Chapitre 10

Mémoire virtuelle

La **mémoire virtuelle** est une technique qui permet d'exécuter des programmes dont la taille excède la taille de la mémoire réelle. L'**espace d'adressage** d'un processus, généré par les compilateurs et les éditeurs de liens, constitue la **mémoire virtuelle** du processus ou **espace d'adressage logique**, comme le montre la figure 10.1. Avec des techniques adéquates, comme on verra dans le présent chapitre, sa taille peut être très supérieure à celle de la mémoire physique ou réelle.

10.1 Introduction

Il serait trop coûteux d'attribuer à tout processus un espace d'adressage complet, surtout parce que beaucoup n'utilisent qu'une petite partie de son espace d'adressage [**?**]. En général, la mémoire virtuelle et la mémoire physique sont structurées en unités d'allocations (pages pour la mémoire virtuelle et cadres pour la mémoire physique). La taille d'une page est égale à celle d'un cadre. Lorsqu'un processus est en cours d'exécution, seule une partie de son espace d'adressage est en mémoire.

Les adresses virtuelles référencées par l'instruction en cours doivent être traduites en adresses physiques. Cette conversion d'adresse est effectuée par des circuits matériels de gestion. Si cette adresse correspond à une adresse en mémoire physique, on transmet sur le bus l'adresse réelle, sinon il se produit un **défaut de page**. Par exemple, si la mémoire physique est de 32 Ko et l'adressage est codé sur 16 bits, l'espace d'adressage logique peut atteindre la taille 2^{16} soit 64 Ko.

L'espace d'adressage est structuré en un ensemble d'unités appelées **pages** ou **segments**, qui peuvent être chargées séparément en mémoire.

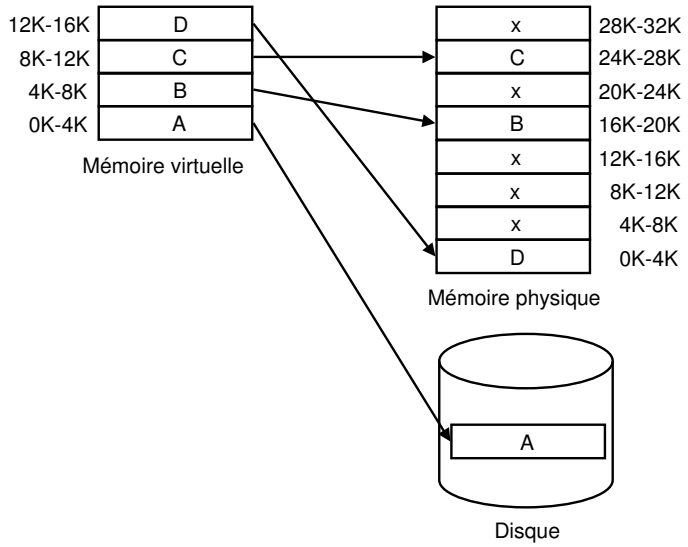

FIG. 10.1 – Espace des adresses virtuelles et espace physique. Le processus dans son espace virtuel continu comprend quatre pages : A, B, C et D. Trois blocs sont situés dans la mémoire physique et un est sur le disque.

L'ensemble de son espace d'adressage (mémoire virtuelle) est stocké sur disque. Pour exécuter un processus, le système d'exploitation charge en mémoire uniquement une ou quelques unités (pages ou segments) y compris celle qui contient le début du programme. Lorsqu'un processus est en cours d'exécution, seule une partie de son espace d'adressage est en mémoire principale. Cette partie est dite résidante. Les parties de cet espace sont chargées en mémoire principale à la demande. Elles peuvent être dispersées en mémoire centrale. Lorsqu'un processus référencie une partie qui ne réside pas en mémoire, l'exécution du processus est suspendue, c'est à dire il passe à l'état bloqué en attente d'une E/S. Le système d'exploitation reprend le contrôle, il modifie l'état du processus (état bloqué), lance une demande d'E/S pour ramener la partie référencée, puis exécute d'autres processus.

Lorsque le transfert en mémoire de la partie référencée est terminé (interruption de fin d'E/S), le système d'exploitation reprend le contrôle et modifie l'état du processus bloqué pour passer à l'état prêt.

10.2 Pagination pure

La mémoire virtuelle et la mémoire physique sont structurées en unités d'allocations appelés **pages** pour la mémoire virtuelle et cases ou **cadres** pour la mémoire physique. La taille d'une page est fixe et égale à celle d'un cadre. Elle varie entre 2 Ko et 16 Ko. 4 Ko est une valeur typique assez répandue. Un schéma de la traduction d'adresses lors de la **pagination pure** est montré sur la figure 10.2 Dans la pagination il n'y a pas de fragmenta-

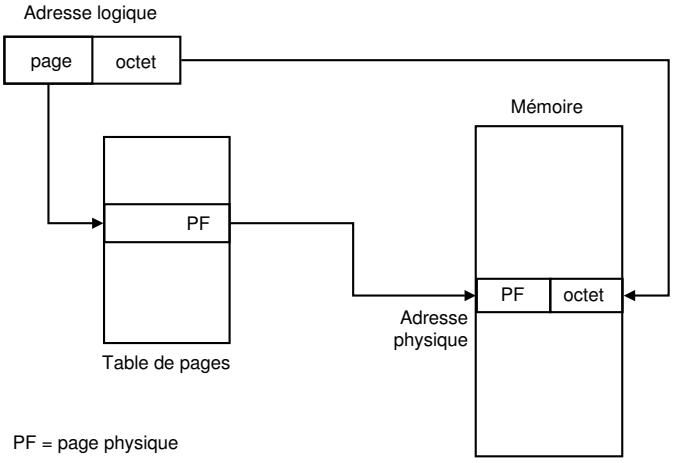

FIG. 10.2 – Pagination pure.

tion externe car toutes les pages sont de même taille. Par contre, il peut y avoir une fragmentation interne si la dernière page de l'espace d'adressage logique n'est pas pleine.

▶ **Exemple 1.** Soit un programme de 64 Ko sur une machine 32 Ko de mémoire. Son espace d'adressage logique est sur le disque. Il est composé de 16 pages de 4 Ko. La mémoire physique est structurée en 8 cadres de 4 Ko, comme montré sur la figure 10.3, prise de [**?**]. Quand le programme tente d'accéder à l'adresse 0 avec une instruction comme `movw $0, %eax`, l'adresse virtuelle 0 est envoyée au MMU (*Memory Management Unit*). Le MMU constate que cette adresse se trouve dans la page 0 (0K - 4K), et la transformation donne la case 2 (8K - 12K) dans la mémoire physique. Cette valeur (8192) est mise dans le bus du système. Ainsi les adresses virtuelles entre 0 et 4095 sont transformées en adresses physiques comprises entre 8192 à 12287.

