Chapitre 9

Gestion de la mémoire

La mémoire principale est le lieu où se trouvent les programmes et les données quand le processeur les exécute. On l'oppose au concept de mémoire secondaire, représentée par les disques, de plus grande capacité, où les processus peuvent séjourner avant d'être exécutés. La nécessité de gérer la mémoire de manière optimale est toujours fondamentale, car en dépit de sa grande disponibilité, elle n'est, en général, jamais suffisante. Ceci en raison de la taille continuellement grandissante des programmes. Nous verrons qu'il y a plusieurs schémas pour la gestion de la mémoire, avec leurs avantages et inconvénients.

9.1 Introduction

9.1.1 Rappels sur la compilation

La production d'un programme passe par plusieurs phases, comme illustré à la figure 9.1 :

- Écriture en langage source (C/C++, Pascal,...).
- Compilation en module objet (langage machine).
  - Peuvent inclure des bibliothèques.
  - Appel aux bibliothèques : point de branchement.
- Assemblage en image binaire.
  - Édition de liens statiques.
  - Édition de liens dynamiques.
- Chargement en mémoire pour exécution.

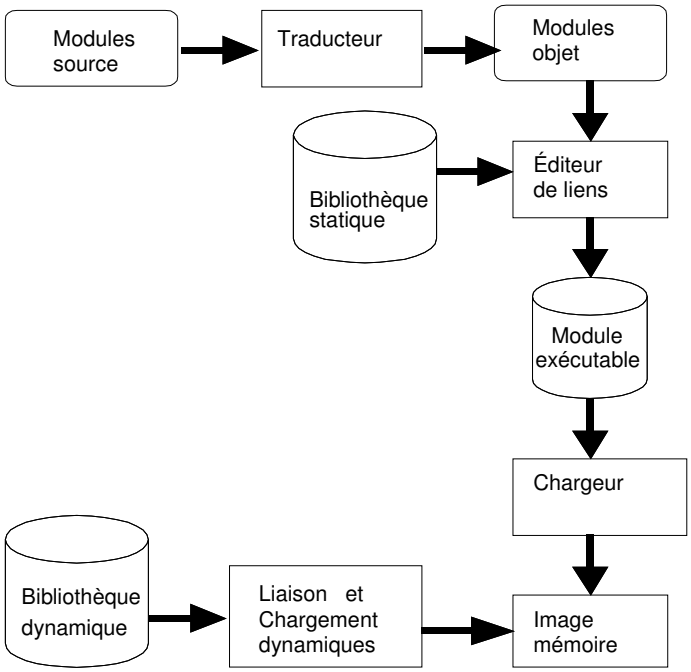


FIG. 9.1 – Phases de la compilation.

Représentation des adresses mémoire

- Code source : **adresses symboliques**. Par exemple : `int compteur`
- Module objet : **adresses traduites**. Par exemple le 50ème mot depuis le début d'espace mémoire.
- Module exécutable, chargement : **adresses absolues**. Par exemple l'emplacement mémoire situé à l'adresse FFF7

Liaison d'adresses

Le programme réside sur disque comme *Fichier exécutable*. Il est ensuite chargé en mémoire pour être exécuté. Il peut résider, le plus souvent, dans une partie quelconque de la mémoire. On les appelle *Programmes translatables*[1]. Les programmes `.com` de MS/DOS constituent cependant une exception car la compilation est statique.

[1] **Translation** sera synonime de **traduction** dans notre contexte, et pourront être utilisés indistinctement.

Liaison des instructions à des adresses mémoire

- Pendant la compilation : Si l'emplacement du processus en mémoire est connu, le compilateur génère des adresses absolues.
- Pendant le chargement : Code translatable si l'emplacement du processus en mémoire n'est pas connu.
- Pendant l'exécution : déplacement dynamique du processus possible. Utilise des fonctions du matériel.

Espace d'adressage logique ou physique

L'unité centrale manipule des **adresses logiques** (emplacement relatif). Les programmes ne connaissent que des adresses logiques, ou virtuelles. L'*espace d'adressage logique* (virtuel) est donc un ensemble d'adresses pouvant être générées par un programme.

L'unité mémoire manipule des **adresses physiques** (emplacement mémoire). Elles ne sont jamais vues par les programmes utilisateurs. L'*espace d'adressage physique* est un ensemble d'adresses physiques correspondant à un espace d'adresses logiques.

9.1.2 La gestion de la mémoire

La mémoire centrale peut avoir une représentation Linéaire et uniforme ou Différenciée :

- **Linéaire et uniforme** : Mémoire non différenciée, avec un espace d'adressage linéaire ou plat (*One level flat address space*).
- **Différenciée** : Avec des **segments linéaires spécifiques et disjoints** pour les procédures, la table de symboles, les programmes, etc. ou avec un **adressage segmenté** à l'aide de deux index, un qui contrôle le numéro de segment et un autre qui contrôle l'adresse dans le segment.

La gestion de la mémoire a deux objectifs principaux : d'abord le partage de mémoire physique entre les programmes et les données des processus prêts, et ensuite la mise en place des paramètres de calcul d'adresses, permettant de transformer une adresse virtuelle en adresse physique. Pour ce faire le gestionnaire de la mémoire doit remplir plusieurs tâches :

- Connaître l'état de la mémoire (les parties libres et occupées de la mémoire).
- Allouer de la mémoire à un processus avant son exécution.
- Récupérer l'espace alloué à un processus lorsque celui-ci se termine.

– Traiter le va-et-vient (*swapping*) entre le disque et la mémoire principale lorsque cette dernière ne peut pas contenir tous les processus.
– Posséder un mécanisme de calcul de l'adresse physique (absolue) car, en général, les adresses générées par les compilateurs et les éditeurs de liens sont des adresses relatives ou virtuelles.

En général, pour ne pas ralentir l'accès à la mémoire le calcul des adresses est réalisé par le matériel. Comme nous allons le voir, les techniques de gestion de la mémoire sont très conditionnées par les caractéristiques du système (monoprogrammé ou multiprogrammé) et du matériel (registres de base et limite, MMU). Nous passerons en revue un certain nombre de méthodes de gestion de la mémoire.

9.2 Système monoprogrammé

9.2.1 Machine nue

Il s'agit d'un des schémas de gestion de mémoire les plus simples. L'utilisateur a tout le contrôle sur l'espace de mémoire, comme le montre la figure 9.2. Cette gestion a comme avantages la flexibilité : l'utilisateur peut contrôler —à sa guise— l'usage de la mémoire et un coût minime, car il n'y a pas besoin de matériel spécifique pour l'implanter. Parmi ses limitations, on peut citer l'inexistance des services d'entrées-sorties, des périphériques externes et de gestion d'erreurs. Ce schéma est utilisé en systèmes dédiés à des tâches spécifiques d'une grande simplicité.

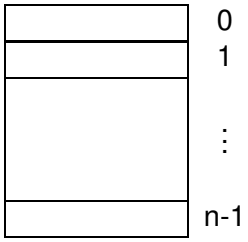


FIG. 9.2 – Machine nue.

