SYSTÈMES D'EXPLOITATION

Notes de cours

par

FARAH

Chapitre 1

La mémoire

1.1. Introduction

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.

De manière encore plus vive que pour les autres ressources informatiques, le prix des mémoires a baissé et la capacité unitaire des circuits a augmenté. Cependant la nécessité de la gérer 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.

1.1.1. La multiprogrammation

Le concept de multiprogrammation s'oppose à celui de monoprogrammation. La monoprogrammation ne permet qu'à un seul processus utilisateur d'être exécuté. Cette technique n'est plus utilisée que dans les micro-ordinateurs. On trouve alors en mémoire, par exemple dans le cas de MS-DOS : le système en mémoire basse, les pilotes de périphériques en mémoire haute (dans une zone allant de 640 ko à 1 Mo) et un programme utilisateur entre les deux.

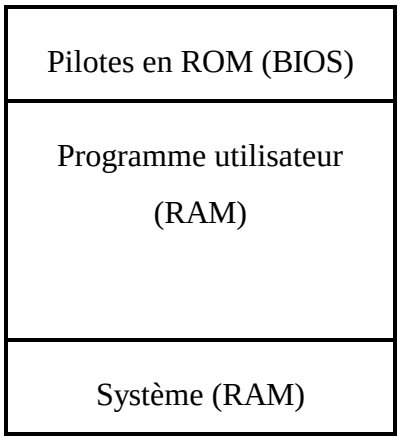

Figure 1 L'organisation de la mémoire du DOS.

La multiprogrammation autorise l'exécution de plusieurs processus indépendant à la fois[1]. Cette technique permet d'optimiser le taux d'utilisation du processeur en réduisant notamment les attentes sur des entrées-sorties. La multiprogrammation implique le séjour de plusieurs programmes en même temps en mémoire et c'est cette technique qui a donné naissance à la gestion moderne de la mémoire.

1.1.2. Les registres matériels

La gestion de la mémoire est presque impossible sans l'aide du matériel. Celui-ci doit notamment assurer la protection. Dans les systèmes multi-utilisateurs, par exemple, on doit interdire à un utilisateur d'accéder n'importe comment au noyau du système, ou aux autres programmes des utilisateurs.

Pour assurer une protection fondamentale, on dispose, sur la plupart des processeurs[2], de deux registres délimitant le domaine d'un processus :

- le registre de base et

---

[1] Il est à noter que MS-DOS, bien qu'il ne soit pas multiprogrammable, n'est pas non plus un système monoprogrammé *stricto sensu*; c'est une créature qui tient des deux.

[2] Malheureusement pas sur le 8086 ce qui a eu des conséquences considérables sur les systèmes DOS et Windows.

- le registre de limite.

La protection est alors assurée par le matériel qui compare les adresses émises par le processus à ces deux registres.

1.2. Concepts fondamentaux

1.2.1. Production d'un programme

Avant d'être exécuté, un programme doit passer par plusieurs étapes. Au début, le programmeur crée un fichier et écrit son programme dans un langage source, le C par exemple. Un **compilateur** transforme ce programme en un module objet. Le module objet représente la traduction des instructions en C, en langage machine. Le code produit est en général relogeable, commençant à l'adresse 00000 et pouvant se translater à n'importe quel endroit de la mémoire en lui donnant comme référence initiale le registre de base. Les adresses représentent alors le décalage par rapport à ce registre.

On peut rassembler les modules objets dans des bibliothèques spécialisées, par exemple au moyen des commandes ar et ranlib (pour avoir un accès direct) sous Unix ou TLIB avec le compilateur C de Borland. On réunit ensuite les bibliothèques dans un répertoire, en général /usr/lib sous Unix.

Les appels à des procédures externes sont laissés comme des points de branchements. L'**éditeur de liens** fait correspondre ces points à des fonctions contenues dans les bibliothèques et produit, dans le cas d'une liaison statique, une image binaire. Certains systèmes, notamment Unix, autorisent des liaisons dynamiques et reportent la phase d'édition de liens jusqu'à l'instant de chargement. Il faut alors construire les objets et les bibliothèques d'une manière légèrement différente. Grâce à cette technique, on peut mettre les bibliothèques à jour sans avoir à recompiler et on encombre moins les disques.

Le **chargeur** effectue les liaisons des appels système avec le noyau, tel que l'appel write par exemple. Enfin, il charge le programme en mémoire.

Outre le code compilé (le texte) et la zone de données initialisées, le fichier exécutable contient un certain nombre d'autres informations. Dans le cas du système Unix, il y a notamment un en-tête composé d'un nombre « magique » (un code de contrôle de type), de diverses tailles (texte, données,…), du point d'entrée du programme, ainsi qu'une table de symboles pour le débogage. Dans le cas de MS-DOS, plusieurs formats cohabitent, notamment COM et EXE.

1.2.2. Principes de gestion

Pour effectuer le chargement, le système **alloue** un espace de mémoire libre et il y place le processus. Il libérera cet emplacement une fois le programme terminé.

Dans beaucoup de cas, il n'est pas possible de faire tenir tous les programmes ensemble en mémoire. Parfois même, la taille d'un seul programme est trop importante. Le programmeur peut, dans ce cas, mettre en œuvre une stratégie de recouvrement (*overlay*) consistant à découper un programme important en modules et à charger ces modules quand ils sont nécessaires. Ceci est cependant très fastidieux et nécessite, pour chaque nouveau programme, un redécoupage.

Les systèmes d'exploitation modernes mettent en œuvre des stratégies qui libèrent le programmeur de ces préoccupations. Il existe deux stratégies principales pour gérer les chargements : le **va-et-vient** et la **mémoire virtuelle**.

1.3. L'allocation

Avant d'implanter une technique de gestion de la mémoire centrale par va-et-vient, il est nécessaire de connaître son état : les zones libres et occupées; de disposer d'une stratégie d'allocation et enfin de procédures de libération. Les techniques que nous allons décrire servent de base au va-et-vient; on les met aussi en œuvre dans le cas de la multiprogrammation simple où plusieurs processus sont chargés en mémoire et conservés jusqu'à la fin de leur exécution.

1.3.1. Etat de la mémoire

Le système garde la trace des emplacements occupés de la mémoire par l'intermédiaire d'une table de bits ou bien d'une liste chaînée. La mémoire étant découpée en unités, en blocs, d'allocation.

1.3.1.1. Tables de bits

On peut conserver l'état des blocs de mémoire grâce à une table de bits. Les unités libres étant notées par 0 et ceux occupées par un 1. (ou l'inverse).

0	0	1	1	0	0	

Figure 2

La technique des tables de bits est simple à implanter, mais elle est peu utilisée. On peut faire la remarque suivante : plus l'unité d'allocation est petite, moins on a de pertes lors des allocations, mais en revanche, plus cette table occupe de place en mémoire.

1.3.1.2. Listes chaînées

On peut représenter la mémoire par une liste chaînée de structures dont les membres sont : le type (libre ou occupé), l'adresse de début, la longueur, et un pointeur sur l'élément suivant.