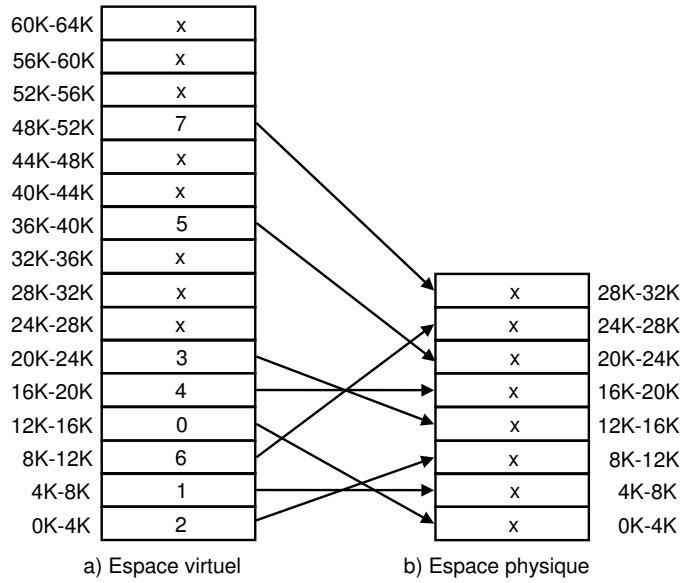

FIG. 10.3 – Espace virtuel et espace physique.

10.2.1 Unité de gestion de la mémoire (MMU)

Les adresses virtuelles, générées par les compilateurs et les éditeurs de liens, sont des couples composés d'un numéro de page et d'un déplacement relatif au début de la page. Les adresses virtuelles référencées par l'instruction en cours d'exécution doivent être converties en adresses physiques. Cette conversion d'adresse est effectuée par le MMU, qui sont des circuits matériels de gestion.

Nous allons considérer la page l'unité de transfert. Le MMU vérifie si l'adresse virtuelle reçue correspond à une adresse en mémoire physique, comme montré à la figure 10.4. Si c'est le cas, le MMU transmet sur le bus de la mémoire l'adresse réelle, sinon il se produit un défaut de page. Un défaut de page provoque un déroutement (ou `TRAP`) dont le rôle est de ramener à partir du disque la page manquante référencée. La correspondance entre les pages et les cases est mémorisée dans une table appelée **Table de pages** (**TP**). Le nombre d'entrées dans cette table est égal au nombre de pages virtuelles. La **Table de pages** d'un processus doit être — en totalité ou en partie — en mémoire centrale lors de l'exécution du processus. Elle est nécessaire pour la conversion des adresses virtuelles en adresses physiques.

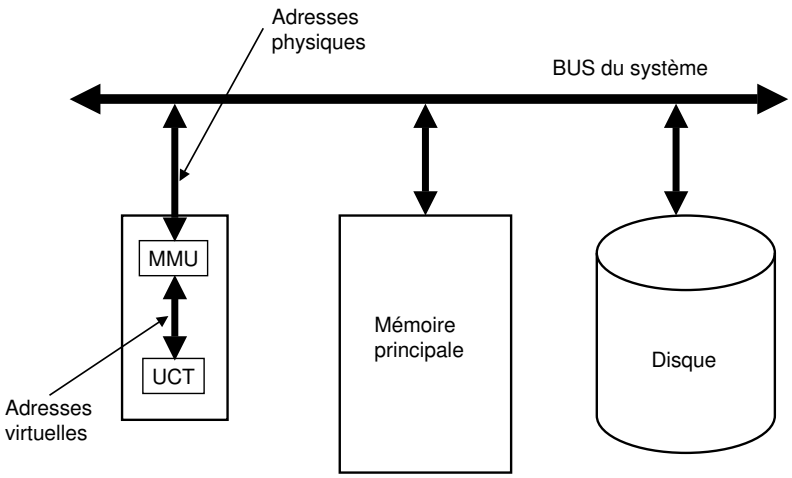

FIG. 10.4 – Unité de gestion de la mémoire *MMU*.

Structure de la Table de pages

Chaque entrée de la **Table de pages** est composée de plusieurs champs, notamment :

1. Le bit de présence.
2. Le bit de référence (R).
3. Les bits de protection.
4. Le bit de modification (M).
5. Le numéro de case correspondant à la page.

La structure d'une table de pages typique, bien que dépendant de la machine, est montrée sur la figure 10.5. La taille varie de machine à machine, mais 32 bits est une taille répandue.

Fonctionnement d'un MMU

Le MMU reçoit, en entrée une adresse virtuelle et envoie en sortie l'adresse physique ou provoque un déroutement. Dans l'exemple de la figure 10.3, l'adresse virtuelle est codée sur 16 bits. Les 4 bits de poids fort indiquent le numéro de page, comprise entre 0 et 15. Les autres bits donnent le déplacement dans la page, entre `0` et `4095`. Le MMU examine l'entrée dans la **Table de pages** correspondant au numéro de page, dans ce cas `2`. Si le bit de présence est à 0, la page n'est pas en mémoire alors le MMU

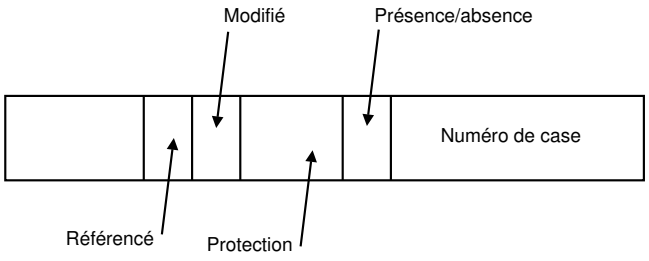

FIG. 10.5 – Structure d'une entrée typique d'une **Table de pages**.

provoque un déroutement. Sinon, il détermine l'adresse physique en recopiant dans les 3 bits de poids le plus fort le numéro de case `(110)` correspondant au numéro de page (`0010`), et dans les 12 bits de poids le plus faible de l'adresse virtuelle. L'adresse virtuelle `8196` (`0010 0000 0000 0100`) est convertie alors en adresse physique `24580` (`1100 0000 000 0100`) comme le montre la figure 10.6. Le SE maintient une copie de la table

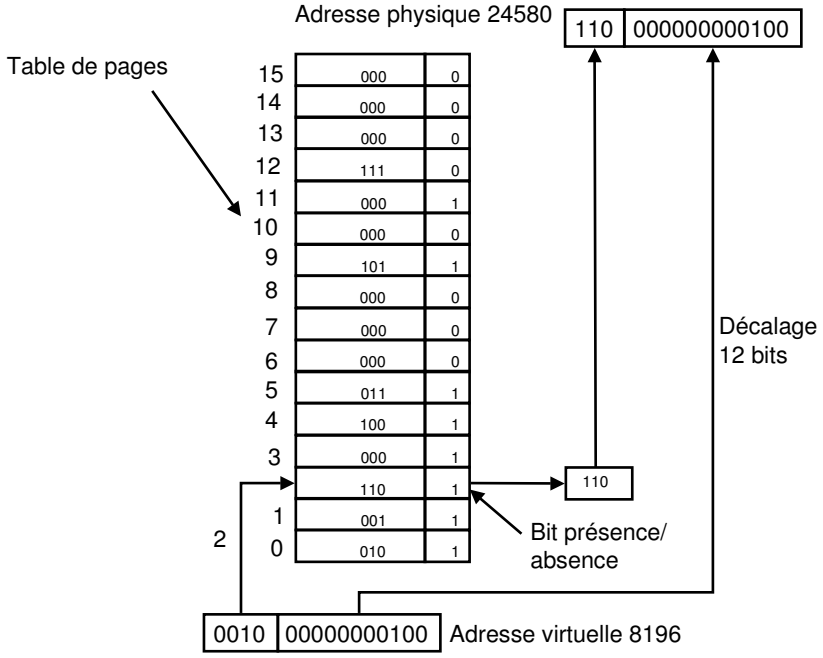

FIG. 10.6 – Opérations du *MMU*.

de pages pour processus, qui permet d'effectuer la traslation des adresses.

