Système d'exploitation

IV. Allocation de la mémoire

Kévin Perrot

Aix-Marseille Université

2014

Ce cours utilise (entre autres) des supports de Jean-Luc Massat en L3 informatique à Luminy.

Table des matières

1 Le Matériel **3**
1.1 Les registres . . . 3
1.2 Les caches . . . 3
1.3 La mémoire centrale (RAM) . . . 3
1.4 Les disques . . . 4
1.5 Interface avec l'OS . . . 5
1.6 Unités . . . 6

2 Introduction : deux problématiques **6**
2.1 Allocation de la mémoire secondaire (ROM) . . . 6
2.2 Allocation de la mémoire principale (RAM) . . . 6
2.3 Liens entre allocation de la mémoire secondaire/principale . . . 7
2.4 Fragmentation et compactage . . . 7

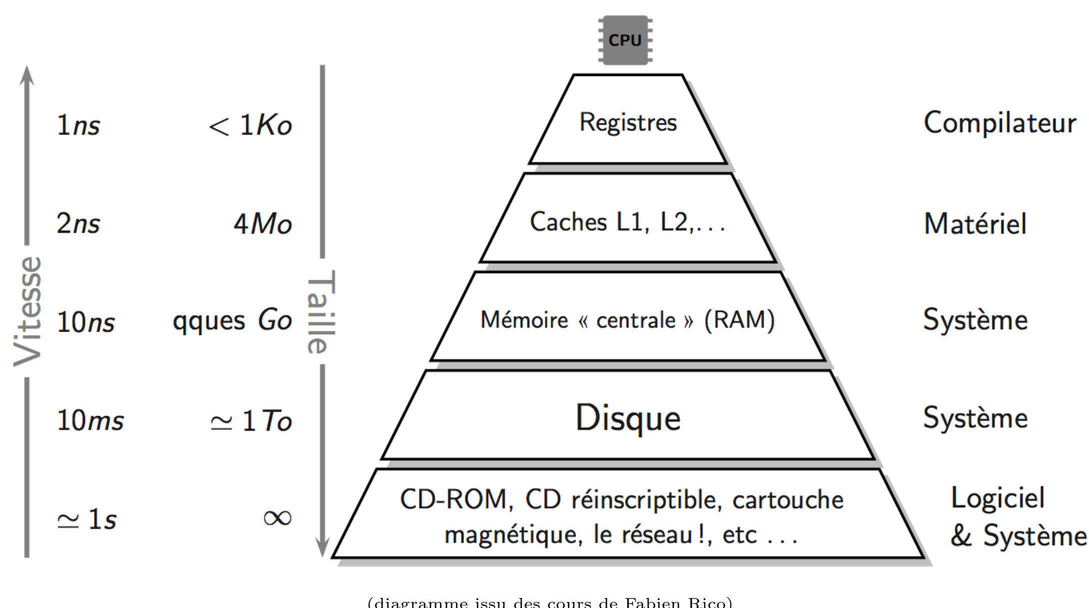
3 Allocation de la mémoire secondaire (ROM) **9**
3.1 Fichiers . . . 9
3.1.1 Meta-données . . . 9
3.1.2 Arborescence et chemin d'accès . . . 10
3.1.3 Organisation des fichiers sur le disque : norme FHS . . . 11
3.2 Systèmes de gestion de fichiers (SGF) . . . 13
3.2.1 Organisation de la mémoire . . . 13
3.2.2 Partitions . . . 13
3.2.3 Blocs . . . 14
3.3 Mécanisme SGF 1 : allocation contiguë . . . 14
3.3.1 Naïf . . . 14
3.3.2 Buddy system . . . 14
3.4 Mécanisme SGF 2 : blocs chainés . . . 15
3.5 Mécanisme SGF 3 : table d'allocation (table d'inode) . . . 16
3.6 Limites des SGF . . . 17

4 Allocation de la mémoire principale (RAM) **17**
4.1 Principe de localité 18
4.2 Mémoire logique 19
4.3 Liaison 19
4.3.1 Descripteur de fichier 20
4.4 Cohérence 21
4.5 Systèmes multi-utilisateurs 21
4.6 Critères d'évaluation 21
4.7 Organisation de la mémoire 21
4.8 Allocation de la mémoire sans réimplantation 22
4.8.1 Système à partition unique (va-et-vient simple) 22
4.8.2 Partition fixe de la mémoire 23
4.9 Système à partitions variables 24
4.9.1 Réimplantation dynamique par registre de base 24
4.9.2 Algorithmes de gestion de la mémoire par zones 24
4.9.2.1 Représentation des partitions 25
4.9.2.2 Algorithmes de sélection 25
4.9.2.3 Libération d'une partition 25
4.9.3 Fragmentation 26
4.10 Mémoire paginée 26
4.10.1 Pagination d'une mémoire contiguë 26
4.10.2 Comportement des processus en mémoire paginée 28
4.10.3 Mémoire associative et pagination 29
4.10.4 Partage et protection de l'information 30
4.11 Mémoire segmentée 32
4.11.1 Principe de la segmentation 32
4.11.2 Pagination d'une mémoire segmentée 33
4.11.3 Partage de segments 33
4.12 Mémoire virtuelle paginée 34
4.12.1 Pagination simple d'une mémoire virtuelle 35
4.12.2 Pagination à deux niveaux d'une mémoire virtuelle 37
4.12.3 Mécanisme du défaut de page 38
4.12.4 Comportement des programmes en mémoire virtuelle 38
4.12.5 Gestion d'une mémoire virtuelle paginée 40
4.12.5.1 Paramètres d'une politique d'allocation 40
4.12.5.2 Algorithmes de remplacement 40
4.12.6 Écroulement d'un système de mémoire virtuelle paginée 43
4.12.6.1 Apparition de l'écroulement 43
4.12.6.2 Algorithme fondé sur l'ensemble de travail 44
4.12.6.3 Algorithme fondé sur la mesure du taux de défaut de page 45

5 Remarque de bonne conduite **45**

1 Le Matériel

Une machine comporte différents types de mémoire, de la plus rapide/coûteuse/petite (les deux premiers adjectifs impliquant le troisième) à la plus lente/bon marché/grande.



(diagramme issu des cours de Fabien Rico)

Figure 1: La mémoire est hiérarchisée et se représente naturelle sous la forme d'une *pyramide mémoire*. Dans la colonne de droite sont indiquées les entités gérant chaque type de mémoire.

1.1 Les registres

Les *registres* sont rattachés au processeur : ce sont les seuls éléments de mémoire qu'il est autorisé à lire. Les instructions exécutées en fin de chaîne par celui-ci sont du type

```
ADD R1 R2 R3
```

par exemple ici pour additionner les contenus de `R1` plus `R2`, et placer le résultat dans `R3`.

C'est le *compilateur* qui gère l'allocation des registres : vous écrivez un programme dans un langage « haut niveau » (par exemple : `variable_3 = variable_1 + variable_2;`), et le compilateur le transforme en une suite d'instructions pour le processeur (du genre de l'instruction `ADD` ci-dessus).

1.2 Les caches

Pour ne pas s'encombrer l'esprit avec trop de détails, on pourra dire que les *caches* ont les mêmes buts et engendrent les même problématiques que la mémoire centrale (RAM), à un niveau plus fin (ce sont des mémoires encore plus rapides/coûteuses/petites).

1.3 La mémoire centrale (RAM)

La *mémoire centrale*, ou *mémoire principale*, ou *RAM*, est la mémoire de base de l'ordinateur, et celle où sont (et doivent être) stockées les données en cours de traitement.

Vous exécutez un programme ? Celui-ci doit être placé sur la RAM. Vous lisez un fichier ? Celui-ci doit être placé sur la RAM, *etc.*

La RAM est une mémoire volatile : elle s'efface lorsqu'il n'y a plus de courant.

La technologie majoritaire actuellement s'appelle *Dynamic Random-Access Mémory* (DRAM). Chaque bit d'information est sauvegardé par un condensateur : condensateur chargé = 1, condensateur déchargé = 0. Comme les transistors qui sont ensuite branchés sur les condensateurs fuient toujours un peu (s'ils sont sensés bloquer le courant du condensateur pour le laisser dans l'état chargé), les condensateurs vont lentement se décharger, jusqu'à changer d'état... pour palier à ce problème, la charge des condensateurs est rafraîchie (remise à jour) périodiquement (de l'ordre de quelques millisecondes), d'où l'adjectif « dynamic ».

1.4 Les disques

Les *disques*, ou *mémoire secondaire*, ou *ROM*, sont des médias de stockage de l'information. Pour qu'un processus (une instance d'exécution d'un programme) utilise un fichier présent en mémoire secondaire, celui-ci doit être copié en mémoire principale.

La ROM est une mémoire non-volatile : elle ne s'efface pas lorsqu'il n'y a plus de courant.

Les disques sont des mémoires magnétiques (un bon hacker a toujours un électro-aimant sous la main pour effacer tous ses disques rapidement en cas d'urgence...), ils sont composés de plateaux rigides en rotation. Chaque plateau est constitué d'un *disque* réalisé généralement en aluminium (il en existe aussi en verre ou céramique), dont les faces sont recouvertes d'une couche magnétique, organisée en *pistes* et *secteurs* (Figure 2), sur laquelle sont stockées les données. Ces données sont écrites en code binaire sur le disque grâce à une tête de lecture/écriture, petite antenne très proche du matériau magnétique. Suivant le courant électrique qui la traverse, cette tête modifie le champ magnétique local pour écrire soit un 1, soit un 0, à la surface du disque. Pour lire, le même matériel est utilisé, mais dans l'autre sens : le mouvement du champ magnétique local engendre aux bornes de la tête un potentiel électrique qui dépend de la valeur précédemment écrite, on peut ainsi lire un 1 ou un 0. Un *disque dur* typique (le *support*) contient un axe central autour duquel les plateaux (*disques*) tournent à une vitesse de rotation constante (Figure 3). Toutes les *têtes de lecture/écriture* sont reliées à une armature qui se déplace à la surface des plateaux, avec une à deux têtes par plateau (une tête par face utilisée). L'armature déplace les têtes radialement à travers les plateaux pendant qu'ils tournent, permettant ainsi d'accéder à la totalité de leur surface. La lecture est réalisée par *bloc* : l'ensemble des *pistes* de même numéro de toutes les faces s'appelle un *cylindre*, et un *bloc* est un sous-ensemble de *secteur(s)* de même(s) numéro(s) (un ou plusieurs secteurs) à l'intérieur d'un *cylindre* (Figure 3). L'électronique associée contrôle le mouvement de l'armature ainsi que la rotation des plateaux, et réalise les lectures et les écritures suivant les requêtes reçues. Les firmwares des disques durs récents sont capables d'organiser les requêtes de manière à minimiser le temps d'accès aux données (grouper les accès à des données proches pour limiter les déplacements des têtes, *etc*), et donc à maximiser les performances du disque.

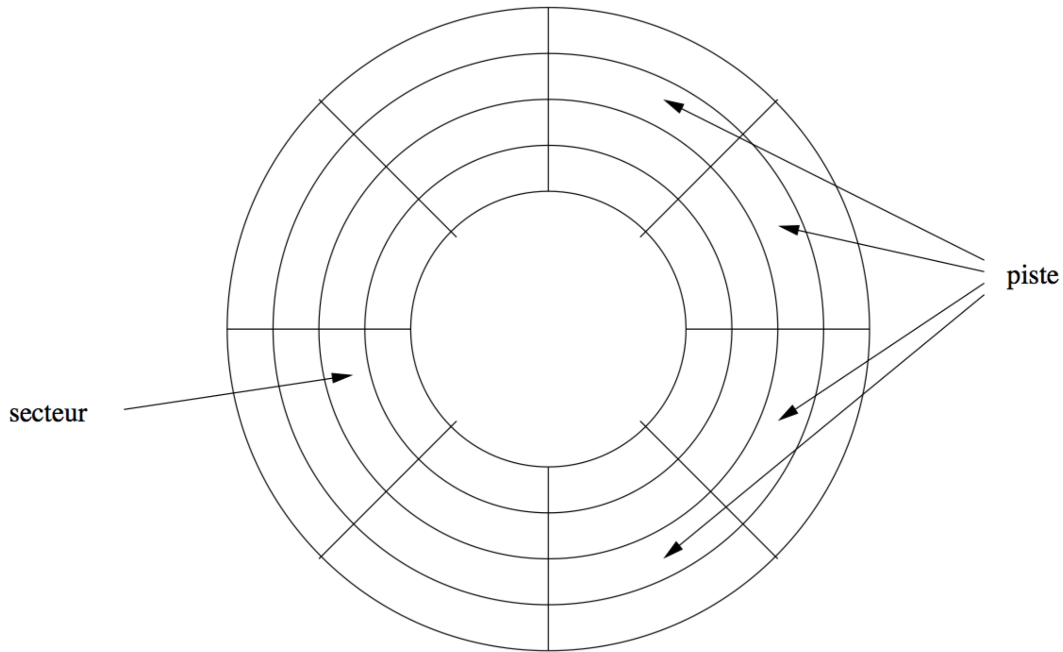


Figure 2: Un *disque* comporte des *pistes* et des *secteurs*, il a généralement deux *faces*.

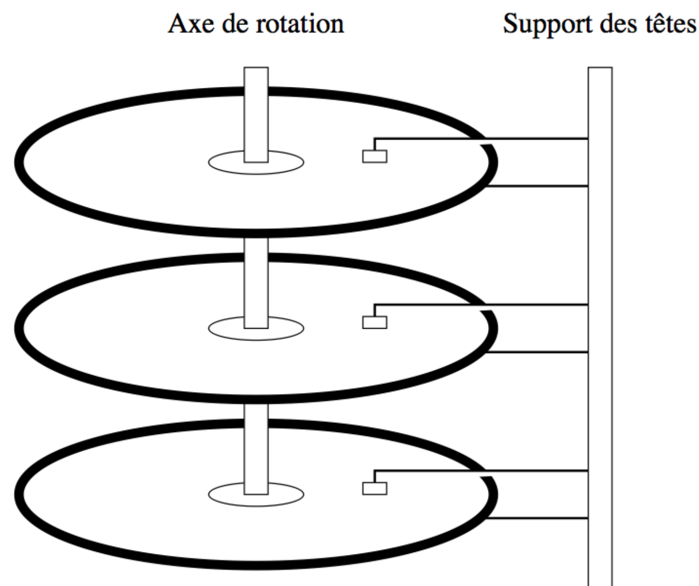


Figure 3: Un *support* est composé de *disques*, un *cylindre* est l'ensemble des *pistes* de même numéro de toutes les *faces*, et un *bloc* (l'unité élémentaire d'E/S) regroupe les *secteurs* de même numéro (un ou plusieurs) à l'intérieur d'un *cylindre*.

1.5 Interface avec l'OS

Nous allons étudier l'utilisation de la RAM et de la ROM. Pour chacune d'elles, on a un *contrôleur mémoire* qui nous donne deux actions possibles (on peut voir cela comme l'*interface* du matériel) :

```
WRITE(adresse,suite_de_bits)
```

`READ(adresse)`.

L'OS doit choisir quoi écrire, où, et quand. C'est l'objet de ce cours.

1.6 Unités

Le *bit* est la plus petite quantité d'information, il peut prendre deux valeurs (0 ou 1). Un *octet* est composé de 8 bits (un *byte* correspond généralement à 8 bits, mais pas toujours). Attention, il y a une distinction entre compter en base 10, et compter en base 2 ! Par exemple, en base 10 on a un *kilo* de $10^3 = 1\ 000$, et en base 2 on a un *kibi* (*kilo binaire*) de $2^{10} = 1\ 024$. Les normes internationales sont les suivantes.

1 kibioctet (kio)	$= 2^{10}$ octets	= 1 024 octets	
1 mébioctet (Mio)	$= 2^{20}$ octets	= 1 024 kio	= 1 048 576 octets
1 gibioctet (Gio)	$= 2^{30}$ octets	= 1 024 Mio	= 1 073 741 824 octets
1 tébioctet (Tio)	$= 2^{40}$ octets	= 1 024 Gio	= 1 099 511 627 776 octets
1 pébioctet (Pio)	$= 2^{50}$ octets	= 1 024 Tio	= 1 125 899 906 842 624 octets
1 exbioctet (Eio)	$= 2^{60}$ octets	= 1 024 Pio	= 1 152 921 504 606 846 976 octets
1 zébioctet (Zio)	$= 2^{70}$ octets	= 1 024 Eio	= 1 180 591 620 717 411 303 424 octets
1 yobioctet (Yio)	$= 2^{80}$ octets	= 1 024 Zio	= 1 208 925 819 614 629 174 706 176 octets
1 kilooctet (ko)	$= 10^{3}$ octets	= 1 000 octets	
1 mégaoctet (Mo)	$= 10^{6}$ octets	= 1 000 ko	= 1 000 000 octets
1 gigaoctet (Go)	$= 10^{9}$ octets	= 1 000 Mo	= 1 000 000 000 octets
1 téraoctet (To)	$= 10^{12}$ octets	= 1 000 Go	= 1 000 000 000 000 octets
1 pétaoctet (Po)	$= 10^{15}$ octets	= 1 000 To	= 1 000 000 000 000 000 octets
1 exaoctet (Eo)	$= 10^{18}$ octets	= 1 000 Po	= 1 000 000 000 000 000 000 octets
1 zettaoctet (Zo)	$= 10^{21}$ octets	= 1 000 Eo	= 1 000 000 000 000 000 000 000 octets
1 yottaoctet (Yo)	$= 10^{24}$ octets	= 1 000 Zo	= 1 000 000 000 000 000 000 000 000 octets

Un ko est presque égale à un kio, mais c'est de moins en moins vrai à mesure que l'ordre de grandeur augmente. La différence reste raisonnable pour les Go comparés aux Gio, et *Windows Vista* se permet même la confusion (la taille est affichée en « Go » alors qu'elle est calculée en Gio).

2 Introduction : deux problématiques

2.1 Allocation de la mémoire secondaire (ROM)

Comment sont stockées les informations sur le(s) disque(s) secondaire(s) ? Le but est d'assurer les fonctionnalités suivantes :

ajout, accès, modification, suppression.

L'allocation de la mémoire secondaire est réalisé par un *système de gestion de fichiers*.

