Chapitre 7 GESTION DE LA MEMOIRE

Pour pouvoir utiliser un ordinateur en multiprogrammation, le SE charge plusieurs processus en mémoire centrale (MC). La façon la plus simple consiste à affecter à chaque processus un ensemble d'adresses contiguës.

Quand le nombre de tâches devient élevé, pour satisfaire au principe d'équitabilité et pour minimiser le temps de réponse des processus, il faut pouvoir **simuler** la présence simultanée en MC de tous les processus. D'où la technique de "va et vient" ou **recouvrement (swapping)**, qui consiste à stocker temporairement sur disque l'image d'un processus, afin de libérer de la place en MC pour d'autres processus.

D'autre part, la taille d'un processus doit pouvoir dépasser la taille de la mémoire disponible, même si l'on enlève tous les autres processus. L'utilisation de pages (mécanisme de **pagination**) ou de segments (mécanisme de **segmentation**) permet au système de conserver en MC les parties utilisées des processus et de stocker, si nécessaire, le reste sur disque.

Le rôle du gestionnaire de la mémoire est de connaître les parties libres et occupées, d'allouer de la mémoire aux processus qui en ont besoin, de récupérer de la mémoire à la fin de l'exécution d'un processus et de traiter le recouvrement entre le disque et la mémoire centrale, lorsqu'elle ne peut pas contenir tous les processus actifs.

1.GESTION SANS RECOUVREMENT, NI PAGINATION

1.1 La monoprogrammation

Il n'y a en MC que :

- un seul processus utilisateur,
- le processus système (pour partie en RAM, pour partie en ROM; la partie en ROM étant appelée BIOS [*Basic Input Output System*])
- les pilotes de périphériques

Cette technique en voie de disparition est limitée à quelques micro-ordinateurs . Elle n'autorise qu'un seul processus actif en mémoire à un instant donné.

1.2 La multiprogrammation

La multiprogrammation est utilisée sur la plupart des ordinateurs : elle permet de diviser un programme en plusieurs processus et à plusieurs utilisateurs de travailler en temps partagé avec la même machine.

Supposons qu'il y ait n processus indépendants en MC, chacun ayant la probabilité p d'attendre la fin d'une opération d'E/S. La probabilité que le processeur fonctionne est $1 - p^n$.(Il s'agit d'une estimation grossière, puisque sur une machine monoprocesseur, les processus ne sont pas indépendants (attente de libération de la ressource processeur pour démarrer l'exécution d'un processus prêt).

Toutefois, on remarque que plus le nombre n de processus en MC est élevé, plus le taux d'utilisation du processeur est élevé : on a donc intérêt à augmenter la taille de la MC.

Une solution simple consiste à diviser la mémoire en **n partitions fixes**, de tailles pas nécessairement égales (méthode MFT [*Multiprogramming with a Fixed number of Tasks*] apparue avec les IBM 360). Il existe deux méthodes de gestion :

- on crée une file d'attente par partition . Chaque nouveau processus est placé dans la file d'attente de la plus petite partition pouvant le contenir. Inconvénients :
  * on perd en général de la place au sein de chaque partition
  * il peut y avoir des partitions inutilisées (leur file d'attente est vide)

- on crée une seule file d'attente globale. Il existe deux stratégies :
  * dès qu'une partition se libère, on lui affecte la première tâche de la file qui peut y tenir. Inconvénient : on peut ainsi affecter une partition de grande taille à une petite tâche et perdre beaucoup de place

  * dès qu'une partition se libère, on lui affecte la plus grande tâche de la file qui peut y tenir. Inconvénient : on pénalise les processus de petite taille.

1.3 Code translatable et protection

Avec le mécanisme de multiprogrammation, un processus peut être chargé n'importe où en MC. Il n'y a plus concordance entre l'adresse dans le processus et l'adresse physique d'implantation. Le problème de l'adressage dans un processus se résout par l'utilisation d'un registre particulier, le **registre de base** : au lancement du processus, on lui affecte l'adresse de début de la partition qui lui est attribuée. On a alors :

adresse physique = adresse de base (contenu de ce registre)
+ adresse relative (mentionnée dans le processus)

De plus, il faut empêcher un processus d'écrire dans la mémoire d'un autre. Deux solutions :

* des **clés de protection** : une clé de protection pour chaque partition, dupliquée dans le mot d'état (PSW = Program Status Word) du processus actif implanté dans cette partition. Si un processus tente d'écrire dans une partition dont la clé ne concorde pas avec celle de son mot d'état, il y a refus.

* un **registre limite**, chargé au lancement du processus à la taille de la partition. Toute adresse mentionnée dans le processus (relative) supérieure au contenu du registre limite entraîne un refus.

2. GESTION AVEC RECOUVREMENT, SANS PAGINATION

Dès que le nombre de processus devient supérieur au nombre de partitions, il faut pouvoir simuler la présence en MC de tous les processus pour pouvoir satisfaire au principe d'équité et minimiser le temps de réponse des processus. La technique du recouvrement (swapping) permet de stocker temporairement sur disque des images de processus afin de libérer de la MC pour d'autres processus.

