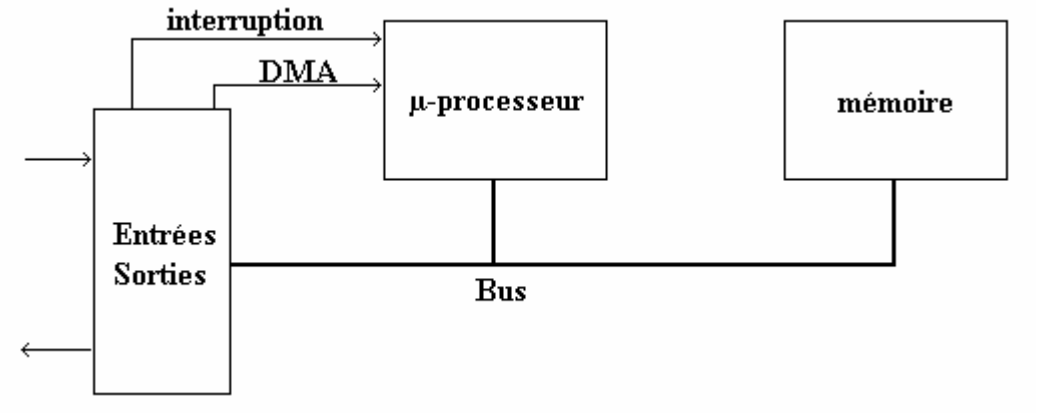
# Rappel

## Structure d'un ordinateur

Globalement la structure d'un ordinateur est composée :

D'une unité de calcul : le µ-processeur[1]
De la mémoire : on englobe mémoire vive et disque
D'entrées / sorties (I/O) : clavier , écran , liaison série , liaison parallèle , etc.
D'un bus : pour les échanges entre le µ-processeur , les mémoires et les I/O




Nous allons donner un tableau des différents temps d'accès des composants d'un ordinateur.

Vitesse d'exécution ou d'accès dans un ordinateur :

| | | | | |
|---|---|---|---|---|
| Le CPU | une instruction toutes les : | cycle de base | 20 ns | |
| La mémoire | dépend du type de mémoire : | cache | 20 ns | (la vitesse du CPU ) |
| | | vive (RAM) | 80 ns | |
| | | disque | 10 ms | |
| Les I/O | dépend du périphérique : | rapide | 50 ns | (carte graphique) |
| | | lent | 100 ms | (mécanique) |

ns : nanoseconde = $10^{-9}$ seconde = 1 milliardième de seconde
ms : milliseconde = $10^{-3}$ seconde = 1 millième de seconde

Nota : Le disque est une mémoire de masse qui est gérée à la fois comme une mémoire et comme une I/O.

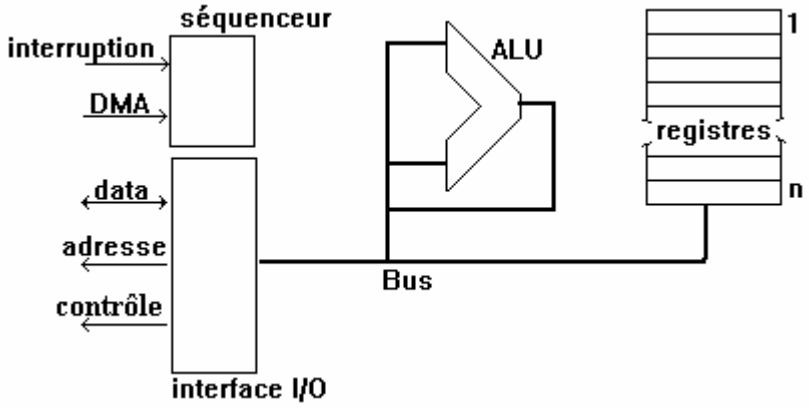
A la vue de ce tableau , il paraît évident qu'il faut harmoniser les vitesses des dispositifs entre eux afin qu'il se comprennent , cela se fait toujours en ralentissant le plus rapide afin de suivre le plus lent. Sinon il faut inventer des systèmes de buffer (boîte au lettre) pour que le plus rapide remplisse/lise ce buffer et fasse autre chose pendant que le plus lent lise/remplisse le buffer.

Nous allons détailler les différentes composantes d'un ordinateur.

[1] Appelé aussi CPU , Central Process Unit (unité centrale de calcul).

# Le µ-processeur

Le µ-processeur est l'unité de calcul de l'ordinateur. Il est à remarquer que sa structure est la même que celle d'un ordinateur.




L'ALU est le centre de calcul du processeur , c'est lui qui , en fonction des instructions reçues , exécute les opérations élémentaires , ex. : or , and , add , mul , etc. Toutes les opérations se font avec les valeurs stockées dans les registres (il existe des processeurs qui savent exécuter une opération en mémoire mais c'est extrêmement compliqué à programmer).

Ce couplage entre l'ALU et les registres correspond au processeur et à la mémoire dans la vision de l'ordinateur.

Une des caractéristiques d'un processeur est le nombre et la taille (en bits) de ses registres : quand on dit que tel µ-processeur est un 16 bits c'est qu'en fait la taille de ses registre fait 16 bits de large , c.a.d. 2 octets.

Les processeurs RISC[2] travaillent seulement en écriture / lecture (wr/rd) entre la mémoire externe et les registres. Ce qui oblige à : lire les valeurs dans la mémoire et les mettre dans les registres , faire les opérations , écrire le résultat des registres dans la mémoire. Cela peut sembler long , mais en fait ce sont des instructions élémentaires et on sait les exécuter très rapidement. C'est pourquoi les processeurs RISC remplacent progressivement les processeurs CISC[3] dans les ordinateurs.

L'interface I/O permet le dialogue avec les autres composants de l'ordinateur. Pour ce faire un bus de data permet d'écrire / lire les informations contenues à l'adresse spécifiée par le bus adresse , les signaux de lecture/écriture sont dans le bus de contrôle. L'interface doit aussi assurer la mise en forme des signaux , le timing des signaux entre eux , la mémorisation des états , etc.

Le séquenceur séquence le déroulement des instructions et prend en compte les demandes d'interruptions et de DMA.

Quand une interruption arrive sur le processeur il doit :

- terminer l'instruction en cours.
- voir si l'interruption est autorisée , si oui , basculer dans un état de traitement des interruptions:
    - mémoriser l'adresse où il est , et sauvegarder le contexte. Placer l'adresse courante dans la pile ainsi que tous les registres utiles de telle sorte qu'à la fin de l'IT on puisse retrouver le déroulement normal de la séquence.
    - exécuter le code placé à l'adresse correspondant à l'interruption reçue (le code de l'IT 1 est en 100 , celui de l'IT 2 en 200 , etc.).

---

2 RISC : Reduced Instruction Set Computers (processeurs à jeu d'instructions réduit).

3 CISC : Complex Instruction Set Computers (processeurs à jeu d'instructions complexe).

C'est exactement comme si , lors d'une interruption , on arrêtait le processeur pour le remplacer par un autre , et qu'à la fin on retrouvait le précédent.

Le DMA[4] permet d'écrire/lire directement dans la mémoire sans passer par le processeur. C'est en fait un circuit qui travaille en alternance avec le processeur pour accélérer les échanges entre périphériques rapides et mémoire sans encombrer inutilement le processeur. Il faut évidemment qu'il prévienne le processeur que c'est à lui d'utiliser le bus de l'ordinateur afin qu'il n'y ait pas de conflit. Le processeur doit donc avoir une entrée lui demandant de libérer le bus.

# Les entrées - sorties

Les I/O permettent au processeur de dialoguer avec l'extérieur , c'est en fait un moyen d'échanger des informations entre la mémoire et les périphériques. ex. imprimer un buffer mémoire , établir une liaison série afin de remplir un buffer mémoire , etc.

## Les interruptions

Une I/O peut être extrêmement importante , il ne faut pas perdre l'information , de plus on ne sait pas quand elle arrive dans le temps. Le seul moyen d'être sûr de ne pas perdre de temps à tester si l'information est là , c'est d'utiliser une interruption. Quand le dispositif doit envoyer/recevoir une information il "fait" une IT au processeur. Celui-ci interrompt son programme , exécute le code de l'IT , et reprend le déroulement de son programme.

ex. le clavier.

Un utilisateur frappe sur une touche à un moment donné , le "scan code" de la touche est stocké dans le gestionnaire de clavier. Celui-ci "envoie" une IT au processeur. Lequel sauvegarde son contexte , et exécute le programme de gestion de clavier qui est : lecture du gestionnaire de clavier , sauvegarde de l'information dans un tampon mémoire réservé au code du clavier , et retour au programme normal.

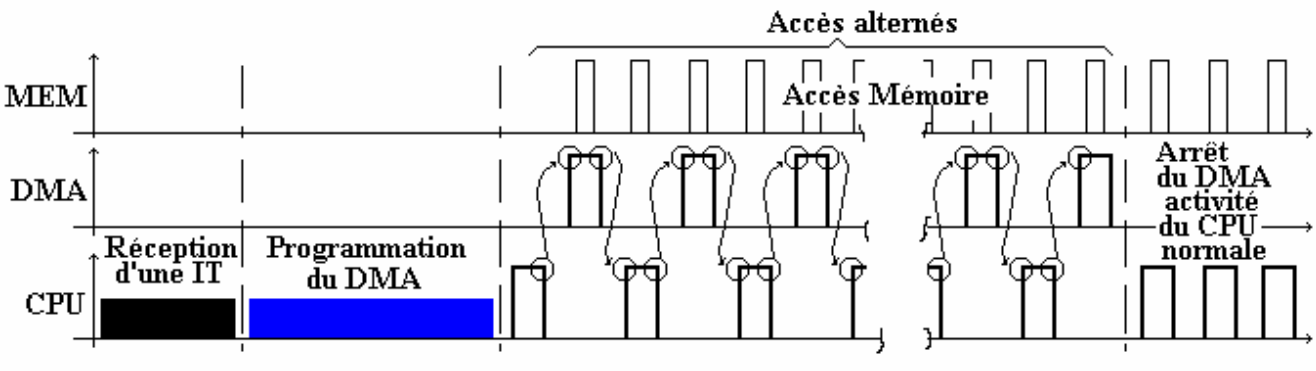
## Le DMA

Il est utilisé souvent avec les interruptions. Cependant il sert dans le cas où c'est une suite d'informations qui doit être transférée.

ex. Le gestionnaire de disque.

Quand on demande la lecture/écriture d'un fichier on découpe le fichier , et on lit/écrit des morceaux d'informations d'une taille définie (512 octets en général). Le contrôleur de disque reçoit l'ordre du processeur de lire/écrire un buffer mémoire placé à l'adresse $\alpha$ dans le secteur disque $\gamma$. Le processeur a programmé le DMA pour qu'il commence son écriture/lecture à l'adresse $\alpha$. Quand le contrôleur de disque détecte que le secteur voulu est arrivé , il demande au DMA de lire/écrire entre la mémoire et le disque. Il fait cela pour tous les octets de façon contiguë dans le buffer mémoire. Le processeur peut toujours travailler entre les moments d'utilisation du DMA. Quand le contrôleur a fini de lire/écrire le buffer il envoie une interruption au processeur pour lui dire qu'il a fini. Au processeur de recommencer le processus.

[4] DMA : Direct Acces Memories (acces directe à la mémoire).



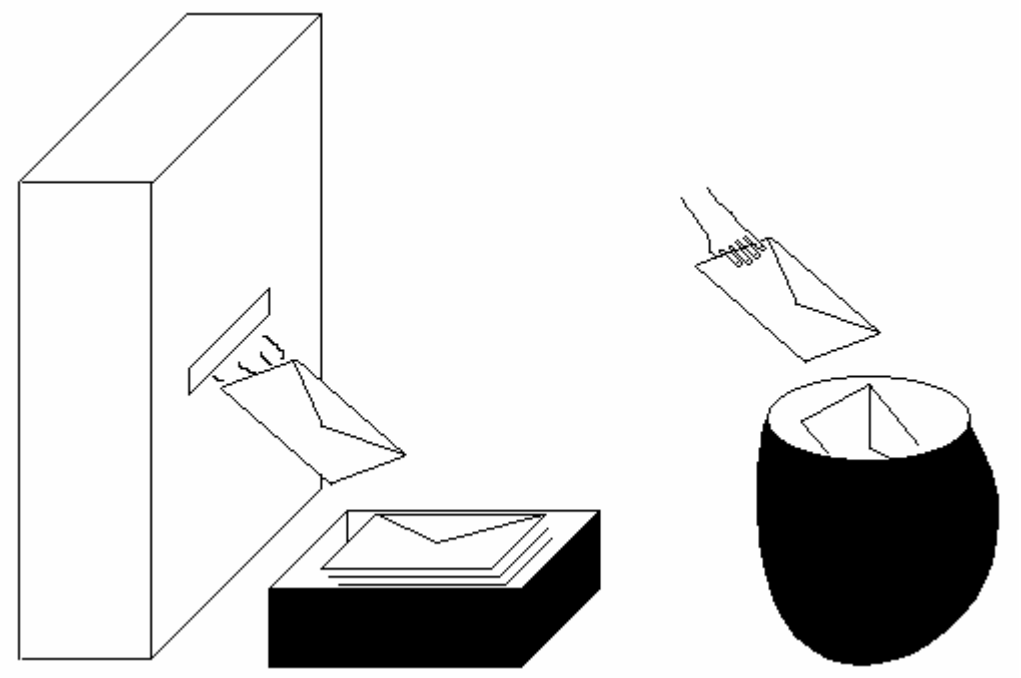

Le DMA sert souvent dans les accès aux périphériques en mode "rafale" tel que les lectures/écritures disque , disquettes , réseaux , streamer , etc.

C'est un dispositif extrêmement pratique qui décharge le CPU des taches fastidieuses de contrôle des I/O mais il prend un temps de lecture/écriture pour chaque accès. Cela permet d'avoir un début de notion de multitâche , pendant que le CPU fait son travail , le DMA exécute les I/O.

C'est typiquement ce qui permet de ne pas se préoccuper des vitesses absolues de chaque partie.

La partie la plus rapide remplit/lit un buffer.

Le DMA lit/remplit le buffer pour la partie la plus lente



# Conclusion

Dans tout ordinateur "bien né" il existe au minimum un CPU avec un contrôleur d'interruption et au moins un DMA. Cela forme le coeur de l'ordinateur , à tel point que certains processeurs possèdent le gestionnaire d'IT et les DMA intégrés (micro-controleur).

Ex. les contrôleurs des imprimantes laser , les contrôleurs de disque , etc.

On appelle le reste  mémoire , I/O ( disque , disquette , bande , liaison série , etc.)  les ressources de l'ordinateur.

# Les processus

**Tous les programmes s'exécutent toujours en mémoire.**

## Rôle du système

### Utilité

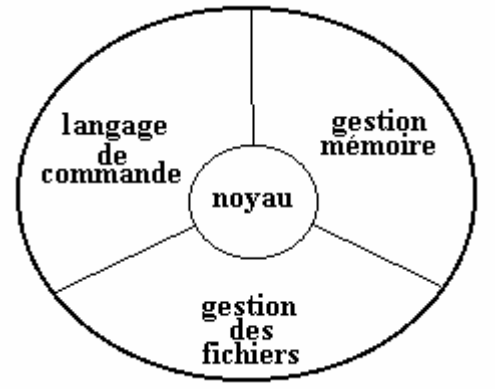
Le système est un programme très particulier. L'utilisateur n'a , en général , pas le droit d'y accéder directement. Il sert à faire marcher de façon optimale les différentes ressources de l'ordinateur (gestion des ressources).

L'utilisateur ne peut que demander , suivant une syntaxe précise , l'exécution d'une commande au système. Pour cela le système est muni d'un programme spécifique qui s'appelle un interpréteur de commande.

Le système doit gérer "au mieux" l'occupation mémoire de l'ordinateur , c.a.d. avoir toujours de la mémoire disponible au cas où il y aurait une demande de mémoire (nouveau programme à exécuter , demande d'allocation de mémoire durant le déroulement d'un programme, etc.).

Il doit aussi gérer le "système de fichier" , c.a.d. comment écrire et lire les fichiers stockés sur le disque , ou tous autre périphérique.

En synthèse , le système doit gérer la mémoire , les fichiers , et avoir un langage de commande.




### Le noyau

Le noyau ( kernel ) permet de gérer les 3 composantes d'un système. Ces composantes peuvent être changées , cependant il faut assurer un dialogue entre elles.

### Fonctionnement

Quand on démarre l'ordinateur il y a un petit bout de programme , appelé "boot" , qui permet d'aller chercher sur disque , ou disquette , le programme système et le charger en mémoire à une place bien précise. Une fois que le système est résident en mémoire il est "lancé" , c.a.d. que le CPU commence à exécuter les instructions de ce programme. Celui-ci initialise toutes les ressources et configure l'ordinateur par l'intermédiaire de fichiers de configuration , afin que l'utilisateur puisse travailler sur son poste.

### Langage de commande

Afin de pouvoir exécuter un programme , un utilisateur doit dire au système "je désire que tu charges ce programme en mémoire et que le CPU exécute celui-ci". Cela se fait en utilisant le langage de commande , en général en tapant simplement le nom du programme à exécuter. L'utilisateur peut vouloir manipuler les fichiers (copier , effacer , renommer , etc.) , pour cela il faut aussi que le langage de commande lui fournisse les fonctionnalités désirées.

# Les programmes

## Découpage fonctionnel

Un programme se compose , grossièrement , de 4 parties.

