quand une nouvelle page est demandée pour la première fois, la routine de défaut de page positionne elle-même le bit R, et charge la page avec des droits en lecture seule. Ainsi, lors du premier accès de la page en écriture, la MMU générera une deuxième interruption, qui servira à positionner le bit M, et à autoriser les accès suivants en écriture.

De façon régulière, par exemple lors des interruptions d'horloge, le système remet à zéro tous les bits R des pages présentes en mémoire, pour différencier des autres les pages qui seront accédées avant l'interruption suivante (dans le cas de l'émulation logicielle, tous les droits d'accès aux pages sont supprimés). Chaque page peut donc être caractérisée par les valeurs de ses bits R et M :

- 0 : $\bar{R}\bar{M}$: non accédée et non modifiée ;
- 1 : $\bar{R}M$: non accédée mais modifiée, ce qui se produit lorsque le bit R d'une page déjà modifiée est remis à zéro par la routine d'interruption ;
- 2 : $R\bar{M}$: accédée mais non modifiée ;
- 3 : RM : accédée et modifiée.

Lorsqu'une page doit être remplacée, le système parcourt la table des pages et choisit une des pages de plus petite valeur binaire. Cet algorithme suppose implicitement qu'il vaut mieux retirer une page modifiée mais non accédée récemment qu'une page non modifiée mais plus fréquemment utilisée, afin de mieux respecter les principes de localité. Cet algorithme NRU (pour *Not Recently Used*) est simple à implémenter, et fournit des performances suffisantes dans la plupart des cas.

Algorithmes de file

L'idée de base de ces algorithmes est de maintenir une file des pages chargées en mémoire, afin de supprimer à chaque fois la page ayant résidé le plus longtemps en mémoire. Comme les pages les plus anciennes peuvent également être celles qui sont les plus utilisées, il faut néanmoins modifier l'algorithme afin de ne pas considérer les pages récemment utilisées.

Pour cela, on peut utiliser le bit R et M de l'algorithme précédent, et remplacer la page la plus ancienne et de plus petite valeur binaire.

Une variante de cet algorithme, appelée *second chance* algorithme, consiste à tester le bit R de la page la plus ancienne. S'il est à zéro, la page est remplacée, sinon il est remis à zéro et la page est remise en queue de file. Si toutes les pages ont été référencées, l'algorithme de la seconde chance est équivalent

En général, il suffit de peu de bits pour donner de très bons résultats. Par exemple, avec un intervalle de rafraîchissement de 20 ms, huit bits suffisent à mesurer des vieillissements allant jusqu'à 160 ms, ce qui est tout à fait acceptable au regard des vitesses des processeurs actuels.

3.3.3 Modélisation des systèmes paginés

Afin de mesurer la performance des différents algorithmes et systèmes de pagination, et de déterminer leurs paramètres critiques à optimiser, plusieurs outils théorique ont été définis.

Pour les employer, il est nécessaire de représenter de façon utilisable les traces d'exécution des programmes, pour en extraire des paramètres pertinents. La base de cette représentation est la chaîne des références, constituée de la liste des numéros des pages accédées dans le temps, dont les valeurs sont comprises entre 0 et $(n - 1)$ pour un programme utilisant n pages différentes. Ainsi, pour un programme utilisant 8 pages de mémoire virtuelle, on peut donc avoir :

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 4 1

On peut en extraire, pour un algorithme donné, un tableau de l'état de la mémoire représentant à tout instant, sur une machine à m pages physiques, l'historique des m pages présentes en mémoire et des $(m - n)$ pages référencées puis retirées. Pour la chaîne de références ci-dessus et un algorithme LRU s'exécutant sur une machine à 4 pages physiques, on a donc :

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	4	1
0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	4	1	
	0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	2	3	4	
		0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	1	7	2	3	
			0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	3	1	7	2	
				0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	5	5	1	7	
					0	2	2	1	1	1	1	1	1	1	6	6	6	6	4	4	5	5		
						0	0	2	2	2	2	2	2	2	2	2	2	2	2	6	6	6	6	
								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Modèle des chaînes de distance

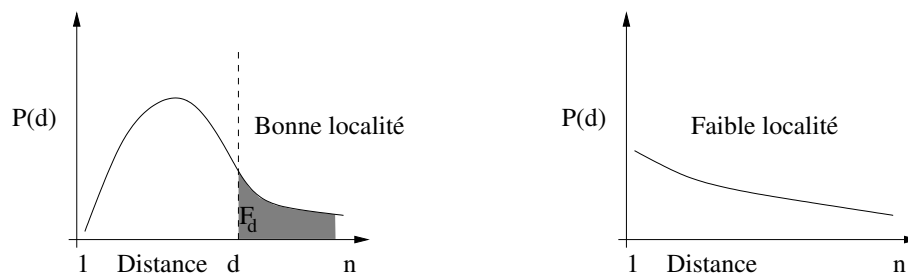
Les algorithmes à pile (mais pas FIFO, voir section 3.3.2) possèdent la propriété que si l'on augmente la mémoire d'une case et qu'on réexécute à nouveau le processus, toutes les pages présentes à un instant donné avant l'ajout de mémoire seront également présentes après l'ajout.

Il est donc possible pour ces algorithmes de synthétiser le tableau de l'état de la mémoire de façon plus abstraite, en calculant la distance entre chaque page demandée et le sommet de la pile ; ce paramètre dépend à la fois de la chaîne de références et de l'algorithme de pagination. On peut ainsi prévoir le nombre de défauts en fonction de la taille de la mémoire physique. Soit c_i le nombre de

demandes de pages situées à la distance i du sommet de la pile. On définit alors

$$F_k = \sum_{i=k}^{n-1} c_i + c_\infty$$

comme le nombre de défauts de page pour la chaîne donnée si l'on dispose de m pages physiques. Avec l'exemple ci-dessus, on a $c_0 = 4$, $c_1 = 2$, $c_2 = 1$, $c_3 = 4$, $c_4 = 2$, $c_5 = 2$, $c_6 = 0$, $c_\infty = 8$ (car on a $n = 8$ pages), et donc: $F = \{23, 19, 17, 16, 12, 10, 8, 8\}$ (on a bien $F_0 = (s - 1)$, où s est la taille de la chaîne de références). On peut ainsi déterminer la taille mémoire nécessaire pour réduire de façon significative le nombre de défauts de page.



Modèle des ensembles de travail

Les principes de localité montrent que, pendant une période de temps donnée, les processus ne référencent qu'un sous-ensemble réduit de leurs pages.

On appelle ω_i ensemble de travail ω_i (ω_i *working set* ω_i) l'ensemble des pages utilisées pendant un certain temps par un processus. Si l'ensemble de travail est entièrement en mémoire centrale, le processus s'exécute rapidement sans provoquer de défauts de page. En revanche, un programme qui génère de nombreux défauts de page peut provoquer un écroulement du système (ω_i *thrashing* ω_i).

Dans les systèmes d'exploitation à temps partagé, les processus sont fréquemment recopiés sur disque pour laisser à d'autres la possibilité d'utiliser le processeur. Si l'on n'utilise que le mécanisme de pagination à la demande, un processus remonté en mémoire à partir du va-et-vient générera des défauts de page jusqu'à ce que son ensemble de travail soit complètement rechargé à partir du disque, ce qui conduit à une mauvaise utilisation de la mémoire centrale. De nombreux systèmes de pagination mémorisent donc les ensembles de travail des différents processus et les chargent complètement en mémoire avant de donner la main aux processus.

