CHAPITRE IV : GESTION DE LA MEMOIRE

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. Le but de ce chapitre est de décrire les problèmes et les méthodes de gestion de la mémoire principale.

4.1 HIERARCHIE DES MEMOIRES :

La grande variété de systèmes de stockage dans un système informatique peut être organisée dans une hiérarchie (voir figure) selon leur vitesse et leur coût.

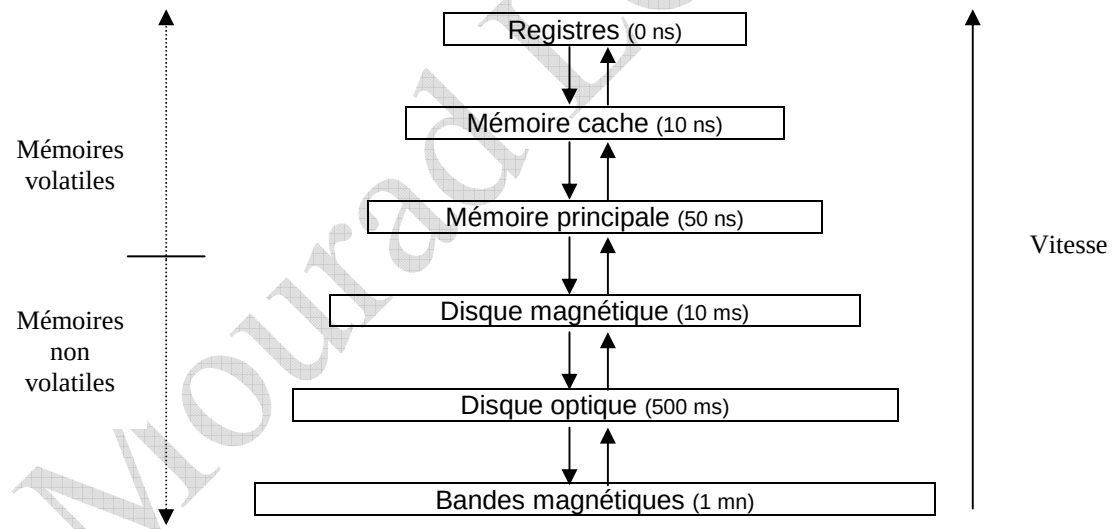


Fig. 4.1 La hiérarchie mémoire

Les niveaux supérieurs sont chers mais rapides. Au fur et à mesure que nous descendons dans la hiérarchie, le coût par bit diminue alors que le temps d'accès augmente.

En plus de la vitesse et du coût des divers systèmes de stockage, il existe aussi le problème de la *volatilité* de stockage. Le stockage volatile perd son contenu quand l'alimentation électrique du dispositif est coupée.

La *mémoire cache* (antémémoire) :

La mise en mémoire cache est un principe important des systèmes informatiques autant dans le matériel que dans le logiciel. Quand une information est utilisée, elle est copiée dans un système de stockage plus rapide, la mémoire cache, de façon temporaire. Quand on a besoin d'une information particulière, on vérifie d'abord si elle se trouve dans la mémoire cache, sinon on ira la chercher à la source et on mémorise une copie dans la mémoire cache, car on suppose qu'il existe une grande probabilité qu'on en aura besoin une autre fois.

4.2 ADRESSAGE LOGIQUE/ADRESSAGE PHYSIQUE :

Une adresse générée par le processeur est appelée ***adresse logique***. Tandis qu'une adresse vue par l'unité mémoire, c'est à dire celle qui est chargée dans le registre d'adresse de la mémoire, est appelée ***adresse physique***.

Il est nécessaire au moment de l'exécution de convertir les adresses logiques en adresses physiques. Par exemple, imaginons un système où une adresse physique est obtenue en ajoutant à chaque adresse logique l'adresse de base contenue dans un registre.

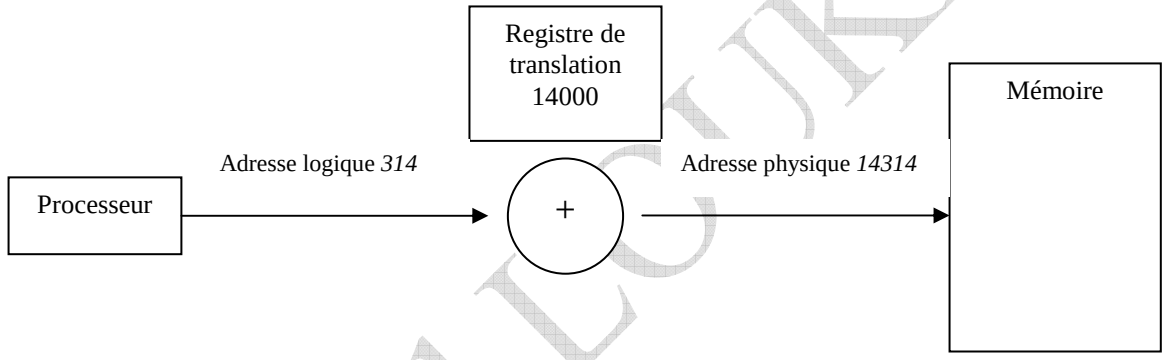


Fig. 4.2 Conversion d'adresses logiques en adresses physiques par translation.

Dans ce schéma, la valeur du registre de translation est additionnée à chaque adresse logique générée par un processus utilisateur. Par exemple, si l'adresse de base est 14000, un accès à l'emplacement 314 est converti à l'emplacement 14314.

Il est à noter que le programmeur n'aperçoit en général pas les adresses physiques ; il manipule uniquement des adresses logiques.

4.3 ALLOCATION CONTIGUË DE LA MEMOIRE :

La mémoire principale peut loger le SE et les différents processus utilisateurs. La mémoire est habituellement subdivisée en deux partitions : une pour le SE résident et l'autre pour les processus utilisateurs

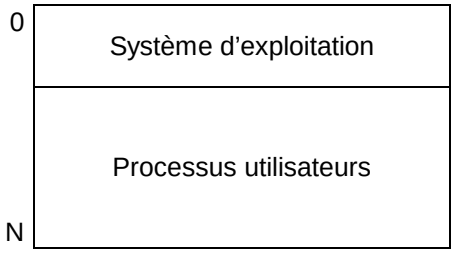


Fig. 4.3 Partition de la mémoire

4.3.1 Allocation multi-partitions :

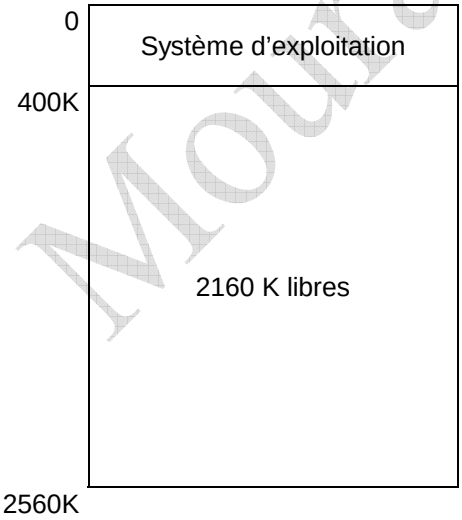
Dans un environnement multiprogrammé, plusieurs processus logent dans la mémoire en même temps. On doit alors prendre en charge le problème de leur allouer la mémoire disponible.

L'un des schémas les plus simples revient à subdiviser la mémoire en partitions de taille fixe. Chaque répartition peut contenir exactement un processus. Ainsi le degré de la multiprogrammation est limité par le nombre de partitions. Quand une partition devient libre, on sélectionne un processus de la file d'attente des processus prêts et on le charge dans la partition libre. Quand le processus se termine, la partition devient libre pour un autre processus. Cette technique a été implémentée sur le OS/360 d'IBM, mais elle est actuellement dépassée.

Une autre méthode consiste à obliger le SE à maintenir une table indiquant les partitions de mémoire disponibles et celles qui sont occupées. Quand un processus arrive en demandant de la mémoire, on recherche un espace suffisamment grand pour contenir ce processus. Si nous en trouvons un, nous allouons seulement la quantité de mémoire nécessaire, laissant le reste disponible pour satisfaire les futures requêtes.