Le code d'exécution (code).
L'espace des variables utilisées par le programme (data).
La pile (stack).
Le "tas" qui est un endroit où on peut allouer de la mémoire dynamiquement (heap).


Le code est l'endroit où s'exécute le programme en mémoire , il peut être à accès en lecture seulement , le CPU ne faisant que lire les instructions.

Les datas sont l'endroit où sont placées les variables que l'on a déclarées dans le source , accès peut se faire aussi bien en lecture qu'en écriture.

Le stack est l'endroit où le programme passe ses paramètres lors d'appel de sous-programme , c'est un emplacement où il faut pouvoir lire et écrire.

Le heap est l'emplacement où le programme peut demander au système d'exploitation l'allocation (mise à disposition) de bout de mémoire.

| emplacement | lecture | écriture |
|---|---|---|
| code | x | |
| data | x | x |
| stack | x | x |
| heap | x | x |

## Chargement en mémoire

Quand un utilisateur , ou un programme , demande l'exécution d'un programme , il demande au système de réserver une place en mémoire pour le charger. En fait ce que l'on appelle un "exécutable" , c'est un fichier qui contient toutes ces indications en plus du code et des datas. Le système lit le fichier , au début il y a toutes les tailles nécessaires au bon fonctionnement. Le système sait donc quelle taille mémoire il doit allouer pour le code , les datas , le heap , et le stack. Si la place est disponible , alors le fichier est lu entièrement et ses différents constituants sont écrits à leur place en mémoire , sinon il y a un message d'erreur (place mémoire insuffisante).

# La mémoire

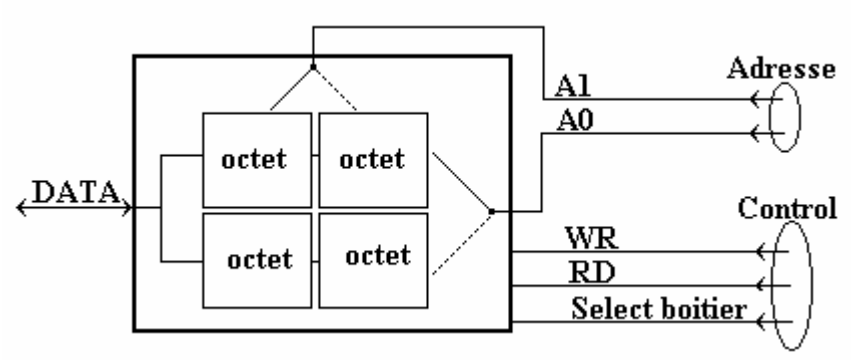
La mémoire est l'endroit où le CPU range les informations déjà traitées , ou à traitées. Il faut donc que ce soit un rangement très rapide afin de ne pas retarder le processeur.

La façon la plus simple d'envisager le rangement , c'est de le faire dans des boîtes élémentaires que l'on appellera un "mot". La mémoire est un composant électronique composés d'une puissance de 2 d'octet[5]. Le nombre d'octets dans une mémoire d'ordinateur est un multiple d'un méga-octet , ce qui ne fait pas 1 million d'octets mais $2^{20}$ = 1 048 576 octets. Il est impératif que l'on puisse accéder à n'importe quel octet de la mémoire , c'est ce que l'on appel une méthode accès aléatoire.

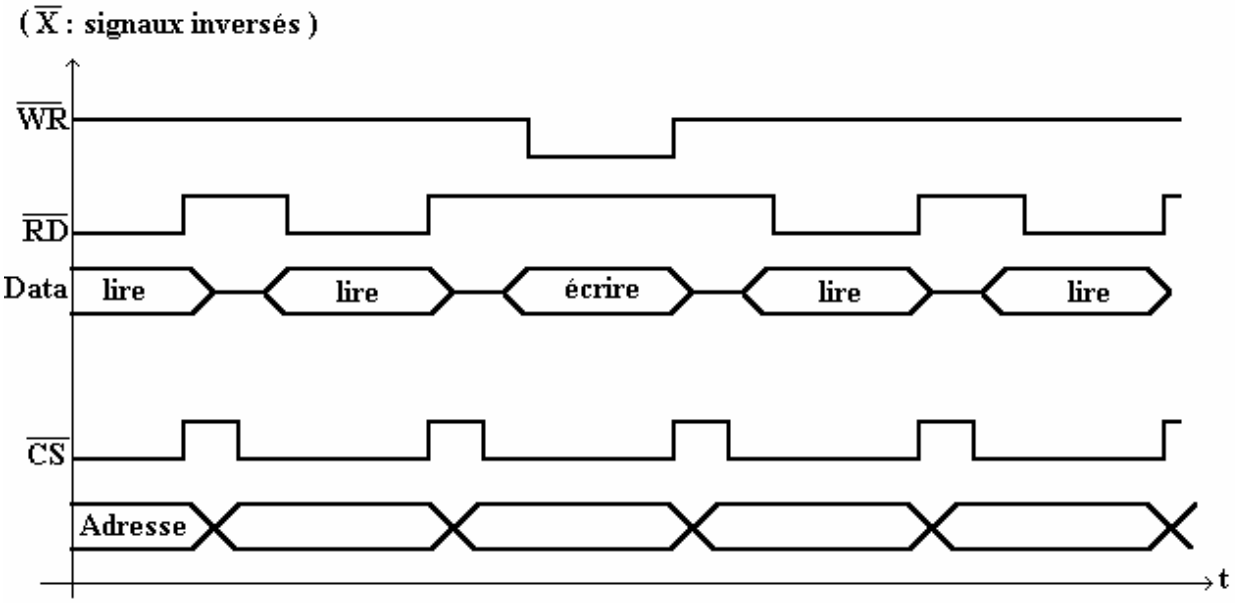
## L'adressage

La mémoire est une suite d'octets qui sont tous sélectionnables. On peut lire ou écrire dans une mémoire. Afin de choisir 1 octet parmi N , il existe un mécanisme d'adressage simple :

LE CPU , ou tout autre dispositif qui a accès à la mémoire , envoie l'adresse (il y a un décodeur qui dit quel boîtier mémoire on veut). S'il écrit , il envoie les datas et l'ordre d'écriture , s'il lit , il envoie l'ordre de lecture et il récupère les datas.




*Mémoire de 4 octets et mécanisme d'adressage*



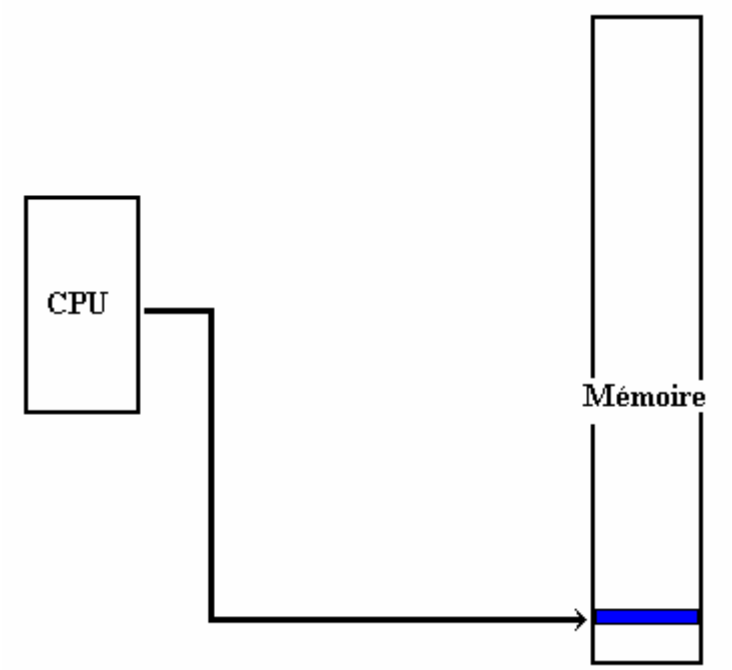

[5] Certains vieux ordinateurs avaient une autre organisation du mot mémoire élémentaire , cependant à partir des années 1980 tous se sont normalisés et maintenant l'octet est l'unité de mémoire. On parle de µ-processeurs à 16 , 32 , 64 bits ,ils forment en fait des multiples de 2 d'octets , respectivement 2 , 4 , 8 octets

*Chronogramme des signaux sur une mémoire*

## Adressage linéaire

C'est l'adressage le plus simple. Le CPU donne explicitement l'adresse de la donnée désirée.

| | | |
|---|---|---|
| Avantages | : | Simple , rapide |
| Inconvénients | : | La mémoire est vue tout entière comme un bloc. |




Cet adressage a été utilisé sur les premiers systèmes , de type mono-tâche (DOS) , mais il est trop primitif au niveau gestion de la mémoire pour être utilisable dans la gestion multitâches.

Il présente néanmoins un avantage certain quant à la simplicité et la rapidité d'accès. Il suffit de donner une adresse pour avoir aussitôt la donnée.

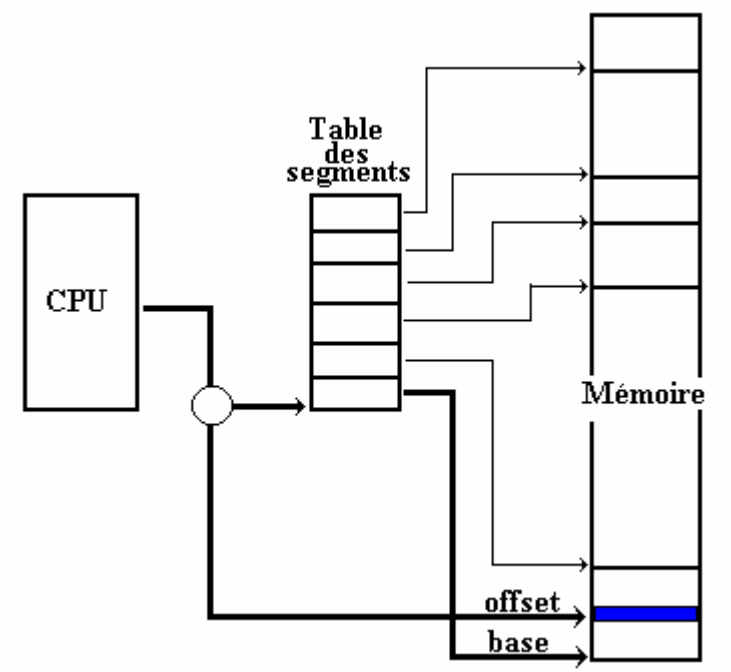
Certains systèmes, dont la vitesse d'accès est le critère le plus important, utilisent ce mode d'adressage : systèmes temps réel axés sur le calcul scientifique , systèmes pour automates industriels.

## Adressage segmenté

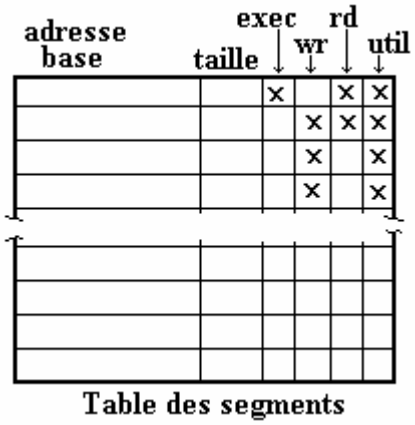
Le CPU donne l'adresse de la donnée désirée comme étant un couple : indice dans la table des segments et offset. Il faut auparavant avoir initialisé la table des segments. Les segments ont une taille variable.

Avantage : localisation des tâches par segments
Inconvénients : Indirections , donc accès plus long. Taille variable.

Quand on demande au système de placer un programme en mémoire , il est intéressant de pouvoir définir précisément des zones pour le code , les variables , la pile , etc. Ces espaces mémoires ne sont pas obligatoirement contiguës , il est donc intéressant de pouvoir créer une table d'indirections qui indique la zone (segment) où se trouve la partie concernée et se déplacer à l'intérieur de cette zone. On doit pouvoir définir , pour ces zones mémoires , des attributs d'écriture , lecture , exécutable , etc.




Il est important que la table des segments contienne des champs d'indications pour les attributs accès, d'occupation , etc., et surtout la taille du segment si les segments sont à taille variable[6].



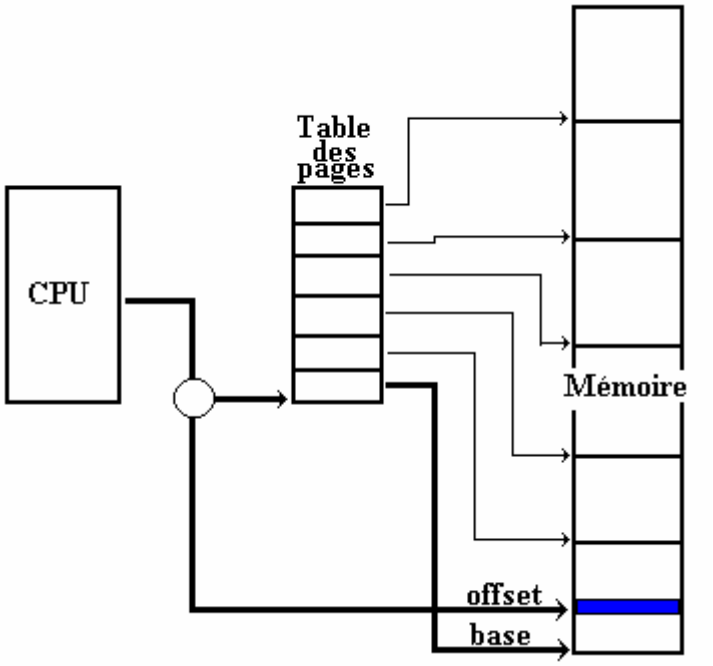

[6] Certains µ-processeurs ont une gestion de segments à taille fixe (ex. 80x86) , d'autres considèrent le couple {adresse ; taille} comme un segment (ex. Z8000)

## Adressage paginé

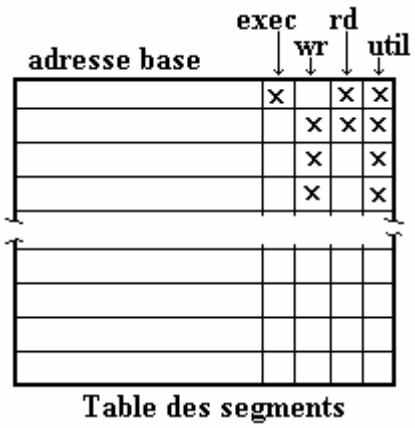
Le CPU donne l'adresse de la donnée désirée comme étant un couple : indice dans la table des pages et offset. Il faut auparavant avoir initialisé la table des pages. Les pages ont toutes une taille fixe.

| | | |
|---|---|---|
| Avantage | : | facilité de gestion de la mémoire ( pages pas forcément contiguës). |
| Inconvénients | : | Indirections , donc accès plus long. Plusieurs pages pour une tâche. |

La "vision" que l'on peut avoir de la mémoire est une suite linéaire et contiguë d'espace. Mais cette linéarité n'existe qu'au travers une table des pages , physiquement elles peuvent être totalement dispersées en mémoire.




Il est important que la table des pages contienne des champs d'indications pour les attributs accès , d'occupation , etc.



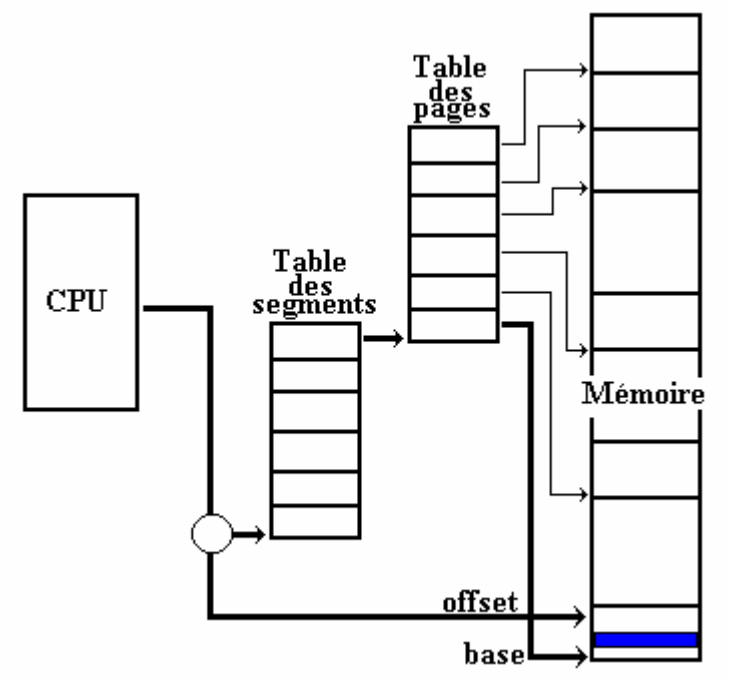

La différence entre la segmentation et la pagination , c'est qu'un segment contient toute la partie concernée du programme. Cela peut être une quantité mémoire importante ex. 800 ko , alors que la page est une unité fixe d'une quantité x de mémoire.

