Support de cours
système d'exploitation

J. Gispert, J. Guizol, J.L. Massat

Département d'informatique
Faculté de Luminy
163, Avenue de Luminy, Case 901,
13288 Marseille, cedex 9

23 février 2012

Chapitre 1

Organisation d'un système d'exploitation

1.1 Fonctionnalités d'un système informatique

En se limitant au seul point de vue d'un utilisateur d'un système informatique, les fonctions devant être assurées par celui-ci peuvent être résumées de la façon suivante :

- **Gestion et conservation de l'information**. Le système informatique doit permettre à tout utilisateur de créer, conserver, retrouver ou détruire les *objets* sur lesquels celui-ci désire effectuer des opérations.
- **Préparation, mise au point et exploitation de programmes**.

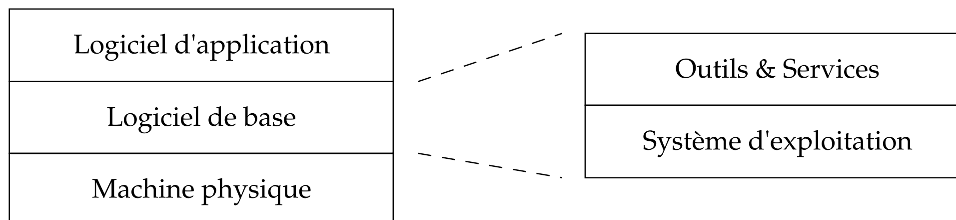


FIG. 1.1 – Structure logicielle d'un système informatique

La figure ?? précise l'organisation logicielle d'un système informatique avec d'une part le logiciel d'application (traitement de textes, gestionnaire de bases de données, compilateurs, etc.) et d'autres part le logiciel de base livré avec la machine.

1.1.1 Fonctions d'un système d'exploitation

Le système d'exploitation est un des éléments clef d'un système informatique. Il reprend à son compte les deux fonctions précédentes en y ajoutant des nouvelles fonctions liées à la « bonne gestion » de la machine physique :

- **Structuration de l'information** (sous forme de fichiers) en vue de sa conservation et de sa modification.
- **Transfert des données** entre les éléments constituants du système informatique (unité centrale, périphériques d'impression ou de lecture, modem, etc.).
- **Gestion de l'ensemble des ressources** pour offrir à tout utilisateur un environnement nécessaire à l'exécution d'un travail.

- **Gestion du partage des ressources**. Le système doit répartir les ressources dont il dispose entre les divers usagers en respectant la règle d'*équité* et en empêchant la *famine*. En particulier, il doit réaliser un ordonnancement des travaux qui lui sont soumis et éviter les *interblocages*.
- **Extension de la machine hôte**. Le rôle du système est ici de simuler une machine ayant des caractéristiques différentes de celles de la machine réelle sur laquelle il est implanté. Chaque utilisateur dispose alors d'une *machine virtuelle* munie d'un *langage étendu* permettant l'exécution et la mise au point des programmes au moyen d'outils plus facilement utilisables que ceux dont est dotée la machine câblée.

1.1.2 Aspects externes

La diversité des tâches à remplir et des matériels utilisés a pour conséquence une grande variété des aspects externes des systèmes :

- les systèmes destinés à la conduite de processus industriels (chimie industrielle, cracking, central téléphonique, guidage de fusée, surveillance médicale ou « monitoring », etc.) ;
- les systèmes gérant les bases de données (réservations de places, gestion de stock, gestion de comptes bancaires, documentation automatique, etc.) ;
- les systèmes destinés à la création et l'exécution de programmes qui peuvent être subdivisés en plusieurs classes selon :
  - le degré d'interaction entre l'utilisateur et ses programmes (traitement par trains ou conversationnel) ;
  - le mode de partage des ressources (mono ou multiprogrammation) ;
  - les possibilités offertes par le langage étendu (accès à un ou plusieurs langages) ;

Malgré cette grande diversité, les systèmes comportent entre eux des parties très ressemblantes, voire identiques, et il serait donc très utile de pouvoir dégager celles-ci afin de profiter de certaines études partielles dans l'élaboration de portions plus complexes. C'est ce souci de rentabilisation qui a conduit à une conception modulaire des systèmes et de leurs différents constituants, technique généralisable à tout logiciel développé sur une machine quelconque.

1.2 Structure interne des systèmes d'exploitation

1.2.1 Conception descendante et structures en couches

Dans cette section, le terme de « langage » sera utilisé dans le sens suivant : un *langage* définit des *objets* (et les mécanismes permettant de les créer), des *actions* (ou *primitives*) permettant de manipuler ces objets et des *règles de composition* de ces actions.

En particulier, tout langage définit une « machine » capable de l'interpréter : les instructions de cette machine représentent l'ensemble des primitives du langage, sa mémoire permet de représenter les objets et son mécanisme d'exécution est celui défini par les règles d'interprétation du langage.

Désirant résoudre un problème, une démarche habituelle consiste à décomposer celui-ci en une succession de plusieurs sous-problèmes que l'on espère résoudre plus aisément. On essaie donc dans un premier temps de définir une machine M_0 dont les primitives rendront la résolution du problème plus facile. Le problème initial (supposé résolu par la machine M_0) se transforme donc en la réalisation de cette machine M_0 sur la machine disponible M. On va alors définir pour cela une machine M_1, etc. jusqu'à l'obtention d'une machine M_n facilement réalisable sur M.

La puissance de cette méthode ne réside pas dans la seule simplification du problème à chaque niveau, mais résulte aussi du processus d'*abstraction* consistant à focaliser l'étude sur les aspects essentiels du problème, concrétisés par la spécification d'une machine, c'est à dire, en fait, celle de son *interface*.

Lorsque la réalisation d'une machine M_i utilise l'interface d'une machine M_j, on dit que M_i dépend de M_j. En fait, cette relation de dépendance ne porte que sur l'interface et non sur la réalisation interne. On peut alors décrire la structure d'un système par un graphe dont les nœuds représentent les machines, et les arcs, les relations de dépendance.

Dans la figure **??**, le schéma (a) représente la structure résultant de la méthode de conception descendante. Celui-ci peut être généralisé en (b) si l'on autorise chaque machine à utiliser les primitives de toute autre machine de niveau inférieur. Enfin, si la seule contrainte est d'avoir un graphe sans circuit, on obtient le schéma (c) où les machines sont classées par niveau d'abstraction, chacune d'elles n'utilisant que les machines de niveau inférieur.

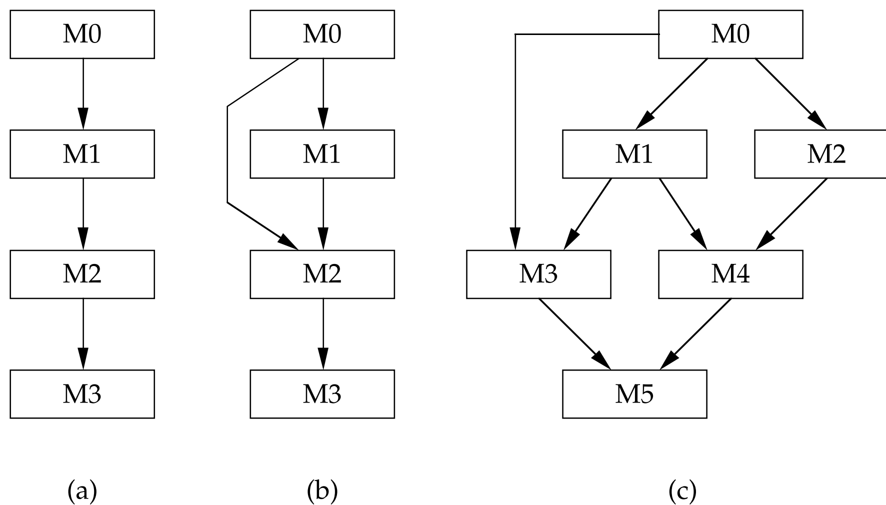


FIG. 1.2 – Décomposition hiérarchique

En réalité, la méthode de conception descendante est rarement utilisable à l'état pur. Plusieurs facteurs sont à prendre en considération : l'expérience du concepteur, l'existence de machines déjà réalisées, la difficulté de fixer à l'avance les spécifications détaillées des interfaces, qui utilisent en fait le résultat d'expérimentation sur des réalisations partielles. En tout état de cause, l'*indépendance* introduite par l'abstraction procure à la structure hiérarchique par niveau plusieurs avantages :

- **Indépendance pour la conception**. Description totale du comportement d'une machine par les spécifications de son interface.
- **Indépendance pour la modification**. Les modifications dans la réalisation d'une machine n'altèrent en rien celles qui l'utilisent si les spécifications d'interface demeurent inchangées.
- **Indépendance pour la mise au point**. Son interface ayant été spécifiée, une machine M peut être mise au point indépendamment de celles qui l'utilisent ; réciproquement, une machine M étant réalisée, les machines utilisant M peuvent être mises au point indépendamment de M.

1.2.2 Notion d'objet

La décomposition hiérarchique que l'on vient de présenter répond mal à certains aspects de structuration des systèmes. En particulier, la description de collections d'éléments ayant des caractéristiques communes, la création ou la destruction dynamique d'éléments seront facilitées grâce

à la notion d'*objet*, nouvel outil de structuration permettant d'exprimer des concepts importants tels que : désignation, liaison, type, protection.

Un objet est concrétisé par une *représentation* : représentation externe à laquelle a accès l'utilisateur et représentation interne qui est celle concernant le système. Cette représentation des objets ainsi que la façon d'y accéder font appel à des *fonctions d'accès*.

Un autre concept permet de regrouper un ensemble d'objets ayant des caractéristiques communes : c'est la notion de *classe*. Les opérations et fonctions d'accès associées à une classe sont applicables à chaque objet de la classe. Il peut advenir que l'on désire définir des ensembles d'objets ayant à la fois des propriétés communes avec une classe déjà existante et des propriétés particulières. La notion de *sous-classe* permet d'y parvenir.

Une sous-classe hérite des propriétés associées à la classe mère, auxquelles s'ajoutent des propriétés spécifiques. Cette sous-classe pourra à son tour être considérée comme classe mère d'un autre ensemble, etc. On obtient ainsi une hiérarchie de classes. Citons quelques classes bien connues, et la ressource physique associée dont elles constituent une abstraction :

- les fichiers → mémoire secondaire,
- les flots → organes périphériques,
- les processus → processeur,
- la mémoire virtuelle → mémoire physique.

Les fonctions d'accès associées à une classe quelconque permettent, entre autre, de *créer* ou de *supprimer* des objets de cette classe et donc d'en faire varier dynamiquement le nombre. Une fois créé, un objet dispose d'un *état* représenté par ses propres données qui peuvent varier dans le temps.

1.2.3 Interfaces et spécifications

Comme nous l'avons vu dans les deux paragraphes précédents, une interface est associée, soit à une machine abstraite, soit à une classe d'objets. Elle comporte trois types d'information :

- des structures de données ;
- des procédures ;
- des règles d'utilisation des données et procédures exprimant des restrictions :
  - restrictions d'accès aux données (lecture seule autorisée...) ;
  - restrictions sur l'ordre des procédures ;
- contraintes de simultanéité dans l'exécution des procédures ou l'accès aux données.

A l'heure actuelle, aucune solution satisfaisante n'a encore été proposée pour exprimer les spécifications d'une interface ou les contraintes d'utilisation, si ce n'est le recours au langage naturel. Deux méthodes sont utilisées pour prendre en compte, dans la spécification, les éventuelles erreurs :

- Chaque procédure comporte un paramètre supplémentaire (code d'erreur) modifiable par la procédure. La valeur finale de cette variable constitue un compte-rendu interprétable à un niveau supérieur.

- A chaque cause d'erreur est associée une procédure de traitement spécifique. En cas d'erreur, un mécanisme que nous détaillerons ultérieurement (déroutement), déclenche automatiquement la procédure correspondante.

Dans tous les cas, le traitement d'une erreur consistera à revenir à un état stable du système où l'exécution puisse reprendre normalement, **en perdant le moins d'information possible**. Des deux méthodes de prise en compte d'erreur présentées plus haut, la seconde nécessite un mécanisme supplémentaire, mais elle sera préférable à la première pour deux raisons essentielles :

– **Sécurité** : le caractère systématique de déroutement en cas d'erreur est supérieur au test de code qui peut être omis, provoquant ainsi une propagation d'erreur.

– **Clarté** : le fait de pouvoir associer une procédure particulière à chaque cause d'erreur permet de séparer clairement le traitement des situations « normales » de celui des situations « exceptionnelles ».

Les méthodes de conception que nous venons de présenter et les concepts ou outils d'abstraction qui leur sont associés permettent, on l'a vu, grâce à la modularité ainsi acquise, de diviser et subdiviser un système informatique en plusieurs parties indépendamment modifiables ou même interchangeables.

Cet aspect s'avère primordial, car les systèmes élaborés à l'heure actuelle sont de plus en plus importants et de plus en plus complexes. Si bien que leur réalisation est confiée à plusieurs personnes, à divers services, voire à plusieurs équipes. La cohérence du tout ne peut être facilement obtenue qu'à la condition d'avoir préalablement clairement défini les spécifications d'interface et l'arborescence des classes d'objets manipulés,... mais seulement cela.

D'autre part, certaines portions du système peuvent être conçues de différentes façons en utilisant différentes stratégies. Dans ces conditions, les concepteurs devront tester celles-ci sur des critères de rapidité, d'optimisation d'occupation, d'utilisation de ressources, etc. Le respect des contraintes d'interfaçage autorisera la mise au point et l'évaluation de performance de ces parties par simple substitution sans affecter le reste du système.

A propos de ce cas de figure, les contraintes s'avèrent très strictes, la substitution d'un module par un autre module n'étant possible qu'à la condition que les autres modules n'y accèdent qu'en utilisant son interface. En particulier, un programme appelant un module ne devra en aucune façon exploiter des renseignements sur la réalisation interne du module. Une méthode efficace pour parvenir à ce but a été proposée par Parnas : laisser les programmeurs d'un module dans l'ignorance de la réalisation des autres.

1.2.4 Les composantes d'un S.E.

L'étude des composantes permet de fixer les rôles de chaque couche logicielle et les rapports entre ces couches. Nous allons par la suite distinguer les modules suivants (voir la figure **??**) :

– Le *gestionnaire d'interruptions* récupère les interruptions matérielles et logicielles et applique le traitement approprié qui varie sur la cause de ces interruptions.

– Les *pilotes de périphériques* (*drivers*) gèrent l'échange des données avec les périphériques. Chaque pilote connaît son périphérique et cache son mode d'utilisation aux couches supérieures du système. Ces « drivers » utilisent les interruptions car le dialogue asynchrone entre CPU et unités externes s'effectue au moyen des interruptions. En d'autres termes, le pilote envoie des ordres à son périphérique qui répond au bout d'un temps non défini par le biais d'une interruption.

– Le *système d'E/S* masque les « drivers » de périphériques. Il offre des fonctions d'E/S qui, bien qu'étant de bas niveau, ne distinguent pas de manière explicite la nature du périphérique. Ces E/S sont réalisées à partir (ou vers) des zones de la mémoire appelés des *tampons* (*buffer*). L'allocation de ces tampons passe donc par le gestionnaire de la mémoire centrale.

- La *gestion de la mémoire centrale* répond aux demandes d'allocation et de libération de zones mémoire. Dans une première approche, la mémoire virtuelle peut être vue comme une extension de la mémoire centrale qui est temporairement rangée sur disque. Ce déplacement d'une partie de la mémoire implique :
  - le retour à la demande des informations utiles et non présentes en mémoire centrale (c'est une opération d'E/S) ;
  - la sauvegarde sur disque des informations présentes mais inutilisées.
- Le *système de gestion des fichiers* (SGF) offre toutes les primitives nécessaires à la création, destruction, modification des fichiers se trouvant en mémoire secondaire.
- La *gestion des processus* répartit la ou les CPU entre les tâches qui en ont besoin. Ces tâches consomment de la mémoire et exploitent des fichiers.
- Les *processus utilisateur* (dont les *interpréteurs de commande* sont un exemple particulier) utilisent le S.E. en lui adressant des requêtes en bonne et due forme. Ces requêtes permettent, au choix :
  - de lancer, de figer ou de tuer d'autres processus,
  - d'exploiter ou de modifier des fichiers,
  - d'allouer de la mémoire, etc.

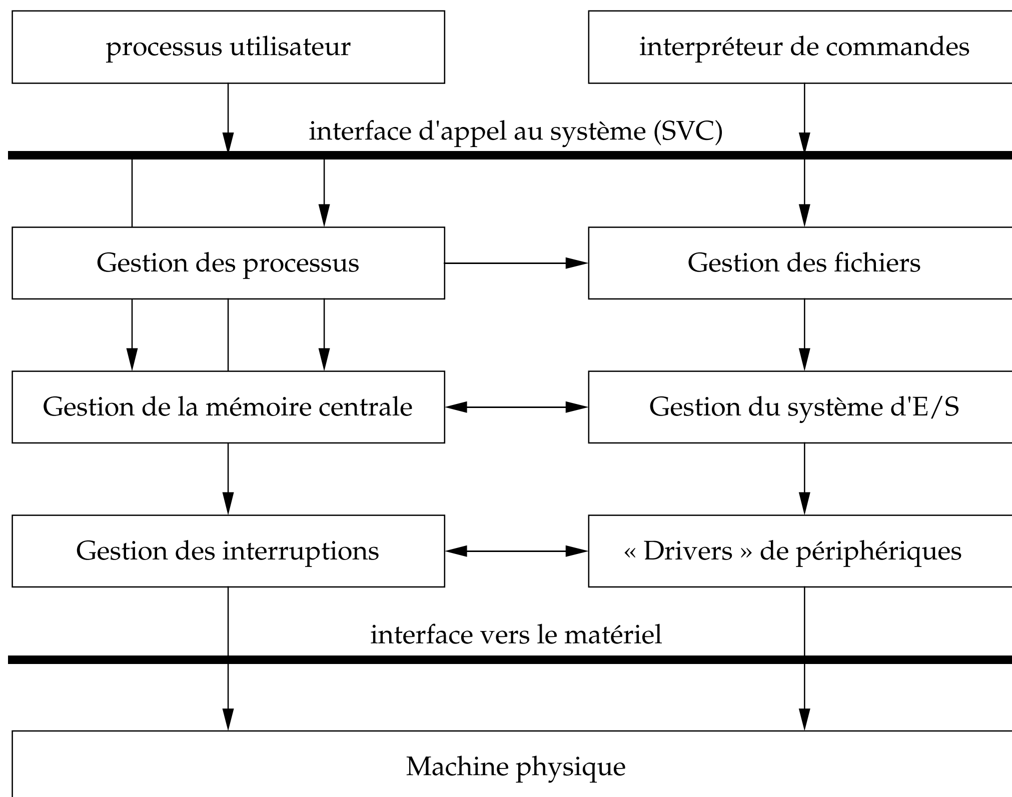


FIG. 1.3 – Les composantes et la structure d'un système

1.3 Historique des systèmes d'exploitation

L'historique est un moyen agréable de présenter les principaux concepts en partant de l'absence de S.E. pour arriver aux systèmes répartis.

1.3.1 Systèmes monoprogrammés

Sur les premiers ordinateurs il n'existe pas de S.E. à proprement parler. L'exploitation de la machine est confiée à tour de rôle aux utilisateurs ; chacun disposant d'une période de temps fixe. C'est une organisation en *porte ouverte*.

Au début des années 50, on voit apparaître le premier programme dont le but est de gérer la machine : c'est le *moniteur d'enchaînement des tâches*. Cet embryon de S.E. a la charge d'enchaîner l'exécution des programmes pour améliorer l'utilisation de l'unité centrale (U.C.). Il assure également des fonctions de protection (vis à vis du programme en cours d'exécution), de limitation de durée et de supervision des entrées/sorties. Pour réaliser ces opérations, le moniteur est toujours présent en mémoire. Il est dit *résident*.

La fin des années 50 marque le début du *traitement par lots* (*batch processing*). Une machine prépare les données en entrée (lecture des cartes perforées à cette époque) tandis que la machine principale effectue le travail et qu'une troisième produit le résultat. Il existe donc un parallélisme des tâches entre lecture, exécution et impression. Les opérations d'E/S ne sont plus réalisées par la CPU, ce qui libère du temps de calcul.

1.3.2 Systèmes multiprogrammés

La *multiprogrammation* arrive au début des années 60. Elle est caractérisée par la présence simultanée en mémoire de plusieurs programmes sans compter le S.E. lui-même. Cette caractéristique s'explique de la manière suivante : l'exécution d'un programme peut être vue comme une suite d'étapes de calcul (les *cycles d'U.C.*) et d'étapes d'E/S (les *cycles d'E/S*) comme le montre la figure **??**. Sur un système monoprogrammé, la CPU est donc inutilisée durant les cycles d'E/S.

L'idée de base est d'utiliser ces temps d'attente pour exécuter un autre programme. Ce programme doit nécessairement être déjà présent en mémoire afin d'éviter l'E/S de chargement puisque justement on cherche à utiliser les temps morts d'E/S. La réalisation pratique de cette idée nécessite :

- des unités matérielles capables d'effectuer des E/S de manière autonome (libérant ainsi la C.P.U pour d'autres tâches) ;
- des possibilités matérielles liées à la protection de la mémoire et/ou à la réimplantation du code pour éviter qu'une erreur d'un programme influence le déroulement d'un autre.

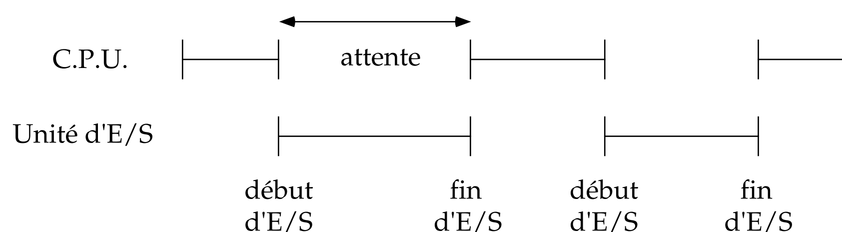


FIG. 1.4 – Cycles de CPU et cycles d'E/S

Dans les années 60/70 les premiers *systèmes en temps partagé* (*time sharing*) sont disponibles. Ces systèmes sont directement liés à l'utilisation interactive des machines au moyen de terminaux vidéo. Ce mode d'utilisation impose un temps de réponse « acceptable » puisque les utilisateurs attendent devant leur terminaux. Pour garantir un bon temps de réponse moyen, le temps d'exécution de la CPU est découpé en tranches appelées des *quanta*. Ces quanta sont allouées aux programmes en cours d'activité. Le temps d'exécution de la CPU est donc **partagé** entre les programmes utilisateurs.

Si le nombre d'utilisateurs n'est pas trop important et, sachant qu'un utilisateur moyen passe 90% de son temps à réfléchir et seulement 10% à exécuter une action, le temps de réponse reste

acceptable et chaque utilisateur à l'impression d'avoir sa propre machine. Sur un plan matériel, le temps partagé est basé sur les possibilités suivantes :

- les programmes sont tous en mémoire ; le temps partagé implique donc la multiprogrammation ;
- le matériel doit permettre l'interruption d'un programme au bout de son quanta de temps pour « passer » la CPU à autre programme ;
- les temps de commutation d'un programme vers un autre doit être aussi faible que possible car durant cette étape la CPU est utilisée par le S.E. au détriment des programmes utilisateurs.

Les *systèmes répartis* se développent durant les années 80. Dans cette organisation, les données mais aussi les programmes sont réparties sur plusieurs machines connectées par un réseau. Les problèmes sont plus complexes puisqu'ils couvrent la communication, la synchronisation et la collaboration entre ces machines, mais ceci est une autre histoire... et un autre cours.

	1950	1960	1970	1980
Gros ordinateurs	pas de logiciels moniteurs compilateurs	traitement par lots temps partagé	multi-utilisateurs	systèmes répartis
Mini ordinateurs		pas de logiciels moniteurs compilateurs	temps partagé	multi-utilisateurs
Les Micros			pas de compilateurs compilateurs moniteurs	multi-utilisateurs et temps partagé

FIG. 1.5 – évolution des systèmes d'exploitation ([SG94])

1.4 Exemples de systèmes d'exploitation

1.4.1 Ordinateur individuel

Toute configuration de base d'un ordinateur individuel comporte une unité centrale et un terminal (écran, clavier et éventuellement « souris »). En général, cet ensemble est augmenté d'une mémoire secondaire (disque dur) et d'une imprimante. L'utilisateur potentiel attend de ce système principalement deux types de services :

- créer et nommer des fichiers ; pouvoir les conserver en mémoire secondaire ; transférer de l'information entre les fichiers et les organes d'entrées-sorties (clavier, imprimante, écran) ;
- exécuter des programmes qui peuvent être livrés avec le système ou créés et introduits sous forme de fichiers ; les données sont introduites au clavier ou lues dans des fichiers ; les résultats sont affichés à l'écran, listés sur l'imprimante ou encore stockés dans des fichiers.

Ce genre de système étant utilisé par un seul usager, la notion de partage de ressources est absente. L'allocation des ressources intervient pour la gestion de la mémoire et de l'espace disque. Pour ce type de système, les qualités essentielles requises sont :

- la fiabilité ;
- l'efficacité (les performances de la machine support étant souvent limitées, il importe de les utiliser au mieux) ;
- la simplicité d'utilisation (Macintosh de Apple) ;
- la facilité d'extension par adjonction de nouveaux programmes utilitaires ou adaptation à des nouveaux périphériques.

Ces deux derniers aspects mettent en évidence l'importance de la conception des interfaces, tant au niveau du langage de commande qu'à celui du système de gestion de fichiers.

1.4.2 Commande de procédés industriels

Imaginons que dans une usine de produits chimiques, un produit C soit synthétisé à partir de deux produits A et B. Le réacteur peut être schématisé comme suit. Le calculateur chargé de conduire le processus de fabrication doit assurer trois fonctions : régulation, enregistrement et sécurité.

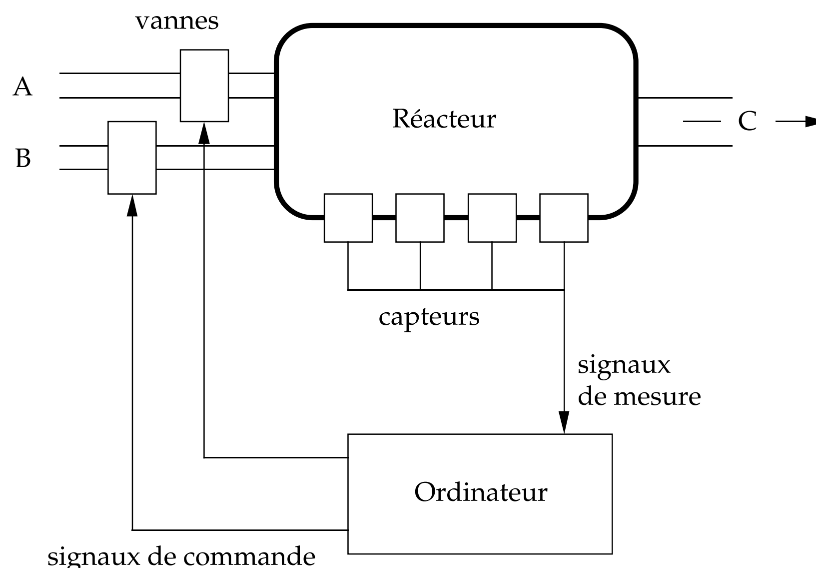


FIG. 1.6 – Conduite d'un réacteur chimique

- **Régulation**. Les divers paramètres de fonctionnement (température, concentration, pression) doivent être maintenues dans des limites fixées pour la bonne marche de la fabrication. Pour cela on doit agir sur les vannes A et B, sur l'alimentation de résistances de chauffage, d'un agitateur, etc. Tous les paramètres sont mesurés à chaque instant par un ensemble de capteurs disposés dans la cuve ; l'ordinateur prélève ces mesures, les interprète et agit en conséquence sur les organes concernés selon un programme de régulation.
- **Enregistrement**. Les mesures effectuées par les divers capteurs sont périodiquement enregistrées ; leur valeur est affichée sur un tableau de bord surveillé par un responsable et stockée dans un fichier (« journal ») en vue d'un traitement ultérieur (statistiques d'exploitation).

– **Sécurité**. Il y a arrêt d'urgence du réacteur si certaines valeurs prélevées dépassent certains seuils pré-définis.

Le mode de fonctionnement précédemment décrit impose des contraintes au système. Le temps nécessaire au traitement d'un ensemble de mesures (prélèvement, enregistrement, détermination et exécution des commandes qui s'imposent) doit être inférieur à la période de prélèvement ; la fonction de priorité doit être prioritaire sur toutes les autres.

Le fait de fixer des limites à la durée d'un traitement informatique, l'existence d'échéances, la notion de traitement prioritaire et la connexion aux organes de commande et de mesure d'un dispositif extérieur sont caractéristiques des applications dites en *temps réel*.

Pour ces systèmes, la qualité principale est la fiabilité. Les conséquences d'une défaillance pouvant être catastrophiques (centrale nucléaire), le système doit être en mesure d'assurer à chaque instant la sécurité du processus qu'il pilote, et en particulier, assurer un service minimal en cas de défaillance du matériel, d'événement accidentel (séisme) ou d'erreur humaine.

1.4.3 Systèmes transactionnels

Ces systèmes sont caractérisés par les propriétés suivantes :

– l'ensemble des informations gérées peut atteindre une taille importante (milliards d'octets) ;

– Sur ces informations peuvent être exécutées un certain nombre d'opérations pré-définies, ou *transactions*, souvent interactives ;

– Le système possède un grand nombre de points d'accès (terminaux) et un grand nombre de transactions peuvent se dérouler en même temps.

Les exemples types d'applications de ce genre sont les systèmes de réservation de places de train ou d'avion, de gestion de comptes bancaires, de consultation ou de documentation. Là encore une des principales qualités requises du système est la fiabilité (intégrité et cohérence des données). Un tel système doit en outre posséder des qualités de disponibilité et des capacités de tolérance aux pannes.

1.4.4 Systèmes en temps partagé

Le rôle d'un système en temps partagé est de fournir ses services à un ensemble d'usagers, chacun bénéficiant de services équivalents à ceux accessibles sur une machine individuelle, mais aussi de services liés à l'existence d'une communauté d'usagers (partage d'information, communication entre usagers). De plus, les coûts étant répartis entre un grand nombre d'usagers, ceux-ci peuvent bénéficier de services qui leur seraient inaccessibles individuellement (périphériques spéciaux ou logiciels nécessitant un grand espace mémoire).