Pour une mémoire ayant l'état suivant :

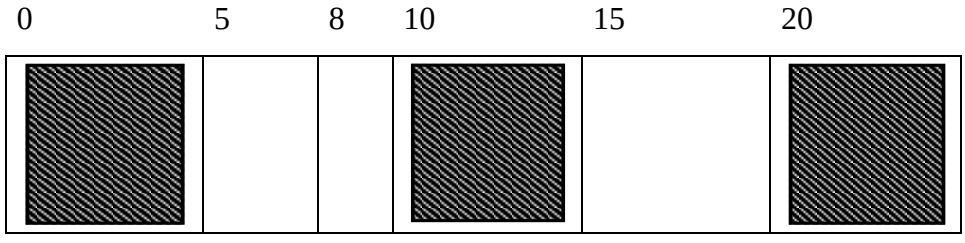

Figure 3

On aurait la liste :

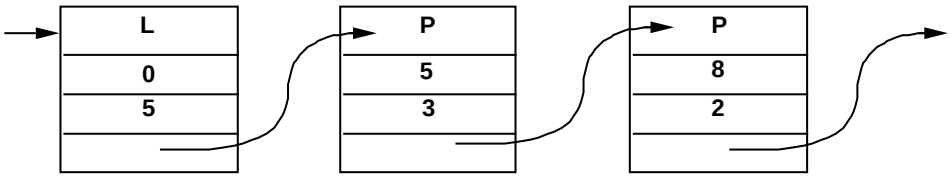

Figure 4

On peut légèrement modifier ce schéma en prenant deux listes : l'une pour les processus et l'autre pour les zones libres. La liste des blocs libres peut elle même se représenter en réservant quelques octets de chaque bloc libre pour contenir un pointeur sur le bloc libre suivant.

Le système MS-DOS utilise une variante de ce procédé grâce à un en-tête de 16 octets qui précède chaque zone (arène) en mémoire. Les en-têtes contiennent notamment le type de l'arène (un pointeur sur le contexte du processus ou 0) et sa taille.

1.3.2. Politiques d'allocation

L'allocation d'un espace libre pour un processus peut se faire suivant trois stratégies principales : le « premier ajustement » (*first fit*), le « meilleur ajustement » (*best fit*), et le « pire ajustement » (*worst fit*).

Dans le cas du « premier ajustement », on prend le premier bloc libre de la liste qui peut contenir le processus qu'on désire charger. Le « meilleur ajustement » tente d'allouer au processus l'espace mémoire le plus petit qui puisse le contenir. Le « pire ajustement » lui prend le plus grand bloc disponible et le fragmente en deux.

Des simulations ont montré que le « premier ajustement » était meilleur que les autres. Paradoxalement, le « meilleur ajustement », qui est plus coûteux, n'est pas optimal car il produit une fragmentation importante.

1.3.3. Libération

La libération se produit quand un processus est évacué de la mémoire. On marque alors le bloc à libre et on le fusionne éventuellement avec des blocs adjacents.

Supposons que X soit le bloc qui se libère, on a les schémas de fusion suivants :

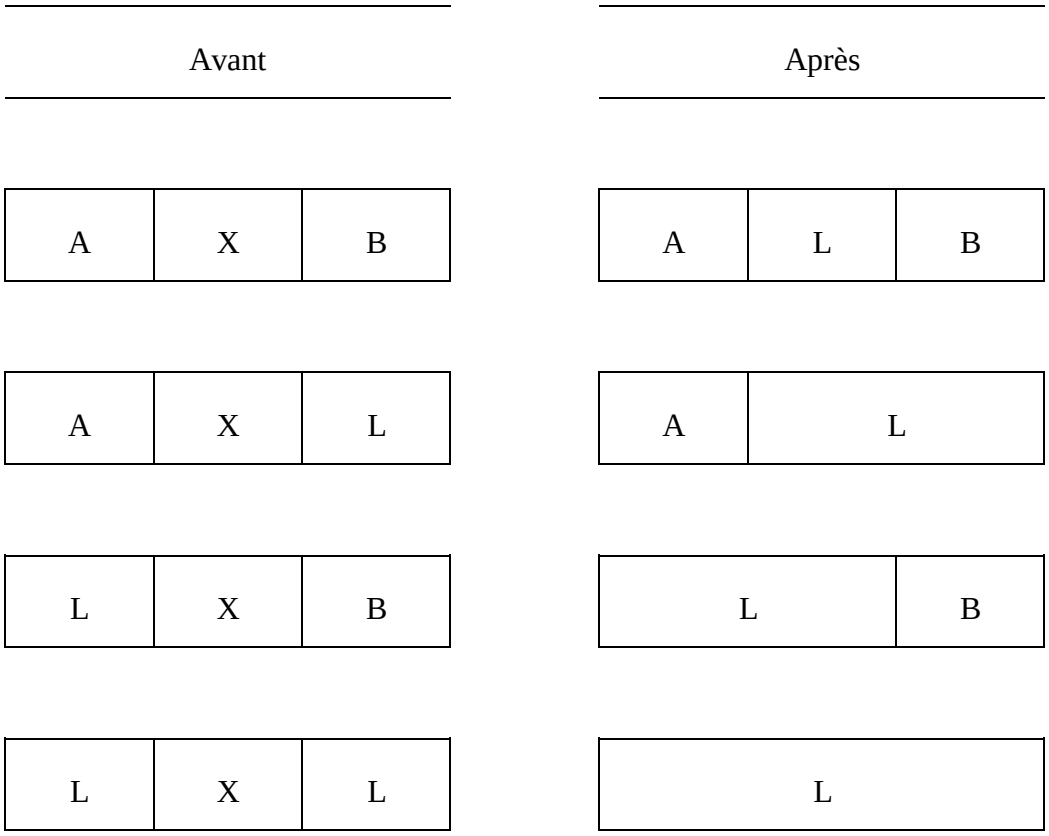

Figure 5

La récupération de mémoire

La fragmentation de la mémoire est particulièrement dommageable car elle peut saturer l'espace disponible rapidement. Ceci est particulièrement vrai pour les gestionnaires de fenêtrage. Pour la diminuer, on peut compacter régulièrement la mémoire. Pour cela, on déplace les processus, par exemple, vers la bas de la mémoire et on les range l'un après l'autre de manière contiguë. On construit alors un seul bloc libre dans le haut de la mémoire. Cette opération est coûteuse et nécessite parfois des circuits spéciaux.

Par ailleurs, une fois qu'un objet ou une zone a été utilisé, le programmeur système doit récupère la mémoire « à la main » par une libération du pointeur sur cette zone - free(). Ceci est une source d'erreurs car on oublie parfois cette opération. Si on alloue dans une fonction, il n'y a plus moyen d'accéder au pointeur après le retour de la fonction. Le bloc de mémoire est alors perdu et inutilisable. On a une « fuite de mémoire » – *memory leak*.