Quand on veut mettre une des parties d'un programme en mémoire , on cherche le nombre de pages qui sont nécessaires. Il est probable que la quantité mémoire ainsi prise soit supérieure à celle demandée. Afin de minimiser l'espace perdu , il faut que la taille des pages soit la plus petite possible. Mais pour avoir une simplicité de gestion des pages , il faut que la taille soit la plus grande possible (cf. un segment). Ces exigences contradictoires définissent un optimum qui caractérise un système (l'ordre de grandeur pour une taille de page est situé entre 512 octets à 2 ko)

## Segmentée et paginée

La segmentation est très intéressante car elle permet de placer et gérer un programme de façon simple et pratique (niveau gestion des processus). La pagination , elle , est très pratique pour bien gérer la mémoire (niveau gestion mémoire). Il est donc pratique de pouvoir combiner les 2 afin de profiter des avantages de chacun.

| | | |
|---|---|---|
| Avantage | : | facilité de gestion de la mémoire. |
| Inconvénients | : | 2 indirections , donc accès plus long |




Deux indirections , c'est beaucoup. Les processeurs modernes stockent une partie des tables dans des registres internes afin d'accélérer l'accès. Une autre technique utilise de la mémoire très rapide (cache) afin de ne pas trop pénaliser le déroulement des séquences.

Cependant, il faut remarquer que l'indirection ne change que lorsqu'il y a changement de segment ou de table ; dans les autres cas , seul l'offset varie. Il est donc plus intéressant de sortir toujours la même base et de ne recalculer les indirections que lorsqu'on change de segments ou de pages. Grâce à cette méthode , les indirections ne sont faites que peu de fois , proportionnellement.

La détection de dépassement de segment ou de page se fait grâce à une comparaison entre le segment et la page stockés dans 2 registres , et les valeurs calculées à chaque instruction. S'il n'y a pas concordance entre ces valeurs , il faut un mécanisme qui prévienne le processeur d'une erreur de segment ou/et de page. Le seul mécanisme possible s'apparente à une interruption. Il est appelé segment/page violation. Lorsqu'une violation est signalée , il faut que le processeur recalcule le segment/page et reprenne l'instruction courante. Cependant il a déjà commencé l'exécution de cette instruction. Si elle est codée sur plus de 1 mot machine , il faut un mécanisme de mémorisation de début d'instruction , ce qui n'est pas simple du tout. C'est pourquoi ce mécanisme ne s'implante simplement dans le silicium que sur des processeurs RISC. Les processeurs CISC demandent beaucoup plus (mémorisation de début d'instruction , etc.) , ce qui entraîne une plus grande complexité du "chip" et donc un prix plus élevé. De plus , la plus grande complexité de calcul entraîne un ralentissement , c'est une des raisons pour lesquelles les RISC sont en général plus rapides que les CISC.

# Allocation mémoire

Nous venons de voir, très succinctement , l'installation par le système d'un processus en mémoire. Il y a un autre cas de demande de ressource mémoire.

## Demande statique / dynamique

### Statique

Quand un programmeur écrit un programme , il manipule des informations qui sont des données. La taille de celles-ci peut lui être connue des le début , ou inconnue car déterminée par des circonstances extérieures.

ex.

Utilisation d'un tableau de x nombres , chaque nombre ne dépassant pas la taille y (c'est la partie data) , et initialisation de ce tableau à une valeur $\alpha$ (c'est la partie code).
Le compilateur sait très bien faire une multiplication , et lors de l'installation du processus par le système il y a une indication de taille à réserver dans l'espace data.

**C'est ce que l'on appelle allocation statique.**

On indique au compilateur une taille de data à réserver et il se charge de transmettre l'information au système. Cette allocation se fait au début , lors du chargement de programme par le système.

### Dynamique

On ne peut pas toujours savoir quelle sera la taille de la donnée.

ex.

L'utilisation d'un tableau de x chaînes de caractères (c'est la partie data) , et lors du déroulement du programme , le remplissage de ces chaînes (c'est la partie code , **mais aussi demande de mémoire**).

Lorsque le programme se déroule , il demande à l'utilisateur de rentrer des informations (texte , adresse , etc.). On ne peut pas savoir à l'avance la taille de ces informations. On ne peut pas réserver de la place à l'avance , il faut le faire une fois que l'on "sait". Une "technique" utilisable dans ce cas , c'est de déclarer un "buffer" (emplacement mémoire) très grand , et quand il y a une saisie à faire , de remplir ce buffer. Une fois que la donnée est rentrée dans le buffer , on calcule sa taille et on ***alloue de la mémoire*** juste à la taille désirée. On recopie alors le buffer afin de le libérer pour une réutilisation future.

**C'est ce que l'on appelle allocation dynamique**

Cette allocation se faisant en cours de déroulement du programme , il faut donner un ordre au système qui demande de réserver $\omega$ emplacement mémoire. Cette commande est une instruction fondamentale de tout système.

On peut pousser plus loin la logique de l'ex. que nous venons de voir.

Le buffer peut lui aussi être alloué dynamiquement , pour faire une saisie il faut donc :

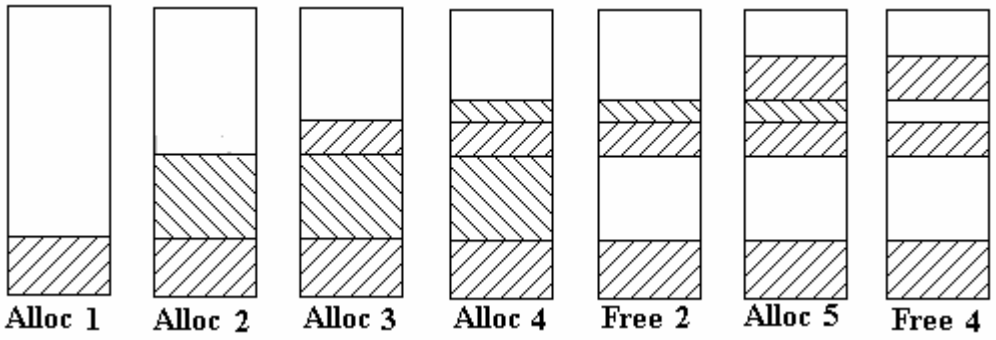
- Allouer la place pour le buffer
- Saisir les informations dans le buffer.
- Calculer la longueur.
- Allouer la place correspondante à cette longueur.
- Copier le contenu du buffer dans l'allocation faite.
- On peut alors ***libérer*** la place occupée par le buffer.

# Technique d'allocation mémoire

Dans tous les cas le système doit savoir donner de la mémoire au programme qui lui en demande , ou lui dire qu'il n'y a plus de mémoire disponible.

## Phénomène de fragmentation

Quand le système donne de la mémoire (suite à la demande d'un programme) , il prend cet espace suivant une stratégie. Un programme bien fait , doit rendre après utilisation l'espace qu'il a demandé . Il peut redemander un autre morceau de mémoire , avant ou après avoir libéré le précédent. Après demande et libération successives , on arrive à un phénomène qui s'appelle fragmentation de la mémoire.




On peut donc rechercher des stratégies de meilleure occupation mémoire afin , soit d'éviter la fragmentation , soit de la minimiser au maximum.

Par étude statistique , on s'est aperçu que les demandes d'allocations mémoire ou les libérations étaient localisées dans le temps. Cela veut dire que la fragmentation augmente encore plus , comme si les demandes groupées dispersaient plus que des demandes réparties (c'est en fait le déroulement séquentiel du programme qui temporellement groupe les demandes et "charge" le système).

On peut allouer/libérer de la mémoire par des méthodes "statiques" ou des méthodes "dynamiques".

## Méthodes "statiques"

On range sous ce vocable les méthodes qui conservent dans une "table" l'évolution de la disponibilité mémoire. On place dans cette table l'adresse de début de tous les blocs alloués , mais aussi l'adresse de début de tous les morceaux libres et leurs tailles.

### Première zone libre

Quand on veut allouer de la mémoire , on cherche le premier emplacement qui contient la demande. Une fois trouvé , on le prend , quelle que soit sa taille. Cette méthode a le mérite de la simplicité , mais demande une recherche séquentielle. Cette stratégie est appelée première zone libre (first-fit).

### Meilleur ajustement

La méthode du meilleur ajustement (best-fit) consiste à sélectionner la plus petite zone de taille suffisante. Cela permet de minimiser la fragmentation , mais on a plus que des petits morceaux quasiment inutilisables. Cette méthode est plus lente que la précédente. On peut tenter de l'accélérer , en triant des zones par ordre croissant de taille dans le tableau les décrivant , et faire une recherche dichotomique.

### Plus grand résidu

La stratégie du plus grand résidu (worst-fit) est basée sur l'idée qu'en prenant la plus grande zone , le reste sera de taille conséquente , grâce à quoi on préparera la satisfaction d'une future demande.

Les méthodes d'allocation mémoire statiques sont utilisées par les langages dit "structurés" (Pascal , C , Modula , Ada ,etc.)

## Méthodes "dynamiques"

Quand on utilise une méthode statique et quoi que l'on fasse , on a morcellement de la mémoire , et il arrive un instant où on a besoin d'un espace mémoire qui n'existe plus de façon contiguë. Aucune des parties libres n'est suffisante pour satisfaire la demande alors que la somme de tous les espaces libres permettrait de répondre au problème.

L'idée c'est , lors d'une libération d'allocation , d'effectuer un déplacement de tous les blocs occupés afin de tasser la place globalement occupée et donc d'avoir la plus grande place disponible en un seul morceau.

### Garbadge-Collector

Le "ramasse-miettes" est une stratégie de déplacement des blocs occupés , qui s'effectue lorsqu'un des blocs vient d'être libéré. Ce déplacement se fait automatiquement sans que le programmeur ait à demander quoi que ce soit , c'est pourquoi c'est une technique que l'on qualifie de dynamique (les blocs n'ont pas une place fixe).

Le compactage des zones occupées peut se faire suivant différentes stratégies.

ex. On a un découpage de la mémoire à un certain instant qui est comme suit :


On veut compacter les blocs utilisés pour avoir , en fin de libération , 11 espaces mémoire libres.

a) On tasse tous les blocs occupés au début. Cela demande 3 déplacements de blocs.


b) On comble les premiers espaces libres avec tous les blocs occupés possibles.


b) On cherche le minimum de déplacement à faire. Cela implique que la zone libre n'est pas obligatoirement à l'extrémité de la mémoire.

Ces stratégies de compactage ont toutes comme but de donner un espace mémoire disponible maximum. Elles ont toutes par contre le désavantage de :

- Prendre du temps (il faut recopier un (ou des) morceau de mémoire).
- Poser le problème de retrouver les éléments ainsi déplacés. Ce qui implique d'avoir des langages qui sont adaptés à cette conception (il ne faut pas utiliser d'adressage absolu).

C'est pourquoi les langages qui sont conçus pour utiliser ces techniques de compactage sont appelés des "langages dynamiques". Le premier langage fut le *Lisp* inventé dans les années 50 par M$^{c}$ Carty. Un autre langage qui est en plus un langage objet s'appelle *Smalltalk*. Ce langage fut conçu par Alan Kay dans les années 70 au Xerox Palo Alto Research Center (le fameux PARC de Rank Xerox). Ces langages ont en commun la caractéristique de ne pas avoir de données typées. Dans les années 85 Bertrand Meyer a inventé un langage objet , Eiffel , qui , bien que dynamique , possède en plus la notion de donnée typée.

## Swaping

Nous avons vu l'allocation/libération d'espace mémoire continu. En reprenant la description de la segmentation et de la pagination qui furent décrites précédemment , nous pouvons étendre encore nos moyens.

Lorsqu'on demande une allocation mémoire et que tout est pris , on peut espérer que le système soit assez bien conçu pour qu'il y ait une trace (marquage) des pages les moins utilisées , et qu'ainsi le système les range sur le disque , afin de libérer de la place que l'on peut récupérer pour avoir ce si précieux espace mémoire , libre.

Dans un système mono-tâche la gestion des pages et des segments n'a pas un grand intérêt , seule l'occasion d'avoir une mémoire virtuelle illimitée peut nous ameer à y avoir recours. Par contre , dans un système multitâche ou/et multi-utilisateur , a segmentation pagination est un mécanisme pratiquement obligatoire.

De même qu'il y a des stratégies d'allocation mémoire , ce qui se fait au niveau des multiples d'un mot mémoire , il y a des stratégies d'allocation/libération des pages. On peut bien sûr considérer que les stratégies s'appliquent autant aux segments qu'aux pages , mais pour des raisons de simplicité nous allons travailler au niveau des pages. La généralisation peut se faire en considérant qu'un segment est un groupe de pages , et en reprenant le raisonnement pour les segments.

Un programme peut ne pas être chargé entièrement en mémoire. Si on a chargé une partie du programme dans des pages , il se déroule normalement jusqu'à ce qu'il dépasse la dernière page. Il faut donc que le système remplace les premières pages par les pages suivantes , afin que le programme continu.

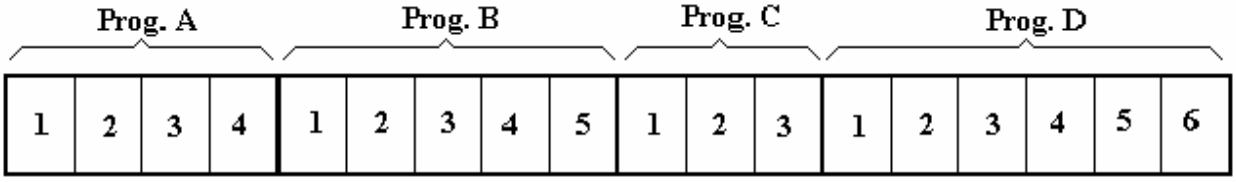
## Gestion de pages

Ce système permet de ne prendre que quelques pages (minimum 1) pour un programme , le reste étant sur disque. On peut ainsi charger plusieurs programmes , qui ont tous en mémoire une partie. A la charge du système de savoir quand un programme va demander la suite qui n'est pas en mémoire , et gérer le remplacement des pages utilisées par celles qui vont suivre. Cette technique de gestion de pages s'appelle le swapping.

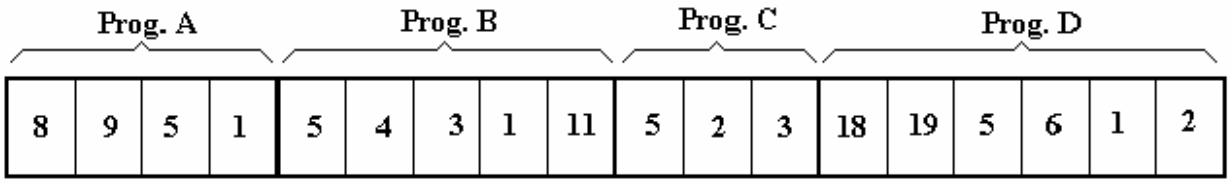
Chaque fois qu'il faut remplacer une page il faut :

- Ecrire la page dans le disque si elle a été modifiée , afin de ne pas perdre les informations.
- Demander au contrôleur de disque d'aller chercher la partie du programme à remplacer , et le ranger dans la page libérée.

ex. Il y a 4 programmes qui "tournent" , les programmes A , B , C , et D qui sont respectivement de taille 12 , 20 , 10 , 30 pages. La mémoire contient 18 pages. Le système a chargé les premières pages des programmes.



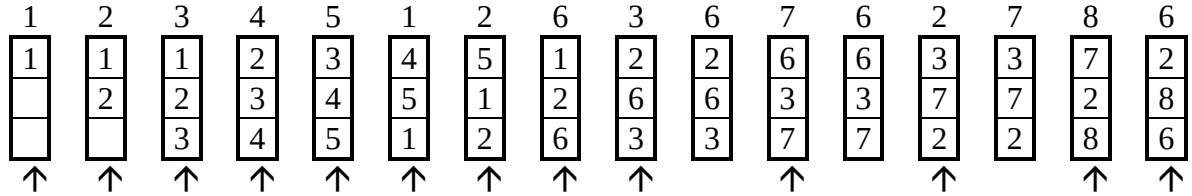

Les programmes se déroulant normalement , il faut remettre à jour les pages. On peut arriver à une situation comme celle dessinée.



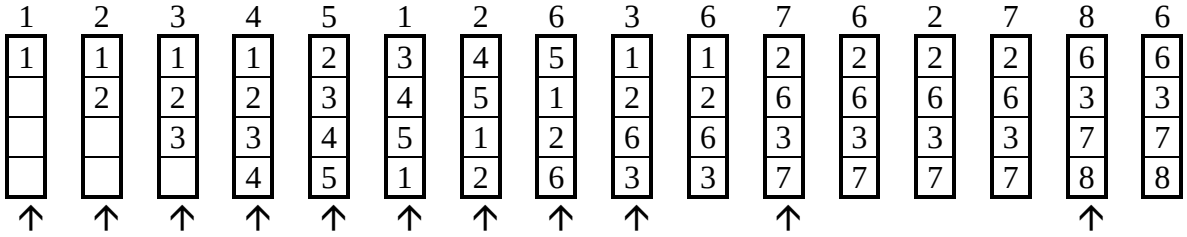
## Fifo

Une des stratégies possibles , c'est : premier entré premier sorti (FIFO first in first out). On remplace le premier qui fut écrit par celui qui est nécessaire. Celui-ci devient le dernier rentré. Il faut pour cela garder dans la table des pages , une numérotation des pages , afin de pouvoir gérer la fifo. Cela est fait en gardant la date. On choisit alors celle qui a la date la plus ancienne comme étant celle à changer. On peut aussi gérer la table comme une FIFO.

ex. Le programme se déroule suivant la séquence < 1 2 3 4 5 1 2 6 3 6 7 6 3 7 8 6> et on a 3 pages.




On a la même séquence mais avec 4 pages.

