Chapitre 5

Gestion de la mémoire

La gestion de la mémoire principale de l'ordinateur est du ressort du système de gestion de la mémoire.

5.1 Gestion mémoire basique

On peut subdiviser les systèmes de gestion de la mémoire en deux catégories :

- les systèmes qui peuvent déplacer les processus en mémoire secondaire pour trouver de l'espace (va-et-vient) ;
- les systèmes qui n'utilisent pas la mémoire secondaire.

5.1.1 Gestion de la mémoire sans va-et-vient ni pagination

Monoprogrammation sans va-et-vient ni pagination

L'approche la plus simple pour gérer la mémoire consiste à n'accepter qu'un seul processus à la fois (monoprogrammation) auquel on permet d'utiliser toute la mémoire disponible en dehors de celle qu'utilise le système.

Cette approche était utilisée, par exemple, dans les premiers micro-ordinateurs IBM PC utilisant MS-DOS comme système d'exploitation (cf. figure 5.1 (c)).

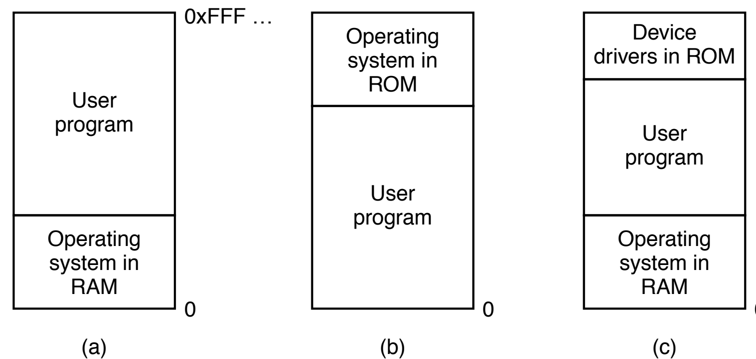


FIG. 5.1 – Trois façons d'organiser la mémoire en monoprogrammation.

Multiprogrammation avec des partitions fixes

Dans un système multiprogrammé, il faut gérer la répartition de l'espace entre les différents processus. Cette partition peut être faite une fois pour toute au démarrage du système par l'opérateur de la machine, qui subdivise la mémoire en partitions fixes.

Chaque nouveau processus est placé dans la file d'attente de la plus petite partition qui peut le contenir (cf. figure 5.2 (a)). 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 (cf. figure 5.2 (b)).

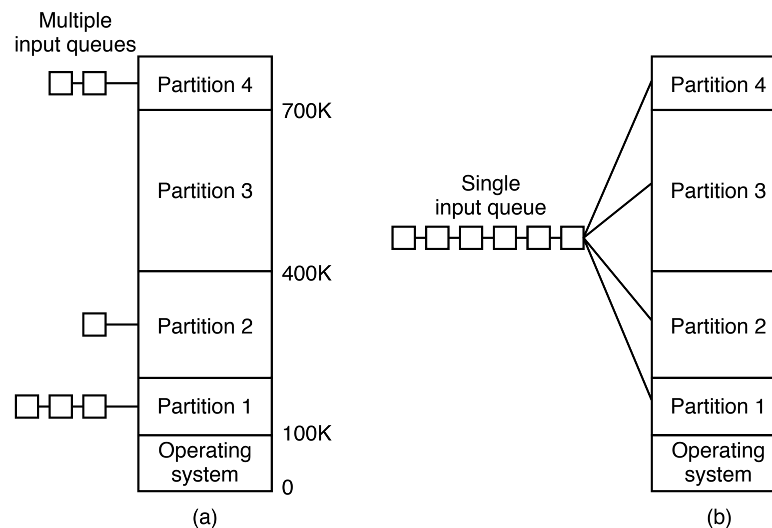


FIG. 5.2 – Partition fixe de la mémoire

Cette approche a été longtemps utilisée dans de gros ordinateurs IBM munis du système OS/360.

5.2 Va-et-vient (*swaping*)

Dans un système qui peut déplacer les processus sur le disque quand il manque d'espace, le mouvement des processus entre la mémoire et le disque (va-et-vient) risque de devenir très fréquent, si on n'exploite pas au mieux l'espace libre en mémoire principale. Sachant que les accès au disque son très lents, les performances du système risquent alors de se déteriorer rapidement. Il faut alors exploiter au mieux l'espace libre en mémoire principale, en utilisant un partitionnement dynamique de la mémoire.

Le fait de ne plus fixer le partitionnement de la mémoire rend plus complexe la gestion de l'espace libre. Celui-ci doit cependant permettre de trouver et de choisir rapidement de la mémoire libre pour l'attribuer à un processus. Il est donc nécessaire de mettre en oeuvre des structures de données efficaces pour la gestion de l'espace libre.

5.2.1 Gestion de la mémoire par tables de bits

La mémoire est divisée en unités (quelques mots mémoire à plusieurs kilo-octets), à chaque unité on fait correspondre dans une table de bits :

- la valeur 1 si l'unité mémoire est occupée ;
- la valeur 0 si l'unité mémoire est libre.

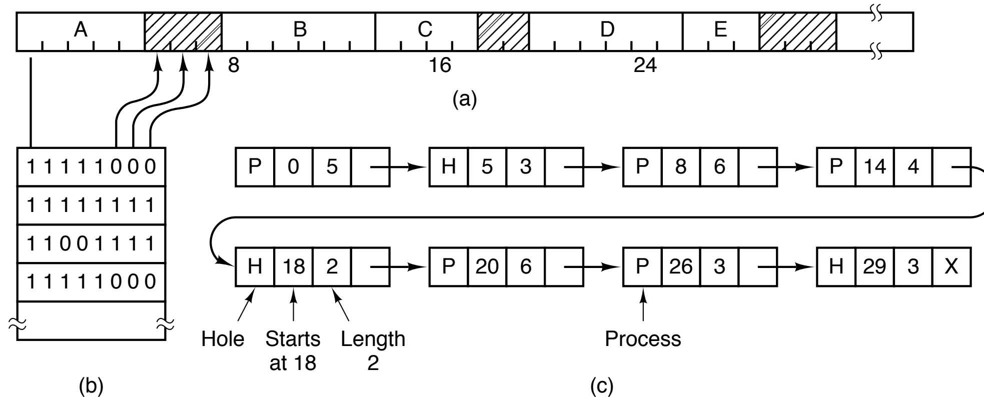


FIG. 5.3 – (a) Une partie de la mémoire occupée par cinq processus (A,B,C,D et E) et trois zones libres. Les régions hachurées sont libres. (b) La table de bits (0 = libre, 1 = occupée). (c) Liste chainée des zones libres.

La figure 5.3 (a) et (b) représente un exemple de table de bits correspondant à l'occupation de la mémoire par cinq processus.

Pour allouer k unités contigües, le gestionnaire de mémoire doit parcourir la table de bits à la recherche de k zéro consécutifs. Cette recherche s'avère donc lente pour une utilisation par le gestionnaire de la mémoire et est rarement utilisée à cet effet.

5.2.2 Gestion de la mémoire par listes chaînées

Une autre solution consiste à chaîner les segments libres et occupés. La figure 5.3 (c) montre l'exemple d'un tel chaînage, les segments occupés par un processus sont marqués (P) les libres sont marqués (H). La liste est triée sur les adresses, ce qui facilite la mise à jour.