Certains langages ou systèmes incorporent la récupération automatique de mémoire - *garbage collection*. Ils libèrent ainsi le programmeur de cette tâche. C'est le cas de Java. Il est alors très facile d'éliminer immédiatement une zone par l'affectation objet = null. Sans ça, le récupérateur doit déterminer tout seul qu'une zone n'ait plus de référence dans le suite du programme. La récupération automatique de mémoire a fait l'objet de controverses mais les machines devenant plus rapides, cette querelle est sans doute dépassée[3].

L'algorithme de récupération peut poser des problèmes car périodiquement le système s'arrête brutalement pour laisser la place au récupérateur. On peut demander son démarrage explicite (synchrone) par la méthode java.lang.Runtime.gc(). La récupération est aussi dite synchrone lorsqu'il n'y a plus de mémoire dans le tas. Pour pouvoir appeler le récupérateur, il faut lancer l'interprète avec l'option : -noasyncgc. On obtient la quantité de mémoire libre avec la méthode Runtime.freeMemory(). Le tas est par défaut de 1 Mo. On peut positionner sa valeur en lançant l'interprète avec l'option -ms16m (pour 16 Mo).

1.4 Mémoire uniforme :

La première version de mémoire offerte, était une M C adressable entièrement. Ce qui offre à l'utilisateur le possibilité de l'utiliser de la manière qu'il veut. Et aucune spécificité Hard n'est requise pour gérer cette mémoire. Cette méthode à ces limites puisque le système d'exploitation n'a aucun contrôle sur les interruptions. Il n'y a pas de moniteur résident pour gérer les appels système ou les erreurs (surtout la lecture par exemple des cartes de contrôle).

1.5 Moniteur résident :

Cette méthode suggère qu'on divise la mémoire en deux parties, une partie pour l'utilisateur et le seconde pour la partie moniteur résident du S.E.. Il est plus commode de placer le moniteur résident dans la mémoire basse, avec l'utilisation d'un registre barre de

---

[3] Selon les tenants du Prolog (ou du Lisp), la récupération est trop importante pour la laisser aux programmeurs et selon ceux du C++, elle est trop importante pour la laisser au système.

verrouillage (fence register) pour protéger le moniteur. Toutes les adresses générées par le programme utilisateur (instructions ou données) doivent être comparés avec ce registre.

Programme génère une adresse

Si adresse >= contenu du registre barre de verrouillage alors accès possible
Sinon erreur, générer une interruption.

Avantages : Simple, La tache dispose de toute la mémoire.
Inconvénients :

- Mauvaise utilisation de la MC si le programme et inférieur à la mémoire,
- Le programme doit avoir une taille inférieure ou égale à la MC
- Si E/S la CPU reste inactive

1.6 PARTITIONS MULTIPLES :

Cette méthode découle directement du principe de la multiprogrammation. Ou on divise la mémoire en plusieurs partitions, chacune affectée à un programme. Pour des besoins de sécurités des utilisateurs deux registres ont été utilisés. Ce qui permet de délimiter l'espace d'adressage d'un programme, ces registres sont :

- Registres limites (Limite inférieure, et limite supérieure):

Utilisés dans le cas d'une localisation statique.

- Registre de Base et registre Limite

Toutes les adresses doivent être inférieures à la Limite, et toutes les adresses sont recalculées en y rajoutant automatiquement le contenu du registre Base.

1.6.1 Partition fixe :

Le découpage de la MC en partitions se fait indépendamment des travaux en cours. Par exemple si on a une mémoire de 32Ko et si on la divise en partitions comme suit :

- Moniteur résident 10 Ko
- Petites taches 4 Ko
- Moyennes Taches 6Ko
- Grandes taches 12 Ko

Avantages :

Facile à réaliser, et possibilité de classer les programmes par leur taille afin d'occuper la zone mémoire qui leur convient.

Inconvénients :

- Limitation du degré de multiprogrammation
- Limitation de la taille des programmes par la taille des partitions
- Mauvaise utilisation de la mémoire, Notion de "fragmentation"

Exemple :

Partitions	Taille partitions (Ko)	Taille programmes(Ko)
1	8	5
2	16	11
3	128	50
4	256	131

Si un programme arrive avec une taille de par exemple 100Ko il ne peut être servi, malgré que l'espace globale non utilisé est supérieur à 100Ko.

1.6.2 Partitions variables :

Le découpage s'effectue au fur et à mesure de l'exécution des travaux de façon à adapter la taille des partitions à celle des taches à exécuter.

Exemple :

Si on dispose d'une MC de 256 Ko, et que le moniteur occupe 40 Ko, ce qui nous laisse 216 Ko à partager entre les utilisateurs.

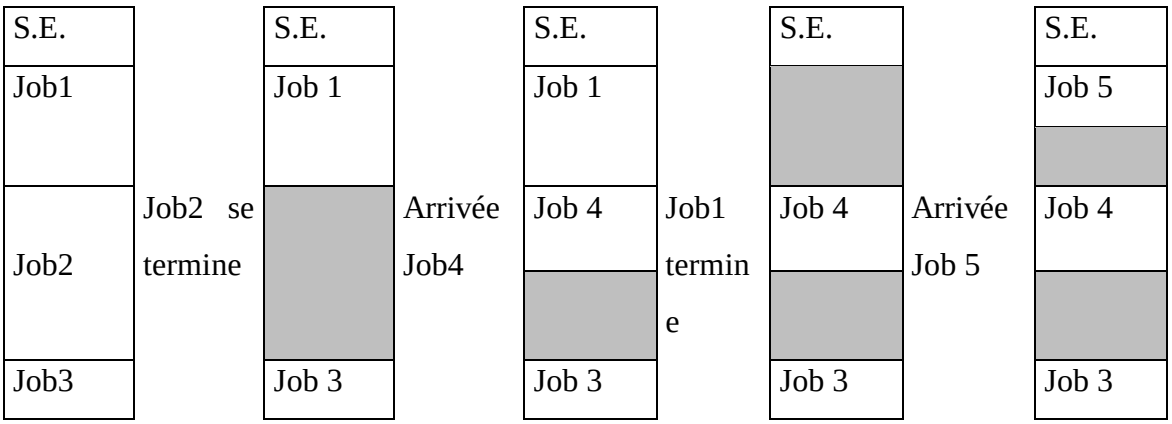

Figure 6

Job 1 =[40k - 100 k]; Job2=[100K-200K]; Job3=[200K-230K]; Job4=[100K-170K]; Job5=[40K-90K]

Avantage :

- Degré de multiprogrammation supérieur

Inconvénients :
- Complexités des Algorithmes d'allocations et de désallocations
- La fragmentation n'est pas éliminée
- La taille du programme et limitée par l'espace mémoire (contiguë) disponible
- Si un programme est chargé (@ virtuelle → @ physique) il n'est plus possible de le déplacer donc impossibilité de récupérer l'espace des fragments libres.

1.7. Le va-et-vient (SWAPING) (Optionnel dans le cours)

Le va-et-vient est mis en œuvre lorsque tous les processus ne peuvent pas tenir simultanément en mémoire. On doit alors en déplacer temporairement certains sur une mémoire provisoire, en général, une partie réservée du disque (*swap area* ou *backing store*).