2.2 Allocation de la mémoire principale (RAM)

Une fois le mécanisme d'allocation de la mémoire secondaire mis en place, et quel que soit ce mécanisme, il faut mettre un place un autre mécanisme, d'allocation de la mémoire principale. La raison est simple : le processeur n'a pas directement accès à la

mémoire secondaire, mais uniquement à la RAM[1] : pour être exécuté, un processus (la suite de ses instructions) doit d'abord être placé en mémoire principale ; pour être lu, un fichier doit d'abord être placé en mémoire principale, *etc.*

Si nous avons un grand nombre de processus s'exécutant de façon concourante, alors il est possible d'arriver à une situation où la quantité d'information que nous souhaiterions placer en mémoire principale soit supérieure à la taille de celle-ci. Dans ce cas, on utilisera la ROM pour stocker les informations qui ne tiennent pas en RAM, et il faudra effectuer des « va-et-vient » entre mémoire principale et secondaire pour que les processus en cours d'exécution (actifs) soient placés en mémoire principale. Le rôle de l'allocateur de mémoire principale est d'assurer que tous les processus peuvent accéder à toutes leurs données à tout moment. De nombreuses questions se posent alors : s'il n'y a plus de place, quelles informations enlever ? Doit-on placer tout le code du processus en mémoire principale, où devrait-on faire ce transfert au fur et à mesure ? Est-ce plus clair si l'on souhaite lire un film de 4 Go : doit-il être entièrement en mémoire principale lors de la lecture, ou peut-on le « charger » au fur et à mesure de son déroulement ? Comment faire si je n'ai que 2 Go de RAM ? *etc.*

2.3 Liens entre allocation de la mémoire secondaire/principale

Une fois qu'un système de gestion de fichiers est mis en place pour la ROM, on peut réfléchir à un mécanisme d'allocation de la RAM. Ces deux mécanismes sont *presque* indépendants : quel que soit la façon de gérer la mémoire secondaire, l'allocateur de la mémoire principale fera des appels à l'allocateur de la mémoire secondaire (le système de gestion de fichiers) pour y lire/écrire des informations. Néanmoins, on sera facilement convaincu que certains couples de mécanismes pour la ROM et la RAM vont mieux ensemble que d'autres. Notamment, nous verrons des choses très similaires entre la gestion de la ROM et de la RAM, mais nous adopterons un point de vue technique différent car les objectifs sont différents : la ROM sert à stocker des informations de façon pérenne, tandis que la RAM est volatile et sert de mémoire de travail pour les processus (notamment en tant qu'intermédiaire entre la ROM et le processeur).

2.4 Fragmentation et compactage

Un problème commun à toute allocation de mémoire (ROM et RAM) est la *fragmentation*, qui correspond à une perte d'espace et/ou de temps. On en distingue deux types.

— *Fragmentation externe* : émiettement de la mémoire qui apparait lors des allocations et libérations successives de mémoire. Les zones libres de la mémoires peuvent alors être réparties en « beaucoup de petits morceaux », ce qui entraîne que les prochains gros fichiers seront stockés en plusieurs fragments (dans un ensemble de zones libres non-contiguës, ce qui demandera plus d'adresses physiques à rechercher pour savoir où lire sur les disques, et davantage de déplacements des têtes de lecture pour la ROM).

— *Fragmentation interne* : il y a toujours une quantité minimale de mémoire (une *unité*) que l'OS peut manipuler (quelques ko). Un fichier plus petit que cette quantité minimale sera donc stocké dans une zone plus grande que ce qui est strictement

1. en passant par les caches et registres, mais nous ne verrons pas ces mécanismes « intermédiaires ».

nécessaire. Il en va de même de la dernière unité allouée à un gros fichier, qui n'est pas entièrement « remplie ».

En cas de forte fragmentation externe, on peut aboutir à une situation où aucune zone libre de taille suffisante n'est disponible pour satisfaire une demande, alors que la somme des tailles des zones libres est largement supérieure. Une solution consiste à compacter les zones allouées en les déplaçant vers une extrémité de la mémoire, laissant apparaître ainsi à l'autre extrémité une zone libre de taille égale à la somme des tailles des zones libres avant compactage.

Le *compactage* de la RAM peut s'effectuer de deux façons (pour la ROM, seul un équivalent de la seconde méthode est envisagé) :

— par recopie à l'intérieur de la mémoire principale en utilisant une instruction de type `MOVE`, opération monopolisant le processeur central :

$$\texttt{MOVE}\ \langle\text{adresse départ}\rangle\ \langle\text{adresse arrivée}\rangle\ \langle\text{longueur}\rangle$$

— par recopies successives des zones occupées sur disque puis du disque en mémoire principale, à la place voulue, en utilisant un processeur d'entrée-sortie. L'opération est alors plus longue, mais a le mérite de libérer le processeur central pour poursuivre l'exécution des autres programmes.

La Figure 4 donne un exemple simple de compactage. Elle montre les différentes stratégies de recopie des zones occupées de manière à limiter la quantité de données à déplacer. On passe dans cet exemple de 200 ko déplacés à 50 ko.

Notons que le compactage de la RAM n'est possible que si l'on a un mécanisme dit de *réimplantation dynamique* : les processus dont la zone mémoire allouée a été déplacée doivent prendre en compte ce déplacement (nous verrons de tels mécanismes plus loin : les processus utilisent alors des *adresses virtuelles* que le gestionnaire de mémoire met en correspondance avec les *adresses physiques* sur la RAM). D'autre part, Knuth a montré que lorsque l'algorithme d'allocation ne peut satisfaire une demande, cela intervient alors que le taux de remplissage de la mémoire est tel qu'après compactage, la même situation va à nouveau apparaître très rapidement, obligeant le système à consacrer une grande partie de son temps à effectuer des compactages successifs.

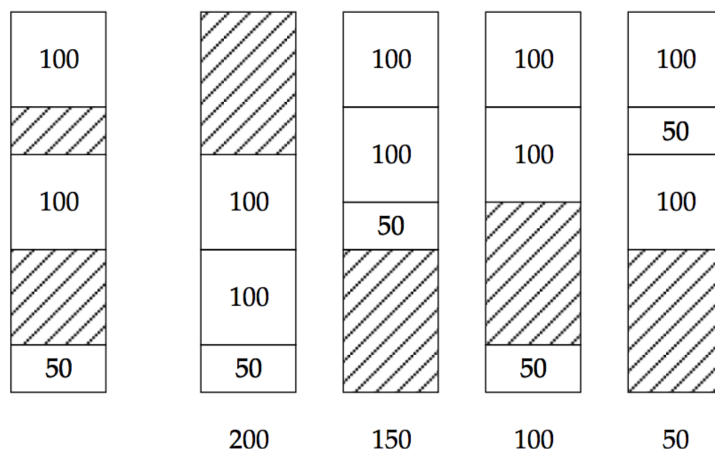


FIGURE 4: Différentes possibilités de compactage de la mémoire, et quantités de données déplacées associées (en ko).

3 Allocation de la mémoire secondaire (ROM)

Nous verrons dans cette section trois mécanismes d'allocation de la mémoire secondaire. Ces mécanismes ont pour but d'assurer les fonctions de stockage et d'accès aux informations :

ajout, accès, modification, suppression.

Ces mécanismes s'appellent des *systèmes de gestion de fichiers* (voir la Section 3.2).

3.1 Fichiers

Dans les systèmes Unix-like, *tout est fichier* ! Un fichier est un ensemble d'informations regroupées en vue de leur utilisation et de leur conservation (une suite de bits). On en a différents types :

— Les fichiers physiques, enregistrés sur le disque dur. Il s'agit du fichier au sens où on l'entend généralement.

— Les répertoires sont des fichiers (noeuds) de l'arborescence pouvant contenir des fichiers ou d'autres répertoires. Un répertoire est une liste de fichiers et répertoires, qui contient à minima un répertoire parent (noté `..`), correspondant au répertoire de plus haut niveau, et un répertoire courant (noté `.`), c'est-à-dire lui-même.

— Les liens sont des fichiers spéciaux permettant d'associer plusieurs noms (liens) à un seul et même fichier. Ce dispositif permet d'avoir plusieurs instances d'un même fichier en plusieurs endroits de l'arborescence sans nécessiter de copie, ce qui permet notamment d'assurer un maximum de cohérence et d'économiser de l'espace disque. Il y a des liens symboliques et des liens physiques.

— Les fichiers virtuels n'ayant pas de réelle existence physique car ils n'existent qu'en mémoire principale (volatile). Ces fichiers, situés notamment dans le répertoire `/proc`, contiennent des informations sur le système (processeur, mémoire, disques durs, processus, *etc*).

— Les fichiers de périphériques, situés dans le répertoire `/dev`, correspondent aux périphériques du système. Les périphériques aussi sont gérés avec des fichiers ! Le plus souvent des *buffers* : fichiers en FIFO où l'on écrit en fin de liste, et lit en début de liste (par exemple pour le clavier : si vous tapez trop vite pour votre OS, la séquences des touches sur lesquelles vous appuyez est enregistrée dans un buffer, et l'OS prend en compte chaque événement dès qu'il peut, dans leur ordre d'arrivée).

3.1.1 Meta-données

Les fichiers comportent des *meta-données* (des données sur les données). Sous les systèmes de type Unix, ces meta-données sont placés dans une structure de données appelée *inode*. Le standard POSIX spécifie qu'un inode doit notamment contenir :

— la taille du fichier en octets,

— l'dentifiant du périphérique contenant le fichier,

— l'identifiant du propriétaire du fichier (UID),

— le numéro d'inode qui identifie (de façon unique) le fichier dans le système de fichiers,

— le mode du fichier qui détermine quel utilisateur peut lire, écrire et exécuter ce fichier,
— des horodatages (timestamps) pour les dates de dernières modification, accès, *etc*,
— une *table d'allocation* : adresses des blocs mémoire où est stocké le contenu.

Le *chemin d'accès* à un fichier nous conduit (par un mécanisme que nous verrons en Section 3.1.2) à l'adresse physique de son inode (ou à une structure similaire suivant le système de gestion de fichiers mis en place), lequel contient les adresses des blocs mémoire où est stocké son contenu (par exemple ici dans la table d'allocation).

3.1.2 Arborescence et chemin d'accès

L'immense majorité des systèmes de fichiers modernes ont une structure *arborescente* : les fichiers sont rangés dans une structure d'arbre où :
— les répertoires sont des noeuds internes,
— les fichiers de données sont des feuilles.

Les répertoires sont codés par une liste linéaire de couples ⟨ nom, adresse ⟩, et on accède aux fichiers par un *chemin d'accès* (Figure 5).

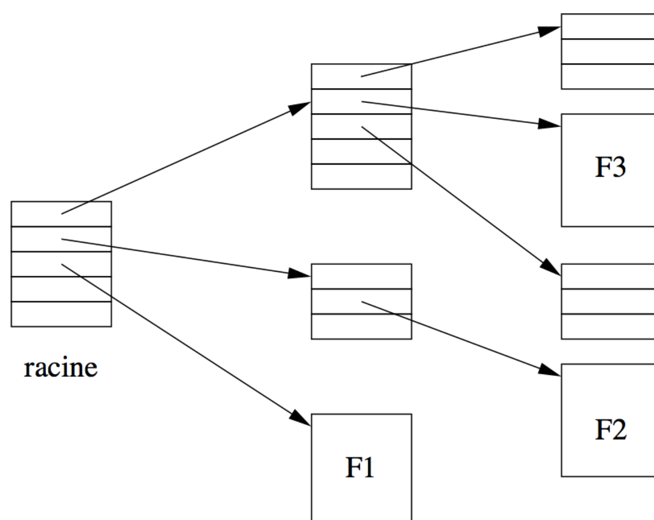


FIGURE 5: Répertoires arborescents, les fichiers ont un *chemin d'accès*.

Il y a deux étapes distinctes lorsqu'on accède (ouvre en lecture, écriture, lecture/écriture) à un fichier à partir de son *chemin d'accès* :

1. un mécanisme de *résolution du chemin d'accès* nous permettra d'obtenir les meta-données du fichier (inode), qui nous indiquent où se trouve son contenu en mémoire secondaire, le fichier peut alors être ouvert : il est copié en RAM ;
   Ce mécanisme est implémenté par un algorithme récursif comme celui-ci :
   (source : `unix.stackexchange.com/questions/146895`)

```
inode find_file(inode where_i_am, string[] remaining_path):
    if remaining_path is empty:
         # Nothing more to look at - we've found the file!
         return where_i_am
    current_item = remaining_path[0]
    rest_of_path = remaining_path[1..]
    for entry in directory_entries(where_i_am):
        if entry.filename == current_item:
            return find_file(entry.inode, rest_of_path)
    return file not found
```

Et l'on trouve le fichier `/home/tim/tim.pdf` avec l'appel :

```
find_file(inode_of_root, ["home", "tim", "tim.pdf"])
```

2. une fois ouvert (et donc copié en RAM), un *descripteur de fichier* nous donne l'adresse où il est stocké en RAM. Chaque processus possède une table de descripteurs de fichiers (stockée en mémoire principale) qui conserve ces informations (voir Section 4.3.1).

3.1.3 Organisation des fichiers sur le disque : norme FHS

(Cette section inspirée du site `http://fr.openclassrooms.com/informatique/cours/le-multiboot-sous-gnu-linux/theorie-les-fichiers-sous-unix`)

Une fois le système de gestion de fichiers établi, certaines conventions (normes) ont été établies pour le placement des fichiers sur le disque : ces normes répondent à la question *quoi placer où ?*

Sous Windows, lorsque votre disque est partitionné (découpé en zones indépendantes, comme s'il y avait plusieurs disques, voir Section 3.2.2), vous avez un disque `C:`, un autre `D:`, *etc.* Chacun de ces disque possède une racine `C:\`, `D:\`, *etc.*

Sous Unix/Linux c'est différent, on a une seule racine `/` où sont placés tous les fichiers, et les différentes partitions sont *montées* et apparaissent comme des dossiers dans le répertoire `/mnt` (prévu à cet effet, mais non représenté sur la Figure 6).

À la racine, on a un certain nombre de dossiers, toujours les mêmes (ou presque) car ils suivent la norme *Filesystem Hierarchy Standard* (FHS), et chacun a un rôle bien défini. La Figure 6 illustre cette organisation, que nous allons succinctement décrire.

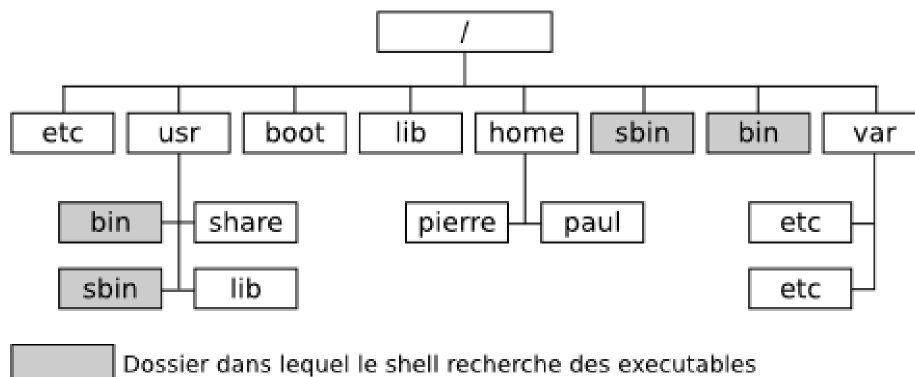


Figure 6: La hiérarchie des répertoires sous Unix/Linux.

- `/etc` : Ce dossier contient les fichiers de configuration du système. Attention : ce sont les paramètres du système uniquement qui sont stockés ici, les paramètres des différents utilisateurs sont stockés dans leurs dossiers personnels respectifs !
- `/dev` : Ce dossier est un peu particulier : il contient tous les périphériques de votre ordinateur. Et oui, sous UNIX/Linux tout est fichier ! Cela veut dire que vos disques durs, votre souris, vos périphériques USB, ... sont vus comme des fichiers, pour permettre une communication facile avec le périphérique.
- `/home` : Ce dossier contient les données des différents utilisateurs du système. Par exemple, `/home/foo` contient les données de l'utilisateur `foo`, ainsi que les préférences des logiciels qu'il utilise. On parle alors du dossier personnel d'un utilisateur.
- `/usr` : Ce dossier contient le gros du système : toutes les applications sont là-dedans. Il est divisé en plusieurs sous-dossiers :
  - `/usr/bin` : ce dossier contient les exécutables des fichiers, c'est-à-dire l'application elle-même.
  - `/usr/share` : ce dossier contient les données des applications. Par exemple, tous les graphismes d'un jeu seront dans `/usr/share/<nom du jeu>`
  - `/usr/lib` : ce dossier contient les bibliothèques de fonctions, c'est-à-dire des applications qui fournissent certaines fonctions communes à plusieurs programmes.
  - `/usr/include` : ce dossier contient les en-têtes des librairies. Ces fichiers ne sont utiles que quand on programme.

  Cette organisation peut sembler étrange pour un habitué de Windows car tous les exécutables sont ensembles, *etc.* Mais en fait, ça simplifie grandement la vie lorsqu'on utilise la console : il suffit que le shell recherche à un seul endroit (en fait à quelques endroits, comme nous le verrons plus loin) pour trouver l'exécutable demandé.
- `/bin` : Ce dossier est un peu comme `/usr/bin`, sauf qu'il ne contient pas de grosses applications, il contient seulement les programmes les plus souvent utilisés : le shell, les commandes principales (`ls`, `ps`, `pwd`, ...) et quelques autres trucs.
- `/sbin` et `/usr/sbin` : Ces deux dossiers contiennent les exécutables des applications qui permettent de modifier la configuration du système : par exemple `ifconfig`, qui permet de configurer la carte réseau, ou `fsck` qui permet de vérifier et de réparer une partition.
- `/boot` : Ce dossier contient le noyau Linux, ainsi que le fichier de configuration du chargeur d'amorçage.
- `/lib` : Ce dossier contient, comme `/usr/lib`, des bibliothèques de fonctions, sauf que celles-ci sont plus bas niveau, c'est-à-dire qu'elles contiennent des fonctions plus « basiques » et utilisées par plus de programmes. Ce dossier contient également les modules du noyau Linux, c'est-à-dire les *pilotes de périphériques*, pour parler en termes Windowsiens.
- `/var` : Ce dossier contient les fichiers de données créés par les services (par exemple les logs systèmes, ou les données d'une BDD MySQL). Il contient plusieurs sous-dossiers :

— `/var/log` : ce dossier contient les logs du système et des applications, c'est-à-dire des fichiers dans lesquels les applications enregistrent leur activité, pour permettre de faire des statistiques ou de détecter des problèmes.
— `/var/lib` : ce dossier contient les données des programmes. Par exemple, c'est là que sont stockées les données de MySQL. C'est aussi là que sont stockés les listes de paquets utilisées par votre gestionnaire de paquets.
— `/var/www` : pour un serveur web, ce dossier contient souvent les données du site web hébergé.