On pourrait utiliser des partitions fixes, mais on utilise en pratique des **partitions de taille variable**, car le nombre, la taille et la position des processus peuvent varier dynamiquement au cours du temps. On n'est plus limité par des partitions trop grandes ou trop petites comme avec les partitions fixes. Cette amélioration de l'usage de la MC nécessite un mécanisme plus complexe d'allocation et de libération.

2.1 Opérations sur la mémoire

Le **compactage** de la mémoire permet de regrouper les espaces inutilisés. Très coûteuse en temps UC, cette opération est effectuée le moins souvent possible.

S'il y a une requête d'**allocation dynamique de mémoire** pour un processus, on lui alloue de la place dans le tas (heap) si le SE le permet, ou bien de la mémoire supplémentaire contiguë à la partition du processus (agrandissement de celle-ci). Quand il n'y a plus de place, on déplace un ou plusieurs processus :

- soit pour récupérer par ce moyen des espaces inutilisés,
- soit en allant jusqu'au recouvrement. A chaque retour de recouvrement (swap), on réserve au processus une partition un peu plus grande que nécessaire, utilisable pour l'extension de la partition du processus venant d'être chargé ou du processus voisin.

Il existe trois façons de mémoriser l'occupation de la mémoire : les tables de bits (*bits maps*), les listes chaînées et les subdivisions (*buddy*).

2.2 Gestion de la mémoire par table de bits

On divise la MC en **unités d'allocations** de quelques octets à quelques Ko. A chaque unité, correspond un bit de la table de bits : valeur 0 si l'unité est libre, 1 sinon. Cette table est stockée en MC. Plus la taille moyenne des unités est faible, plus la table occupe de place.

A un retour de recouvrement (swap), le gestionnaire doit rechercher suffisamment de 0 consécutifs dans la table pour que la taille cumulée de ces unités permette de loger le nouveau processus à implanter en MC, avec le choix entre trois critères possibles :

- la **première zône libre** (*first fit*) : algorithme rapide
- le **meilleur ajustement** (*best fit*), mais on peut créer ainsi de nombreuses petites unités résiduelles inutilisables (fragmentation nécessitant un compactage ultérieur)
- le **plus grand résidu** (*worst fit*), avec le risque de fractionnement regrettable des grandes unités

2.3 Gestion de la mémoire par liste chaînée

On utilise une liste chaînée des zones libres en MC. On applique :

- soit l'un des algorithmes précédents,

- soit un algorithme de **placement rapide** (*quick fit*) : on crée des listes séparées pour chacune des tailles les plus courantes, et la recherche est considérablement accélérée.

A l'achèvement d'un processus ou à son transfert sur disque, il faut du temps (mise à jour des liste chaînées) pour examiner si un regroupement avec ses voisins est possible pour éviter une fragmentation excessive de la mémoire.

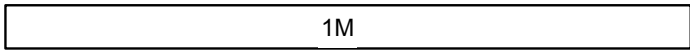
En résumé, les listes chaînées sont une solution plus rapide que la précédente pour l'allocation, mais plus lente pour la libération.

2.4 Gestion de la mémoire par subdivisions (ou frères siamois)

Cet algorithme proposé par Donald KNUTH en 1973 utilise l'existence d'adresses binaires pour accélérer la fusion des zones libres adjacentes lors de la libération d'unités.

Le gestionnaire mémorise une liste de blocs libres dont la taille est une puissance de 2 (1, 2, 4, 8 octets, ..., jusqu'à la taille maximale de la mémoire).

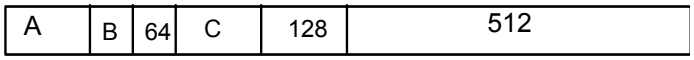
Par exemple, avec une mémoire de 1 Mo, on a ainsi 251listes. Initialement, la mémoire est vide. toutes les listes sont vides, sauf la liste 1 Mo qui pointe sur la zone libre de 1 Mo :



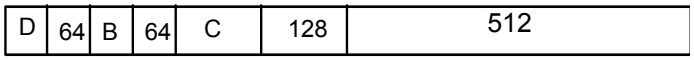
Un processus A demande 70 Ko : la mémoire est fragmentée en deux compagnons (buddies) de 512 Ko; l'un d'eux est fragmenté en deux blocs de 256 Ko; l'un d'eux est fragmenté en deux blocs de 128 Ko et on loge A dans l'un d'eux, puisque $64 < 70 < 128$:

Un processus B demande 35 Ko : l'un des deux blocs de 128 Ko est fragmenté en deux de 64 Ko et on loge B dans l'un d'eux puisque $32 < 35 < 64$:

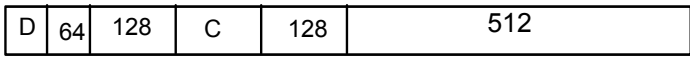
Un processus C demande 80 Ko : le bloc de 256 Ko est fragmenté en deux de 128 Ko et on loge C dans l'un d'eux puisque $64 < 80 < 128$:



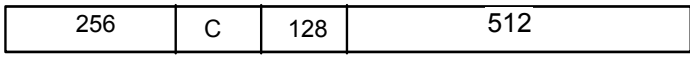
A s'achève et libère son bloc de 128 Ko. Puis un processus D demande 60 Ko : le bloc libéré par A est fragmenté en deux de 64 Ko, dont l'un logera D :



B s'achève, permettant la reconstitution d'un bloc de 128 Ko :



D s'achève, permettant la reconstitution d'un bloc de 256 Ko , etc...



