CHAPITRE 4

Gestion de la Mémoire

IV.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 physique sert de zone de stockage temporaire pour les programmes et données que vous utilisez. De façon générale, plus la quantité de mémoire est importante, plus vous pouvez lancer d'applications simultanément. De plus, 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, quel que soit la puissance de son processeur. D'autre part, plus celle-ci est rapide plus votre système réagit vite, il s'agit donc (pour le système d'exploitation) de l'organiser au mieux pour en tirer le maximum de performances. Pour maximiser le rendement de l'UC dans un système multiprogrammé, plusieurs questions se pose concernant :

- Comment partager les données entre les processus ?
- Comment protéger l'espace mémoire de chaque processus ?
- Qui et comment faire l'allocation d'espace mémoire aux process ?
- Comment et quand la Restitution de l'espace libéré par le process se fait ?

Pour répondre à ces exigences, le SE doit se doter d'un module spécial de gestion de la MC appelé Gestionnaire de la MC -Memory Management Unit (MMU).

Dans ce chapitre, nous verrons qu'il y a plusieurs schémas pour la gestion de la mémoire, avec leurs avantages et inconvénients.

IV.2. Les mémoires

Une mémoire peut être représentée comme une armoire de rangement constituée de différents tiroirs. Chaque tiroir représente alors une case mémoire qui peut contenir un seul élément : des données. Le nombre de cases mémoires pouvant être très élevé, il est alors nécessaire de pouvoir les identifier par un numéro. Ce numéro est appelé « adresse ». Chaque donnée devient alors accessible grâce à son adresse (voir figure 4.1).

Adresse	Case mémoire
7 = 111	
6 = 110	
5 = 101	
4 = 100	
3 = 011	
2 = 010	
1 = 001	
0 = 000	0001 1010

Figure 4.1 : Organisation de la mémoire.

Avec une adresse de **n** bits il est possible de référencer au plus $\mathbf{2^n}$ cases mémoire.

Dans un PC on trouve deux types de mémoires, vives (RAM : Random Access Memory) et mortes (ROM : Read Only Memory).

La grande variété de systèmes de stockage dans un système informatique peut être organisée dans une hiérarchie (voir figure 4.2) selon leur vitesse et leur coût

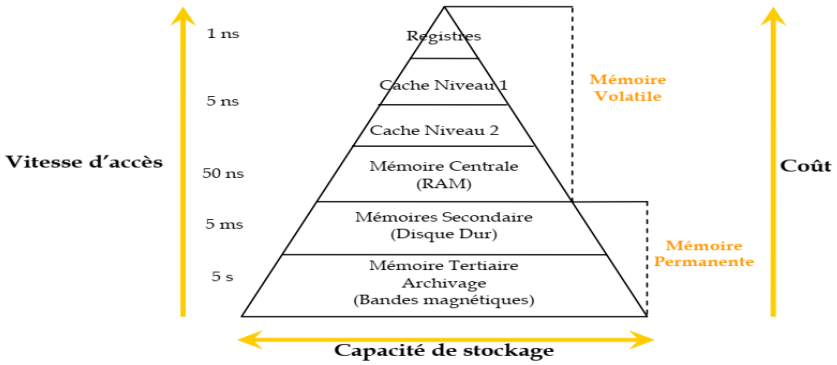


Figure 4.2 : La hiérarchie mémoire.

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.

Généralement, la mémoire contient principalement deux types d'informations :

- Les instructions (informations traitantes : commandes) : qui dictent le traitement des tâches sous forme d'ordres ou de commandes élémentaires,
- Les opérandes (informations traitées) : ce sont les données sur lesquelles est effectué le traitement dicté par les instructions.

D'autre part, deux types d'opérations peuvent s'effectuer sur les mots mémoires ; à savoir la lecture (Read) et l'écriture (Write) :

- Pour lire la mémoire : une adresse doit être choisie à partir du bus d'adresse et le bus de contrôle doit déclencher l'opération. Les données à l'adresse choisie se retrouvent sur le bus de données.
- Pour écrire la mémoire : une adresse doit être choisie à partir du bus d'adresse et le bus de contrôle doit déclencher l'opération. Les données à l'adresse choisie sont remplacées par celles sur le bus de données.

IV.3. Objectifs du gestionnaire de la mémoire centrale

Le gestionnaire de la MC est un module du SE très important ayant la fonction de gérer les accès à la MC. Il vise les objectifs suivants :

- **Organisation logique de la MC (Adressage)** : structuration des données en un ensemble de mots mémoires et convertir les adresses logiques en des adresses physiques au chargement initial et pendant l'exécution.
- **Allocation et Réallocation** : trouver une zone mémoire libre pour le stockage des processus.
- **La libération** : récupérer les espaces-mémoires occupés par des processus ayant terminé leur exécution afin de permettre à d'autres de s'exécuter.

- **La protection** : assurer l'intégrité des espaces-mémoires réservés aux processus, d'où la nécessité des mécanismes de :
  - Contrôle en empêchant les processus d'accéder en dehors de leurs espaces attribués.
  - Récupération en cas de violation mémoire (déroutement).
- **Partager des informations** (données et programmes) entre les process utilisateurs.
- Surpasser les limites physiques de la capacité de la MC.

➔ Ces objectifs doivent être accomplie d'une manière efficace, par augmentation des performances du système informatique et la transparence pour l'utilisateur.

IV.4. Espace d'adressage logique et physique :

Espace d'adressage logique : Une adresse générée par la CPU est appelée « adresse logique ». Elle est également connue sous le nom *d'adresse virtuelle*. L'espace d'adressage logique peut être défini comme la taille du processus. Une adresse logique peut être modifiée.

Espace d'Adressage Physique : Une adresse vue par l'unité mémoire (c'est-à-dire celle chargée dans le registre d'adresses mémoire de la mémoire) est communément appelée « Adresse Physique ». Une adresse physique est également appelée *adresse réelle*. L'ensemble de toutes les adresses physiques correspondant à ces adresses logiques est appelé espace d'adressage physique. Une adresse physique est calculée par MMU. Le mappage au moment de l'exécution des adresses virtuelles aux adresses physiques est effectué par une unité de gestion de la mémoire (MMU) du périphérique matériel. L'adresse physique reste toujours constante.

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.

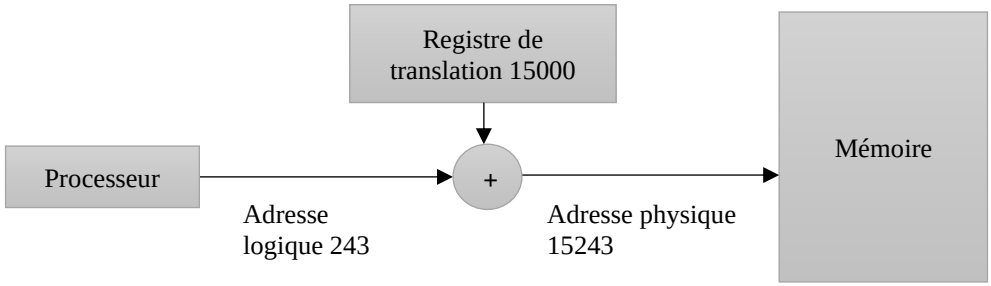


Figure 4.3 : Conversion d'adresses logiques en adresses physiques par translation.

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 15000, un accès à l'emplacement 243 est converti à l'emplacement 15243.
Il est à noter que le programmeur n'aperçoit en général pas les adresses physiques ; il manipule uniquement des adresses logiques.

IV.5. Gestion de la Mémoire centrale dans les systèmes monoprogrammés

Dans un système mono-programmé, la mémoire est subdivisée en deux partitions contiguës,

- Une pour le système d'exploitation résident, par ses différents module (le vecteur d'interruptions, …), souvent placer en mémoire basse
- et l'autre réservé à l'utilisateur actif (elle est totalement à sa disponibilité).

Elle n'autorise qu'un seul processus actif en mémoire à un instant donné dont tout l'espace mémoire usager lui est alloué.

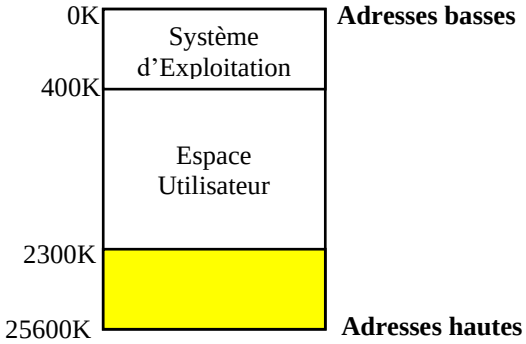


Figure 4.4 : Organisation de la mémoire dans un système mono-programmé.

La gestion de la mémoire s'avère simple ; le système d'exploitation doit garder une trace de deux zones mémoires. L'algorithme d'allocation peut être décrit par l'organigramme ci-contre :

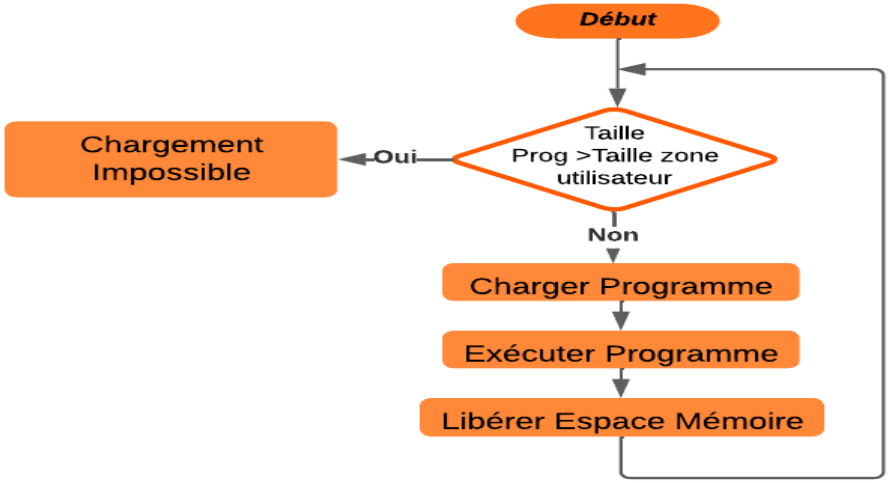


Figure 4.5 : Algorithme d'Allocation mémoire d'un programme utilisateur.