Les fichiers cachés sont des fichiers qui, par défaut, n'apparaissent pas dans votre explorateur de fichier. Ces fichiers sont simplement identifiés par le fait qu'ils commencent par un point, par exemple `/.bashrc`. En général les logiciels stockent leur configuration dans des fichiers cachés, pour ne pas vous encombrer. Par exemple, allez dans votre dossier personnel et affichez les fichiers cachés (commande `ls -a`). Vous devriez en voir beaucoup : chaque application a un fichier de configuration, ou même un dossier entier caché qui contient les différents fichiers que le programme utilise.

La variable (d'environnement) `PATH` indique à l'interprète de commande dans quels répertoire chercher les fichiers exécutables en réponse aux commandes entrées par un utilisateur. `echo $PATH` renvoie quelque chose du genre

```
/bin:/usr/bin:/usr/local/bin:/home/kevin/bin
```

3.2 Systèmes de gestion de fichiers (SGF)

Nous allons maintenant voir différentes façon d'organiser physiquement les fichiers sur un disque afin d'assurer les fonctions de stockage et d'accès à tous ces fichiers : différents *systèmes de fichiers* ou *systèmes de gestion de fichiers (SGF)* (la liste proposée n'est pas du tout exhaustive). Les noms des systèmes de gestion de fichiers sont par exemple FAT, NTFS, Ext3, Ext4, *etc* (les deux premiers pour Windows, les deux derniers pour Linux).

3.2.1 Organisation de la mémoire

On distingue deux grandes familles de SGF (Figure 7) :
— *implantation contiguë* : un fichier est stocké à des adresses physiques contiguës.
— *implantation non-contiguë* : un fichier peut être stocké à des adresses physiques non-contiguës.

implantation contiguë	implantation non-contiguë
Naïf (3.3.1)	FAT (3.4)
Buddy system (3.3.2)	Ext2 (3.5)

Figure 7: Deux familles de SGF (numéro de la section).

3.2.2 Partitions

Un disque dur est divisé en *partitions* (ou *partitions physiques*), et chaque partition contient un système de fichier autonome. Un disque « non-partitionné » comprend en fait une seule partition (attention, plus loin le mot *partition* prendra un sens étendu).

3.2.3 Blocs

La gestion du support (de la ROM) consiste en l'allocation et la libération de *blocs* ou *zones* (ensemble de blocs contigus). Le support est caractérisé par

— l'ensemble des blocs libres,

— l'ensemble des blocs occupés,

— l'ensemble des blocs défectueux.

Chacun de ces ensembles est constitué d'une ou plusieurs zones, ils peuvent par exemple être représentés par

— un chaînage des blocs (un chaînage pour chaque zone de l'ensemble),

— un chaînage des blocs d'index (premiers blocs de chaque zone de l'ensemble),

— une table de bits B telle que $B_i = 1$ sis le bloc i fait partie de l'ensemble. Pour un disque de 1 Gio et un bloc de 4 kio, cette table mesure

$$\left(\frac{2^{10} \times 2^{10}}{2^2}\right) \times \frac{1}{2^{10} \times 2^3} = \frac{2^{20}}{2^{15}} = 32 \text{ kio.}$$

Notons que la taille des blocs dépend de la taille des partitions et du SGF mis en place, mais peut généralement être choisi par l'utilisateur (les choix standards oscillent entre 1 et 4 kio, qui sont bien sûr des multiples de la taille d'un secteur).

3.3 Mécanisme SGF 1 : allocation contiguë

Une première solution est l'*implantation contiguë* : un fichier est stocké à des adresses physiques contiguës.

3.3.1 Naïf

Une unique table d'allocation de fichiers (stockée au début de la mémoire) contient seulement une entrée par fichier, avec l'adresse du bloc de début et la taille du fichier. Une liste chaînée des blocs d'index des zones libres (chaque maillon contenant la taille de la zone) permet de répondre à une demande d'allocation d'une zone de taille n par un des choix suivants :

— First-fit : alloue le premier trou suffisamment grand,

— Best-fit : alloue le plus petit trou suffisamment grand,

— Worst-fit : alloue le trou le plus grand.

First-fit est souvent le choix retenu car il est rapide et a des performances raisonnables. Cette méthode est simple, mais a le grave défaut d'engendrer une importante fragmentation externe, et le compactage prend beaucoup de temps. En plus, il faut déclarer la taille du fichier au moment de sa création (imaginez devoir faire cela quand vous créez un document texte...).

3.3.2 Buddy system

Une autre façon de réaliser cette implantation contiguë est le *buddy system* ou « système du compagnon ». On ne manipule que des zones (libre/occupée/défectueuse) dont la taille est une puissance de 2 : on les découpe en deux, on les rassemble par deux, mais leur

taille reste une puissance de 2 (avec un minimum, par exemple 1 ko). Ces zones sont classées par taille (on peut imaginer le faire avec des tableaux de listes stockés quelque part : dans la case i du premier tableau il y a la liste des zones de taille 2^i, *etc*).

- Au début, nous avons une unique zone libre de taille 2^n.
- Lors d'une demande d'allocation d'une zone de taille 2^k, s'il n'y en a pas de libre, on demande une zone de taille 2^{k+1} pour la casser en deux zones libres (deux « compagnes ») de taille 2^k.
- Lorsqu'une zone se libère, si sa compagne est libre, on les fusionne.

Un arbre binaire peut est utilisé pour représenter les zones libres et occupées (et défectueuses). Un exemple est donné sur la Figure 8.

Request 70	A	128		256		512
Request 35	A	B	64	256		512
Request 80	A	B	64	C	128	512
Return A	128	B	64	C	128	512
Request 60	128	B	D	C	128	512
Return B	128	64	D	C	128	512
Return D	256			C	128	512
Return C	1024					

FIGURE 8: Buddy system : une ligne par étape de temps, à chaque ligne une nouvelle requête (sur la gauche) est prise en compte.

Le grand défaut de cette méthode est la fragmentation interne : pour stocker un fichier de 515 kio, on lui alloue une zone de 1024 kio, soit une perte de 509 kio. Plutôt que des tailles 1,2,4,8,16,32,64,128,256,512,1024,..., il a aussi été imaginé de créer des zones dont les tailles suivent la suite de Fibonacci : 1,1,2,3,5,8,13,21,34,55,89,144,... ($F_n = F_{n-1}+F_{n-2}$). Pourquoi ? Certainement pour les vertus mathématiques (et aphrodisiaques) du nombre d'or :

$$\frac{1+\sqrt{5}}{2} = \lim_{n\to\infty} \frac{F_n}{F_{n-1}} \;!$$

3.4 Mécanisme SGF 2 : blocs chainés

(cette section est issue du cours de Hanifa Boucheneb)

Ce SGF offre sur une *implantation non-contiguë* : un fichier peut être stocké à des adresses physiques non-contiguës.

Les blocs (on rappel que les blocs sont des unités d'allocation de taille fixe) d'un même fichier sont chaînés sous la forme d'une *liste chaînée* comme sur la Figure 9. Chaque bloc contiendra des données ainsi que l'adresse du bloc suivant, et l'adresse du fichier est l'adresse de son premier bloc. Par exemple, si un bloc comporte 1024 octets et si le numéro d'un bloc se code sur 2 octets, alors 1022 octets sont réservés aux données et 2 octets au chaînage du bloc suivant.

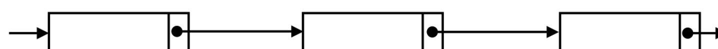


FIGURE 9: Liste chaînée.

Cette méthode rend l'accès aléatoire aux éléments d'un fichier particulièrement inefficace lorsqu'elle est utilisée telle quelle. En effet, pour atteindre un élément sur le bloc n d'un fichier, le système devra parcourir les $n-1$ blocs précédents.

Le système MS-DOS utilise une version améliorée de listes chaînées. Il conserve le premier bloc de chacun des fichiers dans son répertoire (le répertoire auquel le fichier appartient). Il optimise ensuite l'accès des blocs suivants en gardant leurs références (adresses) dans une *Table d'Allocation de Fichiers (File Allocation Table, FAT)* (la taille des FAT, 16 ou 32 bits, correspond à la taille des adresses). Chaque disque dispose d'une FAT et cette dernière possède autant d'entrées qu'il y a de blocs sur le disque. Chaque entrée de la FAT contient le numéro du bloc suivant. Par exemple, dans la table suivante :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x	x	EOF	13	2	9	8	L	4	12	3	E	EOF	EOF	L	...

`x` indique la taille du disque, `L` désigne un bloc libre, `E` un bloc endommagé, et `EOF` désigne la fin du fichier (*End Of File*). Le fichier commençant au bloc 6 sera constitué des blocs $6 \rightarrow 8 \rightarrow 4 \rightarrow 2$. Le parcours de la FAT est nettement plus rapide que la chaîne de blocs. Cependant elle doit constamment être tout entière en RAM, afin d'éviter les accès disque pour localiser un bloc.

La fragmentation externe peut apparaître sous deux formes :

1. les fichiers sont fragmentés,
2. les zones libres sont fragmentées.

Malheureusement, en essayant d'empêcher l'une on favorise l'autre... il faut donc trouver un juste milieu, car le compactage est très lent (la fameuse « défragmentation »).

3.5 Mécanisme SGF 3 : table d'allocation (table d'inode)

(cette section est issue du cours de Hanifa Boucheneb)

Ce SGF offre également sur une *implantation non-contiguë* : un fichier peut être stocké à des adresses physiques non-contiguës.

Comme nous l'avons vu plus haut, chaque inode comprend une table d'allocation contenant les adresses des blocs mémoire où est stocké le contenu du fichier. Elle occupe 13 des 64 entrées des inodes. Les 10 premières contiennent les adresses des 10 premiers blocs composant le fichier. Pour les fichiers de plus de 10 blocs, on a recours à des *indirections*. Le bloc numéro 11 contient l'adresse d'un bloc composé lui-même d'adresses de blocs de données. Si les blocs ont une taille de 1024 octets et s'ils sont numérotés sur 4 octets, le bloc numéro 11 pourra désigner jusqu'à 256 blocs. Au total, le fichier utilisant la simple indirection aura alors une taille maximale de 266 blocs. De la même manière, le bloc numéro 12 contient un pointeur à double indirection, et le bloc numéro 13 un pointeur à triple indirection. La Figure 10 montre l'usage des indirections simples, doubles et triples d'un inode.

Un fichier peut avoir une taille maximale de 16 Go quand il utilise des pointeurs à triple indirection.

Ext2 est un système de fichiers courant sous Linux qui utilise ce mécanisme. Ext3 ajoute un journal (une trace des opérations d'écriture tant qu'elle ne sont pas terminées, en vue de garantir l'intégrité et la cohérence des données en cas d'arrêt brutal), et ext4 est une amélioration (sans changement fondamental) de ext3.

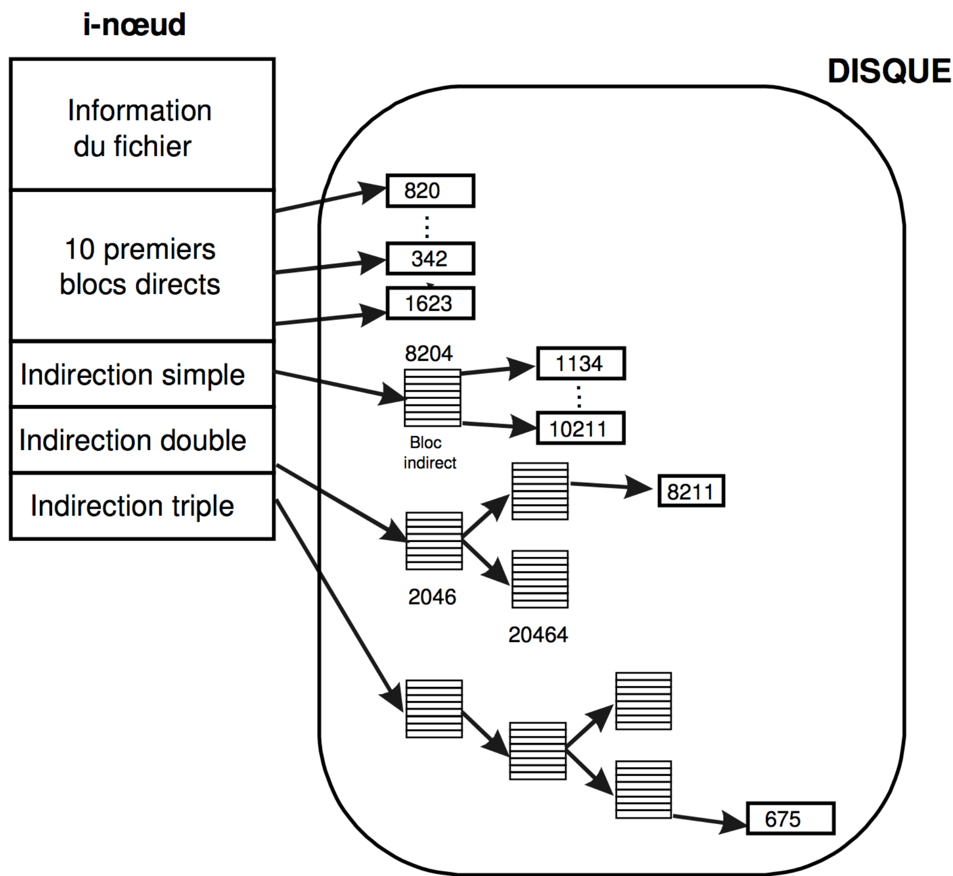


FIGURE 10: Indirections d'un inode.

3.6 Limites des SGF

Les SGF comportent différentes limites : sur la taille maximale du disque qu'il peut gérer, sur la taille maximale d'un fichier, sur la taille maximale du nom d'un fichier, *etc.*

On distingue deux causes à ces limites.

- *Limites architecturales* : elles sont liées à la taille allouée à certaines données. Par exemple, si les adresses sont stockées sur 64 bits, alors on a au plus 2^{64} blocs adressables ! Si la taille d'un fichier est stockée en octets dans un champ de 32 bits, alors elle est limitée à 2^{32} octet ! Si les noms de fichier sont stockés sur 256 octets...
- *Limites d'implémentation* : elles sont plus petites que les limites architecturales, et proviennent d'un choix dans l'implémentation de certains algorithme (en vue de leur optimisation). Il peut aussi y avoir des raisons marketing a mettre des limitations : par exemple Microsoft a limité la FAT pour que les utilisateurs passent au système NTFS.

4 Allocation de la mémoire principale (RAM)

Le processeur n'a pas directement accès à l'information stockée sur la mémoire secondaire, pour la traiter celle-ci soit d'abord être copiée en mémoire principale (que ce soit le code d'un programme à exécuter, ou les fichiers sur lesquels il travail en lecture et ou écriture). L'allocateur de la mémoire principale est en charge de cela : lorsqu'une

information stockée sur le disque est demandée, si ce n'est pas déjà fait il doit la copier en mémoire principale. Mais ce n'est pas tout.

De plus, le programmeur ne manipulera pas directement des *adresses physiques* de la mémoire principale (RAM), mais des *adresses logiques* ou *adresses virtuelles* (qui ne correspondent pas à une adresse physique précise, mais désignent de façon abstraite des cases mémoire allouées au processus, et dont l'implantation physique peut varier (par exemple « la vingt-troisième adresse mémoire qui m'est allouée », « l'adresses où est stocké en RAM le quatrième bloc du fichier `toto.txt` »). Comment assurer la correspondance entre les adresses logiques et physiques ? Cette transformation s'appelle la *liaison*. L'objectif est le suivant : quand le processeur a besoin d'une information, il faut qu'elle soit déjà en mémoire principale, la seule accessible par le processeur (par l'intermédiaire des caches et des registres bien sûr), car le transfert ROM/RAM est très pénalisant (Figure 1 : la ROM est 1 000 000 de fois plus lente (!!) que la RAM).

De nombreux mécanismes sont envisageables, et certains seront plus adaptés que d'autres suivant les usages (les ordinateurs « de tous les jours » utilisent presque tous la pagination, qui offre beaucoup de souplesse).

L'allocation de mémoire doit permettre à un processus l'accès à un objet défini en mémoire logique, en amenant en temps voulu cet objet en mémoire principale (la seule directement adressable). Une politique d'allocation mémoire doit donc apporter une solution aux deux problèmes suivants :

1. réaliser la correspondance entre adresses logiques et adresses physiques ;
2. réaliser la gestion de la mémoire physique principale (allocation des emplacements, transfert de l'information).

Une politique d'allocation de mémoire idéale aurait pour effet d'assurer qu'à tout instant l'information nécessaire à l'exécution de l'instruction en cours soit immédiatement accessible au processeur, donc se trouve en mémoire principale. Cet objectif n'est en général pas atteint : on cherche alors à réduire la probabilité que l'information soit absente de la mémoire lorsqu'elle est nécessaire (*défaut de page* pour les mécanismes de pagination). Le problème se résume alors à deux questions :

- Quand charger un objet en mémoire principale ?
  - lorsqu'on en a besoin (*chargement à la demande*),
  - avant d'en avoir besoin (*pré-chargement*).
- Où charger cet objet ?
  - s'il y a assez de place libre, dans quels emplacements le charger (*placement*) ;
  - sinon, quel(s) objet(s) renvoyer en mémoire secondaire afin de libérer de la place en mémoire principale (*remplacement*).