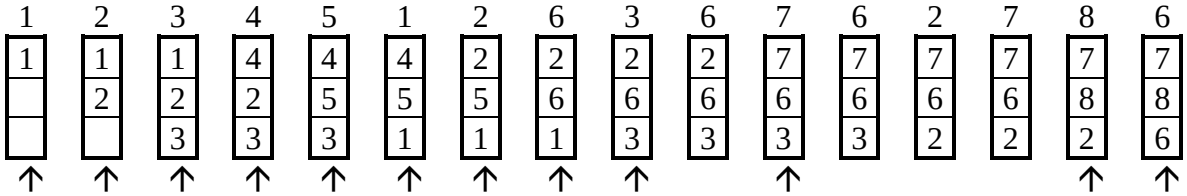



Cependant cette méthode , pour simple et pratique qu'elle soit , ne fournit pas un optimal de gestion de page.

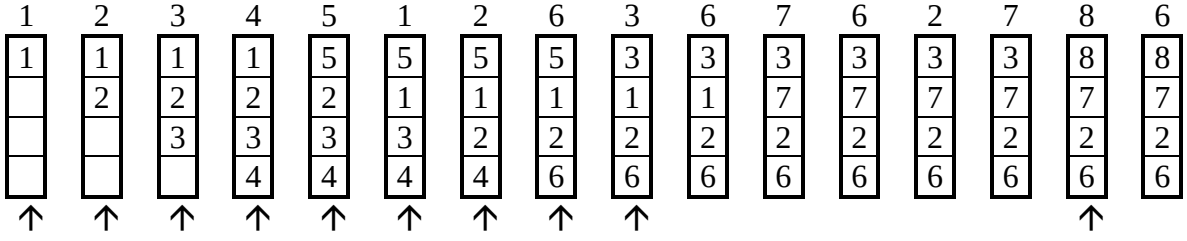
## LRU

Si on considère qu'il y a un principe de localité , c.a.d. que ce sont les pages les plus récemment utilisées qui auront le plus de chance d'être réutilisées , alors ce sont donc les pages les moins récemment utilisées qui doivent être remplacées , d'où le nom de (LRU Last Recently Used).

ex. Même séquence que FIFO < 1 2 3 4 5 1 2 6 3 6 7 6 3 7 8 6> et on a 3 pages.




On a la même séquence mais avec 4 pages.




UNIX utilise l'algorithme LRU dans un contexte de tampons mémoires. Windows sous MS-DOS aussi.

Ces stratégies de gestions de pages sont fondamentales pour un système multitâches multi-utilisateurs. Cependant il y a des précautions à prendre.

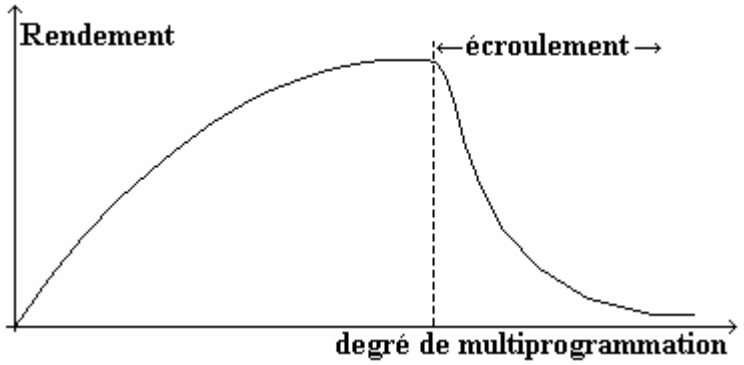
Il doit sembler évident que le "swapping" de pages demande du temps.

- Sauvegarder la page si elle a été modifiée.
- Demander le chargement de la nouvelle page.

Quand le système demande le chargement d'une nouvelle page , il ne doit pas attendre de l'avoir , cela serait une perte de temps. Il doit passer à une autre tâche , donc exécuter le code d'une autre page , mais il peut se faire qu'il advienne encore un défaut de page , et le système recommence alors la séquence de changement de page.

Il faut donc "régler" un système c.a.d. avoir un ratio nombre de pages par application , nombre d'applications , permettant d'être optimal. Si on donne peu de pages par application , on écroule le système. Si par contre on donne beaucoup de pages , on ne pourra pas satisfaire une demande nouvelle de page qui pourrait survenir.

On trace la courbe du rendement en fonction du nombre de pages utilisées par chaque application.




On ne doit pas forcément donner un nombre identique de pages à chaque application. Il semble normal de donner plus de pages à une grosse application qu'a une petite. Cependant la taille des applications n'est pas une indication suffisante. On peutavoir un gros programme qui se déroule séquentiellement et un petit programme qui fait beaucoup de demandes de mémoire. Seul un calcul statistique en cours de fonctionnement peut estimer la demande future possible.

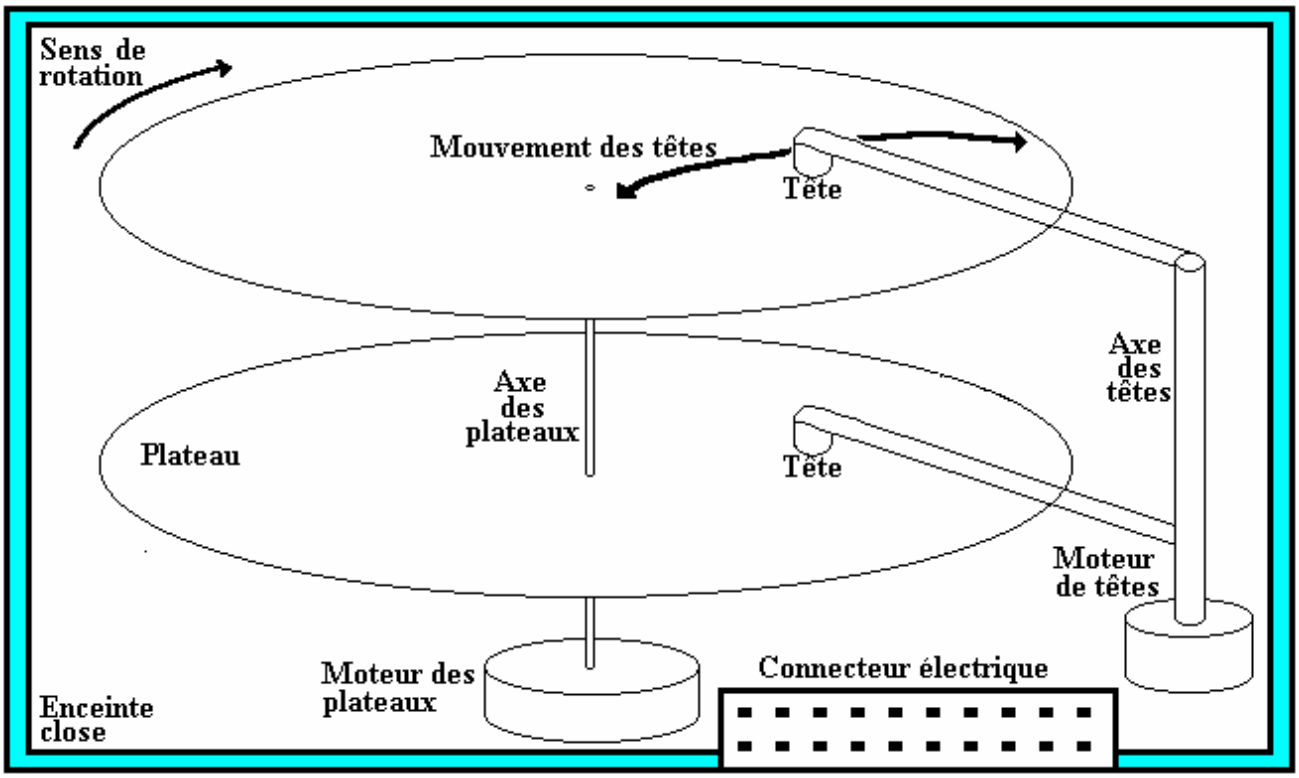
Ces techniques de prévisions sont très complexes et demandent une partie des ressources de calcul. Elles prennent du temps CPU. Il faut donc faire des choix , et ce sont ces choix qui sont une bonne partie des caractéristiques d'un système.

# Le disque

## Constitution physique

Il est constitué de 2 parties : le disque et la carte contrôleur.

Le disque est un dispositif physique qui sert à ranger des informations.




Il est constitué de :

- 1 ou plusieurs plateaux circulaires en aluminium (quelquefois en verre) recouverts d'une couche de matériaux magnétique. Ils sont solidaires d'un axe et entraînés par un moteur.
- Un "peigne" de têtes magnétiques (ex. , les têtes de magnétophones) , qui est actionné par un moteur de têtes. Les têtes "volent" à la surface des plateaux.
- Le tout est scellé dans une enceinte close étanche remplie d'azote (ou d'un autre gaz neutre, style argon).
- Un connecteur électrique qui permet de passer l'alimentation du disque et les signaux de commande des moteur , des têtes , etc.

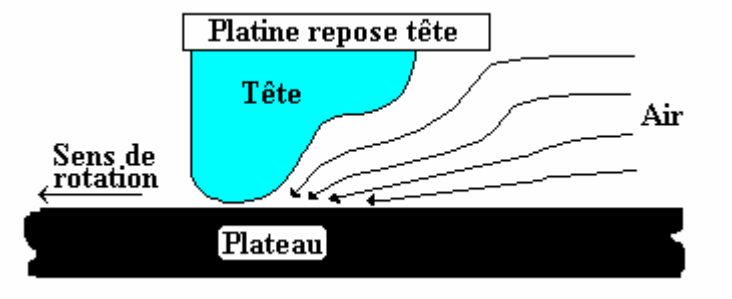
Différents formats (taille) existent. On parle de pouces pour la taille des plateaux. De vieux disques existent encore avec des tailles supérieures à 5 1/4 pouces , mais actuellement il n'y a que quelques formats , cela indépendamment des capacités.

| Ordinateur de bureau | Ordinateur portable |
|---|---|
| 5 1/4 | 3,5 |
| 3,5 | 2,5 |
| | 1,8 |

Le disque fonctionne sur le principe de l'enregistrement magnétique. On applique un champ électrique à un enroulement sur la tête , cela induit un champ magnétique sur la surface. On peut ainsi conserver l'information (orientation du champ),  même en absence de tension (si le dispositif n'est plus alimenté). Lorsqu'on veut relire , il suffit de détecter l'orientation du champ ; on reconstitue alors le signal.

Nous travaillons en binaire , on applique donc une suite de tensions positives ou négatives dans la bobine de la tête , ce qui induit une suite de champs magnétiques , comme si on plaçait une suite d'aimants les uns à la suite des autres (le plateau tourne lorsqu'on enregistre ou on lit).

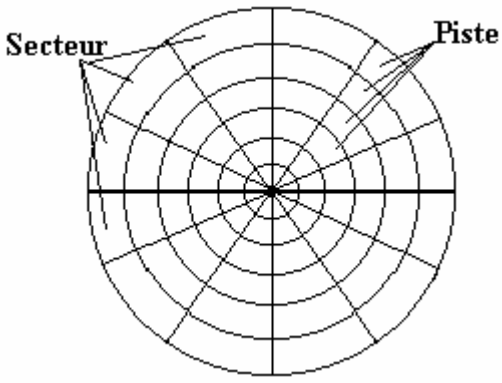
On place toute la mécanique dans une enceinte scellée , car les têtes "volent" à très faible hauteur de la surface d'un plateau (principe Winchester) afin d'avoir un couplage magnétique le plus fort possible (champ inverse au carré de la distance).




On arrive ainsi à avoir des espaces tête - plateau extrêmement petits , la tête suit les déformations du plateau sur "coussin d'air". Il faut faire attention cependant aux chocs , car la tête peut toucher le plateau et "raboter" la surface , donc enlever de la couche magnétique :  on perd alors des informations. Lorsque la rotation des plateaux s'arrête , la tête retombe. Il faut donc prévoir une "piste d'atterrissage" qui ne sera jamais utilisée pour stocker de l'information.

## Organisation physique

Pour avoir un accès relativement rapide , il faut découper chaque plateau en pistes concentriques et chaque piste en secteurs.




Quand il y a plusieurs plateaux , toutes les pistes l'une au-dessus de l'autre forment ce qu'on appelle un cylindre. Un disque avec 900 cylindres et 7 plateaux a 900 x 7 x 2 pistes[7] les unes au dessus des autres.

On accède à une piste donnée par un mouvement des bras porte-tête. Quand on est sur la piste choisie , on attend que le secteur passe sous la tête.

[7] On a des pistes au-dessus et au-dessous du plateau , avec bien sûr une tête au-dessus et une tête au-dessous.

Il doit être évident pour tout le monde que les temps de déplacement des bras et le temps d'attente d'un secteur ne sont pas nuls. C'est de la mécanique !!!

Le temps de déplacement du bras est d'environ de 20 ms pour aller d'un bout à l'autre du plateau. La vitesse de rotation est aux alentours de 4000 tours / mn. Ces valeurs sont celles des disques moyens. Elles peuvent être plus rapides (10 ms et 6000 t/mn) pour les disques les plus performants , mais elles sont loin d'être comparables avec les temps d'accès des composants électroniques (100 ns). Cette disproportion entre les temps oblige à concevoir des stratégies d'accès aux informations stockées sur le disque afin de ne pas pénaliser le CPU.

Le disque , malgré sa lenteur , possède une caractéristique fondamentale extrêmement intéressante, c'est sa contenance. La quantité d'informations est comprise entre 100 Mo pour les plus faibles (1,8 pouce) , à 5 Go pour les plus gros (5 1/4 pouces).

De plus , le ratio prix/capacité est le plus bas qui se puisse trouver , un disque de 1 Go vaut en septembre 93 environ 6000 F à 8000 F (à comparer aux 300 F la barrette de 1 Mo de RAM).

## Buffer

Tous les secteurs d'un disque ont une même taille.

Les informations sont référencées par la piste , le secteur et la place dans le secteur.

On ne **doit pas accéder directement** à un octet sur un disque. Il faut obligatoirement passer par un buffer en mémoire. La taille du buffer est généralement la même que celle des secteurs.

Voici toutes les actions qui doivent être effectuées quand on veut accéder à des informations sur le disque , sans préciser si c'est le système ou le programmeur qui les exécutent.

Pour lire une information , il faut :
- Déplacer le bras sur la piste voulue.
- Attendre que le secteur passe sous la tête pour le lire et le stocker en mémoire.
- Chercher l'information dans le buffer.

Pour écrire une information , il faut :
- Remplir un buffer avec les informations que l'on veut écrire.
- Déplacer le bras sur la piste voulue.
- Attendre que le secteur passe sous la tête pour écrire le buffer dans le secteur.

Pour modifier une information, il faut :
- Déplacer le bras sur la piste voulue.
- Attendre que le secteur passe sous la tête pour le lire et le stocker dans le buffer.
- Modifier l'information.
- Attendre que le secteur passe sous la tête pour écrire le buffer dans le secteur.

On voit que la lecture , l'écriture ou la modification demandent au moins l'attente du passage du secteur sous la tête. Vouloir modifier directement sur le disque est une perte de temps , on peut ralentir[8] énormément le déroulement d'un programme si on fait des accès octet par octet.

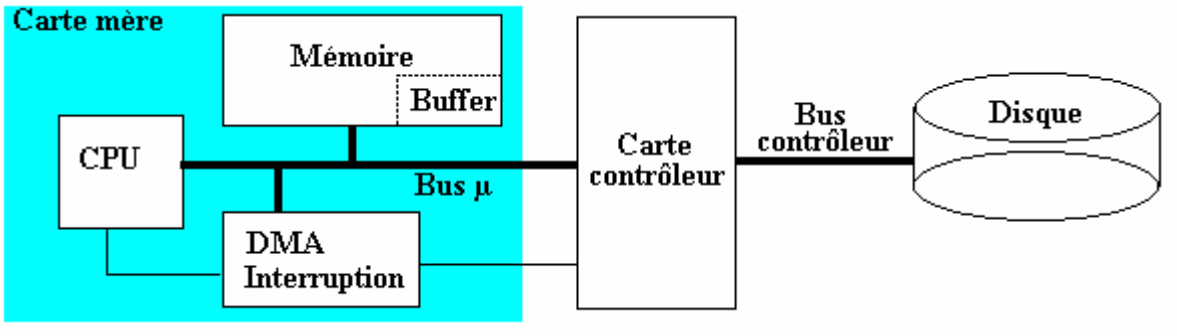
On mesure alors l'importance d'une bonne gestion des échanges disque/buffer.

---

[8] Les bases de données , grandes consommatrices d'accès disque , gèrent souvent elles-mêmes l'organisation du disque afin d'accélérer les échanges. Les gros fichiers sont stockés sur des secteurs situés les uns à la suite des autres et , s'ils dépassent la place d'un cylindre , on les organise en prenant les pistes les unes après les autres pour minimiser les temps de déplacement de bras. Ce sont des organisations du type ISAM , VSAM , etc. Pour certaines applications critiques , qui manipulent de très gros fichies (base de données géographique $\Rightarrow$ taille d'un fichier 3 Go) , on parallèlise les disques , par ex. on en met 20 en parallèles , Il faut alors que le système alloue des buffers en conséquence.

# Carte contrôleur

L'échange de données entre la mémoire et le disque se fait par l'intermédiaire d'une carte contrôleur. Son rôle est d'être une interface qui décharge le CPU des fonctions les plus simples et de pouvoir s'adapter à tous les disques qui possèdent une même caractéristique. Pour cela il faut instaurer des normes.