L'inconvénient majeur de cette stratégie est la sous-utilisation (i.e. mauvaise utilisation) de la mémoire ; dans le sens où les programmes occupent (en général) partiellement l'espace usager. Par ailleurs, la taille des programmes usagers est limitée par la taille de l'espace usager.

IV.5.1. Problématique

En monoprogrammation, si le process actif, le seul en MC, fait une opération d'E/S Bloquante ➔ l'UC est oisive pendant toute la durée de l'E/S même si le système d'E/S est doté de DMA (Direct Memory Access) !

Si l'on admet l'exécution mono-process, on ne peut pas admettre que l'UC soit oisive pendant une longue durée. Il faut trouver un moyen qui permet de remplir ce temps d'oisiveté.

IV.5.2. Technique de Swapping (ou méthode de va et vient)

Dans le mono-programmé, le processus qui occupe un espace mémoire, il le gardera jusqu'à sa terminaison. Pour pallier à ce problème, une libération temporaire de l'espace occupé pour donner l'occasion aux processus en attente dans le disque d'occuper la mémoire centrale.

Le principe de Swapping se base sur l'état d'un processus, si le processus P1 se bloque suite à une E/S, un autre processus P2 est chargé pour s'exécuter (voir figure 4.6).

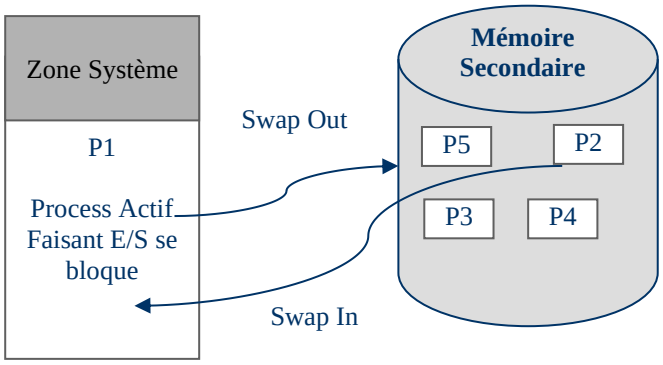


Figure 4.6 : Technique de Swapping

IV.5.3. Technique de recouvrement

La taille d'un programme peut dépasser la taille de la MC. Pour surpasser cette limitation dans un système mono-programmé, le programmeur doit diviser son programme au moment de la conception en un ensemble de modules et les charger dynamiquement à l'exécution en MC de telle sorte qu'il garde que les modules dont il a besoin effectivement. Chaque nouveau module chargé prend la place du module qui doit être déchargé.

IV.6. Gestion de la Mémoire centrale dans les systèmes multiprogrammés

La multiprogrammation permet l'exécution de plusieurs processus à la fois. Elle permet d'optimiser le taux d'utilisation du processeur en réduisant notamment les attentes sur des entrées-sorties. Cette technique nécessite la présence de plusieurs processus en mémoire. La mémoire est donc partagée entre le système d'exploitation et plusieurs processus. Il se pose cependant le problème suivant : Comment organiser la mémoire de manière à faire cohabiter efficacement plusieurs processus tout en assurant la protection des processus ? Deux approches principales sont alors à distinguer :

1. Approche d'allocation contiguë est essentiellement une méthode dans laquelle une seule section/partie de mémoire contiguë est allouée à un processus ou à un fichier qui en a besoin. De ce fait, tout l'espace mémoire disponible réside au même endroit ensemble, ce qui signifie que les partitions de mémoire disponibles librement/inutilisées ne sont pas réparties de manière aléatoire ici et là sur tout l'espace mémoire.

   La mémoire principale est une combinaison de deux parties principales, une pour le système d'exploitation et l'autre pour le programme utilisateur.

   Un programme→ ensemble de mots mémoires contiguës inséparables (insécables).

   On trouve dans cette approche principalement deux types de technique :

   - Technique d'allocation par partitions fixes.
   - Technique d'allocation par partitions variables.

2. Approche d'allocation non contiguë est fondamentalement une méthode contrairement à la méthode d'allocation contiguë, alloue l'espace mémoire présent à différents emplacements au processus selon ses besoins. Comme tout l'espace mémoire disponible est dans un modèle distribué, l'espace mémoire librement disponible est également dispersé ici et là.

   Un programme → ensemble de mots mémoires non contiguës séparables(sécables).

Dans cette technique d'allocation de mémoire, un programme peut être divisé en une succession de morceaux contenant chacun un ensemble de mots contiguës. Chaque morceau peut être traité indépendamment des autres. On trouve dans cette approche principalement les technique de **pagination**, **segmentation** et la **segmentation paginée**.

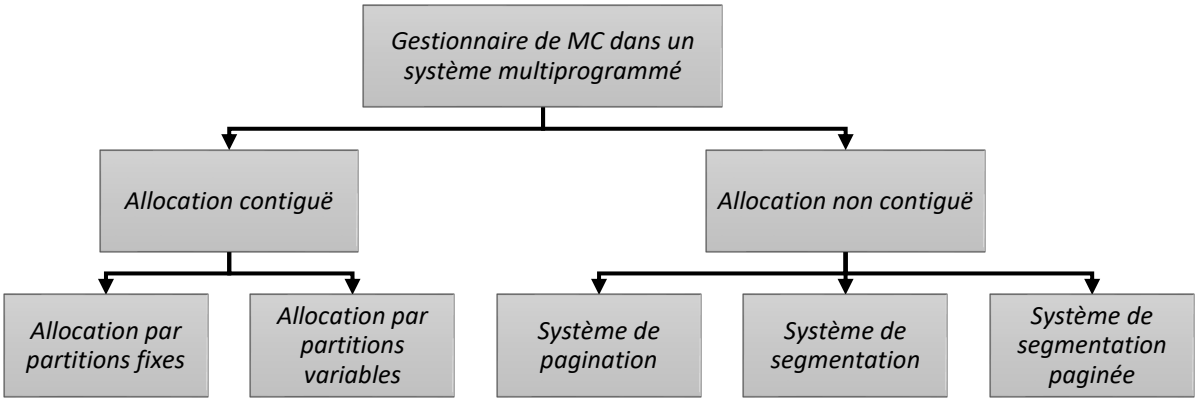


Figure 4.7 : Les approches de gestion de mémoire centrale dans un système multiprogrammé.

IV.6.1. Allocation contiguë (Contiguous Allocation)

Cette stratégie constitue une technique simple pour la mise en œuvre de la multiprogrammation. La mémoire principale est divisée en régions séparées ou partitions mémoires ; chaque partition dispose de son espace d'adressage. Le partitionnement de la mémoire peut être statique (fixe) ou dynamique (variable).

Chaque processus est chargé entièrement en mémoire. L'exécutable contient des adresses relatives et les adresses réelles sont déterminées au moment du chargement.

IV.6.1.1. Allocation contiguë par partitions fixes

Cette solution consiste à diviser la mémoire en partitions fixes, de tailles pas nécessairement égales, à l'initialisation du système. D'autre part, le nombre de ces partition set préfixé (par le système ou, éventuellement, par le constructeur) et leurs tailles sont différentes mais aussi préfixées.

L'un des problèmes de conception de tel système est la détermination du nombre et des tailles adéquats.

➔ Les partitions sont gérées par le système en utilisant une table de partions.

Le système d'exploitation maintient une table de description des partitions (PDT, Partition Description Table) indiquant les parties de mémoire disponibles et celles qui sont occupées.

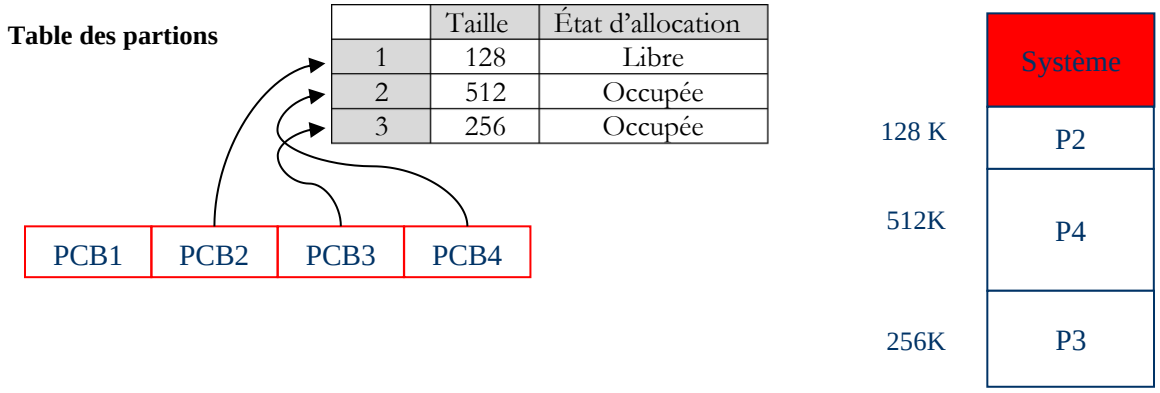


Figure 4.8 : Schéma d'allocation contiguë à partitions multiples fixes.

Les programmes n'ayant pu se loger en mémoire sont placés dans une file d'attente (une file unique ou une file par partition). Dans le cas d'une file d'attente par partition : chaque nouveau processus est placé dans la file d'attente de la plus petite partition qui peut le contenir (voir figure 4.9). Cette façon de faire peut conduire à faire attendre un processus dans une file, alors qu'une autre partition pouvant le contenir est libre.

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 (voir figure 4.10). Dans cette approche, il existe différentes stratégies d'attribution d'une partition libre à un processus en attente :

- Dès qu'une partition se libère, on lui affecte le premier processus de la file qui peut y tenir. L'inconvénient est qu'on peut ainsi affecter une partition de grande taille à un petit processus et perdre beaucoup de place.
- Dès qu'une partition se libère, on lui affecte le plus grand processus de la file qui peut y tenir. L'inconvénient est qu'on pénalise les processus de petite taille.

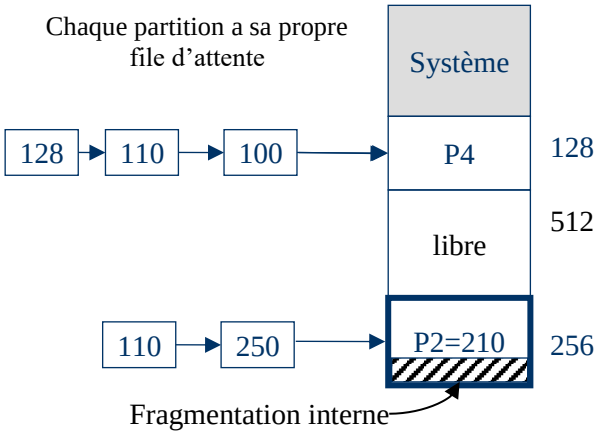


Figure 4.9 : Partitions fixes avec files multiples.