L'allocation et la libération des blocs sont très simples. Mais un processus de taille $2^n + 1$ octets utilisera un bloc de 2^{n+1} octets ! Il y a beaucoup de perte de place en mémoire.

Dans certains systèmes, on n'alloue pas une place fixe sur disque aux processus qui sont en mémoire. On les loge dans un espace de va et vient (*swap area*) du disque. Les algorithmes précédents sont utilisés pour l'affectation.

3. GESTION AVEC RECOUVREMENT AVEC PAGINATION OU SEGMENTATION

La taille d'un processus doit pouvoir dépasser la taille de la mémoire physique disponible, même si l'on enlève tous les autres processus. En 1961, J. FOTHERINGHAM proposa le principe de la **mémoire virtuelle** : le SE conserve en mémoire centrale les parties utilisées des processus et stocke, si nécessaire, le reste sur disque. Mémoire virtuelle et multiprogrammation se complètent bien : un processus en attente d'une ressource n'est plus conservé en MC, si cela s'avère nécessaire.

La mémoire virtuelle fait appel à deux mécanismes : segmentation ou pagination. La mémoire est divisée en segments ou pages. Sans recours à la mémoire virtuelle, un processus est entièrement chargé à des adresses contiguës ; avec le recours à la mémoire virtuelle, un processus peut être chargé dans des pages ou des segments non contigus.

3.1 La pagination

L'espace d'adressage d'un processus est divisé en petites unités de taille fixe appelées **pages**. La MC est elle aussi découpée en unités physiques de même taille appelées **cadres**. Les échanges entre MC et disques ne portent que sur des pages entières. De ce fait, l'espace d'adressage d'un processus est potentiellement illimité (limité à l'espace mémoire total de la machine). On parle alors d'**adressage virtuel**.

Pour un processus, le système ne chargera que les pages utilisées. Mais la demande de pages à charger peut être plus élevée que le nombre de cadres disponibles. Une gestion de l'allocation des cadres libres est nécessaire.

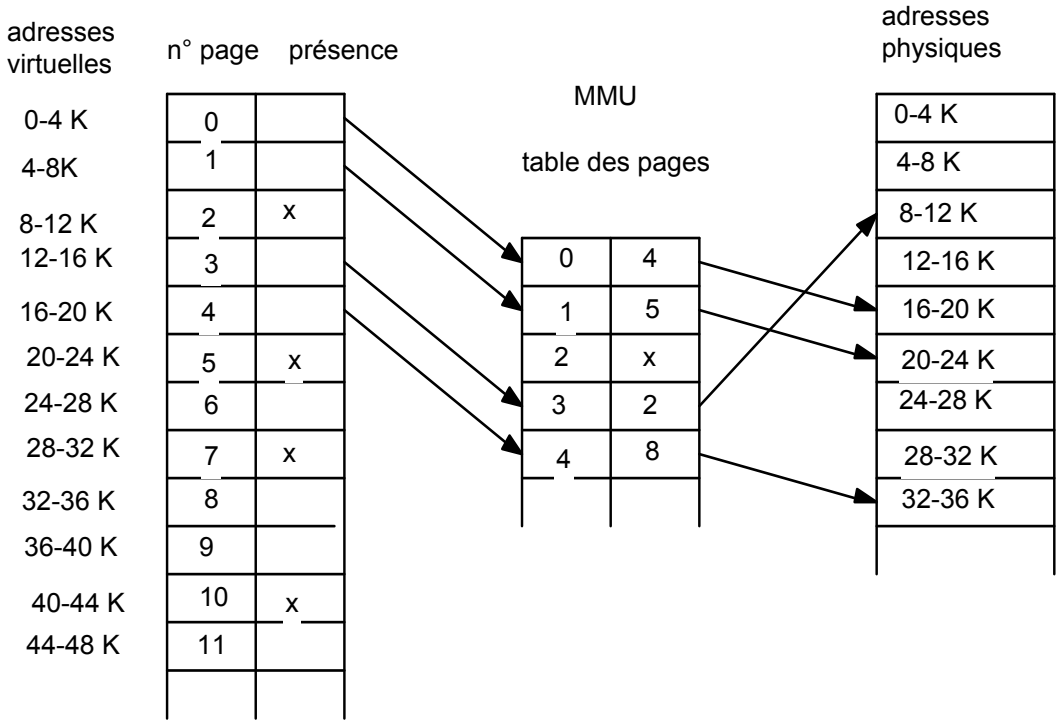
Dans un SE sans mémoire virtuelle, la machine calcule les adresses physiques en ajoutant le contenu d'un registre de base aux adresses relatives contenues dans les instructions du processus. Dans un SE à pagination, un sous-ensemble inséré entre l'UC et la MC, **la MMU** (Memory Management Unit ou unité de gestion de la mémoire) traduit les adresses virtuelles en adresses physiques.