Cette table permet aussi à l'ordonnanceur d'allouer la UCT à un processus qui devient prêt. La pagination augmente alors le temps de changement de contexte.

Table de pages à plusieurs niveaux

La taille de la table de pages peut être très grande : par exemple, on aurait plus de 1 million d'entrées (2^{20}) pour un adressage virtuel sur 32 bits et des pages de 4 Ko. Pour éviter d'avoir des tables trop grandes en mémoire, de nombreux ordinateurs utilisent des tables des pages à plusieurs niveaux. Un schéma de table de pages à deux niveaux est montrée sur la figure 10.7.

Par exemple, une table de pages à deux niveaux, pour un adressage sur 32 bits et des pages de 4 Ko, est composée de 1024 tables de 1 Ko. Il est ainsi possible de charger uniquement les tables de 1 Ko nécessaires. Dans ce cas, une adresse virtuelle de 32 bits est composée de trois champs : un pointeur sur la table du 1er niveau, un pointeur sur une table du 2ème niveau et un déplacement dans la page, de 12 bits. Maintenant, si l'on reprend l'exemple

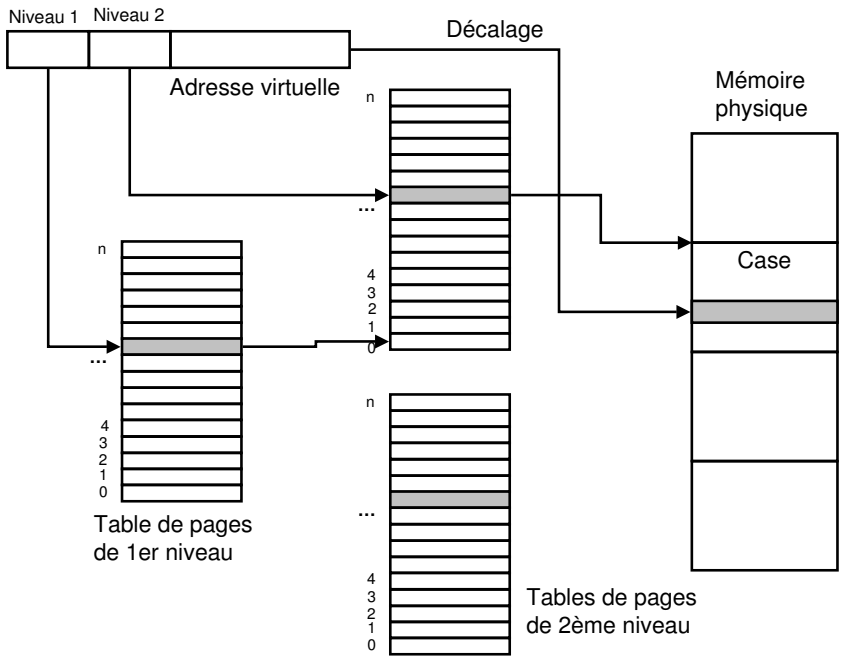

FIG. 10.7 – Table de pages à deux niveaux.

cité auparavant —UCT avec des adresses logiques à 32 bits, des pages à 4 Ko avec des entrées à 4 octets— et si le schéma proposé est de deux niveaux avec 10 bits d'adresses dans chaque niveau. La table de pages de premier niveau aurait une taille de 4 Ko (2^{10} entrées de 4 octets) qui pointeraient vers 1024 tables de pages de deuxième niveau. A leur tour, chaque table de pages de deuxième niveau va utiliser 4 Ko (2^{10} entrées de 4 octets) qui pointeraient vers 1024 cadres. Chaque table de deuxième niveau aura un espace d'adressage de 4 Mo (1024 cadres * 4 Ko).

10.2.2 Accès à la table de pages

L'accès à la table de pages (qui se trouve en mémoire) peut se faire via un registre du MMU qui pointe sur la table de pages. Ceci est une solution lente, car elle nécessite des accès mémoire. Lorsqu'un processus passe à l'état élu, l'adresse de la table de pages est chargée dans le registre. On a constaté que la pagination peut avoir un grand impact sur les performances du système.

MMU avec une mémoire associative

Si la table de pages est grande, la solution précédente n'est pas réalisable. Afin d'accélérer la traduction d'adresses, on peut doter le MMU d'une table de registres machines rapides indexée au moyen du numéro de page virtuelle. Lorsqu'un processus passe à l'état élu, le système d'exploitation charge la **Table de pages** du processus dans la **Table de registres** à partir d'une copie située en mémoire centrale. La solution consiste à doter le MMU d'un dispositif appelé **mémoire associative**, qui est composée d'un petit nombre d'entrées, normalement entre 8 et 64. On a observé que la plupart des programmes ont tendance à faire un grand nombre de références à un petit nombre de pages. La mémoire associative contient des circuits spéciaux pour accéder aux adresses qui sont hautement référencées. La mémoire associative est appelée aussi **TLB** *Translation Lookside Buffer*. Ce composant contient des informations sur les dernières pages référencées. Chaque entrée du **TLB** est composée de :

1. Un bit de validité.
2. Un numéro de page virtuelle.
3. Un bit de modification (M).
4. Deux bits de protection.
5. Un numéro de case.