Exemple : L'état de la mémoire d'un système est décrit par la figure suivante. Le système a une file de travaux décrit par le tableau suivant :

Etat de la mémoire

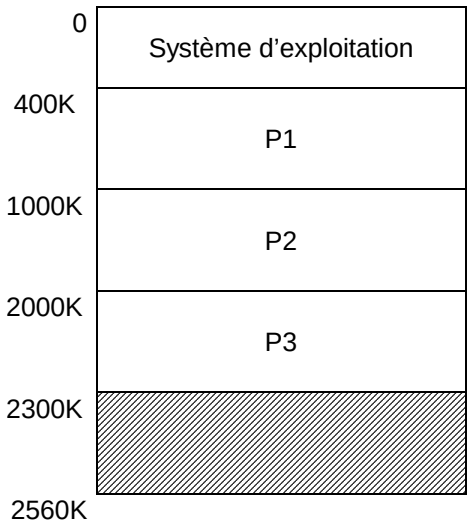


File d'attente des travaux

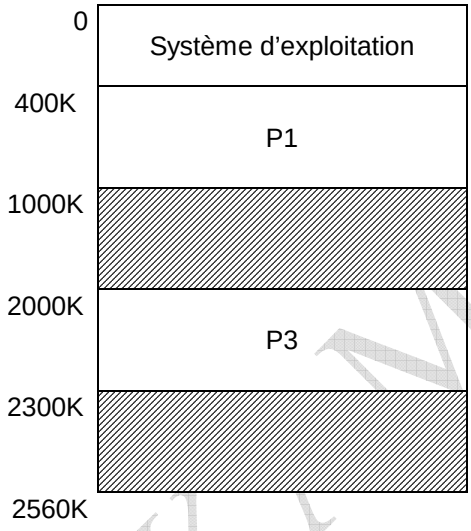
Processus	Mémoire	Temps
P1	600 K	10
P2	1000 K	5
P3	300 K	20
P4	700 K	8
P5	500 K	15

Les figures suivantes montre les différents états successifs de la mémoire après les entrées et les sorties de processus.

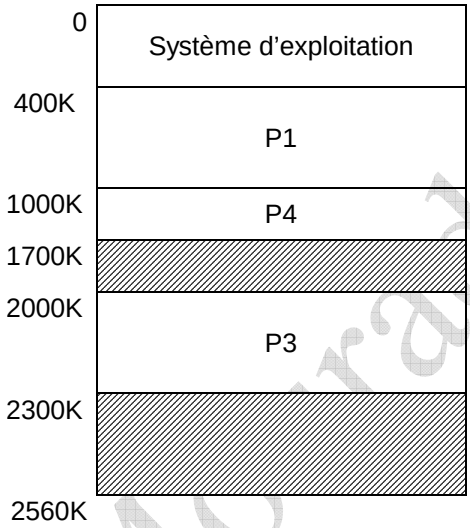
P1, P2 et P3 entrent



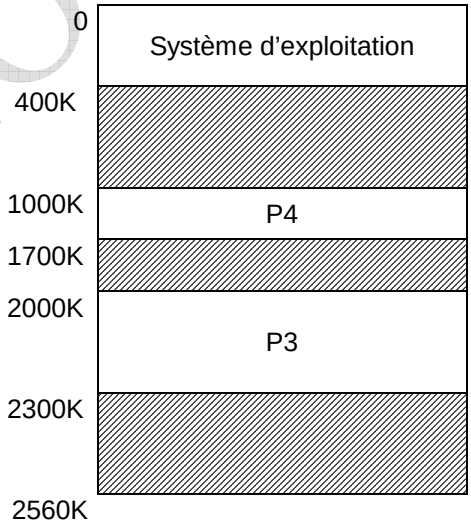
P2 termine



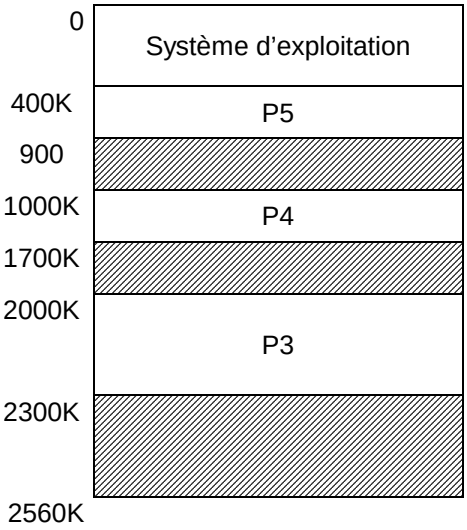
P4 entre



P1 termine



P5 entre

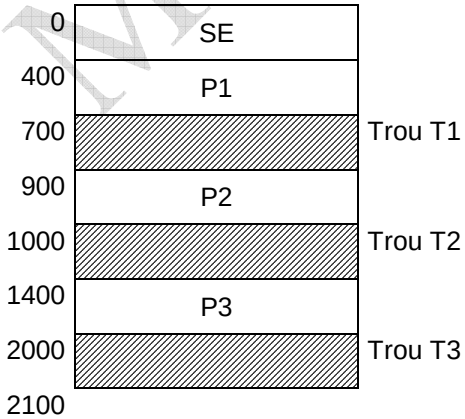


Ainsi à chaque instant, on dispose d'une liste de tailles de blocs disponibles et la file d'attente des processus prêts. Le SE peut scheduler la file d'attente selon un algorithme de scheduling. La mémoire est allouée aux processus jusqu'à ce qu'il n'existe aucun bloc de mémoire suffisamment grand pour contenir ce processus.

Mais comment allouer un espace de taille n à un processus à partir d'une liste de trous libres ?. On peut utiliser l'un des trois algorithmes suivant : *first-fit*, *best-fit*, *worst-fit*.

- Algorithme *First-fit* (Le premier trouvé) : On alloue au processus le premier trou suffisamment grand.
- Algorithme *Best-fit* (Le meilleur choix) : On alloue au processus le trou le plus petit suffisamment grand ; c'est à dire celui qui provoquera la plus petite miette possible. Cet algorithme nécessite le parcours de toute la liste des espaces libres.
- Algorithme *Worst-fit* (Le plus mauvais choix) : On alloue au processus le trou le plus grand. Cet algorithme nécessite le parcours de toute la liste des espaces libres.

Exemple : L'état de la mémoire d'un système est décrit par la figure suivante. On suppose qu'un processus P demande un espace mémoire de 80K.

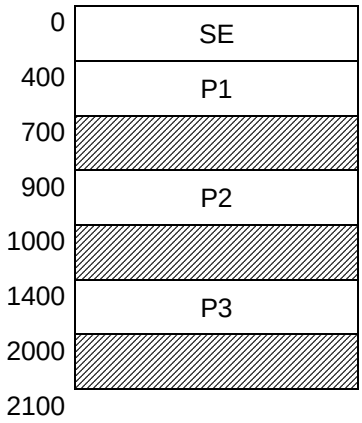


En fonction de l'algorithme choisi, on choisirait le trou T1, T2 ou T3.

Algorithme First-Fit : trou T1.
Algorithme Best-Fit : trou T3.
Algorithme Worst-Fit : trou T2.

4.3.2 La fragmentation :

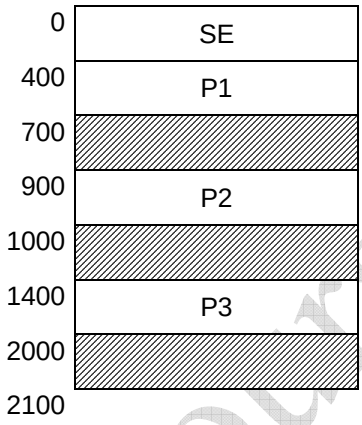
Les algorithmes d'allocation de la mémoire contiguë précédents, produisent la fragmentation de la mémoire. En effet, suite aux différentes entrées et sorties de processus, des miettes séparées se froment dans la mémoire.



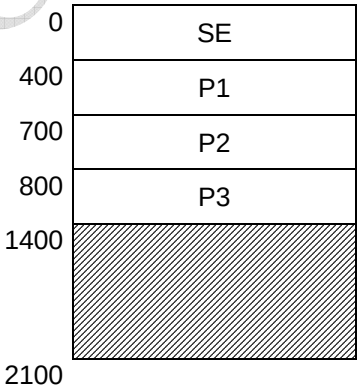
Si le processus P désire entrer dans le système en occupant 500K, il ne pourrait pas bien que l'espace total disponible est de 700 K.

La solution au problème de la fragmentation peut être le compactage. Le principe est de rassembler tous les trous en un seul bloc.

Avant compactage



Après compactage



Inconvénient : L'opération de compactage est une opération très coûteuse pour le SE.