Lorsqu'on libère la mémoire occupée par un segment, il faut fusionner le segment libre avec le ou les segments adjacents libres s'ils existent (cf. figure 5.4).

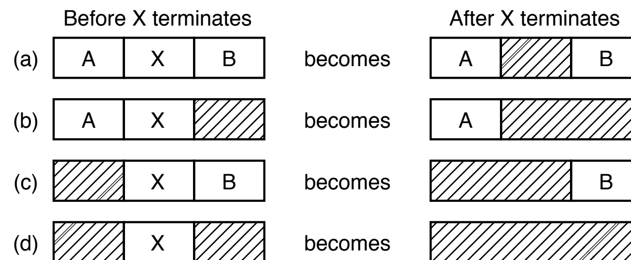


FIG. 5.4 – Quatre combinaisons de voisins possibles d'un processus X qui termine et libère le segment qu'il occupe

On peut utiliser différents algorithmes pour le choix de la zone libre lors d'une allocation dans un système gérant une liste de segments. L'objectif est d'éviter de trop fragmenter la mémoire, tout en ayant un algorithme relativement rapide :

- L'algorithme de la première zone libre : consiste à choisir le premier segment libre de la liste qui est de taille suffisante, le segment est scindé en deux : une zone sera occupée par le nouveau segment occupé et l'autre (s'il reste de l'espace inutilisé) sera un segment libre adjacent au premier. Cet algorithme est rapide puisqu'il requiert peu de recherche.
- L'algorithme de la zone libre suivante : identique à la méthode précédente sauf que la prochaine recherche commence au segment qui suit celui dans lequel on s'est précédement arrété. A la similation ce dernier algorithme ne donne pas de meilleurs résultats que le premier.
- L'algorithme du meilleur ajustement : on cherche dans toute la liste la plus petite zone qui suffit à contenir la taille du segment à allouer. On cherche à éviter de fragmenter la mémoire au dépend du temps (la recherche dans toute la liste nécessite beaucoup de temps). Malheuresement cette méthode a tendance à trop fragmenter la mémoire du fait quelle a tendance à créer de courts espaces libres inutilisables.
- L'algorithme du plus grand résidu : Consiste au contraire de la précédente à choisir toujours la plus grande zone libre, cette méthode ne donne pas de meilleurs résultats.

5.3 La mémoire virtuelle

Le principe de la mémoire vituelle consiste à considérer un espace d'adressage virtuel supérieur à la taille de la mémoire physique, sachant que dans cette espace d'adressage, et grâce au mécanisme de va-et-viens sur le disque, seule une partie de la mémoire virtuelle est physiquement présente en

mémoire principale à un instant donné. Ceci permet de gérer un espace virtuel beaucoup plus grand que l'espace physique **sans** avoir à gérer les changements d'adresses physiques des processus après un va-et-vient : car même après de multiples va-et-vients un processus garde la même adresse virtuelle. C'est notamment très utile dans les systèmes multiprogrammés dans lesquels les processus qui ne font rien (la plupart) occupent un espace virtuel qui n'encombre pas nécéssairement l'espace physique.

5.3.1 Pagination

La plupart des architectures actuellement utilisées reposent sur des processeurs permettant de gérer un espace virtuel paginé : l'espace d'adressage virtuel est divisé en **pages**, chaque page occupée par un processus est soit en mémoire physique soit dans le disque (va-et-vient).

Table de pages

La correspondance entre une page virtuelle et la page physique auquelle elle peut être associée (si elle est en mémoire physique) est réalisée en utilisant une table de pages (cf. figure 5.5).

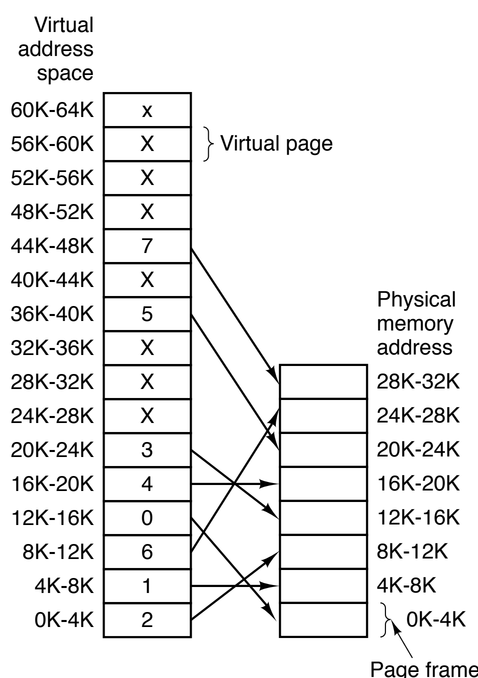


FIG. 5.5 – Espace d'adressage virtuel et sa correspondance avec l'espace physique

Lorsque le processeur doit exécuter une instruction qui porte sur un mot mémoire donné dont il a l'adresse virtuel, il cherche dans la table des pages l'entrée qui correspond à la page contenant le mot. Si la page est présente en mémoire il lit le mot, sinon il déclenche un déroutement de type *défaut de page*. A la suite de ce déroutement, le système de gestion de la mémoire est appellé afin de charger la page manquante à partir du disque (va-et-vient).

Notez que la consultation de la table de pages est à la charge du processeur (tâche cablée), alors que le système d'exploitation intervient lors d'un défaut de page pour prendre en charge le va-et-vient.

Descripteur de pages

Chaque entrée de table de pages consiste en un **descripteur de page** qui contient généralement ces informations (au moins) :
- Numéro de la page en mémoire physique (cf. figure 5.5) si celle-ci est présente en mémoire.
- Un bit qui indique la présence (ou non présence) de la page en mémoire.
- Un bit de modification : qui mémorise le fait qu'une page a été modifiée, ceci permet au gestionnaire de mémoire de voir s'il doit la sauver sur le disque dans le cas ou il veut récupérer l'espace qu'elle occupe pour charger une autre page.
- Un bit de référencement : qui est positionné si on fait référence à un mot mémoire de la page, ceci permet au système de trouver les pages non référencées qui seront les meilleures candidates pour être retirées de la mémoire physique s'il y a besoin et manque de mémoire physique.

Tables de pages à plusieurs niveaux

Une table de pages unique risque d'occuper beaucoup d'espace mémoire physique. Pour éviter cela, on utilise des tables à plusieurs niveaux (cf. figure 5.6). La table de pages de niveau 1, pointe sur une table de pages de niveau 2, ainsi de suite jusqu'a arriver à la dernière table de pages contenant les descripteurs des pages. Seules les tables d'indirections nécessaires à un moment donné doivent être présentes en mémoire physique.

Dans le cas d'une table à deux niveaux (cf. figure 5.6), l'adresse virtuelle d'une case mémoire doit donc comprendre :
- l'index dans la table de niveau 1 (PT1 dans la figure) ;
- l'index dans la table de niveau 2 (PT2) ;
- le déplacement (offset) dans la page.

5.3.2 Les tables de pages inverses

Dans le cas ou l'espace d'adressage virtuel est gigantesque par rapport à l'espace physique, les tables de pages deviennent creuses et occuppent inutilement la mémoire principale et secondaire.