Les problèmes rencontrés sont donc à la fois ceux des ordinateurs individuels et ceux des systèmes transactionnels : définition de la machine virtuelle pour chaque usager ; partage et allocation des ressources physiques communes (processeurs, mémoires, organes de communication) ; gestion de l'information partagée (fichiers) et des communications.

Les qualités attendues sont : disponibilité, fiabilité, sécurité, bonne exploitation des performances du matériel, qualité de l'interface et des services offerts à l'usager, facilité d'extension et d'adaptation.

Chapitre 2

Interruptions, déroutements et appels système

2.1 Exécution de programme

Avant de préciser ce que nous entendons par interruption, il est souhaitable de définir rapidement la notion d'exécution d'un programme sur une machine. En fait, notre objectif est de présenter un modèle général valable sur la plupart des machines.

Une *machine* est composée schématiquement d'un *processeur* (aussi appelée la C.P.U.), d'une *mémoire principale* et d'organes d'E/S. Le processeur est un circuit actif qui comporte des *registres généraux* et des *registres spécialisés*. L'ensemble des registres spécialisés forment le *mot d'état du processeur* (M.E.P. ou P.S.W. pour « *Processor Status Word* »). Parmi ces registres on trouve :

- le compteur ordinal (CO) qui contient l'adresse de la prochaine instruction à exécuter ;
- le mode d'exécution (MODE) qui peut être *maître* ou *esclave* ;
- Le masque d'interruptions que nous détaillerons plus tard.
- Un ou plusieurs pointeur(s) de pile.

  etc.

La notion de mode a été introduite essentiellement pour des raisons de protection, afin qu'un programme « quelconque » ne puisse accéder à des zones ou à des registres propres au S.E. Dans la pratique, la CPU distingue les instructions normales et les *instructions privilégiées*. Ces dernières ne sont utilisables qu'en mode maître. Elles permettent en général la modification des registres spécialisés et le dialogue avec les unités d'E/S.

Partons du principe que le S.E. s'exécute en **mode maître** et que les programmes utilisateur s'exécutent en **mode esclave**. La programmation directe des E/S est donc réservée au S.E. et les E/S des programmes utilisateurs devront passer par des requêtes au S.E. Nous développerons ce point à la section **??**.

Une *exécution* est une évolution **discrète** de l'état de la machine. Cet état est donné par le contenu de la mémoire et la valeur des registres de la CPU. Nous sommes donc capables d'observer l'évolution d'une machine mais seulement sur certains points que nous appellerons les *points observables* ou *points interruptibles*. Ces points sont situés dans le temps **à la fin de l'exécution** d'une instruction de la CPU. Le schéma **??** décrit sommairement l'évolution des registres d'une C.P.U. simplifiée lors de l'exécution d'un programme.

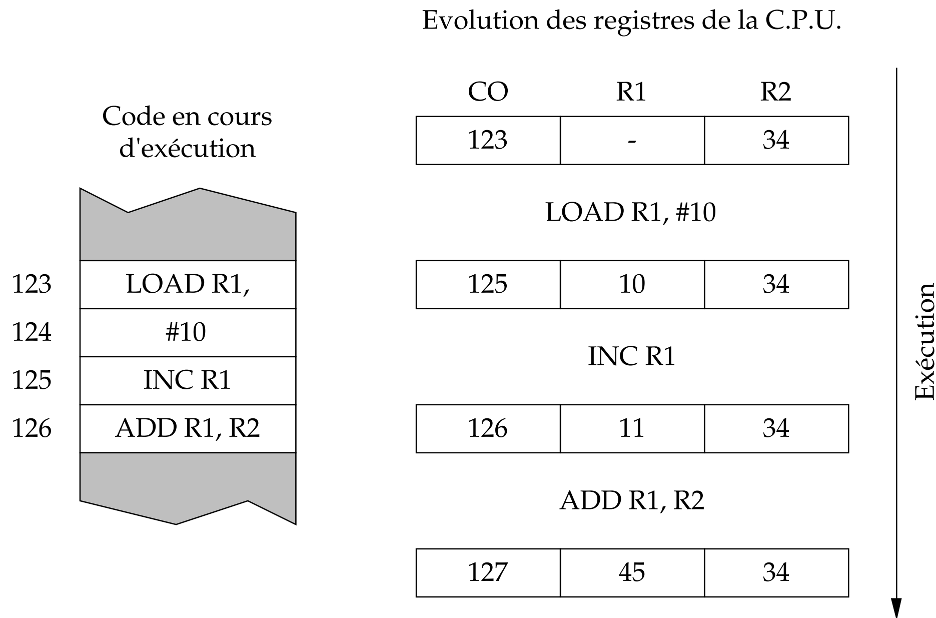


FIG. 2.1 – Un exemple d'exécution

2.2 Le mécanisme des interruptions

Dans tous les types de système, il est toujours nécessaire de considérer un travail courant (le programme en cours d'exécution) et un travail exceptionnel dont le but est de traiter un événement. On peut citer les exemples suivants :

– Dans les systèmes de conduite de processus, certains événements importants (voir même graves) doivent être pris en compte dans les délais les plus brefs. En d'autres termes, il faut donc interrompre le travail courant (relevés des capteurs), pour exécuter un programme prioritaire.

– Il existe toujours un dialogue entre l'U.C. et les organes d'E/S. Notamment, une unité de disque ou une imprimante signalent à l'U.C. que l'E/S est terminée. Dans ce cas également, le travail courant doit être interrompu pour prendre en compte cette nouvelle situation de manière à optimiser l'utilisation des organes d'E/S.

Ces deux exemples ont un point en commun : les événements exceptionnels sont *asynchrones* c'est à dire qu'il n'est pas possible de prévoir leur arrivée. Pour contourner ce problème, les machines disposent d'un mécanisme général permettant de traiter ces événements asynchrones. C'est ce mécanisme complexe qui vous est présenté dans cette section.

Les *interruptions* permettent d'interrompre provisoirement le déroulement d'un programme en cours pour exécuter une routine considérée comme prioritaire. On associe à chaque cause d'interruption un numéro k qui l'identifie. On dispose également dans les adresses basses de la mémoire d'une table appelée le *vecteur d'interruptions* (`vi`). Les cases `vi`$[k]$ de cette table contiennent l'adresse de la routine à exécuter lors d'une interruption de cause k. Cette routine est appelée le *traitant d'interruption* de cause k.

Plus précisément, lors d'une interruption de cause k, la CPU effectue **dès la fin de l'instruction en cours** les actions suivantes :

1. sauvegarder la valeur du compteur ordinal et le mode d'exécution (dans une pile ou dans une case mémoire particulière suivant les C.P.U.) ;

2. passer en mode maître ;

3. forcer dans le compteur ordinal la valeur vi$[k]$, c'est à dire l'adresse de la première instruction de la routine associée à l'interruption de cause k.

L'interruption est donc un **mécanisme matériel** puisque la sauvegarde et l'initialisation du compteur ordinal à partir du vecteur d'interruptions sont des opérations réalisées par la CPU. Le traitant représente la partie logicielle du mécanisme d'interruption. Il a (presque) toujours la structure suivante :

1. Sauvegarder la valeur des registres de la CPU (dans un emplacement particulier de la mémoire). Cette étape est couramment appelée la *sauvegarde du contexte*.
2. Traiter la cause de l'interruption.
3. Restaurer la valeur des registres de la CPU et le mode du programme interrompu. C'est la *restauration du contexte*.
4. Forcer dans le compteur ordinal la valeur préalablement sauvegardée.

De cette description on tire deux conclusions : (1) les traitants d'interruption s'exécutent en mode maître (donc avec des droits étendus) ; (2) l'exécution du programme interrompu n'est pas perturbée par le traitement de l'interruption. L'étape 4 est souvent réalisée au moyen d'une instruction de la CPU qui provoque le retour au programme interrompu (`RTI`). Cette étape est appelée *l'acquittement de l'interruption*. Les principales utilisations du processus d'interruption sont les suivantes :

– Interruption logicielle (ou déroutement) provoquée par la CPU lors de la détection d'une situation « anormale ». Par exemple :
  – appel explicite du S.E.,
  – instruction incorrecte ou inconnue,
  – violation de privilège,
  – dépassement de capacité,
  – division par zéro,
  – tentative d'accès à une zone protégée de la mémoire.
– Interruption matérielle générée par une unité externe à la CPU afin de lui signaler l'apparition d'un événement extérieur. Par exemple :
  – fin d'une E/S,
  – impulsion d'horloge,
  – changement d'état d'un des périphériques,
  – présence d'une valeur intéressante sur un capteur.

Certaines CPU n'ont qu'une seule **cause d'interruption**. Dans ce cas, un « ou » logique de toutes les causes possibles sera effectué et le traitant d'interruption – qui est unique – devra au préalable tester les indicateurs pour connaître la cause.

2.3 Les interruptions matérielles

Les *interruptions matérielles* se présentent comme un ensemble de fils numérotés reliant la CPU et les circuits externes de la machine. La présence d'un signal sur un de ces fils provoque une interruption du programme en cours d'exécution. Le numéro de cette interruption est directement lié au fil qui l'a déclenchée.

En résumé, un circuit extérieur génère une interruption sur la CPU afin de lui signaler un événement. Cette interruption stoppe le programme en cours pour lancer une routine du S.E. L'exécution de cette routine permet, dans les meilleurs délais, la prise en compte par le S.E. de l'événement extérieur.

2.3.1 Système hiérarchisé d'interruptions

Les fils d'interruptions peuvent être *hiérarchisés* c'est-à-dire classés par ordre de priorités respectives. Un traitant d'interruption peut donc être lui-même interrompu par une demande d'interruption intervenant sur un fil de priorité supérieure. Il passe alors à l'état d'attente. La figure **??** représente l'activité des programmes dans le temps pour un système hiérarchisé à 8 niveaux où le niveau 0 est le plus prioritaire, le niveau 7 correspondant au programme d'arrière-plan.

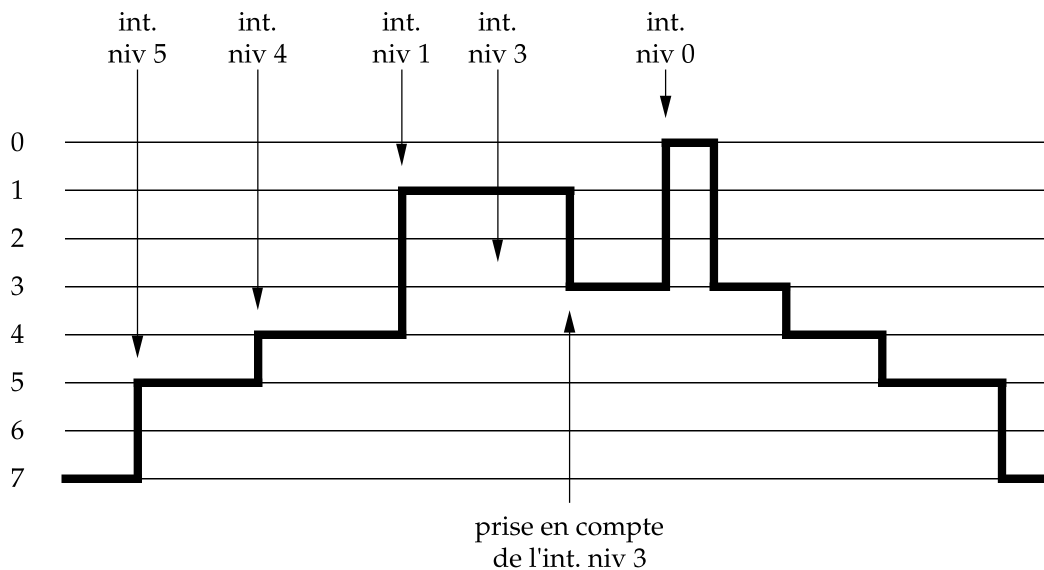


FIG. 2.2 – Effet de la hiérarchisation d'un système d'interruption

Les systèmes d'interruption sont quelquefois plus élaborés et sont constitués d'un type d'organisation très modulaire ayant les caractéristiques suivantes :

- Les interruptions sont groupées en un certain nombre de *niveaux hiérarchisés* (décrits plus haut).
- Un niveau regroupe plusieurs *sous-niveaux* possédant chacun son fil d'interruption et sa priorité à l'intérieur du niveau ; les programmes associés aux sous-niveaux d'un même niveau ne peuvent s'interrompre les uns les autres, leur priorité respective n'intervenant que lors du choix si plusieurs d'entre eux sont en attente simultanément.
- Un sous-niveau regroupe lui-même plusieurs demandes d'interruptions, les causes d'interruption étant recherchées par test d'indicateurs.

2.3.2 Commande du système d'interruption

Chaque niveau d'interruption peut être dans l'un des états suivants :

- *état désarmé* : le niveau n'accepte aucune demande d'interruption.

- *état armé* : le niveau accepte et mémorise une demande d'interruption. On peut *armer* ou *désarmer* un niveau d'interruption par programme en utilisant des *instructions privilégiées*. Cette possibilité est donc **réservée** au S.E.
- *état masqué* : le niveau a été inhibé par programme de sorte que l'interruption a pu être mémorisée mais ne peut être prise en compte par la CPU.
- *état d'attente* : l'interruption peut être prise en compte immédiatement si deux conditions sont remplies :
  - aucun niveau de priorité supérieure n'est en état d'attente ;
  - la CPU se trouve dans une phase interruptible (fin d'instruction).

  Le niveau passe alors à l'état actif.
- *état actif* : il implique la prise en compte de l'interruption par la CPU et dure pendant toute la durée du traitant d'interruption.

Des instructions privilégiées permettent d'armer (ou de désarmer), d'autoriser (ou de masquer), de déclencher un ou plusieurs niveaux d'interruption. Lorsque le nombre de niveaux d'interruption est limité, un registre spécialisé de la CPU contient ce que l'on appelle le *masque d'interruption*. A chaque niveau est associé un bit indiquant s'il est autorisé ou masqué.

2.4 Les appels systèmes

Nous avons vu précédemment que les programmes utilisateur s'exécutent en mode esclave. Les instructions privilégiées permettant la programmation des E/S leur sont donc interdites. Dans ces conditions, toute demande d'E/S et plus généralement toutes les actions demandant des droits étendus, passent par une *requête* en bonne et due forme au S.E.

Cette requête est réalisée par le truchement d'une instruction de la CPU qui provoque une interruption. Nous l'appellerons SVC pour *SuperVisor Call* mais on utilise aussi le terme TRAP. Cette solution, bien que compliquée, a les avantages suivants :

- L'interruption provoque un **branchement** vers le traitant d'interruption mais aussi un **changement de mode**. Il y a donc un passage automatique du programme utilisateur en mode esclave au S.E. en mode maître.
- Il existe un et un seul point d'entrée vers le S.E. pour les processus utilisateur. Il est donc plus facile (du point de vue du concepteur du système) de sécuriser l'appel des primitives système.
- Si on part du principe que le vecteur d'interruptions se trouve dans une zone inaccessible au programme utilisateur, alors ce dernier n'a aucun moyen de passer en mode maître et l'instruction SVC est le seul point de passage.

Généralement, un appel système a la structure ci-dessous. Heureusement, les librairies standards disponibles dans tous les systèmes de développement offrent une interface plus agréable et se chargent de programmer en assembleur l'appel du système. Le choix entre les diverses routines se fait non pas par adressage (comme c'est le cas pour un sous programme) mais au moyen d'un paramètre supplémentaire passé soit dans un registre, soit dans la partie opérande de l'instruction SVC.

⟨préparer les arguments de la requête⟩
⟨préparer le type de la requête⟩
`SVC`
⟨analyser le compte rendu du S.E.⟩

En fait, vu du programme utilisateur, et mise à part la forme de l'instruction d'appel elle même (SVC « supervisor call »), tout semble se passer comme un appel de procédure (empilement de l'adresse de retour, des paramètres...) mais en fait, comme nous venons de le voir, le mécanisme est beaucoup plus complexe.

En résumé, un appel au S.E. permet l'utilisation depuis un programme utilisateur d'un certain nombre de routines système exigeant des droits étendus.

Du coté du S.E., le traitant de l'interruption SVC a la structure suivante :

⟨sauver le contexte du demandeur⟩
⟨vérifier la nature de la requête⟩
⟨vérifier les arguments de la requête⟩
⟨vérifier les droits du demandeur⟩
⟨exécuter la demande⟩
⟨restaurer le contexte du demandeur⟩
⟨retour vers le demandeur⟩

L'ensemble des routines systèmes ainsi offertes à l'utilisateur peut être considéré comme une extension du répertoire des instructions (chaque SVC représentant une macro-instruction) constituant ainsi une nouvelle « machine ».

2.5 Les déroutements

Un *déroutement* est une interruption qui intervient lorsqu'une anomalie a été détectée dans le déroulement d'une instruction, empêchant ainsi son exécution. On distingue trois types de causes :

- données incorrectes (division par zéro, débordement arithmétique, etc.) ;
- tentative de violation d'une protection et/ou d'une interdiction (violation de protection mémoire, utilisation d'une instruction privilégiée en mode esclave, etc.) ;
- impossibilité d'exécution d'une instruction (instruction inconnue ou instruction optionnelle absente de la configuration utilisée, etc.).

Selon la cause d'un déroutement, on peut éventuellement en supprimer l'effet. Ainsi, par exemple, on peut « récupérer » les erreurs arithmétiques ou encore les lectures au delà de la fin de la mémoire. Toutefois, le caractère strictement synchrone des déroutement interdit leur retard de prise en compte comme cela est possible pour les interruptions : en l'occurrence, la notion de masquage ne peut s'appliquer.

En résumé, le mode esclave, les déroutements vers le S.E. en cas d'erreur et le mécanisme des appels système imposent un cadre strict pour l'exécution des programmes utilisateur.

Les systèmes d'exploitation récents sont dits *dirigés par les interruptions* car ils ne s'exécutent que sur demande explicite. Cette demande provenant de l'extérieur (interruption matérielle) ou des programmes en cours d'exécution (déroutement et appel système).

Chapitre 3

Les processus

3.1 Définition

Un *processus* est un programme en cours d'exécution. Il faut d'emblée faire la différence entre un *programme* qui est un fichier inerte regroupant des instructions de la CPU et un processus qui un élément actif. Figeons un processus pour en observer ses composantes. Nous trouvons :

- des *données* (variables globales, pile et tas) stockées dans une zone de la mémoire qui a été allouée au processus ;
- *la valeur des registres* (généraux et spécialisés) de la CPU lors de l'exécution ;
- les *ressources* qui lui ont été allouées par le système d'exploitation (mémoire principale, fichiers ouverts, périphériques utilisés, etc.) ;

L'ensemble de ces composantes forme le *contexte d'exécution* d'un processus ou plus simplement le *contexte*.

3.2 état d'un processus

Un processus n'est pas continuellement en train de s'exécuter. Si la machine comporte n processeurs identiques, à un instant donné il y a au maximum n processus *actifs*. En fait, parmi tous les processus qui sont susceptibles de s'exécuter, seulement un petit nombre s'exécutent réellement. L'allocation de la CPU aux processus qui la réclament est appelée *l'ordonnancement de la CPU*. Elle sera étudiée au chapitre **??**.

L'état opérationnel d'un processus est un moyen de représenter les différentes étapes de ce processus telles qu'elles sont gérées par le système d'exploitation. Le schéma **??** montre les divers états dans lesquels, dans une première approche intuitive, peut se trouver un processus :

- Initialement, un processus est *connu* du système mais l'exécution n'a pas débuté.
- Lorsqu'il est initialisé, il devient *prêt* à être exécuté (1).
- Lors de l'allocation de la CPU à ce processus il devient *actif* (2). Trois cas peuvent alors se présenter :
  - Le processus se *termine* (3).
  - Le processus est *en attente* (5) d'un événement et dès sa réception il redeviendra *prêt* (6).

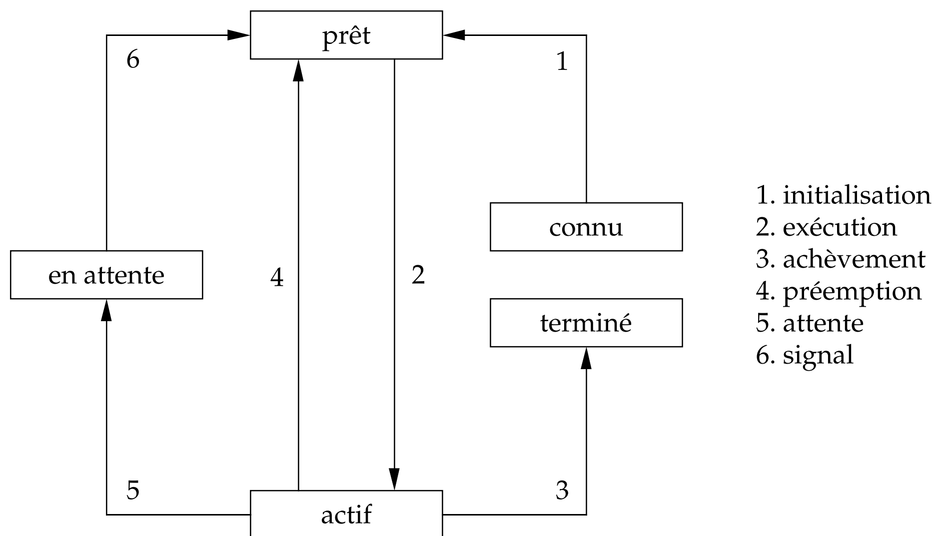


FIG. 3.1 – Schéma simplifié des transitions d'état d'un processus

- Le processus est *suspendu* et se remet dans l'état *prêt* (4). Il y a *réquisition* ou *préemption* de la CPU. Dans ce cas, le S.E. enlève la CPU au processus qui la détient. Ce mécanisme sera vu en détail au chapitre traitant de l'allocation de la CPU.

La notion « d'attente d'un événement » mérite par son importance et sa complexité un petit exemple. Un éditeur de texte enchaîne continuellement la boucle suivante :

répéter
  ⟨lire un caractère⟩
  ⟨traiter ce caractère⟩
jusqu'à . . .

Lorsque le processus éditeur est **actif** il adresse une requête au S.E. pour lui demander une opération d'E/S (la lecture d'un caractère). Deux cas se présentent :

- si il existe un caractère dans le tampon d'entrée, ce dernier est renvoyé par le S.E. ;
- si le tampon d'entrée est vide, le S.E. va « endormir » le processus en changeant son état. Lorsque l'utilisateur frappe une touche du clavier, le S.E. (qui avait préalablement sauvegardé la demande de l'éditeur) « réveille » l'éditeur qui pourra ainsi devenir actif et traiter ce fameux caractère.

  Plus généralement, toutes les opérations lentes (en comparaison de la vitesse de la CPU) provoquent un arrêt momentané du processus demandeur et une reprise ultérieure lorsque l'opération est terminée. C'est notamment le cas pour les opérations d'E/S. Le but de ce mécanisme est de récupérer le temps d'attente pour exécuter un autre processus sur la CPU.

3.3 Représentation d'un processus

Un processus est caractérisé dans le système par :

- un *identificateur* ou *numéro* (par exemple le PID pour *Process IDentification* dans le système UNIX) ;
- un *état opérationnel* (par exemple, un des cinq vus précédemment) ;

- un *contexte* ;
- des *informations* comme les priorités, la date de démarrage, la filiation ;
- des *statistiques* calculées par le S.E. comme le temps d'exécution cumulé, le nombre d'opérations d'E/S, le nombre de défauts de page, etc.

Ces informations sont regroupées dans un *bloc de contrôle de processus* ou PCB (*Process Control Block*). Le système maintient donc un PCB pour chaque processus reconnu. Ce PCB est une mine d'informations. Il représente en fait la principale donnée manipulée par l'allocateur de la CPU.

Lorsqu'un processus quitte l'état **actif**, son PCB est mis à jour et la valeur des registres de la CPU y est sauvegardé. Pour que ce même processus redevienne actif, le S.E. recharge les registres de la CPU à partir des valeurs sauvegardées dans le PCB, il change l'état et finalement il redémarre l'exécution du processus. Nous verrons plus tard dans quelles conditions un processus peut « perdre » la CPU.

Hormis les processus, le système maintient également un certain nombre de files qui regroupent les blocs de contrôle des processus. On trouve ainsi,

- la file des processus prêts,
- la file des processus connus,
- la file des processus en attente qui se décompose en
  - la file des processus qui attendent la disponibilité
    - de la ressource « unité d'E/S 0 »,
    - de la ressource « mémoire »,

      etc...
  - la file des processus qui attendent la fin d'une opération d'E/S,
    - sur l'unité d'E/S 0 (le disque principal),
    - sur l'unité d'E/S 1 (un terminal),

      etc...

  etc...

La figure **??** illustre le chemin suivi par notre éditeur de texte dans les files du système lors de sa demande d'un caractère sur l'unité d'entrée standard.

3.4 Gestion des processus

Les processus sont les principaux éléments actifs du système. Dans ce cadre, il est logique que la création d'un nouveau processus soit demandée par un processus. Il existe donc une filiation entre processus père et le(s) processus fils. Lors du démarrage de la machine, le S.E. lance un processus qui est orphelin puisqu'il n'a pas de père. Ce processus est souvent appelé init. Le premier rôle de ce processus est de lancer des fils qui auront chacun une fonction dans l'organisation générale de la machine. Par exemple,

- un processus pour gérer les E/S asynchrones avec les terminaux,
- un processus pour gérer les connexions au système avec demande et vérification d'un nom d'utilisateur et d'un mot de passe,
- un processus pour gérer l'allocation de la CPU aux processus !

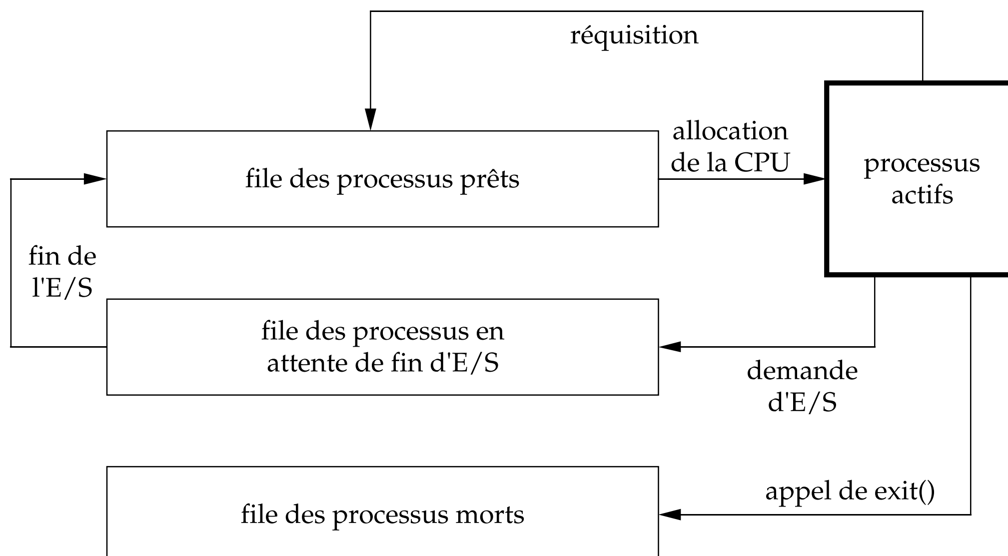


FIG. 3.2 – Déplacements des processus dans les files du système

etc...

Ces processus font partie du système d'exploitation. Ils s'exécutent donc avec des droits étendus. Nous les appellerons les *processus système* ou *démons* (*daemons*) par opposition aux *processus utilisateur*. Le système d'exploitation est donc composé d'un noyau résident qui ne s'exécute que sur demande explicite (interruptions et déroutements) et d'un ensemble de processus système qui ont chacun une fonction précise à assurer. Ce découpage présente deux avantages :

1. la partie résidente du système est réduite en taille ce qui permet d'éviter une trop grande consommation de mémoire par le système ;

2. les processus systèmes ne sont pas forcément toujours prêts ou même toujours présents en mémoire ce qui permet – encore une fois – de réduire la mémoire et le temps CPU consommé par le S.E. au détriment des processus utilisateur.

Si le système est organisé à base de plusieurs processus, des logiciels d'application peuvent également adopter cette structure. Si c'est le cas, il est nécessaire et même vital de fournir des outils permettant une communication et une synchronisation aisée entre les processus d'une même application. C'est l'objet du chapitre « parallélisme et synchronisation ». De plus, cette structuration à base de processus coopératifs est la seule capable d'utiliser facilement une structure matérielle multi-processeurs en associant un processus différent à chaque processeur.

Nous avons parlé de la création d'un processus mais sa disparition est une étape importante ! Elle à lieu sur demande d'un processus étranger (système ou père) ou sur sa propre demande sous la forme d'un suicide. Ce dernier cas correspond à l'appel de la fonction standard `exit()` du langage C.

3.5 Poids lourds et poids légers

Nous avons évoqué plus haut les avantages liés à la structuration des applications sous la forme de processus coopératifs. Mais cette structure comporte également des inconvénients :

- elle implique une communication massive entre les processus ce qui engendre un coût non négligeable de la part du système ;

- elle augmente le nombre de commutations de contexte (c-à-d la sauvegarde et la restauration du contexte d'un processus interrompu) provoquant de ce fait une perte de temps de CPU.

La notion de *thread* et de systèmes *multi-threads* vise à régler ce type de problème. Dans les systèmes *multi-threads* un processus est défini comme un ensemble de threads. Un *thread* (aussi appelé *processus de poids léger* ou *lightweight process* LWP) est un programme en cours d'exécution qui partage son code et ses données avec les autres threads d'un même processus. Bien entendu, les piles sont propres à chaque thread pour éviter que les appels de fonctions et les variables locales ne se mélangent. Cette solution présente plusieurs avantages :

– si un processus ne comporte qu'un seul thread nous revenons au modèle classique ; les systèmes multi-threads sont donc plus généraux ;

– il n'y a plus à mettre en place une communication entre les threads d'un même processus puisqu'ils agissent tous sur les mêmes données ;

– le temps de commutation entre les threads d'un même processus est réduit car le contexte est le même, et seuls les registres de la CPU doivent être sauvegardés ;

– en associant un (ou plusieurs) thread(s) à chaque processeur on peut facilement exploiter une structure multi-processeurs.

