CHAPITRE 5 :

LA GESTION DE LA MEMOIRE

Objectifs spécifiques

- Connaître le principe de gestion de mémoire en monoprogrammation
- Connaître le principe de gestion de mémoire en multiprogrammation
- Connaître le principe de gestion de mémoire avec va et vient
- Connaître la notion de compactage
- Connaître la notion de mémoire virtuelle et son utilité,
- Connaître le concept de pagination et son principe
- Connaître les différents algorithmes de remplacement et maitriser leur application
- Connaître le concept de segmentation.

Eléments de contenu

I. Introduction à la gestion de la mémoire

II. Gestion sans recouvrement ni pagination

III. Gestion avec recouvrement sans pagination

IV. Gestion avec recouvrement, avec pagination ou segmentation

Volume Horaire :

Cours : 4 heures 30

TD : 3 heures

5.1 Introduction

La mémoire physique sur un système se divise en deux catégories :

- la mémoire vive : composée de circuit intégrés, donc très rapide.
- la mémoire de masse (secondaire) : composée de supports magnétiques (disque dur, bandes magnétiques...), qui est beaucoup plus lente que la mémoire vive.

La mémoire est une ressource rare. Il n'en existe qu'un seul exemplaire et tous les processus actifs doivent la partager. Si les mécanismes de gestion de ce partage sont peu efficaces l'ordinateur sera peu performant, quelque soit la puissance de son processeur. Celui-ci sera souvent interrompu en attente d'un échange qu'il aura réclamé. Si un programme viole les règles de fonctionnement de la mémoire, il

en sera de même. Dans ce chapitre on va mettre l'accent sur la gestion de la mémoire pour le stockage des processus concurrents.

5.2 Gestion sans recouvrement ni pagination

5.2.1 La monoprogrammation : Le modèle de base

La représentation de base, pour un système monoprocesseur et mono tâche, est montrée dans figure suivante : il s'agit d'une partition contiguë.

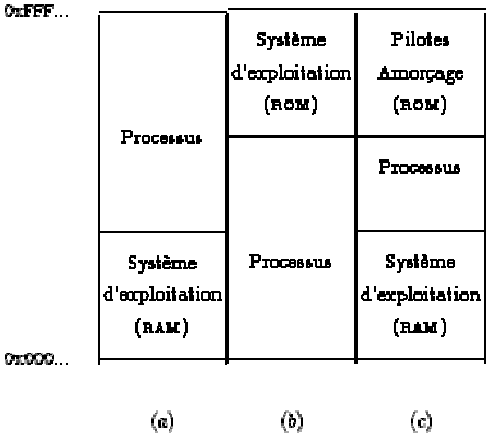


Figure 3 : Les trois organisations de base pour un système mono tâche.

En général, le système d'exploitation se trouve au niveau des premières adresses de la zone mémoire de la RAM. Pour des systèmes avec un SE embarqué (consoles de jeu, téléphones mobiles, *etc*) le système se trouve souvent dans une partie non modifiable (ROM).

Afin de garantir de façon transparente mais flexible, l'amorçage d'un système quelconque, on retrouve souvent une combinaison des deux approches (RAM et ROM). La ROM contient alors un système minimal permettant de piloter les périphériques de base (clavier -- disque -- écran) et de charger le code d'amorçage à un endroit bien précis. Ce code est exécuté lors de la mise sous tension de l'ordinateur, et le « vrai » système d'exploitation, se trouvant dans la zone d'amorçage sur le disque, est ensuite chargé, prenant le relais du système minimal.

5.2.2 La multipligrammation

a. Multiprogrammation avec partitions fixes

i. Partition fixes de tailles égales

Cette partition peut être faite une fois pour toute au démarrage du système par l'opérateur de la machine, qui subdivise la mémoire en partitions fixes de tailles égales. Chaque nouveau processus est placé dans une partition vide. Ceci engendrera le problème de fragmentation interne due au faite que si

la taille de partition est supérieure à l'espace recueilli par un certain processus le reste de cette partition restera vide ce qui cause une perte d'espace mémoire.
Le sous système chargé de la gestion de mémoire met en place une structure des données appelée table des partitions ayant pour rôle l'indication de l'état (libre ou occupée) de chaque partition en identifiant chacune soit par son adresse soit par son numéro.