La caractéristique la plus importante d'un ensemble de travail est sa taille. Si la somme des tailles des ensembles de travail des processus exécutables à un instant donné est supérieure à la taille de la mémoire physique disponible, il se produit un écroulement du système. C'est au système d'exploitation de s'assurer que les ensembles de travail des processus prêts tiennent en mémoire, si besoin en réduisant le degré de multiprogrammation, c'est-à-dire en ayant moins de processus prêts en mémoire.

Pour mettre en œuvre le modèle de l'ensemble de travail, il faut que le système d'exploitation mémorise les pages appartenant à l'ensemble de travail d'un processus donné. Ceci peut se faire simplement en utilisant les compteurs utilisés par l'algorithme de vieillissement : est considérée comme appartenant

à l'ensemble de travail toute page dont le compteur contient un 1 parmi ses b bits de poids le plus fort. La valeur du paramètre b doit être déterminée expérimentalement pour le système, mais sa valeur n'a pas besoin d'être très précise.

Allocation locale et allocation globale

Lorsqu'on cherche à remplacer la page physique la plus âgée pour remédier au défaut de page généré par un processus, on peut supprimer soit la page physique la plus ancienne de ce processus (on parle de remplacement local), soit la page physique la plus ancienne prise sur l'ensemble des processus (remplacement global).

En règle générale, les algorithmes globaux donnent de meilleurs résultats, particulièrement lorsque la taille de l'ensemble de travail peut varier au cours du temps.

Avec un algorithme local, la diminution de la taille de l'ensemble de travail conduit à un gaspillage de mémoire, et l'augmentation de taille de l'ensemble peut conduire à un écroulement même si d'autres processus disposent de pages très âgées qu'ils ne réutiliseront pas.

Avec un algorithme global, le système doit constamment décider du nombre de pages physiques qu'il doit allouer à chaque processus. Il peut pour cela surveiller la table des ensemble de travail grâce aux bits de vieillissement, mais cette approche n'élimine pas le risque d'écroulement lorsque la taille de l'ensemble de travail d'un processus augmente brutalement.

Une synthèse de ces deux approches est l'algorithme de fréquence des défauts de page (PFF, pour *Page Fault Frequency*), qui permet à la fois l'allocation globale de la mémoire et le contrôle de l'écroulement. Lorsque le taux de défauts de page générés par un processus est trop élevé, on lui attribue davantage de pages physiques, afin de réduire le taux. En revanche, si le taux de défauts de page du processus est trop bas, on lui retire des pages physiques, pour les donner à d'autres.

Si le système ne peut garder l'ensemble des fréquences de défaut de page des processus en dessous de la limite maximale, il retire un processus de la mémoire et répartit ses pages physiques entre les procesus restants et la réserve de pages libres, s'il en reste.

Chapitre 4

Fichiers

Le stockage persistant, rapide, et fiable de grandes quantités de données (et de petites !) est un critère déterminant de l'efficacité d'un système d'exploitation.

Pour ce faire, on a très vite formalisé la notion de fichier, correspondant à un objet nommé, résidant en dehors de l'espace d'adressage des processus, mais disposant d'interfaces permettant la lecture et l'écriture de données dans ce dernier. Le nom même de `ij` fichier `ll` provient de l'histoire de l'informatique, lorsque les premières machines mécanographiques étaient exclusivement dédiées au classement et à la gestion de fichiers de cartes perforées. L'espace des fichiers et son organisation interne sont appelées génériquement `ij` système de (gestion de) fichiers `ll`.

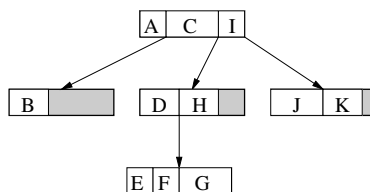
Du point de vue utilisateur, les services attendus d'un système de gestion de fichiers comprennent :

- l'indépendance vis-à-vis du type de périphérique supportant le système de fichiers ;
- la création de fichiers ;
- la suppression de fichiers, avec garde-fous et archivage de versions éventuels ;
- la gestion de protections et de droits d'accès ;
- l'ouverture de fichiers, avec gestion éventuelle des accès concurrents ;
- la fermeture de fichiers, avec libération automatique des ressources systèmes utilisées ;
- la troncature et l'extension automatique de la taille des fichiers ;
- la lecture et l'écriture de données, soit de façon séquentielle, soit par accès à une position donnée par une clé, en sérialisant les accès concurrents ;
- le changement de la position courante, permettant l'accès aléatoire ;
- le mappage et le dé-mappage des fichiers en mémoire (sur AIX et Multics, par exemple) ;
- la structuration des données en enregistrements.

4.1 Structuration des fichiers

Selon les systèmes, différentes organisations sont proposées aux utilisateurs pour organiser les données dans les fichiers. Ceux-ci peuvent être organisés comme :

- des suites d’octets : c’est l’organisation conceptuellement la plus simple. Le système de fichiers ne gère que des suites d’octets sans structure visible ; c’est leur interprétation par les différents programmes et le système (pour les fichiers considérés comme des exécutables) qui leur donne une signification. C’est l’organisation adoptée par de nombreux systèmes, comme les Unix, DOS, ... ;
- des suites d’enregistrements : les fichiers sont structurés en enregistrements de taille fixe, qui ne peuvent être lus et écrits qu’en totalité, sans possibilité d’insertion au milieu de la liste. C’était en particulier le cas de CP/M ;
- un arbre d’enregistrements de taille variable : les fichiers sont organisés en enregistrements de taille variable, indexés chacun par une clé, et groupés par blocs de façon hiérarchique. L’ajout d’un nouvel enregistrement en une position quelconque peut provoquer un éclatement d’un bloc en sous-blocs, tout comme la suppression d’un enregistrement peut provoquer la fusion de blocs peu remplis. Cette organisation, de type ii fichier indexé ii , est proposée par le système de fichiers ISAM (ii *Indexed Sequential Access Method* ii) d’IBM.



4.1.1 Types de fichiers

Dans tous les systèmes de fichiers, la plupart des informations de structure sont elles aussi considérées comme des fichiers (spéciaux), de même que certains moyens de communication inter-processus (pipes nommés ou non, sockets). Les types de fichiers les plus couramment définis sont les suivants :

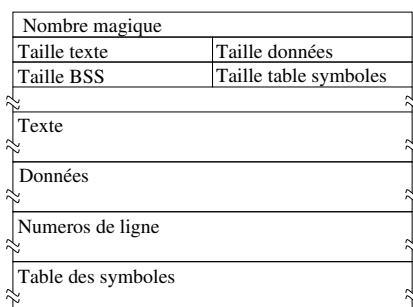
- fichiers ordinaires : ils contiennent les données des utilisateurs ;
- répertoires (ou catalogues) : structure du système de fichiers permettant d’indexer d’autres fichiers, de façon hiérarchique ;
- fichiers spéciaux de type ii caractère ii : modélisent des périphériques d’entrée/sortie travaillant caractère par caractère, comme les terminaux (claviers et écrans), les imprimantes, les pipes, les sockets ;
- fichiers spéciaux de type ii bloc ii : modélisent des périphériques d’entrée/sortie travaillant par blocs, comme les disques.