La traduction d'adresses en utilisant une mémoire associative est montrée à la figure 10.8. Lorsqu'une adresse virtuelle est présentée au MMU,

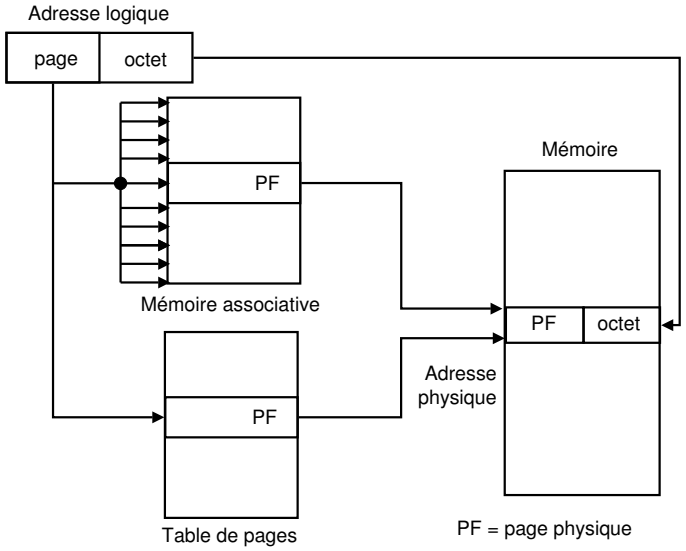

FIG. 10.8 – Mémoire associative.

il contrôle d'abord si le numéro de la page virtuelle est présent dans la mémoire associative (figure 10.8), en le comparant simultanément (en parallèle) à toutes les entrées. S'il le trouve et le mode d'accès est conforme aux bits de protection, la case est prise directement de la mémoire associative (sans passer par la Table de pages). Si le numéro de page est présent dans la mémoire associative mais le mode d'accès est non conforme, il se produit un défaut de protection. Si le numéro de page n'est pas dans la mémoire associative, le MMU accède à la Table de pages à l'entrée correspondant au numéro de pages. Si le bit de présence de l'entrée trouvée est à 1, le MMU remplace une des entrées de la mémoire associative par l'entrée trouvée. Sinon, il provoque un défaut de page.

▶ **Exemple 2.** Supposons qu'il faille 100 ns pour accéder à la table de pages et 20ns pour accéder à la mémoire associative. Si la fraction de références mémoire trouvées dans la mémoire associative (taux d'impact) est s, le temps d'accès moyen est alors :

$$\hat{t} = 20s + 100(1 - s)$$

10.3 Algorithmes de remplacement de page

A la suite d'un défaut de page, le système d'exploitation doit ramener en mémoire la page manquante à partir du disque. S'il n'y a pas de cadres libres en mémoire, il doit retirer une page de la mémoire pour la remplacer par celle demandée. Si la page à retirer a été modifiée depuis son chargement en mémoire, il faut la réécrire sur le disque. Quelle est la page à retirer de manière à minimiser le nombre de défauts de page ?

Le choix de la page à remplacer peut se limiter aux pages du processus qui ont provoqué le défaut de page (**allocation locale**) ou à l'ensemble des pages en mémoire (**allocation globale**).

En général, l'allocation globale produit de meilleurs résultats que l'allocation locale. Des algorithmes de remplacement de page ont été proposés. Ces algorithmes mémorisent les références passées aux pages. Le choix de la page à retirer dépend des références passées. Il y a plusieurs algorithmes de remplacement de page : l'algorithme de remplacement aléatoire qui choisit au hasard la page victime (qui n'est utilisé que pour des comparaisons), l'algorithme optimal de Belady, l'algorithme de remplacement de la page non récemment utilisée (NRU), l'algorithme FIFO, l'algorithme de l'horloge, et l'algorithme de la page la moins récemment utilisée, et bien d'autres.

10.3.1 Remplacement de page optimal (Algorithme de Belady)

L'algorithme optimal de Belady consiste à retirer la page qui sera référencée le plus tard possible dans le futur. Cette stratégie est impossible à mettre en œuvre car il est difficile de prévoir les références futures d'un programme. Le **remplacement de page optimal** a été cependant utilisé comme base de référence pour les autres stratégies, car il minimise le nombre de défauts de page.

► **Exemple 3.** Soit un système avec $m = 3$ cases de mémoire et une suite de références ω={7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1}. À la figure 10.9 on montre une ligne pour chaque case et une colonne pour chaque référence. Chaque élément de ligne i et colonne j montre la page chargée dans la case i après avoir été référencée. Les entrées en **noir** sont des pages chargées après un défaut de page. L'algorithme optimale fait donc 9 défauts de page.

---

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	2	2	2	2	2	2	2	2	2	2	**7**	7	7
1		**0**	0	0	0	0	0	**4**	4	4	**0**	0	0	0	0	0	0	0	0	0
2			**1**	1	1	**3**	3	3	3	3	3	3	3	**1**	1	1	1	1	1	1

FIG. 10.9 – Algorithme optimale de Belady.

10.3.2 Remplacement de la page non récemment utilisée (NRU, *Not Recently Used*)

Cet algorithme utilise les bits **R** et **M** associés à chaque page pour déterminer les pages non récemment utilisées. Le bit **R** est positionné à **1** chaque fois qu'une page est référencée. Le bit **R** est remis à **0** à chaque interruption d'horloge. Le bit **M** est positionné lorsque la page est modifiée (elle n'est plus identique à sa copie sur disque). Lorsqu'un défaut de page se produit, l'algorithme NRU sélectionne la page à retirer en procédant comme suit :

- Il vérifie s'il existe des pages non référencées et non modifiées (**R=0** et **M=0**). Si c'est le cas, il sélectionne une page et la retire.
- Sinon, il vérifie s'il existe des pages non référencées et modifiées (**R=0** et **M=1**). Si c'est le cas, il sélectionne une page et la retire (une sauvegarde sur disque de la page retirée est nécessaire).
- Sinon, il vérifie s'il existe des pages référencées et non modifiées (**R=1** et **M=0**). Si c'est le cas, il sélectionne une page et la retire.
- Sinon, il sélectionne une page référencée et modifiée et la retire (**R=1** et **M=1**). Dans ce cas, une sauvegarde sur disque de la page retirée est nécessaire.

10.3.3 Remplacement de page FIFO

Il mémorise dans une file de discipline FIFO (premier entré, premier sorti) les pages présentes en mémoire. Lorsqu'un défaut de page se produit, il retire la plus ancienne, c'est à dire celle qui se trouve en tête de file.

▶ **Exemple 4.** La suite de références ω={7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1} avec $m = 3$ cases fait 15 défauts de page avec l'algorithme FIFO, comme le montre la figure 10.10.

Cet algorithme ne tient pas compte de l'utilisation de chaque page. Par exemple, à la dixième référence la page 0 est retirée pour être remplacée par la page 3 puis tout de suite après la page retirée est rechargée. L'algorithme

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0	0	0	0	0	**7**	7	7
1		**0**	0	0	0	**3**	3	3	**2**	2	2	2	2	**1**	1	1	1	1	**0**	0
2			**1**	1	1	1	**0**	0	0	**3**	3	3	3	3	**2**	2	2	2	2	**1**

FIG. 10.10 – Algorithme de remplacement FIFO.

est rarement utilisé car il y a beaucoup de défauts de page.

10.3.4 Remplacement de la page la moins récemment utilisée (*LRU Least Recently Used*)

L'algorithme LRU mémorise dans une liste chaînée toutes les pages en mémoire. La page la plus utilisée est en tête de liste et la moins utilisée est en queue. Lorsqu'un défaut de page se produit, la page la moins utilisée est retirée. Pour minimiser la recherche et la modification de la liste chaînée, ces opérations peuvent être réalisées par le matériel. Cet algorithme est coûteux.

▶ **Exemple 5.** La suite de références ω={7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1} avec $m = 3$ cases fait 12 défauts de page avec l'algorithme de remplacement de la page la moins récemment utilisée, comme le montre la figure 10.11.