ii. Partition fixes de tailles inégales

Dans ce cas la mémoire est subdivisée en partitions de tailles inégales. Chaque nouveau processus est placé dans la file d'attente de la plus petite partition qui peut le contenir. Cette façon de faire peut conduire à faire attendre un processus dans une file, alors qu'une autre partition pouvant le contenir est libre.
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. L'alternative à cette approche consiste à n'utiliser qu'une seule file d'attente : dès qu'une partition se libère, le système y place le premier processus de la file qui peut y tenir. Cette solution réduit la fragmentation interne de la mémoire.

b. Multiprogrammation avec partitions variables

Au lancement du système, on crée une seule zone libre de taille maximale. Lorsqu'on charge un programme, on le place dans une zone libre suffisante, et on lui alloue **exactement** la mémoire nécessaire. Le reste devient une nouvelle zone libre. Lorsqu'un programme s'achève, sa partition redevient libre, et peut éventuellement grossir une zone libre voisine. Il n'y a donc ce qu'on appelle fragmentation externe.

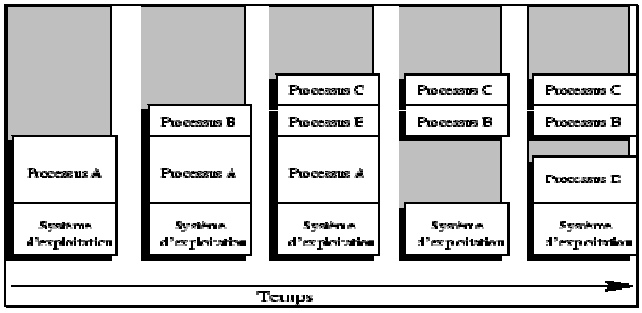


Figure 4: Principe d'allocation de partitions de taille variable.

La solution la plus flexible est d'adopter un système avec des partitions de taille variable et un système de va-et-vient qui permet d'utiliser le disque comme mémoire secondaire et d'y stocker un nombre de processus inactifs ou en attente.

5.3 Gestion avec recouvrement sans pagination

Dès que le nombre de processus devient supérieur au nombre de partitions, il faut pouvoir simuler la présence en mémoire centrale (MC) de tous les processus pour pouvoir satisfaire au principe d'équité et minimiser le temps de réponse des processus. La technique du recouvrement 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.

5.3.1 Le va et vient

Conceptuellement le va-et-vient, ou *swap*, se comporte exactement comme la mémoire vive, à la différence près qu'on ne peut y exécuter des processus (pour exécuter un processus sur le *swap*, il faut le charger en mémoire vive), ainsi que quelques considérations liées au médium de stockage, qui impose un accès et un stockage par blocs, mais que l'on peut négliger.

Un processus qui est inactif (soit bloqué, soit prêt) peut donc être placé dans le swap. Les stratégies de choix des processus à écrire sur disque sont sensiblement identiques à celles mises en œuvre pour l'ordonnancement des processus. Il est à noter qu'il existe deux types de solution : soit on alloue une place fixe dans le swap, pour chaque processus créé, soit on utilise le swap comme une grande zone de stockage dans laquelle les processus sont écrits selon les besoins et à un endroit déterminé en fonction de l'occupation au moment de l'écriture.