La MMU mémorise :

- les cadres physiques alloués à des processus (sous forme d'une table de bits de présence)

- les cadres mémoire alloués à chaque page d'un processus (sous forme d'une table des pages)

On dira qu'une page est **mappée** ou **chargée** si elle est physiquement présente en mémoire.



Dans l'exemple précédent, les pages ont une taille de 4 Ko. L'adresse virtuelle 12292 correspond à un déplacement de 4 octets dans la page virtuelle 3 (car 12292 = 12288 + 4 et 12288 = 12*1024). La page virtuelle 3 correspond à la page physique 2. L'adresse physique correspond donc à un déplacement de 4 octets dans la page physique 2, soit : (8*1024) + 4 = 8196 .

Par contre, la page virtuelle 2 n'est pas mappée. Une adresse virtuelle comprise entre 8192 et 12287 donnera lieu à **un défaut de page**. Il y a défaut de page quand il y a un accès à une adresse virtuelle correspondant à une page non mappée. En cas de défaut de page, un déroutement se produit (trap) et le processeur est rendu au SE. Le système doit alors effectuer les opérations suivantes :

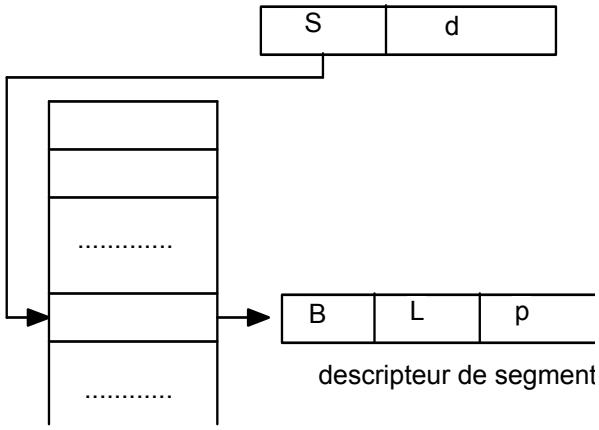
- déterminer la page à charger
- déterminer la page à décharger sur le disque pour libérer un cadre
- lire sur le disque la page à charger
- modifier la table de bits et la table de pages

3.2 La segmentation

Dans cette solution, l'espace d'adressage d'un processus est divisé en **segments**, générés à la compilation. Chaque segment est repéré par son numéro S et sa longueur variable L. Un segment est un ensemble d'adresses virtuelles contiguës.

Contrairement à la pagination, la segmentation est "connue" du processus : une adresse n'est plus donnée de façon absolue par rapport au début de l'adressage virtuel; une adresse est donnée par un couple (S , d), où S est le n° du segment et d le déplacement dans le segment, $d \in [0 , L [$.

Pour calculer l'adresse physique, on utilise une **table des segments** :



B : adresse de base (adresse physique de début du segment)
L : longueur du segment ou limite
p : protection du segment

L'adresse physique correspondant à l'adresse virtuelle (S , d) sera donc B + d, si d <= L

La segmentation simplifie la gestion des objets communs (rangés chacun dans un segment), notamment si leur taille évolue dynamiquement.

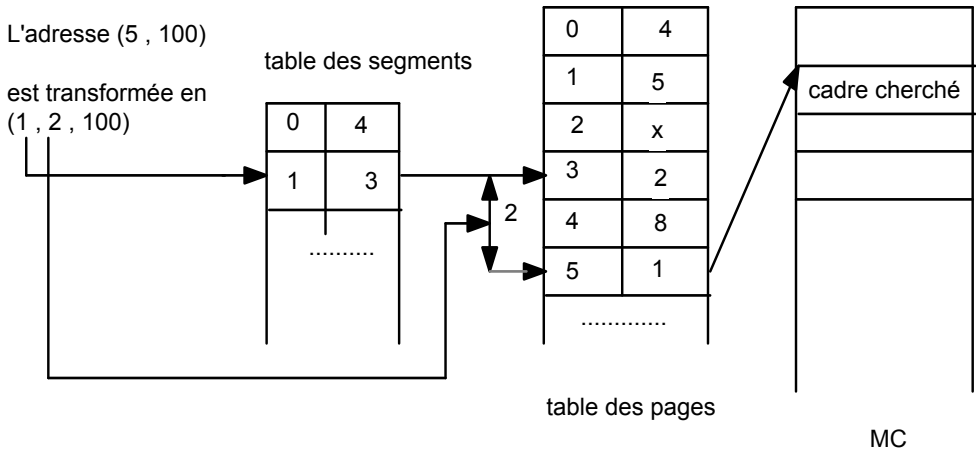
3.3 La pagination segmentée

Lorsque le système possède beaucoup de pages, la consultation de la table des pages peut être lente et inefficace s'il y a beaucoup de pages non chargées. On la segmente alors par processus.

Chaque processus possède une table des segments de la table des pages qui le concernent.

Une adresse (P , r) , avec P n° logique de page et r déplacement dans la page, est transformée à la compilation en adresse (S , P' , r), avec S n° d'un segment et P' .