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0	1	1	1	1	**1**	1	1
1		**0**	0	0	0	0	0	**0**	0	**3**	3	3	3	3	3	**0**	0	0	0	0
2			**1**	1	1	**3**	3	3	**2**	2	2	2	2	2	2	2	2	**7**	7	7

FIG. 10.11 – Algorithme de remplacement de la page la moins récemment utilisée.

Le problème avec cet algorithme est la difficulté d'implémentation, qui requiert un support du hardware. Il faut une manière de mémoriser le temps à chaque fois qu'une page est référencée. On peut aussi utiliser une technique de vieillissement, où un registre de n bits est associé à chaque page. Le bit le plus significatif est mis à 1 chaque fois que la page est référencée. Régulièrement, on décale vers la droite les bits de ce registre. Lorsque qu'on doit expulser une page, on choisit celle dont la valeur est la plus petite. On pourrait aussi mettre une page au dessus d'une pile chaque fois

qu'elle est référencée. On expulsera la page qui se trouve au fond de la pile.

10.3.5 Algorithme de l'horloge

L'algorithme de l'horloge, simple d'implémentation, est une approximation de l'algorithme LRU. Dans cet algorithme , les pages en mémoire sont mémorisées dans une liste circulaire en forme d'horloge (figure 10.12). Un indicateur pointe sur la page la plus ancienne. Lorsqu'un défaut de page se produit, la page pointée par l'indicateur est examinée. Si le bit R de la page pointée par l'indicateur est à 0, la page est retirée, la nouvelle page est insérée à sa place et l'indicateur avance d'une position. Sinon, il est mis à 0 et l'indicateur avance d'une position. Cette opération est répétée jusqu'à ce qu'une page ayant R égal à 0 soit trouvée. À noter que lorsqu'une page est ajoutée, on met son bit de référence à 1.

Cet algorithme est aussi appelé algorithme de la seconde chance. On remarquera qu'il a pour effet de donner une seconde chance aux pages qui ont été référencées depuis la dernière exécution de l'algorithme. Ce n'est que si toutes les pages ont été référencées que l'on revient à la première page pour l'expulser.

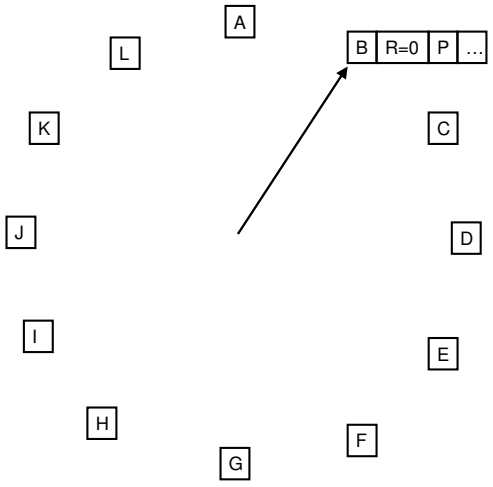

FIG. 10.12 – Algorithme de l'horloge. Si la valeur de $R = 0$ on tue la page et une nouvelle page est insérée, sinon on met $R = 0$ et on avance un tour.

▶ **Exemple 6.** La suite de références ω={7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1} avec $m = 3$ cases fait 14 défauts de page avec l'algorithme de l'horloge,

comme le montre la figure 10.13.

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	**4**	4	4	4	**3**	3	3	3	**0**	0	0	0	0
1		**0**	0	0	0	0	0	0	**2**	2	2	2	2	**1**	1	1	1	**7**	7	7
2			**1**	1	1	**3**	3	3	3	3	**0**	0	0	0	**2**	2	2	2	2	**1**

FIG. 10.13 – Algorithme de remplacement de l'horloge.

Une variante de l'algorithme de l'horloge, que l'on pourrait appeler algorithme de la troisième chance, permet de tenir compte du fait qu'une page ait été modifiée ou non. L'idée de base consiste a expulser d'abord des pages qui n'ont pas été modifiées. Voici en gros le principe utilisé. D'abord, en plus du bit de référence, on utilise un bit pour indiquer si la page a été modifiée. On applique l'algorithme de l'horloge jusqu'à ce qu'on arrive à une page dont les deux bits on la valeur 0 et on expulsera la page en question. Voici les règles appliquées à chaque page visitée :

Valeur des bits avant		*Valeur des bits après*	
Ref	Mod	Ref	Mod
1	1	0	1
1	0	0	0
0	1	0	0*
0	0	expulser	

Dans le cas où le bit Mod est mis à zéro (cas indiqué par l'astérisque dans la table), il faudra sauvegarder la page en mémoire secondaire avant de l'expulser

10.4 Nombre de cases allouées à un processus

On peut allouer un même nombre de cases mémoire à chaque processus. Par exemple, si la mémoire totale fait 100 pages et qu'il y a cinq processus, chaque processus recevra 20 pages. On peut aussi allouer les cases proportionnellement aux tailles des programmes. Si un processus est deux fois plus grand qu'un autre, il recevra le double de cases. L'allocation des cases peut se faire lors du chargement ou à la demande au cours de l'exécution.

10.4.1 Anomalie de Belady

Quand on compare la performance des algorithmes particulièrs, on peut penser à faire varier le nombre de pages allouées au processus. On pourrait ainsi espérer que si l'on incrément le nombre de pages allouées alors le nombre de défauts va diminuer. Mais sous certaines conditions une situation appelée **Anomalie de Belady** peut se produire et l'ajout de plus de pages se traduit par un incrément du nombre de défauts. L'algorithme FIFO est spécialement touché par cette anomalie.

10.5 Écroulement du système

Si le système passe plus de temps à traiter les défauts de page qu'à exécuter des processus on peut avoir des problème de l'écroulement du système. Si le nombre de processus est trop grand, l'espace propre à chacun sera insuffisant, ils passeront alors leur temps à gérer des défauts de pages. C'est ce qu'on appelle l'**écroulement du système**. On peut limiter le risque d'écroulement en surveillant le nombre de défauts de page provoqués par un processus. Si un processus provoque trop de défauts de pages (au-dessus d'une limite supérieure) on lui allouera plus de pages ; au-dessous d'une limite inférieure, on lui en retirera. S'il n'y a plus de pages disponibles et trop de défauts de pages, on devra suspendre un des processus.

10.6 Retour sur instructions

Sur la plupart des processeurs, les instructions se codent sur plusieurs opérandes. Si un défaut de page se produit au milieu d'une instruction, le processeur doit revenir au début de l'instruction initiale pour recommencer son exécution. Ce retour sur instruction n'est possible qu'avec l'aide du matériel.

10.7 Segmentation

Dans un système paginé, l'espace d'adressage virtuel d'un processus est à une dimension. Or en général, un processus est composé d'un ensemble d'unités logiques :

– **Les différents codes :** le programme principal, les procédures, les fonctions bibliothèques.

- **Les données initialisées**.
- **Les données non initialisées**.
- **Les piles d'exécution**.