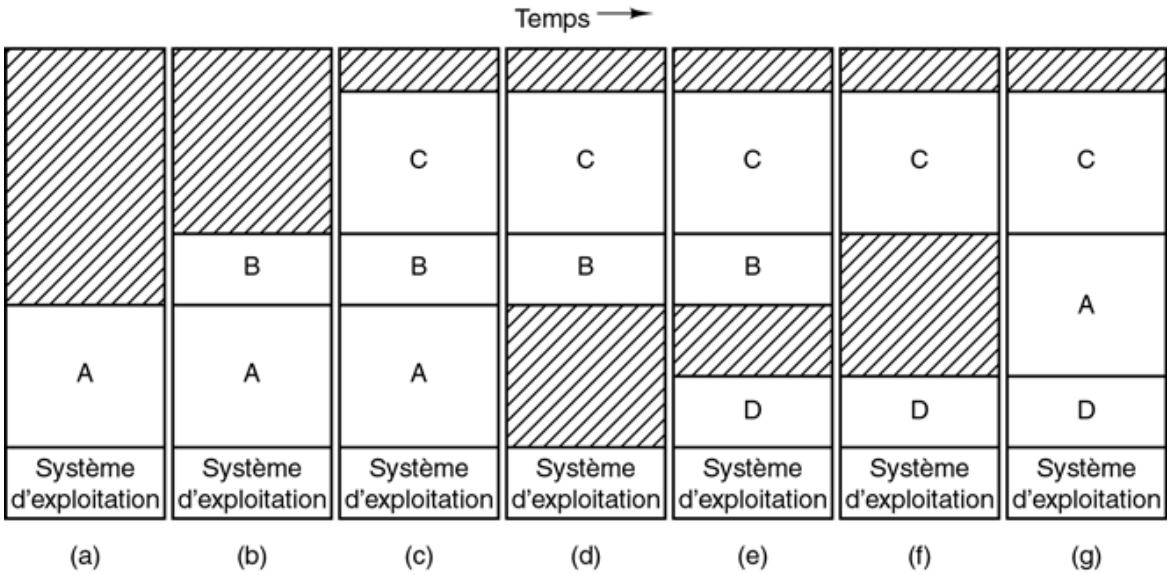


Figure 5 : Gestion de la mémoire avec va et vient

5.3.2 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).

Comme on le voit sur le schéma suivant on distingue trois méthodes de compactage :

- La première consiste tout simplement à recopier de mémoire à mémoire le programme à déplacer. Elle monopolise le processeur pour effectuer la copie ;
- La seconde effectue une copie du programme à déplacer vers le disque, puis une seconde copie du disque vers la mémoire au nouvel emplacement. Curieuse *a priori*, cette méthode s'explique et se justifie si le transfert de mémoire à disque et de disque à mémoire est effectué par un canal d'E/S, processeur spécialisé dans les échanges, qui laisse donc le processeur libre pendant les transferts. La recopie coûte toutefois un certain ralentissement du processeur, par vol de cycles.
- La dernière méthode est encore plus curieuse, mais se déduit de la seconde. On utilise deux canaux d'E/S, et on les boucle. L'utilisation de deux canaux est plus coûteuse que l'usage

d'un disque, mais la copie est plus rapide. Il faut tout de même noter que les deux canaux vont voler des cycles au processeur de calcul, et l'un à l'autre, ce qui ralentit un peu.

Rq : Quelle que soit la méthode choisie, le compactage est coûteux.

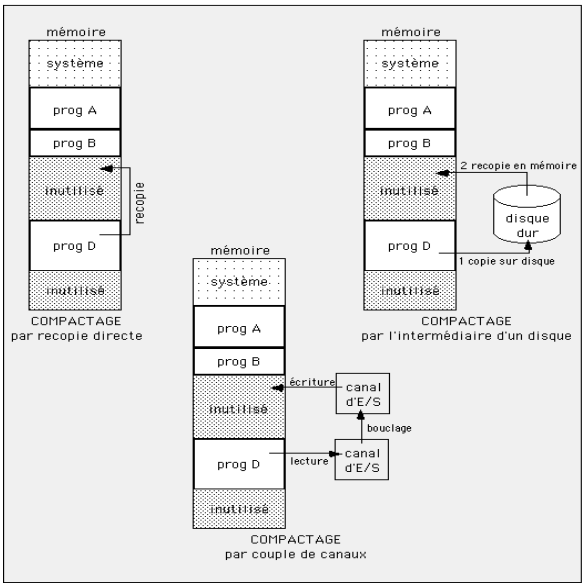


Figure 6 : Différentes stratégies de compactage

5.3.3 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 algorithmes d'allocation possibles :

- **First-fit** : la première zone libre : en parcourant la liste libre, on retient la première zone suffisante ; on alloue la partie suffisante pour le programme, et le reste devient une nouvelle zone libre plus petite.
- **Best-fit :** le meilleur ajustement, on recherche dans la liste la plus petite zone possible. Cette méthode est plus coûteuse et peut créer ainsi de nombreuses petites unités résiduelles inutilisables (fragmentation nécessitant un compactage ultérieur), mais évite de couper inutilement une grande zone.
- **worst fit :** le plus grand résidu, on recherche la zone qui laissera le plus petit résidu avec le risque de fractionnement regrettable des grandes unités.

5.3.4 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.

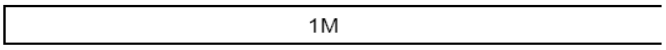
A l'achèvement d'un processus ou à son transfert sur disque, il faut du temps (mise à jour des listes chaînées) pour examiner si un regroupement avec ses voisins est possible pour éviter une fragmentation excessive de la mémoire.

