14

La gestion de la mémoire centrale

A l'origine, la mémoire centrale était une ressource chère et de taille limitée. Elle devait être gérée avec soin. Sa taille a considérablement augmenté depuis, puisqu'un compatible PC a souvent aujourd'hui la même taille de mémoire que les plus gros ordinateurs de la fin des années 60. Néanmoins, le problème de la gestion reste important du fait des besoins croissants des utilisateurs.

14.1. La notion de multiprogrammation

La multiprogrammation a été présentée dans le premier chapitre comme un moyen pour améliorer l'efficacité de l'utilisation du processeur. Un processus est alternativement dans des phases actives, pendant lesquelles il évolue, et dans des phases d'attente d'une ressource, pendant lesquelles il n'a pas besoin du processeur. Dans un système *monoprogrammé*, la mémoire centrale est découpée en deux parties: une partie est réservée au système et le reste est attribué au processus. Lorsque le processus est en attente de ressource, le processeur n'a rien à faire. La *multiprogrammation* consiste à découper la mémoire centrale en plusieurs parties, de façon à mettre plusieurs processus en mémoire au même moment, de telle sorte à avoir plusieurs processus candidats au processeur.

14.1.1. L'intérêt de la multiprogrammation

Supposons que deux processus P_1 et P_2 soient présents en mémoire et que leur comportement corresponde à l'alternance de 100 ms de processeur et 100 ms d'attente d'entrées-sorties. On voit que le processeur va pouvoir faire évoluer P_1 pendant 100 ms jusqu'au lancement de son entrée-sortie, puis faire évoluer P_2 pendant 100 ms jusqu'au lancement de la sienne. A ce moment, l'entrée-sortie de P_1 étant terminée, le processeur pourra le faire évoluer pendant une nouvelle période de 100 ms, et ainsi de suite. De la sorte, d'une part, les deux processus travaillent à leur vitesse maximum, d'autre part, le processeur est utilisé à 100% au lieu de 50% en monoprogrammation.
La situation ci-dessus est en fait idéale. Les durées d'activité ou d'attente d'entrées-sorties des processus ne sont pas en fait constantes. Si la situation est améliorée, c'est-à-dire, si le processeur est mieux utilisé, il faut mettre, dans cet exemple, plus de deux processus en mémoire pour approcher un taux d'activité du processeur de 100%. La figure 14.1 donne le taux d'activité du processeur que l'on obtient en fonction du nombre de processus présents en mémoire, pour différents taux d'attente d'entrées-sorties de ces processus. Le nombre de processus présents en mémoire est encore appelé *degré de multiprogrammation*. En particulier, ces courbes montrent qu'il faut 5 processus ayant un taux d'entrées-sorties de 50% pour approcher 97% du taux d'activité du processeur.

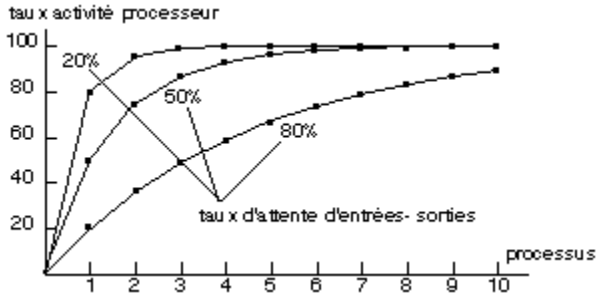

Fig. 14.1. *Taux d'activité du CPU en fonction du nombre de processus.*

La justification de ces courbes est probabiliste. Le taux d'attente d'entrées-sorties peut s'interpréter comme la probabilité que le processus soit en attente d'entrées-sorties à un instant donné. Si le degré de multiprogrammation est n, la probabilité pour que tous les processus soient en attente d'entrées-sorties est alors p^n. Il s'ensuit que la probabilité pour qu'un processus au moins soit prêt, et donc que le processeur soit actif est: `taux_processeur = 1 - p`^{n}. Ces courbes sont le reflet de cette relation.
Notons que lorsque le taux d'activité est de 20%, par exemple dans le cas d'une alternance de 10 ms de processeur et de 40 ms d'entrées-sorties, il faut 10 processus pour atteindre un taux d'activité de 90%. Or les applications de gestion ont souvent un taux d'activité inférieur à une telle valeur, ce qui justifie dans ce cas un degré de multiprogrammation important pour avoir une bonne utilisation du processeur.

14.1.2. Les conséquences de la multiprogrammation

La multiprogrammation a des conséquences sur la gestion de la mémoire centrale. On peut distinguer trois problèmes que doit résoudre le système.