Il devient alors préférable dans ce cas de concevoir des machines dans lesquelles la table des pages est indexée par le numéro de la page physique et dont chaque entrée contient le numéro de page virtuelle. Trouver le numéro de page physique associé à un numéro de page virtuel, revient alors à faire une recherche de ce dernier dans la table de pages inverses. Cette recherche pouvant être couteuse en temps, la technique des tables de pages inverses est toujours couplée avec une mémoire associative qui maintient les numéros de pages physiques des dernières pages référencées.

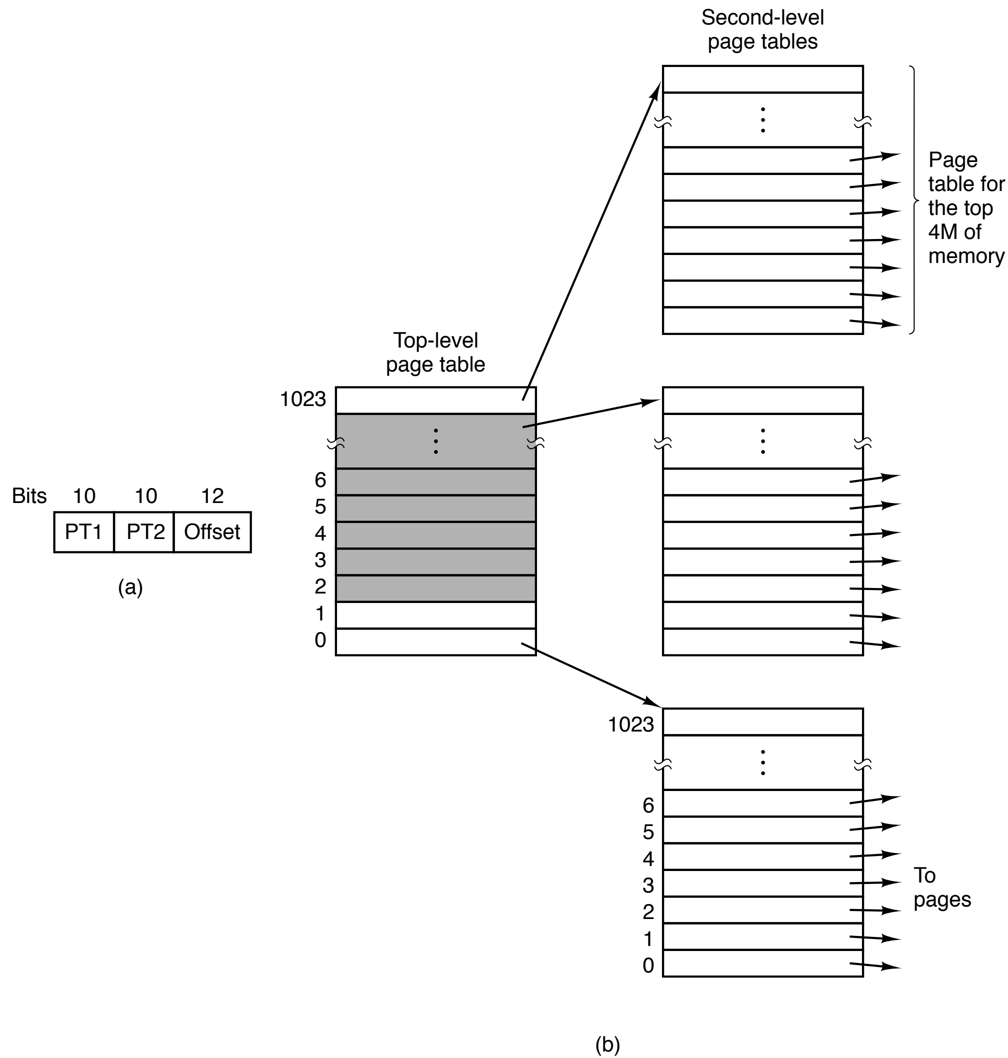


FIG. 5.6 – Tables de pages à deux niveaux.

5.4 Les algorithmes de remplacement de pages

Lorsque le système de gestion de la mémoire manque de mémoire physique, il doit évincer des pages en les sauvant si besoin est (dans le cas où elles ont été modifiées) dans le disque (va-et-vient), ceci afin de les remplacer par des pages demandées par le processeur. Le problème du choix des pages à remplacer se pose alors.

5.4.1 Remplacement de page optimal

L'algorithme de remplacement de page optimal ne peut être mis en oeuvre mais peut servir dans les simulations comme référence pour estimer les performances des autres algorithmes.

L'algorithme consiste à retirer la page qui sera référencée dans le plus lointain avenir. Ce qui suppose de connaître la date à laquelle chaque page sera référencée dans le futur, ce qui est

irréalisable en pratique.

5.4.2 Remplacement de la page non récemment utilisée

L'algorithme essaye d'abors de retirer une page non référencée, s'il n'en trouve pas, il retire une page référencée mais pas modifiée, s'il n'en existe pas, il retire une page quelconque parmis celles qui restent. Périodiquement le bit de référence est remis à 0 afin de différencier les pages récemment référencées des autres pages.

5.4.3 Premier entré, premier sorti

Cet algorithme est peu utilisé tel quel, car la page la plus ancienne n'est pas forcément la moins utilisée dans le futur.

5.4.4 Remplacement avec seconde chance

Cet algorithme utilise le principe du Premier entré, premier sorti modifié comme suit : si la page en tête de file est référencée, elle est remise en queue de la file, puis la suivante est traitée de la même façon, sinon la page est choisie pour être évincée.

5.4.5 Remplacement avec horloge

Cet algorithme est le même que le précédent sauf qu'il utilise une liste circulaire pour implanter la file.

5.4.6 Remplacement de la page la moins récemment utilisée

Consiste à supprimer la page restée inutilisée pendant le plus de temps.

5.5 La segmentation

5.5.1 Implantation de segments purs

La segmentation de la mémoire permet de traiter la mémoire non plus comme un seul espace d'adressage unique, mais plutot comme un ensemble de segments (portions de la mémoire de taille variable), ayant chacune son propre espace d'adressage. Ceci permet aux processus de simplifier considérablement le gestion de mémoire propre.

Ainsi un processus qui utilise différentes tables : table des symboles, code, table des constantes, etc. . .doit se préoccuper de la position relative de ses tables et doit gérer les déplacements de tables quand celles-ci croissent et risquent de se recouvrir (cf. figure 5.7).

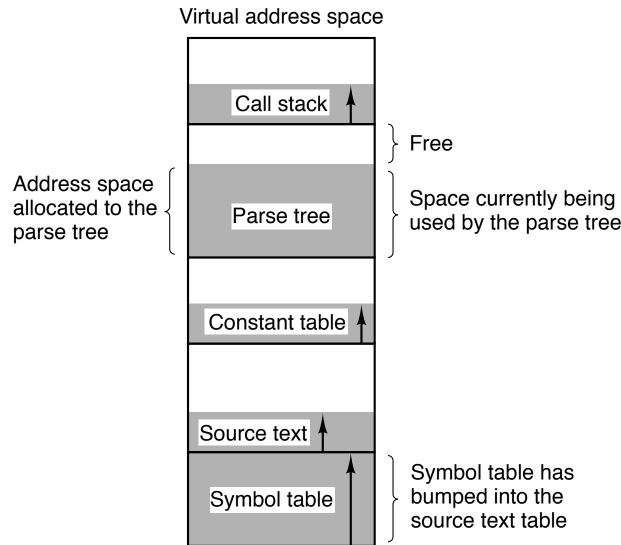


FIG. 5.7 – Espace d'adressage unique

Alors que dans une gestion de la mémoire segmentée, il lui suffit d'allouer un segment pour chaque table, la gestion de la position des segments dans l'espace d'adressage de la mémoire physique devient à la charge du système (cf. figure 5.8).