4.1.2 Fichiers ordinaires

Dans la plupart des systèmes d'exploitation, les fichiers ordinaires sont subdivisés en plusieurs types en fonction de leur nature. Ce typage peut être :

- un typage fort : dans ce cas, le nommage des fichiers fait intervenir la notion d'extension, qui est gérée partiellement par le système (par exemple, sous DOS, un fichier doit posséder l'extension `;; bin ;;`, `;; com ;;`, ou `;; exe ;;` pour pouvoir être exécuté par le système) ;
- un typage déduit : les extensions des noms de fichiers ne sont qu'indicatives, et le système détermine la nature des fichiers par inspection de leur contenu (voir en particulier la commande `;; file ;;` d'Unix) ;
- un typage polymorphe : les fichiers représentent la sérialisation d'objets persistents dans des langages orientés objet ou fonctionnels, comme en Java par exemple. De façon interne, ces fichiers contiennent la description de la classe dont ils sérialisent une instance, ce qui les rapproche du typage déduit.

La structure interne d'un fichier ordinaire dépend de son type. Elle peut être simple, comme dans le cas des fichiers texte, constitués de séquences de lignes terminées par des caractères spéciaux (`;; CR ;;`, ou `;; CR-LF ;;`), lisibles sur un terminal sans traitement spécial. Elle peut être complexe lorsque les données sont organisées selon une structure interne dépendant du type du fichier, comme par exemple pour les fichiers exécutables d'Unix (voir à ce propos le `man` de `a.out`), dont l'information peut être extraite au moyen de nombreux outils (`od`, `strings`, `nm`).



4.1.3 Catalogues

Les catalogues permettent d'organiser les fichiers au sein du système de fichiers. Le plus souvent, les catalogues sont eux-mêmes des fichiers, interprétés de façon spéciale par le système et disposant de droits spécifiques pour empêcher toute manipulation directe par les utilisateurs. Les catalogues sont organisés comme une liste d'entrées, dont chacune fait référence à un fichier unique (en revanche, un même fichier peut être référencé par plusieurs entrées différentes, comme par exemple avec le mécanisme des liens `;; hard ;;` d'Unix, créés avec la commande `ln`). Chaque entrée peut contenir les champs suivants :

- le nom de l'entrée ;

- le type de l'entrée : fichier ordinaire, catalogue, lien symbolique, pipe nommé, socket, fichiers spéciaux de type `ij` caractère `ll` ou `ij` bloc `ll`, etc. ;
- sa taille actuelle ;
- ses informations d'accès, telles que l'identificateur du propriétaire du fichier, les droits d'accès, les dates d'accès et de modification, etc. ;
- l'organisation physique du fichier sur le disque.

La possibilité de référencer des catalogues comme entrées d'autres catalogues permet de définir une organisation hiérarchique arborescente, c'est-à-dire représentable par un graphe orienté sans cycles, mis à part les répertoires `ij . ll` et `ij .. ll` (pour garantir cette propriété, la création de liens `ij` hard `ll` entre répertoires est réservée, sous Unix, au super-utilisateur).

Le plus souvent, on utilise, à la place des liens `ij` hard `ll` qui correspondent à des liens `ij` en dur `ll`, des liens `ij` symboliques `ll`, qui sont interprétés par le système au moment de leur traversée. Cette interprétation des liens symboliques au moment de leur traversée possède comme avantage que l'entrée pointée par le lien symbolique peut changer de contenu sans que le lien symbolique ait besoin d'être régénéré ; l'inconvénient majeur de cette approche est qu'un lien symbolique peut pointer sur un fichier inexistant. Dans certains systèmes (tels celui des machines Apollo), la destination d'un lien symbolique pouvait même contenir des expressions faisant intervenir des variables d'environnement.

Les opérations réalisables sur les catalogues comprennent :

- la création d'une entrée d'un type donné ;
- la suppression d'une entrée existante (qui se traduit par la suppression du fichier référencé si l'entrée supprimée était la dernière référence sur le fichier) ;
- le renommage sur place d'une entrée. Il peut être assimilé à la création d'une nouvelle référence sur le fichier, suivie de la suppression de l'ancienne ; c'est cette philosophie qui a été retenue par les concepteurs d'Unix, pour qui le déplacement et le renommage d'une référence sont réalisés par la même commande `mv`, à la différence de ce qui existe sous DOS, par exemple ;
- l'accès séquentiel aux entrées du catalogue, au moyen d'un itérateur du type `opendir/readdir/closedir`.

4.2 Structure physique des systèmes de fichiers

4.2.1 Stockage des fichiers

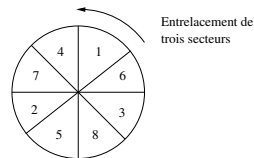
Une grande part de la performance des systèmes de fichiers provient de la rapidité d'accès aux fichiers, qui découle elle-même de leur structuration physique sur le disque.

Allocation contiguë

L'allocation contiguë consiste à allouer les blocs logiques constituant les fichiers de façon contiguë sur le disque. L'avantage principal de cette approche

est la performance des accès (cette approche est implémentée dans le `journal server` `ixi`, justement nommé, du système Amoeba). En revanche, la contrainte de contiguïté pénalise fortement l'allocation de nouveaux fichiers, du fait de la fragmentation externe qu'elle engendre; des outils de compactage doivent être régulièrement utilisés afin de défragmenter l'espace libre. La plupart du temps, ces systèmes de fichiers ne permettent pas l'extension automatique des fichiers: pour étendre un fichier, il est nécessaire de réallouer un nouveau fichier de plus grande taille estimée, dans lequel est recopié le contenu de l'ancien fichier.

Il faut cependant noter que la contiguïté logique des blocs n'implique pas nécessairement la contiguïté physique de ces mêmes blocs sur la surface du disque. En effet, pour optimiser le débit et minimiser la latence entre deux accès à deux blocs consécutifs, la plupart des disques mettent en œuvre un entrelacement (`interleaving` `ixi`) des secteurs, de façon à ce que le délai entre deux demandes d'accès à deux blocs consécutifs corresponde au temps de passage de la tête de lecture d'un bloc logique à l'autre, par rotation du disque.



Organisation par listes chaînées

Avec une organisation par listes chaînées, seul le numéro logique du premier bloc est stocké dans le catalogue, une zone spécifique de chaque bloc contenant le numéro logique du prochain bloc du chaînage.