Chapitre 4

Allocation de ressources et interblocage

4.1 Allocation de ressources

4.1.1 Notion de ressources

Une *ressource* est un objet utilisable par un processus. Cette utilisation passe par le respect d'un *mode d'emploi* qui précise comment manipuler la ressource. Les ressources sont couramment *libres* ou *allouées*. Pour chaque ressource (ou famille de ressources) il existe un *allocateur* qui a la charge de répondre aux requêtes de *demande*, de *libération* et éventuellement de *réquisition*.

Les ressources peuvent être « *réquisitionnables* » (CPU, mémoire) ou pas (unités de bande), *partageables* (mémoire, disques...) ou pas (imprimantes, unités de bande). Elles peuvent être *physiques* (celles que nous venons de citer, coupleurs...) ou *logicielles* (éditeur, canal). Dans ce dernier cas, c'est donc un programme qui est partagé entre plusieurs processus. La duplication de celui-ci en autant de copies qu'il y a de demandeurs est inconcevable du fait de la perte de taille mémoire que cela impliquerait. Les ressources logicielles dont le code ne change jamais (les données étant établies pour chaque demandeur) sont dites *réentrantes*. Les ressources peuvent également être *banalisées* si on dispose de plusieurs occurrences identiques d'une même ressource.

4.1.2 Objectifs et outils de l'allocation de ressources

Face à tous ces types de ressources, il est souhaitable de définir clairement les objectifs de l'allocation de ressources. Ces objectifs se retrouvent dans la plupart des allocateurs que nous avons ou que nous allons étudier.

Le S.E. doit être *équitable* dans l'allocation de ressources tout en respectant les priorités. En d'autres termes, pour un même niveau de priorité, les demandes doivent être traitées sans favoritisme excessif. La forme la plus simple de l'équité consiste à éviter la *privation* de ressource, c'est à dire l'attente infinie par un processus d'une ressource qu'il n'aura jamais. C'est notamment le cas si il y a interblocage entre plusieurs processus (nous verrons ce cas dans les sections suivantes). Finalement, le S.E. doit également éviter la *congestion* c'est à dire la demande excessive de ressources. En d'autres termes, le S.E. doit veiller à ne pas accepter les demandes quand le système est en surcharge.

Un modèle mathématique des files d'attente peut fournir aux « designers » de système des solutions efficaces au problème d'allocation de ressource. Les paramètres de ce modèle sont :

– la loi de distribution des instants d'arrivée,

– la loi de distribution des demandes de service,

– la politique de gestion de la file d'attente,

– l'absence ou la présence d'un mécanisme de réquisition.

4.2 Les interblocages

Le but principal du système dans un environnement multiprogrammé est le partage des ressources disponibles sur le site entre l'ensemble des processus. Or certaines de ces ressources étant non partageables, un processus possédant une telle ressource aura un contrôle exclusif sur celle-ci. Si l'on généralise cela à plusieurs processus et à plusieurs ressources on voit facilement apparaître les risques d'interblocage. Leur potentialité est liée aux conditions suivantes :

– les ressources sont utilisées en *exclusion mutuelle* c'est à dire par un seul processus à la fois (voir le chapitre sur la synchronisation de processus) ;

– chaque processus utilise simultanément plusieurs ressources qu'il acquiert au fur et à mesure de ses besoins sans nécessairement libérer celles qu'il possède déjà ;

– les ressources ne peuvent être réquisitionnées ;

– il existe un ensemble de processus $(p_0, p_1, \ldots, p_n)$ tel que chaque p_i attend une ressource occupée par p_{i+1} et p_n attend une ressource occupée par p_0.

Après avoir présenté quelques exemples, nous étudierons dans les paragraphes qui suivent quelques méthodes employées pour prévenir, éviter, détecter et guérir les interblocages.

– La *prévention* est basée sur le principe de maintenir à chaque instant le système dans un état tel qu'aucun interblocage ne soit possible. Cette attitude est parfaitement efficace tant que l'on ne considère que l'aspect interblocage, mais en contre partie elle engendre un mauvais rendement d'utilisation des ressources. Néanmoins, ce genre de technique est très largement utilisé.

– Dans *l'évitement* de blocage, le but recherché est de rendre moins strictes les conditions imposées au système, comparativement à la prévention, afin de mieux utiliser les ressources. En fait, dans l'évitement, la possibilité de blocage existe à chaque instant, mais chaque fois que celui-ci s'approche, il est prudemment contourné.

– Les méthodes de *détection* se limitent à déterminer si un interblocage est apparu et si c'est le cas, quels sont les processus et les ressources qui sont impliqués. Ce travail étant fait, l'interblocage peut être traité et supprimé.

– Les méthodes de *guérison* sont utilisées pour « guérir » un interblocage en permettant à certains processus impliqués de terminer leur exécution afin de libérer les ressources qu'ils utilisent. En fait, ces techniques, la plupart du temps, consistent à supprimer un ou plusieurs des processus bloqués. Ceux- ci sont repris ensuite, généralement à partir du début, leur exécution précédente ayant été perdue.

4.2.1 Les embouteillages

Un exemple que l'on rencontre hélas très fréquemment est causé par la bêtise et l'individualisme de certains automobilistes abordant un carrefour important à une heure de pointe. Chacun sûr de sa supériorité sur l'autre, ignore les contraintes considérées surannées (en la circonstance) que sont feux ou priorité et accapare la ressource que constitue le carrefour, préférant faire confiance à son agressivité ou sa soi-disant « débrouillardise » pour imposer sa propre notion de priorité.

Ce sont généralement ces mêmes automobilistes qui seront prêts à tout pour « faire respecter la loi » lorsque d'aventure celle-ci va dans leur sens (unique).

4.2.2 Ressource unique

La plupart des risques d'interblocage dans un système sont dus aux ressources à accès unique. La figure **??** illustre ce type de configuration. Nous y voyons deux processus et deux ressources à accès unique. Une flèche allant d'une ressource à un processus indique que celui-ci détient celle-là ; une flèche allant d'un processus à une ressource signifie que celui-là est demandeur de celle-ci.

Nous avons donc dans le cas présent un interblocage puisque le processus A possède la ressource 1 et désire acquérir en plus la ressource 2 alors que celle-ci est détenue par le processus B qui réclame la ressource 1. Cette configuration bouclée est caractéristique des interblocages.

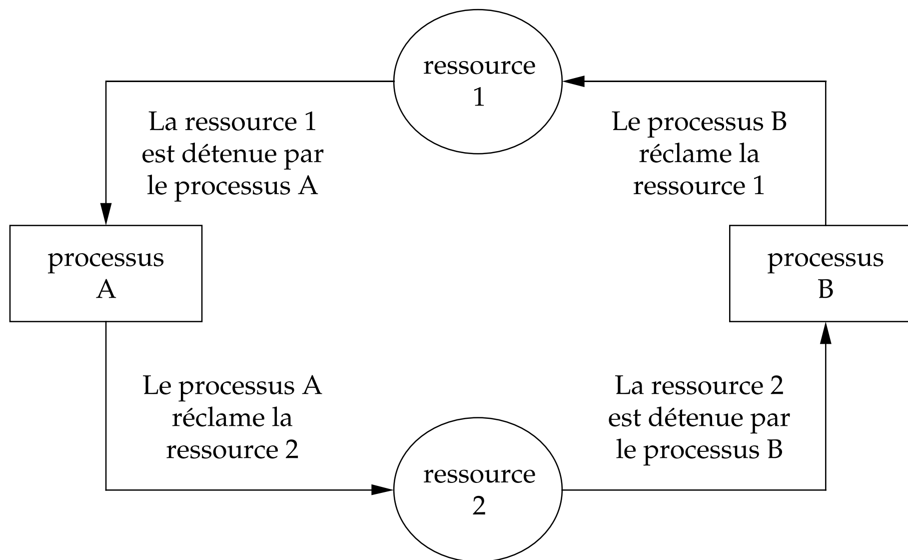


FIG. 4.1 – Interblocage simple

4.2.3 Interblocage dans un système de « spooling »

Rappelons que l'utilité d'un système de spooling est de ne plus assujettir l'exécution d'un programme à la lenteur de certains périphériques tels que l'imprimante. Une sortie sur un tel périphérique sera donc aiguillée vers un support d'accès beaucoup plus rapide (disque magnétique, par exemple) afin de libérer le programme. L'échange effectif avec le périphérique sera effectué ensuite, à partir du fichier « spool » constitué, via une unité d'échange.

Pour reprendre l'exemple de l'imprimante, on ne pourra tolérer qu'un programme qui tourne plusieurs heures au rythme de 100 lignes d'impression toutes les 10 minutes monopolise ce précieux périphérique durant tout ce temps. C'est la raison pour laquelle les fichiers « spool » ne seront imprimés qu'après achèvement des programmes correspondants. Dans ces conditions, le problème qui se pose est celui de la place prévue pour l'ensemble des fichiers « spool ». Le spectre de l'interblocage se dessine peu à peu. Il peut en effet arriver un moment où la zone de « spool » étant saturée, plus aucun processus ne puisse opérer des sorties, mais aucun n'étant achevé, la zone « spool » ne peut se vider !

Que faire à ce moment là ?

- Supprimer un processus (et perdre toute l'exécution) pour récupérer la place qu'il occupait ? Qui nous assure que l'on ne sera pas obligé ensuite d'en supprimer un deuxième puis un troisième ?...
- Commencer à imprimer les sorties d'un des processus ? Lequel choisir ? Monopolisera-t-il l'imprimante longtemps ? Le principe lui-même est-il acceptable ?

– Aurait-on pu prévenir cette situation en interdisant tout accès à un nouveau fichier « spool » (c'est à dire en fait tout nouveau travail) dès que l'occupation avait atteint un certain taux ?

4.2.4 Autre forme de blocage : la famine

Dans tout système où des processus sont en attente pendant que des ressources sont allouées et en particulier le processeur, il est possible que l'activation d'un processus soit indéfiniment retardée alors que les autres sont servis. Cette situation de *famine* est aussi préjudiciable qu'un blocage.

Lorsqu'une ressource est allouée sur la base de priorités, il se peut qu'un processus reste en attente pendant qu'une suite ininterrompue de processus de priorité plus élevée ont la préférence. Or le propre d'un système est d'être à la fois équitable et efficace envers les processus en attente. On verra à la section **??** comment on peut accroître la priorité au fur et à mesure que le temps d'attente augmente. Ce type de technique présenté pour la ressource processeur peut être appliqué pour n'importe quelle ressource.

4.3 La prévention de blocages

C'est assurément la technique la plus utilisée par les « designers » de système. Nous allons voir ici quelques unes des méthodes proposées en considérant leurs effets à la fois sur les utilisateurs et sur le système en particulier du point de vue des performances.

Havender proposa de mettre en défaut les conditions nécessaires d'interblocage vues en **??** en imposant des contraintes aux processus :

– Tout processus doit « annoncer » les ressources qui vont lui être nécessaires et ne « démarrer » que lorsque toutes sont disponibles.

– Si un processus à besoin d'une ressource supplémentaire, il doit libérer celles en sa possession et faire une nouvelle demande incluant la nouvelle.

– Les ressources sont classées par type dans un ordre linéaire auquel devra se soumettre tout processus pour ses demandes d'allocation.

Il est à noter que chacune des stratégies proposées ci-dessus a pour but de mettre en défaut une des conditions nécessaires d'interblocage sauf la première. En effet, nous voulons nous réserver le droit de disposer de ressources *dédiées*.

4.3.1 échec à la condition d'attente

La première stratégie d'Havender impose que toutes les ressources nécessaires à un processus soient libres avant qu'il puisse commencer ; ce sera donc du « tout ou rien ». En aucune façon, les ressources utiles ne pourront être réservées jusqu'à ce que toutes étant libres, l'exécution puisse commencer : elles devront êtres toutes libres simultanément ! La deuxième condition nécessaire est ainsi trivialement mise en défaut ...

... mais à quel prix ! Quel gaspillage de ressources ! Supposons qu'un programme dont l'exécution dure plusieurs heures ait besoin la plupart du temps de un ou deux dérouleurs de bandes sauf pendant un court instant, en fin d'exécution, où dix unités lui sont nécessaires. En appliquant cette stratégie, le processus devra monopoliser les dix dérouleurs pendant toute son exécution. De plus, il devra attendre qu'ils soient tous libres avant de pouvoir être initialisé, ce qui risque d'être long ! Nous nous trouvons devant un risque flagrant de famine.

Une solution utilisée pour remédier à ce gros défaut consiste à opérer par étapes lorsque, comme dans l'exemple que nous venons de voir, le programme s'y prête. Dans ces conditions, l'allocation

des ressources se fera elle aussi par étape, ce qui réduit considérablement la sous-utilisation des ressources mais engendre un coût d'exploitation plus élevé.

4.3.2 échec à la condition de non-réquisition

Supposons que le système autorise un processus à conserver les ressources qu'il détient alors qu'il opère une nouvelle demande. Tant que les ressources supplémentaires demandées sont libres, le blocage n'apparaît pas. Mais si nous arrivons dans un schéma montré dans la figure **??**, nous sommes en situation d'interblocage.

Havender préconise en pareil cas d'imposer à un processus demandeur de libérer les ressources qu'il détient pour ensuite les redemander en y ajoutant la nouvelle. Cette stratégie met en échec la troisième condition nécessaire Mais là encore, à quel prix ? Lors de la libération obligatoire des ressources détenues, tout un travail peut être perdu (bonjour les performances système) ! Si cela se produit peu souvent, c'est tolérable, mais si c'est fréquent ce peut être catastrophique : en particulier des travaux prioritaires et/ou à échéance risquent de voir leur statut sérieusement remis en cause, sans parler des risques évidents de famine.

4.3.3 échec à la condition d'attente circulaire

C'est le but recherché par la troisième stratégie proposée par Havender. Chaque type de ressource ayant un numéro, tout processus ne pourra effectuer ses requêtes que par ordre croissant dans ces types. Cette stratégie a été implantée dans de nombreux systèmes, mais non sans difficulté.

- Les diverses ressources étant requises au moyen de leur numéro d'ordre, l'ajout d'une nouvelle ressource sur un site nécessite la modification de tous les programmes. La portabilité est nulle.
- Le numéro d'ordre alloué aux diverses ressources doit refléter l'ordre d'utilisation de la plupart des programmes susceptibles d'être exécutés sur le site. Si d'aventure un programme ne respecte pas cet ordre « canonique », les ressources doivent être acquises éventuellement longtemps avant leur utilisation effective. D'où un gaspillage.
- Les systèmes tendent de plus en plus aujourd'hui à respecter la contrainte de convivialité. Le moins que l'on puisse dire est que cette stratégie ne répond pas à cette attente.

4.4 Évitement d'interblocage (l'algorithme des banquiers)

Dans un système où les risques d'interblocage existent, il est toujours possible de l'éviter en prenant les précautions nécessaires à chaque allocation de ressource. La technique la plus connue est sans doute l'algorithme des banquiers de Dijkstra ainsi nommée à cause de la grande prudence de ceux-ci en matière de prêts : « On ne se lâche pas des pieds sans se tenir des mains ! »

Partons du principe que le S.E. connaît parfaitement l'état de l'allocation de ressources aux processus. Plus précisément, les données suivantes sont considérées comme disponibles :

- $\mathrm{dispo}[i]$ nombre de ressources R_i disponibles sur le système,
- $\max[i, j]$ nombre maximum de ressources R_i utilisables par le processus P_j,
- $\mathrm{alloc}[i, j]$ nombre de ressources R_i couramment allouées au processus P_j (par définition nous avons donc $\mathrm{alloc}[i, j] \leq \max[i, j]$).

Un processus P_j peut s'exécuter si et seulement si, pour toute ressource R_i, nous avons

$$\max[i,j] - \text{alloc}[i,j] \leq \text{dispo}[i].$$

Un ordre d'exécution $P_1, P_2, \ldots, P_n$ est dit *sain* si et seulement si les processus $P_1, P_2, \ldots, P_n$ peuvent s'exécuter jusqu'à leur terme les **uns après les autres** dans cet **ordre**. Bien entendu l'exécution d'un processus P_k implique la **libération** par P_k des ressources qu'il a utilisées. Un système est dit *sain* si il existe un ordre d'exécution sain des processus. Si un système est sain, alors il ne peut pas y avoir d'interblocages. Par contre, si le système n'est pas sain, un interblocage peut apparaître mais ce n'est pas une obligation.

> *En résumé, l'apparition d'un état non sain n'implique pas pour autant qu'il y aura inévitablement un interblocage. La seule chose que cela implique est qu'une séquence défavorable d'événements peut conduire à un interblocage.*

En se basant sur cette propriété, le S.E. face à une demande d'allocation de la ressource R_i au processus P_j, applique l'algorithme suivant :

1. les annonces sont elles respectées (c-à-d $\text{alloc}[i,j] < \max[i,j]$) ?
2. si j'alloue R_i à P_j l'état obtenu est-il sain ?
3. dans l'affirmative je réalise l'allocation ; sinon je suspends le processus P_j qui ne peut donc pas continuer son exécution.

Le principe de l'algorithme des banquiers est de refuser toute requête ayant pour effet de mettre le système dans un état non sain. En résumé :

- les conditions d'exclusion mutuelle, d'attente et de non réquisition sont autorisées ;
- les processus doivent annoncer leurs besoins en ressources ;
- ils peuvent conserver les ressources en leur possession tout en réclamant des ressources supplémentaires ;
- il n'y aura pas de réquisition ;
- afin d'aider le système, les ressources seront demandées une à une ;
- si une requête n'est pas honorée, le processus demandeur conserve néanmoins les ressources en sa possession et attend un temps fini jusqu'à ce qu'il obtienne satisfaction ;
- seules les requêtes laissant le système dans un état sain sont honorées. Dans l'éventualité contraire, le processus demandeur devra attendre (le système étant toujours dans un état sain, toutes les requêtes pourront être satisfaites tôt ou tard).

Cet algorithme semble donc intéressant et à tout le moins plus convivial et d'un meilleur rendement que les stratégies de prévention proposées par Havender. Toutefois, nous allons voir qu'il contient de nombreuses faiblesses qui font que les concepteurs de systèmes lui préfèrent d'autres approches.

- L'algorithme présuppose et s'appuie totalement sur le fait que le nombre de ressources est invariant. Or il est bien évident que des problèmes de maintenance du matériel ou tout simplement des pannes peuvent mettre en défaut ce postulat.
- Il présuppose aussi que chaque utilisateur annonce le maximum de ressources utilisées. Or à l'heure actuelle, la convivialité grandissante des systèmes fait que rares sont les utilisateurs connaissant précisément les ressources dont ils ont besoin.

– L'algorithme garantit que les requêtes pourront être satisfaites ... dans un temps fini (!) Voilà qui est rassurant mais guère suffisant !

– Réciproquement, il impose aux processus de restituer les ressources ... au bout d'un temps fini. Là encore, on pourrait s'attendre à un peu plus d'exigence !

4.5 Détection des interblocages

Les algorithmes de détection sont utilisés dans des systèmes où les trois premières conditions nécessaires sont autorisées et ont pour but de déterminer s'il y a attente circulaire. L'utilisation de ces algorithmes entraîne un coût d'exploitation non négligeable.

4.5.1 Graphes d'allocation de ressource

L'utilisation de graphes orientés, représentant les allocations et requêtes de ressources, facilite la détection des blocages. Dans les schémas qui vont suivre, les carrés représentent des processus et les cercles des classes de ressources identiques. Les petits cercles contenus dans ces derniers représentent le nombre de ressources de chaque classe. La figure **??** montre les relations pouvant être représentées dans un graphe d'allocation et de requête des ressources. Ces graphes sont modifiés au cours du temps à chaque nouvelle allocation ou libération de ressource sur le site. S'il advient qu'une ressource d'un type donné soit hors service (pour une cause quelconque), cela se traduira dans le graphe par la suppression d'un petit cercle dans le grand cercle correspondant au type en question.

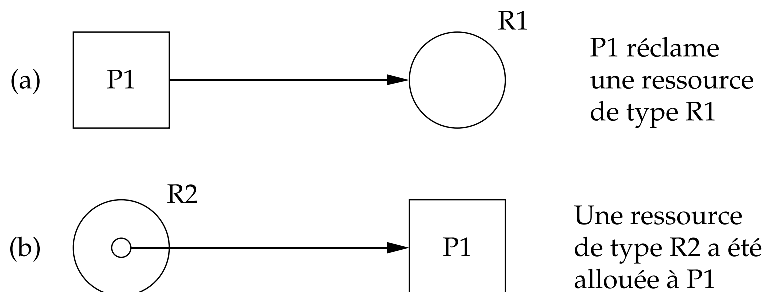


Fig. 4.2 – Graphe de requête et d'allocation de ressource

Afin de déterminer s'il est en situation de blocage, le système devra procéder à une réduction du graphe.

4.5.2 Réduction d'un graphe d'allocation de ressource

Si les requêtes d'un processus peuvent être satisfaites, on dit que le graphe est réduit par ce processus (cela signifie que l'on considère le graphe comme si le processus s'était achevé, libérant ainsi les ressources qu'il détenait). Cela se traduit par la suppression des flèches provenant de ressources et aboutissant à ce processus et de celles partant de ce processus vers d'autres ressources.

Si un graphe peut être réduit par tous ses processus, il n y a pas de blocage. Dans le cas contraire, les processus irréductibles constituent l'ensemble des processus en interblocage.

La figure **??** montre les diverses étapes de réduction d'un graphe permettant d'aboutir à la conclusion qu'il n'y a pas d'interblocage.

L'ordre dans lequel les réductions se font est sans importance : le résultat sera toujours le même.

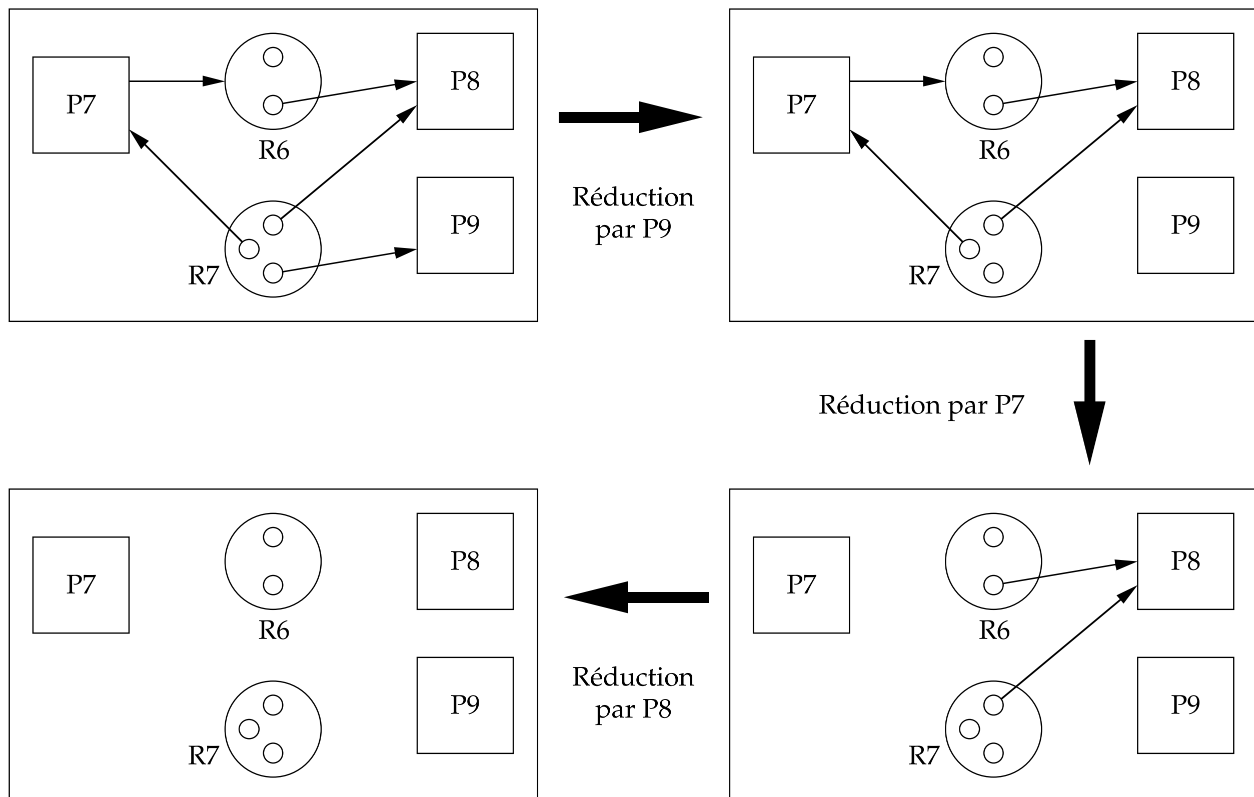


FIG. 4.3 – Réductions d'un graphe d'allocation

4.6 Guérison d'interblocage

Une fois que le système a déterminé qu'il y avait interblocage, il doit le guérir en supprimant une ou plusieurs des conditions nécessaires. Habituellement, un ou même plusieurs processus perdront pour ce faire tout ou partie du travail déjà accompli. Mais mieux vaut cela que le maintien de l'interblocage. La guérison est rendue difficile pour plusieurs raisons :

- Tout d'abord, nous venons de le voir, la détection de l'interblocage n'est pas chose aisée, et le système peut ne pas s'en apercevoir tout de suite.
- La plupart des systèmes n'ont guère de facilité pour suspendre indéfiniment un processus, l'enlever du système et le reprendre plus tard. En particulier, cela est hors de question pour les processus temps réels.
- Même si ces possibilités existaient, elles entraîneraient un coût d'exploitation prohibitif et nécessiteraient les compétences d'un opérateur attentif, ce qui n'est pas toujours possible !...
- La guérison d'un interblocage de proportions modestes peut être opérée avec un coût raisonnable ; mais si l'on est en présence d'un interblocage de grande envergure (faisant intervenir plusieurs dizaines ou même centaines de processus), la quantité de travail sera énorme.

Comme nous l'avons dit, la guérison passe forcément par la destruction d'un processus afin de récupérer les ressources qu'il possédait pour permettre aux autres processus de s'achever. Quelquefois la destruction de plusieurs processus s'impose pour récupérer un nombre suffisant de ressources. Aussi le terme de guérison semble ici un peu exagéré, mais s'adapte parfaitement à la conception occidentale de la médecine qui s'attache surtout à la symptomatique des pathologies. (On peut faire la comparaison avec une amputation d'un membre atteint d'artérite : le patient s'estime-t-il guéri ?)

L'ordre dans lequel les processus vont être supprimés est très important. Va-t-on chercher à minimiser leur nombre ? Va-t-on considérer la quantité de travail déjà accomplie afin de réduire la

perte de rendement ? Va-t-on considérer les priorités des processus ? Va-t-on choisir les processus victimes parmi ceux pour lesquels le retrait des ressources n est pas fatal à 1 exécution (afin de seulement les suspendre, le temps de « guérir », pour ensuite les reprendre en l'état, sans perte de travail) ?

Et le temps d'exploitation de tout cela ? Et si cela arrive sur un site sur lequel se déroule une application temps réel très délicate (surveillance de raffinerie, de centrale atomique...). Voilà autant de questions qui donnent à réfléchir sur la conception des systèmes de demain et qui, loin d'être résolues de façon satisfaisante, occasionnent quelques « nuits blanches » au concepteurs d'aujourd'hui.

4.7 Et demain ?

Nous venons d'avoir un aperçu de ce qui se faisait (ou pourrait se faire) aujourd'hui. En fait, ce sont les méthodes de prévention carrées, brutales, mais efficaces et sans risque qui sont le plus souvent employées, le blocage étant encore tout à fait occasionnel.

Dans les systèmes futurs, les interblocages devront être traités de façon systématique et efficace pour plusieurs raisons :

– Les systèmes s'orienteront de plus en plus vers des opérations parallèles asynchrones, abandonnant les schémas séquentiels. Les bancs de processeurs seront monnaie courante, autorisant un parallélisme énorme.

– L'allocation des ressources sera dynamique. La convivialité grandissante des systèmes fera que l'on utilisera ce que l'on voudra quand on le voudra (sans même le savoir !).

– De plus en plus, les données seront assimilées à des ressources. En conséquence, la quantité des ressources que devra gérer un système atteindra une taille gigantesque.

On peut donc imaginer que ce problème tout en devenant de plus en plus important et de plus en plus difficile à traiter trouvera néanmoins dans les technologies et les structures futures des solutions efficaces.

Chapitre 5

Allocation du processeur

5.1 Introduction

Le partage d'une machine entre plusieurs utilisateurs s'est très rapidement révélé nécessaire pour des raisons d'économie, de rentabilité et de convivialité. Sous cette hypothèse, le problème qui se pose alors, à chaque instant, pour chaque processeur, est de décider s'il doit poursuivre ou interrompre l'exécution du processus courant, et, dans le second cas, de déterminer le prochain processus à activer.

La règle utilisée pour effectuer ce choix est contenue dans l'algorithme d'ordonnancement, plus couramment appelé *ordonnanceur* (*scheduler*). Nous allons présenter ici quelques paramètres intervenant dans l'élaboration des divers algorithmes utilisés, en les justifiant par l'idée directrice qui a motivé leur emploi. Il faut en fait considérer l'ordonnanceur sous trois aspects :

– Au niveau le plus élevé, le plus proche de l'utilisateur, sa fonction consiste à déterminer si un travail soumis doit être admis (tout travail admis devient un processus) ou pas. En d'autres termes, ce niveau a pour mission d'allouer (ou réquisitionner) des *machines virtuelles* aux divers utilisateurs du système selon certaines règles induites par la gestion spécifique du site considéré. Ainsi, l'évaluation des priorités, la gestion des ressources nécessaires (règlement de conflits), le maintien de la charge du système en dessous du seuil d'écroulement et de son intégrité, en particulier en cas d'incidents imprévisibles, sont autant de tâches dont l'ordonnanceur devra s'acquitter avec inévitablement des répercussions sur les travaux soumis.

– Le rôle du niveau moyen est de déterminer l'ensemble des processus pouvant obtenir le contrôle. Autrement dit, il tient à jour les paramètres relatifs aux différents processus qui permettront de dégager ceux étant susceptibles de devenir actifs.

– Le niveau le plus bas, le plus proche du matériel, choisit parmi les processus prêts, en respectant les priorités, celui à qui le processeur va être alloué. C'est le *répartiteur* qui est toujours résident en mémoire.