- Définir un espace d'adresses par processus. Chaque processus doit conserver son indépendance. Il doit pouvoir créer des objets dans son espace d'adresses et les détruire, sans qu'il y ait interférence avec les autres processus. Les caractéristiques du matériel sont importantes, car elles offrent plus ou moins de souplesse pour cette définition.
- Protéger les processus entre eux. Il est nécessaire que “chacun reste chez soi”, du moins pour les objets qui leur sont propres. Il faut donc pouvoir limiter les actions des processus sur certains objets, en particulier interdire les accès à ceux qui ne leur appartiennent pas.
- Attribuer un espace de mémoire physique. Comme un objet n'est accessible au processeur que s'il est en mémoire physique, il faut “allouer” de la mémoire physique aux processus, et assurer la traduction d'une adresse d'un objet dans son espace propre en l'adresse en mémoire physique où il est situé.

Il faut bien distinguer l'espace des adresses d'un processus de l'espace de mémoire physique. Si dans certains cas, il y a une certaine identité, comme dans le partitionnement, nous verrons avec la segmentation que les structures de ces deux espaces peuvent être très différentes.

14.1.3. Les difficultés du partitionnement

Considérons une mémoire centrale linéaire de N emplacements. L'espace des adresses physiques est dans ce cas l'ensemble des entiers compris entre 0 et N-1. La première méthode pour résoudre les problèmes ci-dessus a été de découper cet espace des adresses physiques en zones disjointes, que l'on appelé *partitions*, et de construire ainsi des espaces d'adresses disjoints pour plusieurs processus.

- Lors de sa création, un processus dispose de l'un de ces espaces, par exemple la partition [A..B]. Il est libre de placer les objets comme il l'entend à l'intérieur de cet espace. Si un autre processus est présent en mémoire en même temps que lui, celui-ci disposera d'un autre espace disjoint du précédent, par exemple la partition [C..D], telle que l'on ait B<C ou D<A.
- La protection peut être obtenue simplement en interdisant au processus de construire une adresse en dehors de sa partition. Par exemple, le

processeur se déroutera vers le système si, lorsqu'il exécute une instruction pour un processus dont l'espace des adresses est la partition [A..B], il génère une adresse en dehors de cette partition.

- L'allocation de mémoire physique est en fait assez simple, puisqu'un processus dont l'espace des adresses est la partition [A..B] reçoit la portion de mémoire physique comprise entre les adresses A et B.

On peut définir un partitionnement fixe, par exemple au lancement du système, les partitions n'étant pas forcément toutes de la même taille, comme le montre la figure 14.2. Il est possible de définir a priori la partition dont dispose un processus donné. L'allocation de mémoire physique est alors obtenue en associant une file d'attente par partition, les processus d'une même partition étant amenés les uns après les autres dans la portion correspondante de mémoire physique. Cette méthode présente l'inconvénient qu'une partition peut être libre (pas de processus en mémoire ni dans la file d'attente), alors que d'autres sont surchargées, entraînant une mauvaise utilisation de la mémoire et du processeur.

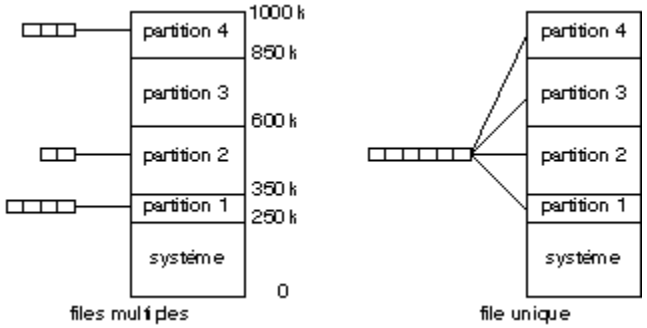

Fig. 14.2. *Gestion mémoire par partitionnement.*

En utilisant un chargeur translatable, il est possible de retarder l'attribution de la partition au processus. On dispose alors d'une seule file d'attente de processus. Lorsqu'une partition est libre, le système recherche un processus de la file qui peut se satisfaire de la taille de cette partition, et le charge en mémoire dans cette partition. La recherche peut s'arrêter sur le premier trouvé, ou rechercher celui qui a le plus gros besoin d'espace, tout en étant inférieur à la taille de la partition; notons que ceci peut entraîner la famine d'un processus qui ne demanderait qu'un tout petit espace.