## Les différents types d'interfaces

### L'interface ST 506

L'interface ST 506 a été introduite par IBM en même temps que les premiers PC équipés de disques durs, les XT. Ce fut la première interface standardisée. Elle a été faite pour pouvoir s'interfacer avec les disques du constructeur de disque Seagate Technology. C'est une interface dérivée de l'interface pour disquette, elle constitue le standard pour les disques durs de faibles capacités. Le codage est de type :

- MFM[9] (Modified Frequency Modulation) qui établit sur le disque une correspondance de type "un-à-un" entre les bits de données et les variations magnétiques.
- RLL (Run Lengh Limited) ou RLL avancé (ARLL), ont été introduits pour augmenter de 50 à 100% la densité de données sur le disque.

Les principales limites de cette interface sont l'utilisation de :

- Nombre de pistes limité à 1024
- Transfert à 5 MHz
- Nombre de têtes limité à 8

**Remarques :**

Cette interface utilise un système de câblage standard composé d'un câble de 20 conducteurs pour les données et de 34 conducteurs pour les signaux. C'est une interface simple. Les bits de synchronisation sont transmis avec les bits de données, et le contrôleur doit faire le tri (appelé séparation de données) , ce qui ralentit le fonctionnement. Une erreur sur un bit de synchronisation oblige à retransmettre les bits de données. C'est pourquoi les câbles sont généralement courts, afin de minimiser les risques d'erreurs.

**ex. de transfert de données :**

- Le CPU envoie une demande au contrôleur pour lire/écrire le cylindre C, la tête T, le secteur S.
- Le contrôleur donne l'ordre au bras de se déplacer sur le cylindre C. Le CPU et le contrôleur sont en attente.
- Le bras étant sur la piste , le contrôleur sélectionne le tête T pour la lecture /écriture.
- Quand le secteur S passe sous la tête , le contrôleur lit/écrit les données du buffer en faisant une "séparation des données[10]" et un calcul de CRC. S'il y a une erreur , il faut recommencer à attendre le passage du secteur.

---

9 MFM et RLL sont des techniques de codage des informations sur le support magnétique. RLL permet une plus grande densité d'information.

10 Technique consistant à extraire/encoder des informations dans un signal (modulation de fréquence).

## L'interface IDE[11]

On trouve sur ces disques 26 à 35 secteurs par piste. L'interface utilisée est une interface de type ST 506 , mais afin d'éliminer les problèmes de câbles entre le disque et le contrôleur, ce dernier est intégré aux disques. Cela simplifie le câblage à un seul câble plat (quand il y en a besoin , entre la carte mère et le disque.

Avec le système IDE le disque peut être connecté au bus de 3 façons :

- On peut avoir une "Hardcard", c'est à dire un disque fixé sur une carte d'extension. La carte est placée dans un des connecteurs d'extension du PC.
- On peut avoir une carte d'extension ne comportant pratiquement pas d'électronique. Il s'agit en fait de simples connecteurs.
- Depuis quelque temps de nombreuses cartes mères des micro-ordinateurs possèdent d'origine un connecteur pour les disques IDE.

**Inconvénients :**

Il est impossible d'effectuer des opérations de maintenance par logiciel. Le disques ne sont pas supposés subir des formatages de bas niveau. Dans certains cas cette opération peut même les endommager. En général ces disques sont plus fiables mais plus difficiles à réparer en cas de problèmes.

## L'interface ESDI[12]

C'est également une interface ST 506 modifiée en 1983 de manière à supprimer les limites de celle-ci (densité des pistes, taux de transfert,.....). Le taux de transfert avec l'ESDI se situe entre 10 et 15 Mb/s.

**Avantages :**

- L'ESDI gère la fonction de synchronisation, déchargeant le contrôleur de ce travail. Cela limite les problèmes de synchronisation dus aux bruits se produisant entre le disque et le contrôleur.
- Possibilité pour le disque d'informer le contrôleur de ses caractéristiques physiques. Il n'est plus nécessaire alors de connaître le nombre de têtes, de cylindres, de secteurs.

**ex. de transfert de données :**

Le transfert de données est identique à celui de l'interface ST 506 , sauf pour l'étape 4 où la séparation des données est effectuée au niveau du disque et non pas du contrôleur , ce qui permet d'aller plus vite.

---

11 IDE = Intégrated Drive Electronics

12 ESDI = Echanced Small Device Interface

## L'interface SCSI[13]

Cette interface existait déjà en 1979 sous le nom de "SASI". L'avantage de la connexion SCSI est liée à son architecture. Avec son bus de données et ses neufs lignes de contrôle supplémentaires, la connexion SCSI permet la connexion de 8 dispositifs au standard , le disque dur pouvant être un de ces dispositifs. Cette interface utilise toujours un encodage de type RLL.

Il existe deux méthodes de gestion des dispositifs SCSI :

**Méthode asynchrone**

Un maximum de 1,5 Mo par secondes peuvent être transmis.

**Méthode synchrone**

Jusqu'à 4 Mo peuvent être transmis. Les disques SCSI peuvent atteindre un taux de transmission effectif des données de 750 Ko/secondes.

ex. de transfert de données :

- Le CPU envoi un ordre de lecture/écriture du cylindre C , tête T , secteur S à l'adaptateur SCSI.
- L'adaptateur convertit cette adresse en linéaire et transmet l'adresse au disque.
- Le contrôleur local du disque prend ensuite la suite du travail. Il fait à peu près la même chose que le ST 506.

*Nota 1*

Il existe des évolutions de l'interface SCSI.

| | | |
|---|---|---|
| SCSI 2 | Grappe de 15 dispositifs | Vitesse de transfert de 15 Mo |
| SCSI 3 | " | " 20 Mo |

*Nota 2*

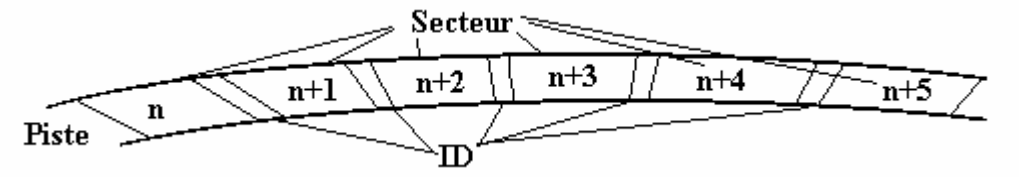
On peut mettre autre chose que des disques sur un bus SCSI. Actuellement de nombreux scanners ont une interface SCSI. Il semblerait que cette interface devienne la norme de connexion des périphériques.

[13] Small Computer System Interface. Système d'interfaçage de plus en plus utilisé. C'est l'interface standard des stations de travail.

# L'entrelacement

Chaque opération de lecture/écriture concerne généralement plusieurs secteurs. Si une opération a besoin du secteur S de la piste P de la face F, elle aura probablement besoin du secteur S+1 , S+2 , etc. Il faut donc trouver la disposition des secteurs permettant une vitesse de lecture maximale.

Lors du formatage de bas niveau d'un disque, des Identificateurs de Secteurs ou ID sont inscrits sur le disque. Ces ID servent à indiquer la limite des secteurs et à les identifier par un numéro.



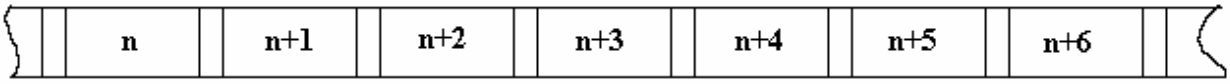

Quand le contrôleur lit/écrit un secteur , le décode/encode , et qu'il établit une transmission avec le buffer , il se passe du temps. La tête peut avoir vu passer le prochain secteur physique. Dans ce cas, on doit attendre une rotation complète avant que le secteur suivant puisse être lu.

Seuls des contrôleurs sophistiqués peuvent supporter une disposition purement séquentielle des secteurs. Lorsque les secteurs d'une piste sont disposés les uns à la suite des autres , le facteur d'entrelacement est de 1:1. Ce ratio indique que la totalité d'une piste peut être lue lors d'une rotation.

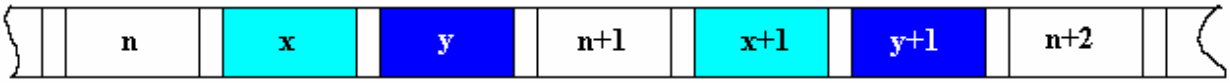
Pour faire semblant de ralentir le débit des secteurs , on peut les disposer de façon décalée. On peut mettre 1, 2, 3 ou plus secteurs entre les 2 secteurs à lire. Ce rapport définit le facteur d'entrelacement.

Les trois facteurs d'entrelacement les plus courants sont :

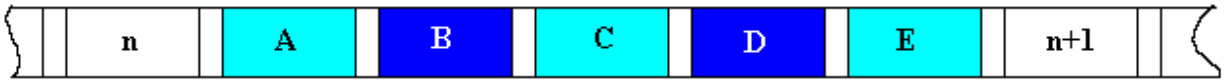
- 1:1 : Les différents secteurs se suivent immédiatement.




- 3:1 : Deux secteurs sont intercalés entre deux secteurs logiques.




- 6:1: Cinq secteurs sont intercalés entre deux secteurs logiques.




Du facteur d'entrelacement dépend donc le temps nécessaire pour lire les secteurs logiques qui se suivent , et donc aussi la vitesse d'accès aux informations sur le disque. Il n'influe cependant pas sur la vitesse générale du disque. Il faut que le facteur d'entrelacement soit adapté au contrôleur ainsi qu'à la vitesse de travail de l'ordinateur. Un facteur d'entrelacement de 1 ne sert pas à grand chose sur un ordinateur lent.

Nota

Avec les interfaces et les disques ESDI , IDE ou SCSI , on ne se préoccupe pas du facteur d'entrelacement , presque tous utilisent un facteur d'entrelacement 1/1.

# Déplacement du bras

Le contrôleur reçoit des demandes du CPU. Celles-ci , à cause des différences de vitesse entre CPU et disque , on tendance à s'accumuler. On garde ces demandes dans une structure de type liste chaînées. On va donc demander à lire/écrire plusieurs secteurs généralement non contigus. Pour satisfaire ces demandes le bras va se déplacer.

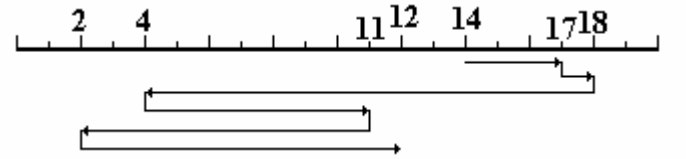
## Algorithmes d'ordonnancement

On a des demandes dans une liste chaînée (file d'attente). On cherche le meilleur ordonnancement.

### Ordre d'arrivée

On effectue les déplacements du bras pour satisfaire les demandes dans l'ordre d'arrivée.

ex. Le CPU a demandé les secteurs situés sur les pistes { 17 , 18 , 4 , 11 , 2 , 12 }. Le bras est en 14.

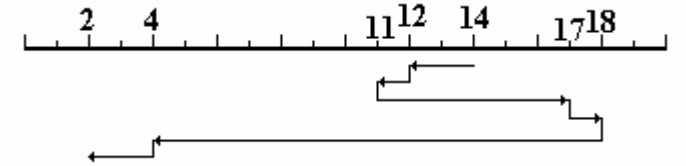



On a effectué : 3 + 1 + 14 + 7 + 9 + 10 = 44 déplacements.

### Le plus court chemin

Les déplacements du bras ressemblent à un chemin suivi dans un graphe. Il existe des algorithmes de recherche du plus court chemin dans un graphe (Dijkstra , Floyd-Warshall , etc.). On ne répond plus aux demandes dans l'ordre d'arrivée , il faut réordonner la liste.

On reprend le même ex. et on cherche à minimiser les déplacements.



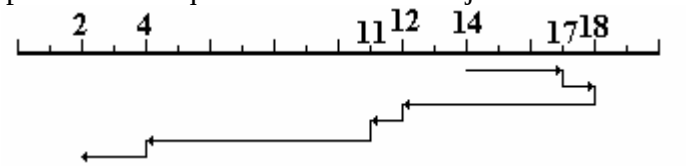

On a effectué : 2 + 1 + 6 + 1 + 14 + 2 = 26 déplacements

Bien que cet algorithme paraisse meilleur , il présente des problèmes difficiles à résoudre.

- Il n'est pas forcément optimal (commencer par les pistes 17 et 18 et redescendre aurait minimisé les déplacements).
- Si de nouvelles demandes arrivent lors du déplacement , l'algorithme doit se recalculer. A chaque ajout de demande dans la liste , il peut y avoir un phénomène de famine , ou alors un phénomène d'oscillations autour d'une position peut se produire.

### Balayage

L'idée est de dire que lorsque le bras est parti dans une direction donnée , il n'en change pas avant d'être arrivé à la fin , et on prend tout ce qui se trouve sur le trajet.




On a effectué : 3 + 1 + 6 + 1 + 7 + 2 = 20 déplacements

Cet algorithme à 2 avantages : il est facile à implémenter , et il minimise les déplacements.

# Organisation des données

Les techniques de déplacement de bras , de lecture/écriture dans des secteurs et les standards de liaison contrôleur disque étant en place , nous allons décrire l'organisation des données sur le disque.

## Vision physique

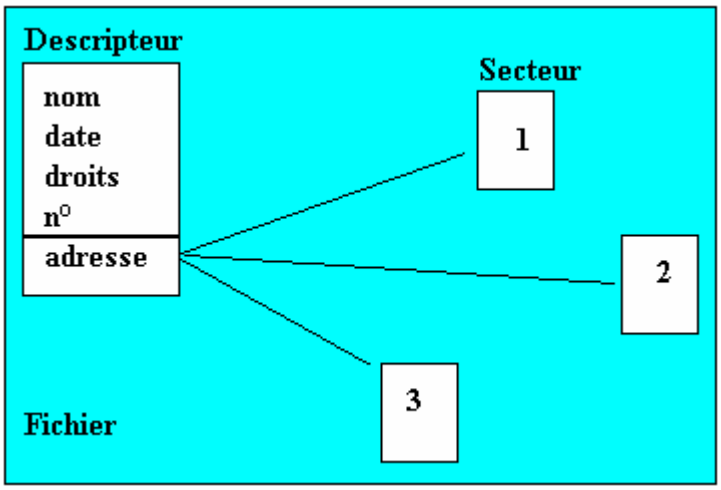
On a vu qu'un disque est constitué de pistes contenant chacune des secteurs. Les données vont être stockées sur ces secteurs. La taille d'un secteur est généralement de 512 octets. Il est rare d'avoir des informations (programmes , datas , etc.) qui tiennent dans 512 octets. La taille des programmes peut être de l'ordre du méga-octet , quand aux datas il suffit de s'imaginer une base de données pour comprendre qu'il n'y a pas de limite. Il faut donc regrouper tout cela dans une structure sur disque que l'on appelle un fichier. Un fichier est composé d'une série de secteurs liés entre eux , chacun portant une partie de l'information. La liaison des secteurs composant le fichier se fait dans un endroit du disque. C'est une structure de données qui contient le nom du fichier , diverses informations comme la date , la taille , etc. , et "l'adresse" des secteurs formant le fichier. On appelle cette structure un descripteur de fichier.

### Les clusters

Dans certains systèmes les secteurs sont regroupés en **unité d'allocation** ou cluster. Un cluster est l'espace minimum pouvant être alloué par le système à un fichier. En créant un fichier d'un seul octet il n'occupera pas un seul octet sur le disque mais une unité d'allocation, c'est à dire un cluster. La taille du cluster varie avec le type de disque ; par ex. pour un disque de taille supérieure à 128 Mo les clusters font 4 ko. Dans la suite nous ne différencierons plus cluster et secteurs : l'un étant un groupe de l'autre , nous emploierons le terme secteur.

### Fichier

Ce que l'on appelle fichier est formé : de la suite des secteurs qui portent les données et du descripteur de fichier.



## Descripteur de fichier

Un descripteur de fichier est composé en général :

- D'une partie accessible par le programmeur
  - * Un nom de fichier avec une extension possible.
  - * Un numéro qui est unique[14].
  - * La date de création ou de dernière modification (jour et heure).
  - * Le propriétaire de ce fichier ( appartenance à un groupe , etc.). De lui découle :
    - # La nature du fichier ( données , programme , etc.)
    - # Les droits d'accès à ce fichier ( lecture/écriture , lecture seule , etc.)
- D'une partie à laquelle le programmeur ne doit pas accéder , et qui est la liste de tous les secteurs composant ce fichier.

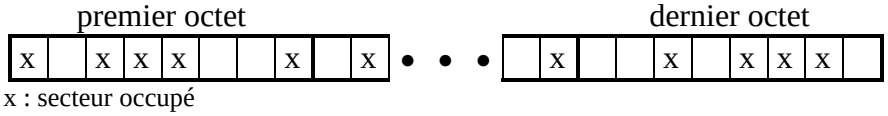
Lorsqu'un programmeur demande l'écriture d'un fichier , le système cherche les secteurs libres pouvant être utilisés et met à jour le descripteur de fichier.

## Bit Map

Pour savoir où sont les secteurs libres sur le disque , il faut au système une indication , généralement sous forme d'une table de bits. Cette table est située dans un endroit précis du disque et le système est toujours capable d'y accéder , mais lui seulement , surtout pas le programmeur. Cette table n'est pas très grande , il n'est pas nécessaire de chercher à la coder.