Inconvénients

- Fragmentation interne : Lorsqu'un process n'occupe pas tout l'espace de la partition. L'espace restant est inutilisable.
- Problème de gestion des files d'attente (surcharge, gaspillage de la mémoire…etc.).
- Le degré de la multiprogrammation est limité par le nombre de partitions.

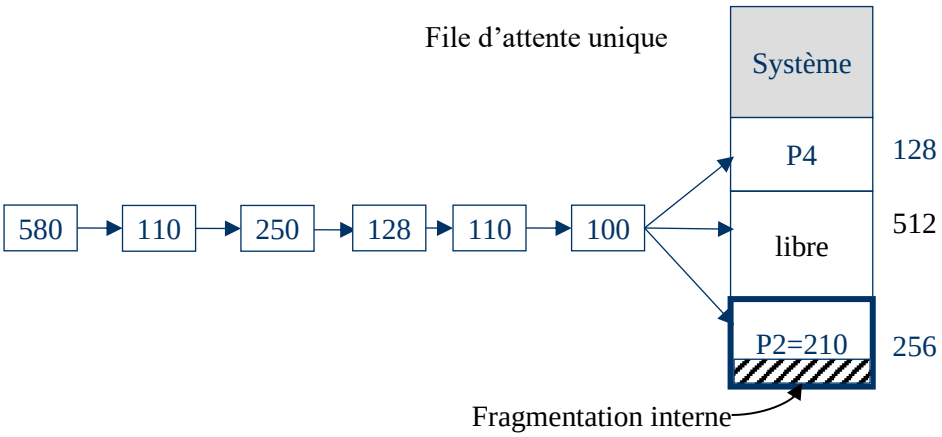


Figure 4.10 : Partitions fixes avec une seule file.

IV.6.1.2. Allocation contiguë par partitions variables

Le gaspillage de mémoire et la fragmentation des systèmes à partitions fixes conduisirent à la conception de partitions variables. Dans ce cas, la mémoire est découpée dynamiquement (partitions variables), suivant la demande des processus et au moment de chargement. Ainsi, à chaque programme est allouée une partition exactement égale à sa taille. Quand un programme termine son exécution, sa partition est récupérée par le système pour être allouée à un autre programme complètement ou partiellement selon la demande. 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 ; par exemple, il faut maintenir une liste des espaces mémoires disponibles.

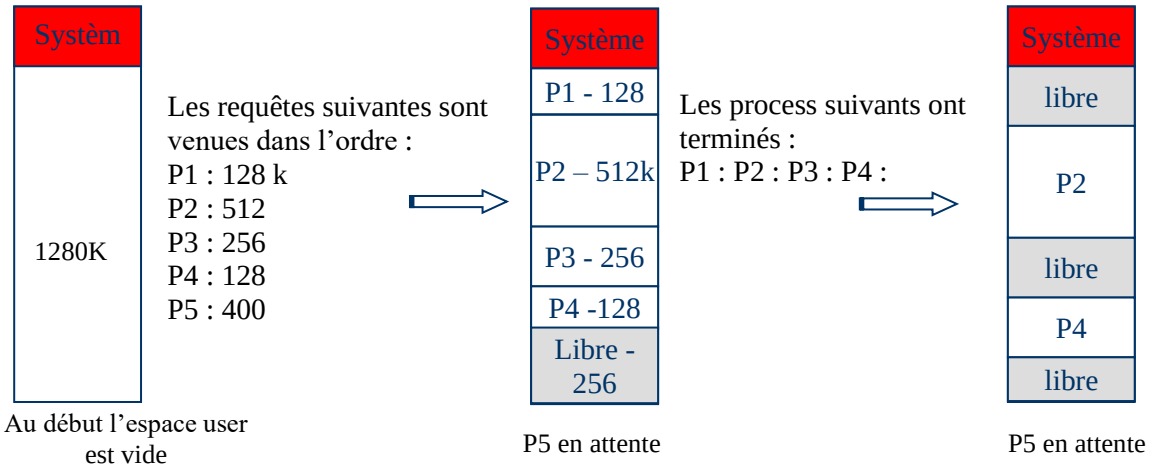


Figure 4.11 : Exemple d'allocation des partitions variables et fragmentation externe.

Inconvénients

- Fragmentation Externe : Lorsqu'un process ne trouve pas une partition de taille convenable alors que la somme des partitions libres le suffit

A. Gestion de l'espace par table de bits (bitmaps)

On divise la MC en un ensemble de blocs d'allocation de quelques octets à quelques Ko. Pour gérer ces blocs, on prévoit pour chacun un bit indiquant sa disponibilité :

- La valeur 1 si le bloc d'allocation (l'unité mémoire) est occupé ;
- La valeur 0 si le bloc d'allocation (l'unité mémoire) est libre.

L'ensemble de ces bits constitue ce qu'on appelle table de bits ou mapping table. Cette table est stockée en MC dans la zone système (protégée). En augmentant la taille de l'unité d'allocation, on réduit la taille de la table de bits mais on perd beaucoup de place mémoire (fragmentation interne).

Inconvénient

- Lorsqu'on doit ramener un processus de k unités, le gestionnaire de la mémoire doit alors parcourir la table de bits à la recherche de k zéros consécutifs. Cette technique est rarement utilisée car la méthode de recherche est lente

B. Gestion de l'espace par Liste Linéaire Chaînée

L'espace mémoire est géré par une liste linéaire chaînée des segments libres et occupés. Un segment est un ensemble d'unités d'allocations consécutives. Un élément de la liste est composé de quatre champs qui indiquent :

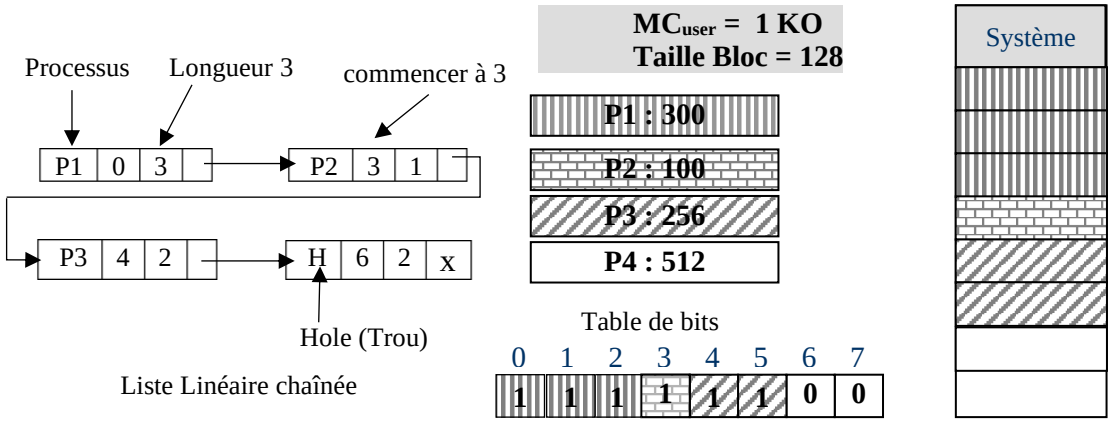


Figure 4.12 : État de MC représenté par bitmap et liste linéaire chaînée.

- L'état libre ou occupé du segment (processus).
- L'adresse de début du segment.
- La longueur du segment.
- Le pointeur sur l'élément suivant de la liste.

La figure 4.12 montre l'exemple d'un tel chaînage, les segments occupés par un processus sont marqués (P), les libres sont marqués (H).

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

C. Stratégies de placement et d'allocation

Dans l'allocation à partitions variables, il existe à chaque instant un ensemble de partitions libres de différentes tailles avec des processus demandeurs d'espace mémoire. Par conséquent, il y a plusieurs algorithmes afin de déterminer l'emplacement d'un programme en mémoire. Le but de tous ces algorithmes est de maximiser l'espace mémoire occupée, autrement dit, diminuer la probabilité de situations où un processus ne peut pas être servi, même s'il y a assez de mémoire. Trois principales stratégies d'allocation existent :

1. First Fit : Parcourir la liste des zones libres jusqu'à la rencontre de la première zone supérieur ou égale à la demande.

Dans l'exemple ci-dessous (voir figure 4.13), la première zone qui convient pour P4 est la zone contenant 140 O. elle lui sera allouée avec un résidu de 50 O.

- Recherche rapide ⇔ Allocation rapide
- La Zone choisie n'est pas toujours la bonne ➔ Possibilité de bloquer d'autres process dont la Zone choisie pouvait les satisfaire. Exemple : si P5 vient demandant 130 ??

✓ Cette stratégie tend à ordonner la liste selon les adresses croissantes des zones libres.

2. Best Fit : Choisir une zone dont la taille est la plus proche de la demande, ce qui implique un parcours complet de la liste.

Dans l'exemple ci-dessous (voir figure 4.13), la meilleure zone qui convient pour P4 est la zone contenant 100. Elle lui sera allouée avec un résidu de 10.

- La Zone choisie est la meilleure
- Fragmentation Externe de petites tailles qu'elles peuvent être au point où ces petits espaces seront rarement réutilisables.

- ➢ Recherche lente ⇔ Allocation lente

✓ On a intérêt à Ordonner la liste par ordre croissant des tailles pour éviter le parcours complet de la liste et accélérer ainsi l'allocation. Mais, la libération (l'insertion) alors prendra beaucoup plus de temps.

3. Worst Fit : Choisir une zone dont la taille est la plus grande parmi toutes les tailles possibles. Ce qui implique un parcours complet de la liste.

Dans l'exemple ci-dessous (voir figure 4.13), la meilleure zone qui convient pour P4 est la zone contenant 220. Elle lui sera allouée avec un résidu de 130.

- ➢ Fragmentation Externe la plus grande possible ➔ possibilité de réutilisation par un autre process.
- ➢ Recherche lente ⇔ Allocation lente

✓ On a intérêt à Ordonner la liste par ordre décroissant des tailles pour éviter le parcours complet de la liste et accélérer ainsi l'allocation. Mais, la libération (l'insertion) alors prendra beaucoup plus de temps.