Exemple :

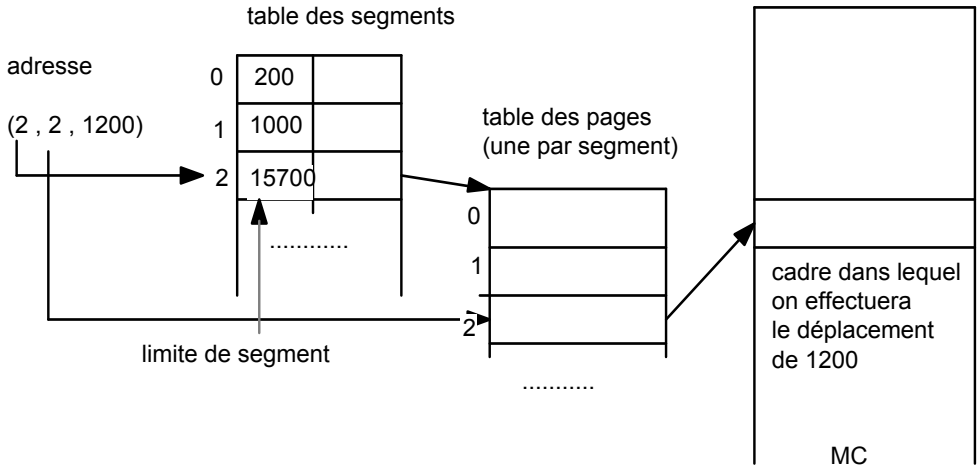


3.4 La segmentation paginée

La taille d'un segment peut être importante, d'où un temps de chargement long qui peut en résulter. La pagination des segments peut être une solution.

Une adresse virtuelle (S , d), avec S n° de segment et d déplacement dans le segment, est transformée en (S , P , d'), où P est un n° de page et d' un déplacement dans la page P.

Exemple : avec l'hypothèse de pages de 4 Ko, l'adresse logique (2 , 9200) est transformée en (2 , 2, 1200)



Dans tous les cas, l'utilisation d'une mémoire associative, stockant des triplets (S, P, adresse de cadre), gérée comme fenêtre au voisinage du dernier cadre accédé, peut accélérer la recherche des cadres.

4. ALGORITHMES DE REMPLACEMENT DE PAGES

Les méthodes reposent sur 3 stratégies :

- une stratégie de **chargement** qui choisit les pages à charger et l'instant de chargement
- une stratégie de **placement** qui choisit un cadre libre pour chaque page à charger
- une stratégie de r**emplacement** destinée à libérer certains cadres. La recherche se fait soit sur l'ensemble des pages (recherche globale), soit sur les pages "appartenant" au processus (recherche locale). Les stratégies de placement sont dépendantes des stratégies de remplacement : on charge nécessairement une page dans le cadre qui vient d'être libéré.

On peut distinguer deux catégories de méthodes :

- **pagination anticipée** : on charge les pages à l'avance; mais comment prévoir efficacement ?
- **pagination à la demande** : le chargement n'a lieu qu'en cas de défaut de page et on ne charge que la page manquante.

On appelle **suite de références** $w = p_1\ p_2\ ...\ p_n$ une suite de numéros de pages correspondant à des accès à ces pages. On dira qu'un algorithme A de pagination sur m cadres est **stable** si :

$$\forall\ m\ ,\ \forall\ w\ ,\ \text{on a : Coût}\ (A\ ,\ m\ ,\ w) <= \text{Coût}\ (\ A\ ,\ m+1\ ,\ w)$$

Cette notion est importante pour éviter l'écroulement du SE (thrashing) par une génération excessive de défauts de pages.

4.1 Remplacement de page optimal

Le SE indexe chaque page par le nombre d'instructions qui seront exécutées avant qu'elle ne soit référencée. En cas de nécessité, le SE retire la page d'indice le plus élevé, c'est à dire la page qui sera référencée dans le futur le plus lointain.

Cet algorithme est pratiquement impossible à appliquer (comment calculer les indices des pages ?). Toutefois, avec un simulateur, on peut évaluer les performances de cet algorithme et s'en servir comme référence pour les suivants. En outre, cet algorithme est stable.

4.2 NRU (not recently used)

Le SE référence chaque page par deux bits R (le plus à gauche) et M initialisés à 0. A chaque accès en lecture à une page, R est mis à 1. A chaque accès en écriture, M est mis à 1. A chaque interruption d'horloge, le SE remet R à 0.

Les bits R-M forment le code d'un index de valeur 0 à 3 en base 10. En cas de défaut de page, on retire une page au hasard dans la catégorie non vide de plus petit index. On retire donc préférentiellement une page modifiée non référencée (index 1) qu'une page très utilisée en consultation (index 2). Cet algorithme est assez efficace.

4.3 FIFO (first in, first out)

Le SE indexe chaque page par sa date de chargement et constitue une liste chaînée, la première page de la liste étant la plus ancienne ment chargée et la dernière la plus récemment chargée. Le SE replacera en cas de nécessité la page en tête de la liste et chargera la nouvelle page en fin de liste.

Deux critiques à cet algorithme :

- ce n'est pas parce qu'une page est la plus ancienne en mémoire qu'elle est celle dont on se sert le moins
- l'algorithme n'est pas stable : quand le nombre de cadres augmente, le nombre de défauts de pages ne diminue pas nécessairement (on parle de l'anomalie de BELADY : L.A. BELADY a proposé en 1969 un exemple à 4 cadres montrant qu'on avait plus de défaut de pages qu'avec 3).