Afin d'accélérer les accès non séquentiels, les chaînages peuvent être séparés des blocs eux-mêmes. On dispose alors d'une table possédant une entrée pour chaque bloc, indiquant le numéro du bloc suivant dans le chaînage du fichier. Ce système est implémenté par exemple par MS-DOS, qui maintient plusieurs copies sur disque de cette table, appelée FAT (pour `File Allocation Table` `ixi`), pour des raisons de fiabilité, ainsi qu'une copie en mémoire centrale pour optimiser les traitements.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
—	—	EOF	13	2	9	8	—	4	12	3	—	EOF	EOF	—

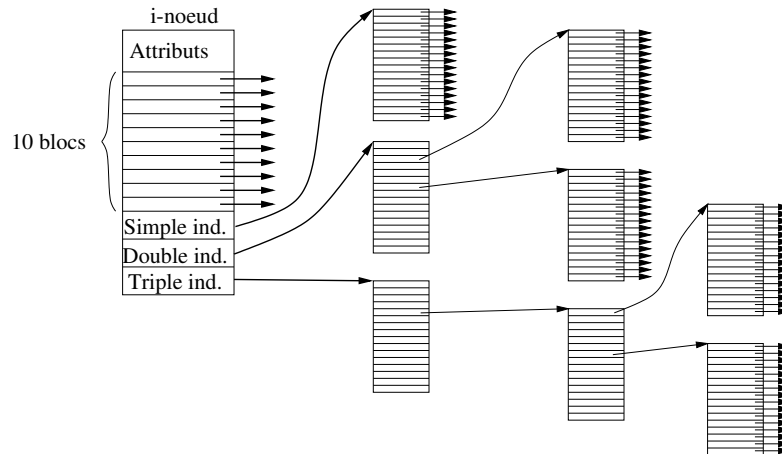
A	6
C	10
B	5

A : 6 → 8 → 4 → 2
 B : 5 → 9 → 12
 C : 10 → 3 → 13

Les avantages de cette implémentation sont l'accélération des accès aux listes chaînées, et la séparation physique entre les données et les pointeurs, qui facilite le traitement et la sauvegarde des informations de chaînage. En revanche, elle implique le maintien de tables de grande taille, même si peu de blocs sont alloués. À titre d'exemple, un disque de 4 Go découpé en blocs de 4 ko nécessite 1 million d'indices sur 32 bits (dont 20 utilisés).

Organisation par I-nœuds

L'idée de l'organisation par i-nœuds est d'associer à chaque fichier la liste de ses blocs, sous la forme d'une table de capacité variable. Afin de ne pas causer de surcoût aux petits fichiers, tout en permettant la gestion efficace des gros, cette table possède plusieurs niveaux d'indirection, selon le même principe que les tables de pages à plusieurs niveaux des MMU.



Le nombre de blocs de chaînage alloués dépend de la taille du fichier, comptée en nombre de blocs :

- moins de 10 blocs : dans l'i-nœud lui-même ;
- moins de 266 blocs : en utilisant un unique bloc d'indirection sur disque ($10 + 256$) ;
- moins de 65802 blocs : en utilisant deux niveaux d'indirection sur disque ($10 + 256 + 256^2$) ;
- jusqu'à 16 millions de blocs : en utilisant les trois niveaux d'indirection sur disque ($10 + 256 + 256^2 + 256^3$).

Ce schéma arborescent est extrêmement puissant, puisqu'au plus trois accès disques sont nécessaires pour obtenir l'indice de bloc correspondant à une position quelconque dans un fichier ; il permet également l'implémentation de fichiers de grande taille à trous, dans lesquels seuls les blocs écrits au moins une fois sont effectivement alloués. Pour optimiser les accès, les blocs d'indirections peuvent être chargés dans le buffer cache, comme tout bloc du disque.

4.2.2 Stockage des catalogues

Lorsque le système d'exploitation doit accéder à un fichier (ordinaire, catalogue, ou autre), il utilise le chemin d'accès qui lui est fourni et parcourt l'arborescence des répertoires, en recherchant à chaque fois l'entrée correspondante du chemin dans le catalogue courant. L'information portée par chaque entrée dépend de la structuration physique du système de fichiers.

Sous CP/M, on disposait d'un unique catalogue, dont chaque entrée possédait une liste de 16 numéros de blocs. Si la taille du fichier dépassait ces 16 blocs, un champ de l'entrée permettait de pointer sur une autre entrée utilisée comme stockage des 16 numéros de blocs suivants, et ainsi de suite. Il faut remarquer

à ce propos que CP/M ne représentait pas la taille d'un fichier en octets, mais seulement en blocs.

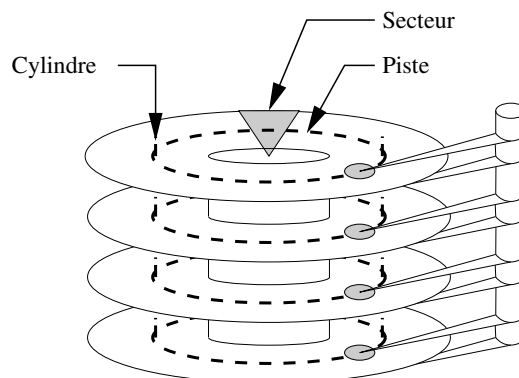
Sous DOS, chaque entrée contient le numéro du premier bloc du fichier, qui sert de point de départ au chaînage des blocs dans la FAT.

Sous Unix, l'entrée contient le numéro de l'i-nœud correspondant au fichier, ce qui permet d'implémenter simplement les liens « hard » au sein du même système de fichiers (ils sont en revanche impossibles entre deux systèmes de fichiers différents). Les liens symboliques, eux, nécessitent un i-nœud et un bloc pour stocker le chemin destination du lien.

4.2.3 Organisation de l'espace disque

Structuration physique

La plupart des disques durs sont construits comme un empilement de disques magnétiques rigides dont la surface est balayée par un ou plusieurs jeux de bras portant des têtes de lecture.



Il en a découlé une terminologie adéquate pour le repérage des blocs sur la surface et l'optimisation des accès :

- piste : zone couverte par une tête de lecture en un tour de disque lorsque le bras reste dans une position donnée ;
- cylindre : zone couverte sur tous les disques par l'ensemble des têtes de lecture en un tour de disque lorsque le bras reste dans une position donnée ;
- secteur : portion de disque représentant une fraction de la surface angulaire totale.

Taille des blocs

Un des paramètres critiques des systèmes de fichiers est la taille des blocs (aussi appelés « unités d'allocation ») utilisés. Une taille trop grande par rapport à la taille moyenne des fichiers génèrera un gaspillage d'espace énorme, alors qu'une taille trop petite ralentira les accès du fait du plus grand nombre de blocs à accéder (déplacement du bras et attente du passage du bon secteur) pour obtenir la même quantité d'informations.

Mémorisation des blocs libres

Pour mémoriser les blocs libres, on a classiquement recours à deux méthodes.

La première consiste à maintenir une liste des blocs libres. Une implémentation possible serait que chaque début de bloc libre contienne le numéro du bloc libre suivant, mais alors on devrait parcourir l'ensemble de la surface du disque pour allouer les blocs les uns après les autres, ce qui pénaliserait grandement la *buffer cache*. L'implémentation classiquement retenue consiste donc à disposer d'un chaînage de blocs libres dont chacun contient le plus possible d'adresses de blocs libres.

Par exemple, pour un disque de 2 Go décomposé en blocs de 2 ko adressés sur 24 bits (dont 20 effectivement utilisés), on peut stocker 680 adresses de blocs libres par bloc en plus du chaînage et du nombre d'emplacements utilisés, et donc le stockage de l'ensemble des blocs libres nécessite au plus 1543 blocs, dont un seul est effectivement utilisé à la fois, avec une grande localité des accès.