Il faut bien comprendre que le fait d'allouer une ressource à un processus favorise celui-ci (au moins de façon temporaire). Ainsi, chaque rouage de l'ordonnanceur a pour effet d'appliquer une certaine politique envers les divers processus, et par conséquent, envers les utilisateurs. Or, c'est justement cette politique qui sera appelée a être éventuellement modifiée afin de satisfaire le plus grand nombre d'usagers. C'est la raison pour laquelle, ces différents points de choix devront être séparés du côté purement logique de l'ordonnanceur.

Sans donner plus de détails sur la façon d'évaluer la priorité d'un processus (ce sera l'objet du paragraphe suivant), le programme simplifié de l'ordonnanceur pourrait s'écrire :

```
pour toujours
  p := prioritaire ;
  tant que (état(p) ≠ prêt) faire
    p = suivant(p) ;
  fin faire
  restaurer_contexte(p) ;
  relancer(p) ;
fin-pour
```

Dans ce programme, on a supposé la liste ordonnée par priorité décroissante à partir de l'entrée « prioritaire », et de plus bouclée.

5.2 Les stratégies d'ordonnancement de la CPU

5.2.1 Les objectifs

Une politique d'ordonnancement doit :

1. être **équitable** : cette contrainte est satisfaite si tous les processus sont considérés de la même manière et qu'aucun n'est retardé indéfiniment ;
2. rendre le **débit maximum** : elle doit faire en sorte de satisfaire le plus grand nombre de demandes par unité de temps ;
3. pouvoir prendre en charge un **maximum d'utilisateurs interactifs** tout en assurant des temps de réponse acceptables ;
4. être **prédictible** : un même processus doit pouvoir s'exécuter dans un temps à peu près équivalent quelle que soit la charge du système ;
5. être **la moins coûteuse possible** afin de ne pas éprouver les performances générales du système en particulier dans les phases instables ;
6. avoir pour effet de **rationaliser la gestion des ressources** en :
   - recherchant une utilisation optimum ;
   - favorisant les tâches peu exigeantes en nombre et en qualité de ressources ;
   - évitant la famine (par exemple en augmentant la priorité au fur et à mesure que l'attente s'accroît).
7. mettre en œuvre des **priorités** fondées sur des critères pertinents ;
8. avoir la possibilité de **réajuster ces priorités**, soit de manière globale (nécessité de modularité), soit de manière ponctuelle au cours du temps (priorités dynamiques) ;
9. **favoriser** les processus ayant un **comportement souhaitable** ;
10. veiller à **ne pas accepter de nouveaux travaux** lorsque le système est en surcharge ;

    etc.

La liste des contraintes que nous venons d'énoncer est loin d'être exhaustive, mais suffit déjà à mettre en évidence les conflits (3/10, 4/8, 1/9,...). Ceci dénote donc une grande complexité dans la détermination des tâches éligibles, ce qui est justement en complète opposition avec le point 5.

Ceci explique en partie la raison pour laquelle les ordonnanceurs ne seront souvent qu'un ensemble de compromis très satisfaisants dans certains cas de figures, mais s'avérant moyennement, voire fortement critiquables dans d'autres circonstances. Ceci explique aussi pourquoi nous ne pourront pas vous présenter ensuite L'Ordonnanceur avec un grand O, mais une panoplie de réalisations dont chacune pourra s'avérer satisfaisante sur un site donné mais intolérable sur un autre.

5.2.2 Critères à considérer

Afin de pouvoir réaliser tout (ou plutôt partie) des objectifs présentés au paragraphe précédent, le mécanisme d'ordonnancement doit considérer :

1. **le taux des entrées/sorties de chaque processus** : après que le processeur lui ait été alloué, ne l'utilise-t-il qu'un temps très court avant de réclamer un échange ?
2. **le taux d'utilisation du processeur** : pour chaque processus, lorsque le processeur lui est alloué, l'utilise-t-il pendant toute la tranche de temps impartie ?
3. **fonctionnement interactif** ou **traitement par lots** : les utilisateurs interactifs émettent généralement des requêtes simples qui doivent être satisfaites très rapidement, alors que les utilisateurs « batch », n'étant pas présents peuvent subir des délais (ceux-ci devant toutefois rester dans des limites raisonnables) ;
4. **degré d'urgence** : un processus « batch » ne requiert pas de réponse immédiate alors qu'un processus « temps réel » nécessite des réponses très rapides ;
5. **priorité des processus** : les processus de forte priorité doivent bénéficier d'un meilleur traitement que ceux de priorité plus faible ;
6. **taux de réquisition** : lorsqu'un processus est de faible priorité par rapport aux autres, il en découle un fort taux de réquisition. Dans ces conditions, le système doit-il essayer de l'avantager ou au contraire attendre que les priorités redeviennent comparables afin d'éviter les temps de commutation effectués en pure perte, vu la forte probabilité de réquisition.
7. **temps cumulé d'allocation du processeur** : doit-on pénaliser un processus ayant bénéficié d'un temps d'exécution important ou au contraire le favoriser car on est en droit de penser qu'il va bientôt s'achever ? Cette question revêt une importance particulière lorsque le système est en surcharge ;
8. **temps d'exécution restant** : le temps d'attente moyen peut être réduit en exécutant de préférence les processus réclamant un temps CPU minimum pour s'achever. hélas, l'évaluation de ce temps restant est rarement possible.

5.2.3 Réquisition ou pas ?

On dit qu'un ordonnanceur n'opère *pas de réquisition* si dès lors qu'il a alloué la CPU à un processus, il lui est *impossible de le lui retirer*. Réciproquement, une politique d'ordonnancement autorise la réquisition si le contrôle peut être retiré à tout moment au processus actif.

Cette possibilité est absolument et trivialement nécessaire sur les sites supportant des applications « temps réel ». C'est la contrainte liée aux temps de réponse qui rend cette technique indispensable dans les systèmes interactifs. Mais la réquisition engendre un surcoût non négligeable à l'exploitation :

coût en temps occasionné par les changements de contexte incessants, coût en espace engendré par la nécessité de partager la mémoire entre tous les processus.

La non-réquisition est gênante pour les travaux courts lorsqu'ils doivent attendre qu'un travail très long s'achève ; mais globalement la philosophie semble plus équitable, les temps de réponse sont mieux prévisibles car l'arrivée de travaux à forte priorité ne vient pas perturber l'ordre des travaux en attente.

La réalisation d'un ordonnanceur à réquisition est, nous l'avons déjà dit, très délicate. En particulier, le calcul des priorités ne doit pas voir l'aspect sophistiqué l'emporter sur le côté signifiant. « Rester simple est le maître-mot, mais si cela n'est pas possible, il faut au moins insister pour demeurer effectif et pertinent dans les choix ! »

5.2.4 Intervalle de temps et interruption d'horloge

Le système dispose d'un moyen très simple pour retirer le contrôle à un processus. Un simple décompteur d'impulsions d'horloge, dont le calibrage peut être modifié, peut déclencher une interruption prioritaire qui aura pour conséquence d'appeler un traitant d'interruption du système d'exploitation.

Un processus utilisera donc le processeur jusqu'à ce qu'il le libère volontairement, ou qu'il y ait interruption d'horloge ou tout autre type d'interruption réclamant une intervention du système. Le système reprenant le contrôle pourra alors le passer à qui bon lui semble.

L'interruption d'horloge aide à garantir des temps de réponse acceptables dans un système interactif, évite au système de rester monopolisé dans une boucle de programme et permet en outre de traiter des applications temps réels. C'est donc une technique simple, efficace et polyvalente qui toutefois demande une attention particulière pour le calibrage du décompteur.

5.2.5 Calibrage de la tranche de temps

La détermination de la tranche de temps ou *quantum* est critique dans un système. Doit-elle être longue ou courte ? Doit-elle être fixe ou variable ? Doit-elle être la même pour tous les utilisateurs ou déterminée séparément pour chacun d'eux ?

Aux conditions limites, selon que l'on fait tendre le quantum vers l'infini ou vers zéro, un processus s'achèvera sans que l'ordonnanceur soit intervenu ou au contraire, tout le temps CPU sera utilisé par l'ordonnanceur lui-même et aucun processus ne pourra se dérouler.

Afin d'ajuster le quantum à une valeur optimale, il nous faut considérer la courbe du temps de réponse moyen (figure **??**). Supposons qu'on ait le moyen de faire varier le quantum grâce à un curseur. Lorsque celui-ci est à zéro, l'ordonnanceur étant la seule tâche active, le temps de réponse est infini. Dès que nous tournons légèrement le curseur, augmentant ainsi la durée du quantum, le temps de réponse commence à diminuer. Si nous continuons, nous allons arriver à une position telle que si nous tournons encore légèrement le curseur, le temps de réponse va commencer à augmenter. Nous aurons atteint la valeur optimale. Si nous tournons le curseur « à fond », le temps de réponse va décroître pour le processus actif à ce moment là, puisqu'il va se terminer sans être interrompu, mais va considérablement augmenter pour tous les autres. Le temps de réponse moyen se stabilisera à un niveau médiocre fonction du nombre moyen de tâches en attente et du temps d'exécution moyen d'une tâche. On sera arrivé à l'algorithme FIFO que nous verront dans le paragraphe suivant.

Considérons donc la valeur optimale que nous avons obtenue précédemment. Elle représente en fait une petite fraction de seconde. Mais cette valeur est-elle vraiment bien adaptée à chacun des types de tâche. Est-elle assez grande pour que la majeure partie des requêtes interactives puissent être traitées en un seul quantum ? En particulier, si nous pouvons avoir une distribution moyenne dans le temps des demandes d'échange émises par m processus interactif, il serait avantageux que la valeur du quantum soit supérieure à l'intervalle entre deux échanges car cela diminuerait d'autant le

nombre de réquisitions inutiles (il vaut mieux qu'un processus n'utilise pas tout son quantum et cède le contrôle à cause d'une demande d'échange, plutôt qu'il soit interrompu, attende « son tour », récupère le contrôle pour aussitôt lancer un échange).

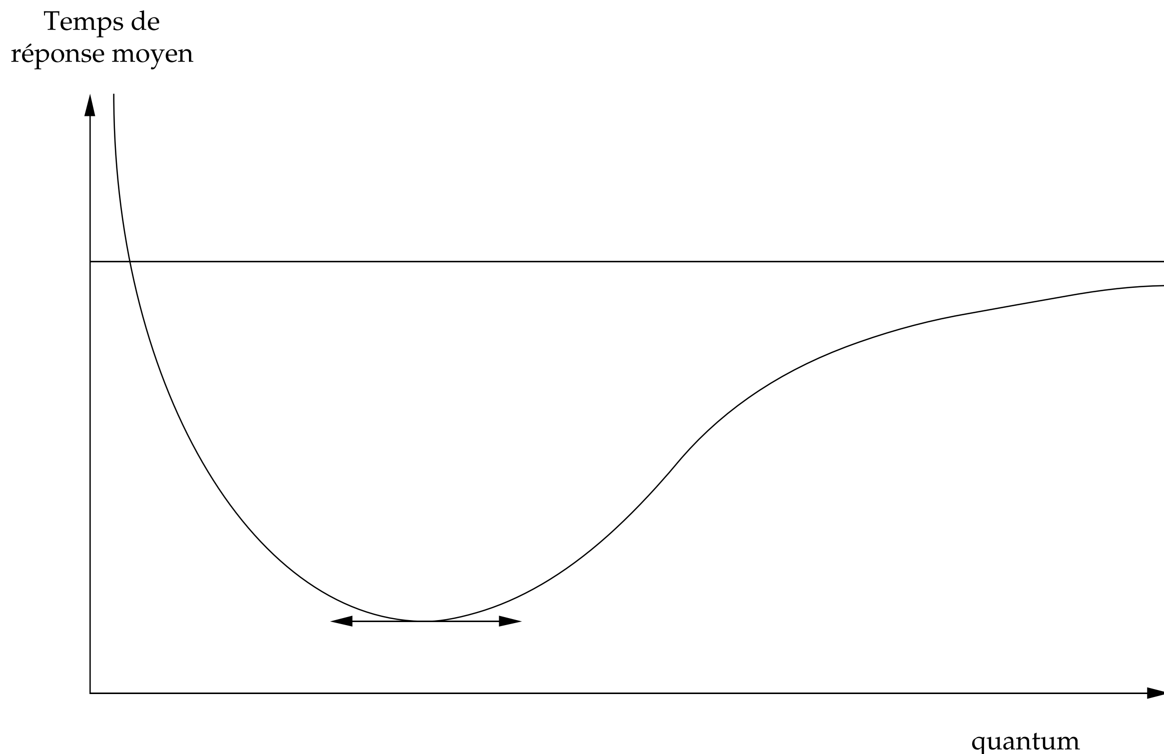


Fig. 5.1 – Influence du quantum sur le temps de réponse moyen

En fait, on s'aperçoit que la valeur de ce quantum va varier d'un système à un autre, mais aussi en fonction du taux de charge. On vient de montrer qu'il peut aussi dépendre du processus.

5.2.6 Priorités

Les priorités peuvent être allouées automatiquement par le système ou de manière externe. Elles peuvent être méritées ou acquises. Elles peuvent être statiques ou dynamiques. Elles peuvent être allouées de façon rationnelle ou arbitraire, en particulier lorsque le système est contraint de faire une distinction entre plusieurs processus sans avoir les moyens d'être sûr de faire le bon choix.

5.2.6.1 Priorités statiques et priorités dynamiques

Par définition, les *priorités statiques* ne changeant pas, elles sont d'une mise en œuvre facile et engendrent un faible coût d'exploitation. Toutefois, elles sont insensibles aux changements survenus dans l'environnement, changements qui justement peuvent nécessiter un ajustement des priorités.

A l'inverse, les *priorités dynamiques* peuvent changer en fonction de la modification de l'environnement. En particulier, nous verrons que la priorité initiale peut être réajustée très rapidement afin d'être mieux adaptée au type du processus considéré.

Il est bien évident que la gestion des priorités dynamiques est beaucoup plus complexe et engendre un coût beaucoup plus grand que celle des priorités statiques. En contre partie, leur emploi permet d'accroître considérablement le débit et la souplesse du système.

5.2.6.2 Priorités acquises

Un système doit offrir un service équitable et raisonnable (ou plutôt raisonnablement équitable ...) à la majorité des utilisateurs d'un site. Mais il doit aussi pouvoir accepter qu'un usager bénéficie d'un traitement particulier. Celui-ci ayant, par exemple, un travail particulièrement urgent, peut désirer « payer un supplément de service » pour *acquérir une priorité* plus forte afin que son programme soit exécuté plus rapidement. Ce supplément se justifie sous deux aspect :

– tout d'abord, ce n'est que justice car vu sa priorité accrue, les ressources qu'il va utiliser (en particulier le processeur) seront enlevées à d'autres utilisateurs plus souvent que s'il avait conservé sa priorité normale. (Il faudrait toutefois s'assurer que le coût pour les autres usagers sera diminué en conséquence !...) ;

– l'autre aspect laisse supposer que les idées libérales se sont infiltrées jusque là car en effet, « si on ne faisait pas payer, tout le monde réclamerait un meilleur service », ce qui est bien évidemment inconcevable !...

5.3 Algorithmes d'ordonnancement

Nous allons, en fonction des problèmes que nous venons d'exposer et des solutions partielles qui ont été envisagées, présenter ici quelques réalisations d'ordonnanceurs en montrant pour chacune d'elles les avantages et les inconvénients envers le système et envers les utilisateurs.

5.3.1 Ordonnancement par échéance

Certains travaux peuvent être soumis accompagnés d'une date d'échéance. Là encore, cette option va entraîner un supplément d'autant plus important que l'échéance est proche de l'instant de lancement, à condition, bien sûr, que cette échéance soit respectée. Par contre, si ce n'est pas le cas, le service « supplémentaire » pourra être gratuit. Ce type d'ordonnancement est complexe pour plusieurs raisons :

– le système doit respecter l'échéance sans que cela implique pour autant une sévère dégradation de performances pour les autres utilisateurs ;

– le système doit planifier parfaitement l'utilisation des ressources toujours à cause de cette échéance fatidique. Or cela est particulièrement difficile car de nouveaux travaux peuvent arriver et émettre des demandes imprévisibles ;

– afin de limiter les ennuis du point précédent, l'utilisateur réclamant une échéance doit fournir au lancement la liste exhaustive des ressources qu'il utilisera, ce qui n'est pas évident, certaines étant « transparentes » pour lui (tampons d'E/S, canaux, etc.) ;

– si plusieurs travaux à échéance sont lancés en même temps, l'ordonnancement va devenir tellement complexe que des méthodes d'optimisation très sophistiquées vont être nécessaires, d'où un coût d'exploitation très lourd ;

– ce surcroît de temps CPU utilisé par l'ordonnanceur ajouté aux faveurs accordées au(x) processus à échéance va inévitablement pénaliser les autres utilisateurs, ce qui risque d'engendrer des conflits de personnes. Ce facteurs doit être considéré avec une grande attention par les concepteurs des systèmes d'exploitation.

5.3.2 Premier arrivé, premier servi (FIFO)

La technique « FIFO » est assurément la plus simple et de ce fait n'engendre qu'un très faible coût propre. C'est une technique sans réquisition (tout travail commencé se poursuit jusqu'à achèvement). L'ordre de priorité correspond de façon naturelle à l'ordre d'arrivée.

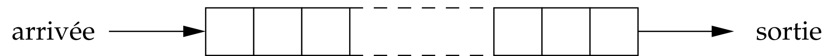


FIG. 5.2 – L'ordonnanceur FIFO

De ce point de vue, il est donc équitable, mais il faut toutefois regretter le fait que les longs travaux occasionnent une longue attente des travaux brefs qui suivent et, réciproquement, une multitude de petits travaux peut provoquer une longue attente de travaux importants.

FIFO présente une faible variance ; il est donc plus prédictible que la plupart des autres techniques. De toute évidence, il n'est pas utilisable dans les systèmes interactifs car il ne garantit pas un bon temps de réponse. C'est en particulier une des principales raisons qui font que cette technique est rarement utilisée aujourd'hui « à l'état brut ». Néanmoins, il faut noter qu'elle peut être associée à une technique plus sophistiquée qui accordera des priorités générales, les processus de même priorité étant considérés selon un schéma FIFO.

5.3.3 Tourniquet (Round Robbin)

L'ordonnancement de type *tourniquet* s'inspire de la technique FIFO, l'association d'une tranche de temps autorisant la réquisition. Les processus, au fur et à mesure qu'ils obtiennent le statut « prêt », sont rangés dans une file. A chacune de ses interventions, le distributeur alloue le contrôle au processus en tête de file. Si le temps d'exécution qui lui est ainsi imparti expire avant son achèvement, il est placé en queue de file et le contrôle est donné au processus suivant.

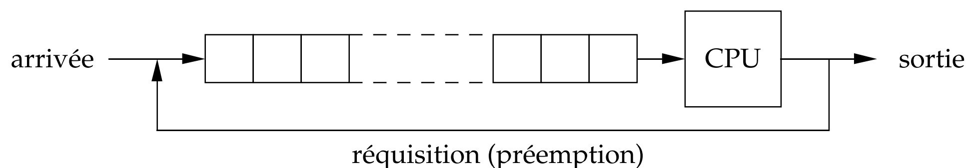


FIG. 5.3 – Ordonnancement par « tourniquet »

Cette technique est satisfaisante dans les systèmes « temps partagé » où les utilisateurs interactifs doivent bénéficier de temps de réponse corrects. Le coût de la réquisition peut être maintenu faible si les mécanismes de commutation sont efficaces et la mémoire suffisante pour contenir plusieurs processus simultanément.

A noter aussi la nécessité d'un réglage judicieux du quantum pour accroître le taux d'utilisation du processeur et donc diminuer les temps de réponse (voir **??**).

5.3.4 Travail plus court d'abord (SJF)

La technique *Plus court d'abord* (SJF pour « *Shortest Job First* ») est encore un schéma d'ordonnancement sans réquisition (donc inutilisable en temps partagé) où le processus ayant le plus faible temps estimé d'exécution jusqu'à achèvement est prioritaire. C'est donc une technique qui a été créée pour pallier partiellement à l'inconvénient de FIFO qui autorisait l'exécution de travaux très longs avant des travaux de plus faible importance, seul leur ordre d'arrivée étant pris en compte.

SJF favorise donc les travaux brefs au détriment des plus importants. De ce fait il entraîne une variance beaucoup plus grande que FIFO, en particulier pour ce qui concerne les travaux longs. SJF

fonctionne de façon à ce que la prochaine exécution puisse s'achever (et donc quitter le système) dès que possible. Cette technique tend donc à réduire le nombre de travaux en attente, ce qui a pour conséquence de diminuer la moyenne des temps d'attente des processus.

Le principal inconvénient de SJF est qu'il requiert une connaissance précise du temps d'exécution, valeur qu'il n'est habituellement pas possible de déterminer. Le seul moyen est de se fier à une estimation donnée par les utilisateurs eux-mêmes. Cette estimation peut être bonne dans des environnements de production où les mêmes travaux sont soumis régulièrement, mais elle s'avère rarement possible dans les environnements de développement.

La connaissance de ce schéma d'ordonnancement pourrait tenter certains de sous-estimer volontairement le temps d'exécution afin de profiter d'une priorité indue. Afin d'éviter ce genre de « malhonnêteté », l'utilisateur est prévenu à l'avance que son travail sera abandonné en cas de dépassement. Cela présente deux inconvénients :

- obligation pour les usagers de majorer les estimations ;
- mauvaise rentabilité du processeur (le temps consacré aux travaux abandonnés faisant rapidement baisser le rendement).

Une seconde possibilité est donc offerte : poursuivre l'exécution du travail durant le temps estimé augmenté, si nécessaire, d'un certain pourcentage (faible en général) puis de le « mettre de côté » dans l'état où il se trouve pour reprendre son exécution plus tard. Bien entendu l'utilisateur sera pénalisé par cette attente mais aussi par un supplément de facturation.

Une troisième possibilité est de ne pas « mettre de côté » le travail, mais de poursuivre son exécution jusqu'à achèvement en facturant le temps excédent à un taux beaucoup plus élevé. Cette solution est finalement mieux acceptée car le supplément correspond effectivement à un meilleur service.

5.3.5 Plus court temps restant (SRT)

La stratégie SRT pour « *Shortest Remaining Time* » est la version avec réquisition de SJF (donc utilisable en temps partagé) où, là encore, priorité est donnée au processus dont le temps d'exécution restant est le plus faible (en considérant à chaque instant les nouveaux arrivants).

Dans SRT, un processus actif peut donc être interrompu au profit d'un nouveau processus ayant un temps d'exécution estimé plus court que le temps nécessaire à l'achèvement du premier. Là encore, et plus particulièrement du fait de la réquisition, le « designer » doit prévoir une dissuasion à l'égard des « malins » connaissant la stratégie d'ordonnancement.

Le coût de SRT est supérieur à celui de SJF : il doit tenir compte du temps déjà alloué aux processus en cours, effectuer les commutations à chaque arrivée d'un travail court qui sera exécuté immédiatement avant de reprendre le processus interrompu (à moins qu'un travail encore plus court ne survienne). Les travaux longs subissent une attente moyenne plus longue et une variance plus grande que dans SJF.

En théorie SRT devrait offrir un temps d'attente minimum, mais du fait de son coût d'exploitation propre, il se peut que dans certaines situations, SJF soit plus performant. Afin de réduire ce coût, on peut envisager plusieurs raffinements évitant la réquisition dans des cas limites :

- supposons que le processus en cours soit presque achevé et qu'un travail avec un temps d'exécution estimé très faible arrive. *Doit-il y avoir réquisition ?* On peut dans ces cas de figure garantir à un processus en cours dont le temps d'exécution restant est inférieur à un seuil qu'il soit achevé quelles que soient les arrivées ;
- autre exemple : le processus actif a un temps d'exécution restant légèrement supérieur au temps estimé d'un travail arrivant. Ici encore, si SRT est appliqué « au pied de la lettre », il

y a réquisition. Mais si le coût de cette réquisition est supérieur à la différence entre les deux temps estimés, cette décision devient absurde !

La conclusion de tout cela est que les « designers » de systèmes doivent évaluer avec beaucoup de précautions les coûts engendrés par des mécanismes sophistiqués car ils peuvent dans bien des cas aller à l'encontre du but recherché : le gain de temps.

5.3.6 Plus grand rapport ensuite (HRN)

En 1971, Brinch Hanssen propose la stratégie HRN (pour « *Highest Response Ratio Next* »). Elle corrige certains travers de SJF et en particulier le favoritisme excessif dont bénéficient les nouveaux travaux courts.

HRN peut être considéré avec ou sans réquisition. La priorité de chaque travail est fonction non seulement du temps de service, mais aussi du temps d'attente. Les priorités sont donc dynamiques et calculées par la formule :

$$\text{priorité} = \frac{\text{temps d'attente} + \text{temps de service}}{\text{temps de service}}$$

en choisissant de préférence les travaux courts si le niveau de priorité est identique. Ce système présente plusieurs avantages :

- Les travaux longs, bien qu'étant défavorisés, voient leur priorité augmenter au fur et à mesure de leur attente. Ils sont donc sûr de récupérer la CPU au bout d'un temps d'attente fini, ce qui élimine le risque de privation.
- Si on utilise HRN avec un mécanisme de réquisition de la CPU, les processus qui restent en sommeil un certain temps (après une demande d'E/S par exemple) voient leur priorité augmenter. Cette augmentation permet de leur allouer plus de temps CPU lors du réveil, ce qui est assez logique.

5.3.7 Tourniquet multi-niveaux

Nous avons vu les problèmes que posait dans SJF et SRT la difficulté de connaître à l'avance la quantité de temps CPU nécessaire à l'exécution d'un programme. Un travail fonctionnant essentiellement en entrée/sortie n'utilisera en fait la CPU que de courts instants. A l'opposé, un travail réclamant le contrôle en permanence monopolisera la CPU durant des heures si l'on suppose un schéma sans réquisition.

En fait, nous l'avons vu plus haut (**??** et **??**), un ordonnanceur doit :

- favoriser les travaux courts ;
- favoriser les travaux effectuant de nombreuses E/S (pour une bonne utilisation des unités externes) ;
- déterminer le plus rapidement possible la nature de chaque travail afin de le traiter en conséquence.

Le *tourniquet multi-niveaux* répond à ces attentes. Un nouveau processus est stocké en queue de la file de plus haut niveau. Il progresse dans cette file FIFO jusqu'à ce qu'il obtienne le contrôle. Si le processus s'achève ou libère la CPU pour une entrée/sortie ou une attente d'événement, il est sorti de la file d'attente. Si le quantum expire avant, le processus est placé en queue de la file de niveau inférieur. Il deviendra à nouveau actif lorsqu'il parviendra en tête de cette file et à condition que celles de niveau supérieur soit vides. Ainsi, à chaque fois que le processus épuisera sa tranche

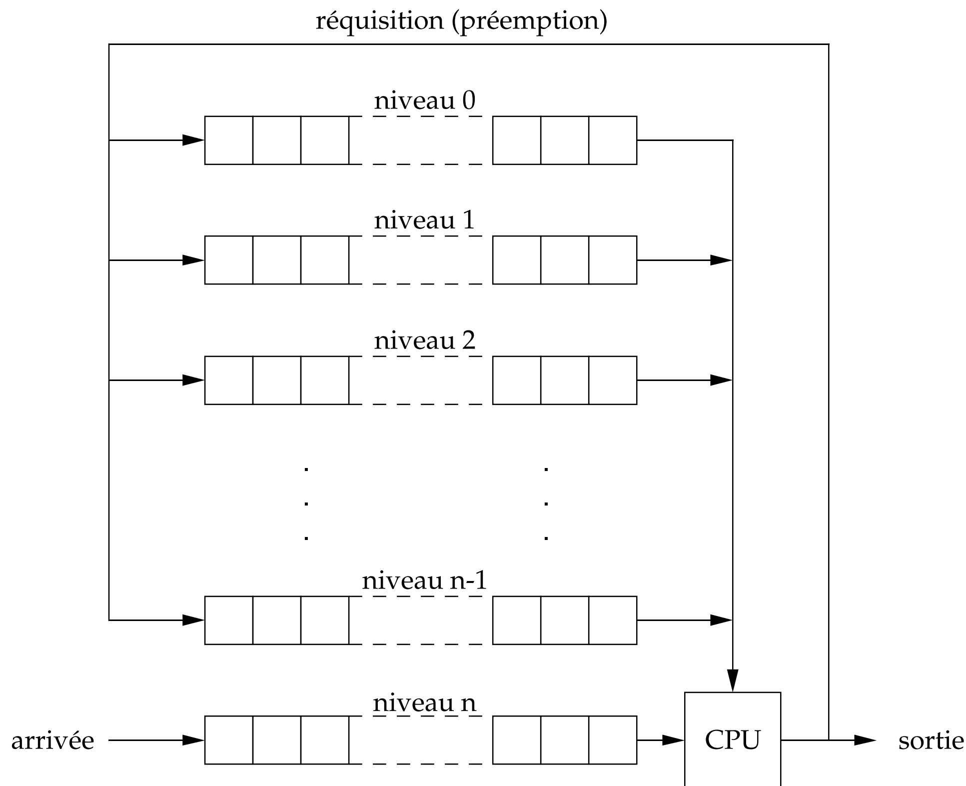


FIG. 5.4 – Schéma de principe du tourniquet multi-niveaux

de temps il passera en queue de la file de niveau inférieur à celle où il se trouvait jusqu'à ce qu'il atteigne la file de plus bas niveau.

Dans ce schéma d'ordonnancement, la taille du quantum s'accroît au fur et à mesure que l'on descend dans les niveaux de file. En conséquence, plus un travail est long, plus le temps CPU dont il bénéficie est grand. Mais en contre-partie, le processeur lui sera alloué plus rarement puisque les processus des files supérieures ont une plus grande priorité. Un processus en tête de quelque file que ce soit ne pourra devenir actif que si les files de niveau supérieur (si elles existent) sont vides. Il y aura réquisition dès qu'un travail arrivera dans la file de plus haut niveau. Considérons à présent la façon dont ce mécanisme s'adapte aux différents types de travaux.

Il favorisera les utilisateurs interactifs dont chaque requête sera envoyée dans la file prioritaire et satisfaite avant l'épuisement du quantum. De même, les travaux travaillant essentiellement en entrée/sortie seront avantagés si l'on suppose que le quantum de la file prioritaire est assez grand pour qu'une demande d'échange survienne avant qu'il expire. Dans ces conditions, dès que la demande d'échange se produit, le processus demandeur est sorti de la file prioritaire et y reviendra lorsque cet échange sera effectué (bénéficiant ainsi entre chaque demande de la priorité accordée à la file de plus haut niveau).