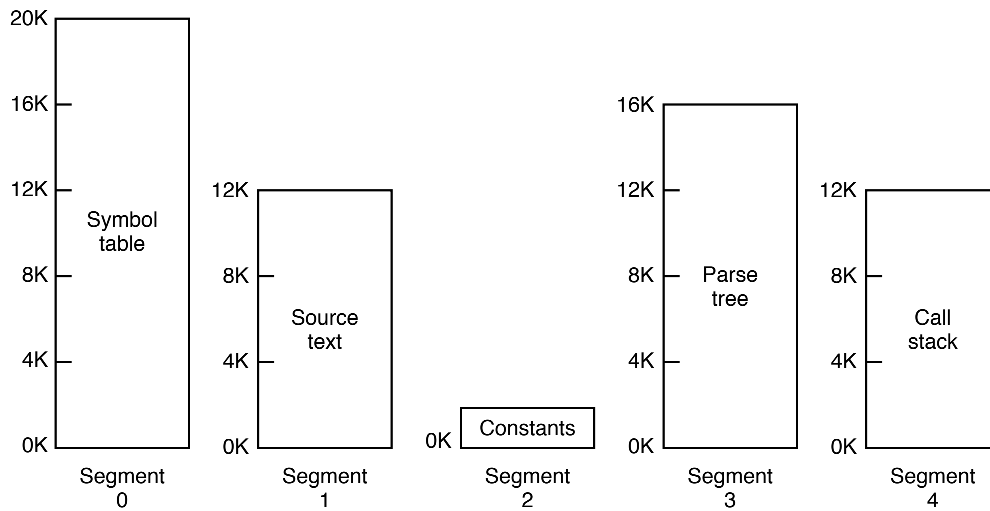


FIG. 5.8 – Espace d'adressage segmenté

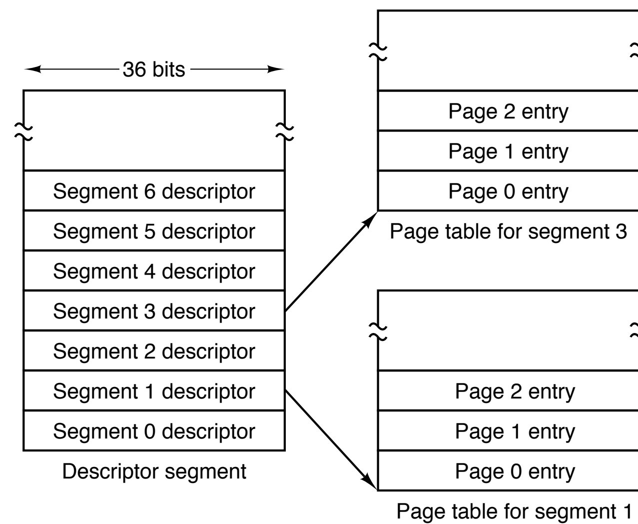
La segmentation permet aussi de faciliter le partage de zones mémoire entre plusieurs processus.

L'implémentation d'un système utilisant la segmentation pure (sans pagination) pose le problème de la fragmentation de la mémoire. La fragmentation est due au fait que les segments (contrairement aux pages) sont de taille non fixe.

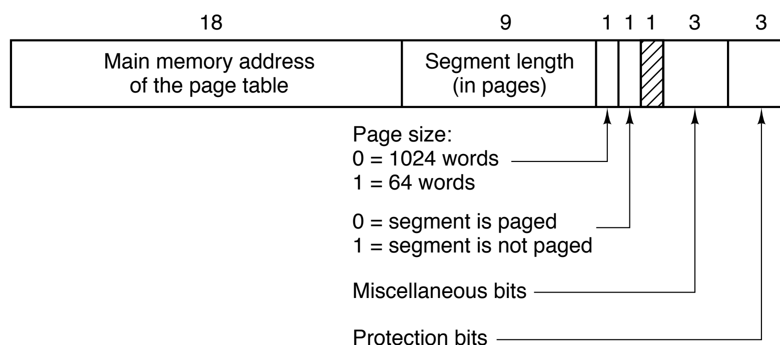
5.5.2 Segmentation avec pagination : MULTICS

L'apparition du système d'exploitation MULTICS dans les années soixante a permis d'introduire divers concepts novateurs encore utilisés dans les systèmes les plus récents. Parmis ces idées, on trouve le principe de la segmentation avec pagination. En combinant segmentation et pagination MULTICS combine les avantages des deux techniques en s'affranchissant des principaux défauts qu'ils ont : fragmentation de la mémoire pour la segmentation, et espace d'adressage unique pour un système utilisant un adressage virtuel paginé.

Sous MULTICS chaque segment possède son propre espace d'adressage paginé. Ainsi chaque segment possède sa propre table de pages (cf. figure 5.9 (a)). Le descripteur de chaque segment contient, en plus de l'adresse de la table de pages qui lui est associée, la taille du segment, la taille des pages (1024 mots ou 64 mots), un indicateur permettant de paginer ou non l'espace d'adressage du segment, des bits de protection, ainsi que d'autres informations (cf. figure 5.9 (b)).



(a)



(b)

FIG. 5.9 – La mémoire virtuelle de MULTICS

Ainsi, pour trouver une case mémoire, MULTICS doit consulter d'abord la table des segments, puis la table des pages, et enfin accéder à la page physique si elle est présente en mémoire (cf. figure 5.11).

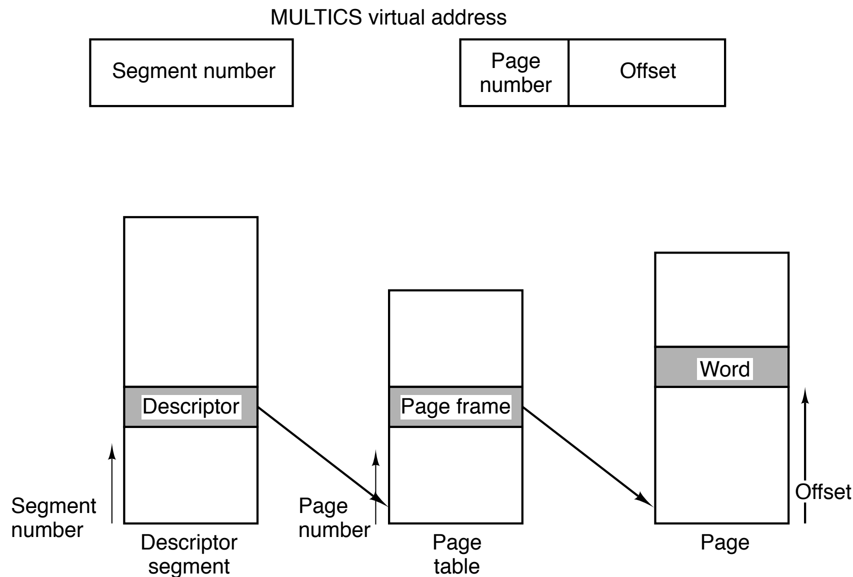


FIG. 5.10 – Conversion d'une adresse MULTICS

Afin d'accélerer le processus, une mémoire associative permet de conserver les numéros des pages physiques correpondants aux pages les plus récemment référencées à partir de leur numéro de segment et de page (cf. figure 5.11).