### Calcul

Prenons un disque de 1 Go = $2^{30}$ octets et des secteurs de 512 octets = $2^{9}$ octets. Il y a donc $2^{21}$ (30-9) secteurs. Si 1 secteur = 1 bit , cela fait $2^{19}$ (21-3) octets = 500 ko. C'est relativement peu important , cela n'occupe en fait que $2^{10}$ (19-9) secteurs , c.a.d. 1024 secteurs. Cela tient généralement sur la première piste du disque.




x : secteur occupé

## Liaison descripteur/secteurs

Les secteurs recevant les données sont liés au descripteur. Les secteurs entre eux possèdent un ordre, il y a le premier secteur , le deuxième , etc. Il faut trouver un système de liaison entre le descripteur et les secteurs , en tenant compte de l'ordre qui existe au sein de l'ensemble des secteurs composant le fichier.

### Liaison Bit Map

On décrit dans le descripteur le bit map des secteurs occupés pour ce fichier.

| | |
|---|---|
| avantage : | Simplicité extrême (rustique sinon fruste) , et très robuste. |
| inconvénient : | Place prise par le bit map à chaque nouveau fichier. En reprenant l'ex. de calcul précédent , pour chaque fichier il faut allouer 1025[15] secteurs , c'est beaucoup !!! |

[14] Il peut être unique pour tout le disque , ou pour la directorie où se trouve le fichier. Ce numéro s'appelle un "***handle***".

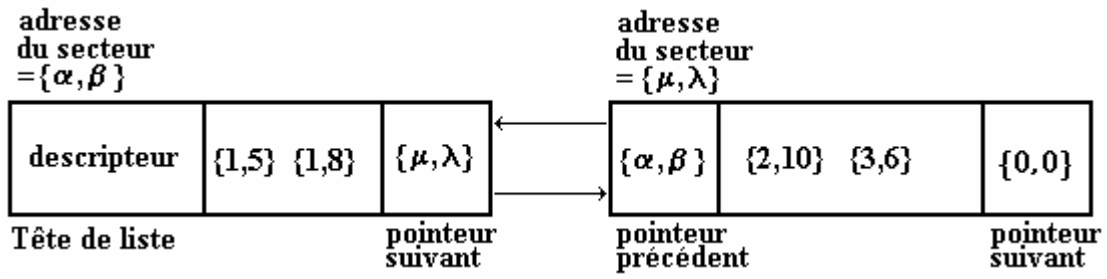
[15] 1024 secteurs pour le bit map et 1 pour le descripteur.

## Liaison chaînée

On peut décrire la suite des secteurs par une liste de leurs numéros d'emplacement sur le disque , le descripteur conservant la tête de liste. Pour une plus grande efficacité , il est évident que cette liste est doublement chaînée , circulation avant et arrière possible.

Cette solution est très simple , mais elle présente quelques défauts , le premier étant le temps qu'il faut pour aller d'un bout à l'autre du fichier (déplacement dans une liste). Mais le gros problème de cette solution , c'est la perte d'un des liens qui rend la reconstitution du fichier très difficile.

Par ex., on a un fichier constitué de 4 secteurs notés comme un couple - piste , numéro de secteur dans la piste - {1,5} ; {1,8} ; {2, 10} ; {3,6}. On peut parfaitement placer ces indications dans le secteur contenant le descripteur , à la suite des informations du descripteur. Le seul petit problème , c'est quand la quantité de secteurs du fichier dépasse la taille d'un secteur , il faut alors créer en fin de secteur une place spéciale qui indique s'il y a une suite et où , ou alors si c'est la fin.




Si on considère qu'une référence a un cluster tienst, question place , sur 2 x 2 octets (65535 pistes et 65535 secteurs par piste au maximum) , on voit que sur 1 secteur de 512 octets on peut mettre : $\frac{512}{4}$ - 8 = 120 références à des secteurs , ce qui fait 120 x 512 = 61440 octets de données dans un fichier possible adressé par 1 secteur.

Sous une forme un peu différente , c'est l'organisation descripteur/secteurs qu'a adopté le DOS.

### Calcul

Si on a un fichier de 1 Go il faut :

(on arrondt le nombre de secteurs adressables par un secteur à $2^{15}$ pour simplifier les calculs)

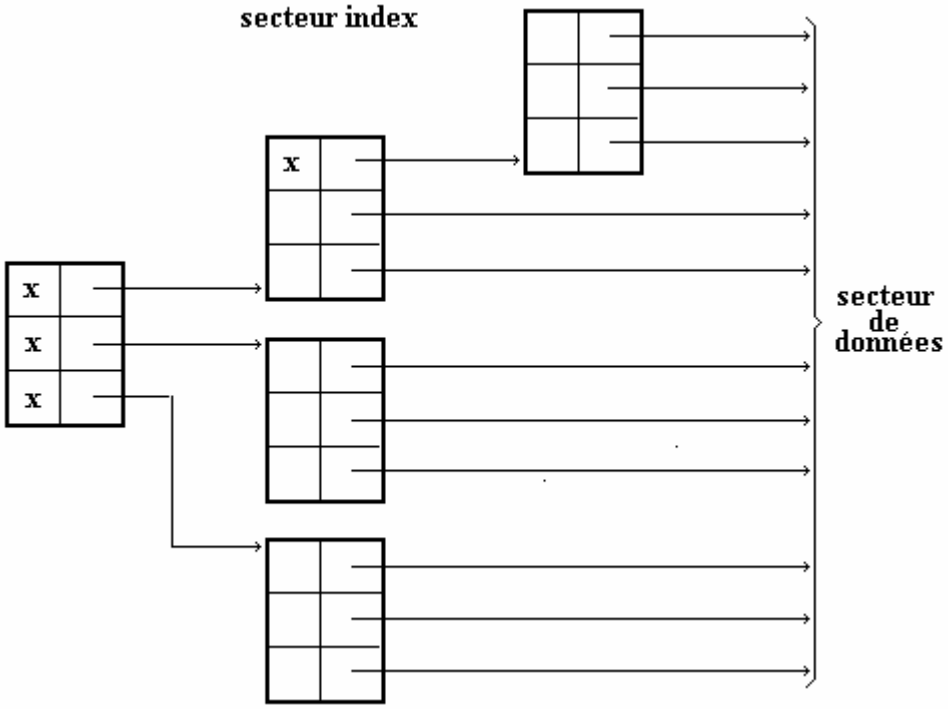
$2^{30} / 2^{9} = 2^{21}$ = 2 097 152 secteurs pour les données

$2^{21} / 2^{15} = 2^{6}$ = 64 secteurs pour adresser les secteurs de données

Il n'y a qu'un seul niveau d'indirection pour les petits fichiers , ceux qui ne contiennent pas beaucoup de données , c'est très intéressant. Mais pour les gros fichiers c'est pénalisant de devoir se déplacer linéairement sur tous les maillons de la liste chaînée. Il peut être plus utile d'augmenter le nombre d'indirections mais de diminuer le nombre de déplacements.

## Liaison arborescente

On rajoute à chaque couple - piste , secteur dans la piste - une marque qui indique si on pointe sur un secteur de données ou un secteur d'index. les secteurs de données étant en général pointés par les derniers secteurs d'index.



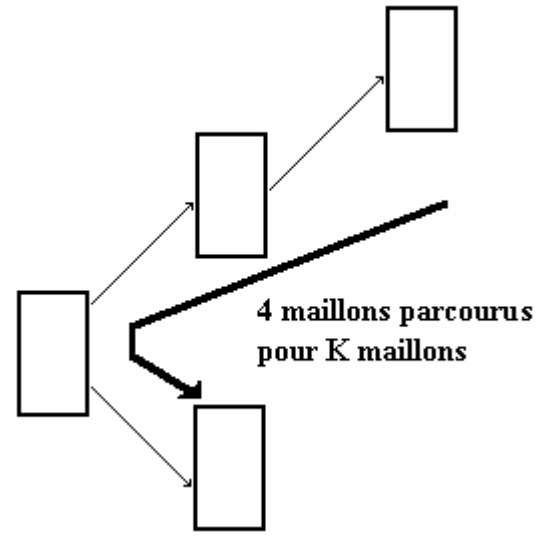

Cette structure de secteur est plus compliquée , mais les accès à un descripteur à partir d'un autre sont plus courts en trajet. En fait , le nombre de maillons à parcourir est borné par la hauteur de l'arborescence.

### Calcul

On prend des secteurs contenant N indirections , et on suppose que l'arbre est complet , c.a.d. que tous les noeuds de l'arbre contiennent une indirection sur d'autre noeuds , sauf pour les indirections sur les feuilles (les feuilles sont des secteurs de données). On a K feuilles à adresser , il nous faut donc x noeuds pour adresser les feuilles.

$\lfloor x \rfloor^N = K$ ce qui revient à dire que la hauteur max de l'arbre est $h = \lceil \log_N(K) \rceil$

On a donc diminué le nombre maximum de maillons à parcourir pour passer d'un secteur à l'autre , au détriment des indirections qui sont plus complexes.



## Morcellement

Quand il faut prendre des secteurs libres , pour y mettre quelque chose dedans , le choix du premier secteur libre trouvé semble le plus simple. Mais cela crée des phénomènes de morcellement des fichiers. Lors des effacements , déplacements , réécritures d'un fichier , les secteurs ne sont pas les uns à la suite des autres. Pour un même fichier on a un secteur ici , un autre là , etc. Le bras , lors de la lecture/écriture , va se déplacer beaucoup. Il est donc souhaitable de réordonner les secteurs alloués au fichier.

Malheureusement c'est une opération extrêmement coûteuse en temps.

Seuls les systèmes de gestion de base de données[16] doivent tenir compte de ces problèmes. Le gain , en temps d'accès , obtenu par une disposition structurée en allocation de piste entière pour un fichier , et d'autres techniques , compensent largement le temps passé à maintenir l'ensemble opérationnel.

Pour les autres systèmes , le réordonnancement se fait sur la volonté d'un programmeur lorsque les accès deviennent trop longs par rapport aux caractéristiques moyennes du disque.

Il existe des programmes de "compression" pour le DOS , qui est très sensible à ce problème. UNIX l'est tout autant mais l'organisation est un peu plus compliquée , ce qui ne facilite pas la réorganisation. Dans tous les cas , lancer un programme de restructuration du disque demande d'avoir du temps devant soi[17] , c'est pourquoi on n'effectue cela qu'une fois par mois.

## Actions sur fichier

Les actions possibles  pour le programmeur , sur les descripteurs de fichiers , sont la lecture afin d'avoir des informations , et la modification du nom ou des droits d'accès. La date et le handle sont mis par le système. Bien sûr les données placées dans les secteurs sont , en fonction des droits d'accès et des protections (lecture seul) , accessibls en lecture/écriture.

On va donner une liste des actions élémentaires exécutables sur des fichiers et nous détaillerons ensuite leurs déroulements.

* Lire le descripteur : afin d'avoir des informations sur le fichier , taille , date , etc.
* Ouvrir un fichier : c'est placer le descripteur dans un endroit de la mémoire afin de le faire connaître au système.
* Fermer un fichier : c'est libérer la mémoire du descripteur de fichier.
* Se déplacer : Pour pouvoir aller en n'importe quel endroit du fichier où sont situées les informations.
* Lire un fichier : c'est placer les informations qui sont enregistrées sur des secteurs dans un espace mémoire.
* Ecrire un fichier : c'est enregistrer les informations placées en mémoire dans des secteurs.

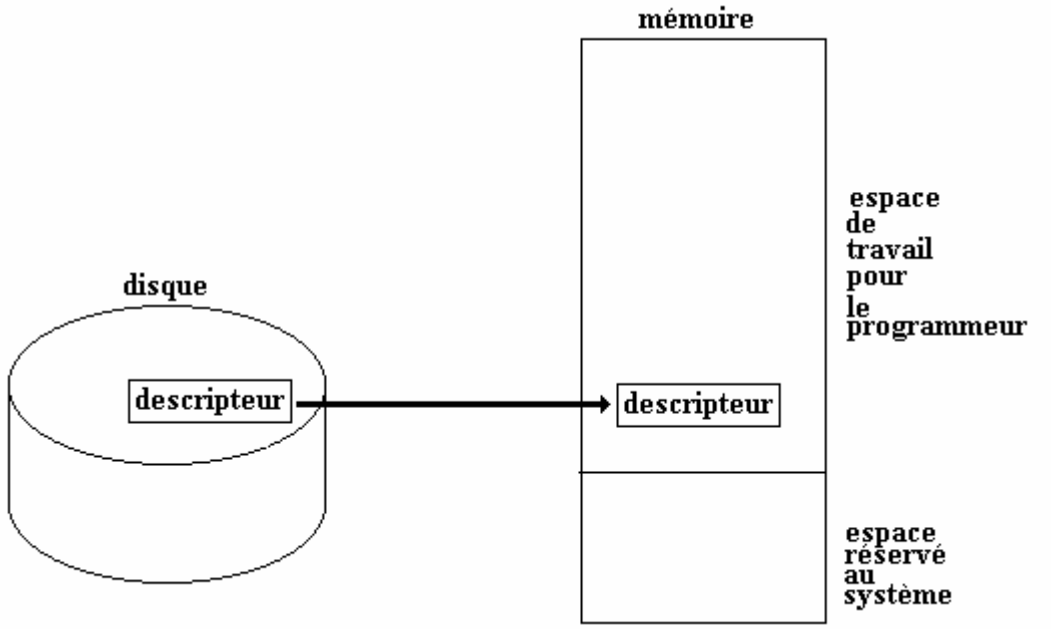
16 Déjà vue à la page 21

17 Expérience personnelle : une partition DOS de 300 Mo à réorganiser , prends environ 2 heures (3 dans le cas de beaucoup de changements, effacement d'un logiciel , rajout d'un autre).

## Lire le descripteur

Avant toute manipulation sur le fichier , il est souvent nécessaire d'avoir des informations sur les caractéristiques de ce fichier , taille , date, droit d'accès , etc. Il faut alors demander au système de nous fournir , sous forme accessible , le descripteur de ce fichier. On donne un ordre au système en passant le nom du fichier en paramètre , et le système nous répond en créant dans une zone mémoire une image du descripteur de fichier. C'est une structure de données comprenant les champs d'informations du descripteur. Ce n'est pas le fichier , c'est simplement une image du descripteur. La politique consistant à demander au système le descripteur de fichier avant tout travail sur ledit fichier , est une des méthodes les plus sûres. On a ainsi le fichier accessible et le descripteur , c'est l'idéal pour le programmeur.

Une fois que l'on a ouvert le fichier , le descripteur est dans un coin de la mémoire. Il est naturel de demander une copie dans une structure de données accessible. L'ordre pour avoir cette structure de donnéesse fait en ***C*** par ***fstat***. Le DOS permet aussi d'avoir accès à une forme de structure de descripteur par l'ordre ***findfirst*** , cependant c'est une façon très particulière d'y accéder.




## Ouvrir un fichier

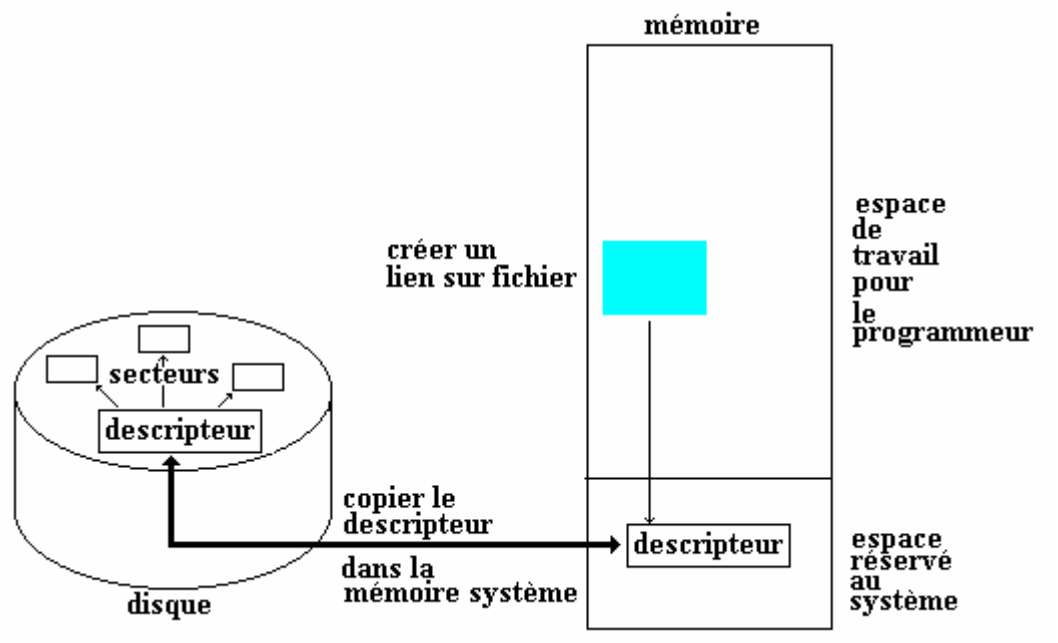
Quand on veut manipuler un fichier pour lire/écrire , il faut en premier lieu que le système connaisse où sont situés les secteurs composant ce fichier. Il est donc indispensable de le lui dire. Cette opération s'appelle l'ouverture du fichier.