En ce qui concerne un travail tendant à monopoliser la CPU, il débutera comme tous les autres dans la file la plus prioritaire, puis, très vite, il descendra les niveaux pour arriver dans la file la moins prioritaire, son quantum expirant à chaque étage. Là, il restera jusqu'à ce qu'il soit achevé mais... dans les hypothèses actuelles, que se passe-t-il si d'aventure un processus de ce type demande un échange ? Il est sorti de la file et lorsque l'échange aura été réalisé, il reviendra... dans la file prioritaire... à moins que le système retienne la file dont il était issu afin de l'y replacer ensuite.

Ce faisant, cette technique présuppose que le comportement passé d'un processus est une bonne indication de son comportement futur. Mais alors, un processus qui après une longue phase de calcul entre dans une phase où les échanges prédominent est désavantagé ! Ceci peut encore être résolu en associant au processus le dernier temps passé dans le réseau de files ou en convenant que tout

processus montera d'un niveau dans le réseau chaque fois qu'il aura volontairement libéré la CPU avant expiration du quantum.

Le tourniquet multi-niveaux est un très bon exemple de *mécanisme adaptatif*. Bien sûr, le coût d'un tel ordonnanceur est supérieur à un qui n'a pas ces facultés d'adaptation, mais la meilleure adéquation de l'attitude du système vis à vis des différents types de travaux justifie amplement cette dépense.

A noter une variante de ce système consistant à maintenir un processus plusieurs tours dans une même file avant qu'il passe au niveau inférieur. Habituellement ce nombre de tours s'accroît (comme la taille du quantum) en descendant dans les niveaux.

Chapitre 6

Allocation de la mémoire centrale

6.1 Concepts de base

6.1.1 Mémoire logique

La notion de ressource logique conduit à séparer les problèmes d'utilisation d'une ressource particulière des problèmes d'allocation de cette ressource. Pour un processus, la *mémoire logique* est le support de l'ensemble des informations potentiellement accessibles, c'est à dire, l'ensemble des emplacements dont l'adresse peut être engendrée par le processeur lors de l'exécution de ce processus.

L'allocation de mémoire consiste à concrétiser cette mémoire logique par des supports physiques d'information tels que *mémoire principale*, *disques magnétiques*, etc. En « bout de chaîne », l'accès d'un processus à une information se traduit par l'accès d'un processeur physique à un emplacement de mémoire principale adressable par ce processeur. L'information accessible à un processus est définie par :

– l'ensemble des informations désignables dans son programme (*objets*) ;

– l'ensemble des informations de désignation (*noms*) ;

– la mise en correspondance noms/objets.

Dans un programme écrit en langage évolué, noms et objets sont définis par ce langage. Ils sont différents de ceux que manipule le processeur physique. Le programme doit donc subir une série de transformations appelée *liaison*. Celle-ci comporte une étape de *traduction* (mise en correspondance des objets avec les emplacements mémoire et des noms avec les adresses relatives correspondantes), une étape d'*édition de lien* (liaison entre programmes traduits séparément), et enfin une étape de *chargement* (fixation définitive des adresses, jusque là définies à une translation près).

La séparation conceptuelle des problèmes de désignation et liaison, d'une part, et des problèmes d'allocation mémoire, d'autre part, peut être schématisée par la figure **??**.

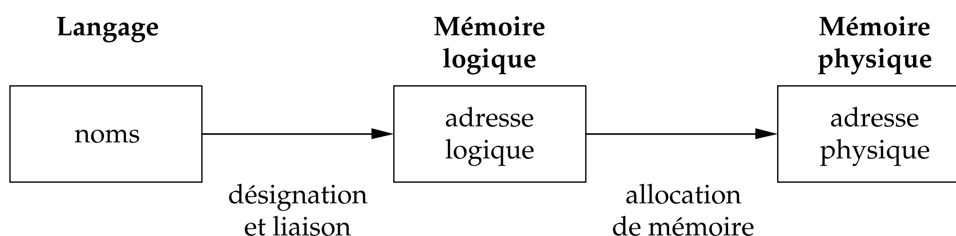


Fig. 6.1 – Transformation des adresses

Le fait que la notion de mémoire logique ne soit pas restée un outil conceptuel, mais ait été mise en œuvre sur certaines machines par des dispositifs physiques de transformation d'adresse a conduit à ce que la séparation des fonctions soit plus ou moins bien respectée.

- *Mémoire logique contiguë* : Elle est constituée d'une suite d'emplacements identiques (*mots*) organisés de manière séquentielle et désignés par des entiers consécutifs appelés *adresses logiques*. Un objet est une information occupant un mot ou plusieurs mots consécutifs ; il est désigné par l'adresse logique du premier mot. Cette organisation est donc identique à celle des emplacements d'une mémoire physique.
- *Mémoire logique non contiguë* ou *segmentée* : Elle est constituée d'un ensemble de segments. Un *segment* est une suite de mots et regroupe généralement des informations de même nature. Il peut avoir une taille variable. Les mots contenus dans un segment sont désignés par des entiers consécutifs appelés *déplacements*. L'adresse logique d'un mot est donc un couple (numéro de segment, déplacement dans le segment) appelé *adresse segmentée*. Un objet, qui peut occuper un segment entier ou une suite de mots consécutifs dans un segment, est désigné par l'adresse segmentée de son premier mot.
- *Mémoire physique non contiguë*. Le placement des mémoires logiques en mémoire physique peut être contiguë ou pas. Dans ce dernier cas, les pages qui composent la mémoire logique sont disséminées dans différentes pages physiques. C'est une organisation en *mémoire paginée* ou *segmentée et paginée*.

6.1.2 Allocation de Mémoire

On distingue différentes manières de réaliser la mise en correspondance entre organisation de la mémoire logique et implantation de cette mémoire logique en mémoire physique :

- *Réimplantation dynamique* : la correspondance logique/physique est variable dans le temps. De ce fait, l'allocation de la mémoire physique se fait par zones (de taille variable) et/ou par pages (de taille fixe).
- *Correspondance fixe* (aussi appelée *implantation statique*). La correspondance est établie une fois pour toutes au moment de la compilation. C'est le cas dans les *systèmes à partition unique* ou *fixes*.
- *Correspondance dynamique* (aussi appelée *réimplantation dynamique*). La correspondance logique/physique peut aussi être dynamique et ce de deux manières.
  - Elle peut être fixée au moment du chargement du processus et donc varier entre deux exécutions. La mémoire est allouée sous forme de zone contiguës appelés des *partitions*, c'est le *système des partitions variables*.
  - Elle peut également varier durant l'exécution du processus. Les objets sont donc déplacés à l'intérieur la mémoire centrale. Bien entendu, ces déplacements, opérés par le système, doivent être « transparents » pour le processus. C'est le cas dans les *systèmes paginés* ou *segmentés* qui allouent la mémoire par *pages* (de taille fixe) ou *segments* (de taille variable).

L'allocation de mémoire doit permettre à un processus l'accès à un objet défini en mémoire logique, en amenant en temps voulu cet objet en mémoire principale (la seule directement adressable). Une politique d'allocation mémoire doit donc apporter une solution aux deux problèmes suivants :

1. réaliser la correspondance entre adresses logiques et adresses physiques ;

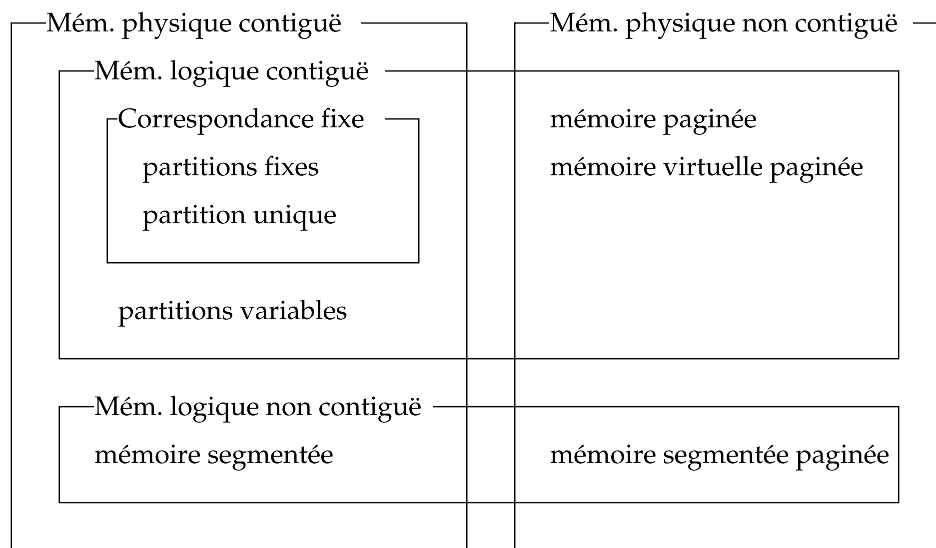


FIG. 6.2 – Les différentes organisations de la mémoire

2. réaliser la gestion de la mémoire physique (allocation des emplacements, transfert de l'information).

Lorsque les informations appartiennent à plusieurs utilisateurs, deux contraintes supplémentaires apparaissent :

3. réaliser le partage d'information entre ces utilisateurs ;
4. assurer la protection mutuelle d'informations appartenant à des usagers distincts.

Une politique d'allocation de mémoire idéale aurait pour effet d'assurer qu'à tout instant l'information nécessaire à l'exécution de l'instruction en cours soit immédiatement accessible au processeur, donc se trouve en mémoire principale. Cet objectif n'est en général pas atteint : on cherche alors à réduire la probabilité que l'information soit absente de la mémoire lorsqu'elle est nécessaire (*défaut de page*). Le problème se résume alors à deux questions :

- Quand charger un objet en mémoire principale ?
  - lorsqu'on en a besoin (*chargement à la demande*),
  - avant d'en avoir besoin (*pré-chargement*).
- Où charger cet objet ?
  - S'il y a assez de place libre, dans quels emplacements le charger (*placement*) ;
  - sinon, quel(s) objet(s) renvoyer en mémoire secondaire afin de libérer de la place en mémoire principale (*remplacement*).

Plusieurs critères seront utilisés pour imaginer, évaluer et comparer les algorithmes d'allocation de mémoire :

- Critères liés à l'utilisation de la ressource mémoire, mesurée par exemple par le taux de place perdue (ou inutilisable).
- Critères liés à l'accès à l'information, comme le temps moyen d'accès ou le taux de défauts de page.
- Critères plus globaux caractérisant des performances induites par l'allocation de la mémoire : taux d'utilisation de la CPU, temps de réponse d'un système interactif, etc.

6.2 Partage de la mémoire sans réimplantation

6.2.1 Système à partition unique (va-et-vient simple)

Dans les *systèmes à partition unique* (aussi appelé *va-et-vient simple* ou « *swapping* »), une zone fixe de mémoire est réservée aux processus des usagers (voir figure **??**). Les programmes sont conservés sur disque sous forme absolue. Pour être exécuté, un programme est d'abord amené en mémoire principale, dans sa totalité. L'allocation de processeur aux programmes détermine donc les transferts. En cas de réquisition du processeur, le programme en cours doit être sauvegardé sur disque avant le chargement de son successeur.

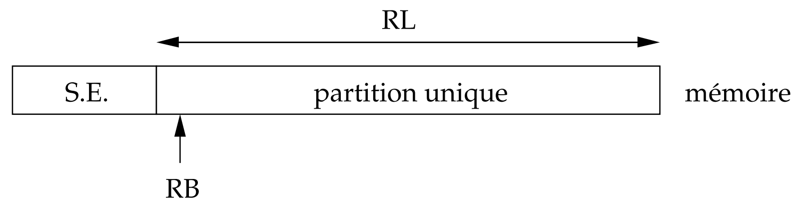


FIG. 6.3 – Système à partition unique

Afin d'éviter que des erreurs d'adressage du processus utilisateur ne viennent altérer le S.E. résident, la partition unique peut être délimitée par des registres de la CPU (*registre de base* RB pour le début et *registre limite* RL pour la taille). A chaque accès à une case d'adresse α, la CPU vérifie que ($\mathrm{RB} \leq \alpha < \mathrm{RB} + \mathrm{RL}$). Si ce test échoue, un déroutement pour erreur d'adressage est généré.

Ce schéma a l'avantage de la simplicité. Son principal inconvénient est de laisser la CPU inutilisée pendant la durée des transferts. Il est employé sur des installations de petite taille lorsque les contraintes de temps de réponse sont compatibles avec la durée et la fréquence des transferts. Des améliorations permettent de réduire le volume d'information transférée et donc la perte de temps pour la CPU :

- lorsqu'un programme est sauvegardé sur disque, on ne range que la partie modifiée (en pratique, la zone des données) ;
- l'algorithme de la « peau d'oignon » permet d'épargner des transferts : lorsqu'un programme est recouvert par un autre de taille plus petite, il suffit pour restaurer le plus gros de recharger la partie recouverte.

Ces améliorations n'apportent néanmoins qu'un gain limité, la taille des transferts n'intervenant que pour une faible part dans le temps requis. Il serait préférable de pouvoir exécuter un programme pendant la durée de transfert d'un autre. Pour ce faire, il faut donc conserver simultanément en mémoire plusieurs programmes, en partie ou en totalité. Ce mode de partage est appelé *multi-programmation*. Une multi-programmation sans réimplantation dynamique est possible par la *partition* de la mémoire.

6.2.2 Partition fixe de la mémoire

Dans un *système à partitions fixes*, la mémoire est partagée de façon statique en un nombre fixe de partitions, les tailles et limites de ces partitions étant définies lors de la génération du système.

Chaque programme est affecté de façon fixe à une partition au moment de la construction de son *image mémoire* par l'étape *d'édition de liens*. Les programmes (sous leur forme « exécutable ») sont conservés sur disque sous forme absolue, et les adresses qui y figurent sont les adresses physiques correspondant à l'implantation de chacun d'eux dans la partition qui lui a été attribuée.

Pendant qu'un programme est transféré (en entrée ou sortie), un autre programme peut être exécuté dans une autre partition ; il faut bien entendu disposer d'un processeur d'entrée/sortie autonome (canal ou ADM). La figure **??** schématise l'implantation des programmes et le chronogramme d'activité dans un système à partitions fixes.

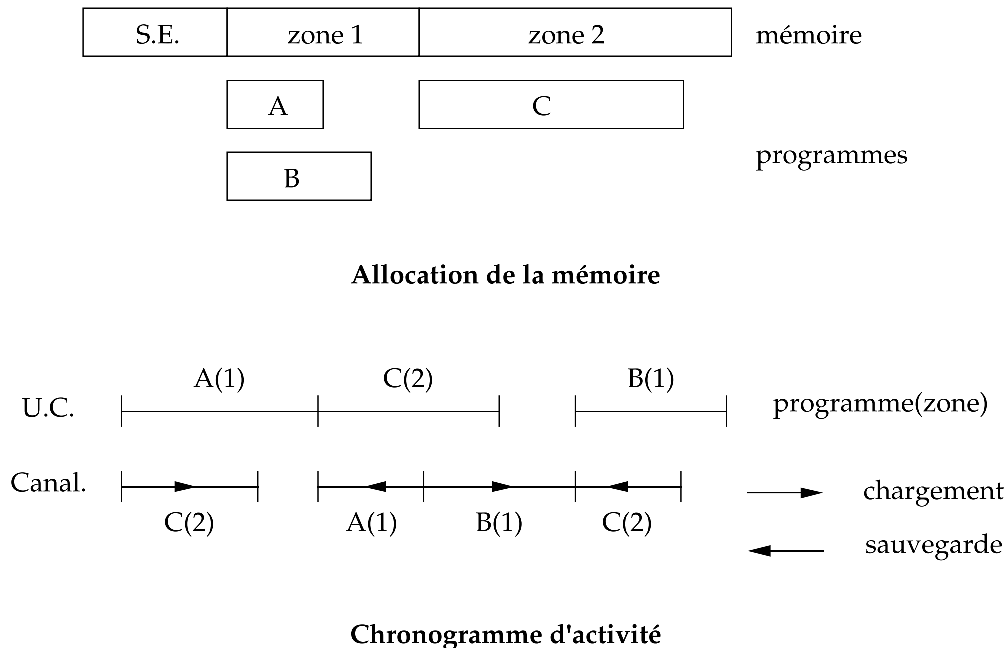


FIG. 6.4 – Système à partitions fixes

En réalité, le chronogramme peut être plus complexe, chaque programme pouvant lui-même exécuter des entrées-sorties. Dans ce cas, le processeur est également affecté à un autre programme.

Les systèmes à partitions fixes sont couramment utilisés sur des petites et moyennes installations où un petit nombre d'usagers « interactifs » coexistent avec un travail de fond. Il est alors possible de définir au moment de la génération du système, des tailles de partitions adaptées aux différentes classes de programmes. Le temps de réponse moyen des processus interactifs dépend du rapport des temps d'exécution aux temps de transferts, lui-même fonction du degré de multiplexage des partitions.

6.3 Système à partitions variables

Dans un *système à partitions variables*, le découpage en partitions n'est pas fixé une fois pour toutes, mais il est redéfini à chaque début d'exécution d'un processus. En conséquence, le chargement d'un programme (fixation des adresses) ne peut être fait qu'au dernier moment, lorsqu'une place lui est attribuée.

L'allocation de la mémoire par partitions de tailles variables suppose l'existence d'un mécanisme de réimplantation dynamique. L'utilité de celui-ci apparaîtra dans la désignation d'objets appartenant à des partitions qui auront du être déplacées en mémoire centrale.

6.3.1 Réimplantation dynamique par registre de base

Le principe que nous allons décrire est simple. Disposant d'un registre particulier ou *registre de base*, son contenu est systématiquement ajouté à toute adresse engendrée par un processus, le résultat constituant une adresse physique de l'information désignée. Si les adresses d'un programme

sont relatives à son début (i.e. si le programme est implanté à l'adresse logique 0), il suffit que le registre de base soit affecté à son adresse d'implantation en mémoire physique (voir figure **??**).

Dans ces conditions, le programme pourra être chargé en n'importe quel endroit de la mémoire. En particulier, déplacer globalement un programme dont l'exécution est commencée, peut s'opérer très facilement, à condition de modifier en conséquence la valeur contenue dans le registre de base. De plus, si programme et données sont atteints par l'intermédiaire de registres distincts, leur déplacement pourra être effectué indépendamment.

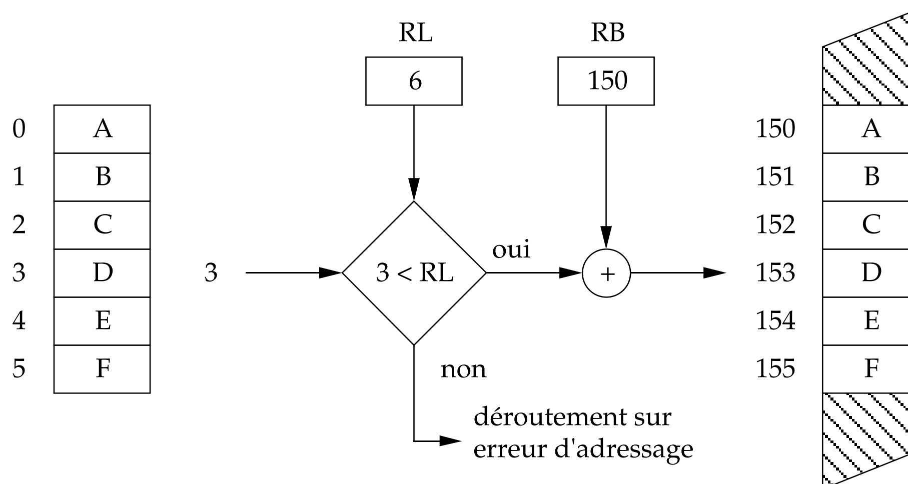


FIG. 6.5 – Passage logique / physique par registre de base et registre limite

6.3.2 Algorithmes de gestion de la mémoire par zones

Disposant d'une file constituée par les programmes en attente de traitement, un choix doit être opéré afin de déterminer leur ordre de lancement. Cet ordre pourra être tout simplement celui de la file d'attente ou dicté par des contraintes de priorités calculées par le système en fonction des demandes de ressources (place mémoire, nombre de périphériques, etc.) ou du temps d'exécution présumé. En tout état de cause, cet ordre sera aussi fonction de la taille des différentes partitions libres. Il faut auparavant résoudre les problèmes suivants :

– choix d'une représentation des partitions,
– définition des critères de sélection d'une partition libre,
– politique de libération d'une partition occupée,
– décision à prendre lorsqu'aucune partition ne convient

6.3.2.1 Représentation des partitions

Une partition est définie par sa taille et son adresse de début, contenues dans un descripteur. En supposant que les tailles demandées sont variables, le nombre de partitions le sera aussi. En conséquence, il est préférable, plutôt que de regrouper les descripteurs dans une table, de les situer dans les partitions elles-mêmes et de les chaîner entre eux.

L'ordre du chaînage a une influence sur l'efficacité des algorithmes. On peut choisir l'ordre de libération des partitions, mais le plus souvent, on utilise l'un des deux classements suivants :

– classement par adresses croissantes ou décroissantes,
– classement par tailles croissantes ou décroissantes.

6.3.2.2 Algorithmes de sélection

Une demande étant émise, on connaît la taille requise pour charger le programme du processus demandeur. Le plus souvent, cette demande sera satisfaite grâce à une partition de taille supérieure ; la différence, ou *résidu* est rattachée à la liste des partitions libres, pour autant que cette différence ne soit pas trop petite. Deux possibilités peuvent être envisagées quant au choix de la partition libre pour satisfaire une demande :

- prendre la première possible, c'est à dire, parcourir la liste jusqu'à ce que l'on en trouve une dont la taille est supérieur ou égale à la demande (« first-fit ») ;
- prendre la partition la plus petite possible, celle donnant le plus petit résidu (« best-fit »).

L'allocation d'une partition à un processus peut se décomposer en deux phases, recherche de la partition selon l'algorithme choisi puis placement du résidu dans la liste. Le classement par tailles croissantes évite de parcourir toute la liste pour trouver la plus petite partition possible (permettant ainsi une implémentation aisée du « best-fit »). Par contre le placement du résidu impose une modification du chaînage.

A l'opposé, le classement par adresses croissantes autorise une gestion rapide des résidus (seule la taille doit être modifiée, le chaînage demeurant inchangé) pour peu que le chargement s'opère en bas de partition. Cette technique est mieux adaptée à l'algorithme du « first-fit ».

On peut constater que certaines tailles sont demandées plus fréquemment que les autres. Dans ces conditions, on améliore l'efficacité de l'allocation en réservant un certain nombre de partitions possédant ces tailles privilégiées. En cas d'épuisement de cette réserve, le mécanisme classique est utilisé.

6.3.2.3 Libération d'une partition

Trois cas sont à considérer lors de la libération d'une partition :

- la partition libérée est entourée de deux partitions libres,
- la partition libérée est entourée d'une partition libre et d'une partition occupée,
- la partition libérée est entourée de deux partitions allouées.

Chaque fois que cela est possible (deux premiers cas), il est utile de regrouper les partitions libres contiguës afin de réduire la *fragmentation* de la mémoire. Il est évident que le classement par adresses croissantes est alors le plus efficace.

6.3.3 Fragmentation et compactage

Le phénomène le plus gênant dans le type d'allocation étudié ici est celui de la *fragmentation de la mémoire*, qui apparaît au bout d'un certain temps de fonctionnement et qui est dû à la multiplication des résidus de petite taille On peut aboutir à une situation où aucune partition de taille suffisante n'est disponible pour satisfaire une demande, alors que la somme des tailles de partitions libre est largement supérieure. Une solution consiste à compacter les partitions allouées en les déplaçant vers une extrémité de la mémoire, laissant apparaître ainsi à l'autre extrémité une partition libre de taille égale à la somme des tailles des partitions libres primitives.

Le compactage peut s'effectuer de deux façons possibles :

- par recopie à l'intérieur de la mémoire physique en utilisant une instruction de type MOVE (voir ci-dessous), opération monopolisant le processeur central,

MOVE ⟨adresse départ⟩, ⟨adresse arrivée⟩, ⟨longueur⟩

– par recopies successives des partitions sur disque puis du disque en mémoire, à la place voulue, en utilisant un processeur d'entrée-sortie. L'opération est alors plus longue, mais a le mérite de libérer le processeur central pour poursuivre l'exécution des autres programmes.

La figure **??** donne un exemple simple de compactage. Il montre les différentes stratégies de recopie des partitions occupées de manière à limiter la quantité de données à déplacer. On passe dans cet exemple de 200 ko à 50 ko.

Il est bien certain que seule une possibilité de réimplantation dynamique permet d'opérer un tel compactage. D'autre part, Knuth a montré que lorsque l'algorithme d'allocation ne peut satisfaire une demande, cela intervient alors que le taux de remplissage de la mémoire est tel qu'après compactage, la même situation va à nouveau apparaître très rapidement, obligeant le système à consacrer une grande partie de son temps à effectuer des compactages successifs.

En conclusion, une telle forme d'allocation n'est guère adaptée à un système interactif, mais convient mieux lorsque le nombre de partitions allouées est faible, et leur temps d'allocation grand (traitement par trains de travaux).

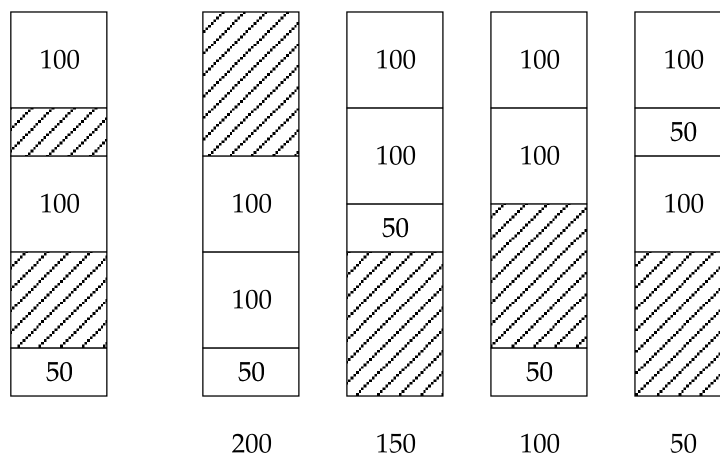


Fig. 6.6 – Différentes possibilités de compactage de la mémoire

6.4 Mémoire paginée

Une *mémoire paginée* est divisée en blocs de taille fixe, ou *pages logiques*, qui servent d'unités d'allocation. La mémoire physique est elle-même divisée en blocs de même taille appelé *pages physiques*. Nous présentons successivement les mécanismes de pagination d'une mémoire contiguë paginée et d'une mémoire paginée segmentée.

6.4.1 Pagination d'une mémoire contiguë

La figure **??** représente le schéma général d'une mémoire contiguë paginée. Le rôle de la boîte marquée « Fonction de pagination » est d'établir une correspondance entre les adresses de pages logiques et les adresses de pages physiques de manière à se qu'une page logique puisse être rangée dans une page physique quelconque. Les pages physiques deviennent ainsi des ressources banalisées dont la gestion est plus simple que celle de partitions de taille variable.

Le nombre d'emplacements d'une page (physique ou logique) est toujours une puissance de 2. Notons 2^l la taille (nombre d'emplacements) d'une page (logique ou physique) et 2^n le nombre de pages. Il y a pour l'instant autant de pages logiques que de pages physiques.

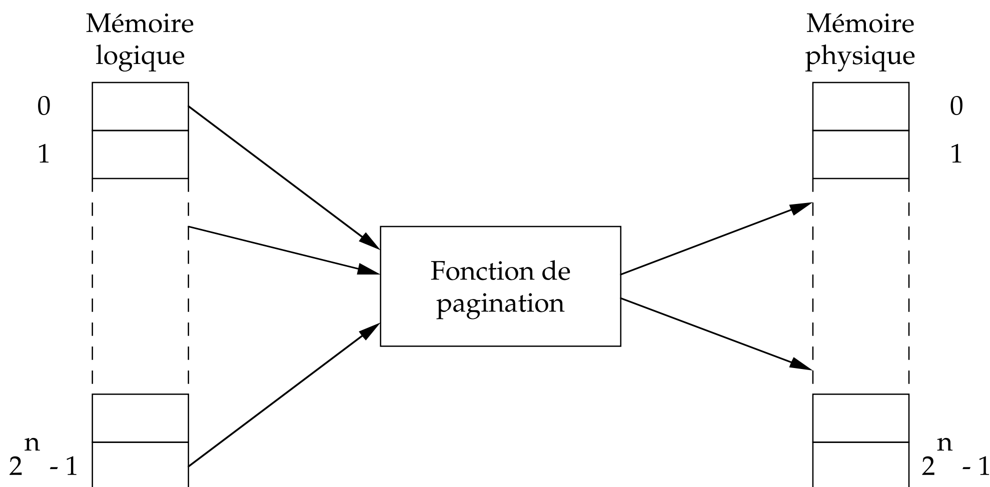


FIG. 6.7 – Mémoire linéaire paginée

Une *adresse logique paginée* est alors construite par concaténation d'un numéro de page logique (n bits) et d'un déplacement dans la page (l bits). De même, une adresse physique est la concaténation d'un numéro de page physique (n bits) et d'un déplacement (l bits). Les tailles de page usuelles vont de 0,5ko à 32ko.

étant donné un numéro de page logique (`npl`), la fonction de pagination permet de trouver le numéro de la page physique (`npp`) qui la contient. Dans un souci d'efficacité, cette fonction est réalisée par un mécanisme matériel.