4.4 LA PAGINATION :

Une autre solution au problème de la fragmentation consiste à permettre que l'espace d'adressage logique d'un processus ne soit plus contigu, autorisant ainsi l'allocation de la mémoire physique à un processus là où elle est disponible. La technique de la pagination est une manière implémenter cette solution.

4.4.1 Le principe de la pagination :

La pagination consiste à découper la mémoire physique en blocs de taille fixe, appelés cadres physiques ou cadres de pages. La mémoire physique est également subdivisée en blocs de la même

taille appelée pages. Quand on doit exécuter un processus, on charge ses pages dans les cadres de pages à partir de la mémoire auxiliaire.

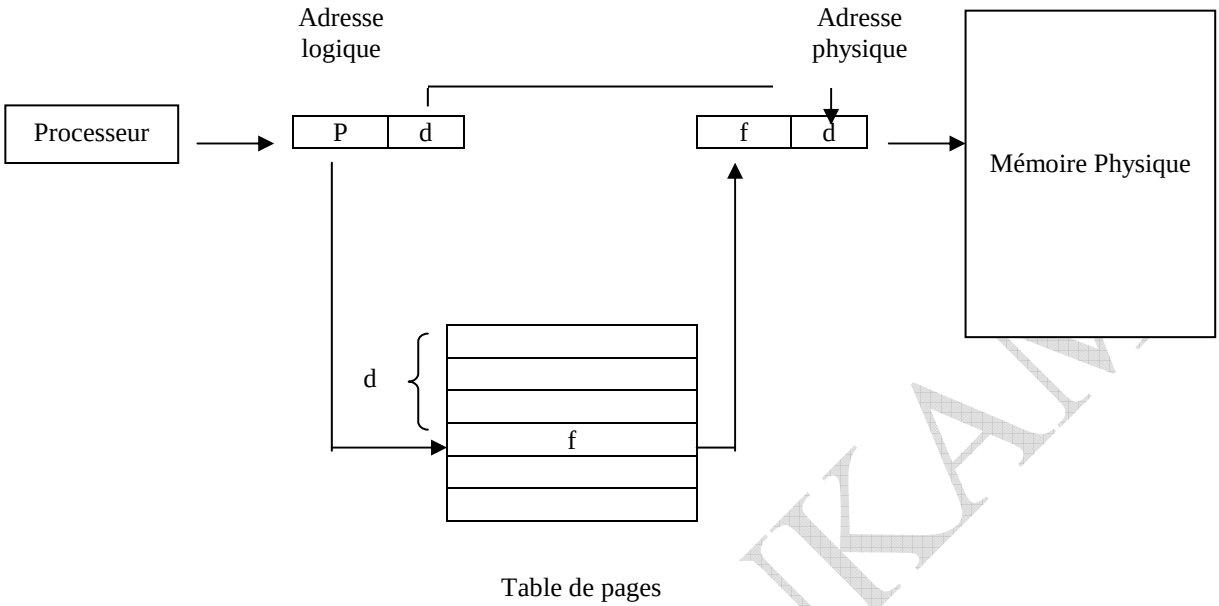
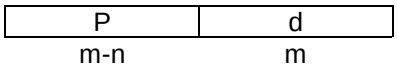


Fig. 4.4 Principe de la pagination.

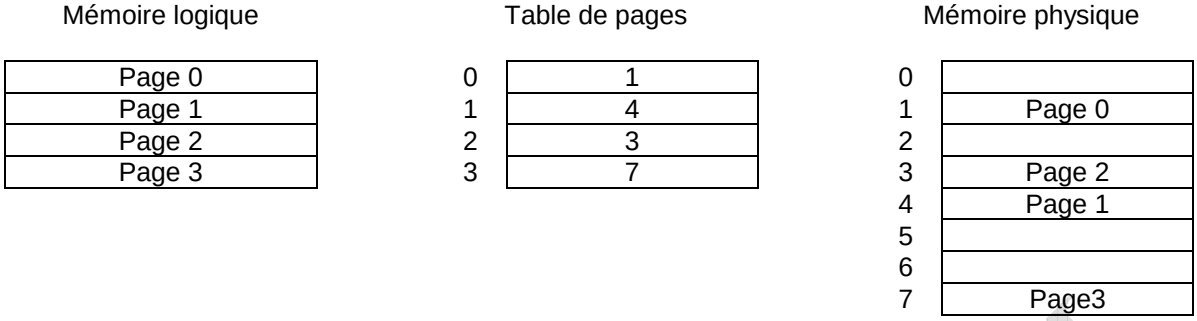
On divise chaque adresse générée par le processeur en deux parties : numéro de page P et déplacement dans la page D.

On utilise le numéro de page comme indice à une *table de pages*. La table de pages contient l'adresse de base de chaque page dans la mémoire physique. Cette adresse de base est combinée avec le déplacement dans la page afin de définir l'adresse physique.

La taille de la table de page est une puissance de 2 variant entre 512 octets et 8192 octets, selon l'architecture de l'ordinateur. Le choix d'une puissance de 2 comme taille de page facilite la traduction d'une adresse logique en un numéro de page et un déplacement dans la page. Si la taille de l'espace adresse logique est de 2^m et la taille de la page est de 2^n unités d'adressage (mots ou octets), les m-m bits de poids forts d'une adresse logique désignent le numéro de page et les n bits de poids faible désignent le déplacement dans la page.



L'exemple suivant montre un système ayant une mémoire logique de 4 pages et une mémoire physique de 8 pages.



Les cadres de pages sont alloués comme des unités. Si les besoins en mémoire d'un processus ne tombent pas sur les limités des pages, le dernier cadre de page peut ne pas être plein. Par exemple, si les pages sont de 2048 octets, un processus de 72766 octets aura besoin de 35 pages, plus 1086 octets. On lui allouera donc 36 cadres de pages, en provoquant une fragmentation de 2048-1086=962 octets.

Dans le pire des cas, un processus demandera n pages plus un octet. On lui allouera n+1 cadres de pages, provoquant ainsi une fragmentation de presque une page entière. Cette considération suggère qu'il est souhaitable d'avoir des pages de petites tailles. Cependant, il existe une surcharge due à chaque entrée de la table de pages et elle est réduite si la taille de la page augmente. De nos jours, les pages sont généralement d'une taille de 2 à 4 Ko.

Un aspect important de la pagination est la séparation nette entre la vue de l'utilisateur de la mémoire et la mémoire physique réelle. Le programme utilisateur conçoit cette mémoire comme un espace unique contigu, mais en réalité le programme utilisateur est éclaté dans toute la mémoire physique qui contient éventuellement d'autres programmes.

4.4.2 Structure d'une table de pages :

Chaque SE possède sa propre méthode pour stocker la table de pages. La plupart allouent une table de page pour chaque processus. On stocke un pointeur sur la table de pages avec les autres valeurs des registres dans le PCB. Lors d'une commutation de contexte, et quand le processus redémarre, sa table de pages est rechargée.

4.4.3 Protection :

La protection de la mémoire dans un environnement paginé est accomplie avec des bits de protection associés à chaque cadre de page. Normalement, ces bits sont stockés dans la table de pages. Un bit peut définir que l'on accède à une page en lecture et écriture, ou en lecture seulement.

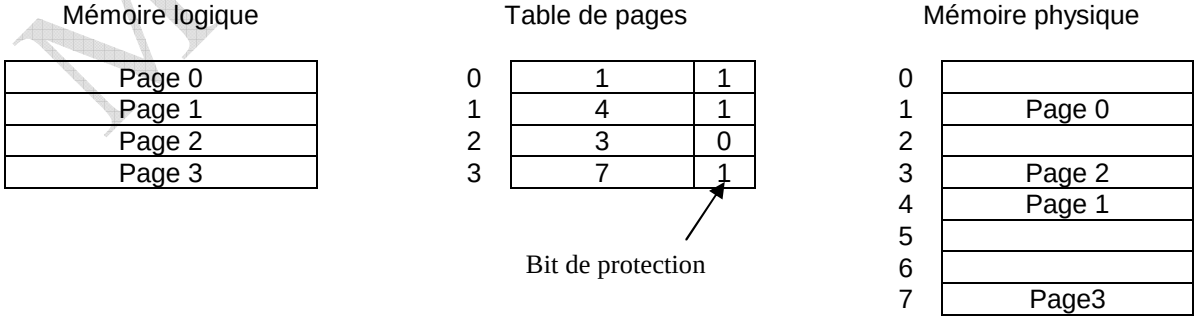


Fig. 4.5 Protection des pages.