L'idée de la technique de **segmentation** est d'avoir un espace d'adressage à deux dimensions. On peut associer à chaque unité logique un espace d'adressage appelé **segment**. L'espace d'adressage d'un processus est composé d'un ensemble de segments. Ces segments sont de tailles différentes (fragmentation externe). Un schéma de traduction d'adresses par segmentation est montré sur la figure 10.14. La segmentation facilite l'édition de

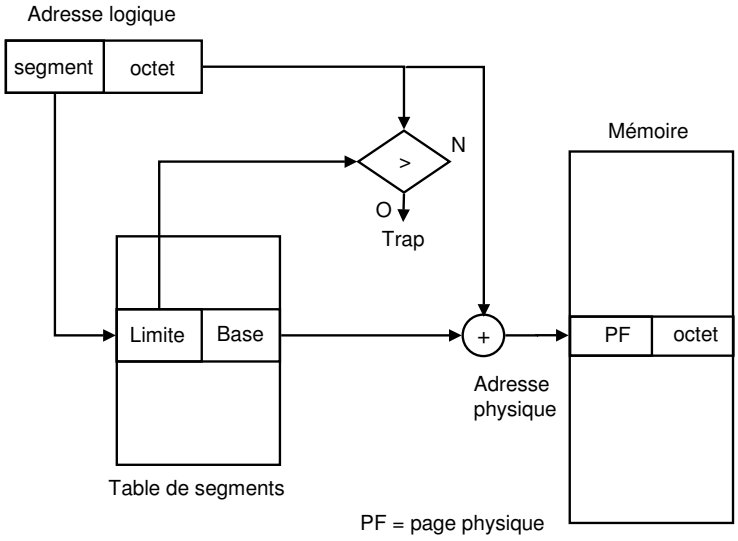

FIG. 10.14 – Segmentation simple.

liens, ainsi que le partage entre processus de segments de données ou de codes.

10.8 Segmentation paginée

La segmentation peut être combinée avec la pagination. Chaque segment est composé d'un ensemble de pages. Les adresses générées par les compilateurs et les éditeurs de liens, dans ce cas, sont alors des triplets :

```
<numéro du segment, numéro de page, déplacement dans la page>
```

Le schéma de traduction d'adresses par segmentation paginée est montré sur la figure 10.15.

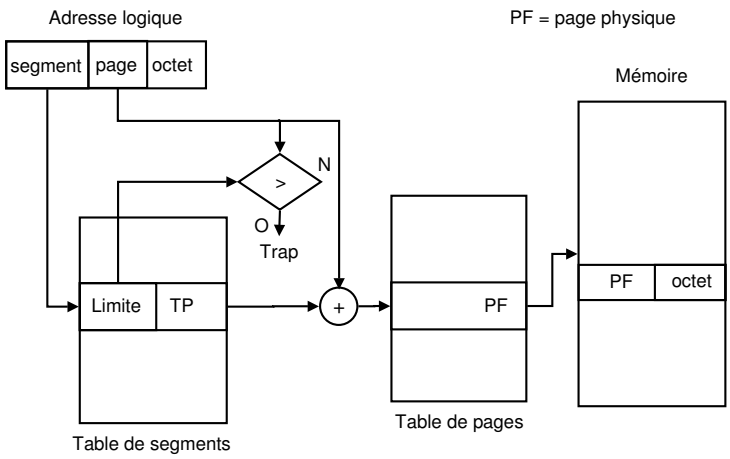

FIG. 10.15 – Segmentation paginée.

10.9 Mémoire cache

La mémoire cache est une mémoire à temps d'accès très court ($\leq$ 10 ns). Elle coûte plus chère. Le temps d'accès à la mémoire principale est 100 ns. La mémoire cache est placée entre le processeur et la mémoire centrale, comme montré sur la figure de la hiérarchie de la mémoire 10.16. Dans un système à pagination, lorsqu'une adresse virtuelle est référencée, le système examine si la page est présente dans le cache. Si c'est le cas, l'adresse virtuelle est convertie en adresse physique. Sinon, le système localise la page puis la recopie dans le cache. Le but du cache est de minimiser le temps d'accès moyen $\hat{t}$:

$\hat{t}$ = temps d'accès + taux d'échec $\cdot$ temps de traitement de l'échec

10.10 Cas d'étude

10.10.1 Unix

Les premiers systèmes Unix utilisaient la technique de va-et-vient. Avant l'exécution, un processus est entièrement chargé en mémoire. S'il n'y a pas assez de place, un ou plusieurs processus sont transférés sur le disque.

Le choix du processus à transférer sur le disque dépend essentiellement de trois facteurs : l'état bloqué ou prêt, la priorité et le temps de résidence en mémoire. Toutes les quelques secondes, le permuteur (*swapper* ou **chargeur**) processus de `pid 0` examine l'état des processus transférés sur le

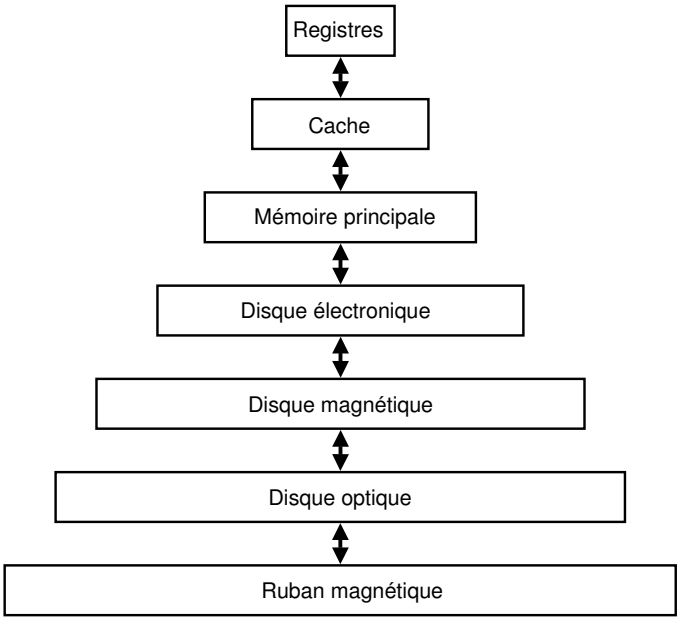

FIG. 10.16 – Hiérarchie de la mémoire.

disque pour voir si certains d'entre eux sont devenus prêts. Si c'est le cas, il choisit celui qui est resté trop longtemps sur le disque pour le transférer en mémoire centrale. Cette procédure est répétée jusqu'à ce qu'il n'y ait plus de processus prêts sur le disque, ou tous les processus soient nouvellement chargés et il n'y ait pas assez d'espace mémoire.

Le permuteur mémorise, dans deux listes chaînées, les espaces libres en mémoire et sur le disque. La stratégie d'allocation utilisée est le premier ajustement (première zone libre suffisamment grande). La pagination à la demande a été introduite dans les systèmes Unix par Berkeley (depuis BSD 4.3 et **System V**). Seules les pages nécessaires à l'exécution sont ramenées en mémoire centrale.