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 (*swap area*). 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.

Le système exécute pendant un certain quantum de temps les processus en mémoire puis déplace un de ces processus au profit d'un de ceux en attente dans la mémoire provisoire. L'algorithme de remplacement peut être le tourniquet.

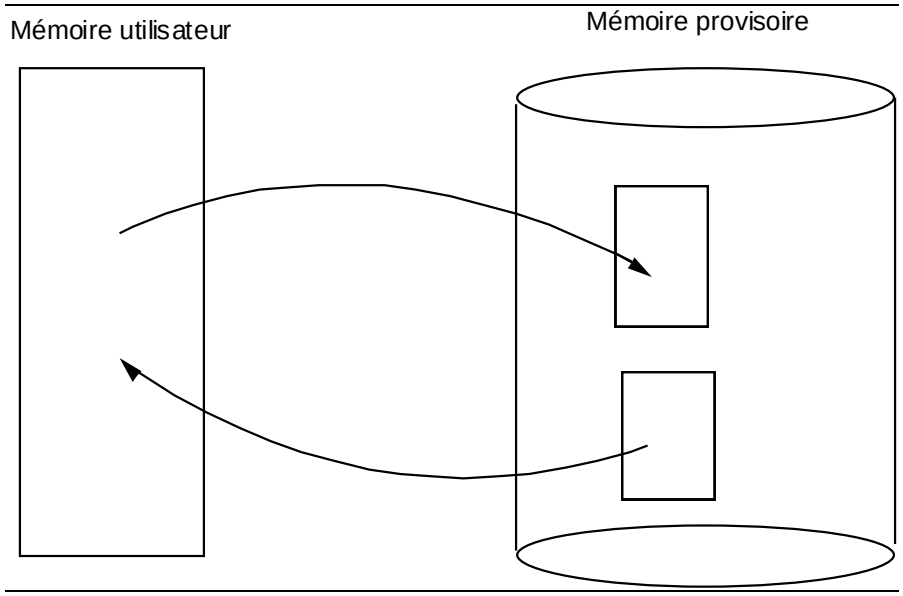

Figure 7

Le système de va-et-vient, s'il permet de pallier le manque de mémoire nécessaire à plusieurs utilisateurs, n'autorise cependant pas l'exécution de programmes de taille supérieure à celle de la mémoire centrale.

Il faut noter que le temps nécessaire pour l'exécution d'un processus doit être largement plus long que celui nécessaire pour faire le Swapping.

1.8. La pagination

La pagination permet d'avoir en mémoire un processus dont les adresses sont non contiguës. Pour réaliser ceci, on partage l'espace d'adressage du processus et la mémoire physique en pages de quelques kilo-octets. Les pages du processus sont chargées à des pages libres de la mémoire.

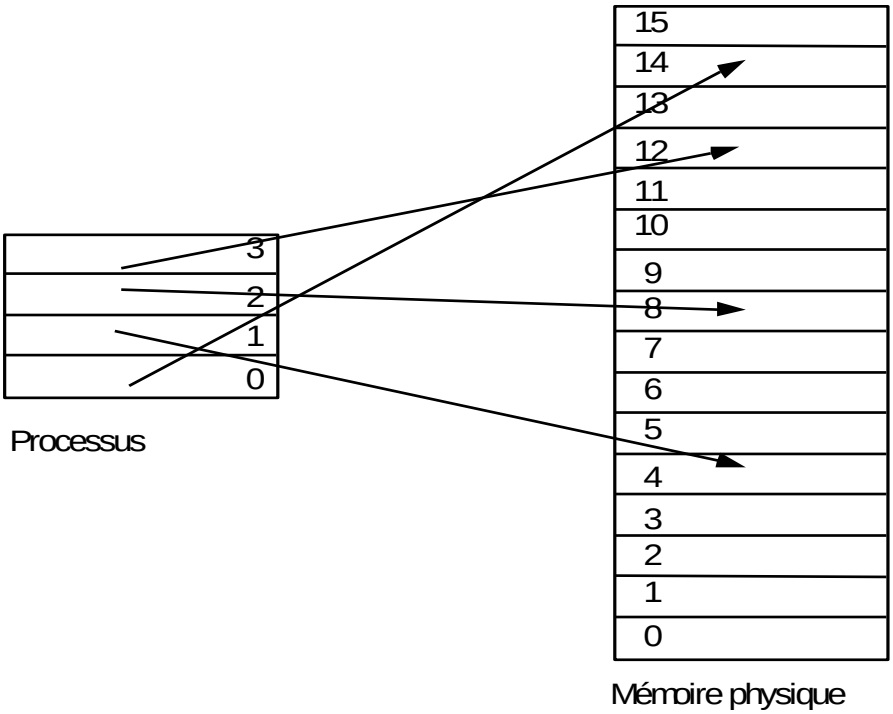

Figure 8

On conserve l'emplacement des pages par une table de transcodage :

0	14
1	4
2	8
3	12

La pagination permet d'écrire des programmes ré-entrants, c'est à dire où certaines pages de codes sont partagées par plusieurs processus.

1.9. La segmentation

Alors que la pagination propose un espace d'adressage plat et indifférencié, (ceci est offert par la famille de µ-processeurs 68000), la segmentation partage les processus en segments bien spécifiques. On peut ainsi avoir des segments pour des procédures, pour la table de symboles, pour le programme principal, etc. Ces segments peuvent être relogeables et avoir pour origine un registre de base propre au segment. La segmentation permet aussi le partage, par exemple du code d'un éditeur entre plusieurs processus. Ce partage porte alors sur un ou plusieurs segments.

Un exemple réduit d'architecture segmentée est donné par la famille 8086 qui possède 3 segments : le code (Code Segment), la pile (Stack Segment) et les données (Data Segment).

1.10. La mémoire virtuelle

1.10.1. Présentation

La mémoire virtuelle permet:

- D'exécuter des programmes dont la taille excède la taille de la mémoire réelle. Pour ceci, on découpe (on « pagine ») les processus ainsi que la mémoire réelle en pages de quelques kilo-octets (1, 2 ou 4 ko généralement).

- La mémoire **physique** étant coûteuse donc en général de capacité réduite, ce qui a donnée l'idée d'utiliser la Mémoire **secondaire** (disques, mémoire étendue, ...) qui sont peu coûteuses. Et essayer d'utiliser la mémoire secondaire "comme" mémoire RAM.

L'encombrement total du processus constitue l'espace d'adressage ou la mémoire virtuelle. Cette mémoire virtuelle réside sur le disque. À la différence de la pagination présentée précédemment, on ne charge qu'un sous-ensemble de pages en mémoire. Ce sous ensemble est appelé l'espace physique (réel).