9.2.2 Moniteur résident

Le schéma suivant est celui qui divise la mémoire en deux sections, une pour les programmes des utilisateurs et une autre pour le **moniteur résident du système**. Le moniteur résident se retrouve typiquement dans les

premières adresses du système, comme illustré à la figure 9.3. Les composantes principales d'un système d'exploitation présentes dans la section du moniteur sont le **vecteur d'interruptions** qui donne accès aux services d'E/S et les **tables PCB** (Bloc de Contrôle de Processus).

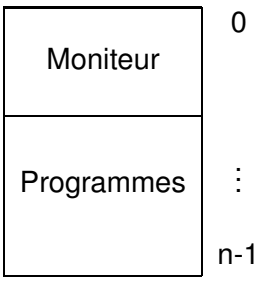


FIG. 9.3 – Moniteur résident.

Un exemple de tel type de moniteur est celui de MS-DOS. Un seul processus utilisateur est en mémoire à un instant donné (cette technique n'est plus utilisée de nos jours). On trouve en mémoire, par exemple, le système d'exploitation en mémoire basse, les pilotes de périphériques en mémoire haute (dans une zone allant de 640 Ko à 1 Mo) et un programme d'utilisateur entre les deux (voir figure 9.4).

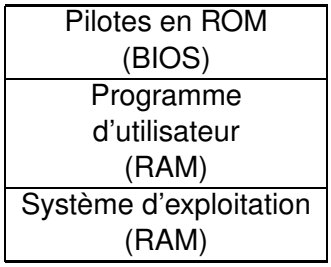


FIG. 9.4 – Organisation de la mémoire du PC sous MS-DOS.

Lorsque la mémoire est organisée de cette manière, il ne peut y avoir qu'un seul processus qui s'exécute à un instant donné. L'utilisateur entre une commande sur un terminal et le système d'exploitation charge le programme demandé en mémoire puis l'exécute. Lorsqu'il se termine, le système d'exploitation affiche une invitation sur le terminal et attend la commande suivante pour charger un nouveau processus qui remplace le précédent.

Pour protéger le code et les données du système d'exploitation des changements que peuvent provoquer les programmes des utilisateurs, il

y a un **registre limite** (montré à la figure 9.5) qui contient l'adresse à partir de laquelle commencent les instructions et données des programmes des utilisateurs. Chaque adresse (instruction ou donnée) générée par un programme est comparée avec le registre limite. Si l'adresse générée est supérieure ou égale à la valeur du registre, on l'accepte, sinon on refuse l'accès et on interrompt le processus avec un message d'accès illégal de mémoire.

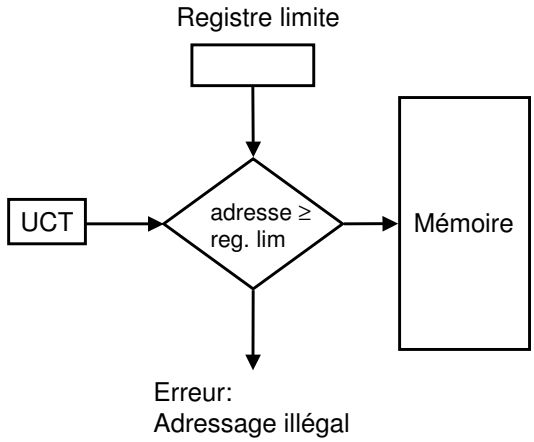


FIG. 9.5 – Registre limite.

Toute référence à la mémoire doit être vérifiée, ce qui implique un ralentissement du système.

Il y a plusieurs stratégies pour implanter le **registre limite**. L'une d'entre elles consiste à utiliser une valeur fixe, enregistrée au niveau matériel ou au début du démarrage du système. Cependant si la valeur choisie est trop petite et que le système d'exploitation grandisse, il se retrouvera dans une zone non protégée, ou au contraire, l'espace pour les programmes serait diminué. Une solution est l'existence d'un registre variable pour permettre d'initialiser différentes valeurs d'adresses au registre limite. Celui-ci peut être chargé par le système d'exploitation qui utilise des privilèges spéciaux, c'est à dire, on exécute le système d'exploitation en **mode moniteur**.

9.3 Système multiprogrammé

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 néces-

site 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 cas sont alors à distinguer :

Multiprogrammation sans va-et-vient. Dans ce cas, un processus chargé en mémoire y séjournera jusqu'à ce qu'il se termine (pas de va-et-vient entre la mémoire et le disque).

Multiprogrammation avec va-et-vient. Dans ce cas, un processus peut être déplacé temporairement sur le disque (mémoire de réserve : *swap area* ou *backing store*) pour permettre le chargement et donc l'exécution d'autres processus. Le processus déplacé sur le disque sera ultérieurement rechargé en mémoire pour lui permettre de poursuivre son exécution.

Ce schéma pose d'autres difficultés : étant donné que les processus sont chargés à différentes adresses, les adresses relatives générées par les compilateurs et les éditeurs de liens sont différentes des adresses physiques, alors comment réaliser la translation d'adresses ? D'un autre coté, la cohabitation de plusieurs processus en mémoire doit assurer leur protection : il ne faut pas permettre à un processus d'un utilisateur d'adresser la mémoire des processus des autres utilisateurs. Quelle est la meilleur manière d'assurer la protection des processus contre les intrusions des autres ?

9.3.1 Translation des adresses lors du chargement et protection

Utiliser un registre limite a des conséquences. D'abord les programmes doivent posséder un mécanisme pour leur permettre de déterminer si un changement dans la valeur du registre limite sera ou non illégal. Pour éviter cela, la solution triviale consiste à recompiler le programme. Mais ceci n'est pas optimal.

La solution optimale est la génération de code relocalisable, c'est à dire un code où les instructions et les données vont se lier à leurs adresses définitives en mémoire en *temps de chargement* et pas en temps de compilation. Cependant, le registre limite ne peut pas être modifié pendant l'exécution d'un programme : le système d'exploitation doit attendre qu'aucun programme ne s'exécute avant de le modifier. Il existe alors deux manières pour que le système d'exploitation puisse changer la taille du moniteur : la première où les programmes des utilisateurs sont chargés depuis de hautes positions de mémoire vers le moniteur (voir la figure 9.6). La deuxième

consiste à effectuer les liens des adresses dans le temps d'exécution, et ajouter à chaque adresse générée par le programme la valeur du registre limite. De cette façon les programmes vont manipuler des adresses logiques et le matériel de gestion de mémoire les convertira en adresses physiques, comme montré à la figure 9.7.

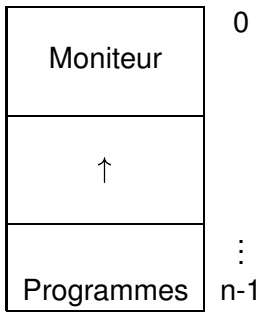


FIG. 9.6 – Chargement des programmes depuis les hautes positions.

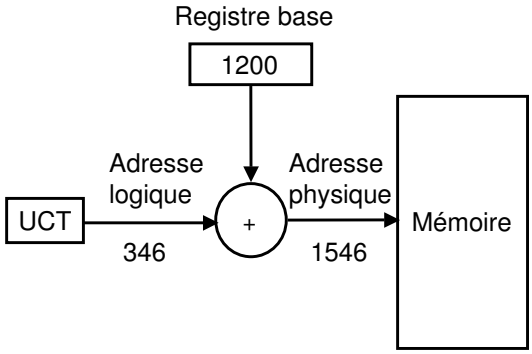


FIG. 9.7 – Registres limite.