4.1 Principe de localité

Les politiques de gestion de la mémoire principale (RAM) sont basées sur le très important *principe de localité* : les programmes tendent à utiliser des instructions et des données accédées dans le passé (localité temporelle) ou proches de celles-ci (localité spatiale). Ce principe permet notamment de faire des choix pour le pré-chargement et le remplacement.

4.2 Mémoire logique

La notion de ressource *logique* (ou *virtuelle*) sert à séparer les problèmes d'utilisation d'une ressource particulière des problèmes d'allocation de cette ressource. Un processus ne se soucie pas des adresses des cases mémoires où il a physiquement été placé en RAM : il fait référence à des adresses logiques, par exemple « la trois mille deuxième case mémoire qui m'est allouée ». L'ensemble des emplacements potentiellement accessibles (adressables) lors de l'exécution d'un processus est sa *mémoire logique*. C'est à l'allocateur de faire la correspondance entre adresses logiques et physiques : la *liaison*.

Presque tous les allocateurs de mémoire principale utilisent des adresses virtuelles. Cela a de nombreux autres avantages potentiels, dont voici un exemple. Il est possible de faire croire à un processus que l'on a plus de RAM (virtuelle) que la quantité réelle de RAM (physique) : si l'on a uniquement de la place pour 2 048 adresses (physiquement), on peut toujours proposer aux processus d'en utiliser plus : comme moyen de stockage temporaire on peut utiliser les disques (c'est le rôle de la partition SWAP sous Linux), et c'est à l'allocateur de la mémoire principale de gérer les transferts entre ROM et RAM, pour placer en RAM les informations auxquelles les processus souhaitent accéder, au moment où ils souhaitent y accéder. Cela demande typiquement un mécanisme de *réimplantation dynamique* : au cours de l'exécution, une même information pourra être déplacée au sein de la mémoire principale (par exemple, lors d'une première demande de l'information dont l'adresse virtuelle est 2 500, celle-ci sera placée à l'adresse physique 1 234, et lors d'une seconde demande trois minutes plus tard, entre temps elle aura été placée en ROM pour libérer de la RAM, et elle sera cette fois placée à l'adresse physique 567). Ce mécanisme est à double tranchant : l'OS aurait pu nous dire « je n'ai pas assez de place en RAM, alors j'arrête tel ou tel processus » ; au lieu de cela, il rame [2] un peu à cause des transferts entre ROM et RAM.

Remarque. Le terme *logique* (adresses relatives, dont l'implémentation est variable) sert à faire la distinction avec le *physique* (adresses réelles dans la mémoire). Le terme *virtuelle* fait davantage référence au fait que l'on simule une mémoire plus grande que la mémoire physique, en utilisant la mémoire secondaire.

4.3 Liaison

L'accès d'un processus à une information se traduit par l'accès à un emplacement physique de mémoire principale adressable par ce processus. Les données propres au processus sont schématiquement :

— son code (en langage machine),

— ses variables et leur contenu.

La mise en correspondance entre noms des variables et contenu est la *liaison*. Celle-ci comporte les étapes suivantes :

1. *traduction* (mise en correspondance des objets avec les emplacements mémoire et des noms avec les adresses relatives correspondantes),
2. *édition de lien* (liaison entre programmes traduits séparément, par exemple si vous utilisez une bibliothèque),
3. *chargement* (fixation des adresses, jusque là définies à une translation près).

2. on notera la subtilité du jeu de mots.

Cette mise en correspondance entre organisation de la mémoire logique et implantation de cette mémoire logique en mémoire physique peut être réalisée à trois moments.

— *Compilation* : La correspondance logique/physique est établie une fois pour toutes au moment de la compilation : on génère un code absolu, avec adresses physiques fixes. Cela nécessite d'avoir une vision globale de l'utilisation mémoire à la compilation.
  ▷ C'est une *correspondance fixe* (ou *implantation statique*), c'est le cas dans les *systèmes à partition unique* et dans les *systèmes à partition fixe*.

— *Chargement* : La compilation génère un code translatable, sous forme d'un décalage par rapport à une adresse de base déterminée au chargement du processus (les adresses physiques vont donc varier entre deux exécutions).
  ▷ C'est une *correspondance dynamique* (ou *réimplantation dynamique*). La mémoire est allouée sous forme de zones contiguës (en un seul bloc) appelées des *partitions*, c'est la cas dans les *systèmes à partitions variables*.

— *Exécution* : La liaison est retardée jusqu'à l'exécution, le processus peut alors être déplacé à l'intérieur de la mémoire principale au cours de son exécution. Bien entendu, ces déplacements, opérés par le système, doivent être « transparents » pour le processus.
  ▷ C'est également une *correspondance dynamique* (ou *réimplantation dynamique*). C'est le cas dans les *systèmes paginés* ou *segmentés* qui allouent la mémoire par *pages* (de taille fixe) ou *segments* (de taille variable).

Petit point de terminologie. Nous pouvons avoir :

— des pages (lorsque la mémoire est découpée en zones de même taille),

— des segments ou zones (lorsque la mémoire est découpée en zones de taille variable),

— des partitions (attention, ce terme désignera des parties de la mémoire dont la taille ne varie pas au cours du temps, mais l'emplacement parfois ; ce sont des « zones indivisibles »).

4.3.1 Descripteur de fichier

Dans la norme POSIX, le terme *descripteur de fichier* est utilisé pour parler de l'identifiant abstrait qui permet d'accéder à un fichier en mémoire principale (une fois qu'il y a été copié). Ils sont rangés dans une *table de descripteurs de fichiers* (chaque processus a sa table). On y retrouve les informations suivantes :

— informations d'état (ouvert/fermé, nombre d'utilisateurs),

— informations sur le contenu du fichier (type : fichier de texte, exécutable, répertoire, *etc*),

— statistiques d'utilisation (date de dernière modification, intervalle de temps entre deux utilisations, *etc*),

— informations sur l'implantation physique du fichier (pointeurs vers les adresses physiques où est stocké le fichier).

Ces informations servent par exemple à assurer la *cohérence* des données entre RAM et ROM (Section 4.4), et à décider quelles données enlever de la RAM lorsqu'on veut libérer de l'espace.

4.4 Cohérence

Un problème important qui peut se poser est celui de la *cohérence* : pour modifier un fichier de notre ROM, il faut d'abord le placer en RAM, et ensuite « mettre à jour » cette modification sur la ROM. Ici l'on voit bien qu'à un instant donné, ROM et RAM diffèrent. Il faut donc faire attention à la cohérences entre ces deux mémoires (si un fichier en cours de modification est à nouveau ouvert, si un fichier est modifié par plusieurs programmes, si le disque plante, si la machine plante, *etc*). Nous n'aborderons pas les solutions techniques apportées à ce problème.

4.5 Systèmes multi-utilisateurs

Lorsque les informations appartiennent à plusieurs utilisateurs, deux contraintes supplémentaires apparaissent :

1. réaliser le partage d'information entre ces utilisateurs ;
2. assurer la protection mutuelle d'informations appartenant à des usagers distincts.

Nous n'aborderons pas les solutions techniques apportées à ces contraintes.

4.6 Critères d'évaluation

Plusieurs critères peuvent être utilisés pour imaginer, évaluer et comparer les algorithmes d'allocation de mémoire :

- Critères liés à l'utilisation de la ressource mémoire, mesurée par exemple par le taux de place perdue (ou inutilisable).
- Critères liés à l'accès à l'information, comme le temps moyen d'accès ou le taux de défaut de page.
- Critères plus globaux caractérisant les performances induites par l'allocation de la mémoire : taux d'utilisation de la CPU, temps de réponse, *etc.*

4.7 Organisation de la mémoire

Comme pour la ROM, il y a deux grandes famille.

- Mémoire *contiguë* : les ressources mémoire (fichiers, programmes, variables, tableaux de variables, *etc*) d'un processus sont placées dans une partie de la mémoire « sans trou » : dans les cases mémoires aux adresses `début` à `fin`.
- Mémoire *non-contiguë* : les ressoures mémoire (fichiers, programmes, variables, tableaux de variables, *etc*) d'un processus sont placées dans plusieurs partie de la mémoire, appelées segments (si les parties sont de tailles variables) ou pages (si les parties ont toutes la même taille) : dans les cases mémoires `début_1` à `fin_1`, et `début_2` à `fin_2`, et `début_3` à `fin_3`, ... (si on a `fin_1-début_1` = `fin_2-début_2` = `fin_3-début_3` = ... alors ce sont des pages).

À la fois la mémoire physique et la mémoire virtuelle peuvent être contiguë ou non, ce qui entraîne quatre familles de mécanismes d'allocation de la mémoire (Figure 11).

	mémoire physique contiguë	**mémoire physique non-contiguë**
mémoire logique contiguë	partitions fixes (4.8.2) partition unique (4.8.1) partitions variables (4.9)	mémoire paginée (4.10) mémoire virtuelle paginée (4.12)
mémoire logique non-contiguë	mémoire segmentée (4.11)	mémoire segmentée paginée (4.11)

FIGURE 11: Les différentes organisations de la mémoire (numéro de la section). Les systèmes des partition fixes et unique n'offrent pas de possibilité de réimplantation dynamique, au contraire de tous les autres.

4.8 Allocation de la mémoire sans réimplantation

4.8.1 Système à partition unique (va-et-vient simple)

Dans les *systèmes à partition unique* (aussi appelé *va-et-vient simple* ou *swapping*), une zone fixe de mémoire est réservée aux processus des usagers (voir Figure 12). Les programmes sont conservés sur disque sous forme absolue (sans recours à des adresses logiques). Pour être exécuté, un programme est d'abord amené en mémoire principale dans sa totalité. L'allocation de processeur aux programmes détermine donc les transferts. En cas de réquisition du processeur, le programme en cours doit être sauvegardé sur disque avant le chargement de son successeur.

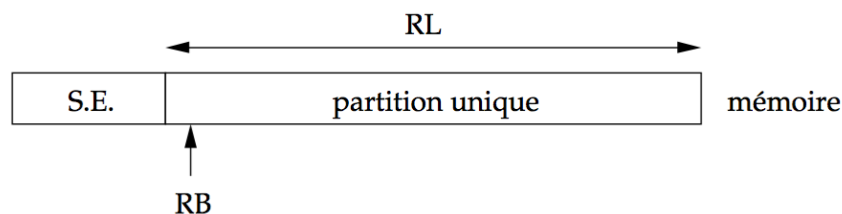


FIGURE 12: Système à partition unique.

Afin d'éviter que des erreurs d'adressage du processus utilisateur ne viennent altérer le S.E. résident, la partition unique peut être délimitée par des registres de la CPU (*registre de base* RB pour le début et *registre limite* RL pour la taille). A chaque accès à une case d'adresse α, la CPU vérifie que ($\text{RB} \leq \alpha < \text{RB} + \text{RL}$). Si ce test échoue, un déroutement pour erreur d'adressage est généré.

Ce schéma a l'avantage de la simplicité. Son principal inconvénient est de laisser la CPU inutilisée pendant la durée des transferts. Il est employé sur des installations de petite taille lorsque les contraintes de temps de réponse sont compatibles avec la durée et la fréquence des transferts. Des améliorations permettent de réduire le volume d'information transférée et donc la perte de temps pour la CPU :

— lorsqu'un programme est sauvegardé sur disque, on ne range que la partie modifiée (en pratique, la zone des données) ;

— l'algorithme de la « peau d'oignon » permet d'épargner des transferts : lorsqu'un programme est recouvert par un autre de taille plus petite, il suffit pour restaurer

le plus gros de recharger la partie recouverte.

Ces améliorations n'apportent néanmoins qu'un gain limité. Il serait préférable de pouvoir exécuter un programme pendant la durée de transfert d'un autre, c'est ce que propose le système de partition fixe (toujours sans réimplantation dynamique).

4.8.2 Partition fixe de la mémoire

Dans un *système à partitions fixes*, la mémoire est partagée de façon statique en un nombre fixe de partitions, les tailles et limites de ces partitions étant définies lors de la génération du système.

Chaque programme est affecté de façon fixe à une partition au moment de la construction de son *image mémoire* par l'étape d'*édition de liens*. Les programmes (sous leur forme « exécutable ») sont conservés sur disque sous forme absolue, et les adresses qui y figurent sont les adresses physiques correspondant à l'implantation de chacun d'eux dans la partition qui lui a été attribuée.

Pendant qu'un programme est transféré (en entrée ou sortie), un autre programme peut être exécuté dans une autre partition ; il faut bien entendu disposer d'un processeur d'entrée/sortie autonome. La Figure 13 schématise l'implantation des programmes et le chronogramme d'activité dans un système à partitions fixes.

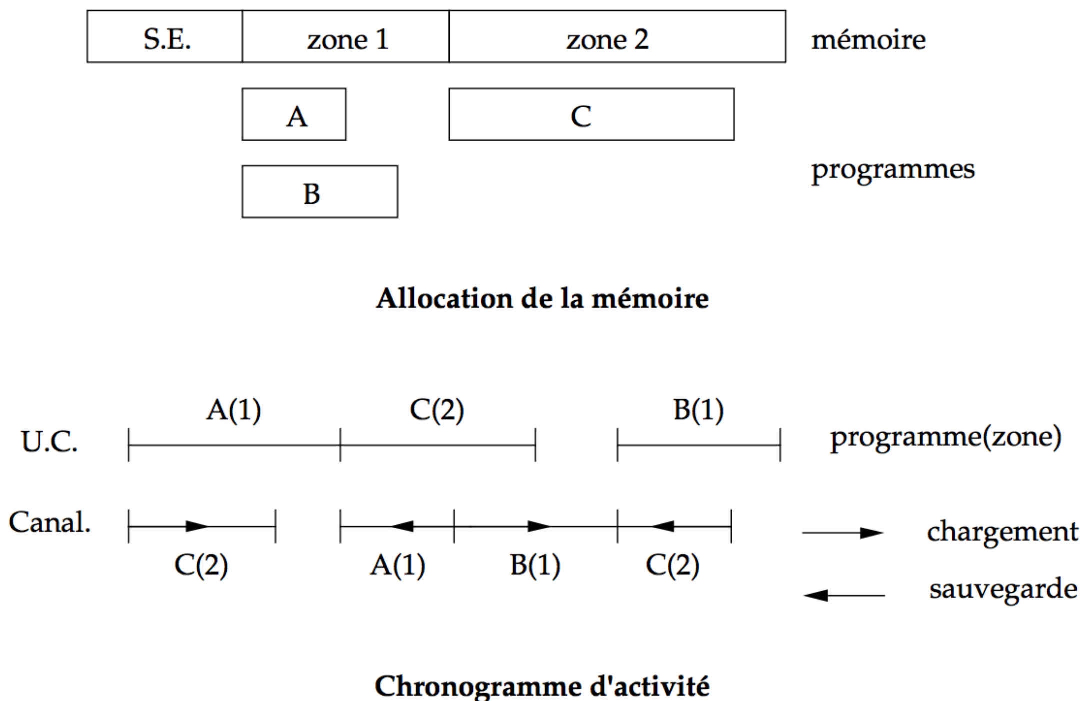


FIGURE 13: Système à partitions fixes.

Les systèmes à partitions fixes sont couramment utilisés sur des petites et moyennes installations où un petit nombre d'usagers « interactifs » coexistent avec un travail de fond. Il est alors possible de définir, au moment de la génération du système, des tailles de partitions adaptées aux différentes classes de programmes.

4.9 Système à partitions variables

Dans un *système à partitions variables*, le découpage en partitions n'est pas fixé une fois pour toutes, mais il est redéfini à chaque début d'exécution d'un processus. En conséquence, le chargement d'un programme (fixation des adresses) ne peut être fait qu'au dernier moment, lorsqu'une place lui est attribuée.

L'allocation de la mémoire par partitions de tailles variables suppose l'existence d'un mécanisme de réimplantation dynamique.

4.9.1 Réimplantation dynamique par registre de base

Le principe que nous allons décrire est simple. Disposant d'un registre particulier ou *registre de base* pour chaque processus, son contenu est systématiquement ajouté à toute adresse engendrée par un processus, le résultat constituant une adresse physique de l'information désignée. Si les adresses d'un programme sont relatives à son début (c-à-d si le programme est implanté à l'adresse logique 0), il suffit que le registre de base soit affecté à son adresse d'implantation en mémoire physique (voir Figure 14).

Dans ces conditions, le programme pourra être chargé en n'importe quel endroit de la mémoire. En particulier, déplacer globalement un programme dont l'exécution est commencée peut s'opérer très facilement, à condition de modifier en conséquence la valeur contenue dans le registre de base. De plus, si programme et données sont atteints par l'intermédiaire de registres distincts, leur déplacement pourra être effectué indépendamment.

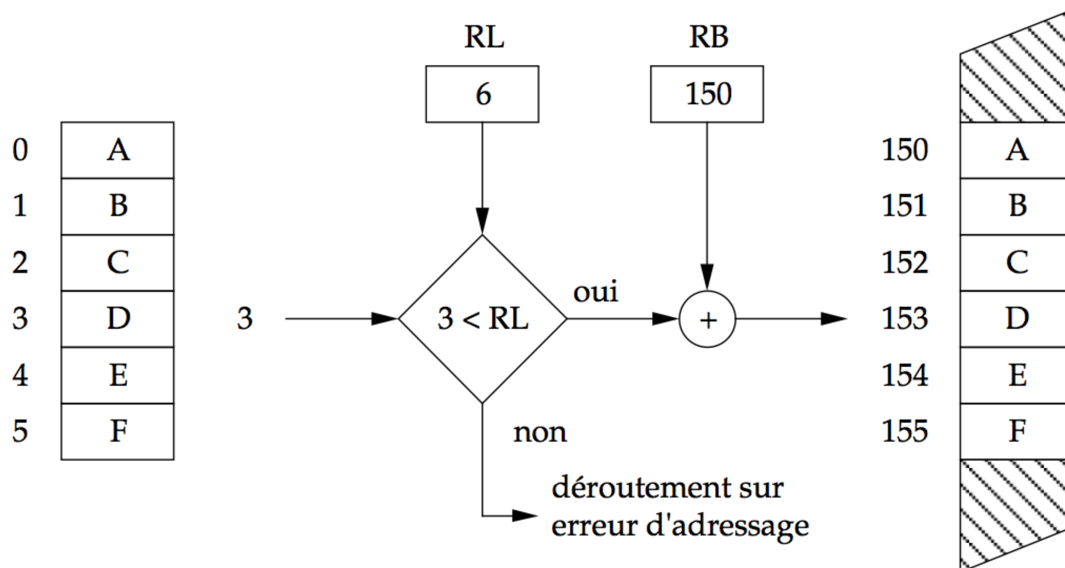


Figure 14: Passage logique / physique par registre de base et registre limite. Sur la gauche est présenté un processus dont le programme est constitué des instructions `A B C D E F`, le registre limite de ce processus est `RL` et son registre de base `RB`.