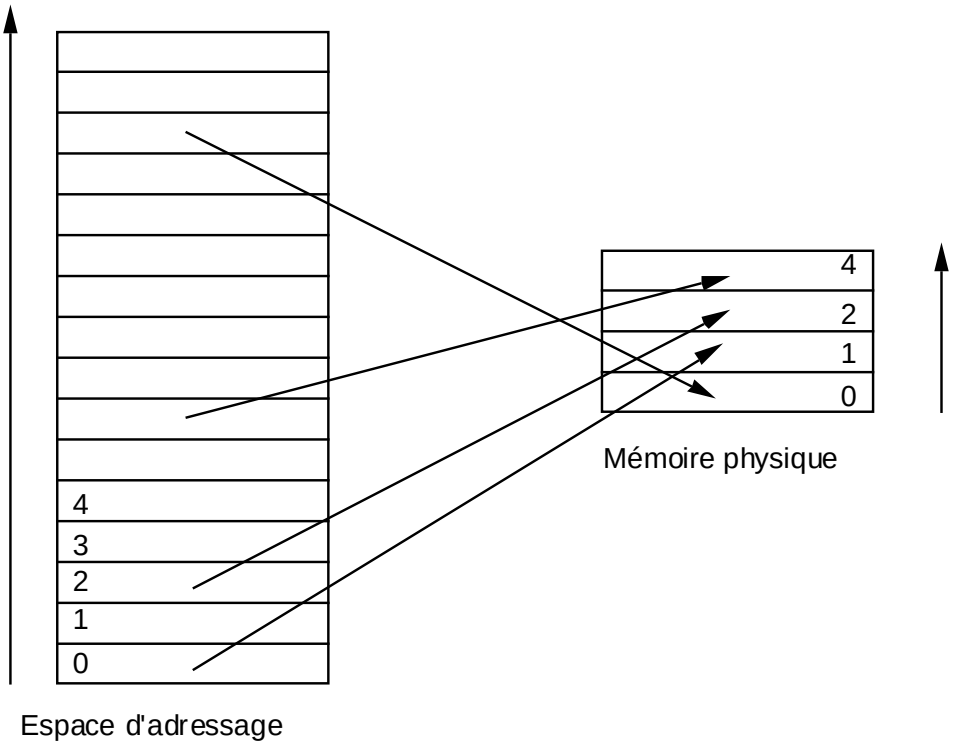

Figure 9

Le problème est de faire correspondre une adresse virtuelle à une adresse réelle en mémoire, de protéger les utilisateurs entre eux, de gérer le partage d'information.

On utilise une fonction topographique qui associe à une adresse virtuelle, une adresse réelle.

Voici la fonction topographique :

Fonction topo(x:*adresse_virtuelle*):*adresse_réelle*;
début
topo:=f(x);
fin

La mémoire virtuelle, avec sa table des pages, est **une** implémentation possible de la fonction topographique.

Lorsqu'une adresse est générée, elle est transcodée, grâce à une table, pour lui faire correspondre son équivalent en mémoire physique. Ce transcodage peut aussi être effectué par des circuits matériels de gestion : Memory Management Unit (MMU). Si cette adresse

correspond à une adresse en mémoire physique, le MMU transmet sur le bus l'adresse réelle, sinon il se produit un défaut de page. Pour pouvoir accéder à la page dont on a généré l'adresse, on devra préalablement la charger en mémoire réelle. Pour cela, on choisit parmi les pages réelles une page « victime »; si cette dernière a été modifiée on la reporte en mémoire virtuelle (sur le disque) et on charge à sa place la page à laquelle on désirait accéder.

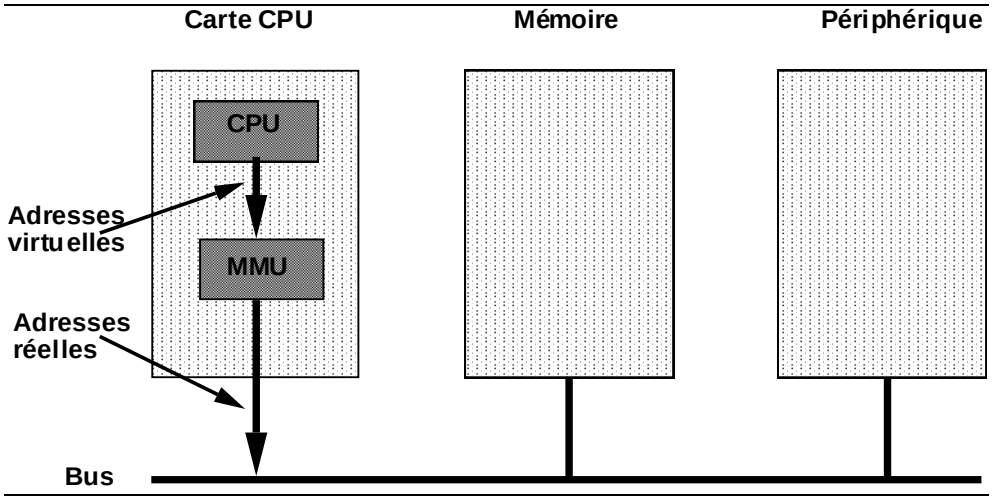

Figure 10 L'unité de gestion de mémoire.

Il est plus facile d'implanter l'algorithme en utilisant des tailles correspondant à des puissances de deux. Pour une adresse virtuelle ou réelle, on réserve les bits de poids forts nécessaires pour coder les pages réelles et virtuelles. Les bits de poids faibles codent les décalages à l'intérieur de chacune de ces pages. Par exemple, supposons que les pages fassent 4 ko; que la taille du processus fasse 16 pages; que la taille allouée en mémoire soit de 4 pages. Il faut 2 bits pour pouvoir coder les pages réelles, 4 bits pour les pages virtuelles et 12 bits pour les décalages.