La deuxième consiste à maintenir une table de bits représentant l'occupation des blocs du disque. Ainsi, pour le même disque, l'ensemble de la table des bits occuperait 64 blocs de 2 ko.

La différence entre les performances des deux approches dépend de la mémoire disponible pour stocker ces structures auxiliaires : la table est plus compacte et donc peut plus facilement être chargée entièrement en mémoire, mais si un seul bloc auxiliaire peut être présent en mémoire centrale ou que le disque est presque plein, la liste est plus intéressante.

Gestion des blocs endommagés

La gestion des blocs endommagés peut se faire de façon matérielle. Dans ce cas, le disque possède un certain nombre de blocs de remplacement non adressables par l'utilisateur, ainsi qu'une table interne contenant la liste des blocs endommagés et leur numéro de bloc de remplacement. Des primitives spécifiques du contrôleur disque permettent la vérification de la surface du disque à la recherche des blocs endommagés et la mise à jour de cette table interne.

Elle peut également se faire de façon logicielle au niveau du système. Dans ce cas, le système de fichiers maintient une table des blocs endommagés, qui ne feront alors jamais partie de la liste des blocs libres. Cette deuxième approche permet de continuer à utiliser des disques vieillissants ayant épuisé leur quota de blocs de remplacement.

Vérification de la cohérence

L'utilisation de tampons et de caches sur les disques et le système pose des problèmes de synchronisation et de cohérence des données en cas d'arrêt brutal du système. Tous les systèmes de fichiers disposent donc de programmes utilitaires (|| `scandisk` || sous Windows, || `fsck` || sous Unix), dont le but est de vérifier et de restaurer la cohérence du système de fichiers, en préservant si possible l'intégrité des données. Pour être exhaustive, cette vérification s'effectue à la fois au niveau des blocs et au niveau des fichiers :

- on commence par construire une table comportant deux compteurs par

bloc, dont l'un mémorise le nombre de fois que le bloc est supposé faire partie d'un fichier, et l'autre le nombre de fois qu'il apparaît dans la liste des blocs libres ;

- on parcourt alors l'ensemble des i-nœuds, en parcourant pour chacun la liste de ses blocs déclarés ;
- on parcourt ensuite la liste des blocs libres ;

L'ensemble du système est dans un état cohérent si tous les blocs ont l'un de leurs compteurs à un et l'autre à zéro. Sinon, il y a erreur, qui peut être de plusieurs types :

- les deux compteurs sont à zéro : le bloc perdu est réaffecté à la liste des blocs libres ;
- le compteur de fichiers est égal à zéro et le compteur de bloc libre est supérieur à un : les doublons du bloc multiples libre sont effacés de la liste des blocs libres ;
- le compteur de fichiers est égal à un et le compteur de bloc libre est supérieur à zéro : les occurrences du bloc faussement libres sont enlevées de la table des blocs libres ;
- le compteur de fichiers est supérieur à un : on effectue autant de copies du bloc multiples utilisé qu'il y a de doublons, et on remplace le numéro du bloc par les numéros des blocs copiés dans chacun des fichiers incriminés. Les données des fichiers sont sûrement corrompues, mais la structure du système de fichiers est restaurée.

Une autre vérification effectuée consiste à parcourir l'ensemble de l'arborescence pour construire une table, indexée par le numéro de l'i-nœud, du nombre de répertoires faisant référence à chacun des i-nœuds. On peut donc comparer le nombre de références trouvées à celui conservé dans chaque i-nœud. Deux erreurs sont possibles :

- la valeur du compteur est supérieure au nombre de références : l'i-nœud ne sera jamais réclamé ;
- la valeur du compteur est inférieure au nombre de références : l'i-nœud peut être libéré de façon anticipée.

Dans les deux cas, on corrige l'erreur en mettant en conformité le compteur avec le nombre de références trouvées. Si celui-ci est égal à zéro, on conserve néanmoins une référence sur l'i-nœud dans un répertoire spécial (appelé *lost+found* `;;` sous Unix, par exemple) destiné à recueillir les fichiers et répertoires orphelins.

4.2.4 Service des requêtes

La performance du disque dépend de trois paramètres :

- le temps de positionnement, nécessaire pour aligner le bras du disque sur le bon cylindre ;
- le temps de latence, nécessaire pour que le bloc désiré passe sous la tête de lecture/écriture ;
- le temps de transfert entre la surface du disque et la mémoire centrale.

Les disques, comme tous les périphériques du système, possèdent une file d'attente des requêtes en cours. Cette file, gérée en interne par le contrôleur de chaque disque, stocke des requêtes constituées de champs tels que :

- la direction de l'entrée/sortie (lecture ou écriture) ;
- l'adresse du disque (numéro de bloc) ;
- l'adresse en mémoire physique à partir d'où, ou vers où, copier les données.

Si l'unité de disque est disponible, la requête est traitée immédiatement. Cependant, pendant que l'unité traite une requête, toutes les requêtes supplémentaires sont placées en file d'attente.

Pour un système multiprogrammé, la file d'attente peut souvent être non vide. Lorsque la requête en cours est terminée, il s'agit donc de choisir la prochaine requête à exécuter afin de minimiser le temps de positionnement des têtes.

La minimisation du temps de positionnement était déjà la motivation principale de l'organisation des systèmes de fichiers en cylindres, et de l'allocation contiguë des blocs des fichiers sur des cylindres proches. Notons cependant que, pour les disques modernes, du fait de la densité croissante des informations sur la surface et des contraintes thermiques et mécaniques sur les bras et les plateaux (dilatations dues à la chaleur), un réalignement fin des têtes est nécessaire avant chaque accès, ce qui constitue un coût incompressible et limite le gain découlant de l'allocation des blocs sur des cylindres proches.

Plusieurs algorithmes d'ordonnement des requêtes (c'est-à-dire de sélection de la prochaine requête à exécuter) ont été proposés.

Ordonnement FCFS

La forme la plus simple de sélection des requêtes est l'ordonnement FCFS (à *First Come, First Served*). Cet algorithme est équitable et facile à programmer, mais ne donne pas les meilleures performances, car il peut provoquer un déplacement frénétique du bras du disque lorsque plusieurs accès séquentiels à des fichiers éloignés les uns des autres sont effectués de façon concurrente.

Ordonnement SSTF

Il semble raisonnable de servir ensemble toutes les requêtes proches de la position courante des têtes avant de déplacer celles-ci vers une autre zone du disque. L'algorithme SSTF (à *Shortest Seek Time First*) sélectionne donc la requête demandant le temps de positionnement le plus petit à partir de la position courante du bras.

L'ordonnement SSTF, bien que réduisant fortement la distance totale parcourue par le bras (il n'est cependant pas optimal), peut conduire à la famine de requêtes éloignées lorsque de nouvelles requêtes proches de la position courante arrivent continuellement.

Ordonnement SCAN

Un algorithme d'optimisation du traitement des requêtes ne peut être équitable que si l'on impose de ne pas laisser continuellement le bras sur la même zone du disque.