Un inconvénient du partitionnement fixe réside dans une mauvaise utilisation de la mémoire physique, dans la mesure où un processus reçoit une portion de mémoire plus grande que nécessaire. Il est possible de rendre variables les tailles des partitions, un processus ne recevant que ses besoins stricts. L'espace alloué à un processus devant être contigu, on retrouve alors les mêmes inconvénients que ceux rencontrés pour l'allocation par zone des objets externes sur disque. Cette méthode est souvent utilisée dans les micro-

ordinateurs, pour la mise en mémoire de plusieurs “programmes”, tout en n'ayant toujours qu'un seul de ces programmes qui s'exécute.

14.2. La notion de mémoire segmentée

Un espace d'adresse linéaire pour un processus est parfois une gène, car cela complique la gestion des objets dans cet espace par le processus. Cette difficulté est accrue si le processus désire partager des données ou du code instruction avec un autre processus. Dans ce cas, comment chacun de ces processus va-t-il désigner ces objets partagés dans son propre espace? Si l'espace est assez grand, les processus peuvent, par exemple, décider a priori de réserver des parties de leur espace à de tels objets.
Pour éviter ces inconvénients, on a introduit la notion de *mémoire segmentée*, qui est une sorte d'espace à deux dimensions (figure 14.3). Chaque processus dispose de son propre espace mémoire constitué d'un ensemble de *segments*, chaque segment étant un espace linéaire de taille limitée. Un emplacement de cet espace est désigné par un couple `<s, d>` où `s` désigne le segment, et `d` un déplacement à l'intérieur du segment. Un segment regroupe en général des objets de même nature (instructions, données ou constantes, partagées ou propres, etc...). On retrouve ici une notion déjà rencontrée lors de l'édition de liens: un module est constitué de sections, et l'éditeur de liens regroupe ensemble les sections de même nature.

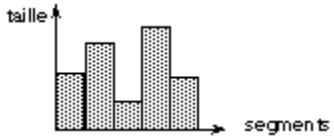

Fig. 14.3. *Représentation d'une mémoire segmentée.*

Cette notion de mémoire segmentée peut ou non être prise en compte par le matériel. Lorsqu'elle ne l'est pas, l'éditeur de liens ou le chargeur place les segments dans un espace linéaire, avant de lancer l'exécution du processus. Ce placement est alors statique, et la taille des segments est souvent invariable. Certains matériels prennent en compte un tel espace, en assurant dynamiquement la traduction d'une adresse segmentée en une adresse dans un espace linéaire, au moyen de tables spécifiques qui contiennent les informations de localisation de chaque segment, encore appelés *descripteurs de segment*. Un tel descripteur contient les informations suivantes:

- L'indicateur de *validité* indique si le segment existe ou non.
- L'*adresse de début* indique où commence le segment dans l'espace linéaire.

- La *taille du segment* indique le nombre d'emplacements du segment. Le processus doit donner des déplacements dans le segment qui soient inférieurs à la taille.
- Les *droits* indiquent les opérations du processus sur le segment qui sont autorisées.

La figure 14.4 montre le fonctionnement de l'adressage segmenté. Lorsque le processeur interprète une adresse segmentée `<s, d>` pour le compte d'un processus, il compare d'abord la valeur de `s` avec le champs `LT` de son registre spécialisé qui définit la table des descripteurs de segments du processus. Si la valeur est acceptable, il lit en mémoire ce descripteur, dont il vérifie l'indicateur de validité puis contrôle que les droits sont suffisants. Il compare ensuite `d` avec le champs `taille` de ce descripteur. Si la valeur est acceptable, il ajoute `d` à l'adresse de début du segment et obtient ainsi l'adresse de l'emplacement dans la mémoire linéaire. En général, pour éviter l'accès mémoire supplémentaire pour obtenir le descripteur de segment, le processeur dispose de registres spécialisés qui contiennent les descripteurs des derniers segments accédés par le processus.

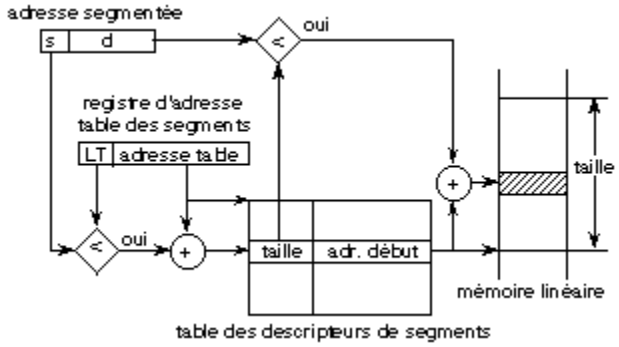

Fig. 14.4. *Fonctionnement de l'adressage segmenté.*

La définition du contenu de la table des descripteurs de segments est à la charge du système, qui doit allouer l'espace dans la mémoire linéaire lors de la création des segments. L'allocation est une allocation par zone contiguë. Mais, notons que le passage systématique par la table des descripteurs de segments permet de déplacer un segment dans la mémoire linéaire, puisqu'il suffit ensuite de modifier le champs `adresse de début` du descripteur pour permettre au processus de continuer son fonctionnement après ce déplacement.
Voici quelques exemples de processeurs disposant de l'adressage segmenté:

- Dans Multics, un processus peut avoir 32768 segments, chacun des segments pouvant atteindre 256 Kmots de 36 bits.
- Dans l'intel iAPX286 (le processeur des PC-AT), la table des descripteurs de segments d'un processus est divisée en deux parties, une table des segments communs à tous les processus, et une table propre à chaque processus. Chacune de ces tables peut avoir 8192 segments, et

chaque segment peut avoir jusqu'à 64 Koctets. La mémoire linéaire est la mémoire physique, et peut atteindre 16 Moctets.

- L'intel iAPX386 (et suivants) est assez voisin de l'iAPX286, du point de vue adressage segmenté, si ce n'est que les segments peuvent atteindre 4 Goctets. La mémoire linéaire peut atteindre 4 Goctets et peut, de plus, être paginée (voir ci-dessous).

14.3. Le mécanisme de pagination

Le mécanisme de pagination a été imaginé pour lever la contrainte de contiguïté de l'espace de mémoire physique allouée aux processus. Pour cela, la mémoire physique est découpée en blocs de taille fixe, que nous appellerons *case* (en anglais *frame*). Par ailleurs, la mémoire linéaire des processus, encore appelée *mémoire virtuelle*, est elle-même découpée en blocs de taille fixe, que nous appellerons *page*. La taille d'une page correspond à la taille d'une case. Chaque page peut alors être placée dans une case quelconque. Le *mécanisme de pagination* du matériel assure la traduction des adresses de mémoire virtuelle en une adresse de mémoire physique, de façon à retrouver la contiguïté de cette mémoire virtuelle.

14.3.1. La pagination à un niveau

Le mécanisme de traduction des adresses virtuelles en adresses réelles doit associer à chaque numéro de page virtuelle le numéro de case réelle qui contient cette page, si elle existe. La figure 14.5 montre le fonctionnement de ce mécanisme.

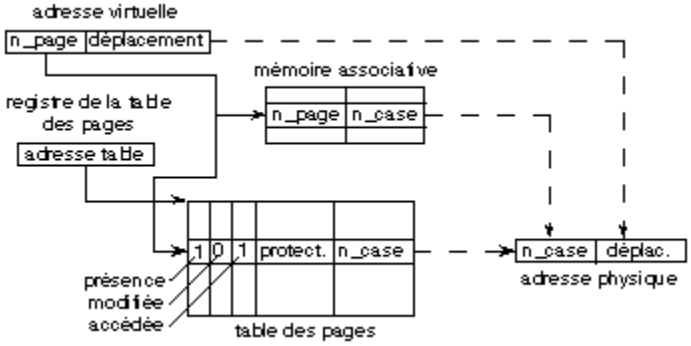

Fig. 14.5. *Fonctionnement de la pagination à un niveau.*

Un registre spécialisé du processeur contient l'adresse de la table qui mémorise cette association; cette table est appelée la *table des pages* du processus. En général, l'adresse de cette table est une adresse physique. Chaque entrée de cette table contient les informations suivantes:

- L'indicateur de *présence* indique s'il y a une case allouée à cette page.

- Le *numéro de case* alloué à cette page.
- Les indicateurs de *protection* de la page indique les opérations autorisées sur cette page par le processus.
- L'indicateur de *page accédée* est positionné lors d'un accès quelconque.
- L'indicateur de *page modifiée* est positionné lors d'un accès en écriture.