La réalisation la plus courante de la fonction de pagination utilise une *table de pages* en mémoire, indexée par un numéro de page logique (table desc de la figure **??**). Lors d'un accès à la mémoire, la correspondance adresse logique/adresse physique (qui est une opération matérielle), est mise en œuvre comme suit :

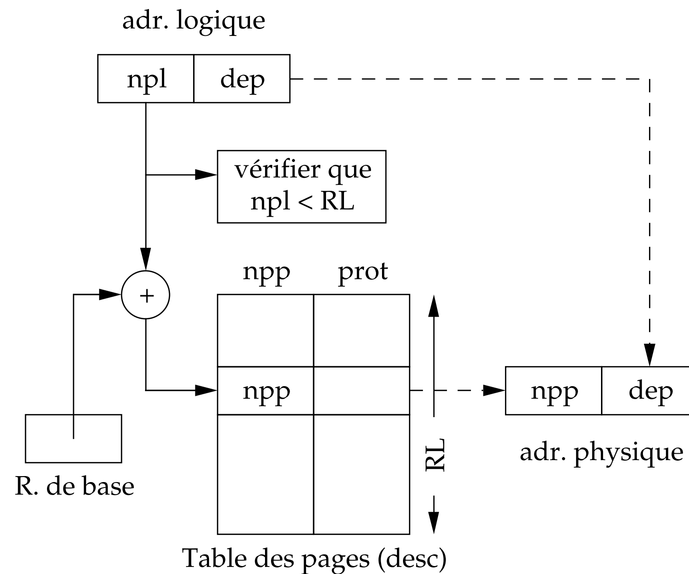


FIG. 6.8 – Organisation d'une table de pages

(`npl, déplacement`) = ⟨adresse logique⟩
si (`npl` $<$ `RL`) **alors**
  si ⟨les protections sont respectées⟩ **alors**
    ⟨adr. physique⟩ = (`desc[npl].npp, déplacement`)
  sinon
    ⟨déroutement sur violation de protection⟩
  fin si
sinon
  ⟨déroutement sur erreur d'adressage⟩
fin si

Le champ `desc[npl].prot` indique le mode d'accès autorisé à la page logique `npl`. Cette information est utilisée par les mécanismes de protection et un accès non autorisé provoque un déroutement pour violation de protection.

Notons qu'une table de pages représente le contenu d'une mémoire logique particulière. Si le système d'exploitation permet à chaque processus, ou à chaque usager du système, de définir une mémoire logique distincte, il doit gérer une table de pages distincte par processus ou par usager. Le pointeur vers l'origine de cette table (`RB`) fait alors partie du contexte du processus ou de l'usager. Les tables des pages se trouvent en mémoire physique, dans la partition réservée au système d'exploitation.

La mémoire logique d'un processus n'est plus représentée d'une manière contiguë en mémoire centrale (voir figure **??***). En effet, l'indirection des accès par la table de pages permet de loger les pages logiques dans n'importe quelle page physique. De ce fait, la gestion de la mémoire physique revient simplement à gérer une liste des pages physiques libres sans idée de regroupement. Les problèmes liés à la fragmentation externe disparaissent mais la fragmentation interne se fait plus présente puisque la page devient l'unité élémentaire d'allocation et de libération (en pratique plusieurs kilo-octets).*

L'accès à une page logique nécessite maintenant deux références à la mémoire en raison de la consultation de la table des pages. Cette augmentation du temps d'accès moyen est bien sur intolérable. La réduction de ce coût passe par deux points :

– observer le comportement des processus (vis à vis de la mémoire),

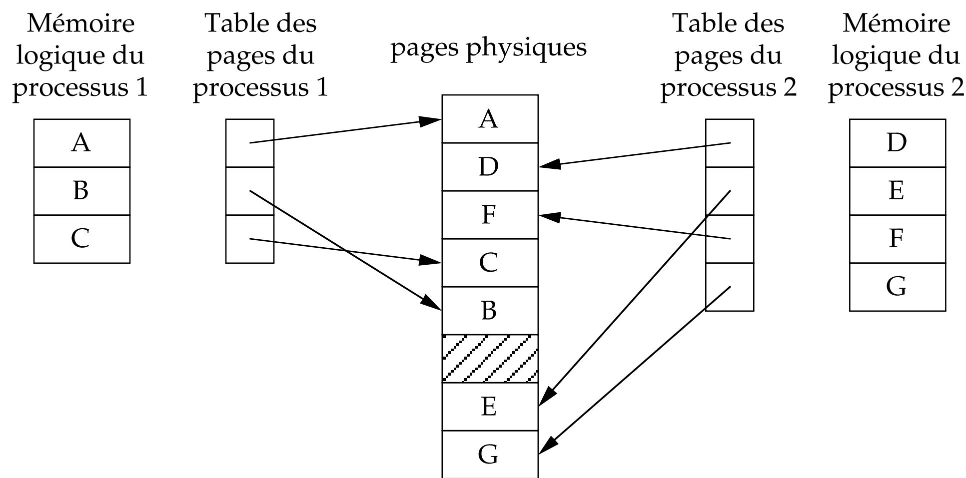


FIG. 6.9 – Un exemple sur deux mémoires logiques paginées

– optimiser la transformation des adresses au moyen d'un circuit particulier : les mémoire associatives.

Pour éviter l'accès à cette table (et donc réduire le temps d'accès moyen), on passe par un circuit particulier : une mémoire associative. Mais avant de présenter cette mémoire il faut discuter de l'utilisation de la mémoire par les processus

6.4.2 Comportement des processus en mémoire paginée

Le comportement d'un processus dans son espace logique détermine ses demandes de mémoire physique. Il est donc utile de connaître les caractéristiques de ce comportement pour améliorer l'efficacité des algorithmes de gestion dynamique de la mémoire. Donnons d'abord quelques définitions :

– L'écoulement du temps est repéré par l'exécution des instructions successives : l'exécution d'une instruction définit une unité de temps. Ce temps est dit virtuel car il suppose que le programme dispose de toutes les ressources nécessaires (mémoire et processeur). En cas de partage de ressources, on peut ainsi raisonner sur un programme donné en faisant abstraction des autres.

– La mémoire logique paginée est découpée en page contiguës de taille fixe. L'accès à un emplacement d'une page est appelé *référence* à cette page. Le numérotage des pages permet d'étiqueter les références.

– Le comportement du processus est défini par la série des numéros de pages référencées au cours de l'exécution. Cette séquence s'appelle *chaîne de référence* pour le processus considéré.

  L'exécution d'une instruction peut donner lieu à plusieurs références distinctes : pages contenant l'instruction, le ou les opérandes.

L'expérience montre que les chaînes de références des processus possèdent des caractéristiques communes que nous définirons d'abord de manière qualitative.

– *Non-uniformité*. Soit n_i le nombre total de références à une page p_i. La répartition des n_i n'est pas uniforme : un faible pourcentage des pages cumule généralement un taux très important du nombre total des références. Il est courant de constater que plus des 75% des références intéressent moins de 20% des pages.

– *Propriété de Localité*. Sur un temps d'observation assez court, la répartition des références présente une certaine stabilité : les références observées dans un passé récent sont en général une bonne estimation des prochaines références.

A partir de cette constatation de localité, on peut créer un modèle de comportement des programmes. Dans ce modèle, le déroulement d'un programme est défini comme une succession de phases séparées par des transitions. Une phase i est caractérisée par un ensemble de pages S_i et un intervalle de temps virtuel T_i. Lorsque le programme entre en phase i, il y reste un temps T_i en effectuant principalement des références à des pages de S_i. Ensuite, il subit un transition durant laquelle les références aux pages sont dispersées, avant d'entrer dans la phase $i + 1$.

Les phases constituent donc des périodes de comportement stable et relativement prévisible, alors que les transitions correspondent à un comportement plus erratique. L'expérience montre que les périodes de transition ne représentent qu'une faible partie du temps virtuel total, la majeure partie du temps virtuel étant occupé par des phases de longues durée (quelques centaines de milliers d'instructions).

Qualitativement, ce type de comportement s'explique par le fait que les programmes sont souvent organisés en procédures possédant chacune un contexte spécifique, que les accès aux données sont souvent concentrés (parcours de tableau), que les programmes comportent des boucles concentrant aussi les références.

La notion d'*ensemble de travail* (« working set ») est également utilisée pour caractériser le comportement des programmes et prévoir d'après l'observation. Soit $W(t, T)$ l'ensemble des pages ayant été référencées entre les temps $t - T$ et t. D'après la propriété de localité, ces pages ont une probabilité plus élevée que les autres de faire l'objet d'une référence au temps t à condition toutefois que la taille de la fenêtre d'observation T soit convenablement choisie. En admettant un comportement suivant le modèle phase/transition, T devra être inférieur à T_i en phase i.

6.4.3 Mémoire associative et pagination

Une *mémoire associative* est un ensemble de couple $\langle \text{entrée}, \text{sortie} \rangle$. La présence d'une valeur sur le bus d'entrée provoque soit l'apparition d'une valeur de sortie soit un signal d'échec indiquant que cette entrée n'existe pas dans la mémoire associative (figure **??**). Ces mémoires ont quelques dizaines à quelques centaines d'entrées et leur coût très élevé a empêché leur extension à des mémoires de plus grande taille.

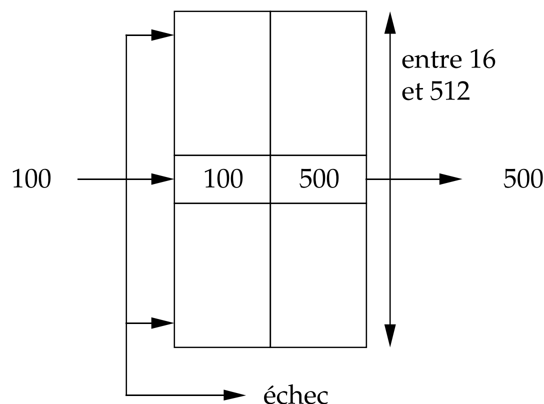


FIG. 6.10 – Schéma d'une mémoire associative

Dans cette mémoire associative on conserve les couples $\langle \texttt{npl}, \texttt{npp} \rangle$ relevés lors des accès les plus récents. En raison de la propriété de localité des programmes, on a une probabilité élevée (80% à 95% avec les tailles usuelles) de trouver dans la mémoire associative le numéro de la page logique adressée et donc de déterminer sa page physique. Ce n'est qu'en cas d'échec que l'on passe par la table des

pages ; la mémoire associative est alors mise à jour, le couple $\langle \texttt{npl}, \texttt{npp} \rangle$ courant remplaçant le plus anciennement utilisé (figure **??**).

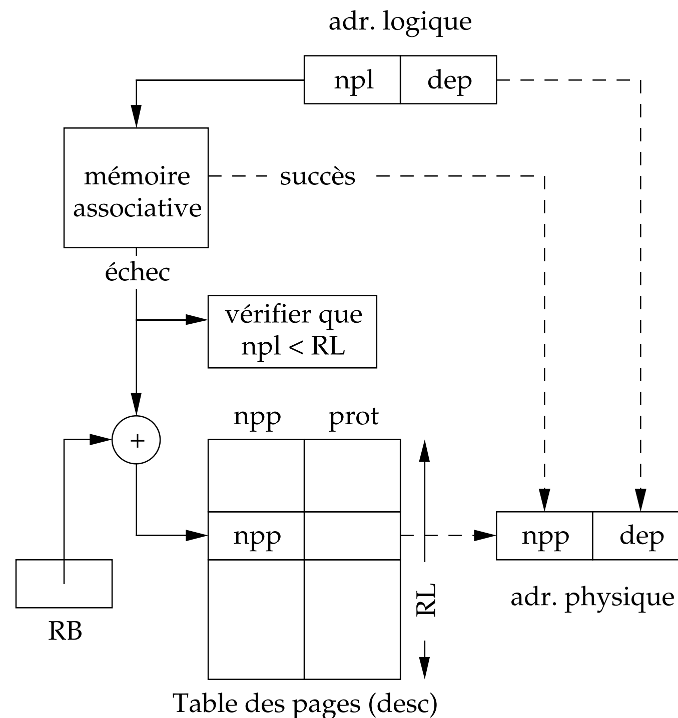


FIG. 6.11 – Pagination avec mémoire associative

En partant du principe que l'accès à la mémoire physique prend 100 ns et que le temps de recherche de la mémoire associative est de 20 ns, le temps moyen d'accès est compris entre

$$\begin{array}{lcl} 0,8 \times (100 + 20) + 0,2 \times (100 + 20 + 100) & = & 140 \text{ ns} \\ 0,95 \times (100 + 20) + 0,05 \times (100 + 20 + 100) & = & 125 \text{ ns} \end{array}$$

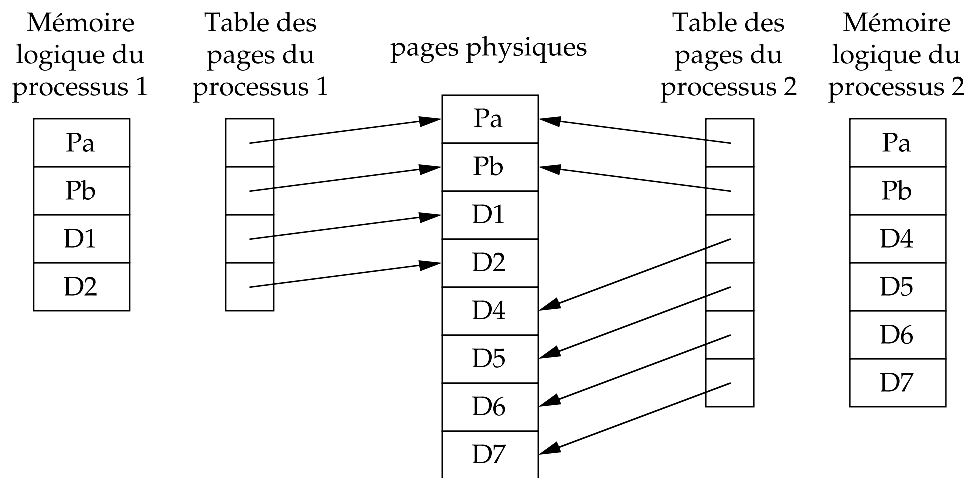
suivant la probabilité de réussite et donc la taille de la mémoire associative. Finalement, le temps d'accès moyen n'a augmenté que de 25%, mais la gestion de la mémoire est beaucoup plus souple et les problèmes de fragmentation externe n'existent plus.

6.4.4 Partage et protection de l'information

L'utilisation d'informations partagées entre plusieurs mémoires logiques soulève trois problèmes :

- la *désignation* : comment adresser de manière uniforme les informations partagées ;
- le *partage physique* : comment assurer que les informations partagées existent en exemplaire unique ;
- la *protection* : comment garantir le respect des règles d'accès (éventuellement sélectives) aux informations partagées.

Dans un système paginé, l'unité élémentaire de partage est la page. Pour être partagés, les informations doivent se trouver sur une (ou plusieurs) page logique partagée. Cette page peut être chargée dans une page physique quelconque ; les tables de pages des mémoires logiques où figure cette page logique contiennent alors, à l'entrée correspondante, le même numéro de page physique. Dans l'exemple présenté par la figure **??** les pages contenant le programme (Pa et Pb) sont partagées mais les pages de données (D1, ..., D7) ne le sont pas.



Les pages contenant le programme (Pa et Pb) sont partagées, mais les pages de données (D1, ..., D7) ne le sont pas.

FIG. 6.12 – Partage de pages entre mémoires logiques paginées

Si l'unité de partage est la page, une page physique partagée peut recevoir des droits d'accès distincts dans chaque mémoire logique où elle figure. Ces droits sont spécifiés à l'entrée correspondante de la table de pages.

6.5 Mémoire segmentée

6.5.1 Principe de la segmentation

Dans les systèmes de mémoire paginée, la division en pages est arbitraire et ne tient pas compte du mode d'organisation des données d'un processus. Notamment, il est fort probable que certaines structures de données vont se trouver « à cheval » sur plusieurs pages ce qui peut être la cause de temps d'accès plus importants.

L'objectif des *mémoires segmentées* est de mettre en rapport la structure logique de la mémoire (vue des processus) et l'implantation physique de cette mémoire. Pour ce faire, la *mémoire logique segmentée* d'un processus est définie comme un ensemble de segments numérotés à partir de zéro (figure **??**). Un *segment* est une zone contiguë de taille variable.

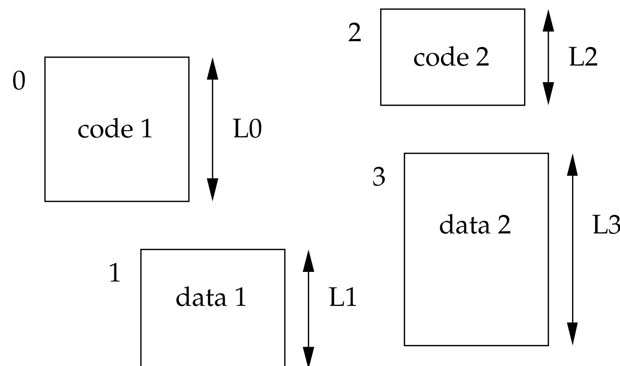


FIG. 6.13 – Une mémoire segmentée

Une *adresse logique* dans un système segmenté (aussi appelée *adresse segmentée*) est un couple

$$\langle \text{n}^\circ \text{ de segment}, \text{déplacement} \rangle$$

Comme dans les mémoires paginées, le S.E. maintient une *table des segments* pour chaque processus (figure **??**). La correspondance proprement dite est établie par le matériel de la manière suivante :

```
⟨seg,dépl⟩ := ⟨adresse logique segmentée⟩
si (seg < RL) et (dépl < desc[seg].taille) alors
   si ⟨les protections sont respectées⟩ alors
      ⟨adresse physique⟩ = desc[seg].origine + dépl
   sinon
      ⟨déroutement sur violation de protection⟩
   fin si
sinon
   ⟨déroutement sur erreur d'adressage⟩
fin si
```

Ce mécanisme doit être couplé à une mémoire associative pour améliorer le temps d'accès moyen en évitant l'utilisation de la table des segments.

Les avantages de cette organisation sont doubles : d'une part, la notion de segment est directement utilisable dans un processus et de ce fait on peut espérer une réduction des temps d'accès moyens (les accès fréquents étant regroupés sur un petit groupe de segments, voire même sur un segment unique) ; d'autre part, la notion de protection est plus facilement utilisable puisqu'elle porte directement sur des segments, c'est à dire des objets logiques.

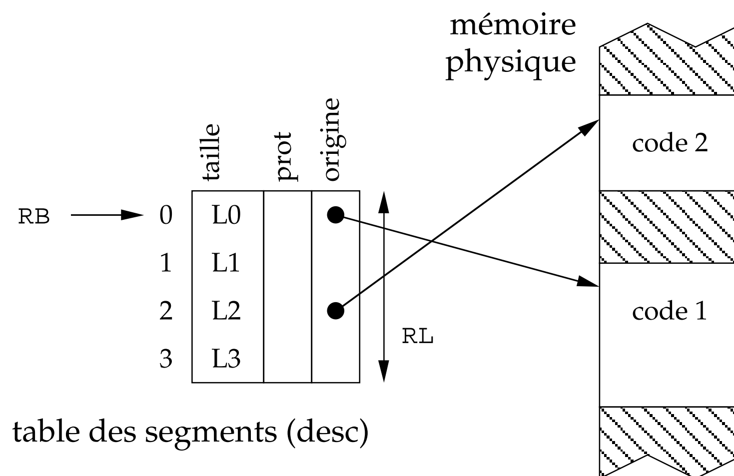


FIG. 6.14 – Table des segments d'un processus

Les segments sont des zones contiguës (du moins pour l'instant). On retrouve donc les problèmes d'allocation/libération de zones et l'apparition d'une fragmentation externe éventuellement corrigée par des compactages de la mémoire. Dans le cas de la mémoire segmentée, ces compactages impliquent une remise à jour des pointeurs « origine » des tables de segments.

6.5.2 Pagination d'une mémoire segmentée

La pagination d'une mémoire segmentée vise à rendre plus souple l'allocation de mémoire aux segments en levant la restriction de contiguïté pour le placement d'un segment. Une entrée de la table des segments contient, outre les informations propres au segment (taille, protection, type), un pointeur vers la table des pages de ce segment. Comme dans les techniques précédentes, une mémoire associative conserve les dernières références.

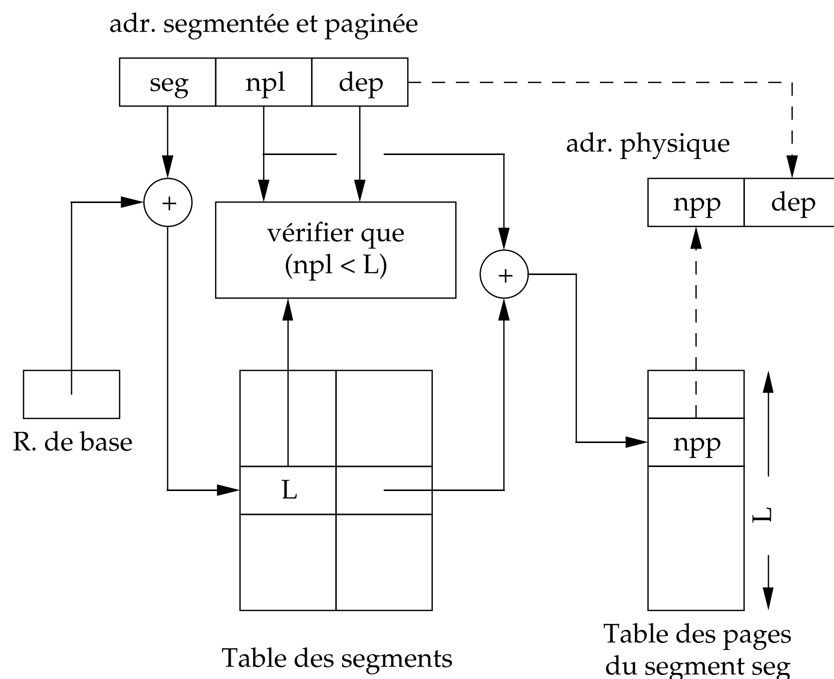


FIG. 6.15 – Pagination d'une mémoire segmentée

Exemple : Le système Multics utilise une allocation par pages pour sa mémoire segmentée. En raison du nombre élevé de segments, les tables de pages des segments et les tables de segments

elles-mêmes peuvent dépasser la taille d'une page et être elles-mêmes paginées. La table des pages d'un segment est maintenue en mémoire principale tant que ce segment est « actif » (c'est à dire que le fichier correspondant est ouvert pour au moins un processus).

6.5.3 Partage de segments

Dans une mémoire logique segmentée, le partage s'applique aux segments et les tables de pages des segments partagés sont elles-mêmes partagées (dans le cas d'un système segmenté paginé). Tous les descripteurs d'un segment contiennent alors, non pas l'adresse de ce segment, mais celle de sa table de pages qui est unique.

Si l'unité de partage est le segment, la protection sélective s'applique globalement au segment. Les droits d'accès à un segment pour un processus figurent dans la table des segments de ce processus. Si des droits d'accès individuels aux pages du segment sont spécifiés, ils figurent dans la table de pages partagée par les processus utilisateurs et sont donc les mêmes pour tous. Ils doivent alors être compatibles avec les droits globaux associés au segment.

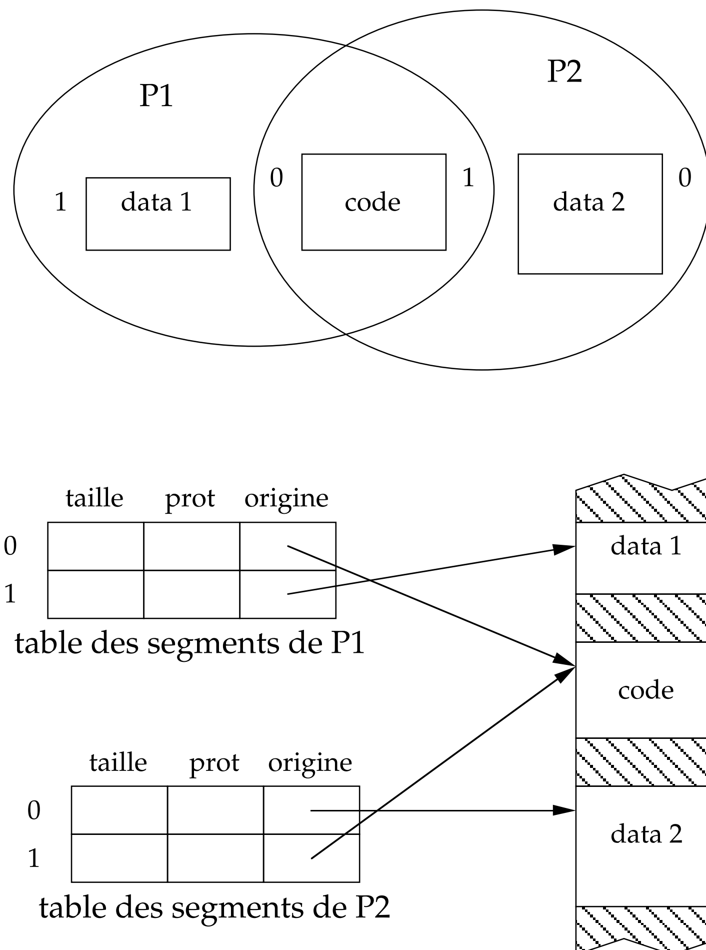


FIG. 6.16 – Partage de segments dans un système segmenté

Chapitre 7

Mémoire virtuelle paginée

Le principe de localité nous dit que, sur un petit intervalle de temps, un processus utilise 20% de ses pages logiques. On peut donc en conclure que 80% de pages logiques sont en mémoire sans raison valable. Il est donc raisonnable d'enlever ces pages inutiles de manière à offrir plus de place mémoire pour d'autres processus et ainsi augmenter le degré de multi-programmation.

7.1 Pagination simple d'une mémoire virtuelle

La figure **??** représente le schéma général d'une mémoire virtuelle paginée. La mémoire virtuelle est plus importante que la mémoire physique et les pages virtuelles qui ne se trouvent pas en mémoire physique sont stockées en mémoire secondaire.

Nous avons maintenant 2^l la taille des pages (virtuelles ou physiques), 2^p le nombre de pages virtuelles (taille de la mémoire virtuelle) et 2^c le nombre de pages physiques (taille de la mémoire physique) avec $p > c$.

Une *adresse virtuelle* est donc un couple ⟨npv, dep⟩ avec « npv » un numéro de page virtuelle (sur p bits) et « dep » un déplacement (sur l bits). De même, une *adresse physique* est un couple ⟨npp, dep⟩ avec « npp » un numéro de page physique (sur c bits) et « dep » un déplacement (sur l bits).

La *fonction de pagination* (figure **??**) a maintenant la charge de transformer les adresses virtuelles en adresse physique et de détecter les *défauts de page*, c'est à dire les accès à des pages virtuelles qui ne sont stockées dans aucune page physique.

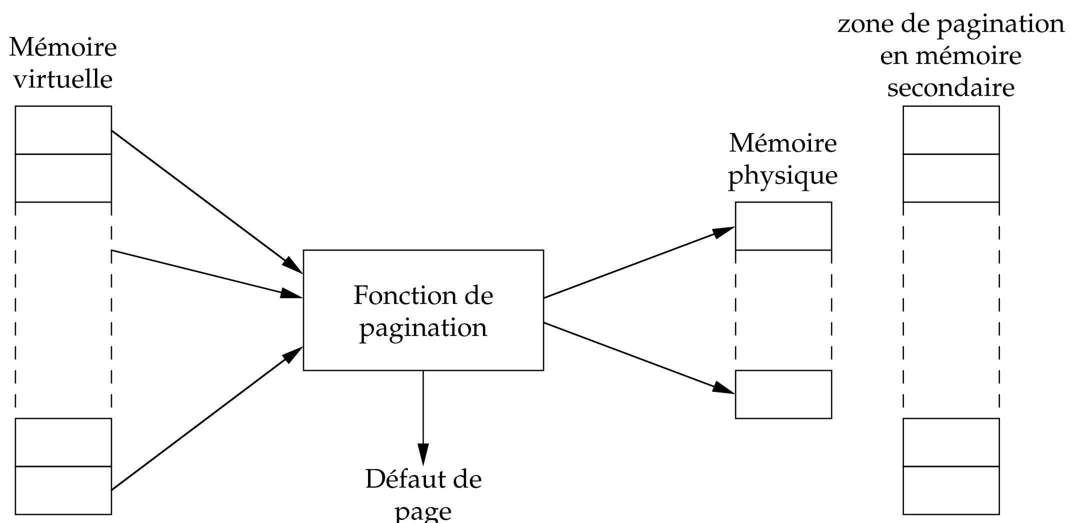


Fig. 7.1 – Mémoire virtuelle paginée

Pour réaliser cette opération, la *table des pages virtuelles* comporte (pour chaque page virtuelle) les informations suivantes (figure **??**) :

- un numéro de page physique (npp),
- un indicateur de présence (présent) (1 bit),
- un indicateur de modification (modif) (1 bit),
- un mode d'accès autorisé (prot).

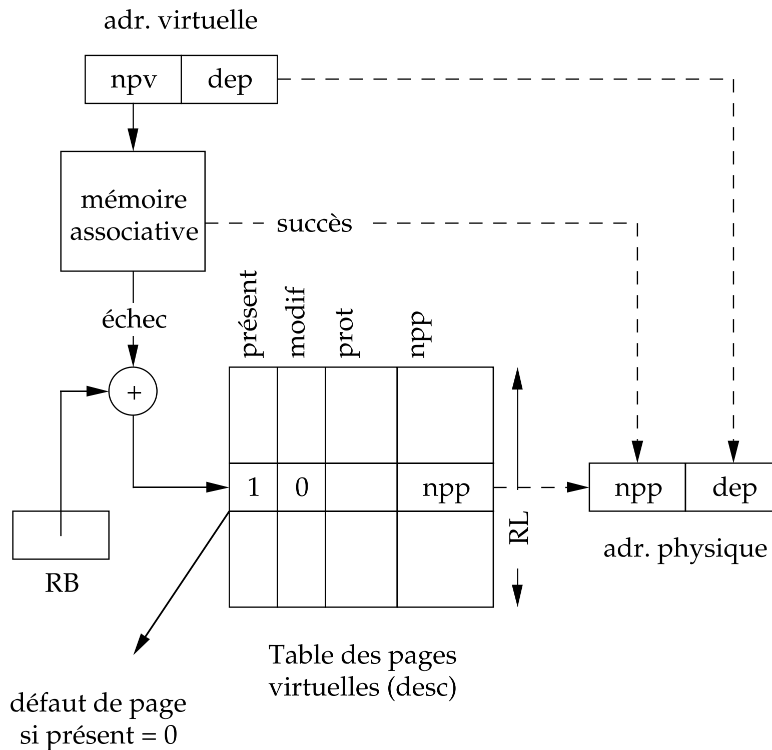


FIG. 7.2 – Transformation adresse virtuelle / adresse physique

Lors d'un accès à la mémoire, la correspondance adresse virtuelle/adresse physique, est mise en œuvre comme suit :

⟨npv, dep⟩ := ⟨adresse virtuelle⟩
⟨vérifier que (npv < RL)⟩
⟨vérifier les protections⟩
si (desc[npv].présent = 1) **alors**
  ⟨adresse physique⟩ := (desc[npv].npp, dep)
sinon
  ⟨déroutement pour défaut de page⟩
fin si

Lorsque la page virtuelle est présente, le bit modif indique si la page virtuelle a été modifiée depuis son chargement en mémoire ; cette information, mise à jour automatiquement par le matériel, est utilisée par l'algorithme de remplacement pour éviter éventuellement la sauvegarde d'une page remplacée.