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

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

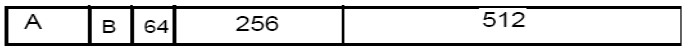
Cet algorithme utilise l'existence d'adresses binaires pour accélérer la fusion des zones libres adjacentes lors de la libération d'unités. Le gestionnaire mémorise une liste de blocs libres dont la taille est une puissance de 2 (1, 2, 4, 8 octets, ..., jusqu'à la taille maximale de la mémoire).

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

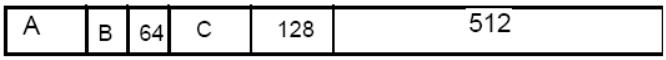


Un processus A demande 70 Ko : la mémoire est fragmentée en deux compagnons (buddies) de 512 Ko; l'un d'eux est fragmenté en deux blocs de 256 Ko; l'un d'eux est fragmenté en deux blocs de 128 Ko et on loge A dans l'un d'eux, puisque 64 < 70 < 128 :

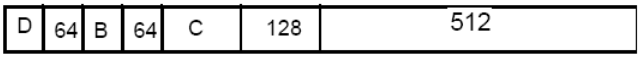
Un processus B demande 35 Ko : l'un des deux blocs de 128 Ko est fragmenté en deux de 64 Ko et on loge B dans l'un d'eux puisque 32 < 35 < 64 :



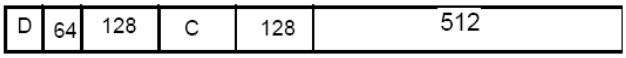
Un processus C demande 80 Ko : le bloc de 256 Ko est fragmenté en deux de 128 Ko et on loge C dans l'un d'eux puisque 64 < 80 < 128 :



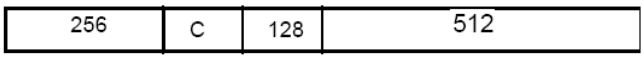
A s'achève et libère son bloc de 128 Ko. Puis un processus D demande 60 Ko : le bloc libéré par A est fragmenté en deux de 64 Ko, dont l'un logera D :



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 2n + 1 octets utilisera un bloc de 2n+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.

5.4 Gestion avec recouvrement, avec pagination ou segmentation

5.4.1 Notion de mémoire virtuelle

La taille d'un processus doit pouvoir dépasser la taille de la mémoire physique disponible, même si l'on enlève tous les autres processus. Afin d'assurer une extension de la mémoire il existe 2 manières :

- En découpant un programme en une partie résidente en mémoire vive et une partie chargée uniquement en mémoire lorsque l'accès à ces données est nécessaire.
- En utilisant un mécanisme de **mémoire virtuelle**, consistant à utiliser le disque dur comme mémoire principale et à stocker uniquement dans la RAM les instructions et les données utilisées par le processeur. Le système d'exploitation réalise cette opération en créant un fichier temporaire (appelé fichier **SWAP**, traduit "**fichier d'échange**") dans lequel sont stockées les informations lorsque la quantité de mémoire vive n'est plus suffisante. Cette opération se traduit par une baisse considérable des performances, étant donné que le temps d'accès du disque dur est extrêmement plus faible que celui de la RAM. Lors de l'utilisation de la mémoire virtuelle, il est courant de constater que la LED du disque dur reste quasiment constamment allumée et dans le cas du système Microsoft Windows qu'un fichier appelé "*win386.swp*" d'une taille conséquente, proportionnelle aux besoins en mémoire vive, fait son apparition.

La **Mémoire Virtuelle** est une mémoire idéale, dont les adresses commencent à 0, et de capacité non limitée ; elle a pour but principal de pouvoir exécuter des processus sans qu'ils soient logés en mémoire en leur totalité ; 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. On appelle **Espace Virtuel** l'ensemble des adresses possibles ; il est fixé par le nombre de bits qui constitue une adresse virtuelle. On parlera d'adresse réelle et d'adresse virtuelle. Toute la difficulté consiste à traduire une adresse virtuelle (A.V.) en une adresse réelle (A.R.) accessible sur la machine.

5.4.2 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**.