Comparison field					Is this entry used?
Segment number	Virtual page	Page frame	Protection	Age	
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

FIG. 5.11 – Mémoire associative de MULTICS

5.5.3 Segmentation avec pagination : Le processeur Intel Pentium

Le processeur Intel Pentium reprend le principe de la segmentation avec pagination. La mémoire virtuelle d'un programme permet d'adresser deux tables de descripteurs de segments :

– la table des descripteurs locaux (LDT) : propre à chaque programme elle permet à un programme d'accéder à son code, ses données, sa pile etc. . .
– la table des descripteurs globaux (GDT) : unique, elle permet de décrire des segments du système.

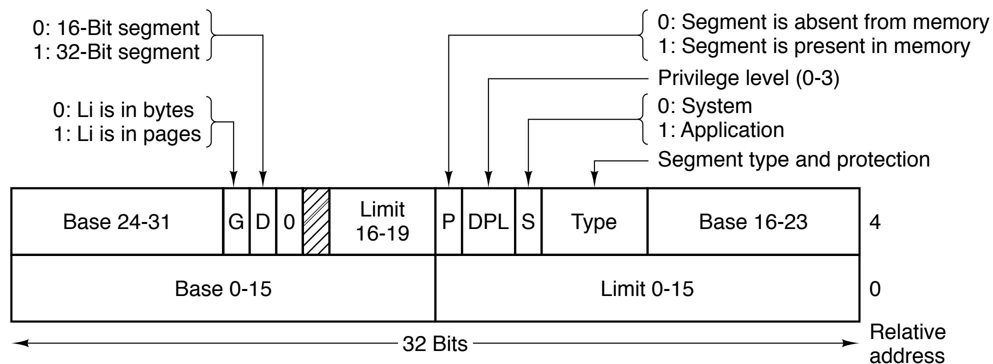


FIG. 5.12 – Descripteur de segment de code du Pentium

Un sélecteur sur 16 bits stocké dans un registre de segment du pentium (CS, DS . . .) contient :
– un index de 13 bits sur une table de descripteurs ;
– un indicateur de 1 bit permettant de déterminer si l'index se réfère à la LDT ou la GDT ;
– un indicateur de niveau de privilège sur 2 bits.

Le sélecteur permet ainsi d'adresser jusqu'a 8K segments dans la LDT et 8K segments dans la GDT.

Le descripteur de segment à une structure assez complexe du au fait qu'il faut assurer la compatibilité ascendante avec les microprocesseurs plus anciens de la famille x86. La figure 5.12 montre le descripteur d'un segment de code. On y trouve donc :
– Base : l'adresse de base du segment dans l'espace d'adressage virtuel, qui est sur 32 bits.
– Limit : la taille du segment, cette taille est en pages si le bit Li est à 1, en octets sinon.
– Le bit P : permettant de déterminer si le segment est présent ou absent de la mémoire physique.
– D'autres informations liées aux attributs de protection et de type du segment.

Les tables de pages du Pentium sont organisées sur deux niveaux : chaque case de la table de pages de niveau 1 dite *page directory*, pointe sur une table de pages de niveau 2 dite *page table* (cf. figure 5.13 (b)).

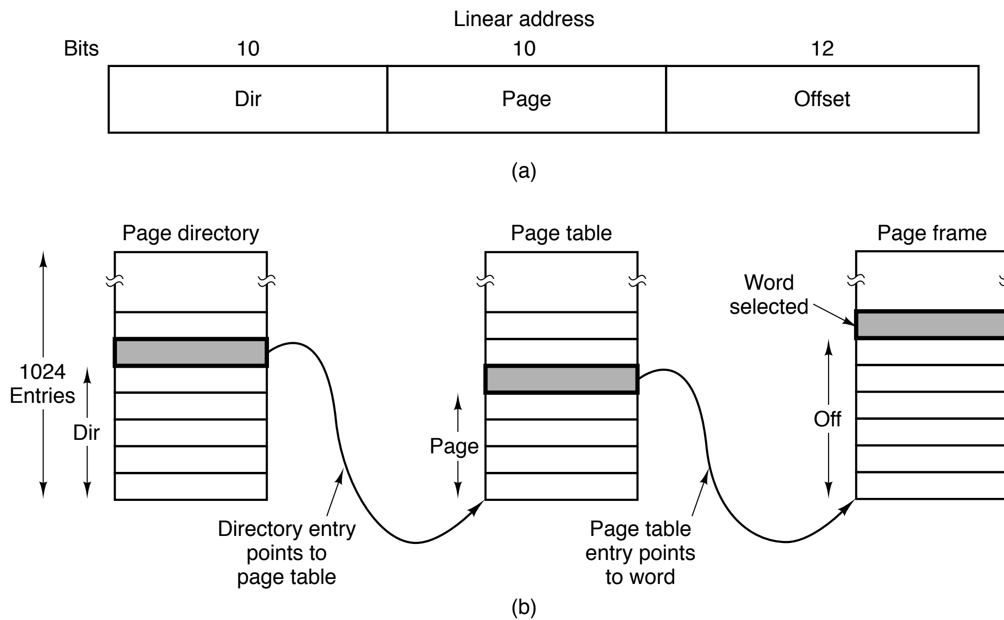


FIG. 5.13 – Tables de pages du Pentium

L'adresse d'un mot mémoire est décrite par un déplacement (offset) dans le segment, et par le sélecteur du segment. A partir du sélecteur, le microprocesseur trouve le descripteur du segment, dans lequel il trouve l'adresse de base du segment. A cette base, le processeur ajoute le déplacement dans le segment et trouve une *adresse linéaire* sur 32 bits (cf. figure 5.13 (a)) qui est composée de trois champs :

- Dir : qui contient sur 10 bits l'index de la *page table* dans *page directory* ;
- Page : qui contient sur 10 bits l'index de la page dans la *page table*.
- Offset : qui contient sur 12 bits le déplacement du mot mémoire dans la page.

5.6 Gestion de la mémoire sous Linux

Comme MULTICS, la gestion de la mémoire sous Linux est basée sur la segmentation avec pagination.

5.6.1 Segmentation

Espace d'adressage d'un processus

L'espace d'adressage d'un processus comprend plusieurs segments appelés régions (*areas*) parmis lesquels figurent typiquement :

- un segment de code ;
- deux segments de données, un concernant les données initialisées (à partir du fichier exécutable), et l'autre les données non initialisées ;

– un segment de pile.

D'autres segments peuvent se trouver dans l'espace d'adressage du processus, ils concernent généralement les segments des librairies partagées qu'utilise le processus.