Amélioration 1 : le SE examine les bits R et M (gérés comme ci-dessus en 4.2) de la page la plus ancienne en MC (en tête de la liste). Si cette page est de catégorie 0, il l'ôte. Sinon, il teste la page un peu moins ancienne, etc... S'il n'y a aucune page de catégorie 0, il recommence avec la catégorie 1, puis éventuellement avec les catégories 2 et 3.

Amélioration 2 : **Algorithme de la seconde chance** : R et M sont gérés comme ci-dessus en 4.2. Le SE examine le bit R de la page la plus ancienne (tête de la liste). Si R = 0, la page est remplacée, sinon R est mis à 0, la page est placée en fin de liste et on recommence avec la nouvelle page en tête. Si toutes les pages ont été référencées depuis la dernière RAZ, on revient à FIFO, mais avec un surcoût.

4.4 LRU (least recently used)

En cas de nécessité, le SE retire la page la moins récemment référencée. Pour cela, il indexe chaque page par le temps écoulé depuis sa dernière référence et il constitue une liste chaînée des pages par ordre décroissant de temps depuis la dernière référence.

L'algorithme est stable. Mais il nécessite une gestion coûteuse de la liste qui est modifiée à chaque accès à une page.

Variantes :

NFU (not frequently used) ou **LFU** (least frequently used) : le SE gère pour chaque page un compteur de nombre de références . Il cumule dans ce compteur le bit R juste avant chaque RAZ de R au top d'horloge. Il remplacera la page qui aura la plus petite valeur de compteur.

Vieillisement (aging) ou **NFU modifié** : avant chaque RAZ de R, le SE décale le compteur de 1 bit à droite (division entière par 2) et additionne R au bit de poids **fort**. En cas de nécessité, on ôtera la page de compteur le plus petit. Mais en cas d'égalité des compteurs, on ne sait pas quelle est la page la moins récemment utilisée.

Matrice : s'il y a N pages, le SE gère une matrice (N , N+1). A chaque référence à la page p, le SE positionne à 1 tous les bits de la ligne p et à 0 tous les bits de la colonne p. En outre, le 1er bit de la ligne p vaut 0 si la page est chargée, 1 sinon. On ôtera la page dont la ligne a la plus petite valeur.

4.5 Améliorations

Le dérobeur de pages : dès qu'un seuil minimal de cadres libres est atteint, le SE réveille un dérobeur de pages . Celui-ci supprime les pages les moins récemment utilisées, par algorithme d'aging, et les range dans la liste des cadres libres, ceci jusqu'à un seuil donné. Le SE vérifie qu'une page référencée sur défaut de page n'est pas dans la liste.

Problèmes d'entrée/sortie : on a intérêt, soit à verrouiller les pages concernées par les E/S pour éviter de les ôter, soit à effectuer les E/S dans des tampons du noyau et à copier les données plus tard dans les pages des utilisateurs.

Pages partagées : lorsque plusieurs processus exécutent le même ensemble de code, on a intérêt à utiliser des pages de code partagées, plutôt que des pages dupliquées. Mais on aura à prévoir une gestion spécifique des pages partagées pour éviter des défauts de pages artificiels.

4.6 Taille optimale des pages

Soit t la taille moyenne des processus et p (en octets) la taille d'une page. Soit e (en octets) la taille de l'entrée de chaque page dans la table des processus. Le nombre moyen de pages par processus est t/p. Ces pages occupent t*e/p octets dans la table des processus. La mémoire perdue en moyenne dans la dernière page d'un processus est p/2 (fragmentation interne). La perte totale q est donc :

$$q = t*e/p + p/2$$

q est minimal si : dq/dp = 0, c'est à dire : $p = (2\ t*e)^{0.5}$

Exemple : t = 64 Ko, e = 8 octets, donc p = 1 Ko
Les valeurs courantes sont 1/2 Ko, 1 Ko, 2 Ko ou 4 Ko

4.7 Le modèle du Working Set

Le modèle working-set de comportement d'un programme est une tentative proposée par P.J. DENNING en 1968 pour comprendre les performances d'un système paginé en multi-programmation. Il s'agit de raisonner non pas en terme de processus isolés, mais d'effet des autres processus présents sur chacun d'eux, de corréler allocation de mémoire et allocation du processeur.

Dans le cas d'un système mono-processeur, lorsque le **degré de multi-programmation** augmente (*nombre de processus présents en mémoire*), l'utilisation du processeur augmente : en effet, l'ordonnanceur a toutes les chances de trouver à chaque instant un processus à exploiter.