4.4.4 Pagination multiniveaux :

La plupart des SE modernes supportent un espace adresse logique très grand (2^{32} à 2^{64}). Dans un tel environnement, la table de pages elle-même devient excessivement grande. Par exemple, si l'espace adresse est de 2^{32}, et si la taille d'une page est de 4Ko (2^{12}), alors la table de pages peut posséder jusqu'à 1 million d'entrée ($2^{32}/2^{12}$). Il convient alors de subdiviser la table de pages en parties plus petites. Par exemple, on peut utiliser un schéma de pagination à 2 niveaux, dans lequel la table de pages elle-même est paginée.

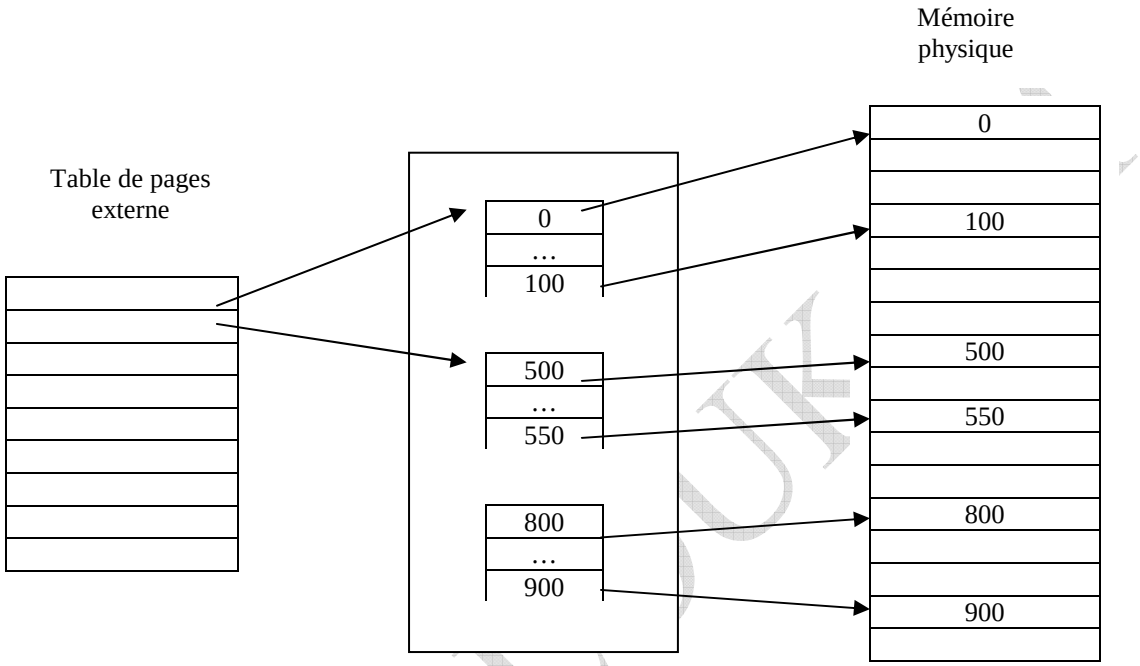


Fig. 4.6 Pagination à 2 niveaux

Dans un système de pagination à 2 niveaux, une adresse logique a la forme suivante :

P1	P2	d
N° de page	N° de page	Déplacement dans la page

P1 est le numéro de la table de pages interne. P2 est le déplacement dans la page de la table de page externe.

La traduction d'une adresse logique en adresse physique dans un système paginé à 2 niveaux est présentée par le schéma suivant :

P1	P2	d

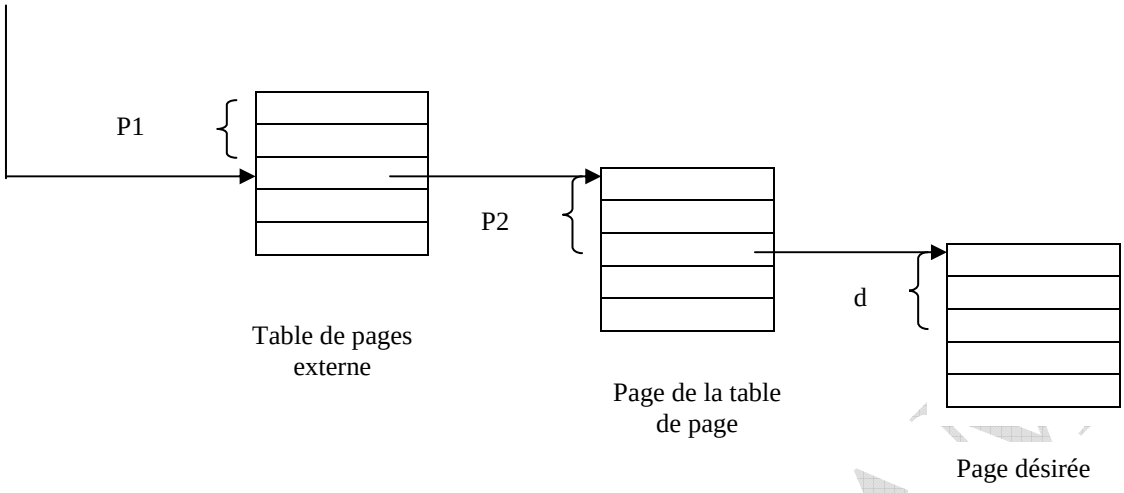


Fig. 4.7 Traduction d'adresses pour une architecture de pagination à 2 niveaux.

4.4.5 Pages partagées :

Un autre avantage de la pagination est la possibilité de partager du code commun. Cette caractéristique est particulièrement importante dans un environnement partagé. Imaginons un système qui supporte 40 utilisateurs, chacun d'eux exécutant un éditeur de textes. Si l'éditeur est constitué de 150 Ko de code et 50 Ko de données, on aura besoin de 8000 Ko pour satisfaire les 40 utilisateurs. Cependant si le code est *réentrant*, il peut être partagé comme le montre le schéma suivant :

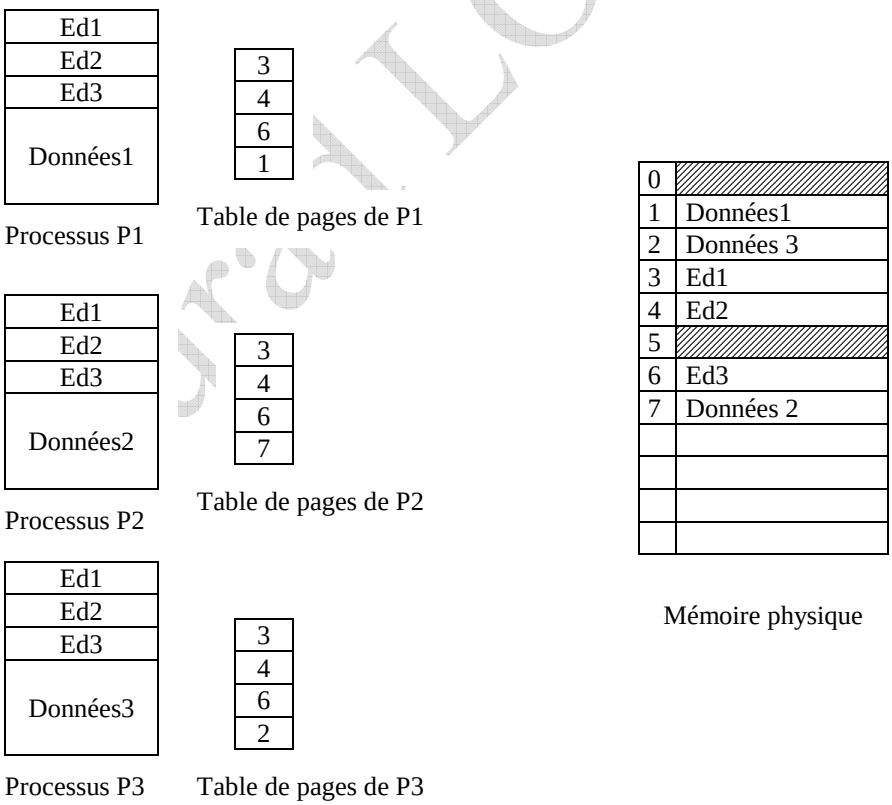


Fig. 4.8 Partage de code dans un environnement de pagination.

Le schéma précédant montre que les 3 pages de code de l'éditeur de textes sont partagées par les 3 processus. Chaque processus, par contre, possède sa propre page de code.