Ces deux derniers indicateurs servent à la gestion de la mémoire physique que nous verrons par la suite.
Lorsque le processeur traduit une adresse virtuelle, il en isole le numéro de page virtuelle qu'il utilise comme déplacement dans cette table des pages pour en trouver l'entrée correspondante. Si l'indicateur de présence est positionné, il contrôle la validité de l'accès vis à vis des indicateurs de protection. Si l'accès est accepté, il concatène le numéro de case au déplacement virtuel dans la page pour obtenir l'adresse en mémoire physique de l'emplacement recherché. Pour éviter l'accès mémoire supplémentaire à la table des pages, une mémoire associative contient les entrées de la table correspondant aux derniers accès.
VAX-VMS utilise un tel mécanisme de pagination à un niveau, avec des pages de 512 octets. Mais pour éviter d'avoir des tailles trop importantes pour la table des pages, la mémoire virtuelle est découpée en trois régions, pouvant comporter chacune jusqu'à 1 Goctets. Chacune de ces régions dispose de sa propre table, et les registres correspondants contiennent leur taille effective.

- La région système est commune à tous les processus. L'adresse de sa table de pages est une adresse de mémoire physique.
- Les régions P0 et P1 sont propres au processus, et leur table des pages est dans la région virtuelle système. L'adresse de leur table de pages respective est donc une adresse virtuelle. Les pages de la région P0 sont dans les adresses virtuelles basses, alors que celles de la région P1 sont dans les adresses virtuelles hautes. L'augmentation de la taille de la région P0 se fait donc vers les adresses virtuelles croissantes, alors que celle de la région P1 se fait vers les adresses décroissantes (figure 14.6).

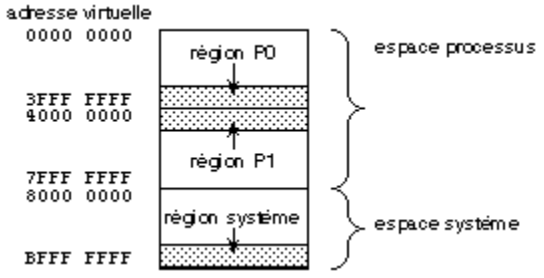

Fig. 14.6. *L'espace virtuel de VAX-VMS.*

14.3.2. La pagination à deux niveaux

La pagination à un niveau d'un espace virtuel de grande taille peut conduire à des tailles importantes de la table des pages. L'exemple précédent de VAX-VMS montre une approche pour réduire cet encombrement, mais impose des contraintes sur la gestion des objets à l'intérieur même de cette mémoire virtuelle, puisqu'il faut qu'ils se trouvent au début ou à la fin de l'espace du processus. La pagination à deux niveaux a pour but de réduire la représentation de la table des pages d'un processus, sans avoir des contraintes aussi fortes.
La figure 14.7 montre le fonctionnement de ce mécanisme. La mémoire virtuelle est divisée en *hyperpages* de taille fixe, chaque hyperpage étant découpée en pages de même taille qu'une case de mémoire physique. Une hyperpage contient donc un nombre limité de pages. À chaque hyperpage, on associe une table de pages analogue à ce qui précède. Par ailleurs, la mémoire virtuelle d'un processus est représentée par une table des hyperpages, qui permet de localiser la table des pages de chaque hyperpage. Le gain de la représentation est obtenu par l'indicateur de présence d'hyperpage qui précise l'existence ou non de la table des pages de cette hyperpage.

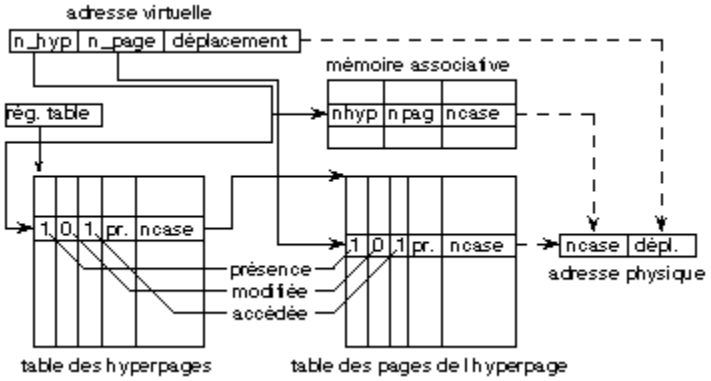

***Fig. 14.7**. Fonctionnement de la pagination à deux niveaux.*

Lorsque le processeur traduit une adresse virtuelle, il isole d'abord le numéro d'hyperpage qu'il utilise comme index dans la table des hyperpages (dont l'adresse est dans un registre spécialisé) pour obtenir l'entrée associée à cette hyperpage. Si l'indicateur de présence est positionné, il vérifie que les indicateurs de protection de l'ensemble de l'hyperpage autorisent l'accès désiré. Puis le processeur isole dans l'adresse virtuelle le numéro de page qu'il utilise comme index dans la table des pages de l'hyperpage pour obtenir l'entrée associée à cette page. Il poursuit avec cette entrée comme pour la pagination à un niveau. Comme précédemment, les accès mémoires supplémentaires peuvent être évités par l'utilisation d'une mémoire associative qui conserve les entrées de pages (repérées par leur numéro d'hyperpage et de page) correspondant aux derniers accès.
Voici quelques exemples de machines ayant la pagination à deux niveaux:

- L'IBM 370 dispose de plusieurs configurations possibles. Sur les machines à mémoire virtuelle sur 16 Moctets, on peut avoir, par exemple, 256 hypergages de 16 pages de 4096 octets.
- L'iAPX386 structure la mémoire virtuelle (appelée aussi mémoire linéaire lors de l'étude de la segmentation) en 1024 hyperpages de 1024 pages de 4096 octets.

14.3.3. La segmentation vis à vis de la pagination

La segmentation et la pagination sont deux notions différentes qui sont utilisées conjointement sur certaines machines (par exemple, Multics, iAPX386). La segmentation doit être vue comme une structuration de l'espace des adresses d'un processus, alors que la pagination doit être vue comme un moyen d'adaptation de la mémoire virtuelle à la mémoire réelle.

- Avec la segmentation, le processus dispose d'un espace des adresses à deux dimensions que le processeur transforme en une adresse dans une mémoire linéaire. Sans la segmentation, le processus dispose directement de cette mémoire linéaire.
- Avec la pagination, on dispose d'une fonction de transformation dynamique des adresses de la mémoire linéaire en adresses de la mémoire physique, qui permet de placer les pages de la mémoire linéaire dans des cases quelconques de mémoire physique. Sans la pagination, les pages de la mémoire linéaire doivent être placées dans les cases de la mémoire physique de même numéro.

Lorsqu'on dispose de la segmentation, les segments doivent être alloués dans la mémoire linéaire en utilisant les techniques d'allocation par zone (portion de mémoire contiguë de taille donnée). Lorsqu'on dispose de la pagination, les pages de la mémoire linéaire doivent être allouées dans la mémoire physique. Il peut sembler complexe de devoir mettre en œuvre les deux notions. Cependant, elles sont complémentaires, et la pagination simplifie l'allocation par zone de la segmentation. D'une part, il peut alors y avoir des pages libres au milieu de pages occupées dans la mémoire linéaire, sans perte de place en mémoire physique. D'autre part, le déplacement d'une page de la mémoire linéaire dans une autre page, peut être obtenue sans déplacement dans la mémoire physique, puisqu'il suffit de modifier les entrées correspondantes de la table des pages. Notons que le partage de données entre deux processus peut être obtenu de différentes façons. Par exemple, il peut être obtenu au niveau des pages de leur mémoire linéaire, en allouant la même case de mémoire physique aux deux processus, au même instant. Il peut être obtenu au niveau des hyperpages de leur mémoire linéaire, en allouant la même table des pages à deux hyperpages

de ces processus. Sans pagination, il peut être obtenu au niveau segment en allouant la même zone de mémoire physique à deux segments de chacun des deux processus.

14.3.4. Le principe de la pagination à la demande

Notons tout d'abord que la mémoire virtuelle telle qu'elle a été présentée ci-dessus, peut être de taille très importante, et beaucoup plus grande que ce que peut être une taille raisonnable de mémoire physique. Dans un contexte de multiprogrammation, il peut être nécessaire de disposer de plusieurs mémoires virtuelles. Pour résoudre cette difficulté, on a imaginé de ne mettre en mémoire physique que les pages de mémoire virtuelle dont les processus ont besoin pour leur exécution courante, les autres étant conservées sur mémoire secondaire (disque). Lorsqu'un processus accède à une page qui n'est pas en mémoire physique (indicateur de présence non positionné), l'instruction est interrompue, et un déroutement au système est exécuté. On dit qu'il y a *défaut de page*. Le système doit alors allouer une case à cette page, lire la page depuis la mémoire secondaire dans cette case et relancer l'exécution de l'instruction qui pourra alors se dérouler jusqu'à son terme.
Lorsqu'un défaut de page se produit, s'il y a une case libre, le système peut allouer cette case. Si aucune case n'est libre, il est nécessaire de réquisitionner une case occupée par une autre page. On dit qu'il y a *remplacement* de page. Plusieurs algorithmes de remplacement de pages ont été proposés qui diffèrent par le choix de la page remplacée. Si celle-ci a été modifiée, il faut auparavant la réécrire en mémoire secondaire.