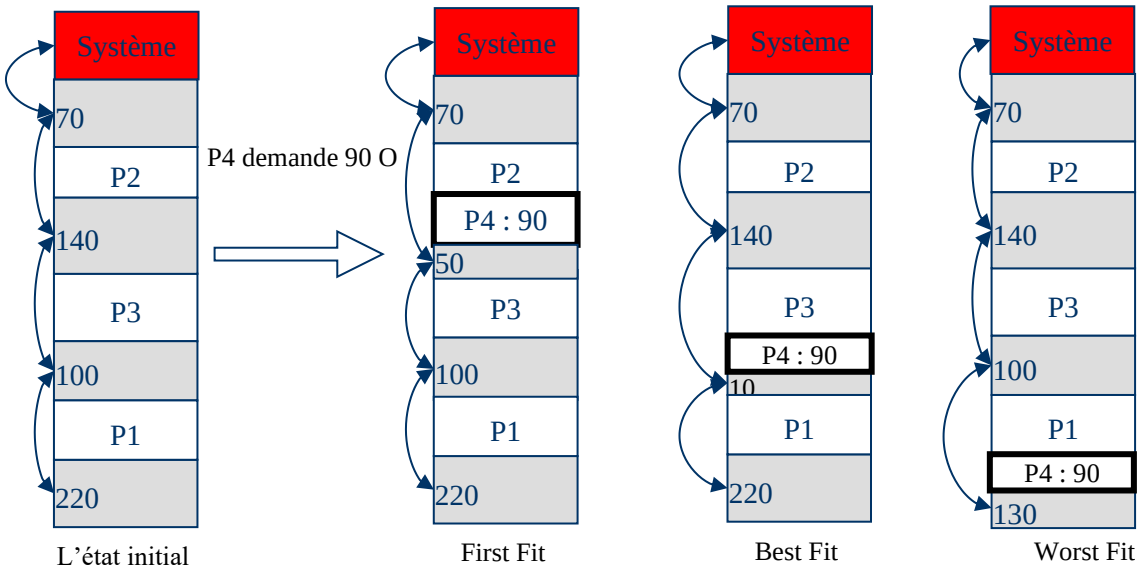


Figure 4.13 : Exemple illustrant les algorithmes de placement.

D. Gestion de la mémoire par subdivision (Buddy system)

C'est un compromis entre partitions de tailles fixes et partitions de tailles variables. La mémoire est allouée en unités qui sont des puissances de 2. Initialement, il existe une seule unité comprenant toute la mémoire. Lorsque de la mémoire doit être attribuée à un processus, ce dernier reçoit une unité de mémoire dont la taille est la plus petite puissance de 2 supérieure à la taille du processus. S'il n'existe aucune unité de cette taille, la plus petite unité disponible supérieure au processus est divisée en deux unités "siamoises" de la moitié de la taille de l'original. La division se poursuit jusqu'à l'obtention de la taille appropriée. De même deux unités siamoises libres sont combinées pour obtenir une unité plus grande.

- ➢ Le grand défaut de cette méthode est la fragmentation interne : pour stocker un fichier de 515 ko, on lui alloue une zone de 1024 ko, soit une perte de 509 ko.

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 :

1M

Un processus A demandé 70 Ko : la mémoire est fragmentée en deux segments de 512 Ko ; l'un d'eux est fragmenté en deux blocs de 256 Ko ; l'un de ces deux derniers est fragmenté en deux blocs de 128 Ko et on loge A dans l'un d'eux, puisque 64 < 70 < 128 :

A	128	256	512

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 :

A	B	64	256	512

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	B	64	C	128	512

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 :

D	64	B	64	C	128	512

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

D	64	128	C	128	512

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

256	C	128	512

- Allocation et libération très simple
- Fragmentation Interne. Un process demandant 2^n +1 (Ex. 257) aura un bloc de 2^{n+1} (ex. 512)

E. Fragmentation mémoire

Les partitions multiples entraînent une fragmentation de la mémoire. Il y aurait suffisamment de mémoire libre pour charger un processus, mais aucune partition n'est de taille suffisante. La fragmentation peut être de deux types :

- **Fragmentation interne** (Internal Fragmentation) La mémoire allouée peut être légèrement plus grande que la mémoire requise. Cette différence est appelée fragmentation interne – de la mémoire qui est interne à une partition mais n'est pas utilisée.
- **Fragmentation externe** (External Fragmentation) La fragmentation externe se présente quand il existe un espace mémoire total suffisant pour satisfaire une requête, mais il n'est pas contigu ; la mémoire est fragmentée en un grand nombre de petits trous (i.e. blocs libres) où un programme ne peut être chargé dans aucun de ces trous.

F. Compactage (Compaction)

Au fur et à mesure de l'allocation et la libération, le nombre de zones libres constituant la fragmentation externes augmente. Il est logique que plus que ce nombre est grand, moins que l'allocation ait une chance de trouver une zone adéquate. ➔ Les performances diminuent !

Ainsi dans l'exemple ci- dessous, le nombre de zones libres est de 4 zones. Aucune de ces zones ne satisfait l'un des trois process en attente, mais leur somme les satisfait les trois.

Ainsi, on a intérêt à ramasser l'espace libre. Ceci est garanti par le déplacement des programmes chargés de telle sorte ne constituer qu'une seule zone libre. Cette opération est appelée le compactage de la mémoire centrale (Ramasse miettes ou Garbage Collector).

Le compactage est une solution pour la fragmentation externe qui permet de regrouper les espaces inutilisés (i.e. les blocs libres) dans une partie de la mémoire (Ex. début, milieu). Cette opération est très coûteuse en temps CPU.

D'autre part, l'opération de compactage est effectuée quand un programme qui demande d'être exécuté ne trouve pas une partition assez grande, mais sa taille est plus petite que la fragmentation externe existante.

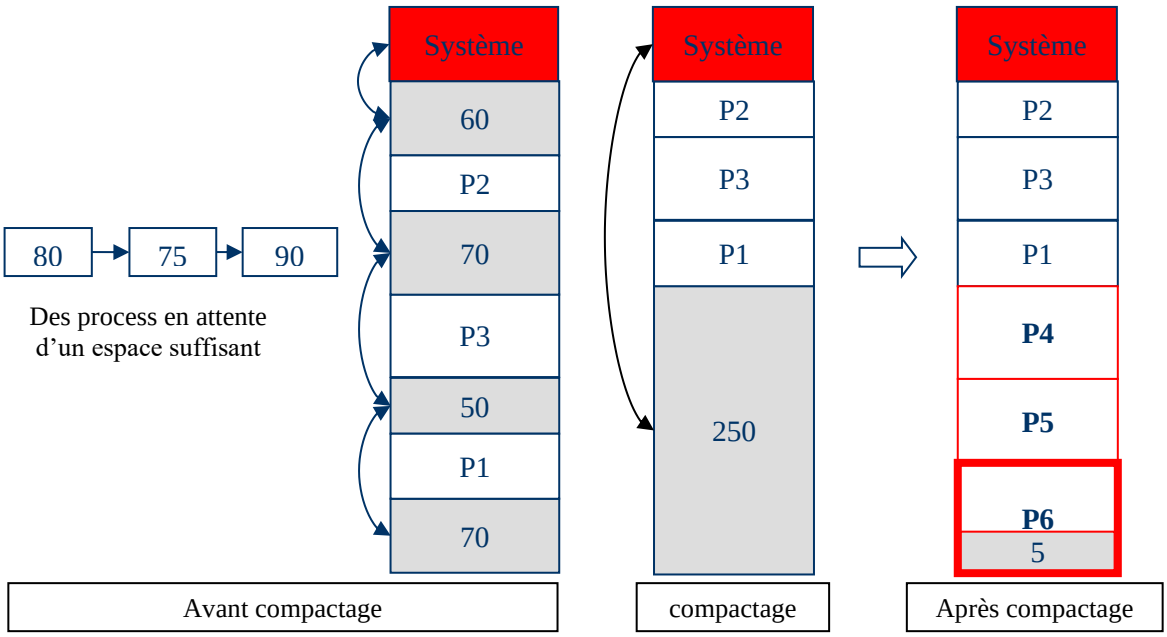


Figure 4.14 : Compactage.

IV.6.2. Allocation non contiguë (Non Contiguous Allocation)

Les techniques d'allocation précédentes, de l'approche d'allocation contiguë, présentent plusieurs limitations :

1. Au moment de l'allocation, le gestionnaire doit trouver un espace libre contigu suffisant pour loger le programme. Même s'il y a des fragments libres dont la somme est suffisante, l'allocation contiguë refuse de le loger. Plus que la taille d'un programme augmente, moins que les chances de trouver un tel espace
2. La fragmentation externe fréquente due aux allocations et désallocations de l'espace. Le compactage en était la solution, mais ce n'est plus la meilleure !
3. Les programmes dont la taille dépasse la mémoire centrale sont impossible à être exécutés

Pour remédier à ces problèmes, il faut que le gestionnaire de mémoire change sa vue vis-à-vis la nature insécable d'un programme. Les fragments d'espace libre éparpillés dans la MC peuvent être exploités si l'on considère que le programme est une entité sécable en plusieurs morceaux insécables dont chacun peut occuper une zone libre qui le suffit indépendamment des autres.

L'objectif principal de l'allocation non contiguë est de pouvoir charger un processus en exploitant au mieux l'ensemble des trous mémoire.

Un programme est divisé en morceaux dans le but de permettre l`allocation séparée de chaque morceau. Les morceaux sont beaucoup plus petits que le programme entier et donc permettent une utilisation plus efficace de la mémoire. Par conséquent, les petits trous peuvent être utilisés plus facilement. En fonction des tailles de ces morceaux, on distingue deux stratégies :

- Si les morceaux du programme sont de tailles fixes et égales, on parle de système de pagination.
- Si les morceaux du programme sont de tailles variables, on parle alors de système de segmentation.

Ces deux techniques peuvent être combinées, on parle dans ce cas de la technique de segmentation paginée.

IV.6.2.1. Système de Pagination de la MC

1. Principe de la pagination

Dans le mécanisme de pagination, l'espace d'adressage des programmes est divisé en un ensemble de morceaux de même taille appelés PAGES. Cet espace est appelé l'espace logique du programme. A son tour, l'espace de la mémoire physique (MC) est lui-même découpé en un ensemble de morceaux de taille fixe appelés CASES ou cadres de page (Frame Page) (voir figure 4.15).

La taille d'une case est égale à la taille d'une page. Cette taille est définie par le matériel et le SE cible. Généralement, c'est une puissance de 2, comme étant 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 particulièrement la traduction d'une adresse logique en un numéro de page et un déplacement dans la page.