L'algorithme SCAN (ou *balayage à l'*, parfois aussi appelé *algorithme de l'ascenseur à l'*) consiste à placer le bras à une extrémité du disque, puis à le déplacer vers l'autre extrémité en s'arrêtant au passage pour traiter les requêtes présentes dans la file. Lorsque le bras a atteint l'autre extrémité du disque, on inverse la direction du sens de mouvement du bras, les têtes balayant ainsi continuellement la surface du disque.

Ordonnancement C-SCAN

Si les requêtes arrivant sur le contrôleur sont uniformément distribuées, lorsque le bras arrive à une extrémité du disque, il existe relativement peu de requêtes proches de lui, et les requêtes les plus anciennes sont situées le plus loin de lui.

L'algorithme C-SCAN (ou *Circular SCAN à l'*), comme l'algorithme SCAN, déplace le bras d'une extrémité à l'autre du disque en servant les requêtes au passage. Cependant, lorsqu'il arrive à l'autre extrémité du disque, il renvoie immédiatement le bras au début du disque, sans traiter de requêtes au passage, afin de servir le plus rapidement les requêtes les plus anciennes, et d'uniformiser ainsi le temps d'attente. Il considère la surface du disque comme si elle était circulaire (torique, en fait), d'où le nom.

Dans la pratique, le bras n'est pas déplacé jusqu'aux extrémités du disque, mais s'arrête à la dernière requête à servir, et est ramené jusqu'à la position de la première requête à traiter. Cette version plus efficace de l'algorithme C-SCAN est appelée C-LOOK.

Les performances relatives de tous ces algorithmes dépendent principalement du nombre et du type des requêtes. Lorsque les contrôleurs sont peu sollicités, l'ordonnancement SSTF donne de très bons résultats, alors que les algorithmes SCAN et C-SCAN se comportent mieux lorsque la charge augmente.

Le service des requêtes dépend très fortement de la méthode d'allocation des fichiers : un programme lisant un fichier de façon séquentielle générera plusieurs accès proches l'un de l'autre, alors qu'un accès à un fichier indexé peut générer des requêtes éparpillées sur le disque. L'emplacement des structures de répertoire et d'index (i-nodes) est également essentiel, puisqu'il faut ouvrir tout fichier préalablement à son utilisation. On organise donc souvent les systèmes de fichiers en groupes de cylindres, chaque groupe disposant de plages d'i-noeuds et de chaînages des blocs libres, afin d'allouer les blocs des fichiers dans des cylindres voisins et de limiter le parcours des têtes lors des opérations sur les fichiers. De même, la partition de swap est habituellement placée au milieu du disque. Cependant, ces choix logiciels de structure ne sont valables que si le contrôleur applique une politique d'ordonnancement compatible.

Chapitre 5

Exemples de systèmes d'exploitation

5.1 Mach

L'ancêtre de Mach est le système d'exploitation Accent, développé à l'Université de Carnegie Mellon. La philosophie et le système de communication de Mach dérivent directement d'Accent, mais plusieurs autres éléments significatifs du système, comme les systèmes de gestion de la mémoire virtuelle, des tâches, des threads, et des architectures multiprocesseurs, sont originaux.

Le développement de Mach s'est appuyé sur celui de l'Unix BSD, en se servant de celui-ci comme d'un échafaudage. Le code de Mach a initialement été introduit dans un noyau BSD 4.2 d'accueil, en remplaçant les composants BSD par des composants Mach équivalents à mesure qu'ils devenaient disponibles. À partir de sa version 3, Mach est devenu un système micro-noyau complet, le code de compatibilité BSD ayant été déplacé depuis le noyau vers des serveurs exécutés en mode utilisateur. Cette modularisation permet de remplacer BSD par un autre système d'exploitation (OSF/1, HPUX, OS/2, DOS, ou autre), voire d'exécuter simultanément plusieurs interfaces de systèmes d'exploitation au dessus du micro-noyau. Les fonctionnalités offertes sont similaires à celles des machines virtuelles (voir section 1.8.3, page 18), mais leur support est logiciel (l'interface du noyau Mach servant de machine virtuelle) plutôt que matériel.

Le système Mach a été conçu pour fournir des mécanismes de base que la plupart des systèmes d'exploitation courants ne possèdent pas. L'objectif de Mach était de concevoir un système d'exploitation compatible avec BSD (à la différence d'Accent), et possédant les fonctionnalités suivantes :

- conception orientée objet tant au niveau interne qu'au niveau de son interface ;
- support pour les architectures diverses, depuis les machines multi-processeurs à mémoire partagée jusqu'au multi-ordinateurs à mémoire distribuée ;
- capacité de fonctionner avec des réseaux d'interconnexion variés, depuis les réseaux d'interconnexions des machines parallèles et les réseaux locaux rapides jusqu'aux réseaux à grande distance ;

- fonctionnement réparti et transparent pour l'utilisateur, avec support de systèmes hétérogènes permettant le fonctionnement entre plusieurs architectures provenant de vendeurs différents.
- simplicité de structure du noyau, basé sur un nombre d'abstractions très réduit, mais suffisamment puissantes pour permettre l'émulation d'autres systèmes d'exploitation au dessus de Mach ;
- gestion mémoire et communication interprocessus intégrés ;

Le système Windows NT est l'un des héritiers de Mach, en reprenant à son compte une bonne partie des idées développées initialement dans le cadre de ce projet.

5.1.1 Principes de conception

Mach est un système orienté objet, dont les données et les méthodes qui les manipulent sont encapsulées dans des objets abstraits. L'utilisation de l'approche orientée objet simplifie la conception et l'utilisation des différents objets système, et rend leur localisation dans un ensemble de systèmes Mach transparente à l'utilisateur.

Mach est construit autour d'un nombre réduit d'abstractions primitives puissantes.

- La *tâche* (*task*) est un environnement d'exécution offrant le support de base pour l'allocation de ressources. Elle est constituée d'un espace d'adressage virtuel, et accède aux ressources protégées du système par l'intermédiaires de *ports*. Une tâche peut contenir un ou plusieurs *threads*.
- Le *thread* est l'unité d'exécution, et ne peut exister que dans une tâche, qui lui fournit l'espace d'adressage nécessaire à son exécution. Tous les threads d'une tâche partagent l'ensemble de ses ressources (ports, mémoire, ...).
- Le *port* est le mécanisme de base servant à référencer les objets système. Il est implémenté comme une voie de communication protégée, qui permet l'envoi de messages sur un port de destination, où ils sont mis en file avant de pouvoir être lus. L'accès aux ports est protégé par le système au moyen de *droits* : une tâche doit posséder un droit de port sur un port donné pour que ses threads puissent lui envoyer un message. Un thread appelle une méthode sur un objet en envoyant un message sur le port associé à cet objet.
- L'*ensemble de ports* définit un groupe de ports partageant une file d'attente de messages commune. Un thread peut attendre des messages d'un ensemble de ports, et ainsi servir plusieurs ports. Chaque message possède un identificateur de port, qui peut être utilisé par le thread pour identifier l'objet destinataire du message.
- Le *message* est l'objet de communication de base entre threads. Il est implémenté comme un ensemble typé d'objets, qui peuvent être passés par valeur ou par référence.
- L'*objet mémoire* est une source de mémoire (mémoire swap, segment de mémoire partagée, fichier, pipe, ...). Les tâches peuvent accéder aux zones mémoire associées à ces objets en transformant les références à ces objets en portions de l'espace d'adressage.