Reste à décider où placer en mémoire chaque processus à chaque instant, c'est l'objet de la section qui suit.

4.9.2 Algorithmes de gestion de la mémoire par zones

Disposant d'une file constituée par les programmes en attente de traitement, un choix doit être opéré afin de déterminer leur ordre de lancement. Cet ordre pourra être tout sim-

plement celui de la file d'attente, ou dicté par des contraintes de priorités calculées par le système en fonction des demandes de ressources (place mémoire, nombre de périphériques, *etc*), ou du temps d'exécution présumé. En tout état de cause, cet ordre sera aussi fonction de la taille des différentes partitions libres. Il faut auparavant résoudre les problèmes suivants :

— choix d'une représentation des partitions,

— définition des critères de sélection d'une partition libre,

— politique de libération d'une partition occupée,

— décision à prendre lorsqu'aucune partition ne convient.

4.9.2.1 Représentation des partitions

Une partition est définie par sa taille et son adresse de début. En supposant que les tailles demandées sont variables, le nombre de partitions le sera aussi. Il est alors préférable, plutôt que de regrouper les descripteurs dans une table dont la taille va varier, de les situer dans les partitions elles-mêmes et de les chaîner entre eux.

L'ordre du chaînage a une influence sur l'efficacité des algorithmes. On peut choisir l'ordre de libération des partitions, mais le plus souvent, on utilise l'un des deux classements suivants :

— classement par adresses croissantes ou décroissantes,

— classement par tailles croissantes ou décroissantes.

4.9.2.2 Algorithmes de sélection

Une demande étant émise, on connaît la taille requise pour charger le programme du processus demandeur. Le plus souvent, cette demande sera satisfaite grâce à une partition de taille supérieure ; la différence (ou *résidu*) est rattachée à la liste des partitions libres, pour autant que cette différence ne soit pas trop petite. Deux possibilités peuvent être envisagées quant au choix de la partition libre pour satisfaire une demande :

— prendre la première possible, c'est à dire, parcourir la liste jusqu'à ce que l'on en trouve une dont la taille est supérieur ou égale à la demande (*first-fit*) ;

— prendre la partition la plus petite possible, donnant le plus petit résidu (*best-fit*).

L'allocation d'une partition à un processus peut se décomposer en deux phases : recherche de la partition selon l'algorithme choisi, puis placement du résidu dans la liste.

Le classement par tailles croissantes évite de parcourir toute la liste pour trouver la plus petite partition possible (permettant ainsi une implémentation aisée du *best-fit*). Par contre le placement du résidu impose une modification du chaînage.

A l'opposé, le classement par adresses croissantes autorise une gestion rapide des résidus (seule la taille doit être modifiée, le chaînage demeurant inchangé) pour peu que le chargement s'opère au début de la partition. Cette technique est mieux adaptée à l'algorithme du *first-fit*.

4.9.2.3 Libération d'une partition

Trois cas sont à considérer lors de la libération d'une partition :

— la partition libérée est entourée de deux partitions libres,

— la partition libérée est entourée d'une partition libre et d'une partition occupée,

— la partition libérée est entourée de deux partitions allouées.

Chaque fois que cela est possible (deux premiers cas), il est utile de regrouper les partitions libres contiguës afin de réduire la *fragmentation* de la mémoire. Il est évident que le classement par adresses croissantes est alors le plus efficace.

4.9.3 Fragmentation

Il y a un fort risque de fragmentation externe. Par conséquent, une telle forme d'allocation n'est guère adaptée à un système interactif, mais convient mieux lorsque le nombre de zones allouées est faible, et leur temps d'allocation grand.

4.10 Mémoire paginée

Une mémoire paginée est divisée en blocs de taille fixe, ou pages logiques, qui servent d'unités d'allocation. La mémoire physique est elle-même divisée en blocs de même taille appelés pages physiques.

4.10.1 Pagination d'une mémoire contiguë

La Figure 15 représente le schéma général d'une mémoire contiguë paginée. Le rôle de la boîte marquée « Fonction de pagination » est d'établir une correspondance entre les adresses de pages logiques et les adresses de pages physiques de manière à se qu'une page logique puisse être rangée dans une page physique quelconque. Les pages physiques deviennent ainsi des ressources banalisées dont la gestion est plus simple que celle de partitions de taille variable.

Le nombre et la taille d'une page (physique ou logique) sont toujours des puissances de 2 (pour une raison qui sera claire dans le paragraphe qui suit). Notons 2^l la taille (nombre d'emplacements) d'une page (logique ou physique) et 2^n le nombre de pages. Il y a pour l'instant autant de pages logiques que de pages physiques.

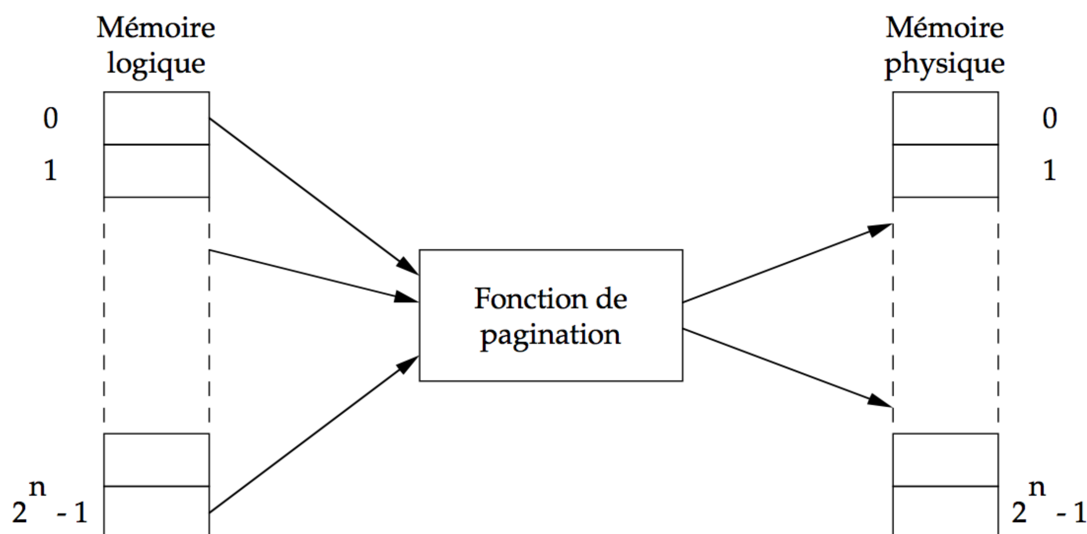


FIGURE 15: Mémoire linéaire paginée.

Une *adresse logique paginée* est alors construite par concaténation d'un numéro de page logique (n bits) et d'un déplacement dans la page (l bits). De même, une adresse

physique est la concaténation d'un numéro de page physique (n bits) et d'un déplacement (l bits). Les tailles de page usuelles vont de 0,5 kio à 32 kio.

Étant donné un numéro de page logique (`npl`), la fonction de pagination permet de trouver le numéro de la page physique (`npp`) qui la contient. Dans un souci d'efficacité, cette fonction est réalisée par un mécanisme matériel.

La réalisation la plus courante de la fonction de pagination utilise une table de pages en mémoire, indexée par un numéro de page logique (table `desc` de la Figure 16). Lors d'un accès à la mémoire, la correspondance adresse logique/adresse physique (qui est une opération matérielle), est mise en œuvre comme suit :

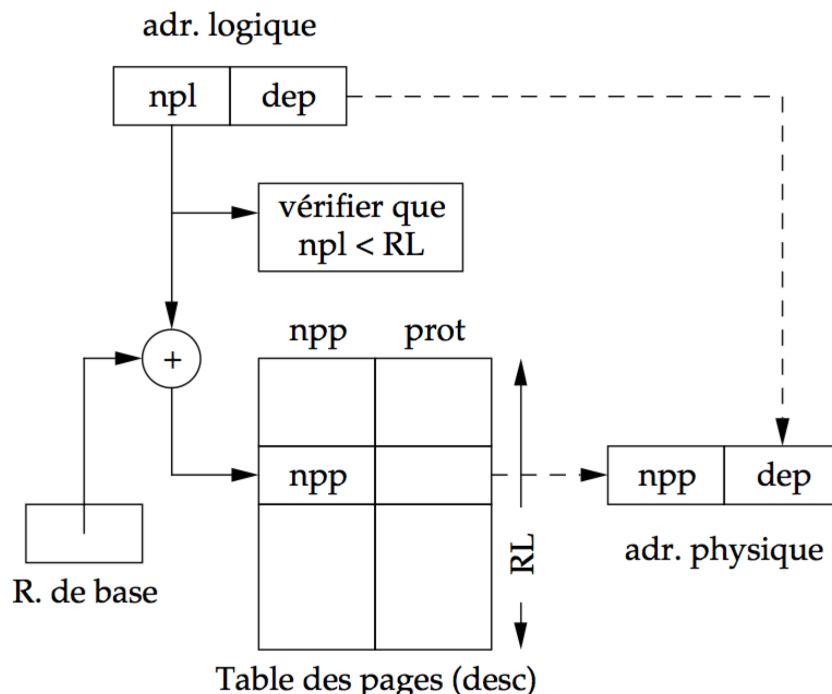


Figure 16: Organisation d'une table de pages.

⟨`npl`, `déplacement`⟩ := ⟨adresse logique⟩
si (`npl` < `RL`) **alors**
| **si** ⟨les protections sont respectées⟩ **alors**
| | ⟨adresse physique⟩ := ⟨`desc[npl].npp`, `déplacement`⟩
| **sinon**
| | ⟨déroutement sur violation de protection⟩
| **fin**
sinon
| ⟨déroutement sur erreur d'adressage⟩
fin

Le champ `desc[npl].prot` indique le mode d'accès autorisé à la page logique `npl`. Cette information est utilisée par les mécanismes de protection et un accès non autorisé provoque un déroutement pour violation de protection.

Notons qu'une table de pages représente le contenu d'une mémoire logique particulière. Si le système d'exploitation permet à chaque processus, ou à chaque usager du système, de définir une mémoire logique distincte, il doit gérer une table de pages distincte par processus ou par usager. Le pointeur vers l'origine de cette table (`RB`) fait alors partie

du contexte du processus ou de l'usager. Les tables des pages se trouvent en mémoire physique, dans la partition réservée au système d'exploitation.

La mémoire logique d'un processus n'est plus représentée d'une manière contiguë en mémoire centrale (voir Figure 17). En effet, l'indirection des accès par la table de pages permet de loger les pages logiques dans n'importe quelle page physique. De ce fait, la gestion de la mémoire physique revient simplement à gérer une liste des pages physiques libres sans idée de regroupement. Les problèmes liés à la fragmentation externe disparaissent mais la fragmentation interne se fait plus présente puisque la page devient l'unité élémentaire d'allocation et de libération (en pratique quelques kibioctets).

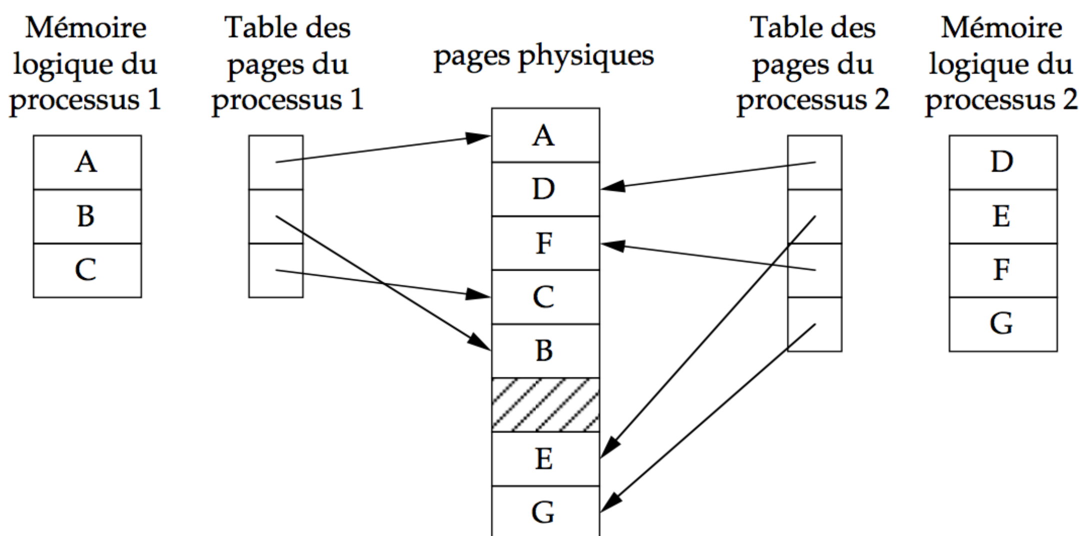


FIGURE 17: Un exemple sur deux mémoires logiques paginées.

L'accès à une page logique nécessite maintenant deux références à la mémoire en raison de la consultation de la table des pages. Cette augmentation du temps d'accès moyen est bien sur intolérable. La réduction de ce coût passe par deux points :

— observer le comportement des processus (vis à vis de la mémoire),

— optimiser la transformation des adresses au moyen d'un circuit particulier : les mémoires associatives.

Donc pour éviter l'accès à cette table (et donc réduire le temps d'accès moyen), on passe par un circuit particulier : une mémoire associative. Avant de présenter cette mémoire, il faut discuter de l'utilisation de la mémoire par les processus

4.10.2 Comportement des processus en mémoire paginée

Le comportement d'un processus dans son espace logique détermine ses demandes de mémoire physique. Il est donc utile de connaître les caractéristiques de ce comportement pour améliorer l'efficacité des algorithmes de gestion dynamique de la mémoire. Donnons d'abord quelques définitions :

— L'écoulement du temps est repéré par l'exécution des instructions successives : l'exécution d'une instruction définit une unité de temps. Ce temps est dit virtuel car il suppose que le programme dispose de toutes les ressources nécessaires (mémoire et processeur). En cas de partage de ressources, on peut ainsi raisonner sur un programme donné en faisant abstraction des autres.

— La mémoire logique paginée est découpée en pages contiguës de taille fixe. L'accès à une page est appelé *référence* à cette page. La numérotation des pages permet d'étiqueter les références.

— Le comportement du processus est défini par la série des numéros de pages référencées au cours de l'exécution. Cette séquence s'appelle *chaîne de référence* pour le processus considéré

L'exécution d'une instruction peut donner lieu à plusieurs références distinctes : pages contenant l'instruction, le ou les opérandes.

L'expérience montre que les chaînes de références des processus possèdent des caractéristiques communes que nous définirons d'abord de manière qualitative.

— *Non-uniformité*. Soit n_i le nombre total de références à une page p_i. La répartition des n_i n'est pas uniforme : un faible pourcentage des pages cumule généralement un taux très important du nombre total des références. Il est courant de constater que 80% des références concernent un sous-ensemble de moins de 20% des pages.

— *Principe de localité*. Sur un temps d'observation assez court, la répartition des références présente une certaine stabilité : les références observées dans un passé récent sont en général une bonne estimation des prochaines références.

A partir de cette constatation de localité, on peut créer un modèle de comportement des programmes. Dans ce modèle, le déroulement d'un programme est défini comme une succession de phases séparées par des transitions. Une phase i est caractérisée par un ensemble de pages S_i et un intervalle de temps virtuel T_i. Lorsque le programme entre en phase i, il y reste un temps T_i en effectuant principalement des références à des pages de S_i. Ensuite, il subit un transition durant laquelle les références aux pages sont dispersées, avant d'entrer dans la phase $i + 1$.

Les phases constituent donc des périodes de comportement stable et relativement prévisible, alors que les transitions correspondent à un comportement plus erratique. L'expérience montre que les périodes de transition ne représentent qu'une faible partie du temps virtuel total, la majeure partie du temps virtuel étant occupé par des phases de longue durée (quelques centaines de milliers d'instructions).

Qualitativement, ce type de comportement s'explique par le fait que les programmes sont souvent organisés en procédures possédant chacune un contexte spécifique, que les accès aux données sont souvent concentrés (*e.g.* parcours de tableau), que les programmes comportent des boucles concentrant aussi les références.

La notion d'ensemble de travail (« working set ») est également utilisée pour caractériser le comportement des programmes et prévoir d'après l'observation. Soit $W(t, T)$ l'ensemble des pages ayant été référencées entre les temps $t - T$ et t. D'après la propriété de localité, ces pages ont une probabilité plus élevée que les autres de faire l'objet d'une référence au temps t à condition toutefois que la taille de la fenêtre d'observation T soit convenablement choisie. En admettant un comportement suivant le modèle phase/transition, pour avoir du sens T devra être inférieur à T_i en phase i.

4.10.3 Mémoire associative et pagination

Une *mémoire associative* est un ensemble de couples ⟨entrée, sortie⟩. La présence d'une valeur sur le bus d'entrée provoque soit l'apparition d'une valeur de sortie, soit un signal d'échec indiquant que cette entrée n'existe pas dans la mémoire associative (Figure 18).

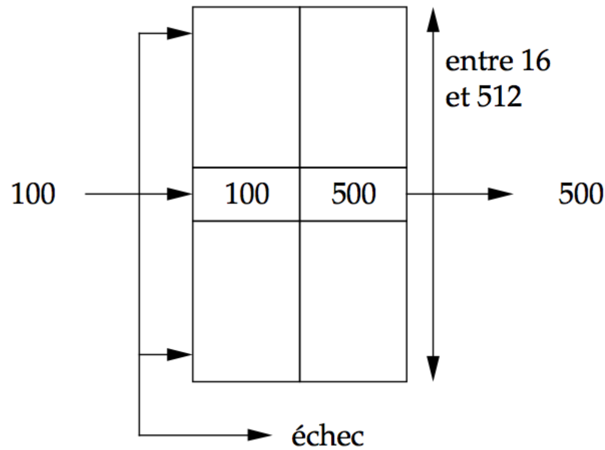


FIGURE 18: Schéma d'une mémoire associative.