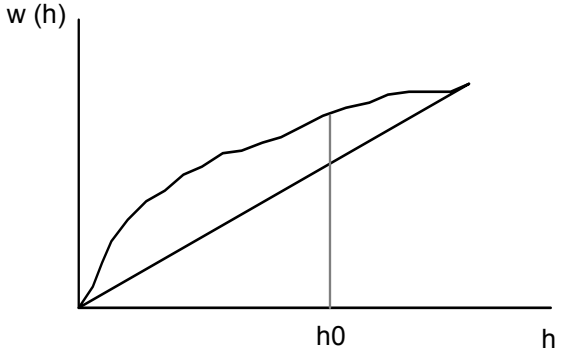
Toutefois, si le degré de multi-programmation franchit un certain seuil, il en résulte une forte croissance du trafic de pagination entre disques et mémoire centrale, accompagnée d'une **diminution brusque de l'utilisation du processeur**. Le haut degré de multi-programmation empêche chaque processus de garder suffisamment de pages en mémoire pour éviter un grand nombre de défauts de pages. Le canal de liaison avec le disque arrive à saturation et beaucoup de processus sont bloqués par attente de transfert de pages. On parle de ***thrashing*** (déconfiture) du processeur.

En d'autres termes, chaque processus exige un nombre minimal de pages présentes en mémoire centrale (appelé son working-set) pour pouvoir réellement utiliser le processeur. Le degré de multi-programmation doit être tel que le working-set de tous les processus présents en mémoire doit être respecté.

Conceptuellement, on peut définir le working-set à un instant t :

w (t , h) = { ensemble des pages apparaissant dans les h dernières références }

DENNING a montré que w(h) avait un graphe tel que celui-ci :



Plus h croît, moins on trouve de pages en excédent dans le working-set. Cela permet d'établir une valeur raisonnable de h (h_0) tel qu'une valeur de h supérieure à h_0 n'accroîtrait pas beaucoup le working-set.

5. PARTAGE DE CODE ET DONNEES EN MEMOIRE CENTRALE

Des procédures peuvent être partagées entre plusieurs processus. On a intérêt à ce qu'il n'y ait qu'une copie du code qui sera exécutée par chaque processus avec ses propres données.

Par exemple deux processus P1 et P2 sont susceptibles d'utiliser une procédure R. On ne peut pas prévoir l'ordre dans lequel ils vont entrer dans R ou sortir de R. Si l'un des deux processus est désalloué pendant qu'il exécute R et qu'il voit ses variables locales modifiées par l'exécution de R par l'autre processus, alors on dit que R n'est pas **ré-entrante**.

5.1 Cas de la pagination

Le partage du code et des données se fait par un double référencement aux pages contenant les informations communes.

5.1.1 Partage du code

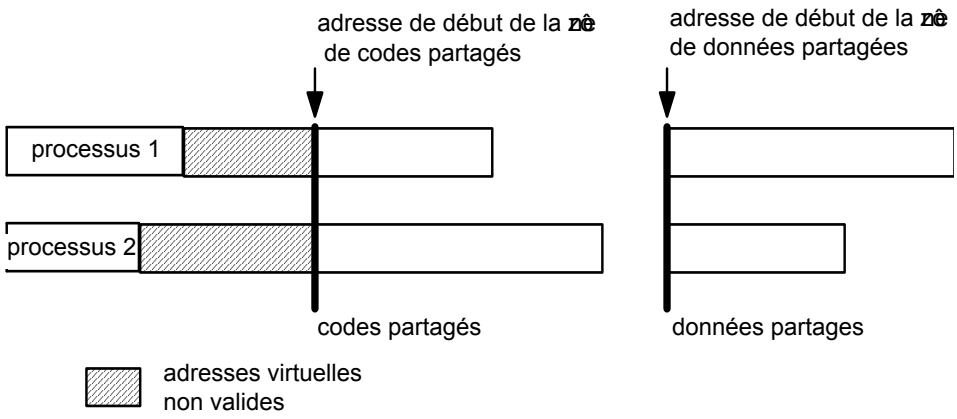
Chaque fois qu'un processus fait référence à son propre code, il utilise un numéro de page fixe qui le fait accéder toujours à la même page virtuelle. Donc, si un processus partage son code, il va toujours référencer la même page virtuelle, quelle que soit la table des pages utilisée. Chaque tâche doit donc mettre le code exécutable de ce processus à la même adresse virtuelle.

5.1.2 Partage de données

Pour partager des données, la zone commune peut être à des endroits différents dans la table des pages de chacune des tâches. Soit une page commune P contenant un pointeur q :

- si D pointe sur une adresse dans P, le numéro de cette page doit être le même dans tous les processus
- si l'objet pointé n'est pas dans P, alors il doit avoir la même adresse virtuelle dans tous les processus

5.1.3 Solution habituelle



5.2 Cas de la segmentation

Pour que deux processus référencent une partie commune, il suffit que cette partie se trouve dans un segment qui possède le même descripteur dans la table des segments de chacun des processus.

6. EXEMPLE : GESTION DE LA MEMOIRE PAR WINDOWS NT

Avec les adresses exprimées sur 32 bits, WINDOWS NT offre à chaque processus 2^{32} octets = 4 Go de mémoire virtuelle : 2 Go pour le SE (threads en mode noyau) et 2 Go pour les programmes (*threads* en mode utilisateur et en mode noyau) paginés en pages de 4 Ko.