Structure des adresses mémoires réelles

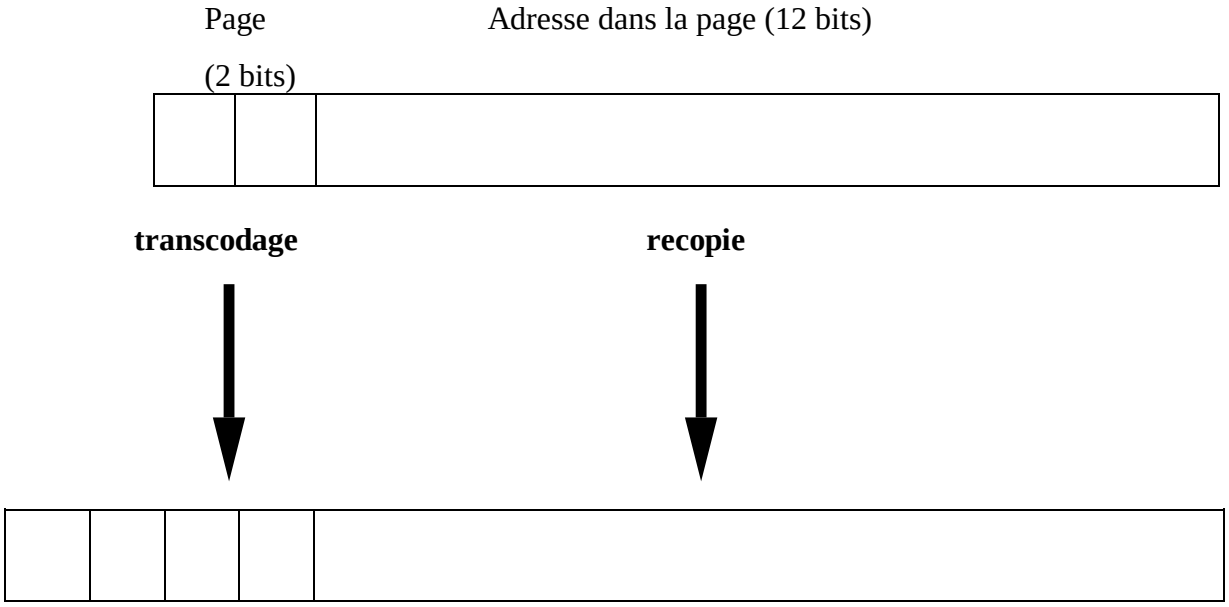

Figure 11 Structure des adresses mémoire de l'espace d'adressage

Pour réaliser le transcodage, on conserve des tables contenant les données nécessaires avec notamment : un bit pour marquer la présence de la page en mémoire réelle et un bit de modification pour signaler si on a écrit dans la page. Dans ce dernier cas, la page devra être reportée sur le disque si on désire la remplacer par une autre.

P. Virt.	P. Réel.	Présent	Modif.
0	01	0	
1	–	X	
2	10	0	
3	–	X	
4	–	X	
15	00		

Figure 12

En général, c'est un bit dans le PSW (Program Status Word) qui indique si l'on utilise ou non la mémoire virtuelle.

Principes et mécanismes de base de la pagination

Soit un processeur avec un bus d'adresse sur 32 bits, il peut adresser *2^32* octets, soit 4 Go. L'ordinateur a une mémoire physique de 8 Mo.

L = taille de la page ou de la case, par exemple 4096 octets, soit *2^12*.

N = nombre de pages de la mémoire virtuelle, par exemple 1 Méga de pages, soit *2^20*.

n = nombre de cases de la mémoire physique, par exemple 2048 cases, soit *2^11*.

le déplacement est le même (les pages physiques et virtuelles ont la même taille) ;

si la page virtuelle n'est pas présente en mémoire physique, alors il se produit un *défaut de page*.

Pour accélérer le processus, on utilise des *mémoires associatives* qui recensent les dernières pages utilisées :

1.10.2 Algorithmes de remplacement de pages

Le choix d'une victime - remplacement

De nombreux algorithmes :

- **FIFO** - First In First Out : ordre chronologique de chargement ;
- **LRU** - Least Recently Used : ordre chronologique d'utilisation ;
- **FINUFO** - First In Not Used, First Out (algorithme de l'horloge ou Clock) : approximation du LRU ;
- **LFU** - Least Frequently Used ;
- **Random** : au hasard ;

Performances: LRU, FINUFO, [FIFO, Random].

Optimisation du système : tenir compte de la **localité** en **préchargeant** des pages **avant** d'en avoir besoin.

Localité : à un instant donné, les références observées dans un passé récent sont (en général) une bonne estimation des prochaines références.

En moyenne, 75% des références intéressent moins de 20% des pages. C'est la **non-uniformité**.

On va essayer d'anticiper la demande.

Le problème de la taille de la TPV

Pour être utilisée, la TPV doit être placée en mémoire physique.

Par exemple, si on a *2^20* pages virtuelles, la TPV aura une taille d'à peu près *2^20 * taille d'une entrée* = 10 Mo si une entrée tient sur 10 octets.

C'est à dire plus que la taille de la mémoire physique !!!!

Solution : on va paginer la TPV.

Pagination à deux niveaux

La mémoire virtuelle est divisée en **Hyperpages** qui sont elles-mêmes divisées en **pages**.

Une adresse virtuelle = numéro d'hyperpage ; numéro de page ; déplacement.

Attention ! L'accès à la mémoire est plus lent d'une indirection (utilisation de mémoires associatives).

L'algorithme de remplacement de page optimal consiste à choisir comme victime, la page qui sera appelée le plus tard possible. On ne peut malheureusement pas implanter cet algorithme, mais on essaye de l'approximer le mieux possible, avec de résultats qui ont une différence de moins de 1 % avec l'algorithme optimal.

La technique FirstIn-FirstOut est assez facile à implanter. Elle consiste à choisir comme victime, la page la plus anciennement chargée.

L'algorithme de remplacement de la page la moins récemment utilisée (*Least Recently Used*) est l'un des plus efficaces. Il nécessite des dispositifs matériels particuliers pour le

mettre en œuvre. On doit notamment ajouter au tableau une colonne de compteurs. Par logiciel, on peut mettre en œuvre des versions dégradées.

Avantages / Inconvénients de la pagination

Avantages :

- Meilleure utilisation de la mémoire physique (programmes implantés par fragments, dans des pages *non-consécutives*).
- Possibilité de ne charger des pages que lorsqu'elles sont référencées (chargement à la demande).
- Indépendance de l'espace virtuel et de la mémoire physique (mémoire virtuelle généralement plus grande).
- Possibilité de ne vider sur disque que des pages modifiées.
- Possibilité de recouvrement dynamique (couplage).

Inconvénients :

- Fragmentation interne (toutes les pages ne sont pas remplies).
- Impossibilité de lier deux (ou plusieurs) procédures liées aux mêmes adresses dans l'espace virtuel.

1.10.3. Autres considérations

1.10.3.1. L'écroulement (Thrashing)

On peut allouer les pages de mémoire physique en nombre égal pour 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 pages proportionnellement aux tailles des programmes. Si un processus est deux fois plus grand qu'un autre, il recevra le double de pages.

Ces techniques d'allocation, utilisées telles quelles, peuvent provoquer un écroulement du système. En effet, ce dernier n'est viable que si les défauts de pages sont contenus au dessous d'une limite relativement basse. Si le nombre de processus est trop grand, l'espace propre à chacun sera insuffisant et ils passeront leur temps à gérer des défauts de pages.

On peut limiter le risque d'écroulement en considérant des abaques de comportement. Le nombre de défauts de pages en fonction du nombre de pages allouées à la forme d'une hyperbole. 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. Si il n'y a plus de pages disponibles et trop de défauts de pages, on devra suspendre un des processus.