Le code *réentrant* (encore appelé code pur) est un code invariant ; c'est à dire qu'il ne change jamais pendant l'exécution. Ainsi, deux ou plusieurs processus peuvent exécuter le même code en même temps. Chaque processus possède sa propre copie de registres et son stockage de données.

On doit maintenir en mémoire physique une seule copie de l'éditeur. La table de pages de chaque processus correspond à la même structure physique de l'éditeur, mais les cadres de pages sont transformés à des cadres de pages différents. Ainsi, pour supporter 40 utilisateurs nous n'avons besoin que d'une copie de l'éditeur (ie 150 Ko), plus 40 copies de l'espace de données de 50 Ko par utilisateur. L'espace requis est donc de 2150 Ko au lieu de 8000 Ko.

4.5 LA SEGMENTATION :

La segmentation rejoint la pagination pour consacrer un principe important dans les SE actuels, à savoir la séparation entre la vue de l'utilisateur de la mémoire et la mémoire physique. La vue de l'utilisateur de la mémoire n'est pas le même que la mémoire physique réelle ; le SE se charge de convertir l'une dans l'autre.

Il est admis que le programmeur n'imagine pas la mémoire comme un tableau linéaire d'octets. Il préfère voir la mémoire comme un ensemble de *segments* de taille variable, sans qu'il y ait un ordre entre les segments.

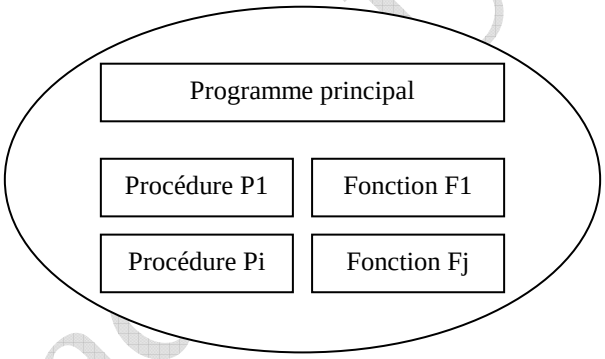
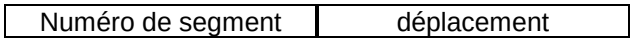


Fig. 4.9 Exemples de segments.

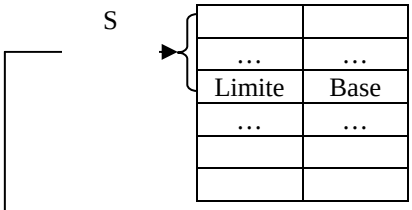
La segmentation consiste à découper l'espace logique en un ensemble de segments. Chacun des segments est de longueur variable ; la taille dépend de la nature du segment dans le programme. Chaque segment possède un numéro et une longueur. Pour cibler une zone mémoire spécifique, on doit donc désigner *deux* paramètres : un numéro de segment et un déplacement. :



De ce point de vue , la segmentation est différente de la pagination, puisque dans la pagination, l'utilisateur ne spécifie qu'une seule adresse qui est décodée par le SE en un numéro de page et un déplacement.

En segmentation la conversion d'une adresse logique en une adresse physique est faite grâce à une table de segments. Chaque entrée de la table de segment possède deux valeurs :

- La base : c'est l'adresse de début du segment en mémoire.
- La limite : spécifie la taille du segment.



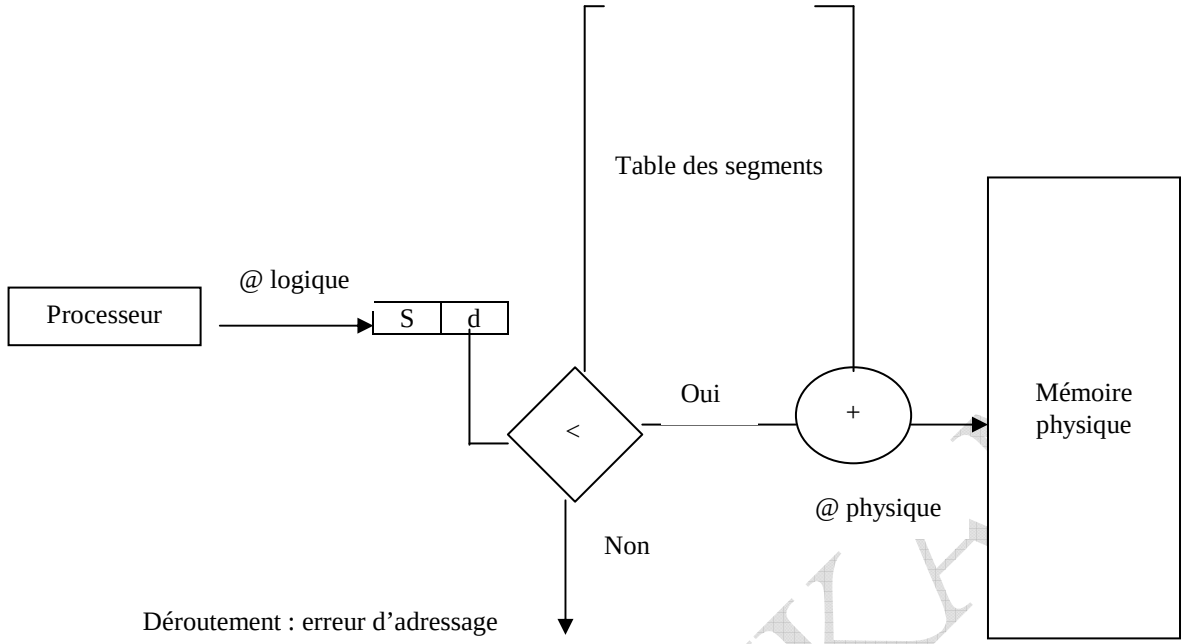
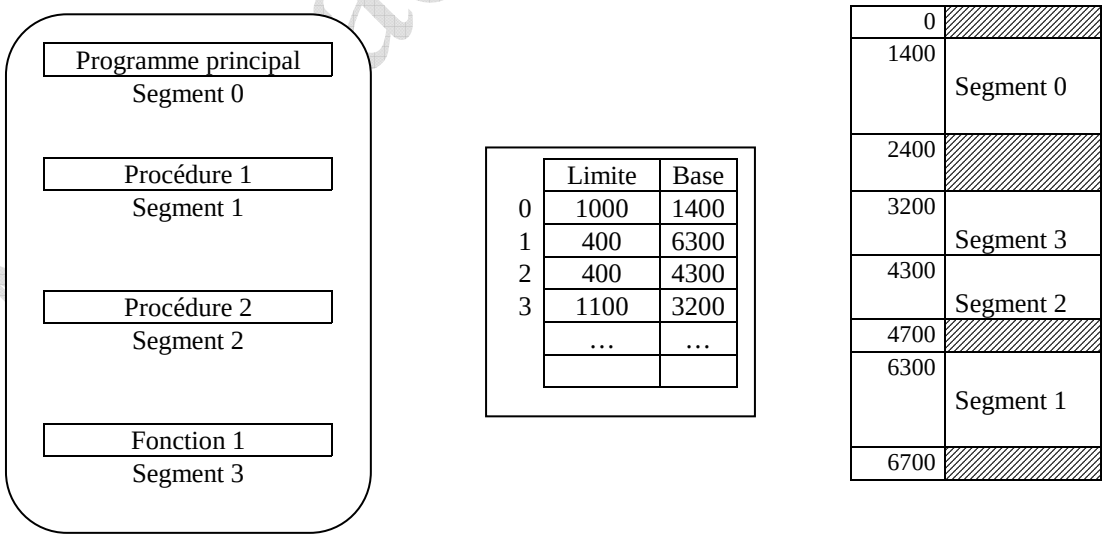


Fig. 4.10 Conversion d'adresse logique en adresse physique – cas de la segmentation

Une adresse logique est constituée de 2 parties : un numéro de segment S et un déplacement dans ce segment d. On utilise le numéro de segment comme indice dans la table de segments. Le déplacement d de l'adresse logique doit se trouver entre 0 et la limite du segment. Si ce n'est pas le cas, il y a un déroutement vers le SE pour tentative d'adressage hors limite. Si le déplacement est correct, on l'additionne à la base du segment pour calculer l'adresse physique de l'emplacement désiré.

Exemple : Soit un espace d'adressage logique contenant quatre segments.