5.1.2 Gestion des processus

Une tâche est constituée d'un espace d'adressage virtuel, d'un ensemble de droits de ports, et d'informations de gestion. C'est une entité passive, qui n'effectue aucun travail sauf si un ou plusieurs threads s'exécutent en son sein.

Les threads sont particulièrement utiles dans les applications serveur, car une tâche peut ainsi utiliser plusieurs threads pour servir concurremment plusieurs requêtes, chaque thread pouvant s'exécuter sur un processeur distinct d'une machine multi-processeurs.

Les algorithmes d'ordonnancement d'un système multiprocesseurs à threads sont plus complexes que ceux d'un système à processus non multi-threadés. Afin de ne pas rendre son ordonnanceur trop complexe, l'ordonnanceur de Mach ne travaille qu'au niveau des threads, ceux-ci étant en concurrence de la même manière pour toutes les ressources, y compris le quantum de temps¹. À chaque thread est associé une valeur de priorité dynamique, calculée selon la moyenne exponentielle de son utilisation de la CPU (c'est-à-dire qu'un thread venant d'utiliser la CPU pendant une longue période de temps obtient la priorité la plus basse). Cette valeur de priorité est utilisée pour placer le thread dans l'une des 32 files globales de priorité. Mach possède également une file de priorité par processeur, recevant spécifiquement les threads attachées à un processeur particulier, comme par exemple les gestionnaires des périphériques attachés à ce processeur.

Au lieu d'avoir un répartiteur centralisé pour affecter les threads aux processeurs, chacun de ceux-ci examine sa file locale et les files globales afin de sélectionner le prochain thread à exécuter. Les threads de la file locale sont prioritaire sur ceux de la file globale, car on considère qu'ils exécutent un service pour le noyau.

Comme il peut y avoir moins de threads exécutable que de processeurs dans le système, l'ordonnanceur de Mach fait varier la durée de son quantum de temps comme l'inverse du nombre de threads en attente, afin de ne pas gaspiller de temps CPU en commutation de tâches inutiles alors que des processeurs sont inactifs. Cependant, l'ordonnanceur maintient un délai maximal d'attente constant sur tout le système, afin de ne pas pénaliser les threads entrants. Ainsi, sur un système avec 10 processeurs, 11 threads en attente, et un temps de quantum de 100 milli-secondes, il doit se produire une commutation de contexte seulement une fois par seconde sur chaque processeur pour que le thread en attente ait une probabilité d'attente maximale proche de la taille du quantum.

Mach a été conçu pour fournir un système unique de traitement des exceptions, simple et consistant, permettant à la fois de traiter les exceptions standard et celles définies par l'utilisateur. Afin d'éviter de dupliquer du code, Mach utilise des primitives noyau existantes chaque fois que c'est possible. Ainsi, une routine de traitement des exceptions n'est qu'un thread supplémentaire dans les tâches où se produisent les exceptions : on utilise les messages d'appel de procédures à

1. On entrevoit ici une autre possibilité d'accaparement de la CPU découlant de la connaissance par l'utilisateur de l'algorithme d'ordonnancement, analogue à celle relatée en page 41 : un utilisateur désirant que son programme de calcul s'exécute plus vite obtiendra un meilleur temps de réponse s'il décompose ses calculs en de nombreux threads concurrents...

distance (RPC) pour synchroniser l'exécution du thread provoquant l'exception et celle de la routine de traitement, les informations liées à l'exception étant passées en arguments dans le message. Le traitement des exceptions fonctionne en deux temps. Tout d'abord, le thread levant l'exception notifie l'occurrence de l'exception à travers un message RPC `raise` envoyé à la routine de traitement, puis appelle une routine d'attente de terminaison du traitement de l'exception. La routine de traitement reçoit la notification de l'exception, et effectue le traitement approprié en fonction du type de l'exception, avant d'autoriser la reprise du thread appelant ou bien de terminer la tâche englobante.

Mach offre également un support des signaux Unix BSD, ce qui n'est pas sans poser problème. En particulier, les signaux ont été conçus pour des environnements mono-threads, ce qui fait que, bien qu'il n'y ait pas de raison que tous les threads d'une tâche perçoivent un signal et s'interrompent, il est difficile de faire en sorte qu'un signal ne soit perçu que par un seul thread de la tâche. Qui plus est, les signaux Unix BSD peuvent être perdus, lorsqu'un autre signal se produit avant que le premier ne soit traité, alors que les exceptions Mach sont mise en file d'attente sans perte du fait de leur implémentation RPC.

Les signaux générés extérieurement, y compris ceux envoyés par un processus BSD à un autre, sont traités par la partie serveur du module de compatibilité BSD du noyau Mach. Leur comportement est le même que sous BSD. Les exceptions matérielles sont traitées de façon particulière, car les programmes BSD s'attendent à recevoir les exceptions matérielles comme des signaux. Pour cela, les RPCs produits par les exceptions matérielles ont pour destinataire par défaut une tâche du noyau dont le thread boucle en continu sur la réception de messages RPC, qu'elle transforme en un signal qu'elle renvoie au thread qui a provoqué l'exception matérielle. À la terminaison du RPC, le thread en question reprend son exécution, aperçoit donc immédiatement le signal, et exécute son code de traitement.

5.1.3 Communication inter-processus

Les deux composants des communications interprocessus sont les ports et les messages. Presque tout dans Mach est un objet, et l'on accède à tous les objets à travers leurs ports de communication.

On implémente un port comme une file d'attente limitée, protégée à l'intérieur du noyau dans lequel réside l'objet. Si une file d'attente est pleine, l'émetteur d'un message peut annuler l'envoi, attendre qu'un emplacement devienne libre dans la file d'attente, ou bien demander au noyau de délivrer le message à sa place.

On peut grouper des ports dans des ensembles de ports. Un port peut être membre d'au plus un ensemble de ports à la fois, et tant qu'un port se trouve dans un ensemble, il ne peut plus être utilisé directement pour recevoir des messages. Au contraire d'un port, un ensemble de ports ne peut pas être transféré au sein de messages. Les ensembles de ports sont des objets qui remplissent un rôle semblable à celui de l'appel système `select()` d'Unix, mais ils sont plus efficaces.

Un message est constitué d'un en-tête de longueur fixe et d'un nombre variable d'objets de données typés. Chaque donnée peut consister en un type

simple, des droits de ports, ou des références sur des données extérieures.

L'utilisation de références permet de transférer l'espace d'adressage entier d'une tâche en un seul message. Les données référencées dans un message envoyé à un port situé sur la même machine ne sont pas copiées entre les espaces d'adressage de l'émetteur et du récepteur ; la table des pages de la tâche réceptrice est modifiée pour partager en lecture (avec copie en écriture) les pages du message.

5.1.4 Gestion de la mémoire

Les concepteurs de Mach ayant souhaité pousser au maximum l'orientation objet de leur système, la gestion de la mémoire s'effectue également au moyen d'objets mémoire.