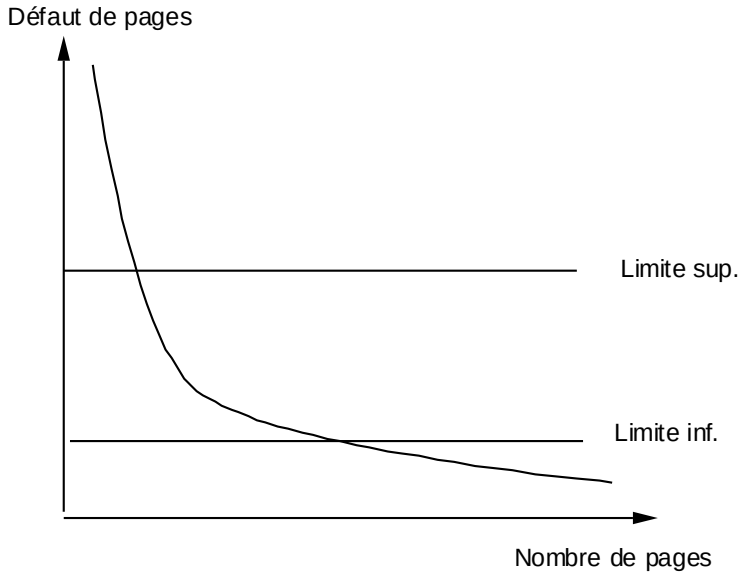

Figure 13

1.10.3.2. L'ensemble de travail

Pour déterminer l'espace viable d'un processus, on utilise le modèle de l'ensemble de travail. L'ensemble de travail est constitué par les zones du processus auxquelles on accède sur un court instant de temps (une dizaine de références à la mémoire). Par exemple, le mise à jour d'un individu dans une base de données, utilisera certaines pages de code et la page correspondant à l'individu.

Des simulations ont montré que cet ensemble de travail est relativement stable à un instant donné. Une allocation optimale pourrait chercher à allouer à chaque processus en

fonctionnement autant de pages que nécessite son espace de travail. Dans ces conditions, les défauts de pages seront provoqués uniquement lors des changements d'espace de travail. En fait ce modèle n'est utilisé que pour la prépagination.

1.10.3.3. L'allocation locale ou globale

Lorsqu'on retire une page de la mémoire centrale, on peut choisir la plus ancienne :

- du point de vue global (la plus ancienne du système);
- du point de vue local (la plus ancienne du processus).

En général, l'allocation globale produit de meilleurs résultats.

1.10.3.4. La prépagination

Lors du lancement d'un processus ou lors de sa reprise après une suspension, on provoque obligatoirement un certain nombre de défauts de pages. On peut essayer de les limiter en enregistrant, par exemple, l'ensemble de travail avant une suspension. On peut aussi essayer de le deviner. Par exemple, au lancement d'un programme, les premières pages de codes seront vraisemblablement exécutées.

1.10.3.5. Le 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 et la réexécuter. On devra alors lui donner les moyens de déterminer l'adresse du premier octet et éventuellement d'annuler certaines incrémentations.

Ce retour sur instruction n'est possible qu'avec l'aide du matériel. Par exemple, le 68010 possède un registre qui mémorise les adresses des premières instructions et les incréments. Il rend possible la pagination. Le 68000 n'en dispose pas et ne peut pas le faire.

1.11. Un exemple de gestion de la mémoire sur un micro-processeur : le 386

L'examen du processeur 386 est intéressant car il combine des techniques de mémoire paginée à des techniques de segmentation[4]. Il dispose de 16 k segments indépendants ayant une capacité allant jusqu'à 1 milliard de mots de 32 bits.

La mémoire virtuelle est fondée sur deux tables :

- la table locale de descripteurs, (LDT), propre à chaque programme. Elle contient ses segments avec notamment les segments de code, de pile et de données;
- la table globale de descripteurs, (GDT), unique pour tout le système et partagée par tous les processus. Elle contient les segments du système et notamment ceux du noyau.

Le 386 dispose de six registres de segments. Pour accéder à un segment particulier, par exemple le segment de code d'un processus, il charge un sélecteur dans l'un des registres. Ce sélecteur correspond à un indice dans l'une des deux tables. Chaque entrée de ces tables contient l'adresse de base du segment, l'adresse de limite ainsi que certains autres champs. Grâce à l'adresse de base, le microprocesseur peut convertir les décalages générés en adresses linéaires de 32 bits.

Le 386 dispose d'une pagination optionnelle avec des pages de 4 ko. Lorsqu'elle est activée, l'adresse précédente est interprétée comme une adresse virtuelle et elle est convertie en adresse réelle. La pagination se fait sur deux niveaux. Pour ceci l'adresse générée est interprétée en fonction de 3 champs :

10 bits	10 bits	12 bits
Dir	Page	Offset

Figure 14

[4] C'est par ailleurs le processeur 32 bits le plus répandu.

Dir est un indice dans un tableau contenant un pointeur sur une table de pages. Dans cette table Page est un autre indice contenant lui aussi un pointeur (double indirection) sur une page réelle. À l'intérieur de cette page réelle, Offset est l'adresse de l'élément recherché.

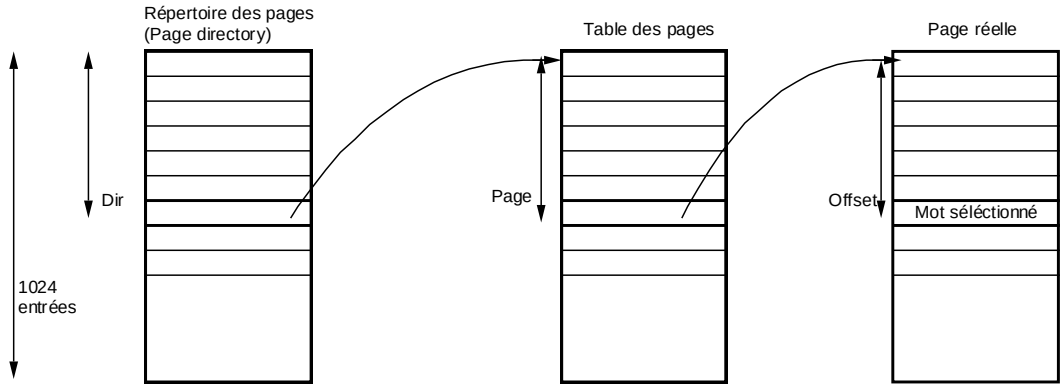

Figure 15

1.12. Appels système Unix

fork() entraîne une allocation de mémoire

exec() entraîne une modification de mémoire

wait() entraîne une libération de mémoire

Les fichiers exécutables Unix contiennent notamment le texte et les données globales initialisées; les données non initialisées (BSS) sont allouées au chargement. En mémoire, les segments de code ne sont pas modifiables dans la plupart des langages compilés et sont partageables sur la plupart des systèmes modernes. Cette dernière caractéristique optimise l'occupation de la mémoire. En revanche, les autres segments (données et pile) sont propres à chaque processus.

Les segments de données et de pile sont continuellement modifiés au cours de l'exécution et leur taille peuvent varier. Les appels modifiant ces tailles sont les suivants :

int brk(caddr_t addr) permet de déplacer la frontière de la zone de données à addr. Elle peut être utilisée par malloc. Elle rend -1 si échec.