9.4 Swapping

Le schéma de moniteur résident a été utilisé depuis l'introduction des premiers systèmes d'exploitation multiprogrammés. On utilise une partie de la mémoire pour le moniteur résident et le reste pour les programmes des utilisateurs. Quand le système d'exploitation rend le contrôle au programme suivant, le contenu de la mémoire est copié à un stockage temporaire, et les données et les instructions d'un nouveau programme sont récu-

pérés depuis le stockage secondaire vers la mémoire principale. Ce schéma est connu sous le nom de *swapping*.

Le stockage temporaire peut être un disque rapide, ayant l'espace nécessaire pour contenir les images mémoire de tous les programmes en exécution (résidents dans la file de prêts). Évidemment le temps de *swapping* est trop grand comparé au temps de l'UCT, c'est pourquoi il est préférable que chaque processus qui obtient le contrôle de l'UCT s'exécute pour un temps supérieur au temps de *swap* (voir figure 9.8).

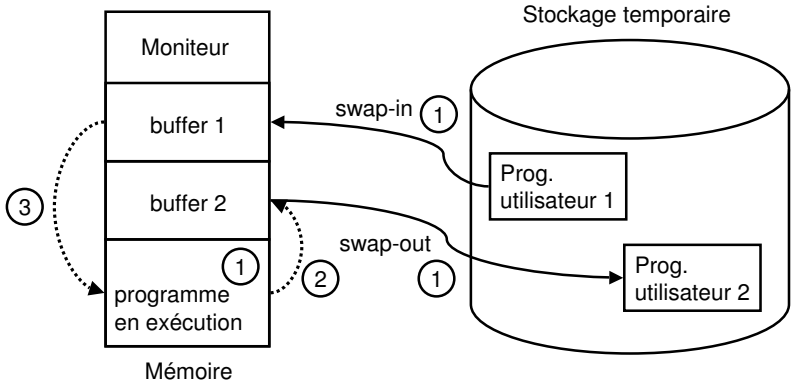


FIG. 9.8 – (1) Tant que le programme s'exécute, le système d'exploitation fait un swapping de 2 processus, l'un qui sort (swap out) et l'autre qui rentre (swap-in). (2) Quand le processus en exécution se termine, ses données et instructions passent au buffer 2 pour ensuite faire swap-out de ce processus. (3) Depuis le buffer 1 on extrait les données et les instructions d'un nouveau processus à s'exécuter.

9.5 Partitions fixes sans va-et-vient

Dans ce schéma, plusieurs programmes peuvent partager la mémoire. Pour chaque utilisateur il existe une partition. La méthode la plus simple consiste à diviser la mémoire en n partitions qui peuvent être de tailles égales ou inégales. On appelle ceci *Mutiprogramming with a Fixed Number of Tasks* (**MFT**). [2]

[2]Ce partitionnement peut être fait par l'opérateur au démarrage du système, comme c'est le cas du OS/MFT 360 de IBM.

Avec **MFT**, le nombre et la taille des partitions sont fixés à l'avance. Par exemple, avec une mémoire de 32 Ko, et un moniteur utilisant 10 Ko, le reste de 22 Ko peut être divisé en 3 partitions fixes de 4Ko, 6 Ko et 12 Ko. La décision sur le nombre et la taille des partitions dépend du niveau de multiprogrammation voulu. En tout cas, le programme avec le plus grand besoin de mémoire doit avoir au moins, une partition pour y mettre ses données et instructions.

MFT doit implanter des mécanismes de protection pour permettre à plusieurs programmes de s'exécuter sans envahir l'espace d'adresses d'autre programme. Pour cela il existe deux registres qui indiquent la plage des adresses disponibles pour chaque programme. Les deux interprétations courantes sont les **registres limite** et les **registres base-limite**.

9.5.1 Registres limite

Dans ce cas, les valeurs des registres indiquent la valeur de l'adresse de mémoire la plus petite et la plus grande qu'un programme puisse atteindre. Il y a une paire de valeurs de ces **registres limite** pour chaque programme en exécution. Chaque accès en mémoire entraîne deux vérifications : la première pour ne pas aller en dessous de la limite inférieure, et la seconde pour ne pas aller au dessus de la limite supérieure de mémoire, comme illustré sur les figures 9.9 et 9.10.

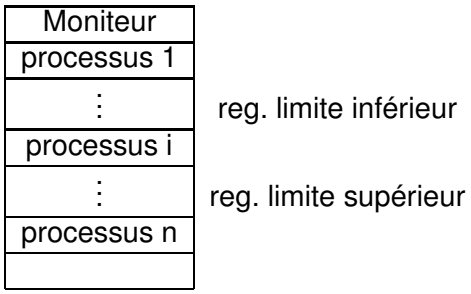


FIG. 9.9 – Registres limite et mémoire.

9.5.2 Registres de base et limite

Pour éviter les problèmes de translation d'adresse au cours du chargement d'un processus et de protection, une autre solution consiste à doter la machine de deux registres matériels spéciaux : le **registre de base** et le **registre limite**, comme montré sur la figure 9.11.

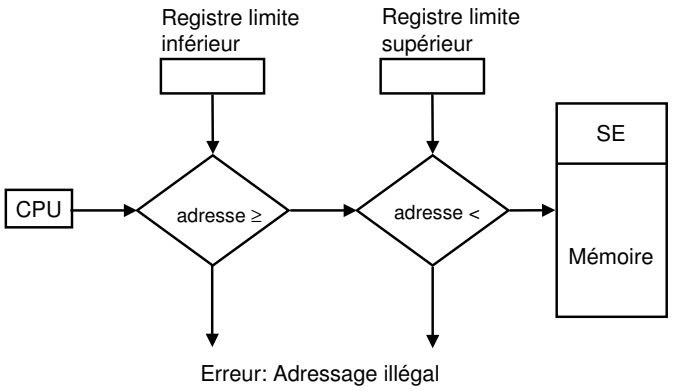


FIG. 9.10 – Registres limite.

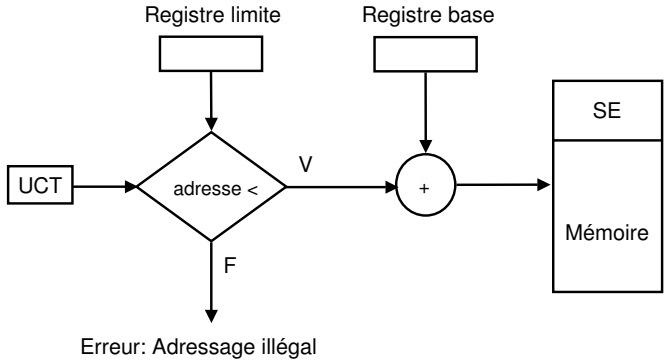


FIG. 9.11 – Registres de base et limite.

Quand on lance un processus, on charge dans le registre de base la limite inférieure de la partition qui lui est attribuée (adresse absolue du début de la partition) et dans le registre limite la longueur de la partition. Pendant l'exécution d'une instruction, les adresses relatives référencées par l'instruction sont translatées en adresses physiques en ajoutant le contenu du registre de base à chaque adresse mémoire référencée, on vérifie ensuite si les adresses ne dépassent pas le registre limite afin d'interdire tout accès en dehors de la partition courante[3].