Exemple : DEC VAX 11/780. La mémoire virtuelle accessible à un processus sur le DEC VAX 11/780 est définie par une adresse de 32 bits. 30 bits définissent l'espace virtuel accessible à un usager (numéro de page virtuelle sur 21 bits, déplacement sur 9 bits). Les 2 bits restants permettent de définir une extension de l'espace virtuel utilisée par le système d'exploitation. Chaque entrée de la table des pages virtuelles d'un processus comporte 32 bits :

bits	signification
31	bit de présence
3-27	protection
26	bit d'écriture
2-21	champ réservé au système d'exploitation
20-0	numéro de page physique

Une mémoire associative retient les couples ⟨npv, npp⟩ les plus récents.

7.2 Pagination à deux niveaux d'une mémoire virtuelle

La tendance à la croissance de la taille des mémoires virtuelles se heurte au problème de l'encombrement de la mémoire principale par les tables de pages virtuelles, dont la taille augmente en proportion. La pagination à deux niveaux (voir même à plusieurs niveaux) vise à résoudre ce problème en limitant les tables de pages virtuelles aux seules parties effectivement utilisées de la mémoire virtuelle.

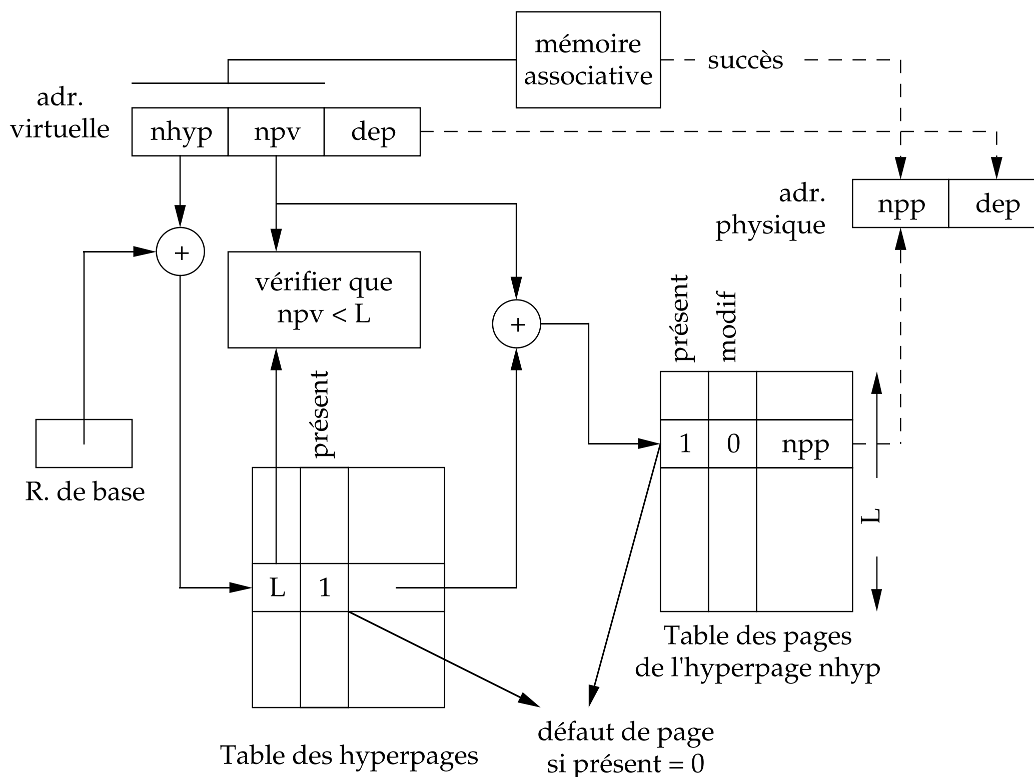


FIG. 7.3 – Pagination à deux niveaux

Dans ce schéma, la mémoire virtuelle est divisée en *hyperpages*, elles même divisées en pages (virtuelles). Une adresse virtuelle a maintenant la forme ⟨nhyp, npv, dep⟩. Le numéro « nhyp » permet d'accéder à une table d'hyperpages dont chaque entrée comporte un pointeur vers la table de pages de l'hyperpage. Seules les hyperpages effectivement utilisées se trouvent en mémoire principale. Dans certains systèmes les tailles de pages et d'hyperpages pouvant être variables, un champ taille permettra de contrôler la limitation et détecter une éventuelle erreur d'adressage. Les tables de pages sont utilisées comme dans la pagination simple. Une mémoire associative qui conserve les triplets ⟨nhyp, npv, npp⟩ les plus récents, accélère la consultation (qui nécessite deux accès supplémentaires à la mémoire en cas d'échec).

Exemple : La pagination a deux niveaux a été introduite sur l'IBM 360/67 et reprise sur ses successeurs de la série 370. La taille des pages et des hyperpages peut être choisie parmi plusieurs combinaisons.

h	**p**	**d**
8	4	12
8	5	11
4	8	12
8	9	11

On peut donc choisir entre deux tailles de pages (2 ou 4 Ko) deux tailles d'hyperpages (64 ou 1024 Ko).

Notons que la mémoire virtuelle reste contiguë : le dernier emplacement de l'hyperpage h a pour successeur le premier emplacement de l'hyperpage $h+1$. C'est donc par abus de langage que cette technique est souvent appelée *segmentation*. Ce découpage de la mémoire virtuelle peut néanmoins être utilisé pour simuler une mémoire segmentée.

7.3 Mécanisme du défaut de page

Outre la traduction proprement dite des adresses (correspondance npv/npp), le mécanisme d'accès à une mémoire paginée doit réaliser les opérations suivantes :

– mise à jour du bit d'écriture et du bit d'utilisation (si ils existent),

– détection du défaut de page (desc[npv].présent = 0) qui provoque un déroutement.

Le programme du traitement de déroutement pour défaut de page doit :

1. trouver en mémoire secondaire la page virtuelle manquante,
2. trouver une page physique libre en mémoire principale ; s'il n'y a pas de page physique libre, il faut en libérer une en
   – choisissant une page virtuelle à enlever de la mémoire (c'est la *victime* du défaut de page) ;
   – sauvegardant la victime (si nécessaire) dans la zone de pagination ;
3. provoquer le chargement de la page virtuelle dans la page physique ainsi rendue libre.

L'étape (1) nécessite d'avoir pour chaque mémoire virtuelle, une description d'implantation. Sa forme la plus simple est celle d'une table qui indique pour chaque page virtuelle son adresse en mémoire secondaire. Une mémoire segmentée comporte une telle table pour chaque segment. Une autre forme de description combine mémoire virtuelle et fichiers : à une zone de mémoire virtuelle est associé le contenu d'un ou de plusieurs fichiers. La localisation d'une page virtuelle en mémoire secondaire est alors déterminée en consultant la table d'implantation du fichier dont elle contient un élément.

Exemple : Dans le système Multics, il n'y a pas de distinction entre segment et fichier : les tables d'implantation des fichiers décrivent la localisation des pages virtuelles en mémoire secondaire. Réciproquement, l'accès aux fichiers est opéré par association de leur articles aux pages d'une mémoire virtuelle. Cette opération est appelée *couplage*.

L'étape (2) met en œuvre un *algorithme de remplacement* (voir **??**). Elle nécessite de conserver une table d'occupation de la mémoire physique qui indique pour toute page physique occupée l'identité de la page virtuelle qui l'occupe ainsi que les renseignements complémentaires (protection, partage, etc.).

7.4 Comportement des programmes en mémoire virtuelle

Le comportement d'un programme en mémoire restreinte, pour un algorithme de remplacement donné, est caractérisé par la suite de ses défauts de pages. L'expérience montre que l'on observe des propriétés communes au comportement de la majorité des programmes, relativement indépendantes de l'algorithme de remplacement utilisé. Ce sont donc des caractéristiques intrinsèques du comportement.

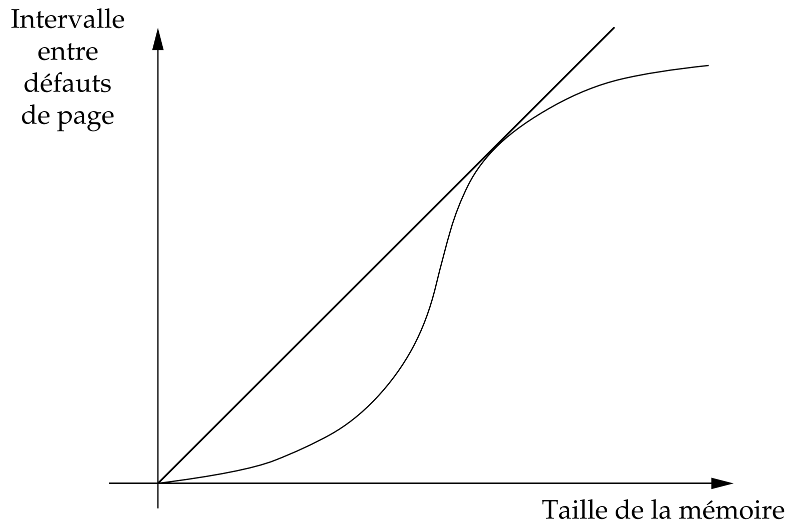


FIG. 7.4 – Intervalle moyen entre défauts de page en mémoire restreinte

Elles peuvent être illustrées par deux courbes obtenues en exécutant un programme avec différentes tailles de mémoire principale. L'allure de ces courbes est représentative d'un type de comportement fréquemment observé. La figure **??** représente l'intervalle moyen entre deux défauts de page successifs, dit durée de vie. La figure **??** représente le nombre total de défauts de page observés au cours de l'exécution d'un programme.

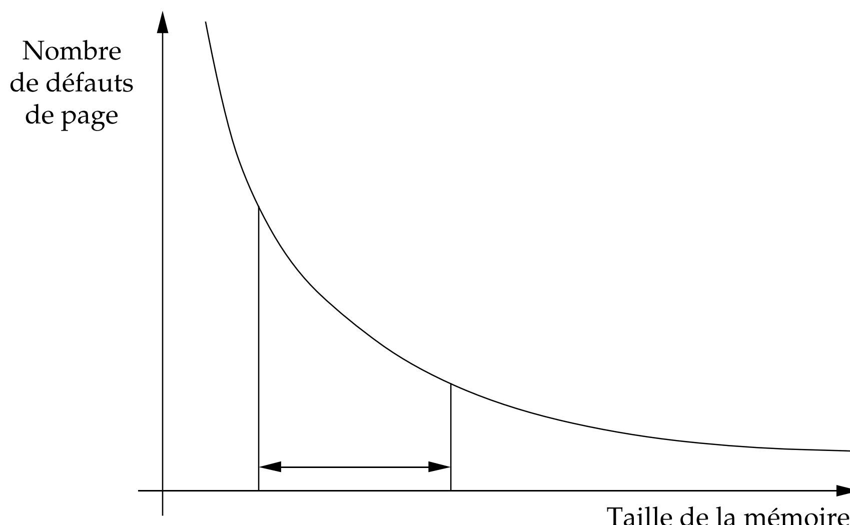


FIG. 7.5 – Nombre total de défauts de page en mémoire restreinte

On constate que lorsque la taille de mémoire diminue, ce nombre croît d'abord lentement. Au dessous d'une certaine taille, la croissance devient exponentielle.

7.5 Gestion d'une mémoire virtuelle paginée

7.5.1 Paramètres d'une politique d'allocation

Les politiques d'allocation d'une mémoire paginée peuvent être classées selon plusieurs critères. Nous supposons que le système est multi-programmé entre plusieurs processus dont chacun possède sa propre mémoire virtuelle.

- *Partition fixe ou variable.* Dans une politique à partition fixe, un nombre fixe de pages physiques est attribué à chacun des processus ; notons que ce nombre n'est constant que pendant les périodes où le nombre de processus multiprogrammés est lui-même constant. Dans une politique à partition variable, le nombre de pages physiques attribuées à chaque processus varie au cours du temps. Les pages physiques étant banalisées, c'est leur nombre (et non leur identité) qui est le paramètre significatif

- *Pagination à la demande.* une page virtuelle n'est chargée en mémoire qu'à la suite d'une référence donnant lieu à un défaut de page.

- *Pré-chargement.* Lorsqu'une page virtuelle est chargée à l'avance, avant toute référence à une information qu'elle contient, on dit qu'il y a pré-chargement.

- *Remplacement local* ou *global.* Il y a remplacement de page lorsqu'une page virtuelle est chargée dans une page physique occupée, c'est à dire contenant une page virtuelle chargée antérieurement et susceptible d'être encore utilisée (cette dernière page est souvent appelée la *victime*). L'algorithme de remplacement est dit *local* ou *global* selon que la victime est choisie parmi les pages virtuelles du processus qui provoque le chargement ou parmi l'ensemble de toutes les pages virtuelles présentes en mémoire.

Avant d'étudier et de comparer les algorithmes de remplacement de pages, il faut mentionner des critères valables quel que soit l'algorithme, et qui sont appliqués en priorité.

1. *Pages virtuelles « propres » ou « sales ».* Toutes choses égales par ailleurs, il est toujours moins coûteux de remplacer une page virtuelle qui n'a pas été modifiée depuis son chargement (page *propre*) plutôt qu'une page modifiée (page *sale*). Une page propre possède une copie conforme en mémoire secondaire et ne doit donc pas être sauvegardée. L'indicateur « modif », entretenu automatiquement permet d'appliquer ce critère.

2. *Pages virtuelles partagées.* Une page virtuelle utilisée par un seul processus doit être remplacée de préférence à une page partagée entre plusieurs processus.

3. *Pages virtuelles à statut spécial.* Dans certains cas, on souhaite donner temporairement à une page virtuelle un « statut » spécial qui la protège contre le remplacement. Ce cas se présente surtout pour des pages utilisées comme tampons d'entrée-sortie pendant la durée d'un transfert.

7.5.2 Algorithmes de remplacement

Nous présentons d'abord deux algorithmes qui servent de référence : l'algorithme optimal, qui suppose une connaissance complète du comportement futur du programme, et un algorithme « neutre » qui n'utilise aucune information.

- *Algorithme optimal* (OPT). Pour une chaîne de références donnée, on peut montrer que l'algorithme suivant minimise le nombre total de défauts de pages : lors d'un défaut de page, choisir comme victime une page virtuelle qui ne fera l'objet d'aucune référence ultérieure, ou,

à défaut, la page qui fera l'objet de la référence la plus tardive. Cet algorithme suppose une connaissance de l'ensemble de la chaîne de références ; il est donc irréalisable en temps réel. Il permet d'évaluer par comparaison les autres algorithmes.

– *Tirage aléatoire* (ALEA). La victime est choisie au hasard (loi uniforme) parmi l'ensemble des pages virtuelles présentes en mémoire. Cet algorithme n'a aucune vertu particulière, car il ne tient aucun compte du comportement observé ou prévisible du programme ; il sert lui aussi de point de comparaison.

– *Ordre chronologique de chargement* (FIFO ou « First In, First Out »). Cet algorithme choisit comme victime la page virtuelle la plus anciennement chargée. Son principal intérêt est sa simplicité de réalisation : il suffit d'entretenir dans une file les numéros des pages physiques où sont chargées les pages virtuelles successives.

– *Ordre chronologique d'utilisation* (LRU ou « Least Recently Used »). Cet algorithme tente d'approcher l'algorithme optimal en utilisant la propriété de localité. Son principe est le suivant : puisque les pages virtuelles récemment utilisées ont une probabilité plus élevée que les autres d'être réutilisées dans un futur proche, une page virtuelle non utilisée depuis un temps élevé a une probabilité faible d'être utilisée prochainement. L'algorithme choisit donc comme victime la page virtuelle ayant fait l'objet de la référence la plus ancienne.

  La réalisation de l'algorithme impose d'ordonner les pages physiques selon la date de dernière référence de la page virtuelle qu'elles contiennent. Pour cela, une information doit être associée à chaque page physique et mise à jour à chaque référence. Cette information peut être une date de référence ; une solution plus économique, mais encore chère, consiste à utiliser un compteur incrémenté de 1 à chaque référence ; la page physique dont le compteur a la valeur la plus faible contient la victime. Les compteurs ayant une capacité limitée, il doivent être remis à zéro dès que l'un d'eux atteint sa capacité maximale ; la réalisation de LRU n'est donc qu'approchée. En raison de son coût, cette solution n'a été utilisée que sur des installations expérimentales. Si la taille du compteur est réduite a 1 bit, on obtient l'algorithme suivant dont le coût est acceptable.

– *Algorithme « de la seconde chance »* (FINUFO ou « First In Not Used, First Out »). Cet algorithme est une approximation très grossière de LRU. A chaque page physique est associé un bit d'utilisation (noté « U »), mis à 1 lors d'une référence à la page qu'elle contient. Les pages physiques sont ordonnées dans une liste circulaire et un pointeur « victime » désigne la dernière page physique chargée. L'algorithme s'écrit comme suit :

```
tant que (U[victime] = 1) faire
  U[victime] := 0 ;
  victime := suivant(victime) ;
fin faire
U[victime] := 1 ;
victime := suivant(victime) ;
```

  Le pointeur progresse jusqu'à la première page physique dont le bit est à zéro ; les pages physiques rencontrées en route (dont le bit est donc à 1) reçoivent une « seconde chance » (de n'être pas prises comme victime) et leur bit est forcé à zéro. Cet algorithme est également appelé *algorithme de l'horloge* (« clock »), la progression du pointeur étant analogue à celle de l'aiguille d'une horloge. La mise à 1 du bit lors d'une référence peut être réalisée par un mécanisme matériel ou logiciel.

De nombreuses études expérimentales ont permis d'évaluer les algorithmes de remplacement. La figure **??** est une représentation synthétique des résultats obtenus. On constate :

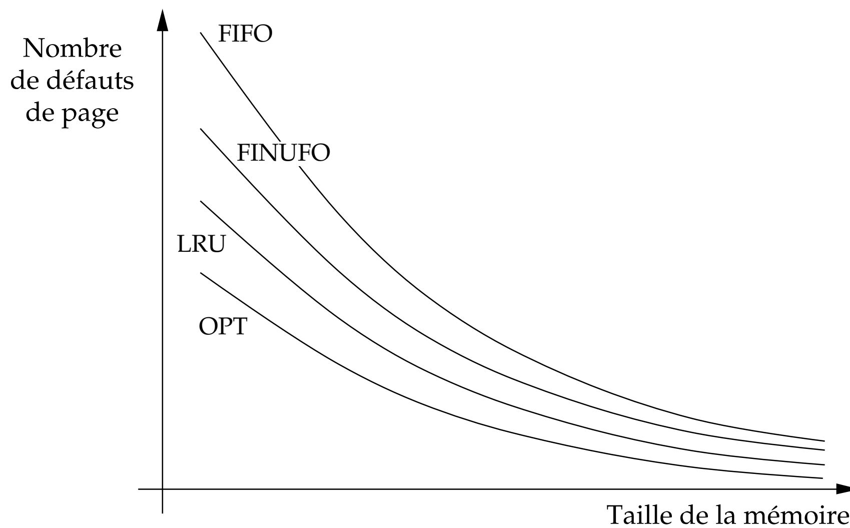


FIG. 7.6 – Performances des algorithmes de remplacement

– que les algorithmes se classent en moyenne (par performances décroissantes), dans l'ordre OPT, LRU, Clock, FIFO (les performances de FIFO sont du même ordre que celles du tirage au hasard),

– que l'influence de la taille de mémoire est très largement supérieure à celle de l'algorithme de remplacement ; autrement dit, on améliore davantage les performances d'un programme en augmentant le nombre de pages physique allouées plutôt qu'en raffinant l'algorithme de remplacement, et cela d'autant plus que les performances initiales sont mauvaises.

7.6 écroulement d'un système de mémoire virtuelle paginée

7.6.1 Apparition de l'écroulement

Sur les premiers systèmes à mémoire virtuelle paginée, on constatait à partir d'une certaine charge (mesurée, par exemple, par le nombre d'usagers interactifs), une dégradation brutale des performances. Ce phénomène, appelé *écroulement* (« thrashing ») se traduit par une chute du taux d'utilisation du processeur et un fort accroissement des échanges de pages ; le temps de réponse atteint des valeurs inacceptables.

Une explication qualitative de l'écroulement permet de mettre en évidence ses causes et de proposer des remèdes. Considérons un système à mémoire paginée, entre un ensemble de processus dont chacun correspond à un usager interactif. La mémoire physique est partagée équitablement entre les processus dont le comportement moyen est supposé identique ; ce partage est mis en œuvre par un algorithme de remplacement global.

Au delà d'un certain nombre de processus (figure **??**), le nombre moyen de pages physiques allouées à chacun d'eux ne permet plus de stocker en mémoire centrale leur ensemble de travail, c'est à dire le sous-ensemble de leur pages virtuelles fréquemment utilisées. La probabilité globale de défaut de page croît dès lors très rapidement avec le nombre de processus.

Si le nombre de défauts de page augmente, le nombre de processus bloqués pour traitement du défaut de page augmente également. Donc le degré de multi-programmation et le taux d'utilisation de la CPU baissent. Face à cette baisse, l'ordonnanceur à long terme peut décider de ramener des processus en mémoire (*swapping in*) pour améliorer le taux d'utilisation de la CPU ! Bien entendu, ces nouveaux venus vont prendre des pages physiques, provoquer des défauts de page, et augmenter le nombre – déjà important – de défauts de page chez les autres processus. C'est un effet « boule

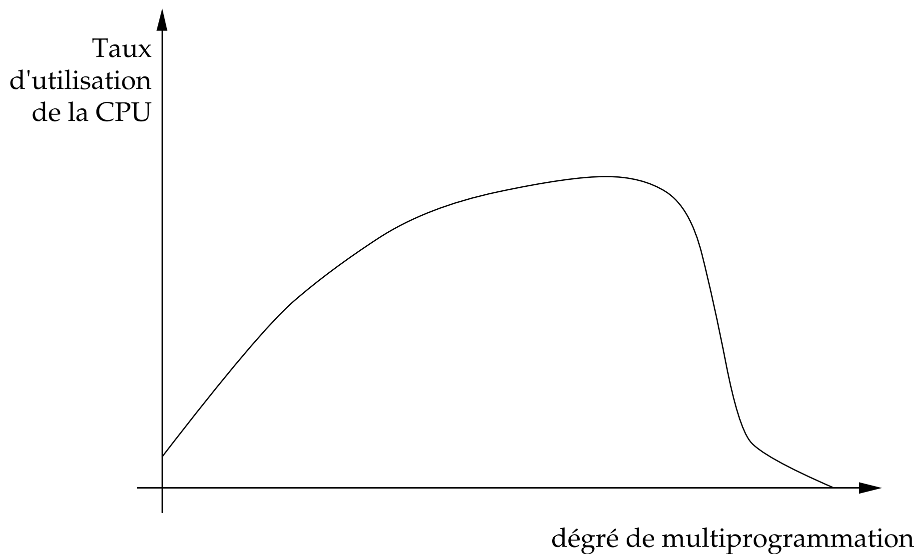


FIG. 7.7 – écroulement d'un système de mémoire virtuelle paginée

de neige » qui entraîne l'écroulement.

Les résultats qui viennent d'être présentés montrent qu'il semble préférable d'essayer d'allouer à chaque programme une taille de mémoire bien adaptée à son comportement, donc variable dynamiquement. Cela peut être tenté de deux façons :

- en utilisant un algorithme à *partition variable* afin de réquisitionner des pages physiques mal utilisées par un processus pour les redistribuer à d'autres processus ;
- en utilisant *une répartition de charge par variation du degré de multi-programmation* afin
  - d'enlever des processus de la mémoire centrale quand il y a surcharge (*swapping out*) ;
  - de faire revenir des processus en mémoire si le couple de ressources (CPU, mémoire) est sous-utilisé (*swapping in*).

  La répartition globale de la charge consiste à agir sur le degré de multiprogrammation pour maintenir les performances du système dans des limites acceptables. A charge faible ou modérée, la multiprogrammation permet d'augmenter le taux d'utilisation du processeur en utilisant les temps morts dus au blocage ou à l'attente de pages ; à forte charge, on voit apparaître l'effet inverse, qui caractérise l'*écroulement*. Ce comportement suggère l'existence d'une valeur optimale du degré de multiprogrammation, qui maximise le taux d'utilisation du processeur pour une configuration matérielle et une charge donnée. Cela est confirmé par l'étude de modèles analytiques et par l'expérience. Un algorithme idéal de régulation de charge devrait maintenir le degré de multiprogrammation au voisinage de cette valeur optimale.

  Plusieurs critères empiriques d'optimalité ont été proposés. Ils reposent tous sur le même principe : tenter de détecter le début de l'écroulement (par la mesure du taux de défauts de page) et maintenir le système au-dessous de ce point critique.

L'expérience montre qu'il est utile de prévoir un certain amortissement pour éviter le pompage (oscillations brutales provoquées par la régulation lorsque la charge est proche du seuil critique). Cela est notamment obtenu en conservant une réserve de pages physiques libres destinées à absorber les variations transitoires du taux de défaut de pages, et en introduisant un temps de retard dans les réactions du régulateur. Un pic transitoire ne provoque donc pas de réaction s'il n'épuise pas la réserve et si sa durée est inférieure au temps de retard. La durée de ce délai et la taille de la réserve doivent être déterminés par l'expérience.

La régulation de charge permet d'utiliser au mieux les ressources disponibles en présence d'une charge donnée. Pour absorber une charge plus élevée, et pour améliorer les performances d'un système, on peut être amené à améliorer les performances du matériel par extension ou remplacement. Il faut alors veiller à ce que la configuration reste équilibrée pour exploiter pleinement ces gains de performance.

Dans tous les cas de figure, le S.E. doit être capable d'évaluer la charge pour prendre la bonne décision et éviter l'apparition d'un écroulement. Cette évaluation se base sur deux méthodes :

- la méthode de *l'ensemble de travail*,
- l'observation de la fréquence d'apparition des défauts de page (PFF ou « *Page Fault Frequency* »).

7.6.2 Algorithme fondé sur l'ensemble de travail

On entretient en permanence pour chaque processus son ensemble de travail ; lors d'un défaut de page, une page virtuelle n'appartenant à aucun des ensembles de travail présents en mémoire est choisie comme victime. S'il n'existe pas de telle page, on réquisitionne les pages physiques qui contiennent l'ensemble de travail du processus le moins prioritaire (la priorité étant déterminée de façon externe ou par le temps de résidence en mémoire). Un processus ne peut recevoir de mémoire que s'il y a assez de pages physiques libres pour recevoir son ensemble de travail courant. La réalisation de cet algorithme nécessite de pouvoir identifier l'ensemble de travail. Aussi, en dehors de réalisations expérimentales, il n'a été mis en œuvre que de façon approchée.

L'estimation de l'ensemble de travail peut se faire par le biais d'un simple bit d'utilisation qui est forcé à 1 lors de chaque référence à une page. Ce bit existe déjà si on utilise un algorithme de remplacement de type FINUFO ou LRU. Périodiquement le S.E. peut, pour chaque page physique,

- recopier ce bit à l'intérieur d'un mot en le décalant, et
- forcer à 0 le bit d'utilisation.

Pour chaque page physique, nous avons donc une suite de bits qui donne un historique d'utilisation de la page physique (i désigne l'intervalle de temps entre deux relevés du bit d'utilisation) :

$t-0i$	$t-1i$	$t-2i$	$t-3i$	$t-4i$	$t-5i$
0	0	1	0	0	1

Cet historique peut facilement être utilisé pour construire l'ensemble de travail. Dans notre exemple, cette page appartient à W(t,T) avec $T \geq 2i$, mais elle n'appartient pas à W(t,T') avec $T' \leq i$.

7.6.3 Algorithme fondé sur la mesure du taux de défaut de page

Une manière indirecte de détecter que le nombre de pages physiques allouées à un processus donné est insuffisant, consiste à mesurer son taux de défaut de page. L'algorithme PFF est fondé sur ce principe ; quand ce taux, dépasse un seuil supérieur, spécifié pour chaque processus, celui-ci reçoit une page physique supplémentaire ; inversement, une page physique lui est retirée si son taux de défaut de page tombe au dessous d'un seuil inférieur.

7.6.4 Exemples

Tous les systèmes multiprogrammés actuels comportent un mécanisme d'ajustement dynamique du degré de multiprogrammation qui utilise l'une ou l'autre des méthodes ci-dessus.

- VAX/VMS est un système utilisant une estimation approchée de l'ensemble de travail.
- IBM VM/370 et Multics sont des systèmes utilisant une régulation de charge globale.

Chapitre 8

Parallélisme et synchronisation

Des processus peuvent se dérouler successivement sur un même processeur ou bien en simultanéité réelle sur des processeurs distincts, à condition d'assurer la communication des données et résultats entre eux.

Le but de ce chapitre est l'étude des processus, des ressources qu'ils utilisent et de leur mise en oeuvre sur différents types de machines. Nous étudieront successivement le parallélisme simulé sur machine traditionnelle, puis le parallélisme réel des architectures nouvelles.

8.1 Exécution de processus

8.1.1 Quelques rappels et définitions

Une *ressource* est une entité pouvant servir à l'exécution d'un travail. Les organes de la machine : Unité centrale, Mémoire centrale, Périphériques sont des ressources ; certaines entités logiques aussi, comme par exemple les fichiers.

Un *processus* est une suite d'instructions réalisant une opération que l'on considère comme élémentaire relativement à son algorithme et à l'ensemble des ressources qu'elle nécessite. C'est une structure séquentielle. Il ne faut pas la confondre avec l'appel de procédure qui suspend l'appelant, et n'augmente pas le degré de parallélisme. Une *application* est un ensemble de processus *coopérant* et/ou en *compétition* pour l'acquisition de ressources.

Un processus nécessite un processeur pour s'exécuter. Pour plusieurs processus concurrents, il faut plusieurs processeurs (*parallélisme vrai*) ou bien un seul qu'ils se partageront dans le temps (*parallélisme simulé*).

8.1.2 état des processus

Les ressources étant en nombre limité, il n'est pas possible de les attribuer globalement à un processus au moment de sa création. On peut donc arriver à la situation dans laquelle un processus ne possède pas toutes les ressources nécessaires pour exécuter l'instruction suivante. On dit que ce processus est *bloqué*. C'est le cas par exemple dans un système paginé lorsqu'un programme requiert une page non présente en mémoire centrale. Dans le cas contraire, il est dit *actif*.