Ces mémoires ont quelques dizaines à quelques centaines d'entrées et leur coût très élevé a empêché leur extension à des mémoires de plus grande taille.

Dans cette mémoire associative on conserve les couples ⟨`npl`,`npp`⟩ relevés lors des accès les plus récents. En raison du principe de localité des programmes, on a une probabilité élevée (80% à 95% avec les tailles usuelles) de trouver dans la mémoire associative le numéro de la page logique adressée et donc de déterminer sa page physique. Ce n'est qu'en cas d'échec que l'on passe par la table des pages ; la mémoire associative est alors mise à jour (Figure 19).

En partant du principe que l'accès à la mémoire physique prend 100 ns et que le temps de recherche de la mémoire associative est de 20 ns, le temps moyen d'accès est compris entre

$$\begin{aligned} 0,8 \times (100 + 20) + 0,2 \times (100 + 20 + 100) &= 140ns \\ 0,95 \times (100 + 20) + 0,05 \times (100 + 20 + 100) &= 125ns \end{aligned}$$

suivant la probabilité de réussite et donc la taille de la mémoire associative. Finalement, le temps d'accès moyen n'a augmenté que de 25% par rapport aux mécanismes sans réimplantation, mais la gestion de la mémoire est beaucoup plus souple et les problèmes de fragmentation externe n'existent plus.

4.10.4 Partage et protection de l'information

L'utilisation d'informations partagées entre plusieurs mémoires logiques soulève trois problèmes :

— la *désignation* : comment adresser de manière uniforme les informations partagées ;

— le *partage physique* : comment assurer que les informations partagées existent en exemplaire unique ;

— la *protection* : comment garantir le respect des règles d'accès (éventuellement sélectives) aux informations partagées.

Dans un système paginé, l'unité élémentaire de partage est la page. Pour être partagés, les informations doivent se trouver sur une (ou plusieurs) page logique partagée. Cette page peut être chargée dans une page physique quelconque ; les tables de pages des mémoires logiques où figure cette page logique contiennent alors, à l'entrée correspondante, le même numéro de page physique (Figure 20).

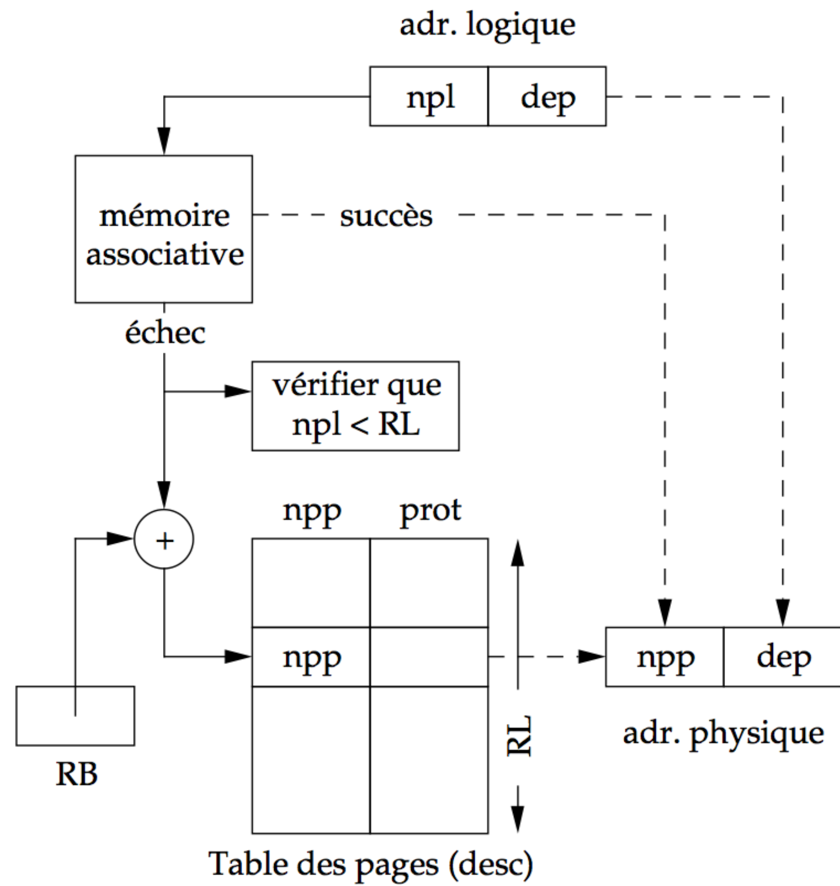


Figure 19: Pagination avec mémoire associative.

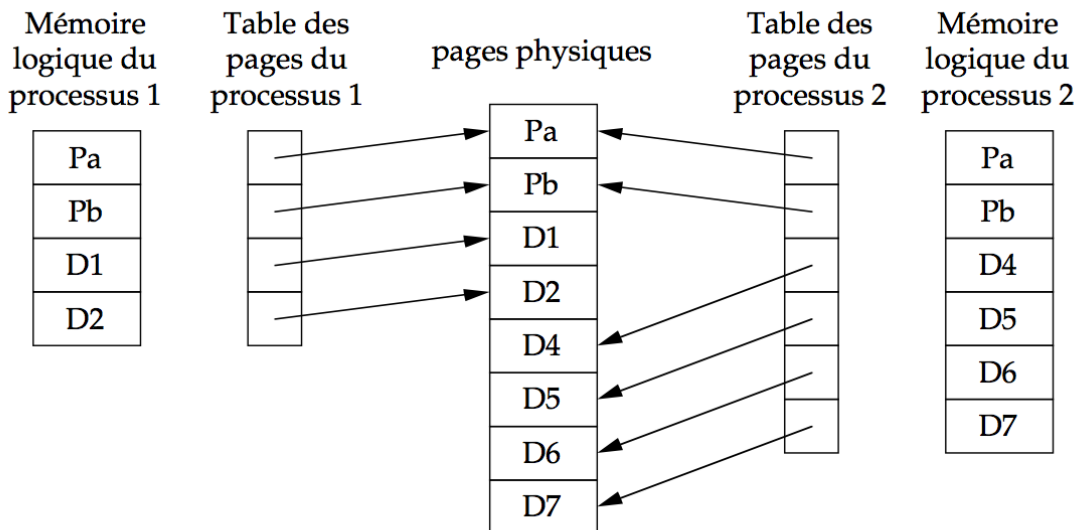


Figure 20: Partage de pages entre mémoires logiques paginées.

Si l'unité de partage est la page, une page physique partagée peut recevoir des droits d'accès distincts dans chaque mémoire logique où elle figure. Ces droits sont spécifiés à

l'entrée correspondante de la table de pages.

4.11 Mémoire segmentée

Nous sommes ici dans le cas où la mémoire logique est non-contiguë.

4.11.1 Principe de la segmentation

Dans les systèmes de mémoire paginée, la division en pages est arbitraire et ne tient pas compte du mode d'organisation des données d'un processus. Notamment, il est fort probable que certaines structures de données se trouvent « à cheval » sur plusieurs pages, ce qui peut être la cause de temps d'accès plus importants.

L'objectif des *mémoires segmentées* est de mettre en rapport la structure logique de la mémoire (vue des processus) et l'implantation physique de cette mémoire. Pour ce faire, la *mémoire logique segmentée* d'un processus est définie comme un ensemble de segments numérotés à partir de zéro (Figure 21). Un segment est une zone contiguë de taille variable.

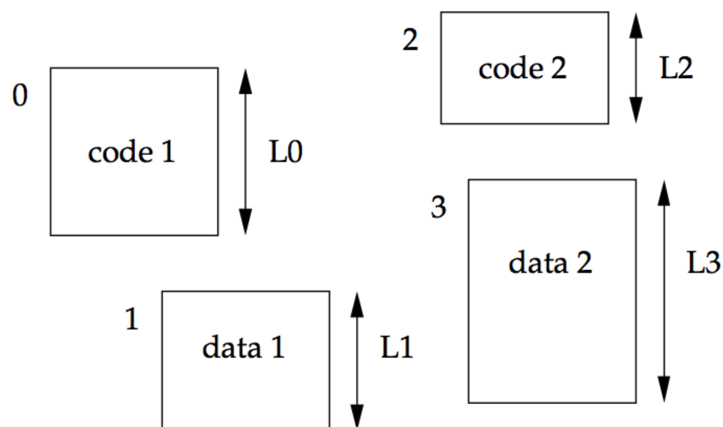


Figure 21: Une mémoire segmentée.

Une *adresse logique* dans un système segmenté (aussi appelée *adresse segmentée*) est un couple

$$\langle \text{numéro de segment, déplacement} \rangle$$

Comme dans les mémoires paginées, le S.E. maintient une table des segments pour chaque processus (Figure 22). La correspondance proprement dite est établie par le matériel de la manière suivante :

```
⟨seg,dépl⟩ = ⟨adresse logique segmentée⟩
si ((seg < RL) et (dépl < desc[seg].taille)) alors
|   si ⟨les protections sont respectées⟩ alors
|   |   ⟨adresse physique⟩ = desc[seg].origine + dépl
|   sinon
|   |   ⟨déroutement sur violation de protection⟩
|   fin
sinon
|   ⟨ déroutement sur erreur d'adressage ⟩
fin
```

Ce mécanisme doit être couplé à une mémoire associative pour améliorer le temps d'accès moyen en évitant l'utilisation de la table des segments.

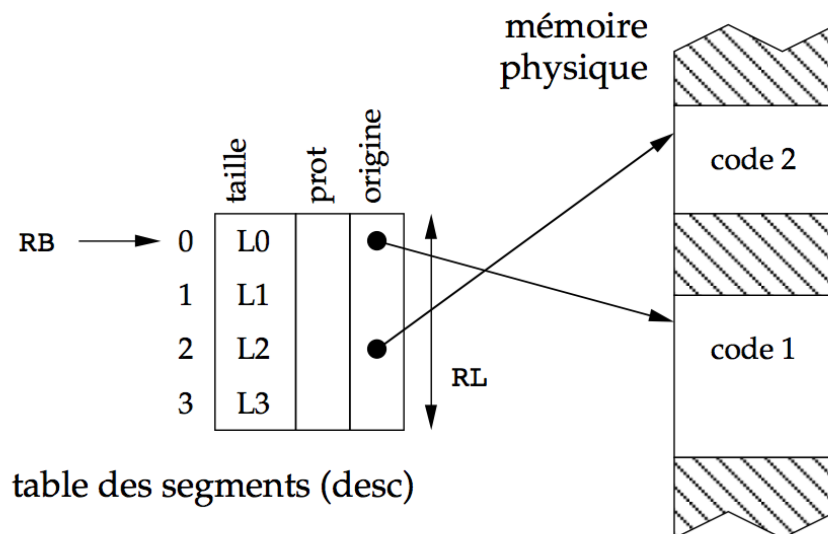


FIGURE 22: Table des segments d'un processus.

Les avantages de cette organisation sont doubles : d'une part, la notion de segment est directement utilisable dans un processus et de ce fait on peut espérer une réduction des temps d'accès moyens (les accès fréquents étant regroupés sur un petit groupe de segments, voire même sur un segment unique) ; d'autre part, la notion de protection est plus facilement utilisable puisqu'elle porte directement sur des segments, c'est à dire des objets logiques.

Les segments sont des zones contiguës (du moins pour l'instant). On retrouve donc les problèmes d'allocation/libération de zones et l'apparition d'une fragmentation externe éventuellement corrigée par des compactages de la mémoire. Dans le cas de la mémoire segmentée, ces compactages impliquent une remise à jour des pointeurs « origine » des tables de segments.

4.11.2 Pagination d'une mémoire segmentée

La pagination d'une mémoire segmentée vise à rendre plus souple l'allocation de mémoire aux segments en levant la restriction de contiguïté pour le placement d'un segment. Une entrée de la table des segments contient, outre les informations propres au segment (taille, protection, type), un pointeur vers la table des pages de ce segment (Figure 23). Comme dans les techniques précédentes, une mémoire associative conserve les dernières références.

4.11.3 Partage de segments

Dans une mémoire logique segmentée, le partage s'applique aux segments et les tables de pages des segments partagés sont elles-mêmes partagées (dans le cas d'un système segmenté paginé). Tous les descripteurs d'un segment contiennent alors, non pas l'adresse de ce segment, mais celle de sa table de pages qui est unique.

Si l'unité de partage est le segment, la protection sélective s'applique globalement au segment. Les droits d'accès à un segment pour un processus figurent dans la table

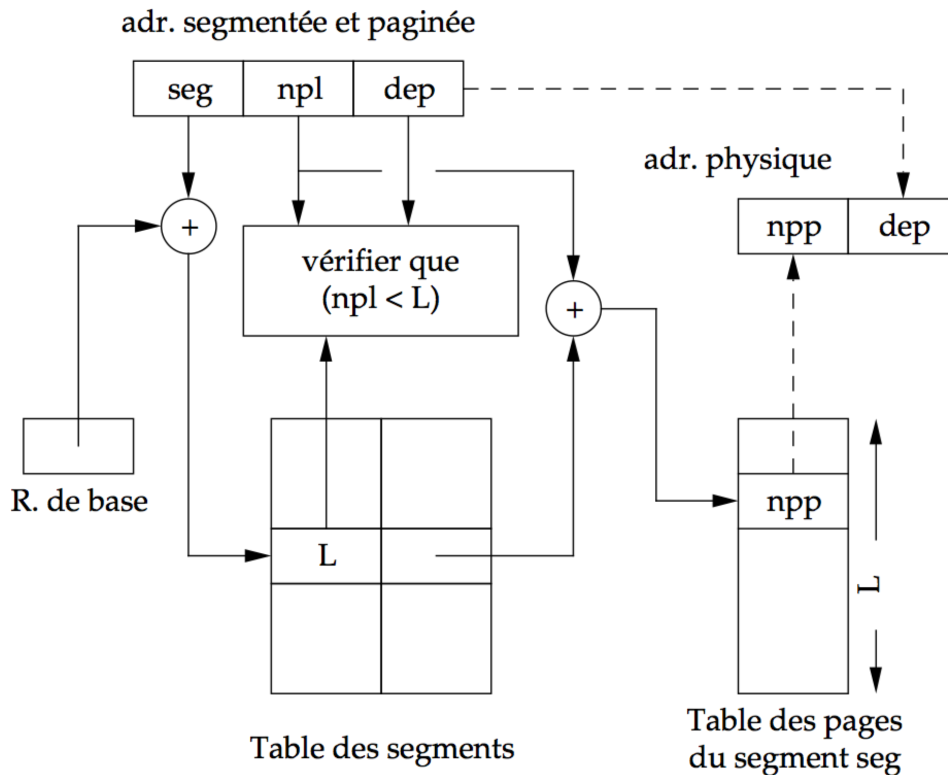


FIGURE 23: Pagination d'une mémoire segmentée.

des segments de ce processus (Figure 24). Si des droits d'accès individuels aux pages du segment sont spécifiés, ils figurent dans la table de pages partagée par les processus utilisateurs et sont donc les mêmes pour tous. Ils doivent alors être compatibles avec les droits globaux associés au segment.

4.12 Mémoire virtuelle paginée

Ce mécanisme est similaire à la mémoire paginée (Section 4.10), mais ajoute l'extension de la mémoire principale, c'est le principe de mémoire virtuelle : on simule une RAM plus grande qu'elle ne l'est vraiment, et pour cela on utilise (en plus de la mémoire principale) une partie de la mémoire secondaire pour stocker des informations. Lorsqu'une page (l'unité d'information utilisée dans les mécanismes de pagination) est demandée par un processus :

- soit elle est en RAM et il faut retrouver son adresse physique dans cette mémoire principale,
- soit elle a été placée en ROM et il faut la transférer en RAM puis donner son adresse physique dans cette mémoire principale.

La politique de choix des pages à garder en RAM / placer en ROM va reposer sur l'important *principe de localité* (Section 4.1 que nous répétons ici) : sur un petit intervalle de temps, un processus utilise 20% de ses pages logiques. On peut donc en conclure que 80% des pages logiques sont en mémoire principale sans raison valable. Il est donc raisonnable d'enlever ces pages inutiles de manière à offrir plus de place mémoire pour d'autres processus et ainsi augmenter le degré de multi-programmation.

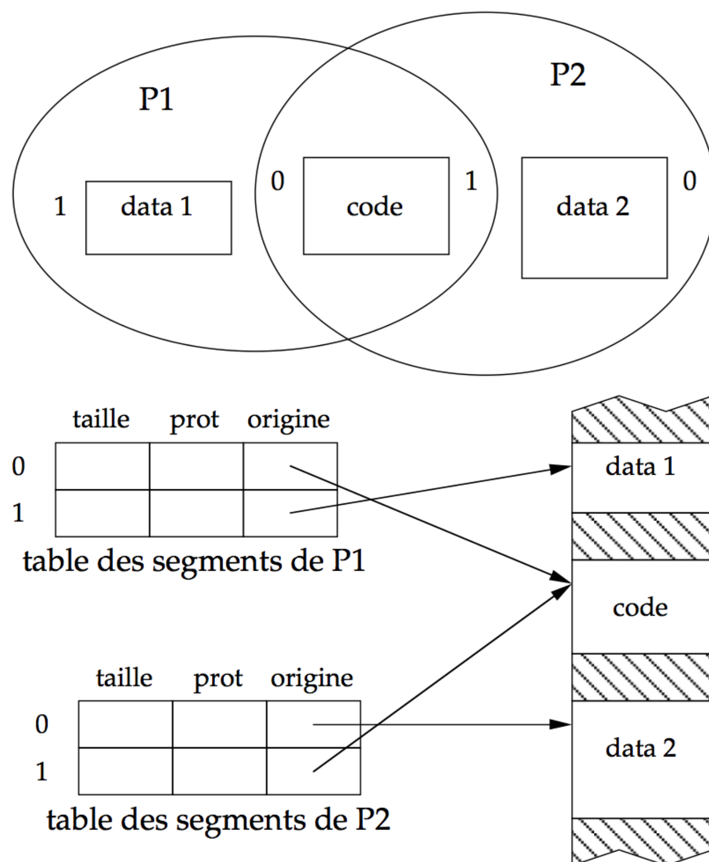


FIGURE 24: Partage de segments dans un système segmenté.