Un important avantage du registre de base est que les programmes peuvent être déplacés en mémoire *après* le début de leur exécution. Il suffit

[3]Le PC sous MS-DOS utilise une version simplifiée de ce mécanisme. Voir la note à la fin de ce chapitre.

pour cela de changer la valeur du registre de base, et éventuellement celle du registre limite.

9.5.3 Fragmentation

Avec le schéma **MFT**, on fait face au problème de **fragmentation**, puisqu'on génère deux types d'espaces de mémoire non utilisés : la partie d'une partition non utilisée par un processus est nommée **fragmentation interne**. Ceci est dû au fait qu'il est très difficile que tous les processus rentrent exactement dans la taille de la partition. Les partitions qui ne sont pas utilisées, engendrent la **fragmentation externe**. Le phénomène de fragmentation est montré sur la figure 9.12.

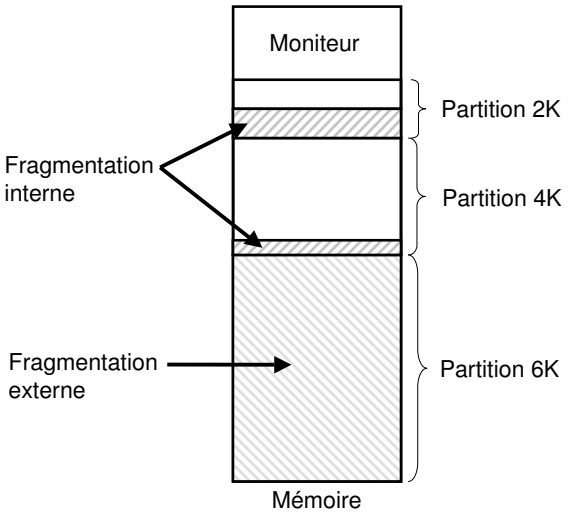


FIG. 9.12 – Fragmentation interne et externe.

9.5.4 Files d'entrée

Un autre aspect important est celui de la gestion des requêtes de la mémoire. Il peut y avoir une **file unique** qui stocke toutes les requêtes pour toutes les partitions ou bien une file pour chaque partition. La figure 9.13 à gauche montre une configuration à **files d'entrée multiples**, et à droite, une file unique pour les requêtes.

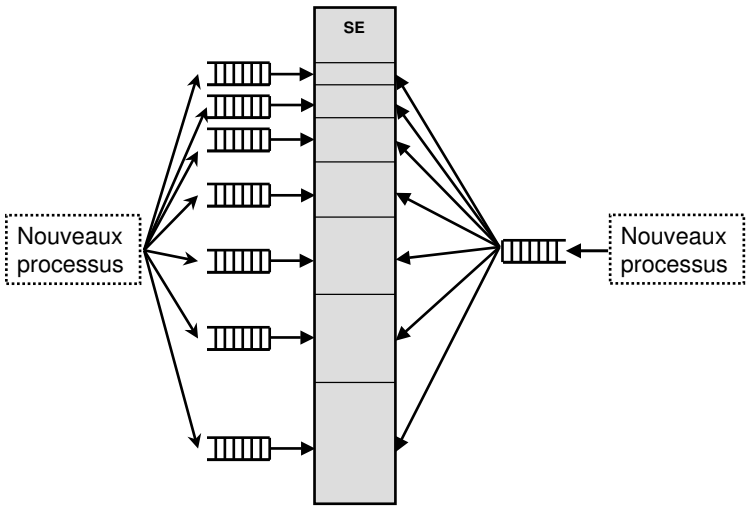


FIG. 9.13 – File d'entrée multiple à gauche, et file unique à droite.

Files d'entrée multiples

Chaque nouveau processus est placé dans la file d'attente de la plus petite partition qui peut le contenir. L'espace inutilisé dans une partition allouée à un processus est perdu à cause de la fragmentation interne. La situation suivante peut se produire : des processus en attente alors qu'il y a des partitions vides. La solution à ce problème consiste en l'utilisation d'une seule file d'attente.

File d'entrée unique

Les processus sont placés dans une seule file d'attente jusqu'à ce qu'une partition se libère. Lorsqu'une partition se libère, il faut choisir dans la file d'entrée le prochain processus à charger dans la partition. Le processus chargé résidera en mémoire jusqu'à ce qu'il se termine (pas de va-et-vient). Il y a cependant des inconvénients comme la fragmentation interne et le fait que le nombre de processus en exécution est fixe.

▶ **Exemple 1.** Considérez la configuration montrée sur la figure 9.14 pour une file d'entrée unique, on voit que le processus qui a besoin de 5 Ko entre dans la partition de 6 Ko, le processus de 2 Ko dans la partition de 2 Ko. Le troisième processus de la file reste en attente, bloquant le quatrième processus, qui pourrait s'exécuter dans la partition de 4K, qui est libre, mais

aussi le processus de 1 Ko pourrait être exécuté dans cette partition. Cette décision montre l'alternative entre le concept *best-fit-only* (c'est à dire, exécuter le processus le plus adéquat en termes d'espace) et *first-fit only*, où on exécute le processus avec une priorité plus haute, qui est en tête de la liste.

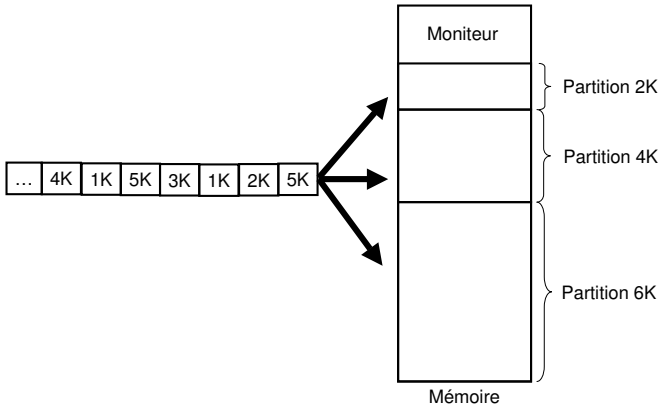


FIG. 9.14 – Configuration de file d'entrée unique.

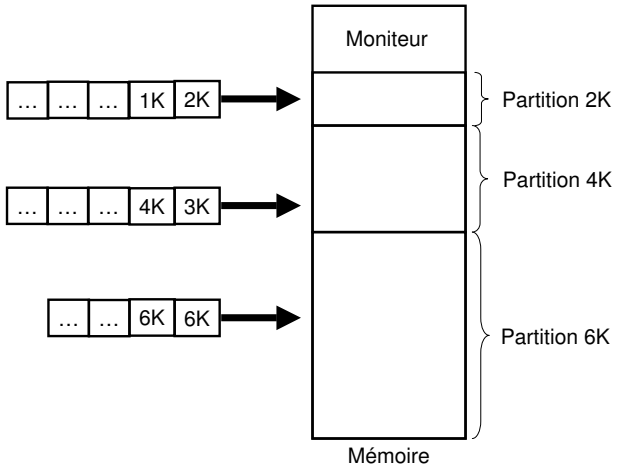


FIG. 9.15 – Configuration de files multiples.