Lorsque dans un système, plusieurs processus coopèrent à la réalisation d'un même travail, l'un d'entre eux peut se trouver dans l'impossibilité de progresser pour une raison logique : l'attente d'un signal en provenance d'un autre processus. Par exemple, le processus P élabore un résultat que le processus Q utilise ; Q ne peut s'exécuter que lorsque P a produit ce résultat. On distingue donc deux types de blocage :

– Technologique pour l'absence de ressource ;

– Intrinsèque pour la synchronisation.

Du point de vue du système, il est commode de considérer comme des ressources les signaux de synchronisation : le blocage ne se produit plus alors que pour l'absence d'une ressource.

Au contraire, du point de vue du programmeur, qui ne se préoccupe que du blocage intrinsèque, il est agréable d'envisager que chaque processus s'exécute sur une *machine virtuelle* qui comprend toutes les ressources nécessaires à son achèvement. La correspondance entre les ressources de chaque machine virtuelle et les ressources physiques est laissée à la charge du système.

8.1.3 Accès aux ressources

Une ressource est dite *locale* à un processus s'il est seul a pouvoir l'utiliser. Elle doit obligatoirement disparaître à la fin de l'exécution de ce processus. Une ressource qui n'est locale à aucun processus est dite *commune*. Une ressource est dite *partageable* avec n points d'accès $(n > 1)$ si cette ressource peut être attribuée, au même instant, à n processus au plus.

Un ensemble de processus peut :

– entrer en *compétition* pour l'accès à une ressource ou à une information partagée ;

– fonctionner en *coopération* pour mener à bien une application.

Dès l'instant où deux processus ont une ressource en commun, l'ordre dans lequel ils s'exécutent n'est pas indifférent. C'est le problème de la *synchronisation*. Lorsqu'ils échangent des données, on parle de *communication*. Ce chapitre traite de ces problèmes en *architecture centralisée*, i.e. avec mémoire commune à tous les processus.

8.2 Synchronisation de processus

8.2.1 Exclusion mutuelle

C'est le problème qui se pose lorsqu'une ressource ne peut appartenir qu'à un seul processus à la fois, et que plusieurs processus concurrents en ont besoin pour se dérouler. Par exemple :

– partage d'une imprimante (ressource physique) ;

– accès à un fichier en lecture pour n utilisateurs simultanément, mais en écriture pour un seul (ressource logique).

Dans le premier exemple, une variable booléenne du système indiquera si l'imprimante est libre ou non. Il ne faut pas que deux processus puissent lire la valeur de cette variable simultanément, la trouver vraie, et s'approprier donc tous deux l'imprimante.

8.2.2 Programmation de l'exclusion mutuelle

Une ressource non partageable (simultanément) est dite *critique*. Toute séquence de programme qui utilise une ressource critique est dite *section critique* (S.C.). Les vitesses de calcul des différents processus sont inconnues. On supposera seulement que tout processus qui entre en section critique en sort au bout d'un temps fini (pas de blocage).

La solution doit vérifier certaines conditions :

– **Exclusion** : à tout instant, un processus au plus est en section critique ;

– **Accès** : si plusieurs processus sont bloqués à l'entrée d'une section critique libre, l'un d'eux doit y entrer au bout d'un temps fini ;

– **Indépendance** : le blocage par cette section critique doit être indépendant des autres types de blocage, donc si un processus est bloqué hors de la section critique, il ne doit pas empêcher un autre processus d'y entrer ;

– **Uniformité** : aucun processus ne doit jouer de rôle privilégié ; ils doivent tous utiliser les mêmes mécanismes.

La programmation de l'exclusion passe par trois sections de code. La première (*initialisation*) est exécutée une seule fois (par le S.E. ou par les processus qui désirent se synchroniser). Elle prépare les variables d'état qui indique que la section critique est libre. La seconde (*prologue*) effectue la demande d'entrée en S.C. Si l'entrée est impossible, le processus est bloqué au niveau du prologue. La dernière (*épilogue*) signale la fin de la S.C. ce qui permet d'en autoriser l'accès à un autre processus qui est en attente.

⟨initialisation⟩

⟨prologue⟩
⟨section critique⟩
⟨épilogue⟩

8.2.3 La méthode des drapeaux

Pour réaliser l'exclusion mutuelle une première solution consiste à bloquer les processus qui veulent entrer en S.C. (dans le prologue) si celle-ci n'est pas libre. Pour programmer ce blocage, on se donne une variable booléenne partagée et nous obtenons le code suivant :

```
⟨initialisation⟩            libre := vrai ;

⟨prologue⟩          (1) tant que (libre = faux) faire
                    (2) fin faire
                    (3) libre := faux

⟨épilogue⟩                  libre := vrai
```

Partons du principe qu'aucun processus n'est en S.C. (donc `libre = vrai`) et que deux processus (notés p_1 et p_2) désirent entrer en S.C. Il est possible que l'exécution du prologue se déroule ainsi

$$(1_{p_1},\ 2_{p_2},\ \langle\text{interruption}\rangle,\ \ldots,\ 1_{p_2},\ 2_{p_2},\ 3_{p_2},\ \ldots,\ \langle\text{interruption}\rangle,\ 3_{p_1})$$

On voit bien que p_1 n'est pas eu le temps de modifier la valeur du drapeau ce qui provoque l'entrer simultanée des deux processus en S.C. Ce mécanisme est donc mis en défaut par l'apparition des *interruptions d'horloge* dans les systèmes en *temps partagé*.

Pour corriger ce problème on pourrait envisager de *masquer les interruptions* mais cette solution n'est pas envisageable pour deux raisons :

– D'une part, le blocage des interruptions n'est possible qu'en mode maître ce qui interdit son utilisation dans des processus utilisateur.

– D'autre part, ce blocage ne règle le problème que sur les machines mono-processeur. En effet, si plusieurs processeurs sont disponibles, nous pouvons avoir la séquence suivante :

Exécution de p_1 sur la CPU 1 : $(1_{p_1}, 2_{p_1}, 3_{p_1})$
Exécution de p_2 sur la CPU 2 : $(1_{p_2}, 2_{p_2}, 3_{p_2})$

Les deux processus entrent donc simultanément en S.C. En fait, le problème vient du fait que les deux opérations (test du drapeau et modification) ne sont pas réalisées de manière atomique.

8.2.4 L'algorithme de peterson

On envisage alors de signaler d'abord que le processus demande l'accès à la S.C. en affectant un booléen, puis y entre effectivement s'il n'y a pas de conflit d'accès, i.e. si l'autre processus n'a pas signalé aussi son intention d'entrer en S.C. Trois phases sont à considérer pour l'entrée en S.C. :

1. pas de demande d'accès ;
2. demande d'accès non encore achevée ;
3. accès effectif : le processus est en S.C.

Les difficultés surviennent lorsque deux processus sont simultanément dans la phase 2. Chaque processus doit pouvoir déterminer dans quelle phase se trouve l'autre. On utilisera un booléen pour chaque processus, qu'il affectera en entrant dans la phase 2 :

demande[i] := vrai ;

Les deux processus peuvent effectuer simultanément ces instructions, aussi l'accès en S.C. ne doit être effectif que s'il n'y a pas de conflit, i.e. si l'autre n'a rien signalé. Sinon, il faut donner une priorité à l'un des processus, et mettre l'autre en attente. S'il n'y a pas de conflit, la priorité n'est pas prise en compte, et n'interdit donc pas l'accès à la S.C. En cas de conflit, le processus prioritaire entre en S.C., l'autre dans une boucle d'attente contrôlée par le mécanisme de priorité. Lorsqu'un processus sort de la section critique, il donne systématiquement la priorité à l'autre.

Solution programmée pour le processus i (algorithme de Peterson) : on se donne les variables suivantes :

var tour : **entier** ;
    demande : **tableau** [0 .. 1] de **booléen** ;

Les trois sections de code permettant de programmer l'exclusion mutuelle pour le processus i (avec $0 \leq i \leq 1$) s'écrivent de la manière suivante :

⟨init⟩ tour := 0 ;
demande := (faux, faux) ;

⟨prologue⟩ demande[i] := vrai ;
tour := $1 - i$;
répéter
jusqu'à (tour = i) **ou** (demande[$1 - i$] = faux)

⟨épilogue⟩ demande[i] := faux ;

Cet algorithme sera étudié plus précisément en travaux dirigés.

8.2.5 Dispositifs de synchronisation câblés

Pour régler ces problèmes, et pour gagner en efficacité, de nouvelles instructions ont été ajoutées au processeur de manière à faciliter l'écriture de programme qui s'exécutent en exclusion mutuelle. Nous allons détailler maintenant l'instruction TAS « Test And Set » qui est définie de la manière suivante :

```
instruction TAS m, verrou
début
   ⟨bloquer la case mémoire mem[verrou]⟩
   mem[m] := mem[verrou]
   mem[verrou] := 0
   ⟨débloquer la case mémoire mem[verrou]⟩
   CO := CO + ⟨taille de l'instruction TAS⟩
fin
```

Cette instruction permet de sauvegarder (dans la case mémoire m) et de modifier le contenu de la case verrou en une seule opération de la C.P.U. Dans un système mono-processeur, l'interruption d'horloge ne pose plus de problème puisqu'elle a lieu soit **avant** soit **après** l'exécution de TAS mais jamais au milieu.

Soit p une variable entière (propre à chaque processus) et mutex un variable entière partagée par les processus qui désirent entrer en S.C. L'entrée en S.C. est réalisée par :

```
⟨init⟩        mutex := 1 ;

⟨prologue⟩    répéter
                  TAS p, mutex ;
              jusqu'à (p = 1)

⟨épilogue⟩    mutex := 1 ;
```

Pour programmer l'exclusion mutuelle d'accès à une ressource, on a utilisé un dispositif câblé d'exclusion mutuelle à mutex. Cette méthode n'est pas utilisable si la séquence critique est longue, car elle monopolise le processeur sur l'attente active.

8.2.6 Verrou

Pour libérer la CPU, on utilise une file d'attente dans laquelle on place tout processus demandeur ne pouvant entrer en section critique. Lorsqu'un processus sort de la S.C., un processus de la file est activé si celle-ci est non vide. Un *verrou* v est un couple (d, f) dans lequel d est un booléen et f une file d'attente de processus.

```
procédure init(var v : verrou)
début
   v.d := faux ;
fin

procédure verrouiller(var v : verrou)
début
   si (v.d = vrai) alors
      ⟨entrer le processus appelant dans la file v.f⟩
      ⟨suspendre le processus appelant⟩
   sinon
      v.d := vrai ;
   fin si
fin
```

```
procédure déverrouiller(var v : verrou)
début
   si ⟨la file v.f est vide⟩ alors
      v.d := faux ;
   sinon
      ⟨sortir un processus Q de la file v.f⟩
      ⟨réveiller le processus Q⟩
   fin si
fin
```

Pour programmer l'exclusion mutuelle entre plusieurs processus on se donne un verrou `mutex` qui est accessible à tous les processus et les trois sections de code deviennent :

```
⟨init⟩        init(mutex) ;

⟨prologue⟩    verrouiller(mutex) ;

⟨épilogue⟩    déverrouiller(mutex) ;
```

Le verrou est une ressource critique qu'il faut protéger. Les deux procédures `verrouiller` et `déverrouiller` sont des sections critiques, elles font partie du système et se comportent pour l'utilisateur comme des instructions. On les appelle donc primitives. Pour réaliser leur exclusion mutuelle :

- si la machine est monoprocesseur, il suffit de masquer les interruptions ;
- si elle est multiprocesseur, le masquage des interruptions est inopérant ; on utilise alors une instruction TAS. L'attente active ne dure que le temps d'exécution de la primitive.

8.2.7 Sémaphores

Un ensemble de tampons est une ressource critique à plusieurs points d'entrée. On généralise les verrous en sémaphores. Un *sémaphore* s est un couple (c, f) constitué d'une variable entière $s.c$ et d'une file d'attente $s.f$ de processus. Il est initialisé par :

$$s = (n, \{\}) \quad \text{avec} \quad n \geq 0$$

Les primitives sont :

```
procédure P(var s : sémaphore)
début
   s.c := s.c − 1
   si (s.c < 0) alors
      ⟨entrer le processus appelant dans la file s.f⟩
      ⟨suspendre le processus appelant⟩
   fin si
fin

procédure V(var s : sémaphore)
début
   s.c := s.c + 1
   si (s.c ≤ 0) alors
      ⟨sortir un processus Q de la file s.f⟩
      ⟨reprendre le processus Q⟩
   fin si
fin
```

Soient : n_p le nombre de primitives P(s) exécutées depuis l'initialisation ; n_v le nombre de primitives V(s) ; n_f le nombre de processus ayant franchi P depuis l'initialisation et c_0 la valeur initiale du sémaphore. Si les primitives P et V sont seules à modifier la valeur de $s.c$, on a toujours :

$$s.c = c_0 - n_p + n_v$$

Les processus ayant franchi P sont ceux qui n'ont pas été bloqués, ou qui ont été débloqués depuis. On a donc toujours la relation :

$$n_f = \min(n_p, c_0 + n_v)$$

– Si aucun processus n'a encore effectué V, $n_v = 0$.

– Si $(n_p < c_0)$ la demande de ressource a été inférieure aux disponibilités, donc tous les demandeurs sont passés. Ainsi, $n_f = n_p = \min(n_p, c_0)$.

– Si $(n_p > c_0)$, les c_0 premiers sont passés, les $(n_p - c_0)$ autres ont été bloqués. Donc c_0 demandeurs sont passés. Donc $n_f = c_0 = \min(n_p, c_0)$. Par conséquent, si $n_v = 0$, la relation est vraie.

– Si maintenant $(n_v \neq 0)$, par définition les primitives P et V sont inséparables, et à toute exécution de V correspond une exécution de P (par le même processus ou par un autre). La différence $(n_p - n_v)$ représente le nombre de processus engagés dans la section critique, ou bloqués sur le sémaphore. n_v est le nombre de processus ayant exécuté P, puis V ; donc

  $$n_f = n_v + \langle \text{nombre de processus encore dans la section critique} \rangle$$

  Sur les $(n_p - n_v)$ processus demandeurs, e_0 avaient le droit de passer. Donc il en est passé : $\min(n_p - n_v, c_0)$. Par conséquent,

  $$n_f = n_v + \min(n_p - n_v, c_0) = \min(n_p, c_0 + n_v).$$

Si on utilise un sémaphore pour réaliser l'exclusion mutuelle, on l'initialise à 1. On vérifie à l'aide de la relation précédente qu'il permet l'exclusion mutuelle, car le nombre de processus actuellement en section critique est :

$$n_f - n_v = \min(n_p - n_v, 1) \leq 1.$$

Un processus au plus est donc en section critique. De même, si aucun processus n'est en section critique :

$$n_f - n_v = 0 = \min(n_p - n_v, 1)$$

donc $n_p - n_v = 0$ et $n_p = n_v$, aucun processus n'est bloqué. Certaines précautions sont nécessaires pour utiliser les sémaphores et les verrous :

– leur modification ne doit se faire que par les primitives P et V, et donc ils ne doivent pas faire partie de l'espace adressable de l'utilisateur ;

– si un processus est détruit en section critique, le système doit veiller à libérer la section critique.

– enfin, les primitives n'indiquent pas l'ordre dans lequel les processus en attente sont activés. Dans le cas où l'on utilise des priorités, rien n'empêche une coalition de processus de forte priorité d'en bloquer d'autres indéfiniment.

8.2.8 Sémaphores privés

Un sémaphore s est un *sémaphore privé* d'un processus si seul ce processus peut exécuter P(s), les autres processus ne pouvant agir que par V(s). Les sémaphores privés sont utilisés pour programmer l'envoi, l'attente et la reception d'un signal par un processus.

Si un processus doit attendre un signal d'un autre, il se bloque par P(s) derrière son sémaphore privé initialisé à 0. Si le processus P_1 est actif lorsque P_2 exécute V(s), s étant un sémaphore privé de P_1, l'état de P_1 est inchangé, mais la valeur du sémaphore a augmenté d'une unité, et le processus P_1 ne se bloquera pas à la prochaine primitive P(s). Le signal est donc mémorisé. Ce mécanisme permet de mémoriser n signaux successifs.

On peut combiner les sémaphores d'exclusion mutuelle et privés. Pour se poursuivre, un processus a besoin de tester des variables d'état modifiées par d'autres processus. Il ne peut faire le test qu'à l'intérieur d'une section critique. Mais il ne doit pas s'y bloquer. Il faut donc qu'à l'intérieur de la section critique il se donne un droit de passage par V(s), s étant un sémaphore privé de P_1, dans le cas où il ne doit pas se bloquer, rien sinon. A la sortie de la section critique, il utilise son droit de passage éventuel par P(s). S'il y a eu V(s), il passe, sinon il se bloque hors de la section critique.

P_1 : P(*mutex*)
si ⟨le blocage est inutile⟩ **alors**
V(*sempriv*)
fin si
V(*mutex*)
P(*sempriv*)

séquence exécutée par P_1 ; *sempriv* appartient à P_1. L'activation par un autre processus s'écrit :

P_2 : P(*mutex*)
si ⟨le processus P_1 doit être débloqué⟩ **alors**
V(*sempriv*)
fin si
V(*mutex*)

8.2.9 Modèle du producteur et du consommateur

Ce modèle est basé sur les deux points suivants : (1) aucune hypothèse n'est faite sur les vitesses relatives des processus ; (2) un tampon a une capacité de n messages, $n > 0$. L'algorithme du producteur est le suivant :

répéter
⟨produire un message⟩
⟨le déposer dans le tampon⟩
jusqu'à ...

L'algorithme du consommateur est

répéter
⟨prélever un message depuis le tampon⟩
⟨le consommer⟩
jusqu'à ...

Règles :

- exclusion mutuelle au niveau du message : le *Consommateur* ne peut prélever un message que le *Producteur* est en train de ranger ;

- le *Producteur* ne peut pas placer un message si le tampon est plein ;
- le *Consommateur* doit prélever tout message une fois et une seule ;
- si le *Producteur* est en attente parce que le tampon est plein, il doit être prévenu dès que ceci n'est plus vrai. De même pour le *Consommateur* avec tampon vide.

```
NPlein :  sémaphore  = (0, {})
NVide :   sémaphore  = (n, {})
```

Le producteur s'écrit

répéter
  P(NVide) ;
  ⟨produire un message⟩
  ⟨le déposer dans le tampon⟩
  V(NPlein) ;
jusqu'à ...

et le consommateur :

répéter
  P(NPlein) ;
  ⟨consommer⟩
  V(NVide) ;
jusqu'à ...

- $(\texttt{NPlein} \geq 0)$, indique le nombre de messages disponibles dans le tampon ;
- $(\texttt{NPlein} < 0)$, indique que le consommateur attend un message ;
- $(\texttt{NVide} \geq 0)$, indique le nombre de tampons libres ;
- $(\texttt{NVide} < 0)$, indique que le producteur attend un tampon libre.

8.3 Communication entre processus

La coopération de plusieurs processus nécessite en général l'échange d'informations. Dans tout ce qu'on vient de voir, le seul message transmis est l'autorisation de continuer. On considère maintenant un problème d'échange d'information orienté constitué de deux processus spécialisés, l'un émettant des messages successifs à son rythme propre, l'autre utilisant ces messages. L'utilisation peut être lente pour certains messages, rapide pour d'autres ; aucune hypothèse n'est faite sur les vitesses, et l'ensemble ne fonctionnera que si les deux processus sont correctement synchronisés.

8.3.1 Sémaphore avec message

Pour résoudre le problème de la communication d'informations, on peut modifier les primitives P et V de telle sorte que V, qui donne l'autorisation de continuer, transmette en même temps un message que P reçoit.

A chaque exécution de V, un nouveau message est transmis, qui doit être placé dans une file d'attente. Celle-ci n'évolue pas comme la file des processus qui est remplie par P. On appelle donc f_p la file des processus et f_m la file des messages. Les nouvelles primitives P_m et V_m (aussi appelées *send* et *receive*) sont :

procédure `Pm(`**var** s `: sémaphore avec message ;` **var** m `: message)`
début
$s.c$:= $s.c - 1$
si $(s.c < 0)$ **alors**
⟨entrer le processus appelant dans la file $s.f_p$⟩
⟨suspendre le processus appelant⟩
sinon
⟨sortir m de la file $s.f_m$⟩
fin si
fin

procédure `Vm(`**var** s `: sémaphore à message ;` m `: message)`
début
$s.c$:= $s.c + 1$
si $(s.c \leq 0)$ **alors**
⟨sortir un processus Q de la file $s.f_p$⟩
m(Q) := m ;
⟨reprendre le processus Q⟩
sinon
⟨entrer m dans la file $s.f_m$⟩
fin si
fin

La gestion de messages de longueur variable est difficile, aussi ce mécanisme impose en général une taille fixe. Le message peut être par exemple une adresse échangée par les deux processus. Ceci permet en particulier de résoudre simplement le partage d'un ensemble de tampons dans une relation Producteur-Consommateur : le producteur passe au consommateur le message contenant l'adresse du tampon où se trouve l'objet à consommer.

Chapitre 9

Évaluation de performance des systèmes

9.1 Introduction

Comme on le verra tout au long des chapitres qui vont suivre, l'étude d'un système informatique présente un grand nombre de choix possibles dans la réalisation certaines de ses parties. A cela vient s'ajouter la prise en compte du type d'application qui sera traité par le système selon le site où il sera implanté. On peut ainsi désirer obtenir des réponses à des questions du style :

– Le taux d'utilisation de l'UC sur le site S_1 sera-t-il supérieur à celui du site S_2 ?

– Le temps de réponse serait-il modifié si l'on augmentait la mémoire de 32 méga-octets ?

– La taille totale de la mémoire physique paginée demeurant la même, quel est le couple « taille de page / nombre de pages » optimum ?

– Quel est le nombre moyen de programmes en attente

Nous allons voir dans un premier temps l'ensemble des possibilités d'évaluation.

9.2 Les méthodes d'évaluation

Sur un système déjà en ordre de marche, on va pouvoir l'évaluer grâce à des *moniteurs matériels* et *logiciels*. Les premiers comportent un ensemble de sondes, une unité de traitement et des unités de stockage pour mémoriser les mesures. Les sondes sont « piquées » en divers points de la machine à étudier afin de pouvoir capter les différents signaux qui transitent. Le processeur peut sélectionner les mesures et éventuellement opérer un pré-traitement en temps réel, mais le traitement global est le plus souvent effectué hors ligne lorsque la campagne de mesures est terminée.

Les difficultés de cette méthode sont entre autres :

– la très grande quantité de mesures à traiter et à interpréter ;

– la pose des sondes aux endroits choisis ne se fait pas sans risque de court-circuit, d'autant plus que l'intégration grandissante des éléments ne facilite pas la tâche ;

– Le matériel de test est très onéreux et d'une utilisation très délicate : seuls les grands constructeurs, quelques grands utilisateurs ou des sociétés spécialisées disposent de moniteurs matériels.

Par contre, les résultats des mesures ne sont pas perturbés et de ce fait très fiables.

Les moniteurs logiciels effectuent eux aussi des mesures, mais grâce à un *espion logiciel* situé dans le moniteur du système à évaluer. Chaque fois que se produit un événement significatif, l'espion procède à un déroutement pour prendre le compte et mémoriser les valeurs correspondantes. Là

encore, l'utilisation de ces moniteurs logiciels est réservée à des spécialistes capables de mettre en place l'espion (écriture dans le moniteur), d'interpréter les mesures et surtout, le plus délicat, d'apprécier la déformation des résultats due à l'utilisation de la machine par l'espion lui-même. En particulier, il s'avère très difficile de déduire des valeurs maxima d'utilisation puisque l'espion requiert lui-même de plus en plus de temps.

Les deux méthodes que l'on vient de présenter permettent une évaluation du système dans des conditions de charge précises. Il est donc très difficile d'extrapoler sur ce que deviendraient les performances si on augmentait la capacité mémoire, si on ajoutait une entrée/sortie...

Une autre façon d'évaluer les performances d'une machine consiste à la comparer à une ou plusieurs autres. Dans ce cadre, les « benchmarks » trouvent leur utilité. En général, les comparaisons médiatiques entre différentes machines sont faites à partir de ces pseudo-caractéristiques. Mais il faut savoir que ces benchmarks sont conçus par les constructeurs et de ce fait peuvent être particulièrement bien adaptés à leur matériel et à l'opposé, très peu à celui des concurrents !... Toutefois, pour des utilisateurs ayant des besoins précis, la consultation de ces tests peut favorablement influencer leur choix.

Autre méthode employée, la modélisation permet d'évaluer un système par l'intermédiaire d'un modèle le décrivant plus ou moins précisément. Ce modèle est résolu soit par des méthodes analytiques regroupant des techniques mathématiques autour d'un système de files d'attente, soit par des simulations. Dans le premier cas, si la solution est une formule mathématique, les calculs seront simples et de nombreux cas de figure pourront être étudiés (optimisation par utilisation des dérivées). Dans le cas de la simulation d'un modèle réalisé, la finesse de description du modèle aura une incidence directe sur le rapport entre l'unité de temps de la machine simulée et celle de la machine étudiée, rapport pouvant varier de 0.01 à 100. Ne pouvant s'appuyer sur des expressions mathématiques comme dans le cas précédent, l'optimisation sera donc plus compliquée.

A cause de leur faible coût, de leur facilité de mise en œuvre et de la rapidité d'obtention de résultats, les méthodes analytiques sont de plus en plus utilisées. On s'oriente à présent vers l'élaboration de progiciels d'évaluation de performance rendant accessibles à des non-initiés les méthodes très sophistiquées de la modélisation

9.3 La modélisation et les méthodes de résolution

La modélisation consiste à passer d'un système concret connu par ses spécifications à un modèle qui est une simplification du système à évaluer. Il utilise plusieurs types de représentation : réseaux de Pétri, stochastiques ou non, réseaux de files d'attente, combinaison des deux... Aucune théorie concernant le passage du système au modèle n'existe ; seule l'expérience permet d'aboutir à un « bon modèle ». En particulier, la simplification nécessaire pour obtenir un modèle simple, risque d'impliquer des écarts importants avec la réalité, et ceux-ci ne pourront être chiffrés qu'approximativement avec l'habitude.

En fait, les modèles sont généralement conçus de façon à ce qu'un écart de valeur de paramètre ne change pas fondamentalement le résultat. Par exemple, les valeurs optimales sont le plus souvent excellentes, la variations des performances s'opérant dans le bon sens (sous-évaluation).

Si le modèle est trop complexe, on a recours à un *simulateur* écrit soit dans un langage universel (Fortran, Pascal, ...) qui lui confère la rapidité des calculs mais aussi des difficultés de mise au point, soit dans un langage adapté (Simula, GPSS, ...). Un générateur de nombres aléatoires est nécessaire afin d'engendrer les différents tirages des lois de service ou d'arrivée. De plus, ces générateurs doivent être suffisamment « longs » afin d'éviter des cycles. La simulation permet de décrire exactement le modèle choisi, mais il faut tenir compte du fait qu'à l'erreur de modélisation dont nous avons déjà parlé, vient s'ajouter l'erreur du simulateur.

Les *méthodes analytiques* limitent beaucoup plus les modèles envisageables. Si l'on choisit la discipline « premier arrivé, premier servi », il ne faut utiliser que des lois exponentielles et les différentes

classes de clients (programmes) doivent avoir la même moyenne de temps de service aux guichets d'une station (organes de la machine). Dans le cas de disciplines moins usitées (infinité de serveurs, service temps partagé ou discipline « dernier arrivé, premier servi »avec priorité absolue), on peut choisir pour chaque classe de clients une loi générale de service.

Avec la puissance des machines de traitement, on fait également de plus en plus appel aux *méthodes numériques* dans lesquelles toutes les transitions possibles sont décrites (résolution de la matrice des transitions). Dans ce cadre, les réseaux de Pétri prennent une importance de plus en plus grande à cause de la manière simple d'obtenir l'ensemble des transitions

9.4 Conclusion

L'évolution depuis le début des années 80 consiste à produire des progiciels intégrant les principales méthodes de résolution analytique, un simulateur et un programme qui, suivant le modèle proposé, adopte le bon mode de résolution. Ils comportent d'autre part un langage de description du modèle. De plus, une interaction avec l'utilisateur permet de le conseiller de simplifier telle ou telle partie afin de pouvoir utiliser un méthode analytique à la place d'une simulation.

A l'heure actuelle, l'effort porte sur l'interface homme-machine pour la description du modèle. En particulier, grâce à des interfaces graphiques, on pourra dans quelques temps dessiner sur un écran tactile le cheminement des clients et les caractéristiques des serveurs.

Il ne faut cependant pas croire que ces logiciels vont permettre une « démocratisation » complète des outils d'évaluation En effet, malgré les aides apportées à la modélisation, celle-ci est constituée d'un ensemble d'heuristiques plus ou moins heureuses, et seul un spécialiste pourra dégager les mieux adaptées et estimer les compromis provoqués par le passage du système au modèle. Un effort de recherche à été entrepris dans cette partie de conception (au moins aussi importante que la partie résolution). Un système expert proposant des modèles pré-enregistrés de différents sous-ensembles classiques (unité centrale, périphériques, réseaux, multiprocesseurs) permet à l'utilisateur la mise en place du modèle le plus sophistiqué possible tout en restant pour la plupart des sous-modèles dans le cadre de résolutions analytiques.

Beaucoup de travaux restent à faire, aussi bien dans la modélisation que dans l'interprétation des résultats. De plus la recherche de nouvelles méthodes analytiques permettra la prise en compte de modèles de plus en plus précis sans pour autant augmenter les temps de conception et de calcul.

Bibliographie

SG94 "*Principes des systèmes d'exploitation*",
A. Silberschatz & P.B. Galvin, Addison-Wesley, 1994.

TE94 "*Systèmes d'exploitation*",
A. Tanenbaum, Prentice Hall, InterEditions, 1994.

BB90 "*Systèmes d'exploitation, concepts et algorithmes*",
J. Beauquier & B. Bérard, Mc Graw-Hill, 1990.

KR87 "*Principes des systèmes d'exploitation des ordinateurs*",
S. Krakowiak, Dunod Informatique, 1987.

BA89 "*Conception du système UNIX*",
M.J. Bach, Masson, Prentice-Hall, 1989.

Table des matières