- L'*algorithme optimal* consiste à choisir la page qui sera accédée dans un avenir le plus lointain possible. Évidemment un tel algorithme n'est pas réalisable pratiquement, puisqu'il n'est pas possible de prévoir les accès futurs des processus. Il a surtout l'avantage de la comparaison: c'est en fait celui qui donnerait la meilleure efficacité.
- L'*algorithme de fréquence d'utilisation* (encore appelé *LFU* pour *least frequently used*) consiste à remplacer la page qui a été la moins fréquemment utilisée. Il faut maintenir une liste des numéros de page ordonnée par leur fréquence d'utilisation. Cette liste change lors de chaque accès mémoire, et doit donc être maintenue par le matériel.
- L'*algorithme chronologique d'utilisation* (encore appelé *LRU* pour *least recently used*) consiste à remplacer la page qui n'a pas été utilisée depuis le plus longtemps. Il faut maintenir une liste des numéros de pages ordonnée par leur dernier moment d'utilisation. Cette liste change lors de chaque accès mémoire, et doit donc être maintenue par le matériel.

- L'*algorithme chronologique de chargement* (encore appelé *FIFO* pour *first in first out*) consiste à remplacer la page qui est en mémoire depuis le plus longtemps. Sa mise en œuvre est simple, puisqu'il suffit d'avoir une file des numéros de pages dans l'ordre où elles ont été amenées en mémoire.
- L'*algorithme aléatoire* (encore appelé *random*) consiste à tirer au sort la page à remplacer. Sa mise en œuvre est simple, puisqu'il suffit d'utiliser une fonction comme `random`, qui est une fonction standard de presque tous les systèmes.

D'autres algorithmes ont été construits qui sont des compromis entre le matériel et le logiciel.

- L'*algorithme de la seconde chance* est dérivé de l'algorithme FIFO et utilise l'indicateur de page `accédée`. Lors d'un remplacement, si la dernière page de la file a son indicateur positionné, elle est mise en tête de file, et son indicateur est remis à zéro (on lui donne une seconde chance), et on recommence avec la nouvelle dernière, jusqu'à en trouver une qui ait un indicateur nul et qui est alors remplacée. Cela évite de remplacer des pages qui sont utilisées pendant de très longues périodes.
- L'*algorithme de non récente utilisation* (encore appelé *NRU* pour *not recently used*), est dérivé de LRU, et utilise les indicateurs de page `accédée` et de page `modifiée` de la table des pages, pour définir six catégories de pages (figure 14.8). Lorsque les pages sont dans les catégories `C1` à `C4`, elles sont présentes dans la table des pages, et le matériel gérant les indicateurs peut les faire changer de catégories (transitions marquées `*` sur la figure). Lorsqu'elles sont dans les catégories `C5` et `C6`, elles ne sont pas présentes dans les tables; un accès à une telle page provoquera donc un défaut de page. Périodiquement, le système transfère les pages des catégories `C1` à `C4` dans la catégorie qui est située immédiatement à droite sur la figure. De plus, il lance de temps en temps la réécriture sur mémoire secondaire de celles qui sont dans `C5`. Lorsqu'une telle réécriture est terminée, si la page est toujours dans `C5`, elle est transférée dans `C6`. Lorsqu'un défaut de page survient, le système recherche d'abord si la page demandée n'est pas dans `C5` ou `C6`, pour la remettre alors dans la table, et ainsi la transférer dans `C3` ou `C4` suivant le cas (le matériel la mettra ensuite dans `C1` ou `C2`). Si elle n'y est pas, il y a remplacement de la première page trouvée dans une catégorie, par numéro décroissant, c'est-à-dire, en commençant par `C6`.

Fig. 14.8. *Les catégories de l'algorithme NRU. Les transitions marquées * sont provoquées par le matériel.*

On peut mesurer l'efficacité d'un algorithme de remplacement par la chance que la page soit présente en mémoire lors d'un accès, ou encore par la probabilité d'avoir un défaut de page lors d'un tel accès. On constate que les algorithmes se comportent presque tous assez bien et de la même façon. Évidemment l'algorithme optimal est le meilleur, suivi par les algorithmes LRU et LFU, ensuite par NRU et la seconde chance, et enfin par FIFO et aléatoire. Mais cette probabilité est beaucoup plus influencée par le nombre de cases attribuées à un processus que par l'algorithme lui-même.