Ainsi, un programme de taille **L** sera divisé en un nombre de '**p**' pages tel que '**p**' est le premier entier vérifiant la relation : **L >= n * taille_page**.

- Au moment du chargement, chaque page d'un programme sera chargée dans une case libre quelconque de la mémoire centrale. L'ensemble des cases hébergeant les différentes pages ne sont pas obligatoirement contiguës.
- Dans un système de pagination pure, un programme ne peut être chargé que s'il y a un nombre de cases libres égal au nombre de pages du programme.

Si la taille L d'un programme n'est pas un multiple de la taille d'une page, alors la dernière page dans l'espace logique du programme ne sera pas entièrement remplie. Un espace vide irrécupérable apparaît constituant une fragmentation interne.

2. Schéma de translation d'adresses

L'association d'une page logique avec une page physique est décrite dans une table appelée table de pages (Page Table). La translation des adresses logiques en adresses physiques est à la charge de la MMU. Le support matériel pour la pagination est montré dans la figure 4.16 :

3. Correspondance Adresses logiques - Adresses physiques

On divise chaque adresse (logique) générée par l'UC en deux parties :

- Un numéro de page (P, Page Number) qui est utilisé comme un index dans la table des pages. La table des pages contient l'adresse de base de chaque page dans la mémoire physique.

- Un déplacement dans la page (D, Page Offset) qui est combinée à l'adresse de base de la page pour définir l'adresse mémoire physique qui sera envoyée à l'unité mémoire.

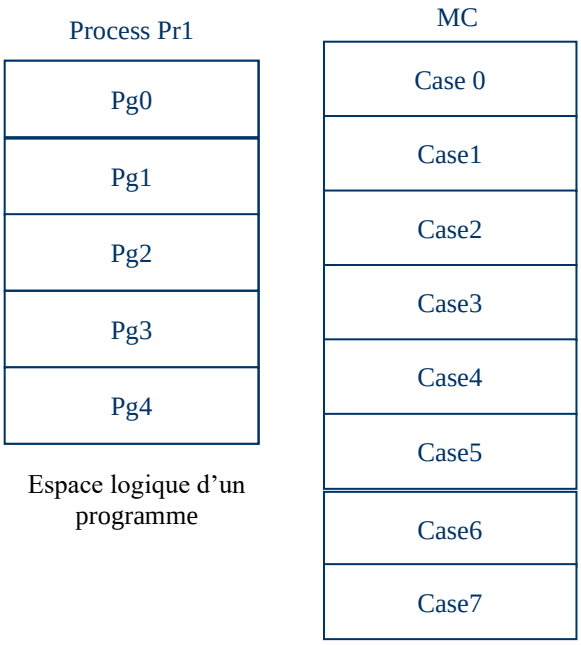


Figure 4.15 : Espace logique vs Espace physique

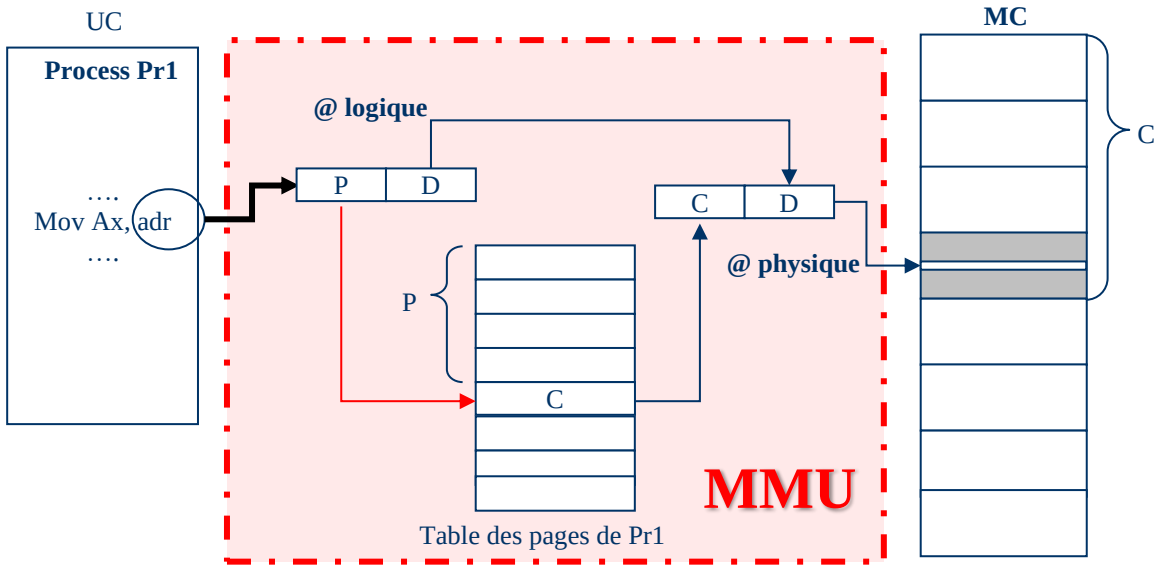


Figure 4.16 : Matériel pour la pagination

Pour un espace d'adressage logique de 2^m octets, en considérant des pages de 2^n octets, les m - n premiers bits d'une adresse logique correspondent au numéro de page P et les n bits restants au déplacement D dans la page.

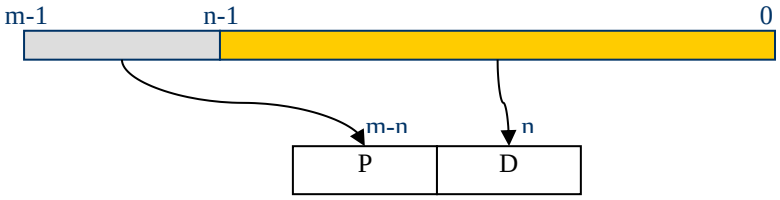


Figure 4.17 : L'adresse paginée

Donc, si T est la taille d'une page et U une adresse logique, alors l'adresse paginée (P, D) est déduite à partir des formules suivantes :

- P = U Div T (où, Div est la division entière)
- D = U Mod T (où, Mod est le reste de la division)

4. Obtenir l'adresse Physique ?

L'adresse physique correspondante à une adresse logique adr = <P, D> est obtenue en remplaçant le numéro de page P par le numéro de la case C, ou en pratique son adresse d'implantation, contenant cette page. Le déplacement D dans la page étant le même dans la case puisque les deux ont la même taille. Le numéro de la case ou son adresse étant obtenu en indexant la table des pages du process actif par la valeur P. la case correspondante nous renvoie C, ou l'adresse équivalente.

- Au moment de l'exécution, et pour un processeur donné, une seule table des pages qui est active ; c'est celle correspondant au process en cours d'exécution. Chaque opération de commutation de contextes implique le changement de la table des pages active au niveau processeur.

Exemple : Une adresse AB = 1010 1011 sur 8 bits, m=8 bits. Nous avons aussi des pages de taille 2^6 mots. Donc le nombre de pages est de $2^{m-n}=2^{8-6}=4$.

Le déplacement à l'intérieur d'une page est de n = 6 bits.

Alors l'adresse logique devienne adresse paginée <P, D> comme suit :

- P= 171 Div 64 = 2, P=2. ➔ Le numéro de la page est 2
- D =171 Mod 64 = 43, ➔ D=43 est le déplacement (offset) dans la page numéro 2.

1010 1011 → <2 , 43>

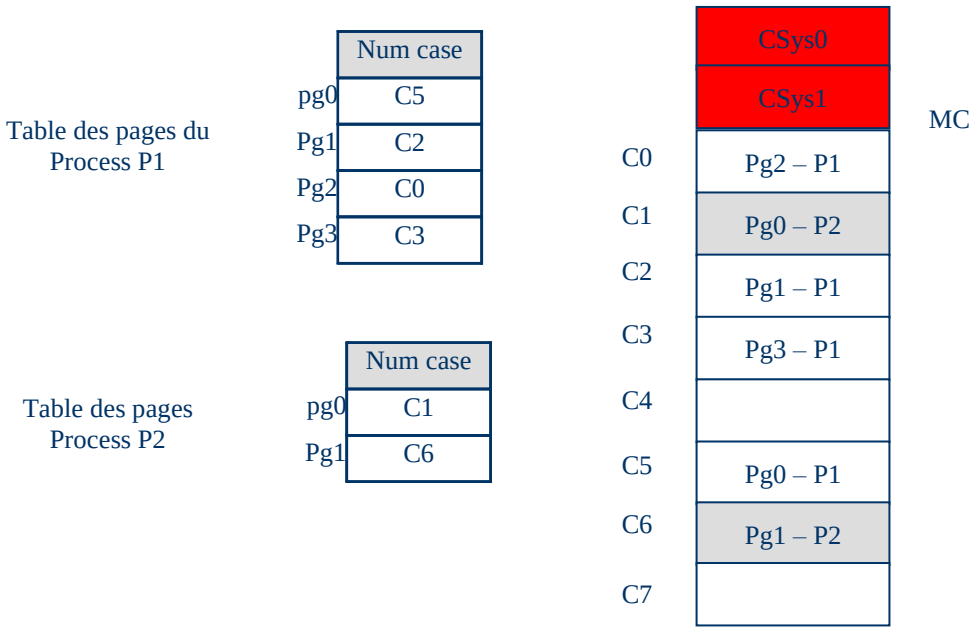


Figure 3.18 : Modèle de pagination de la mémoire logique et physique

IV.6.2.2. Système de Segmentation de la MC

1. Principe de la segmentation

Le découpage tel qu'il a été présenté dans un système de pagination ne correspond pas à la vision et à la manière de pensée de l'utilisateur. Ce dernier voit un programme comme étant un ensemble d'unités logiques : code, données, pile, programme principal, procédures, une dll, … appelées généralement Segments.

- La pagination ne projette pas cet aspect de raisonnement.
- La MC n'est pas découpée en bloc comme dans le cas de pagination.

L'espace d'adressage d'un programme est divisé en un ensemble de morceaux n'ayant pas la même taille mai ayant une structure logique correspond à un point de vue de l'utilisateur. Cette vue utilisateur n'est pas reflétée dans une mémoire paginée. Chaque segment d'un programme peut être hébergé dans un espace mémoire indépendant des autres segments. La segmentation est une stratégie de gestion mémoire qui reproduit le découpage mémoire tel qu'il est décrit logiquement par l'usager.