On peut trouver les informations concernant l'espace d'adressage d'un processus en visualisant le contenu du fichier `maps` du repertoire concernant ce processus dans le système de fichier `/proc`. Dans l'exemple qui suit, on découvre la liste des segments concernant un processus portant le numéro 798 associé à un shell `/bin/tcsh` :

```
$ cat /proc/798/maps
08048000-08083000 r-xp 00000000 03:03 28828      /bin/tcsh
08083000-08087000 rw-p 0003a000 03:03 28828      /bin/tcsh
08087000-080df000 rwxp 00000000 00:00 0
2aaab000-2aabf000 r-xp 00000000 03:03 34954      /lib/ld-2.1.1.so
2aabf000-2aac0000 rw-p 00013000 03:03 34954      /lib/ld-2.1.1.so
2aac0000-2aac1000 r--p 00000000 03:03 437930     /usr/share/locale/fr_FR/LC_MESSAGES/SY
2aac1000-2aac4000 r--p 00000000 03:03 435875     /usr/share/locale/fr_FR/LC_CTYPE
2aac4000-2aac5000 r--p 00000000 03:03 435876     /usr/share/locale/fr_FR/LC_MONETARY
2aac5000-2aac6000 r--p 00000000 03:03 435878     /usr/share/locale/fr_FR/LC_TIME
2aac6000-2aac7000 r--p 00000000 03:03 435877     /usr/share/locale/fr_FR/LC_NUMERIC
2aac7000-2aac8000 r-xp 00000000 03:03 324948     /usr/lib/gconv/ISO8859-1.so
2aac8000-2aac9000 rw-p 00000000 03:03 324948     /usr/lib/gconv/ISO8859-1.so
2aaca000-2aadd000 r-xp 00000000 03:03 34974      /lib/libnsl-2.1.1.so
2aadd000-2aade000 rw-p 00012000 03:03 34974      /lib/libnsl-2.1.1.so
2aade000-2aae2000 rw-p 00000000 00:00 0
2aae2000-2aae5000 r-xp 00000000 03:03 35015      /lib/libtermcap.so.2.0.8
2aae5000-2aae6000 rw-p 00002000 03:03 35015      /lib/libtermcap.so.2.0.8
2aae6000-2aaeb000 r-xp 00000000 03:03 34963      /lib/libcrypt-2.1.1.so
2aaeb000-2aaec000 rw-p 00004000 03:03 34963      /lib/libcrypt-2.1.1.so
2aaec000-2ab13000 rw-p 00000000 00:00 0
2ab13000-2ac05000 r-xp 00000000 03:03 34961      /lib/libc-2.1.1.so
2ac05000-2ac0a000 rw-p 000f1000 03:03 34961      /lib/libc-2.1.1.so
2ac0a000-2ac0d000 rw-p 00000000 00:00 0
2ac0d000-2ac15000 r--p 00000000 03:03 435874     /usr/share/locale/fr_FR/LC_COLLATE
2ac1f000-2ac27000 r-xp 00000000 03:03 34992      /lib/libnss_files-2.1.1.so
2ac27000-2ac28000 rw-p 00007000 03:03 34992      /lib/libnss_files-2.1.1.so
7ffe5000-80000000 rwxp fffe6000 00:00 0
```

Le fichier `maps` permet ainsi de voir les principaux attributs d'un segment :

– L'adresse de début et de fin du segment dans l'espace d'adressage **virtuel**. On voit dans cette implémentation sur processeur x86 que les adresses virtuelles sont sur 32 bits et permettent donc d'adresser quatre giga-octets.
– Les droits d'accès au segment, le caractère `p` indique que le segment peut être partagé. A partir des droits on peut en déduire que le premier segment associé à `/bin/tcsh` est son segment de code, le second son segment de données (initialisé).
– Le déplacement du début de segment dans l'objet qui lui est associé (le fichier exécutable dans le cas de `/bin/tcsh`).
– Le numéro de périphérique contenant l'objet.
– Le numéro d'i-node de l'objet associé au segment (nul s'il n'existe pas).

– Le chemin de l'objet.

Descripteur de segment

Le descripteur associé à un segment est défini dans le fichier `<linux/mm.h>` :

```
struct vm_area_struct {
struct mm_struct * vm_mm; /* VM area parameters */
unsigned long vm_start;
unsigned long vm_end;

/* linked list of VM areas per task, sorted by address */
struct vm_area_struct *vm_next;

pgprot_t vm_page_prot;
unsigned short vm_flags;

/* AVL tree of VM areas per task, sorted by address */
short vm_avl_height;
struct vm_area_struct * vm_avl_left;
struct vm_area_struct * vm_avl_right;

/* For areas with inode, the list inode->i_mmap, for shm areas,
 * the list of attaches, otherwise unused.
 */
struct vm_area_struct *vm_next_share;
struct vm_area_struct **vm_pprev_share;

struct vm_operations_struct * vm_ops;
unsigned long vm_offset;
struct file * vm_file;
unsigned long vm_pte; /* shared mem */
};
```

On y retrouve notamment des champs correspondant à des attributs vus précedement :
- adresse de début et de fin du segment (`vm_start` et `vm_end`) ;
- les droits d'accés (drapeau `vm_flags`) ;
- le déplacement du début de segment dans le fichier associé (`vm_offset`) ;
- le fichier associé (`vm_file`).

On y trouve aussi plusieurs pointeurs sur des descripteurs de segment correspondant aux structures de données suivantes :
- une liste des segments associés au processus (pointeur `vm_next`) ;
- une liste des segments partagés (pointeurs `vm_next_share` et `vm_pprev_share`) ;
- un arbre AVL des segments du processus (pointeurs `vm_avl_left` et `vm_avl_right`).

Organisation des descripteurs de segment

Ainsi les descripteurs de segment associés à un même processus sont référencés par l'espace d'adressage de deux façons differentes : par une liste chaînée d'une part et par un arbre AVL d'autre

part.

Quand le noyau recherche le descripteur d'un segment particulier il n'utilise pas la liste mais l'arbre AVL, ce qui permet de réduire la complexité de la recherche de $O(n)$ à $O(\log n)$, n étant le nombre de segments adressés par le processus.