Faisons la conversion des adresses logiques suivantes en adresses physiques :

Adresse logique		Adresse physique
N°de segment	Déplacement	
2	53	4300+53=4353
3	852	3200+852=4052
0	1222	Provoque un déroutement vu que le déplacement est hors limite

4.6 MEMOIRE VIRTUELLE :

Les techniques de gestion de la mémoire vue précédemment possèdent le même objectif : maintenir plusieurs processus en mémoire simultanément afin de permettre la multiprogrammation. Cependant, elles tendent à demander que le processus se trouve en mémoire dans sa totalité avant de pouvoir être exécuté.

La *mémoire virtuelle* est une technique autorisant l'exécution de processus pouvant ne pas être complètement en mémoire. Le principal avantage de ce schéma est que les programmes peuvent être plus grand que la mémoire physique.

Le concept de mémoire virtuelle a été mis au point après que l'on a constaté que dans de nombreux cas, on n'a pas besoin d'un programme dans sa totalité. Exemples :

- Les programmes possèdent souvent du code pour manipuler des situations exceptionnelles. Ce code n'est alors que très rarement exécuté.
- On alloue souvent aux tables et tableaux plus de mémoire que ce dont ils ont réellement besoin. On peut déclarer un tableau 100x100, alors qu'on n'utilise que 10x10 éléments.

Ainsi la possibilité d'exécuter un programme se trouvant partiellement en mémoire présenterait plusieurs bénéfices :

- Un programme n'est plus limité par la quantité de mémoire physique disponible. Les utilisateurs pourront écrire des programmes pour un espace adresse virtuel extrêmement grand, simplifiant ainsi la tâche de programmation.
- Comme chaque programme utilisateur pourrait occuper moins de mémoire physique, il serait possible d'exécuter plus de programme en même temps.

Lé mémoire virtuelle fournit donc un espace d'adressage extrêmement grand alors que la mémoire physique est limitée. Cela est possible en utilisant une mémoire auxiliaire comme espace de travail pour charger et décharger les différentes pages par le SE.

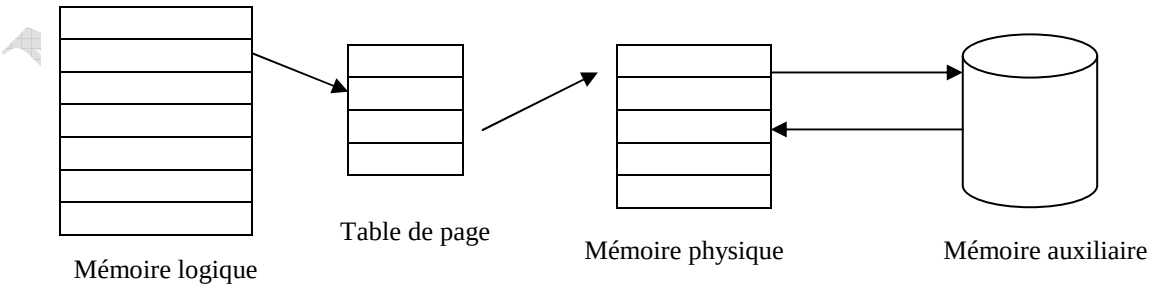


Fig. 4.11 Mémoire virtuelle plus grande que la mémoire physique

4.6.1 Pagination à la demande :

La mémoire virtuelle est communément implémentée avec la pagination à la demande. Quand un processus doit être transféré en mémoire, la routine de pagination devine quelles pages seront utilisées avant que le processus soit mis en mémoire axilliaire de nouveau. Au lieu de transférer en mémoire un processus complet, la routine de pagination ramène seulement les pages qui lui sont nécessaires. Ainsi, elle évite que l'on charge en mémoire des pages qui ne seront jamais employées, en réduisant le temps de swaping et la quantité de mémoire dont on a besoin.

Avec cette technique, le SE doit disposer de moyens pour distinguer les pages qui sont en mémoire, et celles qui sont sur disque. Par exemple, on peut utiliser dans la table des pages un bit valide/invalide pour décrire si la page est chargée en mémoire ou non.

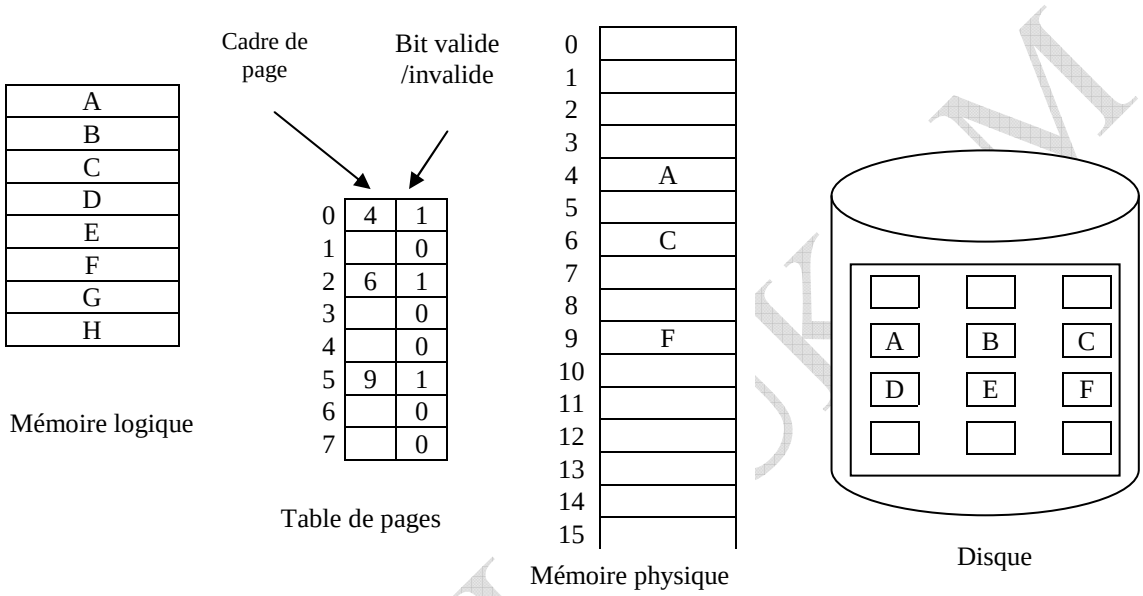


Fig. 4.12 Mémoire paginée avec certaines pages en mémoire.

Que se passe-t-il si un processus essaie d'utiliser une page qui n'est pas en mémoire ?. L'accès à une page marquée invalidé provoque *un défaut de page*. En essayant d'accéder à cette page, il y a un déroutement vers le SE. La procédure permettant de traiter ce défaut de page est la suivante :

- S'assurer que la référence de la page est correcte.
- S'assurer que la page désirée est bien en mémoire auxilliare.
- Trouver un cadre de page libre et charger la page.

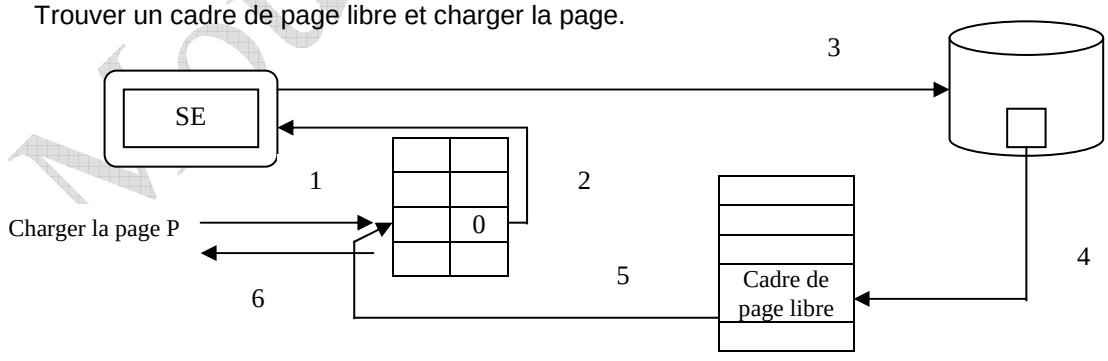


Fig. 4.13 Procédure de traitement d'un défaut de page.

Légende :

1 : on fait référence à la page P qui n'existe pas en mémoire.

2 : Il y a un déroutement vers le SE.
3 : Le SE vérifie si la page existe bien en mémoire auxilliare.
4 : Le SE vérifie s'il y a un cadre de page libre, auquel cas il y charge la page absente.
5 : Le SE met à jour la table de page.
6 : Redémarrer l'instruction du processus qui a provoqué le défaut de page.