Le **voleur de pages** est le processus qui exécute l'algorithme de remplacement de pages. Il est réveillé périodiquement (toutes les secondes) pour voir si le nombre de cases libres en mémoire est au moins égal à un seuil `min`. Si le nombre de cases libres en mémoire est inférieur à `min`, le voleur de pages transfère des pages sur le disque jusqu'à ce que `max` cadres soient disponibles. Sinon, il se remet au sommeil.

Pour le choix des pages à transférer, le voleur de page utilise une version améliorée de l'algorithme de l'horloge. Chaque case est dotée d'un bit

de référence **R** qui est mis à `1` à chaque référence. Il parcourt les pages en mémoire et teste leurs bits de référence. S'il est à `0` alors le voleur de pages incrémente une variable "`âge`" associée à la page, sinon le bit de référence est remis à zéro. Lorsque `âge` dépasse une valeur donnée, la page est placée dans l'état disponible pour le déchargement.

10.10.2 Linux

Sous Linux le format des fichiers exécutables est divisé en régions ou zones, comme suit :

- Type du fichier (les deux premiers octets).
- Zone de code `.text`
- Zone des données `.data`
- Zone des données initialisées `.bss`
- Zone de la pile.

La taille d'un fichier exécutable peut être determinée par la commande `size` :

```
leibnitz> size /vmunix
text data bss dec hex
3243344 494336 856864 4594544 461b70
```

La zone de la pile ou *stack* est visible uniquement à l'exéecution, et elle contient les variables locales et les arguments des fonctions, enre autres.

Chaque processus sur une machine `80x86` de 32-bits a 3 Go d'espace d'adressage virtuel, le Go restant est réservé aux Tables de pages et certaines données du Kernel du système d'exploitation (voir figure 10.17). Les programmes s'exécutent dans l'espace d'utilisateur et les accès à la mémoire se font dans des adresses $\leq$ `0xbfffffff`. Des mécanismes de protection empêchent les programmes d'accéder à l'espace du Kernel, mais on permet de basculer l'exécution vers cet espace lors des appels au système et l'accès aux Entrées/Sorties. C'est le SE qui est chargé d'initialiser correctement l'espace utilisateur lors du chargement d'un programme. Toutes les informations sont contenues dans l'en-tête du fichier exécutable. Un format couramment utilisé sous Linux est le format **ELF** (*Executable and Linkable Format*), qui contient de l'information sur les segments, les librairies dynamiques, l'adresse de la première instruction à exécuter, entre autres. D'autres détails sur le contenu des registres du processeur lors du chargement d'un exécutable sous format **ELF** se trouvent dans [**?**].

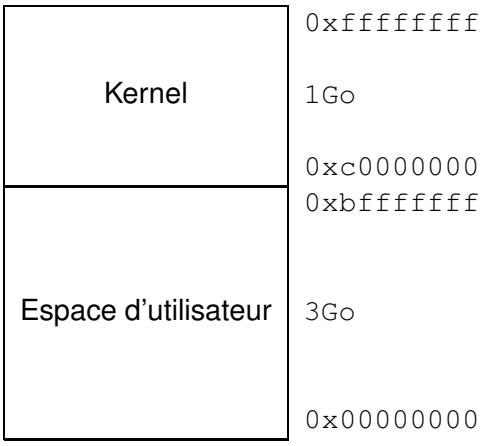

FIG. 10.17 – Espace mémoire sous Linux.

L'espace d'adressage est composé d'un ensemble de régions. Chaque région est un ensemble de pages contiguës de 4 Ko. Par exemple, le segment de code est une région[1].

Segmentation paginée dans le 80x86

Le processeur 80x86 utilise la segmentation paginée pour gérer la mémoire. La mémoire physique est divisée en pages de 4Ko. Le nombre maximum de segments par processus est de 16K, et la taille de chaque segment peut aller jusqu'à 4Go. L'idée est d'effectuer une translation d'adresses logiques en adresses linéaires puis en adresses physiques (figure 10.18).

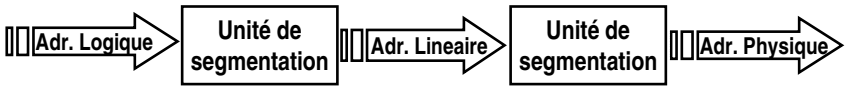

FIG. 10.18 – Segmentation paginée du 80x86 sous Linux (1).

Une **adresse logique** est une paire (**sélecteur**, **déplacement**). Le sélecteur est codé sur 16 bits, et le déplacement sur 32 bits (figure 10.19).

Une **adresse linéaire** de 32 bits est divisée en 3 parties (2 niveaux) :

– Pointeur de répertoire de pages sur 10 bits.

– Pointeur de table de pages sur 10 bits.

[1]Pour l'assembleur GNU `as` le segment de code est nommé `.text`, le segment des données `.data`, le segment des données initialisées à zéro `.bss`, etc.

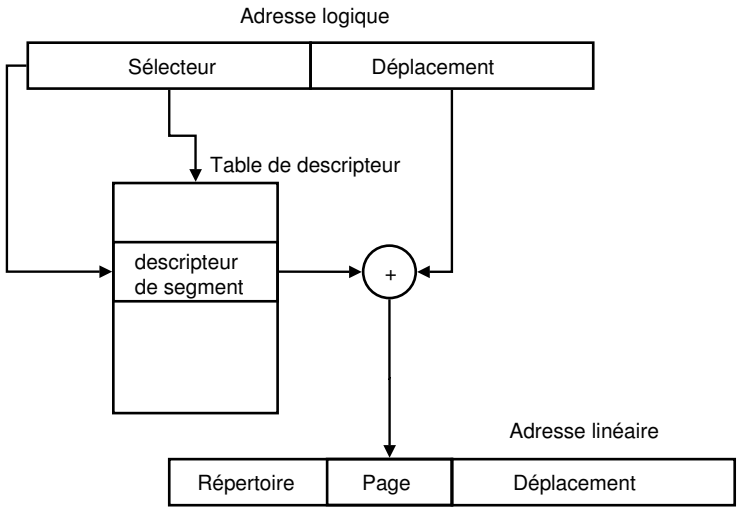

FIG. 10.19 – Traslation d'adresse logique → adresse linéaire.

– Déplacement dans la page sur 12 bits.

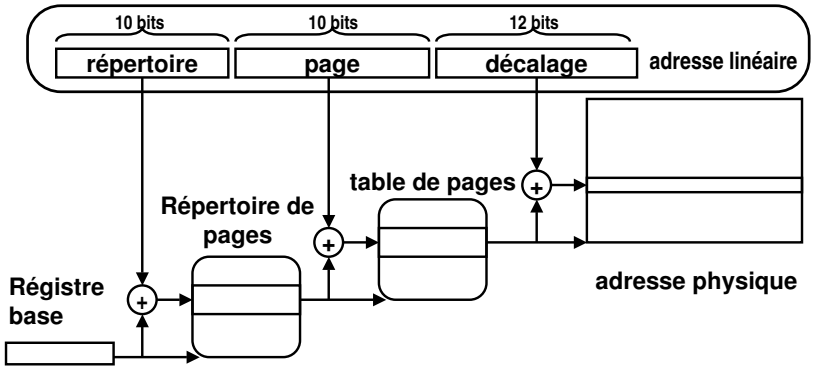

FIG. 10.20 – Segmentation paginée du 80x86 sous Linux (2).