Pour assurer la protection des processus dans le système IBM 360, IBM fut obligée à :

- Diviser la mémoire en blocs de 2Ko et assigner 4 bits de protection à chaque bloc.
- Le mot d'état ou PSW (*program status word*) contenait une clé de 4 bits.
- Le 360 déroutait toutes tentatives d'accès à la mémoire dont les codes de protection différaient de la clé du PSW.
- Ce mécanisme interdisait les interactions entre les processus ou entre les processus et le système d'exploitation.

L'organisation de la mémoire en partitions fixes sans va-et-vient est une méthode facile à mettre en œuvre pour les systèmes à traitement par lots. Mais avec les systèmes à temps partagé, la situation est différente. Puisque la mémoire ne peut pas contenir les processus de tous les utilisateurs, il faut placer quelques uns sur le disque, en utilisant le va-et-vient. Il faut, bien sûr, ramener ces processus en mémoire principale avant de les exécuter. Le mouvement des processus entre la mémoire principale et le disque est appelé va-et vient (*swapping* en anglais). Un autre inconvénient de la partition fixe est la perte de place à cause des programmes qui sont plus petits que les partitions.

9.6 Partitions variables et va-et-vient

Le problème principal avec **MFT** est la détermination optimale des tailles des partitions pour minimiser la fragmentation externe et interne. En systèmes hautement dynamiques, où on ignore le nombre et la taille des processus cela est impossible. Même dans des systèmes avec une charge connue à l'avance, ceci peut devenir problématique.

Par exemple, imaginez un système simple avec une mémoire de 120 Ko, tous les processus utilisent moins de 20 Ko, sauf un qui utilise 80 Ko et qui s'exécute une fois par jour. Évidement pour permettre à tous les processus de s'exécuter le système doit posséder au moins une partition de 80 Ko, et les autres partitions de 20 Ko. Ainsi le système utilisera seulement une fois par jour les 80 Ko, et le reste du temps elle sera inutilisée, avec une fragmentation externe de 80 Ko ; ou bien elle sera occupée partiellement par un processus de 20 Ko avec une fragmentation interne de 60 Ko. Dans les deux cas on n'utilise que la moitié de la mémoire du système.

La solution consiste à faire que les zones de mémoire varient dynamiquement. Cette technique s'appelle assignation de **multiples partitions de mémoire à taille variable** (**MVT**). Le système d'exploitation a une liste de toutes les zones de mémoire non utilisées et de toutes celles utilisées, ainsi

quand un nouveau processus doit s'exécuter, on trouve un espace de mémoire assez grande pour lui contenir et on met toutes ses données et instructions dans cet espace pour l'exécuter.

Avec des partitions variables le nombre et la taille des processus en mémoire varient au cours du temps. L'allocation de la mémoire varie en fonction de l'arrivée et du départ des processus et de la mémoire principale. Cette forme d'allocation conduit éventuellement à l'apparition des trous.

▶ **Exemple 2.** Considérez les caracteristiques suivantes d'un ensemble de processus :

Processus	Besoins de mémoire	Temps
1	60K	10
2	100K	5
3	30K	20
4	70K	8
5	50K	15

TAB. 9.1 – Leur ordonancement et besoins mémoire sont montrés sur les figures 9.16 et 9.17.

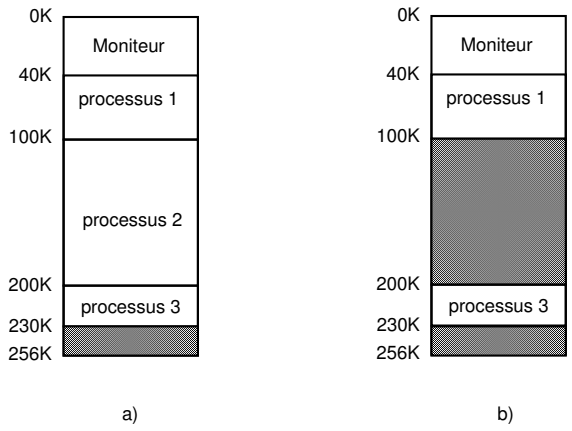


FIG. 9.16 – Sous un ordonnanceur *robin-round* (a) le premier qui termine et libère de la mémoire est le processus 2.

On peut réunir les espaces inutilisés en une seule partition en déplaçant tous les processus vers le bas, comme le fait le **compactage de mémoire** montré sur la figure 9.18. Cette méthode n'est généralement pas utilisée car

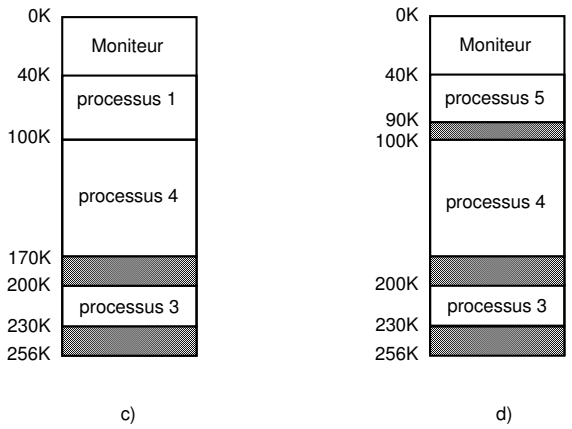


FIG. 9.17 – Selon FCFS (b) le processus 4 entre. Après le processus 1 termine (c) et le processus 5 entre en exécution (d).

elle requiert beaucoup de temps processeur. En revanche, elle est utilisée sur les grands ordinateurs car ils possèdent des circuits spécialisés [4].

L'allocation est simple si les processus ont une taille fixe qu'ils ne modifient jamais. Si par contre, les tailles des segments de données des processus peuvent varier (par exemple par l'utilisation d'allocation dynamique), on a des difficultés dès qu'un processus demande de la mémoire. S'il existe un espace libre adjacent au processus, ce dernier pourra occuper cet espace mémoire. Si cela n'est pas le cas, il faut soit le déplacer dans un emplacement plus grand qui peut le contenir, soit déplacer un ou plusieurs processus sur le disque pour créer un emplacement de taille suffisante. Si la mémoire est saturée et si l'espace de va-et-vient sur le disque ne peut plus contenir de nouveaux processus, il faut tuer le processus qui a demandé de la mémoire.

La taille des processus augmente, en général, au cours de leur exécution. On peut donc allouer à chaque processus déplacé ou recopié à partir du disque un peu plus de mémoire qu'il en demande afin de réduire le temps système perdu pour cette gestion. Il faut cependant recopier sur disque uniquement la mémoire réellement utilisée par le processus. Pour gérer l'allocation et la libération de l'espace mémoire, le gestionnaire doit connaître l'état de la mémoire. Il existe trois manières de mémoriser l'occupation de la mémoire :

[4] Il faut 1 seconde pour compacter la mémoire de 1 Mo sur un PC. La CDC Cyber a de circuits qui compactent la mémoire à la vitesse de 40 Mo/s.

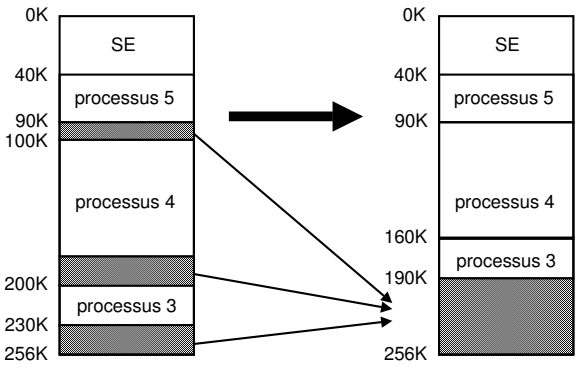


FIG. 9.18 – Compactage de mémoire.