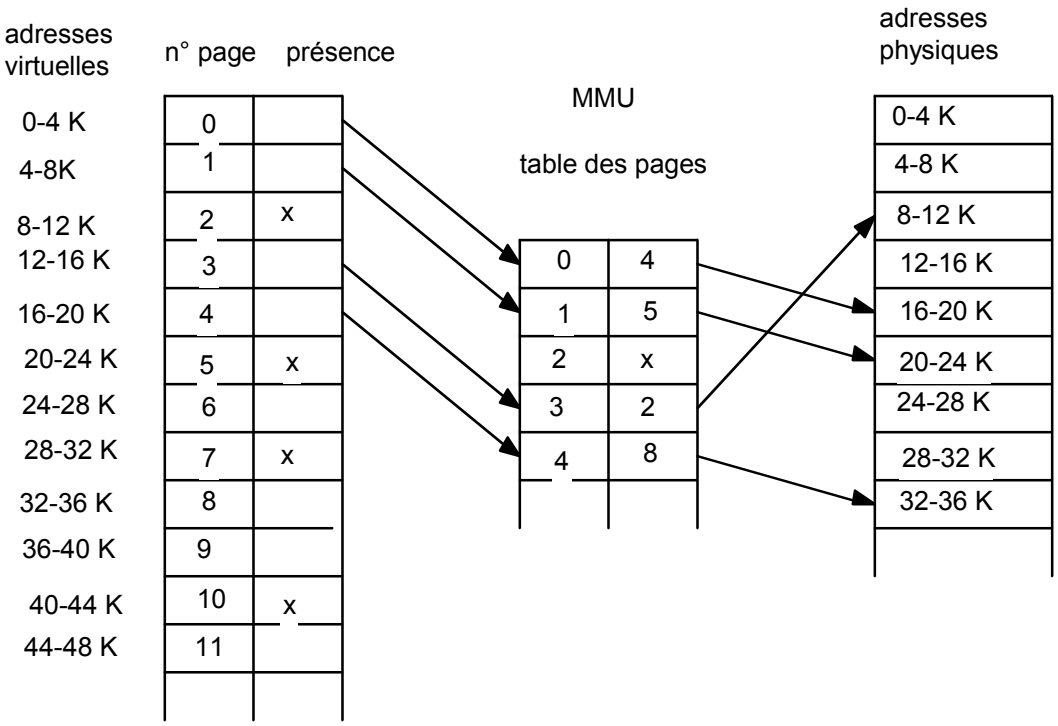
Pour un processus, le système ne chargera que les pages utilisées. Mais la demande de pages à charger peut être plus élevée que le nombre de cadres disponibles. Une gestion de l'allocation des cadres libres est nécessaire.

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 (page victime)
- 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

La sélection de page est réalisée par les algorithmes de remplacement. Le principe de fonctionnement de la majorité de ces algorithmes repose sur le principe de localité. Ces algorithmes sont en général divisés en deux grandes catégories :

- les algorithmes dépendants de l'utilisation des données: LRU, LFU, etc…
- les algorithmes indépendants de l'utilisation des données : aléatoire, FIFO.

Ci-dessous sont listés quelques d'algorithmes.

a. Algorithme optimal

Cet algorithme, utilise la mémoire cache de manière optimale : il retire la page qui sera référencée le plus tard possible. Cet algorithme doit connaître pour chaque page le nmbre d'instructions à exécuter avant quelle ne soit référencée cause pour laquelle set algorithme est irréalisable. Cet algorithme constitue néanmoins un excellent moyen afin de mesurer l'efficacité d'un algorithme de remplacement en fournissant une référence.

b. 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.

En cas de défaut de page, on retire une page au hasard dans la catégorie non vide de plus petit index (RM).

c. 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 anciennement 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.

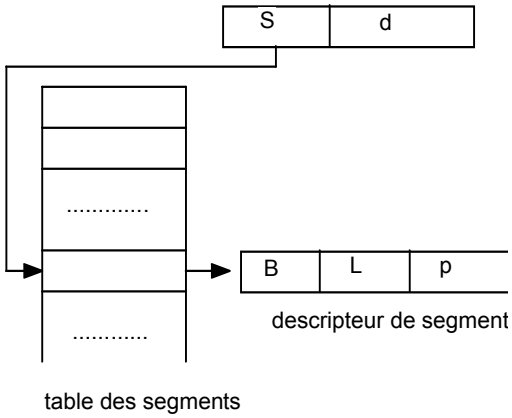
d. LRU (least recently used)

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.

5.4.3 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 ∈ [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.