WINDOWS NT protège sa mémoire grâce aux mécanismes suivants :

- espace d'adressage virtuel différent pour chaque processus

- fonctionnement en mode noyau pour l'accès au code et aux données du système

6.1 Le gestionnaire de la mémoire virtuelle

Il optimise l'utilisation de la mémoire selon plusieurs techniques :

- **évaluation différée** : plusieurs processus peuvent accéder aux mêmes données en lecture, sans les dupliquer. Si l'un des processus veut modifier des données, la page physique est recopiée ailleurs dans la mémoire; l'espace d'adressage virtuel du processus est mis à jour

- **protection de mémoire par page** : chaque page contient le mode d'accès autorisé en mode noyau et en mode utilisateur (lecture, lecture/écriture, exécution, pas d'accès, copie à l'écriture, page de garde)

Le gestionnaire de la mémoire virtuelle fournit des services natifs aux processus :

- lecture/écriture en mémoire virtuelle

- blocage des pages virtuelles en mémoire physique

- protection des pages virtuelles

- déplacement des pages virtuelles sur disque

- allocation de la mémoire en deux phases : réservation (ensemble d'adresses réservées pour un processus) et engagement (mémoire retenue dans le fichier d'échange avec le disque du gestionnaire de la mémoire virtuelle)

Le gestionnaire de pages utilise des techniques classiques :

- **pagination à la demande** : à la suite d'un défaut de pages, on charge les pages requises *plus quelques pages voisines*

- **placement des pages virtuelles en mémoire** : à la première page physique disponible

- **remplacement** selon la stratégie FIFO d'une page virtuelle d'un processus, suite à un défaut de page

- **table des pages à deux niveaux**

En outre, lorsqu'une page est requise suite à un défaut de page, et qu'elle est non utilisée, un *indicateur de transition* permet de déceler si elle peut devenir utilisable sans la faire transiter par le disque.

Toute page physique peut prendre l'un des **6** états suivants utilisés dans la base de données des pages physiques :

- **valide** : page utilisée par un processus

- **à zéro** : libre et initialisée avec des 0 pour des raisons de sécurité ("niveau C2")

- **modifiée** : son contenu a été modifié et n'a pas encore été recopié sur disque

- **mauvais** : elle a généré des erreurs au contrôle

- **en attente** : retirée d'un ensemble de pages de travail

Le gestionnaire des pages étant réentrant, il utilise un verrou unique pour protéger la base de données des pages physiques et n'en autoriser l'accès qu'à une tâche (thread) simultanée.

6.2 La mémoire partagée

Définition : c'est de la mémoire visible par plusieurs processus ou présente dans plusieurs espaces d'adressage virtuels

WINDOWS NT utilise des objets-sections pour utiliser des vues de fenêtres de la zone de mémoire partagée à partir de différents processus. La mémoire partagée est généralement paginée sur disque.

EXERCICES

1. On considère la table de segments suivante :

segment	base	longueur
0	540	234
1	1254	128
2	54	328
3	2048	1024
4	976	200

Représenter la mémoire. Calculer les adresses physiques correspondant à :
(0 , 128) , (1, 99) , (4 , 100) , (3 , 888) , (2 , 465) , (4 , 344)

2. Dans un système à pagination de 100 octets, donner la suite des pages correspondant aux adresses suivantes : 34 , 145 , 10 , 236, 510, 412, 789
Proposer une formule de calcul.

3. Pourquoi combine-t-on parfois pagination et segmentation ? Décrire un mécanisme d'adressage où la table des segments et les segments sont paginés.

Soit un processus dont la table des segments est :

segment	base	longueur
0	540	234
1	1254	128
2	54	328

La taille de page est 100 octets et les cadres sont alloués dans l'ordre suivant :
3, 4, 9, 0, 8, 7, 6, 11, 15, 5, 17

Représenter graphiquement la nouvelle table des segments et la mémoire (le processus étant entièrement chargé).
Traduire les adresses suivantes : (0 , 132) , (2 , 23) , (2 , 301)

4. Soit une machine à 3 pages dans laquelle le temps de traitement d'un défaut de page est 1 ms s'il n'y a pas de recopie, 3 ms s'il y a recopie. Le processus en cours effectue 5 ms de calcul entre deux références de page. Les bits R et M sont remis à 0 toutes les 25 ms de calcul, mais le traitement d'un défaut de page suspend l'horloge.
L'astérisque désignant un accès en écriture, w = 0 2* 4 1* 2 3* 0 4* 2* 4 3* 4* 5 3* 2, calculer le temps total d'exécution du processus avec les algorithmes :
   - FIFO seconde chance,
   - NFU avec un compteur sur 4 bits

Comparer avec le remplacement optimal.

5. Soit le programme C :

```
int A [1000], B [1000], C [1000], i;
main ()
{
```

```
        for i = 0; i < 1000; i++) C [i] = A [i] + B [i] ;
}
```

La MC est de 2 K mots et un entier occupe un mot. Le code du programme et la variable i occupent 256 mots.
Calculer le nombre et le taux (nombre de défauts de pages/nombre de référencements) des défauts de pages lorsque :

- une page = 128 mots, taille de la zone de données = 3 pages
- une page = 128 mots, taille de la zone de données = 6 pages
- une page = 256 mots, taille de la zone de données = 6 pages

pour les algorithmes FIFO, LRU et remplacement optimal.