1. Les tables de bits ou *bitmaps*.
2. Les listes chaînées.
3. Les subdivisions.

9.6.1 Gestion de la mémoire par table de bits

La mémoire est divisée en unités d'allocation dont la taille peut varier de quelques mots à plusieurs Ko. A chaque unité, on fait correspondre un bit dans une **Table de bits** ou *bitmap* qui est mis à `0` si l'unité est libre et à `1` si elle est occupée (ou vice versa), comme illustré à la figure 9.19. Plus l'unité d'allocation est faible plus la table de bits est importante. 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).

0	0	1	1	0	0	0	1	1	1	···

FIG. 9.19 – Table de bits.

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.

La figure 9.20 montre l'état de la mémoire avec trois processus qui tournent et la table de bits correspondante.

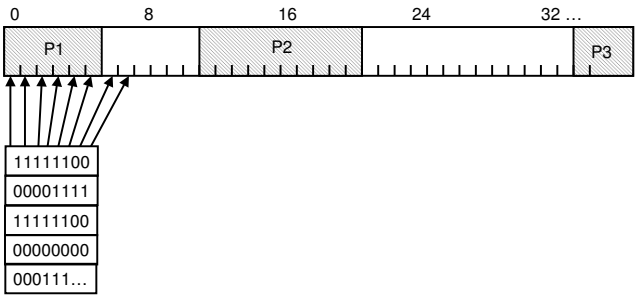


FIG. 9.20 – État de mémoire avec trois processus et son *bitmap*.

9.6.2 Gestion de la mémoire par liste chaînée

Pour mémoriser l'occupation de la mémoire, on utilise une liste 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 :

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

Les listes peuvent être organisées en ordre :

- de libération des zones,
- croissant des tailles,
- décroissant des tailles,
- des adresses de zones.

Par exemple, pour une mémoire ayant l'état d'occupation montré sur la figure 9.21a) on aurait la liste 9.21b).

Libération d'espace

Cet exemple, pris de [**?**], montre que quatre cas peuvent se présenter lors de la libération de l'espace occupé par un processus X.

- **1er cas.** Modifier l'indicateur de l'état du segment (il passe à libre).
A	X	B	→	A		B
- **2ème cas.** Fusionner les deux segments (il passe à libre).
A	X		→	A	
- **3ème cas.** Modifier l'indicateur de l'état du segment (il passe à libre). Fusionner les deux segments.
	X	B	→		B

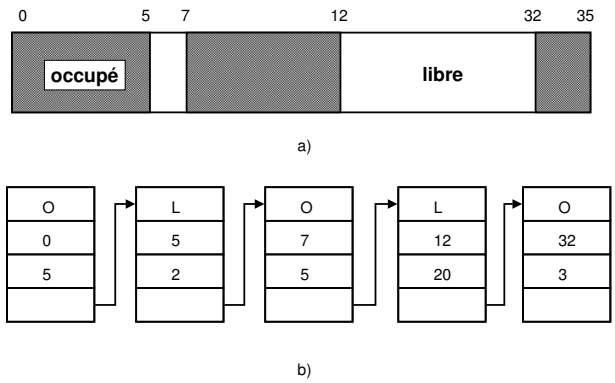
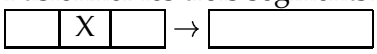


FIG. 9.21 – a) État de mémoire. b) Liste chaînée.

– **4ème cas.** Modifier l'indicateur de l'état du segment (il passe à libre). Fusionner les trois segments.



Allocation d'espace

L'allocation d'un espace libre pour un processus peut se faire suivant quatre stratégies principales : le « premier ajustement » ou *first fit*, la « zone libre suivante », le « meilleur ajustement » ou *best fit* et le « pire ajustement » ou *worst fit*.

1ère zone libre (*first fit*) Dans le cas du « premier ajustement », on recherche le premier bloc libre de la liste qui peut contenir le processus qu'on désire charger. La zone est scindée en deux parties : la première contient le processus et la seconde est l'espace inutilisé (s'il n'occupe pas toute la zone).

Inconvénient : Fragmentation externe (Segments libres entre processus).

Zone libre suivante Il est identique au précédent mais il mémorise en plus la position de l'espace libre trouvé. La recherche suivante commencera à partir de cette position et non à partir du début.

Meilleur ajustement (*best fit*) On cherche dans toute la liste la plus petite zone libre qui convient. On évite de fractionner une grande zone dont on pourrait avoir besoin ultérieurement. Cet algorithme est plus lent que le premier. Il a tendance à remplir la mémoire avec de petites zones libres inutilisables.

Plus grand résidu ou pire ajustement (*worst fit*) Il consiste à prendre toujours la plus grande zone libre pour que la zone libre restante soit la plus grande possible. Les simulations ont montré que cette stratégie ne donne pas de bons résultats.

9.6.3 Algorithme utilisant deux listes séparées

La complexité de la gestion est accrue car cela nécessite des déplacements d'éléments d'une liste à une autre. L'algorithme de placement rapide utilise des listes séparées pour les tailles les plus courantes (la liste des zones de 4 Ko, la liste des zones de 8 Ko, et ainsi de suite).

La recherche est plus rapide, mais lors de la libération il faut examiner les voisins afin de voir si une fusion est possible, sinon on aura une fragmentation en petites zones inutilisables.

9.6.4 Gestion de la mémoire par subdivision

Dans le schéma de **gestion par subdivision** ou *Buddy system*), le gestionnaire de la mémoire mémorise une liste des blocs libres dont les tailles sont de 1, 2, 4, $\ldots 2^n$ octets jusqu'à la taille maximale de la mémoire.

Initialement, on a un seul bloc libre. Lorsqu'il n'y a plus de bloc de taille S_i, on subdivise un bloc de taille S_{i+1} en deux blocs de taille S_i. S'il n'y a plus de bloc de taille S_{i+1}, on casse un bloc de taille S_{i+2}, et ainsi de suite. Lors d'une libération de bloc de taille S_i, si le bloc compagnon est libre, il y a reconstitution d'un bloc de taille S_{i+1}. Pour un système binaire[5] la relation de récurrence sur les tailles est donc :

$$S_{i+1} = 2 \cdot S_i$$

Par exemple, supposons que la taille maximale de mémoire principale est de 1 Mo et que un processus de 70 Ko demande à être chargé. Le gestionnaire détermine d'abord la taille du bloc qui lui sera alloué : qui est égale à la plus petite puissance de 2 qui lui est supérieure, soit 128 Ko. Comme il n'y a pas de blocs libres de taille 128 Ko, 256 Ko ou 512 Ko, la mémoire de 1 Mo est divisée en deux blocs de 512 Ko. Le premier bloc est divisé en deux blocs de 256 Ko. Enfin, le premier des deux blocs nouvellement créés est divisé en deux blocs de 128 Ko. L'espace alloué au processus est situé entre l'adresse 0 et 128 Ko.