14.4. La notion d'espace de travail

Si un processus a en mémoire toutes les pages dont il a besoin pour s'exécuter pendant une période suffisante, la probabilité d'avoir un défaut de page est nulle. Au contraire, si toutes les pages ne sont pas présentes, il y aura des défauts entraînant des remplacements de pages qui peuvent redevenir nécessaires par la suite. En d'autres termes, si un processus a un nombre de cases suffisant, presque toutes les pages dont il a besoin seront en mémoire, il y a peu de défaut. Si ce nombre tombe en dessous d'un seuil dépendant du processus, la probabilité de défaut augmente brutalement. Or chaque défaut de page bloque le processus en attendant que la page soit présente en mémoire. On augmente ainsi son temps global d'attente d'entrées-sorties, et nous avons vu au début du chapitre l'influence de ce paramètre sur le taux d'utilisation du processeur.

Un processus, qui fait au total N accès à la mémoire, avec une probabilité de défaut de pages p, sera bloqué pour défaut de page, pendant Td=pNT, si T est le temps pour lire une page. Comme, par ailleurs, on peut évaluer son temps de processeur à Tuc=Nt, où t est le temps pour un accès à la mémoire physique, on en déduit que Td=p(T/t)Tuc. Or T/t est environ de 10^4 actuellement, ce qui montre que p doit être maintenu à des valeurs très faibles sous peine d'avoir un temps Td très grand par rapport à Tuc, entraînant une augmentation du taux d'attente d'entrées-sorties du processus, et par voie de conséquence une diminution du taux d'utilisation du processeur. Ce phénomène, assez brutal du fait des coefficients, est appelé l'*écroulement* du système (en anglais *thrashing*).

On appelle *espace de travail* (en anglais *working set*) les pages dont a besoin un processus, à un instant donné, pour s'exécuter. Le paragraphe précédent montre

l'intérêt d'évaluer la taille de cet espace de travail, puisqu'elle correspond au nombre de cases nécessaires au processus. En dessous, il y aura des défauts de pages, au-dessus la mémoire physique sera mal utilisée. Si T(P) est la taille de cet espace pour le processus P, il faut alors déterminer l'ensemble E des processus que l'on peut mettre en mémoire de telle sorte que l'on ait:

$$\sum_{P \in E} T(P) \leq M$$

où M est la taille totale de la mémoire disponible. VAX-VMS utilise cette notion pour décider quels sont les processus qui sont résidents en mémoire centrale, et ceux dont l'état est temporairement conservé sur mémoire secondaire.

14.5. Conclusion

+ La multiprogrammation a pour but d'améliorer l'efficacité du processeur par la présence de plusieurs processus en mémoire de telle sorte qu'il y en ait presque toujours un qui soit prêt.
+ Les trois problèmes à résoudre sont la définition de l'espace d'adresses des processus, la protection de leurs objets propres et l'attribution d'une partie de la mémoire physique à chacun d'eux.
+ Dans le partitionnement, les processus reçoivent une partie contiguë de la mémoire physique qui constitue leur espace d'adresses. Cela conduit à une mauvaise utilisation de la mémoire physique.
+ La mémoire segmentée fournit aux processus un espace d'adresses à deux dimensions leur permettant de regrouper ensembles des objets de même nature, et de conserver cette structuration à l'exécution lorsque le matériel supporte ce mécanisme. Cela simplifie la gestion de leur mémoire virtuelle.
+ La pagination est un mécanisme qui permet de plaquer la mémoire virtuelle sur la mémoire réelle, en les découpant respectivement en pages et en cases de même taille, et en mettant les pages dans n'importe quelles cases sans devoir respecter la contiguïté.
+ La pagination peut être à un niveau, une table des pages assurant la correspondance entre les pages et les cases. La table des pages peut être réduite, par une pagination à deux niveaux, en découpant la mémoire virtuelle en hyperpages elles-mêmes découpées en pages, une table des hyperpages repérant la table des pages de chacune des hyperpages.
+ La segmentation et la pagination peuvent être utilisées conjointement puisque ce sont des mécanismes qui s'adressent à deux problèmes différents.
+ La pagination à la demande consiste à n'allouer une case à une page que lorsqu'un processus en a besoin. Cela implique de mettre en œuvre un algorithme de remplacement lorsque toutes les cases sont occupées. Ces

algorithmes peuvent être uniquement logiciels ou utiliser certains dispositifs matériels. Ils se comportent presque tous assez bien.
+ Chaque processus a besoin d'un ensemble de pages pour s'exécuter, et qui constitue son espace de travail. Le nombre de cases dont dispose un processus doit être égal à la taille de cet espace de travail. S'il est plus grand, la mémoire est mal utilisée. S'il est plus petit, le nombre de défaut de pages augmente, ce qui peut conduire à l'écroulement du système.