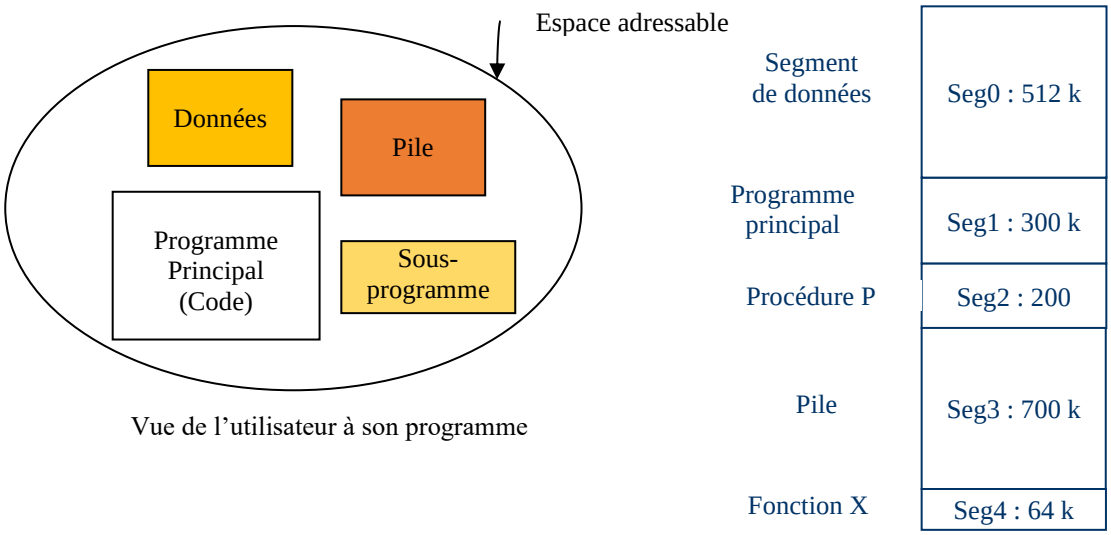


Figure 3.19 : Vue de l'utilisateur d'un programme & Segmentation

Dans une mémoire segmentée, chaque unité logique d'un programme usager est stockée dans un bloc mémoire, appelé « segment » à l'intérieur duquel les adresses sont relatives au début du segment. Ces segments sont de tailles différentes. Un programme sera donc constitué d'un ensemble de segments de code et de données, pouvant être dispersés en MC.

La segmentation facilite l'édition de liens, ainsi que le partage entre processus de segments de données ou de codes.

Contrairement au schéma de pagination, la segmentation permet d'éliminer la fragmentation interne car l'espace mémoire alloué à un segment est de taille égale exactement à la taille du segment. Cependant, une fragmentation externe peut se produire due au fait qu'on fait une allocation dynamique de l'espace mémoire.

2. Schéma de translation d'adresses

Chaque segment est repéré par son numéro S et sa longueur variable L. Un segment est un ensemble d'adresses logiques contiguës. Une adresse logique est donnée par un couple (S, D), où S est le numéro du segment et D le déplacement dans le segment.

L'association d'une adresse logique à une adresse physique est décrite dans une table appelée table de segments (Segment Table).

Chaque entrée de la table de segments possède un segment de base et un segment limite. Le segment de base contient l'adresse physique de début où le segment réside en mémoire, tandis que le registre limite spécifie la longueur du segment.

La translation des adresses logiques en adresses physiques est à la charge de la MMU. Le support matériel pour la segmentation est montré dans la figure 4.20 :

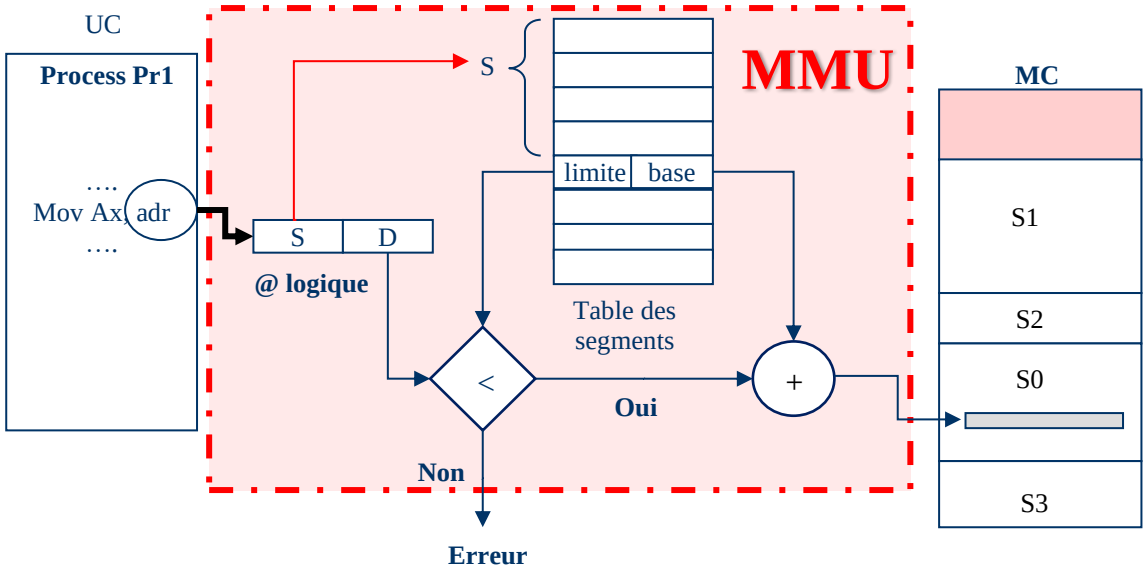


Figure 3.20 : Matériel pour la segmentation

Le déplacement D dans le segment doit être compris entre 0 et la limite du segment (D ∈ [0, L [) et dans ce cas l'adresse physique est calculée en additionnant la valeur de D à la base du segment. Dans le cas contraire, une erreur d'adressage est générée.

3. Correspondance Adresses logiques - Adresses physiques

Dans un système segmenté, une adresse logique référencée par une instruction est de la forme d'un couple <S, D> tel que S : numéro de segment, et D déplacement dans le segment. Le gestionnaire de mémoire doit garder la trace de l'emplacement mémoire dans lequel est hébergé le segment S. Pour ce faire il construit pour chaque process une structure, généralement logicielle, appelée table des segments dans laquelle il fait la correspondance entre chaque segment chargée dans la MC et son adresse d'implantation (dans la MC) ainsi que sa taille.

Exemple :

Sur un système utilisant la segmentation simple, calculez l'adresse physique de chacune des adresses logiques, à partir de la table des segments ci-après. On suppose que les adresses soient décimales au lieu de binaires.

Segment	Base	Limite
0	1100	500
1	2500	1000
2	200	600
3	4000	1200

Adresse logique	Adresse physique
<0, 300>	300<500, donc @=1100+300=1400
<2, 800>	800>600, donc Erreur d'adressage
<1, 600>	600<1000, donc @=2500+600=3100
<3, 1100>	1100<1200, donc @=4000+1100=5100
<1, 1111>	1111>1000, donc Erreur d'adressage

IV.6.2.3. Système de Segmentation paginée

La taille d'un segment peut être importante, d'où un temps de chargement long qui peut en résulter. La segmentation paginée combinant entre la segmentation et la pagination, ce qui peut être une

solution. Dans cette technique, les programmes sont alors divisés en segments et chaque segment en pages. Cette technique a été inventée pour le système Multicast.

A chaque processus, on associe une table de segments et une table de pages. Donc chaque adresse de segment n`est pas une adresse de mémoire, mais une adresse au tableau de pages du segment Une adresse logique (S, D), avec S numéro de segment et D déplacement dans le segment, est transformée en (S, P, d), où **P** est un numéro de page et **d** un déplacement dans la page P. chaque adresse devient alors un triplet (S, P, d).

La traduction d'une adresse logique en adresse physique se fait selon le schéma suivant :

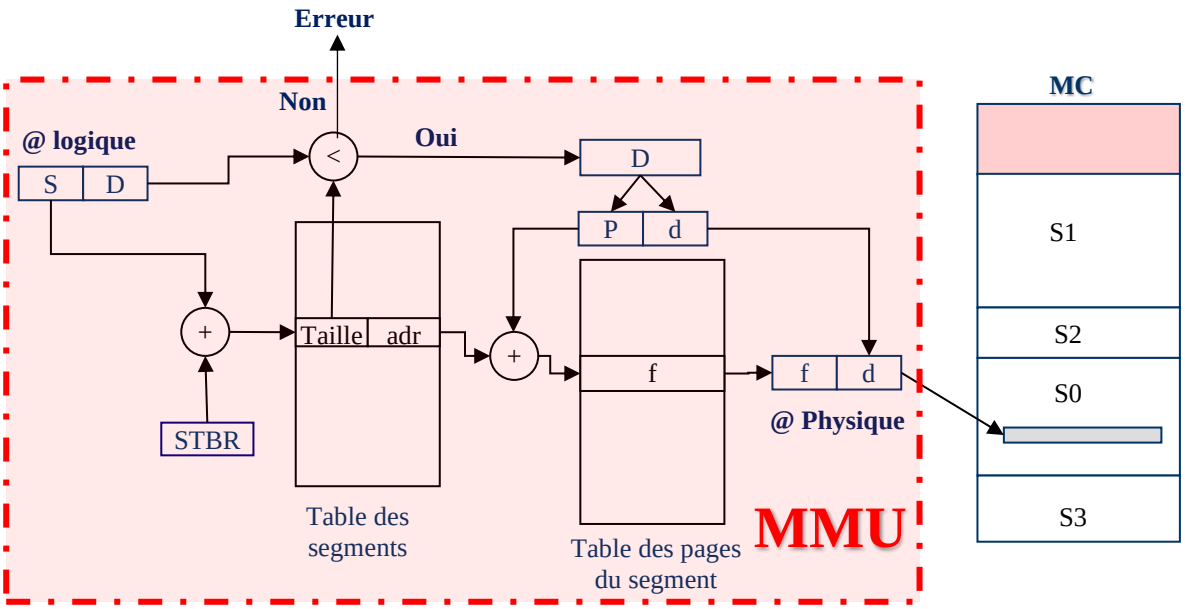


Figure 3.21 : Matériel pour la segmentation paginée

Bien que cette technique élimine la fragmentation externe, elle introduit de nouveau la fragmentation interne.