[5] Il peut y avoir d'autres systèmes, comme par exemple celui de Fibonnaci, où la relation de récurrence est : $S_{i+1} = S_i + S_{i-1}; S_0 = S_1 = 1$

L'allocation par subdivision est rapide mais elle est assez inefficace en ce qui concerne l'utilisation de la mémoire, car arrondir les tailles à une puissance de 2 implique une certaine fragmentation interne. Linux utilise cet algorithme pour l'assignation de mémoire [**?**].

9.7 Allocation d'espace de va-et-vient

Le va-et-vient est mis en œuvre lorsque tous les processus ne peuvent pas tenir simultanément en mémoire. On doit alors déplacer temporairement certains d'entre eux sur une mémoire provisoire, en général une partie réservée du disque (appelée mémoire de réserve, *swap area* ou encore *backing store*), comme montré à la figure 9.22.

Sur le disque, la zone de va-et-vient d'un processus peut être allouée à la demande dans la zone de va-et-vient générale. Quand un processus est déchargé de la mémoire centrale, on lui recherche une place. Les places de va-et-vient sont gérées de la même manière que la mémoire centrale.

La zone de va-et-vient d'un processus peut aussi être allouée une fois pour toute au début de l'exécution. Lors du déchargement, le processus est sûr d'avoir une zone d'attente libre sur le disque.

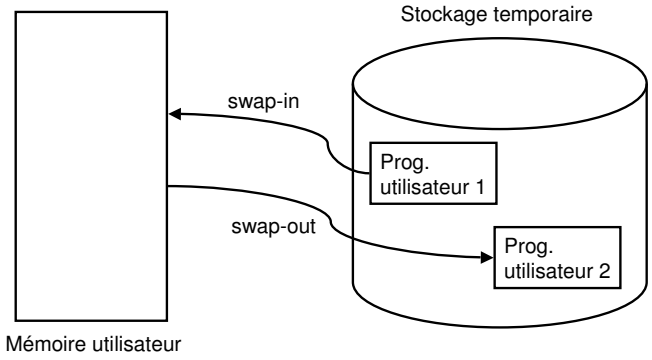


FIG. 9.22 – Espace de va-et-vient.

Le système de va-et-vient, s'il permet de pallier le manque de mémoire nécessaire à plusieurs processus, n'autorise cependant pas l'exécution de programmes de taille supérieure à celle de la mémoire centrale. Une solution consiste à permettre à l'utilisateur de diviser son programme en segments (**segments de recouvrement** ou *overlays* en anglais) 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.

Cette solution n'est pas intéressante, car c'est le programmeur qui doit se charger du découpage de son programme en segments de recouvrement. Une autre solution consiste à décharger le programmeur de cette tâche. Le système d'exploitation charge en mémoire les parties du programme qui sont utilisées. Le reste du programme est stocké sur le disque. On peut ainsi exécuter des programmes dont la taille dépasse celle de la mémoire. Ces techniques appelées mémoire virtuelle seront l'objet d'étude du Chapitre **??**, *Mémoire virtuelle*.

9.8 Projection de fichiers en mémoire

On peut utiliser une technique pour **projeter un fichier** dans la mémoire. Si un programme accède à une adresse mémoire qui appartient à la zone associée au fichier, il sera en correspondance directe au fichier. Le programme n'aura plus besoin des services du système d'exploitation pour lire (`read`) et/ou écrire (`write`) dans le fichier.

Il y a quelques avantages de cette technique par rapport à l'accès conventionnel des fichiers par des appels au système.

- Diminution du nombre d'appels au système pour accéder au fichier.
- On évite les copies intermédiaires, car le système d'exploitation transfère directement l'information entre la zone de mémoire et le fichier.
- Plus de facilités de programmation des accès au fichier. Une fois que le fichier est projeté, on peut l'accéder de la même manière que n'importe quelle structure de données du programme.

9.9 Services Posix de projection de fichiers

Les tâches de gestion de mémoire faites par le système d'exploitation sont plutôt internes. C'est pourquoi il n'y a pas beaucoup de services Posix dédiés aux programmes d'utilisateur [**?**][6]. Les services principaux sont en rapport avec la projection de fichiers.

Projection d'un fichier Ce service permet d'inclure dans la mémoire d'un processus un fichier au complet ou une partie. Cette opération crée une zone en mémoire associée à l'objet stocké dans le fichier. Il est possible de spécifier quelques propriétés de la zone mémoire, par exemple le type de protection, ou si la zone est partagée ou privée.

[6] D'autres techniques de partage de variables et zones mémoire seront vues au Chapitre **??** *Communications IPC*, dans la section **??** *Mémoire partagée*.

Libération de la projection Ce service élimine une projection entière ou une partie.

9.9.1 Projection d'un fichier

L'appel au système `mmap()` a le prototype :

```
#include <sys/mman.h>
caddr_t mmap(caddr_t adresse, size_t longueur, int protection,
             int attribut, int fd, off_t décalage);
```

- `adresse` : indique l'adresse où on projettera le fichier. Normalement on met la valeur `NULL` pour laisser au système le choix.
- `longueur` : longueur de la projection.
- `protection` : mode de protection. `PROT_READ` (lecture), `PROT_WRITE` (écriture) ou `PROT_EXEC` (exécution) ou leurs combinaisons. Le mode de protection doit être compatible avec le mode d'aperture du fichier.
- `fd` : descripteur du fichier.
- `attribut` : permet d'établir des certaines propriétés de la zone.
- `décalage` : correspond à la position dans le fichier de la projection. On n'est pas obligé de projeter tout le fichier.
  - `MAP_SHARED` : Zone partagée. Les modifications vont affecter le fichier. Un processus fils partage cette zone avec le père.
  - `MAP_PRIVATE` : Zone privée. Les modifications n'affectent pas le fichier. Un processus fils ne partage pas cette zone avec son père, il obtient un dupliqué.
  - `MAP_FIXED` : Le fichier va se projeter exactement dans l'adresse spécifiée par le premier paramètre s'il est $\neq$ `0`.

9.9.2 Libération de la projection

L'appel au système `munmap()` avec le prototype :

```
int munmap(void *zone, size_t longueur);
```

libère la projection d'une certaine `longueur` qui se trouve dans la `zone` indiquée. Lorsqu'il réussit il renvoi `0`.

▶ **Exemple 3.** L'exemple suivant, inspiré de [**?**] montre l'utilisation des appels système[7] `mmap()` et `munmap()`. Il projettera un fichier comme un ta-

[7] L'emploi de l'appel système `stat()` sera aussi utilisé pour déterminer la taille du fichier en question. `stat()` fait partie des appels système pour la gestion du *Système de fichiers* (voir chapitre **??**).

bleau de caractères, et ensuite manipulera cette projection en mémoire pour changer le premier caractère avec le dernier, le deuxième avec l'avant dernier, etc. Effectuer la même tâche avec des opérations `read()` et `write()` peut s'avérer beaucoup plus compliqué.