Ouvrir un fichier , cela revient à donner un ordre avec des paramètres , le nom de fichier et les actions à faire sur ce fichier , au système. Trois possibilités se présentent alors :

- L'action demandée est une modification du fichier et il n'existe pas de fichier de ce nom dans l'endroit où sont stockés les descripteurs de fichier. Le système répond qu'il y a une erreur.
- L'action demandée est la création d'un nouveau fichier portant le nom donné en paramètre. Le système doit créer un nouveau descripteur de fichier et le rajouter dans l'endroit où sont stockés les descripteurs de fichier.
- L'action demandée est une lecture/écriture et le fichier existe déjà. Le système charge alors dans un endroit de la mémoire le descripteur de fichier. En fait , à cet emplacement le système charge les "adresses" des différents secteurs composant le fichier , plus des informations qui lui sont indispensables à une bonne gestion , n° du handler , taille , etc. Malheureusement cet emplacement mémoire est réservé pour le système , et le programmeur n'y a pas accès sous peine de faire d'énormes erreurs de manipulation , il ne peut demander directement après une ouverture la taille , la date , etc.

Ouvrir un fichier , c'est créer un bloc qui permettra d'avoir un accès sur ce fichier , on peut voir l'ouverture d'un fichier comme la création d'un "pointeur" sophistiqué (un lien) sur le fichier.

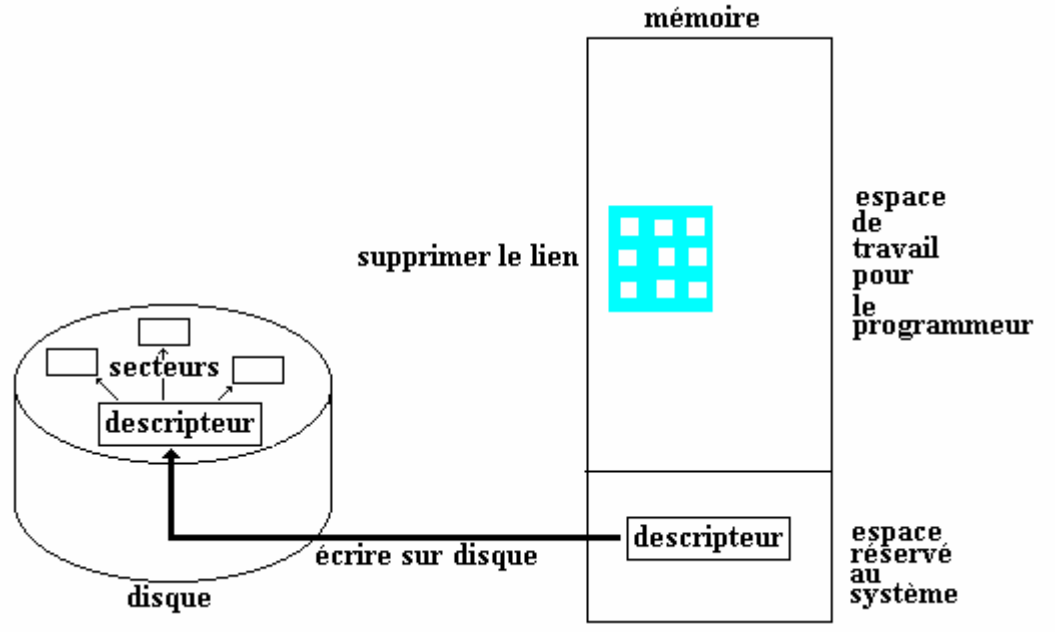
L'ouverture d'un fichier se fait en ***C*** par l'ordre ***fopen*** et ses dérivés.




## Fermer un fichier

Lorsqu'on a travaillé sur un fichier et que l'on n'a plus besoin d'y accéder pour la suite du programme, où que l'on veut empêcher la manipulation sur le fichier , on le "ferme". Alors le système enregistre , dans la zone des descripteurs de fichier , le descripteur concerné en ayant mis à jour les champs de celui- ci , date , taille , nom , secteurs liés à ce descripteur , etc. C'est lors de la fermeture que l'on est certain d'avoir un fichier parfaitement à jour. Si pour une raison indépendante du programmeur , la fermeture du fichier n'était pas faite alors que des informations avaient été rajoutées , aucun système ne garantit la pérennité de ces informations. C'est tellement vrai que lorsqu'un programme se termine et si par maladresse le programmeur a oublié de fermer un ou plusieurs fichiers , le système le fait pour lui. Ce qui ne veut pas dire que l'on doit le laisser faire au système , au contraire , il faut bien prendre soin de fermer les fichiers quand on n'en a plus l'utilité , cela permet au système d'aller plus vite et cela augmente la sécurité.

L'ordre de fermeture se fait en ***C*** par ***fclose*** et ses dérivés.

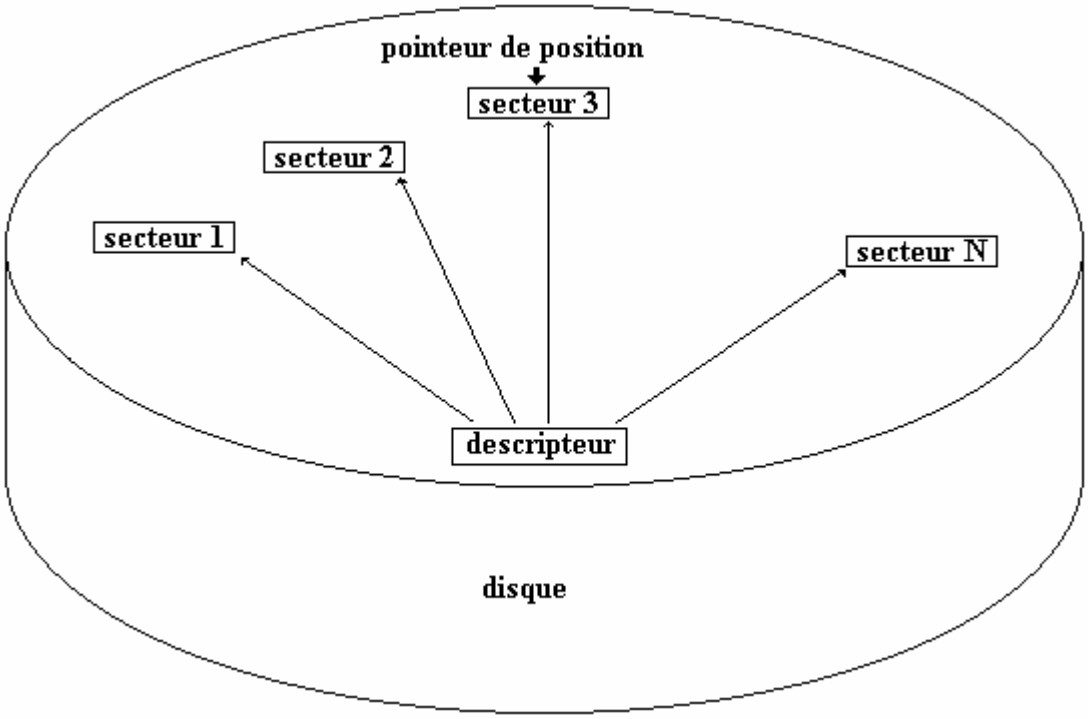


## Se déplacer

Quand on a ouvert le fichier on a accès aux données stockées dans ce fichier. Il est possible que l'on veuille lire/écrire seulement à un endroit du fichier , il faut donc avoir une technique de déplacement.

On peut se représenter les données comme une suite logique d'informations , une sorte de tableau , un indice permet alors de demander la case N de ce tableau. Cet indice permet de se déplacer dans le fichier. Bien sûr le déplacement réel n'est pas celui qui existe comme dans un tableau. Il faut au système calculer le déplacement à effectuer afin de le convertir en un multiple de taille de secteur , passer ainsi du secteur courant au secteur demandé , et ensuite se déplacer à l'intérieur du secteur. Les fichiers ont un type d'organisation du style base + offset , la base étant le secteur contenant la donnée et l'offset la place de la donnée dans le secteur. Afin de faciliter la vie au programmeur et le dispenser de calculs fastidieux , le système fait le travail de conversion entre base offset et adressage logique linéaire. Quand on demande un déplacement , qu'il soit relatif ou absolu , on demande un nombre , au système de calculer la décomposition en secteurs et offset dans le secteur. A lui aussi de savoir que l'on a demandé un déplacement trop grand , et de stopper en fin ou au début suivant le sens du déplacement demandé.

L'ordre ***C*** de déplacement se fait par ***fseek*** et ses dérivées.




## Rappel

**Déplacement relatif :**

C'est un déplacement qui se fait à partir de la position courante.

ex. On est sur la position 30 et on demande d'aller en -5 , on se retrouve en 25 , on demande ensuite d'aller en +6 et on se retrouve en 31.

**Déplacement absolu :**

C'est un déplacement qui s'effectue toujours par rapport à un point fixe.

ex. On est en 30 et on veut aller en 31 par rapport au début , on demande un déplacement de 31. On est en 10 , le fichier fait 100 , et on veut aller en 15 par rapport à la fin : on demande un déplacement de -85.

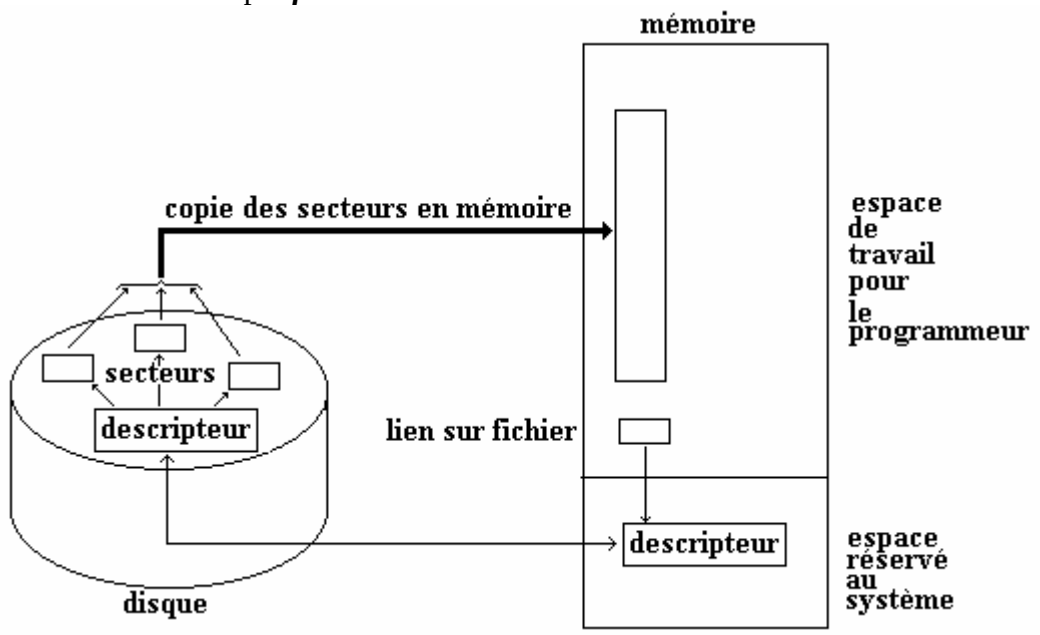
Nota :

Lors de l'ouverture d'un fichier , l'indice - ou pointeur - , de déplacement , se trouve au début du fichier.

## Lire un fichier

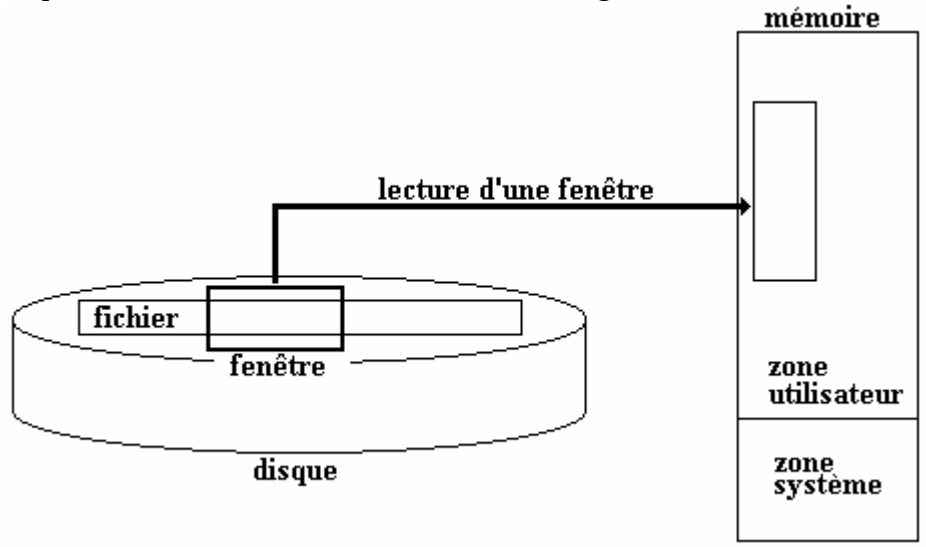
Lire un fichier , c'est avoir un espace mémoire disponible pour transférer des informations issues du ou des secteurs composant ce fichier. **C'est au programmeur de demander au système cet espace mémoire**. En fait , le système transfert les données de tout un secteur à la fois dans un buffer de travail , quitte à recopier ce buffer dans l'espace mémoire du programmeur (cas de demande de lecture d'un volume de données plus petit qu'un secteur).

L'ordre ***C*** de lecture se fait par ***fread*** et ses dérivées.




On peut lire un fichier de deux façons :

- En entier : il faut s'assurer que sa taille est contenue dans la mémoire disponible . Si c'est le cas , on demande au système de lire le fichier d'un seul coup , on a alors une copie en mémoire de tout le fichier.
- Par morceau : on a alors un comportement le lecture à fenêtre , c.a.d. que l'on a une fenêtre en mémoire du fichier. Cette fenêtre peut être petite , mais il faut impérativement que sa taille soit au moins égale ou supérieure à la taille d'un secteur , sinon les performances d'accès au fichier sont très dégradées[18].




[18] Tout programme , réaliste , qui fait des accès octet par octet , est une aberration informatique. Le programmeur qui l'a conçu n'a rien compris au système (c'est malheureusement souvent le cas). Ce serait comme utiliser une Formule 1 pour aller faire les commissions chez le boulanger au bout de la rue.

## Ecrire un fichier

Pour écrire dans un fichier , il faut avoir des data dans un espace mémoire. Il est ridicule d'écrire octet par octet (voir lecture) , il faut avoir une quantité d'informations minimale sinon on risque de pénaliser le système , une bonne valeur c'est évidemment la taille d'un secteur.

Il y a trois possibilités d'écriture :

- Le fichier est à créer et on commence à écrire au début.
- Le fichier existe déjà et on écrit à la suite des data déjà enregistrées.
- Le fichier existe déjà et on veut ajouter des data à un endroit du fichier qui n'est pas la fin.

Il existe une quatrième possibilité : lors de l'ouverture du fichier on peut demander à ouvrir un fichier déjà existant , il y a déjà des informations enregistrées , mais elles ne nous intéressent plus , donc on demande une ouverture en écriture seule avec écrasement des informations déjà existantes. Cela revient à travailler en mode de création , donc au premier cas.

Toutes ces nuances d'ajout en fin , de création , de lecture , etc. sont des indications que l'on passe en paramètre lorsqu'on demande l'ouverture du fichier.

L'ordre ***C*** d'écriture se fait par ***fwrite*** et ses dérivées.

### Ecriture en début , en fin

Que l'on écrive dans un fichier vide donc au début , ou dans un fichier existant mais à la fin, les actions à exécuter pour le système sont presque les mêmes.

#### Au début

Le système doit allouer un ou des nouveaux secteurs et écrire les informations qui sont situées en mémoire. Quand on donne un ordre d'écriture au système , on lui dit combien d'octets on veut écrire. Il est facile pour le système de faire un simple calcul lui permettant de savoir le nombre de secteurs nécessaires. Il doit ensuite remplir ses secteurs avec les informations qu'il lit séquentiellement dans la mémoire.

#### A la fin

Il faut que le système fasse la même chose que s'il enregistrait au début , mais en faisant un petit calcul en plus.

Le dernier secteur n'est pas forcément plein , il faut donc :

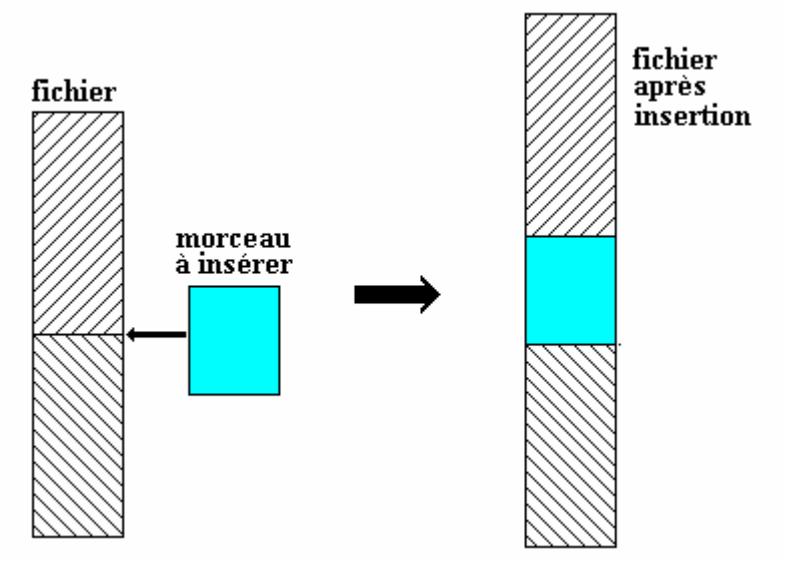
- Lire le dernier secteur pour voir combien de place reste disponible.
- Ecrire les premières data en mémoire afin de compléter ce qui était le dernier secteur.
- Reprendre le même genre de calcul que l'écriture au début , mais en ajoutant les nouveaux secteurs alloués à la suite de l'ancien dernier secteur.