IV.6.3. Mémoire virtuelle

Les systèmes de pagination et de segmentation tels qu'ils ont été introduits présentent plusieurs limitations :

- Un programme ne peut être chargé que s'il y a des cases libres (zones libres) égales au nombre de pages du programme sinon il sera mis en attente. Une conséquence directe de cette limitation est qu'un programme dont la taille est supérieure à la taille de la MC ne pourra jamais s'exécuter !!!!
- Pour un process actif, une seule page (ou segments) est active à un moment donné ; c'est celle depuis laquelle il lit les instructions ou à laquelle il accède pour besoin de données. Mais ce process a préalablement chargé toutes ses pages (ou segments) dans la MC alors qu'il y a des programmes qui attendent d'être chargées.

La mémoire virtuelle est une technique autorisant l'exécution de processus pouvant ne pas être complètement en mémoire. Elle se base sur la technique de virtualisation de la MC (à l'instar d'une UC virtuelle, imprimante virtuelle, …). Le principe est de laisser les process voir qu'ils ont une mémoire de taille théoriquement infinie. Une partie de code du programme est chargée dans la MC,

l'autre partie se situe dans une mémoire extension de la MC. D'autre part, la 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.

Pour ce faire, les programmes au niveau de la Mémoire secondaire (MS) sont divisés en un certain nombre de morceaux dont chacun est de taille inférieure à la taille physique de la MC. A l'exécution on charge quelques morceaux ; dès qu'un autre est nécessaire on le charge dans la MC tout en faisant, éventuellement si l'espace mémoire ne suffit pas, sortir un autre. Ainsi, le SE considère la MS comme étant une extension logique de la MC. Dans la pratique, une partie de la MS uniquement, appelée zone de swap ou MV, est étendue à la MC.

Ainsi, un process demandeur, sera chargé entièrement dans la nouvelle mémoire (MC + Swap). Uniquement une partie de son code est chargé dans la MC.

- On insiste sur le fait que la MC pour les process utilisateur est celle formée de la MC réelle + la zone swapp. Les échanges entre les deux sont transparents par rapport à l'utilisateur.

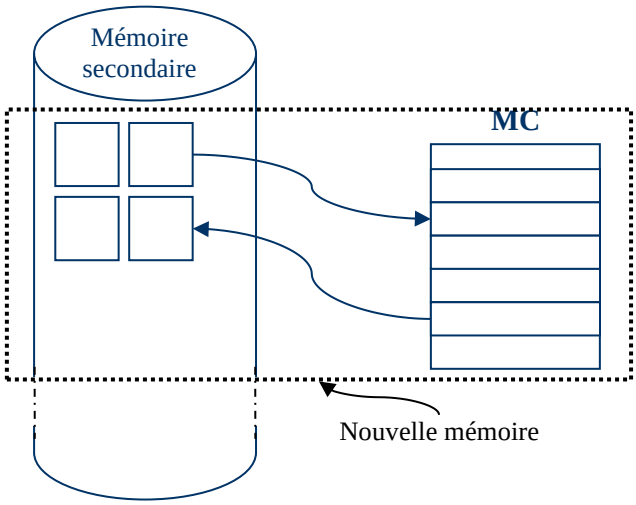


Figure 3.22 : Mémoire virtuelle.

IV.6.3.1. Recouvrement (Overlay)

Le recouvrement (Overlay) est une technique qui permet de remplacer une partie de la MC par une autre. À un instant donné, un programme n'utilise qu'une petite partie du code et des données qui le constituent. Les parties qui ne sont pas utiles en même temps peuvent donc se "recouvrir" ; c'est-à-dire, occuper le même emplacement en mémoire physique. Cette technique consiste à permettre à l'utilisateur de diviser son programme en segments, appelés segments de recouvrement (Overlays) et à charger un segment en mémoire. Le segment 0 s'exécute en premier. Lorsqu'il se termine, il appelle un autre segment de recouvrement qui sera chargé dans le même emplacement mémoire.

IV.6.3.2. Pagination à la demande

Le principe de la mémoire virtuelle est couramment implémenté avec la pagination à la demande (On-Demand Paging) ; c'est-à-dire que les pages des processus ne sont chargées en MC que lorsque le processeur demande à y accéder. La pagination à la demande est semblable à un système de pagination avec va-et-vient (swapping).

Comment le processeur puisse distinguer entre les pages qui sont en mémoire, et celles qui sont sur disque afin de détecter leur éventuelle absence ?

Avec cette technique, le SE dispose de moyens pour distinguer les pages qui sont en mémoire, et celles qui sont sur disque. Il utilise dans la table des pages un bit supplémentaire V appelé bit de validation. (Valide/invalide) pour décrire si la page est chargée en mémoire ou non.

- V = 1 ➔ La page est chargée dans la MC, l'accès à elle est valide
- V = 0 ➔ La page n'est pas chargée dans la MC, l'accès à elle n'est pas valide

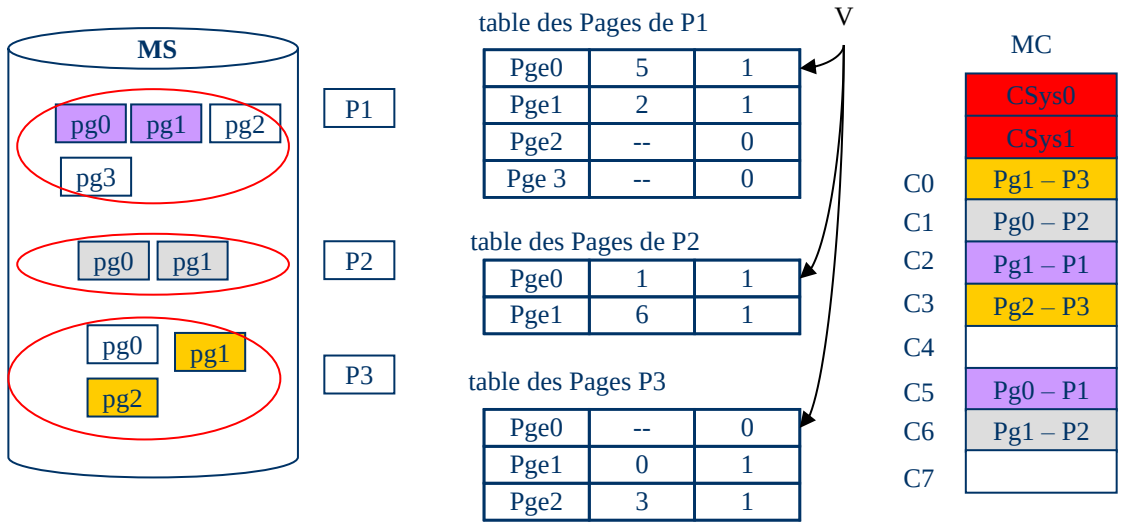


Figure 3.23 : Tables de pages avec pages absentes

Suite à une référence à une adresse 'adr' quelconque, cette dernière est convertie par la MMU en un couple <P, D>. La valeur P permet d'indexer la table de pages, avant de convertir on doit consulter d'abords le bit V associé à elle. Si ce bit est à 1 ceci signifie que la page est chargée en MC alors on peut continuer la conversion. Sinon, un déroutement est généré par la MMU signalant que le process veut accéder à une page qui n'existe pas dans la MC. Ce déroutement est appelé défaut de pages (voir figure 3.24).

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 (Fault Page) vers le système d'exploitation. 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 auxiliaire.
- Trouver un cadre de page libre et charger la page.

IV.6.3.2.1. Stratégies de 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. Déterminer si la référence mémoire est valide. Dans le cas négatif ; c'est-à-dire l'adresse ne se trouve pas dans l'espace d'adressage, il s'agit d'une erreur d'adressage.

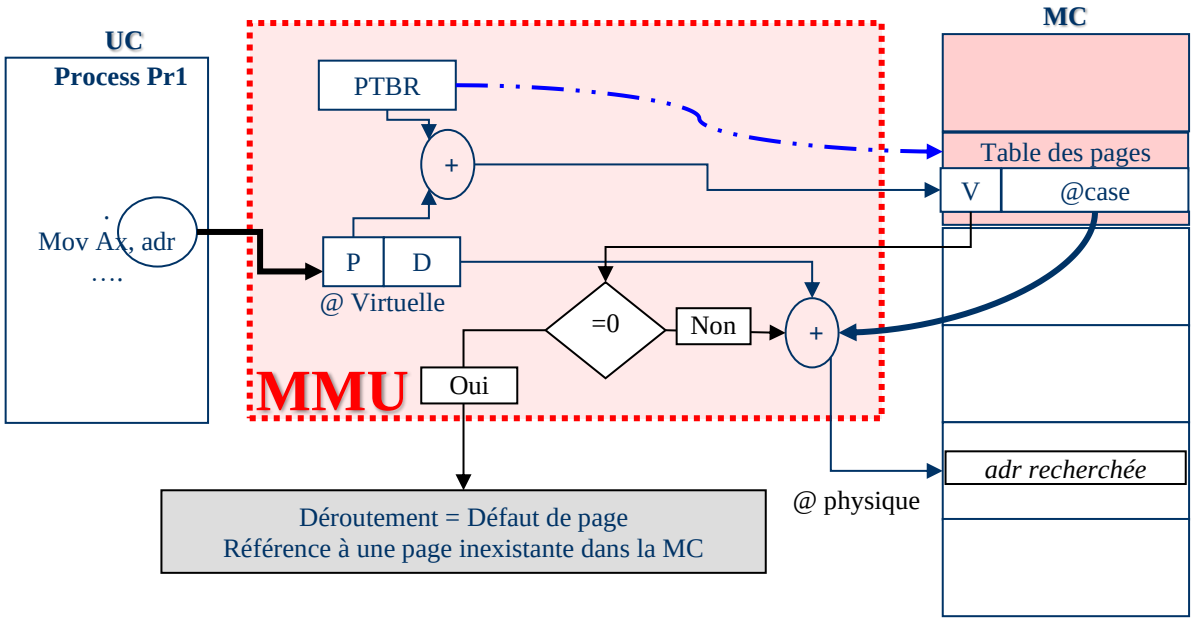


Figure 3.24 : Défaut de page

2. S'il s'agit d'une référence valide mais la page n'est pas encore chargée en mémoire, on doit la charger.
3. Trouver un frame libre pour charger la page manquante, sinon utiliser un algorithme de remplacement de pages pour sélectionner un cadre de page victime.
4. Lancer une opération d'E/S pour lire la page manquante à partir de l'unité de swapping.
5. Mise-à-jour de la table de pages.
6. Redémarrer l'instruction interrompue, le processus peut alors accéder à la page.