4.12.1 Pagination simple d'une mémoire virtuelle

La Figure 25 représente le schéma général d'une mémoire virtuelle paginée. La mémoire virtuelle est plus importante que la mémoire physique et les pages virtuelles qui ne se trouvent pas en mémoire physique sont stockées en mémoire secondaire.

Nous avons maintenant 2^l la taille des pages (virtuelles et physiques), 2^p le nombre de pages virtuelles (taille de la mémoire virtuelle) et 2^c le nombre de pages physiques (taille de la mémoire physique) avec $p > c$.

Une *adresse virtuelle* est donc un couple $\langle \texttt{npv}, \texttt{dep} \rangle$ avec `npv` un numéro de page virtuelle (sur p bits) et `dep` un déplacement (sur l bits). De même, une *adresse physique* est un couple $\langle \texttt{npp}, \texttt{dep} \rangle$ avec `npp` un numéro de page physique (sur c bits) et `dep` un déplacement (sur l bits).

La *fonction de pagination* a maintenant la charge de transformer les adresses virtuelles en adresse physique et de détecter les défauts de page, c'est à dire les accès à des pages virtuelles qui ne sont stockées dans aucune page physique.

Pour réaliser cette opération, la *table des pages virtuelles* comporte (pour chaque page virtuelle) les informations suivantes (Figure 26) :

- un numéro de page physique (`npp`),
- un indicateur de présence (`présent`) (1 bit),
- un indicateur de modification ou *dirty bit* (`modif`) (1 bit),
- un mode d'accès autorisé (`prot`).

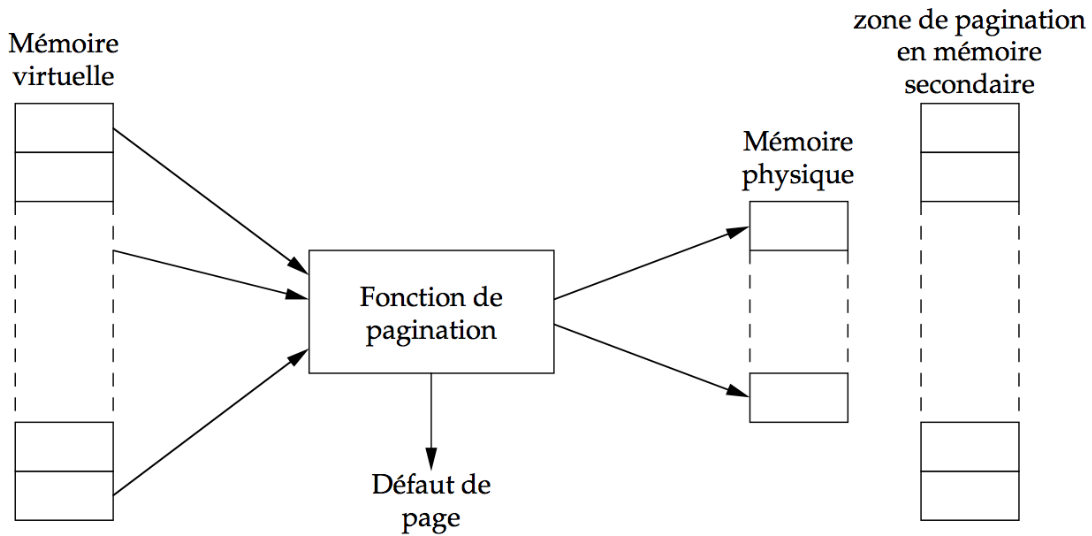


FIGURE 25: Mémoire virtuelle paginée.

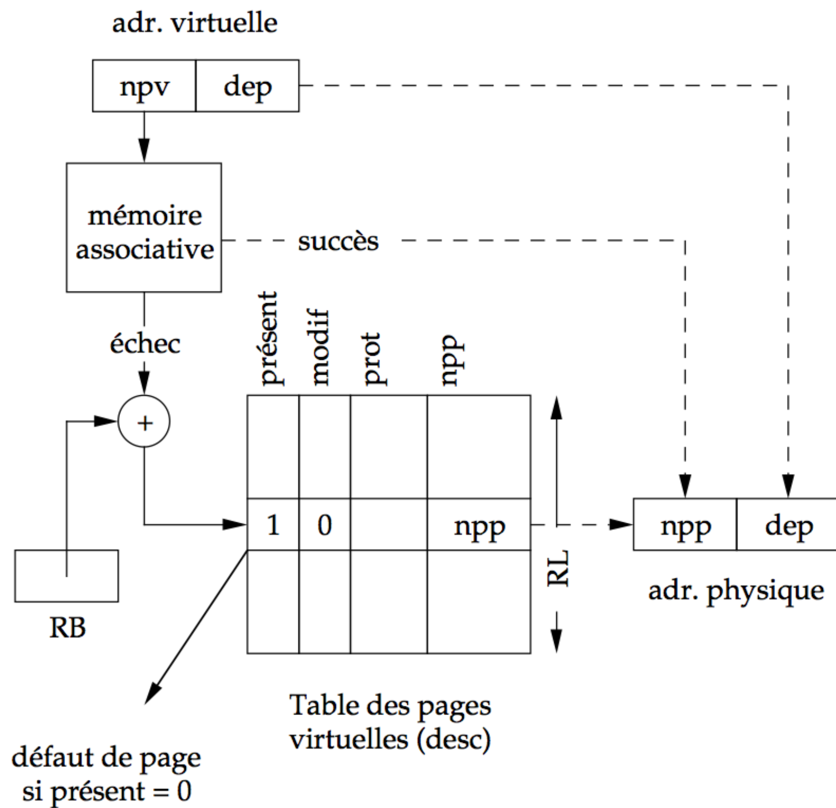


FIGURE 26: Transformation adresse virtuelle / adresse physique.

Lors d'un accès à la mémoire, la correspondance adresse virtuelle / adresse physique est mise en oeuvre comme suit :

⟨`npv`, `dep`⟩ := ⟨adresse virtuelle⟩
⟨vérifier que (`npv` $<$ RL)⟩
⟨vérifier les protections⟩
si (`desc[npv].présent` $= 1$) **alors**
| ⟨adresse physique⟩ := (`desc[npv].npp`, `dep`)
sinon
| ⟨déroutement pour défaut de page⟩
fin

Lorsque la page virtuelle est présente, le *dirty bit* `modif` indique si la page virtuelle a été modifiée depuis son chargement en mémoire ; cette information, mise à jour automatiquement par le matériel, est utilisée par l'algorithme de remplacement pour éviter éventuellement la mise en mémoire secondaire d'une page modifiée récemment (principe de localité).

4.12.2 Pagination à deux niveaux d'une mémoire virtuelle

La tendance à la croissance de la taille des mémoires virtuelles se heurte au problème de l'encombrement de la mémoire principale par les tables de pages virtuelles, dont la taille augmente en proportion. La pagination à deux niveaux (voir même à plusieurs niveaux) vise à résoudre ce problème en limitant les tables de pages virtuelles aux seules parties effectivement utilisées de la mémoire virtuelle (Figure 27).

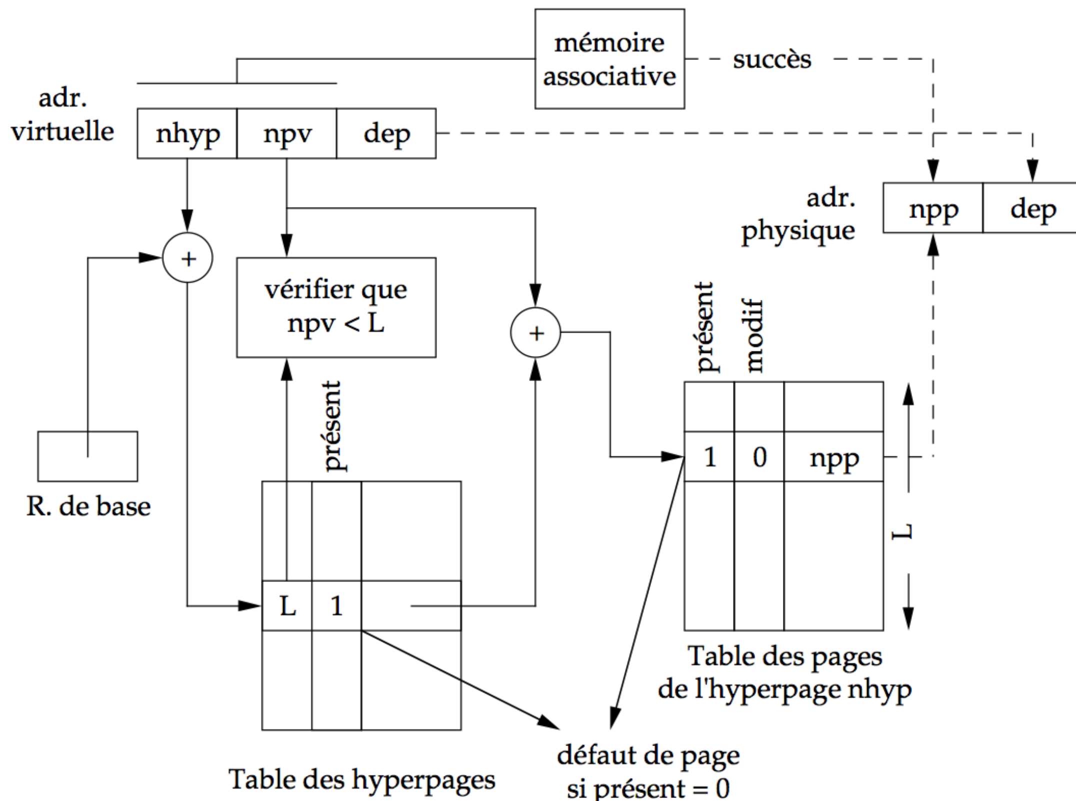


Figure 27: Pagination à deux niveaux.

Dans ce schéma, la mémoire virtuelle est divisée en *hyperpages*, elles même divisées en pages (virtuelles). Une adresse virtuelle a maintenant la forme ⟨`nhyp`,`npv`,`dep`⟩. Le numéro

`nhyp` permet d'accéder à une table d'hyperpages dont chaque entrée comporte un pointeur vers la table de pages de l'hyperpage. Seules les hyperpages effectivement utilisées se trouvent en mémoire principale. Dans certains systèmes les tailles de pages et d'hyperpages pouvant être variables, un champ taille permettra de contrôler la limitation et détecter une éventuelle erreur d'adressage. Les tables de pages sont utilisées comme dans la pagination simple. Une mémoire associative, qui conserve les triplets ⟨`nhyp`,`npv`,`npp`⟩ les plus récents, accélère la consultation (qui nécessite deux accès supplémentaires à la mémoire en cas d'échec).

Notons que la mémoire virtuelle reste contiguë : le dernier emplacement de l'hyperpage h a pour successeur le premier emplacement de l'hyperpage $h+1$. C'est donc par abus de langage que cette technique est souvent appelée *segmentation*. Ce découpage de la mémoire virtuelle peut néanmoins être utilisé pour simuler une mémoire segmentée.

4.12.3 Mécanisme du défaut de page

Outre la traduction proprement dite des adresses (correspondance npv/npp), le mécanisme d'accès à une mémoire paginée doit réaliser les opérations suivantes :

— mise à jour du bit d'écriture et du bit d'utilisation (si ils existent),

— détection du défaut de page (`desc[npv].présent` $= 0$) qui provoque un déroutement.

Le programme du traitement de déroutement pour défaut de page doit :

1. trouver en mémoire secondaire la page virtuelle manquante,
2. trouver une page physique libre en mémoire principale ; s'il n'y a pas de page physique libre, il faut en libérer une en
   — choisissant une page virtuelle à enlever de la mémoire (c'est la victime du défaut de page) ;
   — sauvegardant la victime (si nécessaire) dans la zone de pagination en mémoire secondaire ;
3. charger de la page virtuelle dans la page physique ainsi rendue libre.

L'étape (1) nécessite d'avoir pour chaque page virtuelle, une description d'implantation. Sa forme la plus simple est celle d'une table qui indique pour chaque page virtuelle son adresse en mémoire secondaire. Une mémoire segmentée comporte une telle table pour chaque segment. Une autre forme de description combine mémoire virtuelle et fichiers : à une zone de mémoire virtuelle est associé le contenu d'un ou de plusieurs fichiers. La localisation d'une page virtuelle en mémoire secondaire est alors déterminée en consultant la table d'implantation du fichier dont elle contient un élément.

L'étape (2) met en œuvre un *algorithme de remplacement*. Elle nécessite de conserver une table d'occupation de la mémoire physique qui indique pour toute page physique occupée l'identité de la page virtuelle qui l'occupe ainsi que les renseignements complémentaires (protection, partage, etc.).

4.12.4 Comportement des programmes en mémoire virtuelle

Le comportement d'un programme par rapport à une mémoire virtuelle, pour un algorithme de remplacement donné, est caractérisé par la suite de ses défauts de pages. L'expérience montre que l'on observe des propriétés communes au comportement de la majorité des programmes, relativement indépendantes de l'algorithme de remplacement utilisé. Ce sont donc des caractéristiques intrinsèques du comportement.

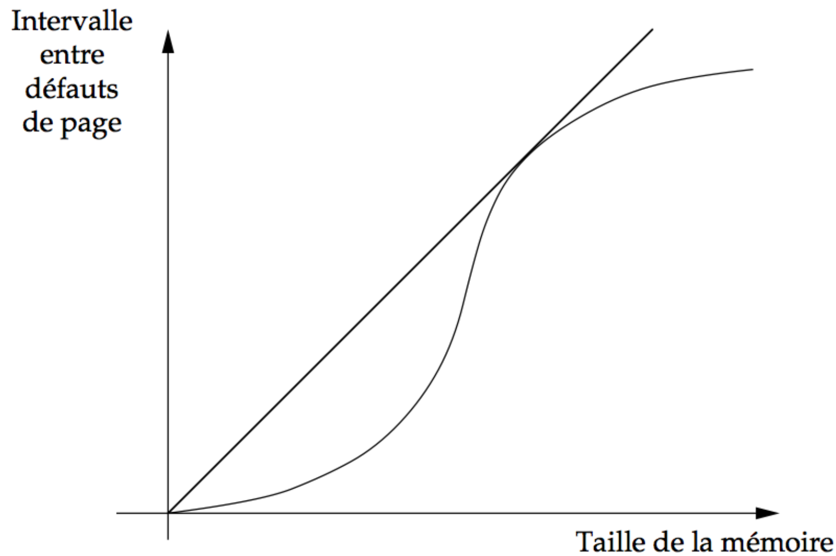


Figure 28: Intervalle moyen entre défauts de page en mémoire restreinte.

Elles peuvent être illustrées par deux courbes obtenues en exécutant un programme avec différentes tailles de mémoire principale. L'allure de ces courbes est représentative d'un type de comportement fréquemment observé. La Figure 28 représente l'intervalle moyen entre deux défauts de page successifs, dit durée de vie. La Figure 29 représente le nombre total de défauts de page observés au cours de l'exécution d'un programme.

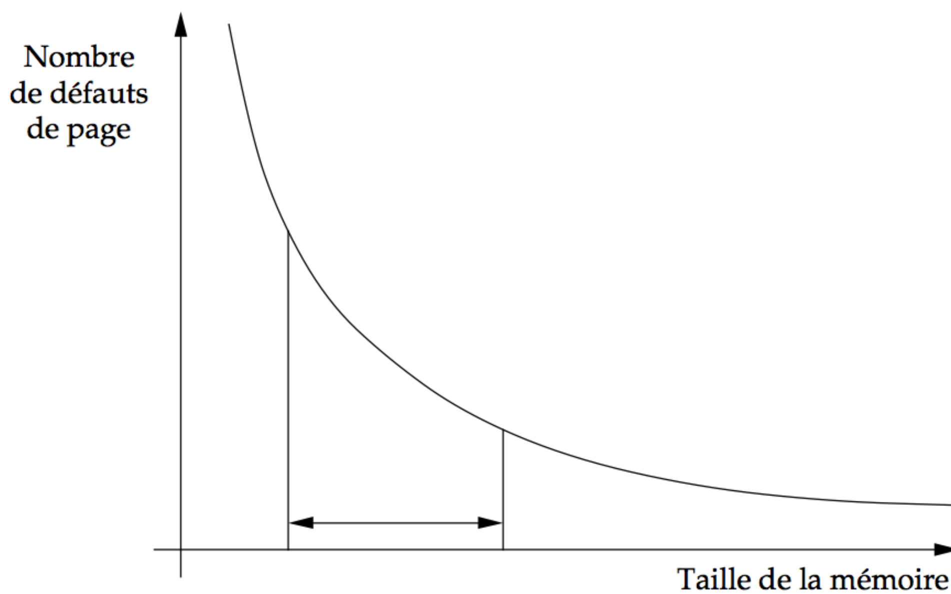


Figure 29: Nombre total de défauts de page en mémoire restreinte.

On constate que lorsque la taille de mémoire diminue, ce nombre croît d'abord lentement. Au dessous d'une certaine taille, la croissance devient exponentielle.

4.12.5 Gestion d'une mémoire virtuelle paginée

4.12.5.1 Paramètres d'une politique d'allocation

Les politiques d'allocation d'une mémoire paginée peuvent être classées selon plusieurs critères. Nous supposons que le système est multi-programmé entre plusieurs processus dont chacun possède sa propre mémoire virtuelle.

— *Partition fixe ou variable*. Dans une politique à partition fixe, un nombre fixe de pages physiques est attribué à chacun des processus ; notons que ce nombre n'est constant que pendant les périodes où le nombre de processus multiprogrammés est lui-même constant. Dans une politique à partition variable, le nombre de pages physiques attribuées à chaque processus varie au cours du temps. Les pages physiques étant banalisées, c'est leur nombre (et non leur identité physique) qui est le paramètre significatif

— *Pagination à la demande*. Une page virtuelle n'est chargée en mémoire qu'à la suite d'une référence donnant lieu à un défaut de page.