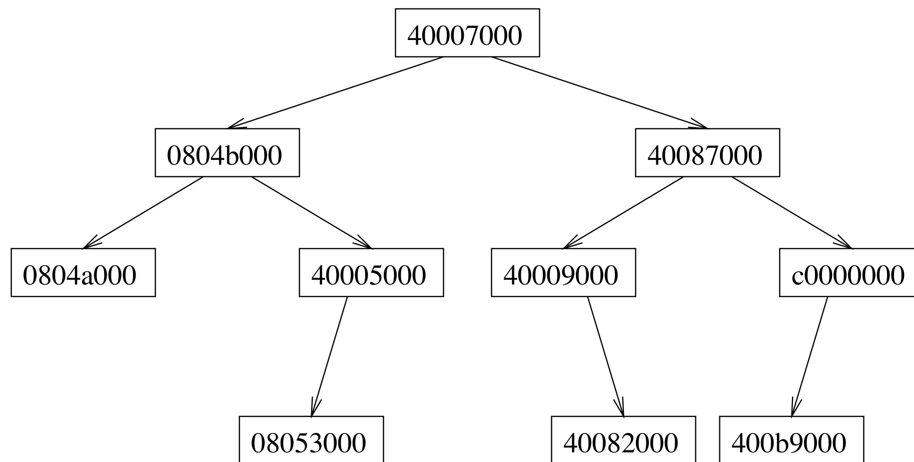


FIG. 5.14 – Arbre AVL

Un arbre AVL (Adelson-Velskii et Landis) est **un arbre de recherche équilibré**. Dans cet arbre chaque noeud associé à un descripteur de segment a pour clé l'adresse de fin du segment dans l'espace d'adressage virtuel. Comme pour tout arbre de recherche les clés du sous arbre gauche d'un noeud x valent au plus la clé de x, et celles du sous-arbre droit valent au moins la clé de x. Le fait que l'arbre soit équilibré lui assure une profondeur qui croît en $O(\log n)$. La figure 5.14 montre l'arbre AVL correspondant à un processus dans l'espace d'adressage. Ce processus contient les segments suivants :

```
08048000-0804a000 r-xp 00000000 03:02 7914
0804a000-0804b000 rw-p 00001000 03:02 7914
0804b000-08053000 rwxp 00000000 00:00 0
40000000-40005000 r-xp 00000000 03:02 18336
40006000-40007000 rw-p 00000000 00:00 0
40007000-40009000 r--p 00000000 03:02 18255
40009000-40082000 r-xp 00000000 03:02 18060
40082000-40087000 rw-p 00078000 03:02 18060
40087000-400b9000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp ffff0000 00:00 0
```

A chaque nœud de l'arbre est associé un poids (champs `vm_avl_height` dans le descripteur de segment), qui correspond à la différence entre les profondeurs de son sous-arbre gauche et de son sous-arbre droit. Tant que l'arbre est équilibré ce poids doit toujours avoir comme valeur -1, 0, ou 1. Si ce poids est modifié en dehors de ces limites, après ajout ou suppression d'un segment, des opérations de rotation sont effectuées afin de rééquilibrer l'arbre (cf. figure 5.15).

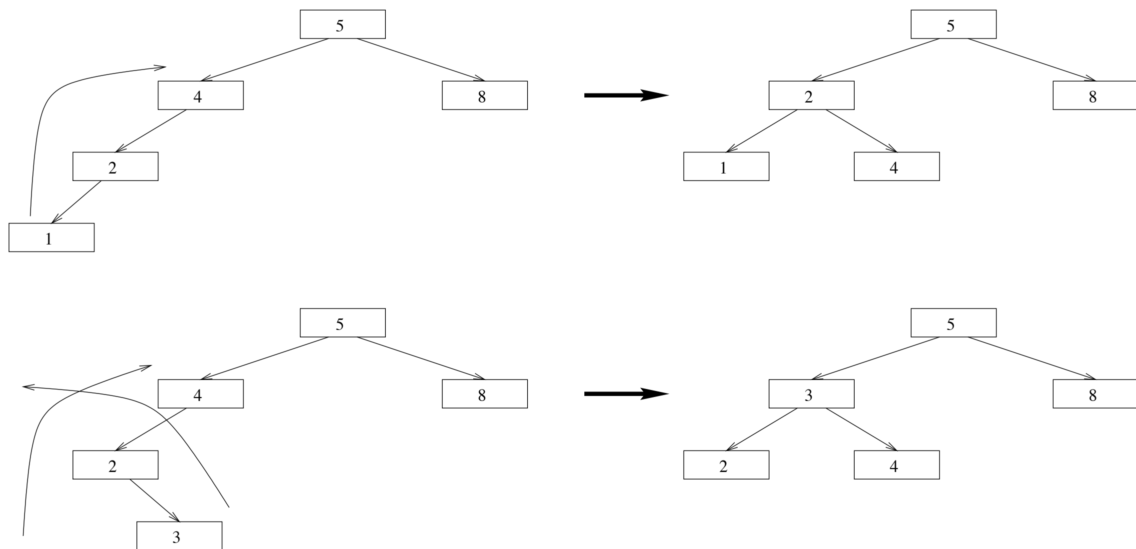


FIG. 5.15 – Exemples de rotations dans un arbre AVL

5.6.2 Gestion des pages et va-et-vient

Gestion des tables de pages

Les tables de pages gérées par Linux sont organisées en trois niveaux :
- la table globale dont chaque entrée pointe sur une table intermédiaire ;
- les tables intermédiaires dont chaque entrée pointe sur une table de page ;
- les tables de pages dont les entrées contiennent les adresses de pages physiques.

Selon la plateforme sur laquelle Linux est implanté, le nombre de niveaux effectifs peut être plus réduit. Sur un processeur de type x86, par exemple, étant donné que les pages sont organisées en deux niveaux seulement (cf. section 5.5.3), le noyau Linux considère que la table intermédiaire ne contient qu'une seule entrée.

Descripteur de page

Le descripteur associée à une page est définie dans le fichier `<linux/mm.h>` :

```
typedef struct page {
struct page *next;
struct page *prev;
struct inode *inode;
unsigned long offset;
struct page *next_hash;
atomic_t count;
unsigned long flags;
struct wait_queue *wait;
struct page **pprev_hash;
```

```
struct buffer_head * buffers;
} mem_map_t;
```

Lorqu'une page est libre elle est chainée avec les autres pages libres (pointeurs `next` et `prev` de la structure `page`).

Le cache des pages

Linux maintient les pages chargées en mémoire dans un cache. Dans ce cache les pages mémoire associées à un même i-node (pointeur `inode` de la structure) sont chainées dans une liste de hachage (pointeurs `next_hash` et `pprev_hash`). Lorsqu'une page associée à un i-node est chargée en mémoire, elle est insérée dans le cache (dans la liste de hashage correspondante). Lors d'un prochain accès à cette page, son chargement ne sera plus nécessaire tant qu'elle restera dans le cache.

Remplacement de pages

Lorsque Linux manque de mémoire, il évince des pages mémoire. Pour ce faire le processus `kswapd` (endormi la plupart du temps) est réveillé. Ce processus explore la liste des processus et essaye d'évincer des pages. Si une page est réservée ou verrouillée, ou si elle a été récemment accédée, elle n'est pas évincée. Dans le cas contraire, son état est testé. Si la page a été modifiée, elle doit être sauvegardée sur disque avant d'être libérée (retirée de la mémoire donc du cache).

Ce mécanisme est utilisé également par Linux (depuis la vesion 2.0) pour les lectures de fichiers. Lors d'un appel à `read` la lecture est effectuée en chargeant en mémoire les pages correspondantes de l'i-node. Ceci permet d'utiliser le cache de pages pour optimiser l'accès aux fichiers. Ceci n'est réalisé que pour les lectures, les écritures sur fichiers. Les manipulations de répertoires s'effectuent normalement dans le cache du système de fichiers.

5.6.3 Allocation de mémoire pour le noyau

Allocation de pages mémoire

Le noyau maintient les listes d'*ensembles de pages* libres en mémoire. Ces ensembles sont de taille fixe, ils peuvent contenir 1, 2, 4, 8, 16, ou 32 pages, et se réferent à des pages contigües en mémoire. Il y a une liste pour chaque ensemble de pages de même taille. Le noyau ne permet d'allouer que les ensembles de pages de taille prédéterminée correspondant aux tailles gérées dans les listes.