Le **gestionnaire de mémoire** est un système de pagination à la demande (voir figure 10.20). La mémoire centrale est gérée comme suit : initialement, la mémoire est composée d'une seule zone libre. Lorsqu'une demande d'allocation arrive, la taille de l'espace demandé est arrondie à une puissance de 2. La zone libre initiale est divisée en deux. Si la première est trop grande, elle est, à son tour, divisée en deux et ainsi de suite. Sinon, elle est allouée au demandeur. Le gestionnaire de la mémoire utilise un tableau qui contient

des têtes de listes. Le premier élément du tableau contient la tête de la liste des zones de taille 1. Le deuxième élément contient la tête de la liste des zones de taille 2 et ainsi de suite. Cet algorithme *Buddy* conduit vers une importante fragmentation interne. Les espaces non utilisés (de la fragmentation interne) sont récupérés et gérés différemment. Lors de la libération de l'espace, les zones contiguës de même taille sont regroupées en une seule zone.

Comme dans Unix, un démon se charge de maintenir au moins un certain nombre de pages libres en mémoire. Il vérifie périodiquement (toutes les secondes) ou après une forte allocation d'espace, l'espace disponible. Si l'espace disponible devient insuffisant, il libère certaines cases (pages) en prenant soin de recopier sur disque celles qui ont été modifiées, en utilisant l'algorithme de l'horloge.

Commande `vmstat`

L'appel système `vmstat` fourni des renseignements et des statistiques sur l'usage de la mémoire virtuelle du système :

```
leibnitz> vmstat
   procs                  memory   swap     io    system         cpu
 r  b  w  swpd  free  buff cache si  so bi  bo  in    cs  us sy  id
 1  0  0  1480 79980 20192 99232  0   0  3   4 226    90  88  1  11
```

où (d'après `man vmstat`) :

`procs`
- `r` : Processus en attente d'UCT.
- `b` : Nombre de processus en *sleep* non interrompible.
- `w` : Nombre de processus *swapped out* mais prêts.

`memory`
- `swpd` : Quantité de mémoire virtuelle utilisée (Ko).
- `free` : Quantité de mémoire libre (Ko).
- `buff` : Quantité de mémoire utilisée comme buffers (Ko).

`swap`
- `si` : Quantité de mémoire *swapped in* depuis le disque (Ko/s).
- `so` : Quantité de mémoire *swapped* au disque (Ko/s).

`io`
- `bi` : Blocs envoyés (blocks/s).
- `bo` : Blocs reçus (blocks/s).

`system`
- `in` : Interruptions à la seconde, incluant l'horloge.

– `cs` : Changements de contexte à la seconde.

`cpu` (Pourcentages de l'utilisation de l'UCT)

– `us` : Temps utilisateur.
– `sy` : Temps du système.
– `id` : Temps non utilisé.

10.10.3 MS-DOS

MS-DOS ne fait ni de va-et-vient ni de pagination à la demande. Plusieurs programmes peuvent être en même temps en mémoire. La mémoire utilisateur est partagée en zones contiguës (arenas). Chaque zone commence par une en-tête. Ce dernier contient notamment la taille de la zone et un pointeur sur le descripteur du processus qui occupe la zone ou 0 si la zone est libre. Lorsque MS-DOS doit allouer de la mémoire, il parcourt la chaîne des zones jusqu'à trouver une zone libre assez grande (premier ajustement). Si la zone trouvée est très grande, elle est scindée en deux parties. Une partie est allouée et l'autre est libre. S'il n'y a pas assez d'espace, il retourne un message d'erreur à la demande d'allocation. La taille des programmes est limitée aux 640 Ko. Des techniques de segments de recouvrement et de mémoire paginée permettent de s'affranchir de cette limite, mais il faut pour cela installer des pilotes de mémoire étendue.

10.11 Exercises

1. Expliquez un avantage majeur que procure l'utilisation de la mémoire virtuelle sur la performance du système.
2. Quelle est la différence principale entre un algorithme de remplacement statique de pages et un algorithme de remplacement dynamique de pages ?
3. Considérez une mémoire virtuelle avec une taille de mémoire physique (principale) de 1 Mo, et supportant des blocs de 128 octets. Aussi, supposez un processus occupant un espace d'adresse logique de 22 Ko.
   – Calculez le nombre de cadres dans l'espace d'adresse physique et le nombre de pages dans l'espace d'adresse logique.
   – Montrez les formats des adresses physique et logique, soit le nombre de bits pour les blocs, les cadres et les pages.
   – Déterminez l'adresse physique dans laquelle se situe l'adresse logique 10237, si l'on suppose que la page contenant l'adresse 10237 se trouve dans le cadre 1839.
4. Considérez la séquence de références de pages ω=0, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9 faite par un processus. Montrez la séquence d'allocation des pages en mémoire pour chacun des algorithmes énumérés ci-dessous. Aussi, calculez le taux de faute de pages produit par chacun des algorithmes de remplacement de pages de mémoire.
   (a) L'algorithme optimal de Belady.
   (b) L'algorithme du moins récemment utilisé.
   (c) L'algorithme du moins fréquemment utilisé.
   (d) L'algorithme du premier-arrivé-premier-servi (PAPS).
   (e) L'algorithme de l'espace vital avec une taille de fenêtre V=3. Variez la valeur de V afin de déterminer le point d'allocation optimal. Que survient-il si l'on alloue plus de cadres au processus que son niveau optimal ?
5. On considère un ordinateur dont le système de mémoire virtuelle dispose de 4 cases (frames ou cadres) de mémoire physique pour un espace virtuel de 8 pages. On suppose que les quatre cases sont initialement vides et que les pages sont appelées dans l'ordre suivant au cours de l'exécution d'un processus par le processeur : 1, 2, 3, 1, 7, 4, 1, 8, 2, 7, 8, 4, 3, 8, 1, 1. Indiquez tout au long de la séquence d'exécution

quelles pages sont présentes dans les cases de la mémoire physique et le nombre de défauts de page selon que l'algorithme de remplacement de pages est :

(a) PAPS.

(b) L'algorithme de remplacement optimal.

6. Supposez un système de mémoire paginée ayant 2g+h adresses virtuelles et 2h+k adresses en mémoire principale, d'où g, h, et k sont des nombres entiers. Spécifiez la taille des blocs et le nombre de bits pour les adresses virtuelle et physique.

7. Supposez une taille de blocs de 1 Ko, déterminez le numéro de page et le numéro de ligne pour chacune des adresses virtuelles suivantes.

(a) 950

(b) 1851

(c) 25354

(d) 11842