À l'inverse de la plupart des systèmes, qui supportent de façon spécifique au niveau noyau la gestion de la mémoire de masse et des systèmes de fichiers, Mach traite les objets représentant la mémoire masse comme tous les autres objets mémoire du système. Chaque objet mémoire possède un port, et peut être manipulé par l'intermédiaire de messages envoyés à son port. Les objets mémoire, à l'opposé des routines de gestion mémoire des noyaux monolithiques, permettent une expérimentation simple de nouveaux algorithmes de gestion de la mémoire.

L'espace d'adressage virtuel d'une tâche est généralement creux, constitué d'une succession de plages libres et de plages allouées. Par exemple, en plus de la mémoire virtuelle utilisée par le va-et-vient, des plages d'adresses distinctes sont attribuées à chaque fichier mappé en mémoire, et les longs messages sont échangés entre tâches sous forme de segments de mémoire partagée. À mesure que de nouveaux objets sont alloués ou supprimés de l'espace d'adressage, des plages libres apparaissent dans l'espace d'adressage.

Mach n'essaye pas de compacter l'espace d'adressage, malgré le fait qu'une tâche avorte si elle ne peut allouer suffisamment de place contiguë à un objet donné, car la capacité d'adressage virtuel des processeurs est toujours bien plus grande que nécessaire (plusieurs téra-octets pour les processeurs récents). La gestion efficace de la table des pages pour un espace d'adressage aussi étendu ne peut se faire qu'au moyen de tables de pages à plusieurs niveaux (voir page 55) ou, sur les systèmes plus anciens, de façon logicielle en parcourant une liste chaînée des plages allouées afin de trouver la portion de table de pages correspondante.

Mach offre les fonctionnalités standard de gestion de la mémoire virtuelle, telles que l'allocation, la libération, et la copie de mémoire virtuelle. Quand on alloue un nouvel objet de mémoire virtuelle, le thread appelant peut fournir lui-même une adresse pour l'objet, ou laisser le noyau choisir l'adresse. La mémoire physique n'est pas allouée avant le premier accès effectif aux pages allouées.

Les objets de la mémoire auxiliaire en cours d'utilisation par une tâche sont mappés dans l'espace d'adressage virtuel de celle-ci, et une partie seulement de leurs pages est conservée en mémoire physique, de façon tout à fait classique. Cependant, un défaut de page se produisant quand un thread veut accéder à une page non présente en mémoire physique est implémenté comme l'envoi d'un message au port de l'objet mémoire responsable de la page. De même, lorsqu'un

objet mémoire est sur le point d'être détruit, c'est à son gestionnaire mémoire de synchroniser toutes les pages modifiées avec la mémoire auxiliaire ; Mach ne fait aucune supposition quant au contenu ou à l'importance des objets mémoire, qui sont totalement indépendants du noyau. La notion d'objet mémoire, géré par des mécanismes n'appartenant pas au noyau, est très puissante : elle permet à l'utilisateur d'écrire ses propres gestionnaires mémoire.

La politique de transfert de pages vers la mémoire de masse (pageout) est implémentée par un thread interne au noyau, le *démon pageout*. Un algorithme de pagination basé sur un algorithme FIFO avec seconde chance (voir page 58) est utilisé pour sélectionner les pages à remplacer. Les pages sélectionnées sont alors envoyées à leur gestionnaire mémoire (fourni par l'utilisateur ou par défaut) pour transférer effectivement les pages vers la mémoire de masse. Si un gestionnaire fourni par l'utilisateur n'arrive pas à réduire le nombre de pages résidant en mémoire physique quand le noyau le lui demande, celui-ci appelle le gestionnaire par défaut, qui transfère dans l'espace de swap les pages du gestionnaire mémoire utilisateur. Une fois que celui-ci aura surmonté le problème qui l'empêchait d'effectuer ses propres transferts, il cherchera à référencer ses pages, qui seront alors ramenées en mémoire par le noyau avant d'être enfin re-transférées par le gestionnaire utilisateur.

Lorsqu'un thread doit accéder à des données contenues dans un objet mémoire (comme par exemple un fichier), il invoque l'appel système `vm_map` avec le port identifiant l'objet et le gestionnaire mémoire responsable de la zone. Le noyau exécute alors des appels sur ce port lorsqu'il doit lire ou écrire des données dans cette zone.

Les gestionnaires mémoire sont responsables de la cohérence des contenus d'un objet mémoire visibles par des tâches situées sur des machines différentes (les tâches d'une même machine partageant une copie unique de l'objet mémoire). Dans le cas où les tâches de deux machines distinctes tentent de modifier simultanément la même page d'un objet, c'est au gestionnaire mémoire de l'objet de décider si les modifications doivent être sérialisées ou non : un gestionnaire conservateur implémentant une politique de cohérence forte forcera la sérialisation des écritures, alors qu'un gestionnaire plus sophistiqué pourra permettre que les accès s'effectuent de façon concurrente s'il sait que les zones de la pages modifiées par les deux pages sont distinctes et pourront être fusionnées dans le futur, ou bien si une cohérence faible suffit (pour un objet vecteur solution distribué dans un calcul matriciel itératif, par exemple).

Quand le premier appel `vm_map` est effectué sur un objet mémoire, le noyau envoie un message au port du gestionnaire mémoire passé en paramètre, en appelant la routine `memory_manager_init` que le gestionnaire mémoire doit fournir comme élément de son support d'un objet mémoire. Quand un thread provoque un défaut de page pour une page de l'objet mémoire, le noyau envoie un `memory_object_data_request` au port de l'objet mémoire, au nom du thread qui a causé le défaut de page. Le thread est mis en attente jusqu'à ce que le gestionnaire mémoire retourne la page avec un appel `memory_object_data_provided`, ou bien qu'il renvoie une erreur appropriée au noyau. Toutes les pages modifiées ou précieuses que le noyau doit enlever de la mémoire physique (pour libérer de la place pour le va-et-vient, par exemple) sont envoyées à l'objet mémoire à travers un appel `memory_object_data_write`. Les pages précieuses sont celles qui peuvent ne pas avoir été modifiées mais que l'on ne peut pas simplement

détruire parce que le gestionnaire mémoire n'en possède pas de copie; elles permettent d'économiser la duplication et la recopie inutiles de mémoire.

Mach utilise la mémoire partagée pour augmenter l'efficacité de nombreuses fonctions du système; elle permet en particulier une communication inter-tâches extrêmement rapide.

Dans le cas où les tâches sont localisées sur la même machine, Mach offre naturellement un service de mémoire partagée cohérente. Une tâche père peut déclarer lesquelles de ses zones mémoire seront héritées par ses fils, et lesquelles seront en lecture seule ou en écriture.

Dans le cas de la mémoire partagées entre machines distinctes, Mach permet l'utilisation de gestionnaires mémoire externes. Un ensemble de tâches désirant partager un segment de mémoire peuvent utiliser le même gestionnaire mémoire externe et accéder par son intermédiaire aux mêmes zones de mémoire. L'implémenteur du gestionnaire mémoire externe peut réaliser une routine de pagination externe aussi simple ou bien aussi complexe que nécessaire. Un algorithme relativement simple consiste à invalider toutes les copies distantes d'une page lors d'une écriture, en interdisant tout accès concurrent pendant la modification. Il est implémenté dans Mach en tant que *Network Memory Server*, et sous une forme plus évoluée en tant que *XMM* dans la version de Mach 3.0 pour architectures à mémoire distribuée.