4.6.2 Remplacement de pages :

Lorsque le SE se rend compte au moment de charger une page qu'il n'existe aucun cadre de page disponible, il peut faire recours à un remplacement de page. Ainsi le code complet d'une procédure de traitement d'un défaut de pages est le suivant :

1. Trouver l'emplacement de la page désirée sur disque.
2. Trouver un cadre de page libre. S'il existe un cadre de pages libre, l'utiliser, sinon utiliser un algorithme de remplacement de pages pour sélectionner un cadre de page victime.
3. Enregistrer la page victime dans le disque et modifier la table de page.
4. Charger la page désirée dans le cadre de page récemment libéré et modifier la table de pages.
5. Redémarrer le processus utilisateur.

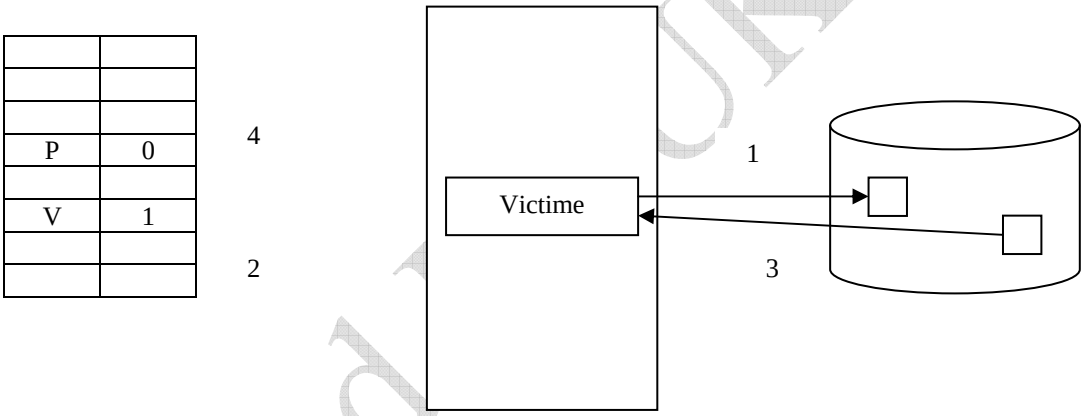


Fig. 4.14 Remplacement de pages.

Légende :

1 : Swap out de la page victime.
2 : Mettre le bit de la page à invalide
3 : Swap in de la page désirée.
4 : Mettre le bit de la page à valide

4.6.3 Algorithmes de remplacement de pages :

Il existe plusieurs algorithmes différents de remplacement de pages. En général, on souhaite celui qui provoquera le taux de défauts de pages le plus bas.

On évalue un algorithme en l'exécutant sur une séquence particulière de références mémoires et en calculant le nombre de défauts de page. Cette chaîne est appelée : chaîne de références.

Afin de déterminer le nombre de défauts de pages pour une chaîne de références et un algorithme de remplacement particulier, on doit également connaître le nombre de cadres de pages disponibles.

Evidemment, au fur et à mesure que le nombre de cadres de pages augmente, le nombre de défauts de pages doit diminuer, comme le montre la figure suivante :

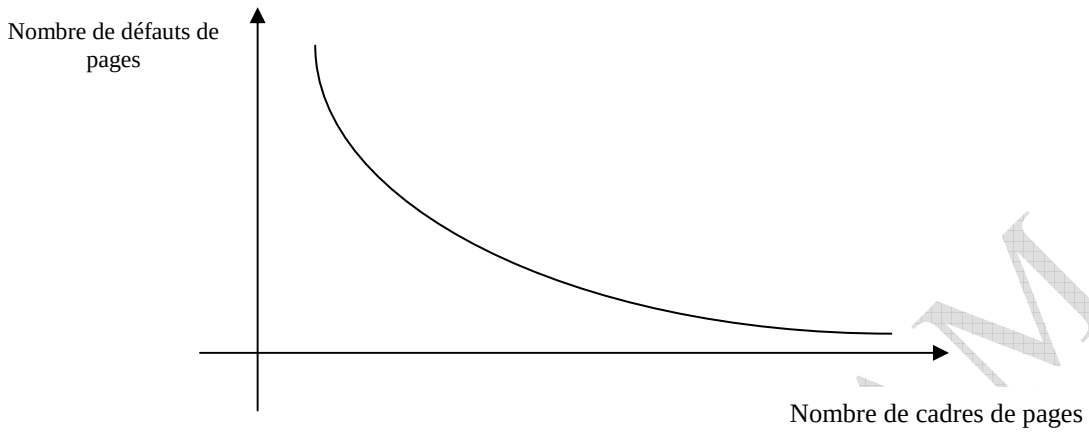


Fig. 4.15 Défauts de pages et cadres de pages

4.6.3.1 Algorithme FIFO :

L'algorithme de remplacement FIFO (First In First Out) est le plus simple à réaliser. Avec cet algorithme, quand on doit remplacer une page, c'est la plus ancienne qu'on doit sélectionner.

Exemple : Considérons un système à mémoire paginée ayant 3 cadres de pages (pages physiques), et soit la chaîne de références suivante : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1. Quel est le nombre de défauts de pages si l'algorithme de remplacement choisi est l'algorithme FIFO ?.

Pour répondre à cette question, faisons un déroulement qui fait ressortir les états successifs du système et en marquant les différents défauts de pages.

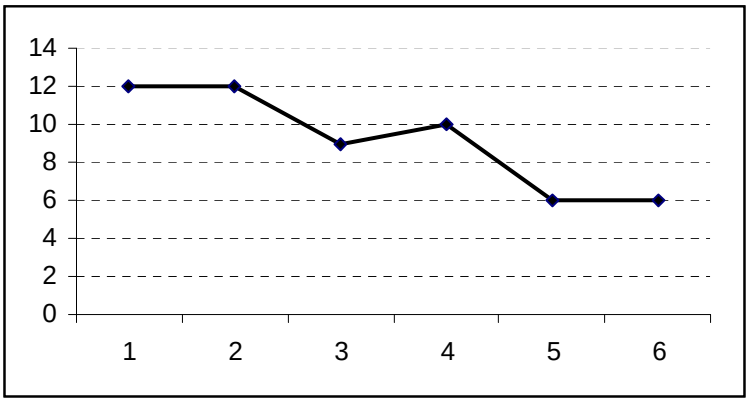
Chaîne de référence →	7	0	1	2	0	3	0	4	2	3
Cadres de page	7	7	7	2	2	2	2	4	4	4
		0	0	0	0	3	3	3	2	2
			1	1	1	1	0	0	0	3
Défauts de page →	X	X	X	X		X	X	X	X	X

	0	3	2	1	2	0	1	7	0	1
	0	0	0	0	0	0	0	7	7	7
	2	2	2	1	1	1	1	1	0	0
	3	3	3	3	2	2	2	2	2	1
	X			X	X			X	X	X

Nombre de défauts de pages : 15.

L'algorithme de remplacement FIFO est simple à implémenter mais ses performances ne sont pas toujours bonnes.

Afin d'illustrer les problèmes rencontrés avec un algorithme de remplacement FIFO, on peut envisager la chaîne de références suivantes : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. La figure suivante montre le nombre de défauts de pages en fonction des cadres de pages disponibles.



Cadres de pages	Défauts de pages
1	12
2	12
3	9
4	10
5	6
6	6

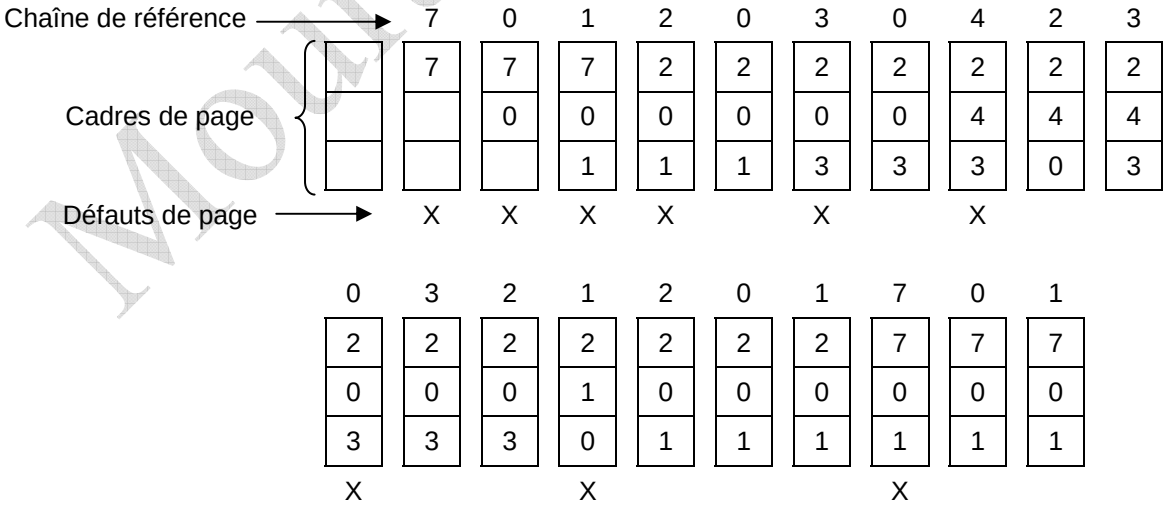
Fig. 4.16 Anomalie de Belady

On remarque que le nombre de défauts de pages pour 4 cadres de pages (ie : 10) est supérieur au nombre de défauts de pages pour 3 cadres de pages (ie : 9) !. Ce résultat fort inattendu est connu sous le nom de l'anomalie de Belady. Elle décrit une situation où le taux de défauts de pages augmente au fur et à mesure que le nombre de cadres de pages augmente, contrairement à la règle générale.