Listing 9.1 – mmap1.c

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main(int argc, char *argv[])
{
char   *projection;
int    fichier;
struct stat etat_fichier;
long   taille_fichier;
int    i;
char   tmp;

if(argc != 2)
  { printf("Syntaxe: %s fichier a inverser\n",argv[0]);
    exit(1);
  }
fichier = open(argv[1], O_RDWR);
stat(argv[1], &etat_fichier);

taille_fichier = etat_fichier.st_size;
projection = (char *) mmap(NULL, taille_fichier,
                           PROT_READ | PROT_WRITE,
                           MAP_SHARED, fichier, 0);
if(projection == (char *) MAP_FAILED)
   {
     perror("mmap");
     exit(0);
   }
close(fichier);

for(i=0; i<taille_fichier/2; i++)
{
   tmp = projection[i];
   projection[i] = projection[taille_fichier - i -1];
   projection[taille_fichier - i -1] = tmp;
}
munmap((void *) projection, taille_fichier);
return 0;
}
```

```
leibnitz> cat > test.txt
No hay nada que canse mas a una persona
que tener que luchar,
no contra su espiritu,
sino contra una abstraccion.
-José Saramago.
^D
leibnitz> mmap1 test.txt
leibnitz> cat test.txt

.ogamaraS ésoJ-
.noiccartsba anu artnoc onis
,utiripse us artnoc on
,rahcul euq renet euq
anosrep anu a sam esnac euq adan yah oNleibnitz>
leibnitz> mmap1 test.txt
leibnitz> cat test.txt
No hay nada que canse mas a una persona
que tener que luchar,
no contra su espiritu,
sino contra una abstraccion.
-José Saramago.
leibnitz>
```

9.10 Note sur MS-DOS

Le système MS-DOS dispose d'un modèle de mémoire nettement plus compliqué que ceux d'Unix et Linux. Il faut interpréter cette complexité comme un défaut : la simplicité d'Unix/Linux n'est en aucune façon synonyme de médiocrité, bien au contraire. La gestion de mémoire du DOS est, par ailleurs, très dépendante du matériel sous-jacent —Intel 80x86— qu'elle exploite jusqu'à la corde. L'architecture originale sur laquelle le MS-DOS a été développé disposait d'un bus d'adresses de 16 bits, permettant d'adresser jusqu'à 64 Ko de mémoire. Ultérieurement, l'adressage s'est fait à partir d'une base dont la valeur est contenue dans des registres spécifiques de 16 bits. Ces registres correspondent généralement à des fonctions de pile (SS), de données (DS) et de code (CS). Les registres contiennent les bits de poids forts d'une adresse de 20 bits, ce qui correspond à un intervalle de 1 Mo, comme montré sur la figure 9.23.

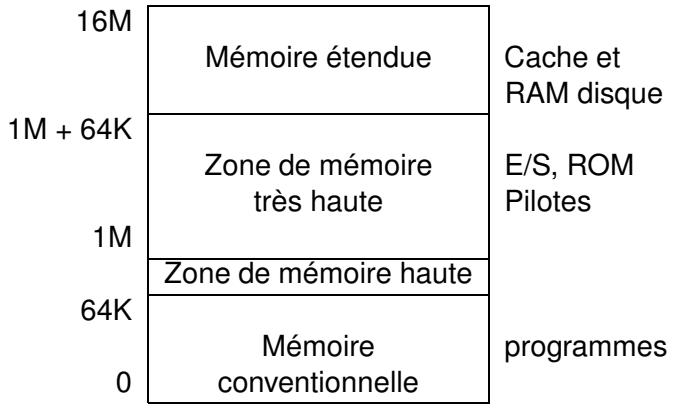


FIG. 9.23 – Mémoire dans MS-DOS.

9.11 Exercices

1. Expliquez la fonction principale d'un éditeur de liens dans la génération de l'espace adresse d'un processus. Pourquoi doit-on relocaliser les adresses dans le fichier exécutable qui est généré par l'éditeur de liens ? Quels éléments du code doivent être relocalisés ? Pourquoi doit-on aussi relocaliser des adresses lors de l'exécution d'un programme ?
2. Expliquez l'une des raisons pour laquelle un processus peut être déalloué de la mémoire principale avant la fin de son exécution.
3. Comparez la fragmentation interne à la fragmentation externe pour la mémoire, et expliquez laquelle pénalise le plus la performance du système.
4. Identifiez les zones de mémoire où sont sauvegardés les divers éléments d'un programme en état d'exécution.
5. Donnez un avantage pour l'utilisation de partitions fixes de mémoire, et un avantage pour l'utilisation de partitions variables de mémoire.
6. Laquelle des stratégies d'allocations de partitions (*first fit*, *best fit*, *worst fit*) minimise le plus la fragmentation ?
7. Quel avantage procure au système d'exploitation l'utilisation de registres de relocation ?
8. Lequel des algorithmes d'allocation de mémoire nous fournirait le meilleur degré de multiprogrammation ? Expliquer.

9. Parmi les algorithmes d'ordonnancement de processus, lequel résulterait en un meilleur degré de multiprogrammation ? Expliquez.
10. Considérez un système dont l'espace mémoire utilisateur compte 1 Mo. On décide d'effectuer une **partition fixe** de cet espace mémoire en trois partitions de tailles respectives 600 Ko, 300 Ko, 100 Ko. On suppose que les processus arrivent dans le système comme montré sur la table 9.2.

Instant t	Événement
t = 0	A(200 Ko, 35) arrive
t = 10	B(400 Ko, 65) arrive
t = 30	C(400 Ko, 35) arrive
t = 40	D(80 Ko, 25) arrive
t = 50	E(200 Ko, 55) arrive
t = 70	F(300 Ko, 15) arrive

TAB. 9.2 – Besoins de mémoire et temps d'arrivage.

A(200 Ko, 35) veut dire que le processus A nécessite une partition de 200 Ko et que son temps de séjour en mémoire centrale est 35 unités de temps. Bien entendu, un processus qui ne peut pas être chargé en mémoire est placé sur la file des processus en attente de chargement en mémoire. Un processus chargé en mémoire y séjournera jusqu'à la fin de son exécution. Donnez les états successifs d'occupation de la mémoire si :
   - L'ordonnanceur de haut niveau fonctionne selon le schéma plus court d'abord (SJF) et le mode d'allocation des trous utilise un algorithme du type *best fit*.
   - L'ordonnanceur fonctionne selon PAPS (FCFS) et le mode d'allocation des trous utilise un algorithme du type *First Fit*.
11. Considérez un gestionnaire de mémoire utilisant la stratégie de partitions variables. Supposez que la liste de blocs libres contient des blocs de taille de 125, 50, 250, 256, 160, 500, et 75 octets. Le bloc de 125 octets est en tête de liste et le bloc de 75 octets est en fin de liste.
   - Lequel des blocs représente le meilleur espace disponible pour un processus nécessitant une taille de mémoire de 84 octets.
   - Lequel des blocs représente le plus grand espace disponible pour un processus nécessitant une taille de mémoire de 221 octets.
   - Lequel des blocs représente le premier espace disponible pour un processus nécessitant une taille de mémoire de 178 octets.

12. Pour chacune des stratégies d'allocation de blocs de mémoire libre (meilleur, plus grand, ou premier), expliquez de quelle façon la liste de blocs libres devrait être organisée afin d'obtenir une performance optimale.
13. Considérez un système n'utilisant pas de registres de relocation. Par conséquent, c'est le chargeur de programme qui s'occupe de la relocation des instructions dans l'image mémoire à partir de l'exécutable. Serait-il possible de décharger et de recharger l'image mémoire du programme durant le cycle d'exécution ? Expliquez comment cela pourrait être réalisable ou pourquoi cela ne pourrait pas être réalisable.