Les listes d'ensembles de pages sont gérées selon le principe du *Buddy system* : à chaque demande d'allocation, un ensemble de pages de la liste non vide contenant les ensembles de pages de taille immédiatement supérieure à la taille demandée est choisi. L'ensemble de pages choisi est alors décomposé en deux parties : les pages correspondant à la taille mémoire spécifiée, et le reliquat de pages qui restent disponibles. Le reliquat peut être inséré dans les autres listes. Par exemple, si 8 pages doivent être allouées et que seul un ensemble de 32 pages est disponible, le noyau utilise cet ensemble, alloue les 8 pages demandées, et répartit les 24 pages restantes en un

ensemble de 16 pages et un autre de 8 pages. Ces deux derniers ensembles sont insérés dans les listes d'ensembles correspondant à leur taille.

Lors d'une désallocation d'un ensemble de pages, le système essaie de fusionner cet ensemble avec avec les ensembles de pages libres contigues, afin d'obtenir un nouvel ensemble de taille maximale. Ainsi lors de la désallocation d'un ensemble de 8 pages, le système teste si les 8 pages adjacentes sont libres, si ce n'est pas le cas les 8 pages sont libérées et insérées dans la liste des pages libres correspondante. Si l'ensemble des 8 pages libres existe, sa taille est modifiée pour inclure les 8 pages nouvellement libérées. Le système fait alors la même chose en testant s'il peut fusionner avec un autre ensemble de 16 pages, et ainsi de suite.

5.6.4 Appels systèmes Unix

Changement de taille du segment de données

```
#include <unistd.h>

int brk(void *end_data_segment);
void *sbrk(ptrdiff_t increment);
```

`brk` positionne la fin du segment de données (le premier mot mémoire hors de la zone accessible) à l'adresse spécifiée par `end_data_segment`. `end_data_segment` doit être supérieur à la fin du segment de texte (code), et doit être 16 Ko avant la fin de la pile.

`sbrk` incrémente l'espace de données du programme de `increment` octets.

Allocation et désallocation de mémoire

Les fonctions de la bibliothèque standard du langage C :

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

utilisent de manière interne les fonctions `brk` et `sbrk` pour allouer et désalouer des zones mémoire.

Protection de pages mémoires

La fonction :

```
#include <sys/mman.h>

int mprotect(const void *addr, size_t len, int prot);
```

permet de contrôler la manière d'accéder à une portion de la mémoire.
`prot` est un OU binaire (|) entre les valeurs suivantes :
- `PROT_NONE` : On ne peut pas accéder du tout à la zone de mémoire.
- `PROT_READ` : On peut lire la zone de mémoire.
- `PROT_WRITE` : On peut écrire dans la zone de mémoire.
- `PROT_EXEC` : La zone de mémoire peut contenir du code exécutable.

La nouvelle protection remplace toute autre protection précédente.

Verrouillage de pages mémoires

```
#include <sys/mman.h>

int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
int mlockall(int flags);
int munlockall(void);
```

`mlock` désactive la pagination pour la portion de mémoire débutant à l'adresse `addr` avec une longueur de `len` octets. Toutes les pages contenant une partie de la zone mémoire spécifiée résident en mémoire principale et y resteront jusqu'à un déverrouillage par la fonction `munlock` ou `munlockall`, ou encore jusqu'à ce que le processus se termine ou se recouvre avec `exec`.

`munlock` revalide la pagination pour la zone de mémoire commençant à l'adresse addr et s'étendant sur len octets.

`mlockall` désactive la pagination pour toutes les pages représentées dans l'espace d'adressage du processus appelant. Ceci inclut les pages de code, de données, et le segment de pile, tout autant que les bibliothèques partagées, l'espace utilisateur dans le noyau, la mémoire partagée et les fichiers projetés en mémoire. Le paramètre `flags` détermine si les pages dont il faut désactiver la pagination sont celles qui sont actuellement présentes en mémoire physique (`MCL_CURRENT`), celles qui le seront dans le futur (`MCL_FUTURE`), ou les deux à la fois.

`munlockall` revalide la pagination pour toutes les pages de l'espace d'adressage du processus en cours.

Il y a deux domaines principaux d'application au verrouillage de pages : les algorithmes en temps réel, et le traitement de données confidentielles. Les applications temps réel réclament un comportement temporel déterministe, et la pagination est, avec l'ordonnancement, une cause majeure de délais imprévus.

Les logiciels de cryptographie manipulent souvent quelques octets hautement confidentiels, comme des mots de passe ou des clés privées. A cause de la pagination ces données secrètes risquent d'être transférées sur un support physique où elles pourraient être lues longtemps après que le logiciel se soit terminé.

Projection en mémoire

La projection en mémoire est un mécanisme hérité de MULTICS, qui permet de se dispenser des entrées/sorties et facilite donc la programmation. Sous Unix les appels suivant permettent de

le réaliser :

```
#include <unistd.h>
#include <sys/mman.h>

void  * mmap(void  *start,  size_t length, int prot,
              int flags, int fd, off_t offset);
int munmap(void *start, size_t length);
```

La fonction `mmap` demande la projection en mémoire de `length` octets commencant à la position `offset` depuis un fichier (ou un autre objet) indiqué par `fd`, de préférence à l'adresse pointée par `start`. Cette adresse n'est qu'une préférence, généralement `0`. La véritable adresse où l'objet est projeté est renvoyée par la fonction `mmap`.

L'appel `munmap` détruit la projection dans la zone de mémoire spécifiée, et s'arrange pour que toute référence ultérieure à cette zone mémoire déclenche une erreur d'adressage.

L'argument `prot` indique la protection que l'on désire pour cette zone de mémoire (en utilisant `PROT_NONE`,`PROT_READ`, `PROT_WRITE`, `PROT_EXEC` comme pour `mprotect`).

Le paramètre `flags` indique le type de fichier projeté, les options de projection, et si les modifications faites sur la portion projetée sont privées ou doivent être partagées avec les autres références. Les options sont :

- `MAP_SHARED` : partager la projection avec tout autre processus utilisant l'objet,
- `MAP_PRIVATE` : créer une projection privée, utilisant la méthode de copie à l'écriture,
- `MAP_FIXED` : n'utiliser que l'adresse indiquée. Si c'est impossible, `mmap` échouera. Si `MAP_FIXED` est spécifié, `start` doit être un multiple de la longueur de page. Il est déconseillé d'utiliser cette option.

Ces trois attributs sont décrits dans POSIX.1b (anciennement POSIX.4). Linux propose également les options `MAP_DENYWRITE`, `MAP_EXECUTABLE` et `MAP_ANON`(`YMOUS`).

Synchronisation de pages mémoire

La fonction :

```
#include <unistd.h>
#include <sys/mman.h>

int msync(const void *start, size_t length, int flags);
```

permet d'écrire sur le disque les modifications qui ont été effectuées sur la copie d'un fichier qui est projeté en mémoire par `mmap`. La portion du fichier correspondant à la zone mémoire commençant en `start` et ayant une longueur de `length` est mise a jour.

L'argument flags comprend les bits suivants :

- `MS_SYNC` : indique que l'appel à `msync` est bloquant : attend qu'elle se termine avant de revenir.
- `MS_ASYNC` : indique que l'appel n'est pas bloquant.

– `MS_INVALIDATE` demande la désactivation de toutes les autres projections du même fichier, afin qu'elles soient toutes remises à jour avec les nouvelles données écrites.