4.6.3.2 Algorithme optimal :

L'algorithme de remplacement optimal fournit le taux de défauts de pages le plus bas de tous les autres algorithmes, de plus il ne souffre pas de l'anomalie de Belady. Son principe est de remplacer la page qui mettra le plus de temps à être de nouveau utilisée.

Exemple : Reprenons la même chaîne de références que l'algorithme FIFO précédent, et calculons le nombre de défauts de pages avec cet algorithme.



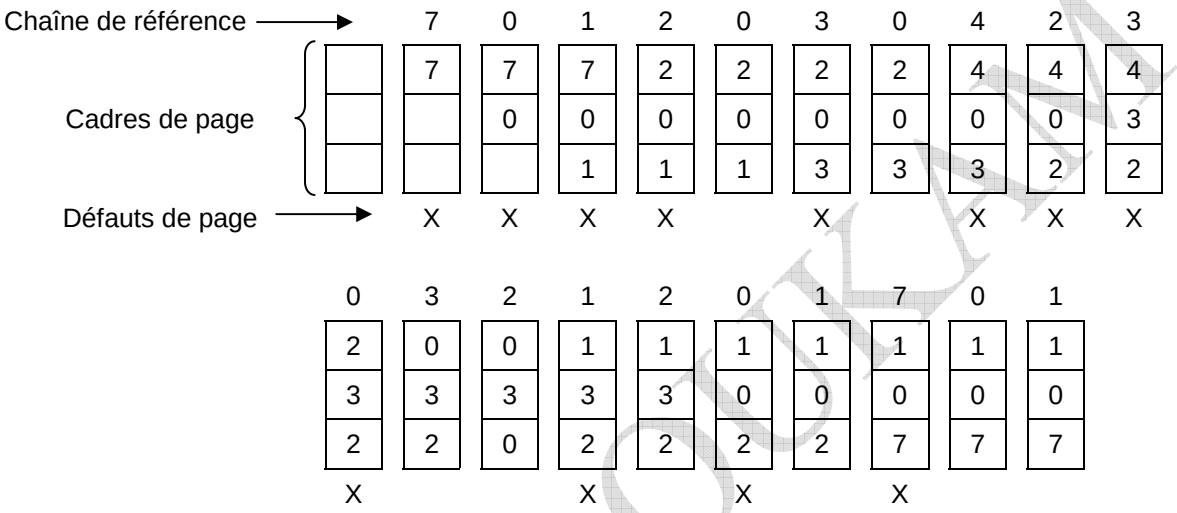
Nombre de défauts de pages : 09.

Malheureusement, l'algorithme optimal est difficile à mettre en œuvre car il requiert une connaissance future de la chaîne de références. Il est utilisé essentiellement pour faire des études comparatives.

4.6.3.3 Algorithme LRU :

L'algorithme LRU (Least Recently Used) sélectionne pour le remplacement d'une page victime, la page la moins récemment utilisée. C'est à dire qu'on doit remplacer une page, l'algorithme sélectionne la page qui n'a pas été utilisée pendant la plus grande période de temps.

Exemple : Reprenons la même chaîne de références que l'algorithme FIFO précédent, et calculons le nombre de défauts de pages avec cet algorithme.



Nombre de défauts de pages : 12.

4.6.3.4 Algorithmes basés sur le comptage :

Il existe de nombreux algorithmes de remplacement de pages basés sur le comptage du nombre de références effectuées à chaque page. Pour cela, on utilise un compteur pour chaque numéro de page.

A. L'algorithme LFU :

L'algorithme de remplacement LFU (Least Frequently Used) requiert que l'on remplace la page la moins fréquemment utilisée ; c'est à dire celle ayant le compteur le plus petit. La raison de cette sélection est qu'une page activement utilisée devrait avoir un grand compte de références.

Exemple : En reprenant la même chaîne de références que l'algorithme FIFO précédent, le nombre de défauts de pages trouvé avec cet algorithme est de 11.

Cet algorithme présente un problème quand une page est très utilisée pendant la phase initiale d'un programme, mais qu'elle n'est plus employée à nouveau par la suite. Comme elle a été amplement utilisée, elle possède un grand compte et reste donc en mémoire même si elle n'est pas utilisée.

B. L'algorithme MFU :

L'algorithme de remplacement MFU (Most Frequently Used) requiert que l'on remplace la page la plus fréquemment utilisée. La raison de cette sélection est que la page ayant le compte le plus petit vient probablement d'être ramenée en mémoire et doit encore être utilisée.

Exemple : En reprenant la même chaîne de références que l'algorithme FIFO précédent, le nombre de défauts de pages trouvé avec cet algorithme est de 14.

4.6.4 Allocation de cadres de pages :

Comment allouer la quantité de mémoire libre entre les différents processus ?. par exemple, si on dispose de 93 cadres de pages libres et 2 processus, combien de cadres de pages obtient chacun d'eux ?.

La manière la plus simple de répartir m cadres de page entre n processus est de donner à chacun des portions égales, c'est à dire m/n cadres de pages. Par exemple, si on possède 93 cadres de pages et 5 processus, chaque processus obtiendra 18 cadres de pages. Les 3 cadres de pages restant pourraient faire partie d'un pool de buffers de cadres de pages libres. Ce schéma est appelé *allocation équitable*.

Il existe une autre méthode qui admet que les processus ont des besoins différents et demandent des quantités différentes de mémoires. Par exemple, nous disposons d'une mémoire composée de 62 cadres de pages et de deux processus différents : un processus utilisateur qui demande 10 Ko et un processus de Bases de données qui demande 127 Ko.

Cette seconde méthode utilise l'allocation proportionnelle, c'est à dire qu'elle affecte la mémoire disponible entre les processus en fonction de leur besoin. Soit S_i la taille de la mémoire virtuelle d'un processus Pi. On a :

$$S=\Sigma S_i$$

Si le nombre total de cadres de pages disponibles est m, on alloue a_i cadres de pages au processus P_i , où :

$$a_i = (s_i / S) * m$$

Ainsi, le nombre de cadres de pages affectés à chaque processus de notre exemple est :

- Processus utilisateur : (10/137)*62=4 cadres de pages.
- Processus Base de données : (127/137)*62=57 cadres de pages.

4.6.5 Ecroulement :

Si la quantité de cadres de pages allouée à un processus de basse priorité descend en dessous du nombre minimum requis par l'architecture de l'ordinateur, nous devons arrêter l'exécution de ce processus. Nous devrions ensuite transférer les pages restantes sur disque, libérant ainsi tous les cadres de pages qui lui ont été alloués.

En fait, on peut envisager le cas d'un processus ne possédant pas suffisamment de cadres de pages. Bien qu'il soit techniquement possible de réduire le nombre de cadres de pages alloués au minimum, il existe un nombre de pages (supérieur) en utilisation active. Si le processus ne possède pas ce nombre de cadres de pages, il produira très rapidement des défauts de pages. A ce moment là, il doit remplacer une page. Cependant comme toutes ses pages sont en utilisation active, il devra à nouveau remplacer une page nécessaire tout de suite après. Il produira donc très rapidement un défaut de pages à nouveau, et ainsi de suite. Le processus continuera à produire des défauts de pages, en remplaçant des pages dont il aura besoin tout de suite après.

Cette haute activité de pagination est appelée écroulement. Un processus s'écroule s'il passe plus de temps à paginer qu'à s'exécuter.