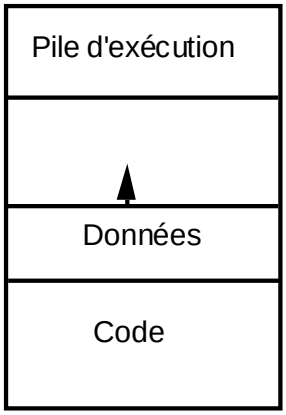

Figure 16

caddr_t sbrk(int incr) est similaire à brk. Au lieu de fixer l'adresse, elle étend la zone de données de incr.

void * malloc(int incr) alloue un espace contigu de taille incr dans la zone de données. Elle rend un pointeur sur la zone et NULL en cas d'échec. Malloc est une fonction assez primitive d'allocation de mémoire. Elle utilise l'algorithme du premier ajustement et ne recompacte pas les blocs libres de la zone de données. Elle entraîne de la fragmentation. Dans l'ouvrage sur le langage C de Kernighan et Ritchie, on trouve le code d'implantation de malloc. Il est intéressant à lire.

void * calloc(int n_obj, int size) rend un tableau.

void realloc(void *p, int size) réétend la zone allouée à une variable à size. Elle rend NULL si échec.

void free(void *p) libère la zone pointée par p.

2.13. La mémoire du DOS

Le système MS-DOS dispose d'un modèle de mémoire nettement plus compliqué que celui d'Unix. Il faut interpréter cette complexité comme un défaut : la simplicité d'Unix 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 8086) qu'elle exploite jusqu'à la corde. L'architecture originale sur laquelle le 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.

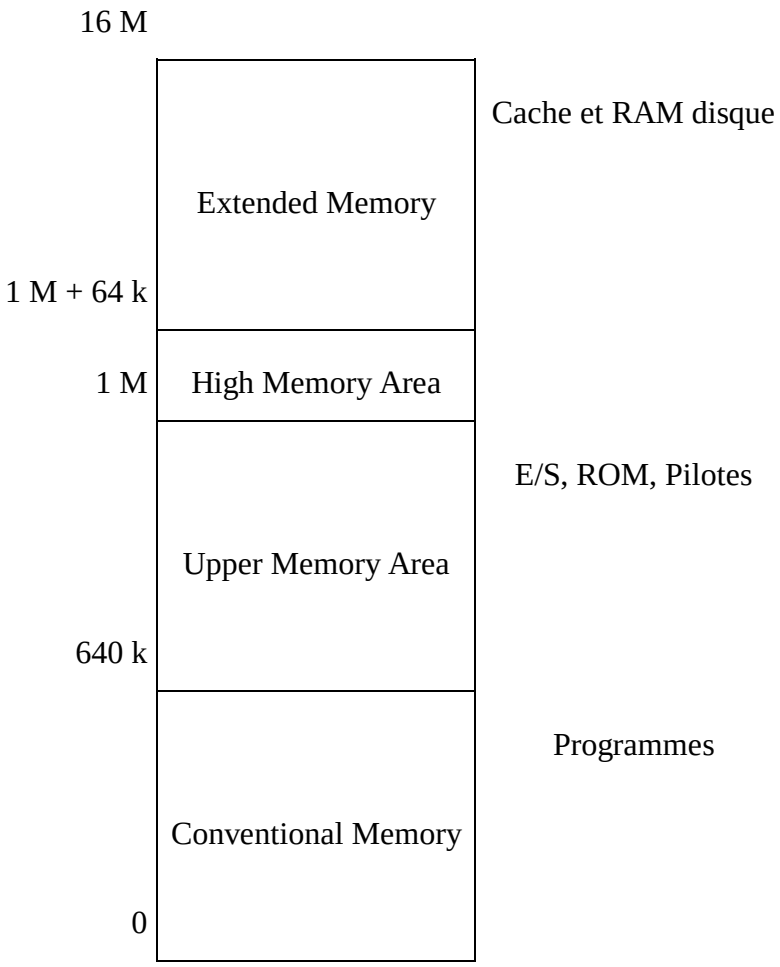

Figure 17

L'adressage du DOS est en fait restreint à 640 ko, la partie supérieure (Upper Memory Area) étant réservée à divers programmes de pilotage d'entrées-sorties.

Le modèle de mémoire du DOS permet des adresses au delà de 1 Mo. En prenant comme base, la valeur 0xFFFF, il est possible d'aller jusqu'à 1 Mo + 64 ko. Ceci

correspond à la zone de mémoire haute (High Memory Area). L'exploitation de cette mémoire dépend de la manière (variable) dont est câblée la ligne d'adresse 21.

Les processeurs Intel actuels permettent des adressages étendus. Le 286 autorise 16 Mo; les 386 et suivants, 4 Go. Pour être compatible avec les processeurs 8086, Intel a fournit un mode de fonctionnement qui permet d'exploiter les machines actuelles avec l'ancienne gestion de mémoire : le mode « réel ». Ce mode limite la taille à 640 ko. Cependant des modifications de MS-DOS permettent d'exploiter la mémoire étendue pour des disques RAM, par exemple.

1.14 La mémoire de Windows

Les premières versions de Windows ont beaucoup souffert de l'architecture du 8086 et de l'héritage du DOS. Cependant la gestion de mémoire de Windows devient maintenant plus facile avec l'interface de programmation Windows. En effet, avec cette API, le programmeur voit la mémoire comme une zone d'adresses plate.

Windows réalise automatiquement le recompactage des données de la mémoire par un mécanisme de récupération des zones libres - en anglais le *garbage collecting*. Ceci est absolument nécessaire car le fenêtrage nécessite de multiples créations et destructions de mémoire et sans ce système, la mémoire se trouverait très vite en « mille morceaux ».

Après une opération de recompactage, dans le meilleur des cas, les blocs libres ne forment plus qu'un seul et grand segment. Dans la réalité, le compactage n'est souvent pas total, mais il permet une allocation ultérieure plus facile. Le recompactage à lieu régulièrement à l'initiative du système d'exploitation ou bien pour allouer de la mémoire qui à un moment donné ferait défaut. La récupération des zones libres existait déjà dans certains langages de programmation tels que le Prolog ou le Lisp et dans des systèmes d'exploitation tels que celui du Macintosh.

L'allocation de mémoire utilise des pointeurs de pointeurs - des *handles*. Ces handles (HGLOBAL) sont conservés dans un segment de mémoire : le BurgerMaster[5]. Le handle qui référence le bloc mémoire sera constant, en revanche, le pointeur sur la mémoire réelle lui sera variable au gré du système d'exploitation. Pour manipuler les données, on devra verrouiller le segment et utilisateur les pointeurs qui alors seront constants.

HGLOBAL GlobalAlloc(UNIT fuFlags, DWORD cbBytes) alloue un segment de longueur cbBytes avec les options fuFlags. Parmi ces options GMEM_MOVEABLE signale que le segment est relogeable, GMEM_FIXED signale que le segment est fixe. Si il y un échec, la fonction rend NULL et on en peut connaître la raison par la fonction GetLastError.

LPVOID GlobalLock(HGLOBAL) verrouille un segment et rend un pointeur. Si il y un échec, la fonction rend NULL.

BOOL GlobalUnlock(HGLOBAL) déverrouille un segment. En fait elle décrémente le compteur des verrous sur le segment. Si ce compteur passe à zéro, la valeur rendue est FALSE, sinon TRUE.

HGLOBAL GlobalFree(HGLOBAL) libère un segment. En cas de succès, la fonction rend NULL.

---

[5] C'était le restaurant préféré des développeurs de Windows, paraît-il.