— *Pré-chargement*. Lorsqu'une page virtuelle est chargée à l'avance, avant toute référence à une information qu'elle contient, on dit qu'il y a pré-chargement.

— *Remplacement local ou global*. Il y a remplacement de page lorsqu'une page virtuelle est chargée dans une page physique occupée, c'est à dire contenant une page virtuelle chargée antérieurement et susceptible d'être encore utilisée (cette dernière page est souvent appelée la *victime*). L'algorithme de remplacement est dit *local* ou *global* selon que la victime est choisie parmi les pages virtuelles du processus qui provoque le chargement ou parmi l'ensemble de toutes les pages virtuelles présentes en mémoire.

Avant d'étudier et de comparer les algorithmes de remplacement de pages, il faut mentionner des critères valables quel que soit l'algorithme, et qui sont appliqués en priorité.

1. Pages virtuelles « propres » ou « sales ». Toutes choses égales par ailleurs, il est toujours moins coûteux de remplacer une page virtuelle qui n'a pas été modifiée depuis son chargement (page *propre*) plutôt qu'une page modifiée (page *sale*). Une page propre possède une copie conforme en mémoire secondaire et ne doit donc pas être sauvegardée. L'indicateur `modif` (*dirty bit*), entretenu automatiquement, permet d'appliquer ce critère.
2. Pages virtuelles partagées. Une page virtuelle utilisée par un seul processus doit être remplacée de préférence à une page partagée entre plusieurs processus.
3. Pages virtuelles à statut spécial. Dans certains cas, on souhaite donner temporairement à une page virtuelle un « statut » spécial qui la protège contre le remplacement. Ce cas se présente surtout pour des pages utilisées comme tampons d'entrée-sortie pendant la durée d'un transfert.

4.12.5.2 Algorithmes de remplacement

Nous présentons d'abord deux algorithmes qui servent de référence : l'algorithme optimal, qui suppose une connaissance complète du comportement futur du programme, et un algorithme « neutre » qui n'utilise aucune information.

— *Algorithme optimal* (OPT). Pour une chaîne de références donnée, on peut montrer que l'algorithme suivant minimise le nombre total de défauts de pages : lors d'un défaut de page, choisir comme victime une page virtuelle qui ne fera l'objet d'aucune référence ultérieure, ou, à défaut, la page qui fera l'objet de la référence la plus tardive. Cet algorithme suppose une connaissance de l'ensemble de la chaîne de références ; il est donc irréalisable en temps réel. Il permet d'évaluer par comparaison les autres algorithmes.

— *Tirage aléatoire* (ALEA). La victime est choisie au hasard (loi uniforme) parmi l'ensemble des pages virtuelles présentes en mémoire. Cet algorithme n'a aucune vertu particulière, car il ne tient aucun compte du comportement observé ou prévisible du programme ; il sert lui aussi de point de comparaison.

— *Ordre chronologique de chargement* (FIFO ou *First In, First Out*). Cet algorithme choisit comme victime la page virtuelle la plus anciennement chargée. Son principal intérêt est sa simplicité de réalisation : il suffit d'entretenir dans une file les numéros des pages physiques où sont chargées les pages virtuelles successives.

— *Ordre chronologique d'utilisation* (LRU ou *Least Recently Used*). Cet algorithme tente d'approcher l'algorithme optimal en utilisant la propriété de localité. Son principe est le suivant : puisque les pages virtuelles récemment utilisées ont une probabilité plus élevée que les autres d'être réutilisées dans un futur proche, une page virtuelle non utilisée depuis un temps élevé a une probabilité faible d'être utilisée prochainement. L'algorithme choisit donc comme victime la page virtuelle ayant fait l'objet de la référence la plus ancienne.

  La réalisation de l'algorithme impose d'ordonner les pages physiques selon la date de dernière référence de la page virtuelle qu'elles contiennent. Pour cela, une information doit être associée à chaque page physique et mise à jour à chaque référence. Cette information peut être une date de référence ; une solution plus économique, mais encore chère, consiste à utiliser un compteur incrémenté de 1 à chaque référence ; la page physique dont le compteur a la valeur la plus faible contient la victime. Les compteurs ayant une capacité limitée, il doivent être remis à zéro dès que l'un d'eux atteint sa capacité maximale ; la réalisation de LRU n'est donc qu'approchée. En raison de son coût, cette solution n'a été utilisée que sur des installations expérimentales. Si la taille du compteur est réduite a 1 bit, on obtient l'algorithme suivant dont le coût est acceptable.

— *Algorithme « de la seconde chance »* (FINUFO ou *First In Not Used, First Out*). Cet algorithme est une approximation très grossière de LRU. Il utilise une liste FIFO, dans laquelle les pages sont placées par ordre d'arrivée (une nouvelle page est placée en queue de liste). À chaque page physique est de plus associé un bit d'utilisation (noté `U`), initialement à 0 et mis à 1 lors d'une référence à la page qu'elle contient. Lorsque l'on cherche une page à remplacer, on parcours la liste FIFO à partir du début (la tête de la liste), et l'on donne une seconde chance aux pages dont le bit `U` est à 1 : pour ce faire, on les replace en queue de la liste avec le bit `U` à 0. La page remplacée sera la première rencontrée dont le bit d'utilisation `U`

est à 0 (la plus vieille non utilisée). L'algorithme s'écrit comme suit :

```
victime := ⟨tête de liste⟩
tant que (U[victime] = 1) faire
  | U[victime] := 0;
  | ⟨placer victime en queue de liste⟩;
  | victime := suivant(victime);
fin
retourner victime;
```

Le pointeur progresse jusqu'à la première page physique dont le bit est à 0 ; les pages physiques rencontrées en route (dont le bit est donc à 1) reçoivent une « seconde chance » (de n'être pas prises comme victime) et leur bit est forcé à 0. Notons que `suivant` de la queue de liste donne la tête de liste : si tous les bits `U` sont à 1, l'algorithme va prendre la page la plus vieille (comme aurait fait FIFO). Cet algorithme est également appelé *algorithme de l'horloge* (*Clock*), la progression du pointeur `victime` étant analogue à celle de l'aiguille d'une horloge : comme on parcours les pages physiques toujours dans le même ordre, si `U[victime] = 1` il n'est pas nécessaire de prendre cette tête de liste pour la placer en queue, il suffit de considérer la liste de toutes les pages physiques comme circulaire, et de conserver le pointeur `victime` comme position courante de la tête de liste (la première instruction de l'algorithme devient alors obsolète, mais il faut incrémenter le pointeur `victime` à la fin de l'algorithme pour que la dernière page chargée soit en queue de liste). La mise à 1 du bit `U` lors d'une référence peut être réalisée par un mécanisme matériel ou logiciel.

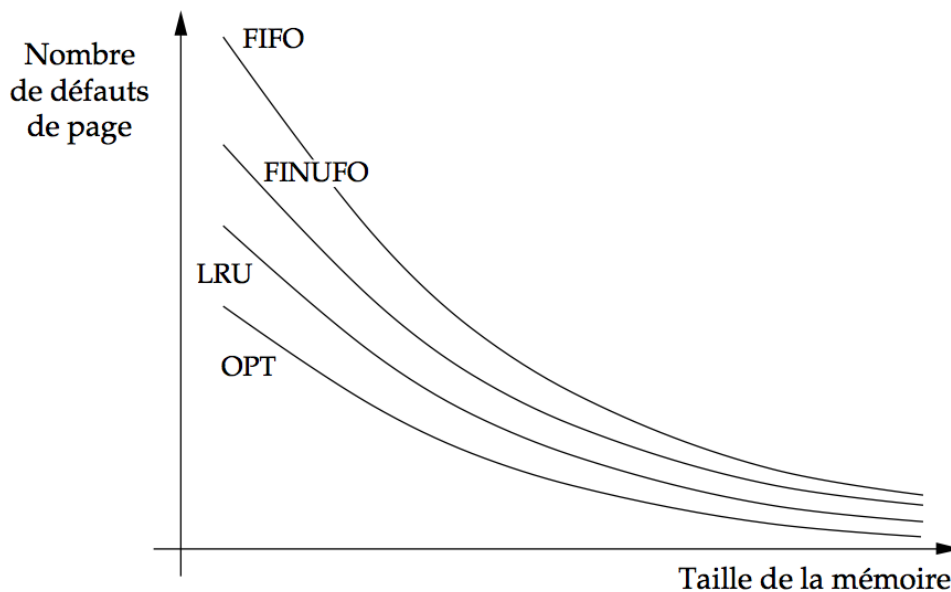


FIGURE 30: Performances des algorithmes de remplacement.

De nombreuses études expérimentales ont permis d'évaluer les algorithmes de remplacement. La Figure 30 est une représentation synthétique des résultats obtenus. On constate :

— que les algorithmes se classent en moyenne (par performances décroissantes), dans l'ordre OPT, LRU, Clock, FIFO (les performances de FIFO sont du même ordre que celles du tirage au hasard),

— que l'influence de la taille de mémoire est très largement supérieure à celle de l'algorithme de remplacement ; autrement dit, on améliore davantage les performances d'un programme en augmentant le nombre de pages physiques allouées plutôt qu'en raffinant l'algorithme de remplacement, et cela d'autant plus que les performances initiales sont mauvaises.

4.12.6 Écroulement d'un système de mémoire virtuelle paginée

4.12.6.1 Apparition de l'écroulement

Sur les premiers systèmes à mémoire virtuelle paginée, on constatait à partir d'une certaine charge (mesurée, par exemple, par le nombre d'usagers interactifs), une dégradation brutale des performances. Ce phénomène, appelé *écroulement* (*thrashing*) se traduit par une chute du taux d'utilisation du processeur et un fort accroissement des échanges de pages ; le temps de réponse atteint des valeurs inacceptables.

Une explication qualitative de l'écroulement permet de mettre en évidence ses causes et de proposer des remèdes. Considérons un système à mémoire paginée, entre un ensemble de processus dont chacun correspond à un usager interactif. La mémoire physique est partagée équitablement entre les processus dont le comportement moyen est supposé identique ; ce partage est mis en œuvre par un algorithme de remplacement global.

Au delà d'un certain nombre de processus (Figure 31), le nombre moyen de pages physiques allouées à chacun d'eux ne permet plus de stocker en mémoire centrale leur ensemble de travail, c'est à dire le sous-ensemble de leurs pages virtuelles fréquemment utilisées. La probabilité globale de défaut de page croît dès lors très rapidement avec le nombre de processus.

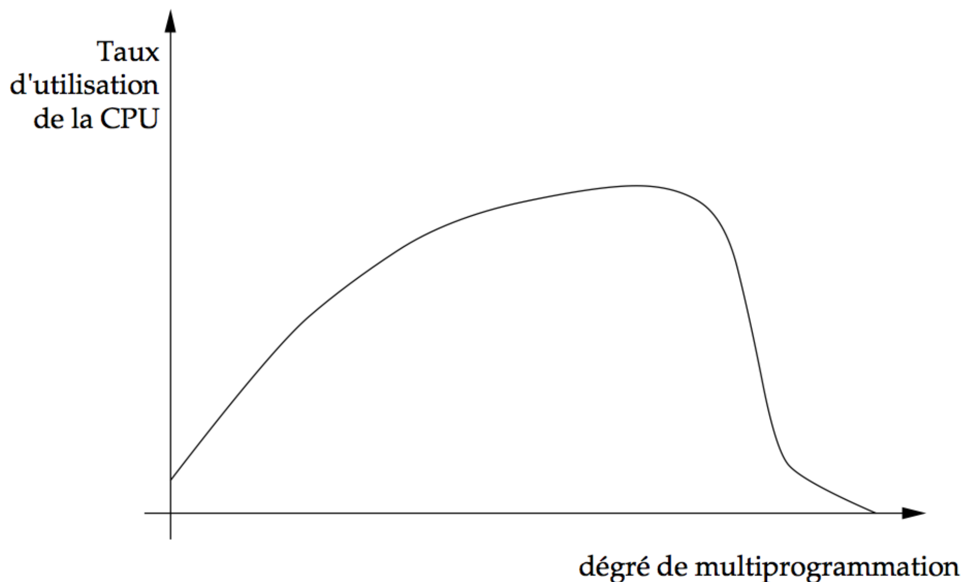


FIGURE 31: Écroulement d'un système de mémoire virtuelle paginée.

Si le nombre de défauts de page augmente, le nombre de processus bloqués pour traitement du défaut de page augmente également. Donc le degré de multi-programmation et le taux d'utilisation de la CPU baissent. Face à cette baisse, l'ordonnanceur à long terme peut décider de ramener des processus en mémoire principale (*swapping in*) dans le but (qui ne va en fait qu'empirer les choses) d'améliorer le taux d'utilisation de la CPU ! Bien

entendu, ces nouveaux venus vont prendre des pages physiques, provoquer des défauts de page, et augmenter le nombre – déjà important – de défauts de page chez les autres processus. C'est un effet « boule de neige » qui entraîne l'écroulement.

Les résultats qui viennent d'être présentés montrent qu'il semble préférable d'essayer d'allouer à chaque programme une taille de mémoire bien adaptée à son comportement, donc variable dynamiquement. Cela peut être tenté de deux façons :

- en utilisant un algorithme à *partition variable* afin de réquisitionner des pages physiques mal utilisées par un processus pour les redistribuer à d'autres processus ;
- en utilisant une *répartition de charge par variation du degré de multi-programmation* afin
  - d'enlever des processus de la mémoire principale quand il y a surcharge (*swapping out*) ;
  - de faire revenir des processus en mémoire si le couple de ressources (CPU, mémoire) est sous-utilisé (*swapping in*).

  La répartition globale de la charge consiste à agir sur le degré de multiprogrammation pour maintenir les performances du système dans des limites acceptables. À charge faible ou modérée, la multiprogrammation permet d'augmenter le taux d'utilisation du processeur en utilisant les temps morts dus au blocage ou à l'attente de pages ; à forte charge, on voit apparaître l'effet inverse, qui caractérise l'*écroulement*. Ce comportement suggère l'existence d'une valeur optimale du degré de multiprogrammation, qui maximise le taux d'utilisation du processeur pour une configuration matérielle et une charge donnée. Cela est confirmé par l'étude de modèles analytiques et par l'expérience. Un algorithme idéal de régulation de charge devrait maintenir le degré de multiprogrammation au voisinage de cette valeur optimale.

  Plusieurs critères empiriques d'optimalité ont été proposés. Ils reposent tous sur le même principe : tenter de détecter le début de l'écroulement (par la mesure du taux de défauts de page) et maintenir le système au-dessous de ce point critique.

L'expérience montre qu'il est utile de prévoir un certain amortissement pour éviter le pompage (oscillations brutales provoquées par la régulation lorsque la charge est proche du seuil critique). Cela est notamment obtenu en conservant une réserve de pages physiques libres destinées à absorber les variations transitoires du taux de défaut de page, et en introduisant un temps de retard dans les réactions du régulateur. Un pic transitoire ne provoque donc pas de réaction s'il n'épuise pas la réserve et si sa durée est inférieure au temps de retard. La durée de ce délai et la taille de la réserve doivent être déterminés par l'expérience.

Dans tous les cas de figure, le S.E. doit être capable d'évaluer la charge pour prendre la bonne décision et éviter l'apparition d'un écroulement. Cette évaluation se base sur deux méthodes :

- la méthode de *l'ensemble de travail*,
- l'observation de la *fréquence d'apparition des défauts de page* (PFF ou *Page Fault Frequency*).

4.12.6.2 Algorithme fondé sur l'ensemble de travail

On entretient en permanence pour chaque processus son ensemble de travail ; lors d'un défaut de page, une page virtuelle n'appartenant à aucun des ensembles de travail présents en mémoire est choisie comme victime. S'il n'existe pas de telle page, on réquisitionne les

pages physiques qui contiennent l'ensemble de travail du processus le moins prioritaire (la priorité étant déterminée de façon externe ou par le temps de résidence en mémoire). Un processus ne peut recevoir de mémoire que s'il y a assez de pages physiques libres pour recevoir son ensemble de travail courant. La réalisation de cet algorithme nécessite de pouvoir identifier l'ensemble de travail. Aussi, en dehors de réalisations expérimentales, il n'a été mis en œuvre que de façon approchée.

L'estimation de l'ensemble de travail peut se faire par le biais d'un simple bit d'utilisation qui est forcé à 1 lors de chaque référence à une page. Ce bit existe déjà si on utilise un algorithme de remplacement de type FINUFO ou LRU. Périodiquement le S.E. peut, pour chaque page physique,

— recopier ce bit à l'intérieur d'un mot (en le décalant), et

— forcer à 0 le bit d'utilisation.

Pour chaque page physique, nous avons donc une suite de bits qui donne un historique d'utilisation de la page physique (i désigne l'intervalle de temps entre deux relevés du bit d'utilisation) :

$t-0i$	$t-1i$	$t-2i$	$t-3i$	$t-4i$	$t-5i$
0	0	1	0	0	1

Cet historique peut facilement être utilisé pour construire l'ensemble de travail. Dans notre exemple, cette page appartient à $W(t,T)$ avec $T \geq 2i$, mais elle n'appartient pas à $W(t,T')$ avec $T' \leq i$.

4.12.6.3 Algorithme fondé sur la mesure du taux de défaut de page

Une manière indirecte de détecter que le nombre de pages physiques allouées à un processus donné est insuffisant, consiste à mesurer son taux de défaut de page. L'algorithme PFF est fondé sur ce principe ; quand ce taux dépasse un seuil supérieur, spécifié pour chaque processus, celui-ci reçoit une page physique supplémentaire ; inversement, une page physique lui est retirée si son taux de défaut de page tombe au dessous d'un seuil inférieur.

5 Remarque de bonne conduite

Deux raisons de ne pas trop remplir sa mémoire secondaire (disque dur) :

— En l'absence de partition SWAP, une partie de la mémoire secondaire (quelques Go sur la partition où se trouve votre OS) est utilisée pour étendre la RAM (mémoire virtuelle). Si cette partition est pleine, ce n'est plus possible.

— La fragmentation est beaucoup plus forte sur un disque quasi plein car il y a très peu de choix dans les emplacements libres, et le compactage est d'autant plus difficile.