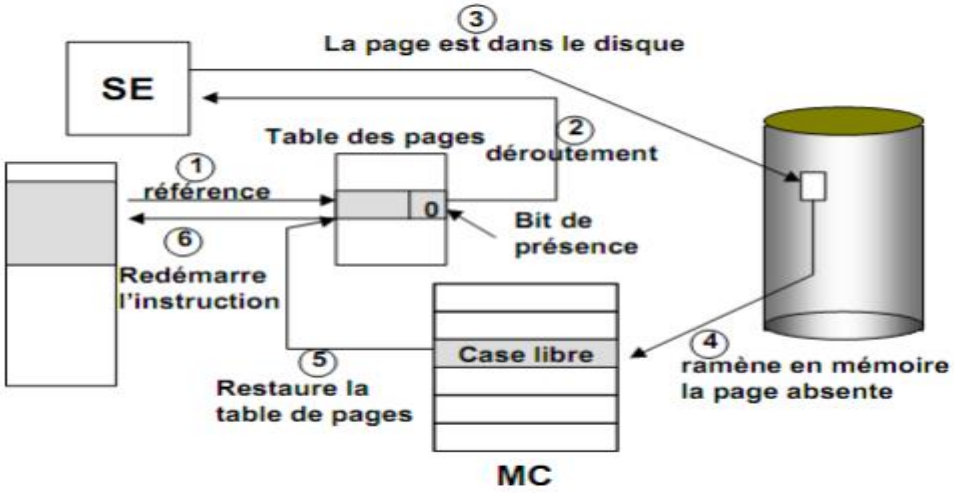


Figure 3.25 : Procédure de traitement d'un défaut de pages.

IV.6.3.2.2. Quelques 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.

A un instant donné, la séquence des numéros de pages référencées par un process depuis son exécution est appelée chaîne de Références du process. Le comportement futur d'un programme peut être contrôlé par la connaissance de cette chaîne de référence.

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.

Exemple : Soit un processus divisé en 5 pages, la taille d'une page = 100.

- Un Programme P fait successivement référence aux adresses suivantes : 100, 210, 355, 120,420, 110, 200, 550, 139, 201, 395, 404, 505.
- La chaîne de Références associée est : 1,2,3,1,4,1,2,5,1,2,3,4,5.

Quel est le nombre de défauts de pages avec différents algorithmes de remplacement ?

- FIFO (First In First Out)
- Optimal
- LRU (Least Recently Used)
- NFU (Not Frequantly Used)
- MFU (Most frequently used)
- Algorithme de la seconde chance

A. Stratégie de remplacement FIFO

Principe : Remplacer l'ancienne Page chargée

Raison : Il est fort probable que l'ancienne page n'est pas fréquemment référencée comme une page récente

Implémentation : Prévoir pour chaque page dans la table des pages un champs (date+temps) initialisé à l'heure de chargement.

Exemple : mémoire de 3 cases

Demandes	1	2	3	1	4	1	2	5	1	2	3	4	5
Case 1	1	1	1	1	4	4	4	5	5	5	5	5	5
Case 2		2	2	2	2	1	1	1	1	1	3	3	3
Case 3			3	3	3	3	2	2	2	2	2	4	4
Défaut de page	D	D	D		D	D	D	D			D	D	

➔ 9 défauts de pages pour une mémoire de 3 cases

➔ On peut se dire que si l'on augmente le nombre de cases le nombre de défaut de pages diminuera !

Nous ajoutons une case. Et on va voir :

Demandes	1	2	3	1	4	1	2	5	1	2	3	4	5
Case 1	1	1	1	1	1	1	1	5	5	5	5	4	4
Case 2		2	2	2	2	2	2	2	1	1	1	1	5
Case 3			3	3	3	3	3	3	3	2	2	2	2
Case 4					4	4	4	4	4	4	3	3	3
Défaut de page	D	D	D		D			D	D	D	D	D	D

➔ Le résultat est l'augmentation du nombre de défauts de pages à 10.

➔ Ce phénomène est appelé **l'anomalie de Belady**.

B. Stratégie de remplacement optimal

Principe : la page victime sera la page qui ne sera pas référencée dans le futur immédiat.

Raison : c'est la case la plus adéquate, car toutes les autres vont être référencée avant elle.

Implémentation : Comment peut-on savoir qu'une page sera référencée dans le futur?

➔ Algorithme non implémentable, il est utilisé pour référence de comparaison des autres algorithmes.

Exemple : mémoire de 4 cases

Demandes	1	2	3	1	4	1	2	5	1	2	3	4	5
Case 1	1	1	1	1	1	1	1	1	1	1	1	4	4
Case 2		2	2	2	2	2	2	2	2	2	2	2	2
Case 3			3	3	3	3	3	3	3	3	3	3	3
Case 4					4	4	4	5	5	5	5	5	5
Défaut de page	D	D	D		D			D				D	

➔ 6 défauts de pages pour une mémoire de 4 cases !!

C. Stratégie de La page la moins récemment utilisée (LRU : Least Recently Used)

Principe : la page victime sera la page la moins référencée dans un passé proche.

Raison : Principe de localité spatiale et localité temporelle favorise qu'une page récemment utilisée à une grande chance d'être référencée dans une future très proche. Donc la page victime devrait être une page anciennement référencée.

Implémentation : un champ pour chaque entrée dans la table des pages indiquant le temps de la dernière référence à la page. A Chaque référence (en écriture ou e lecture) à la page, il sera mis à jour.

➔ La page victime est celle ayant le plus ancien temps de référence.

➔ C'est un algorithme un petit peu lourd à implémenter. Utilisé pratiquement mais avec des versions un petit peu légère.

Exemple : mémoire de 4 cases

Demandes	1	2	3	1	4	1	2	5	1	2	3	4	5
Case 1	1	1	1	1	1	1	1	1	1	1	1	1	5
Case 2		2	2	2	2	2	2	2	2	2	2	2	2
Case 3			3	3	3	3	3	5	5	5	5	4	4
Case 4					4	4	4	4	4	4	3	3	3
Défaut de page	D	D	D		D			D			D	D	D

➔ 8 défauts de pages pour une mémoire de 4 cases !!

D. Stratégie de NFU : not frequently used (Least Frequently Used)

Principe : la page victime est la page qui n'est pas fréquemment utilisée

Raison : c'est la page la moins utilisée dans le passé proche parmi les autres, donc on peut se dire qu'elle a une faible probabilité qu'elle soit référencée dans le future proche.

Implémentation : un champ pour chaque entrée dans la table des pages indiquant le nombre de fois que cette page a été référencée.

➔ La page victime est celle ayant la plus petite valeur.

Exemple : mémoire de 4 cases

Demandes	1	2	3	1	4	1	2	5	1	2	3	4	5
Case 1	1	1	1	1	1	1	1	1	1	1	1	1	1
Case 2		2	2	2	2	2	2	2	2	2	2	2	2
Case 3			3	3	3	3	3	5	5	5	5	4	4
Case 4					4	4	4	4	4	4	3	3	5
Défaut de page	D	D	D		D			D			D	D	D

➔ 8 défauts de pages pour une mémoire de 4 cases !!

E. Stratégie de MFU : Most frequently used

Principe : la page victime est la page qui est fréquemment utilisée

Raison : une page qui n'est pas fréquemment utilisée, possible qu'elle le soit dans le proche future.

Implémentation : un champ pour chaque entrée dans la table des pages indiquant le nombre de fois que cette page a été référencée.

➔ La page victime est celle ayant la plus grande valeur.

Exemple : mémoire de 4 cases :

Demandes	1	2	3	1	4	1	2	5	1	2	3	4	5
Case 1	1	1	1	1	1	1	1	5	5	5	5	4	4
Case 2		2	2	2	2	2	2	2	1	1	1	1	5
Case 3			3	3	3	3	3	3	3	2	2	2	2
Case 4					4	4	4	4	4	4	3	3	3
Défaut de page	D	D	D		D			D	D	D	D	D	D

➔ 10 défauts de pages pour une mémoire de 4 cases !!

F. Stratégie de La Seconde chance

Principe : c'est l'algorithme FIFO avec les modifications suivantes : une page victime choisie, si elle est référencée on lui donne une seconde chance et on cherche une autre.

Raison : améliorer FIFO

Implémentation : de plus celle de FiFO on ajoute un bit Référence qui sera mis à 1 à chaque référence à la page. Une fois la page a été sélectionnée comme victime : si ce bit est à zéro, la page sera la victime sinon, le bit est remis à 0 et on cherche une autre page plus ancienne.

➔ C'est une approximation du LRU

Exemple : mémoire de 4 cases :

Demandes	1	2	3	1	4	1	2	5	1	2	3	4	5
Case 1	1_1^+	1_1^+	1_1^+	1_1^+	1_1^+	1_1^+	1_1^+	5_1	5_1	5_1	5_1^+	4_1	4_1
Case 2		2_1	2_1	2_1	2_1	2_1	2_1	2_0^+	1_1	1_1	1_1	1_0^+	5_1
Case 3			3_1	3_1	3_1	3_1	3_1	3_0	3_0^+	2_1	2_1	2_0	2_0^+
Case 4					4_1	4_1	4_1	4_0	4_0	4_0^+	3_1	3_0	3^0
Défaut de page	D	D	D		D			D	D	D	D	D	D

➔ 10 défauts de pages pour une mémoire de 4 cases !!

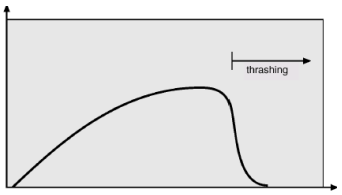
IV.6.4. Notion d'Ecroulement

Un process s'écroule lorsqu'il passe plus de temps à paginer qu'à s'exécuter.

Ceci est dû au fait que le process n'a pas assez de pages dans la MC et que l'allocation était globale.

Solution de l'Ecroulement : notion de working Set

Un working set**, Δ,** du process **P** est un entier définissant le nombre minimum de pages nécessaire que **P** doit avoir dans la MC pour commencer son exécution. Ce nombre doit être maintenu durant toute l'exécution du process. Autrement dit, si l'une des pages de **P** est choisie comme victime, ce choix sera accepté par **P** si le nombre de ses pages chargées dans la MC est strictement supérieur à Δ sinon il sera refusé.



En fait, le **Working set** est un compromis établi entre le nombre de défaut de pages toléré par le système et le nombre de process pouvant être simultanément dans la MC.