Que ce soit en écriture au début ou à la fin , le système sait le faire , le programmeur n'a pas à se charger du travail. Il dit seulement où il veut écrire lors de l'ouverture , et ensuite il donne un ordre d'écriture.

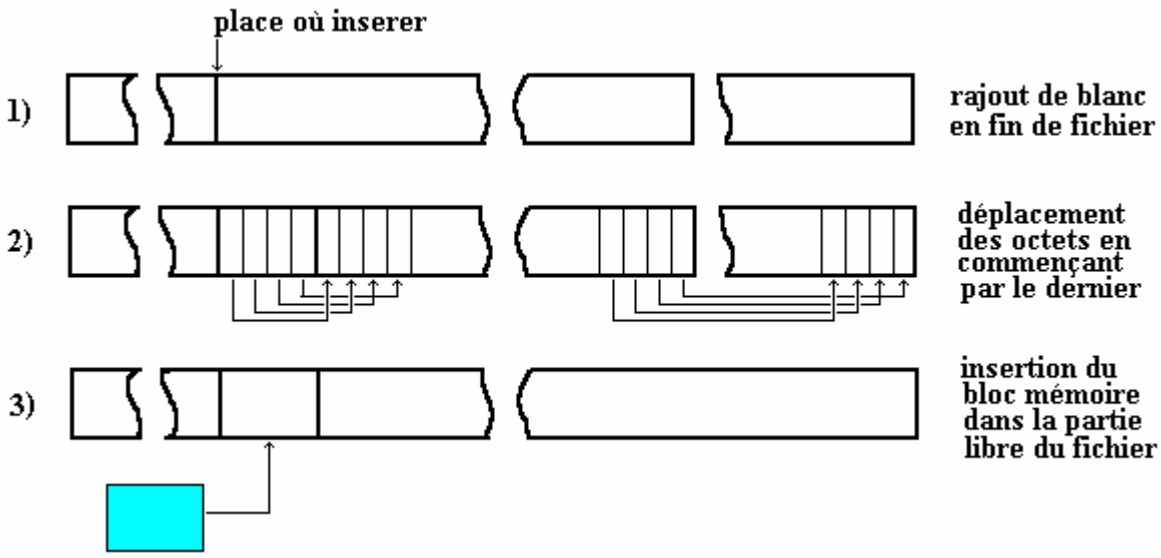
Il n'en va pas de même pour l'ajout à l'intérieur d'un fichier déjà existant.

### Ecriture à "l'intérieur"

Le problème qui se pose au système , c'est qu'à ce moment on peut "taper" à l'intérieur d'un secteur. Il faut donc déplacer tout ce qui suit , de la quantité d'informations que l'on veut inclure (on connaît le nombre d'octets à ajouter). Toutes les méthodes d'écriture en insertion doivent être faites par le programmeur. Le système n'étant pas assez doué pour cela , c'est à la charge de celui qui programme de trouver une stratégie d'insertion qui soit convenable.




Une des façons les plus triviales serait de dire : il faut ajouter à la fin un nombre d'octets "blancs" correspondant au nombre d'octets en mémoire à ajouter , ce qui revient à ajouter à la fin , et on sait faire. Puis ensuite déplacer octet par octet en partant du dernier. C'est une méthode qui a pour elle la simplicité , mais il est hors de question de faire des déplacements d'octets sur le disque (problème déjà vu). Il faut donc trouver autre chose[19].




Cette technique peut parfaitement marcher si on copie le fichier en mémoire. On exécute cette euristique en mémoire et on recopie la mémoire dans le fichier. Pour cela il faut que le fichier "tienne" en mémoire.

---

[19] De nombreux logiciels fonctionnent ainsi par déplacement 1 par 1. Il leur est possible d'agir de cette manière car ils n'ont pas beaucoup de déplacements. Mais il est impossible d'appliquer la même systématique à de gros fichiers qui nécessitent de nombreux ajouts , bases de données , images , etc.

Si le fichier ne tient pas en mémoire , il est possible de le découper en tranches (multiple entier de secteur) qui , elles , seront traitées comme si c'était le fichier. On recommence jusqu'à arriver sur la partie à insérer où , là , on fait l'écriture.

idée d'algorithme :

*variables de départ*

N : nombre d'octets à ajouter
X : place mémoire pouvant contenir x secteurs
J : indice de position dans le fichier
P : position où l'on doit insérer

*Algo*

allouer de la place mémoire pour X + N
rajouter N blanc à la fin du fichier
**tant que** pas arrivé sur l'endroit où insérer // tant que (J ≠ P )
    lire X secteurs à la position J et les placer dans la mémoire allouée
    déplacer de N les X octets mémoire en commençant par la fin
    réécrire la mémoire dans le fichier à la position J
    déplacer J vers le début de x // (J = J - x)
insérer les octets de la mémoire à la position P

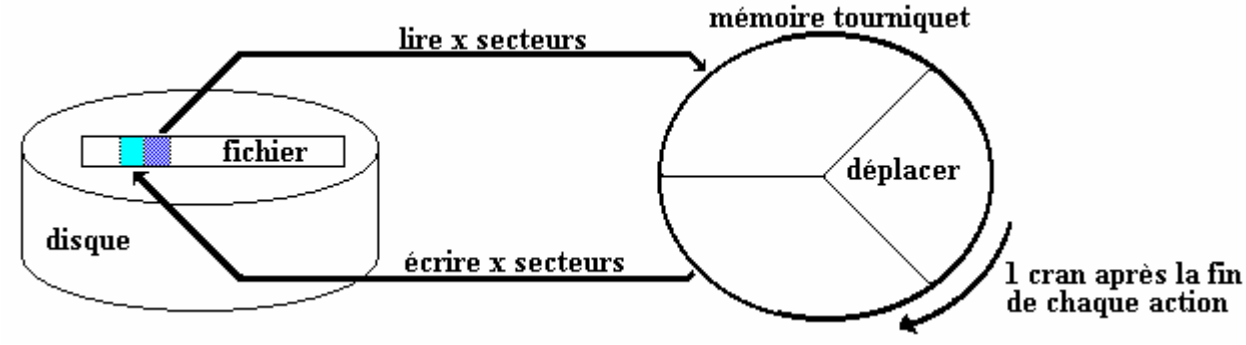
On a alors une méthode qui permet par lecture et écriture successives d'ajouter des informations dans un fichier.

On peut optimiser l'algorithme. En entremêlant les lectures et écritures des secteurs du fichier , on voit que le contrôleur de disque sera en train de faire des accès fichiers et travaillera avec le DMA , pendant ce temps le CPU fera des déplacements mémoires. On entrelace ainsi les actions , ce qui diminue le temps total de fonctionnement.

Pour arriver à faire cela il faut plusieurs espaces mémoires : un où le contrôleur transfère les secteurs du fichier , un où le CPU déplace , et un où le contrôleur transfère depuis la mémoire vers le disque.

On peut utiliser une structure de données qui s'appelle une file circulaire , ou tourniquet , afin de réaliser cette action.

Quand on est à l'étape ***i*** , un des espaces sert de récepteur du disque , un de déplacement , et un d'émetteur vers le disque. A l'étape ***i+1*** , on fait tourner l'ensemble de 1 , celui qui était émetteur devient récepteur , celui qui servait de déplacement devient émetteur , et celui qui était récepteur devient déplacement.




Nota

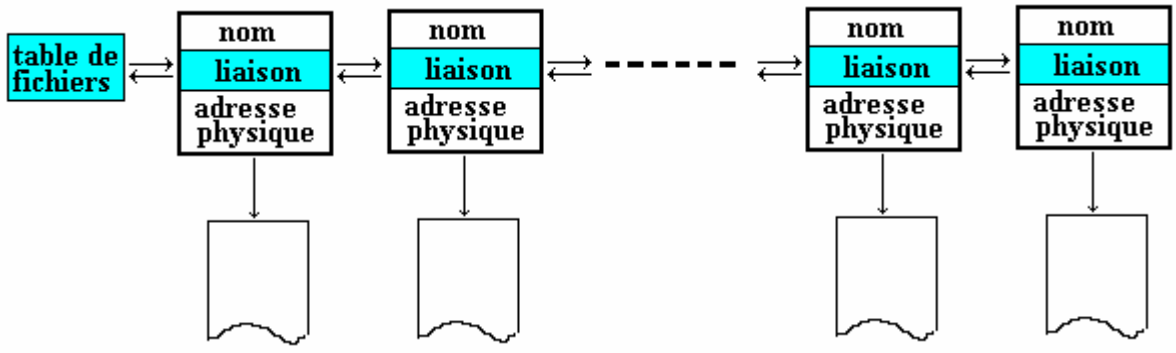
Cet algorithme est tout à fait parallélisable.

# Organisation des fichiers

Pour organiser tous ces fichiers, il est utile d'imaginer une structure qui permettrait de pouvoir lire le nom de ces fichiers afin de savoir lesquels sont présents , de manipuler ces fichiers par envoi de commande au système , copie , effacement , renommer , etc.

## Organisation linéaire

L'organisation la plus simple , c'est de faire un tableau liant les accès logiques (par le nom) aux accès physiques (adresse physique). On appelle cette table une directory , ou directorie en francisant le terme.




Cette forme présente un seul avantage , la simplicité , mais par contre beaucoup de désavantages : très difficile à utiliser quand le nombre de fichiers augmente. Il suffit d'imaginer que l'on recherche 1 fichier parmi 200. Peu pratique pour le système car il doit parcourir toute la liste pour aller d'un bout à un autre.

Cette organisation est maintenant abandonnée mais elle à été la première à être implémentée sur les premiers micro , CPM.

## Organisation arborescente

L'idée est de constater que très peu de choses différencient la tête de la table de fichiers d'un descripteur de fichier. Si on pousse la logique jusqu'au bout , on peut dire qu'un descripteur peut servir à décrire un fichier comme une liste de fichiers.. On a donc une même structure qui décrit la liste et les fichiers. En appliquant un raisonnement récursif , on est capable de dire que la liste peut contenir une sous-liste , et on obtient une arborescence.

Un directorie contient des fichiers et des sous-directories.

Cela présente de nombreux avantages :

* Avoir peu de fichies par directorie , cela créant un besoin de classement hiérarchisé du plus heureux effet.
* Avoir des fichiers qui portent le même nom mais qui ne sont pas dans la même directorie[20].
* Pouvoir placer des indicateurs de droit d'accès , permettant de protéger l'accès de sous-directories et donc protéger les fichiers qui y sont.
* Etre capable de créer des sous-arbres dédiés à un seul utilisateur. On a ainsi un système multi-utilisateurs.

[20] Cela semble stupide au premier abord , mais on s'apperçoit que le choix des noms pour les fichiers est un vrai casse- tête , il faut que le nom ait une signification , et bien sûr , le nombre de lettres qui le composent est limité. C'est un problème qui existe aussi en programmation au niveau des noms de variables , et qui est loin d'être simple.

Nous allons donner à titre d'ex. l'organisation que nous avons actuellement sur notre disque DOS , et dont une partie nous sert pour rédiger ce texte (à la date de 1997 sous système DOS 6.2).

| | |
|---|---|
| Partition DOS de | 300 Mo |
| Nombre de directories : | 427 |
| Nombre de fichiers : | 5 482 |
| Nombre de secteurs utilisés : | 450 218 |
| Place en octets prise par les fichiers : | 229 137 684 |

Nous donnons la liste de nos fichiers et sous-directories situés sous la racine.

```
 Volume in drive C is DOS
 Volume Serial Number is 0D50-15E3
 Directory of C:\

3X           <DIR>                05.07.93   15:59
AR           <DIR>                05.07.93   16:08
MICH         <DIR>                25.08.93   11:01
TEL          <DIR>                04.07.93   19:42
TP           <DIR>                05.07.93   10:10
U            <DIR>                04.07.93   19:42
AUTOEXEC BAT          398         06.07.93   11:29
COMMAND  COM        47856         15.09.91   16:03
CONFIG   ASP           74         21.10.92   10:39
CONFIG   SYS          336         04.10.93   10:57
G        BAT           30         11.09.93    2:08
PTH      BAT          699         04.10.93   10:37
WINA20   386         9349         09.04.91    5:00
MIRROR   FIL        95744         04.10.93   13:02
      15 file(s)     170890 bytes
                   49307648 bytes free
```

Tous les systèmes modernes utilisent au moins cette organisation arborescente.

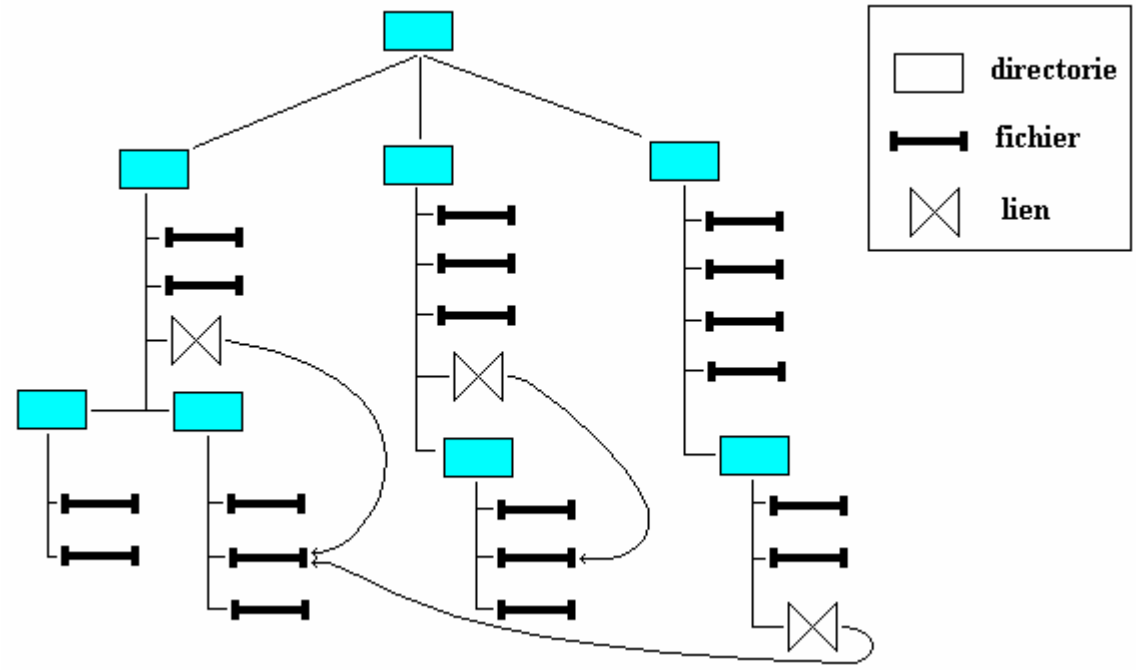
## Les DAG

Dans une organisation arborescente pure , si on désire que 2 sous-arbres puissent avoir un même fichier , configuration d'une ressource commune : par ex. une imprimante , il faut :

§ Copier le fichier 2 fois. Il y a alors un problème de mise à jour du fichier.

§ Mettre ce fichier dans un autre sous-arbre , et dire aux 2 sous-arbres qu'il existe un fichier à cet endroit. Il y a alors un problème de mise à jour de l'indication de l'endroit où se trouve le fichier.

§ Le fichier est dans l'un des sous-arbres et on crée une liaison sur ce fichier dans l'autre sous-arbre. Cela règle le problème des mises à jour , mais l'arbre devient alors un graphe. C'est ce que l'on appelle un DAG[21].




Le système d'organisation de fichiers au moyen d'un DAG , présente l'avantage d'avoir plusieurs références possibles sur 1 seul fichier. Cela permet une économie de place , par contre il faut prendre un soin extrême à ne pas créer de cycle dans les liens qui conduisent au fichier , sous peine de perdre l'accès au fichier.

UNIX contient une petite ébauche de DAG avec le système de référence de fichier.

[21] Directed Acyclic Graph. C'est un graphe qui ne possède pas de cycle mais on peut aller par différents chemins sur un même élément.

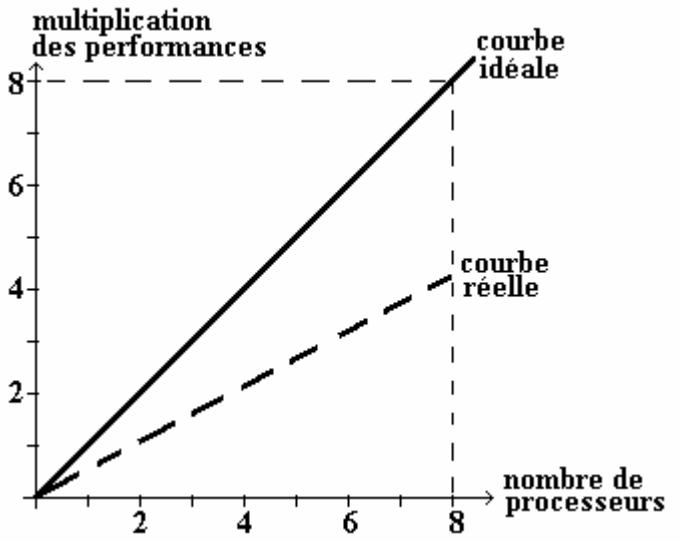
# Système multiprocesseurs

Bien que la puissance des ordinateurs augmente de modèle en modèle , il faut se rendre à l'évidence , et admettre que de nombreux pans de recherches , ou d'activités , ne peuvent pas résoudre des quantités importantes de problèmes , à cause de la lenteur de calcul encore trop grande des meilleurs calculateurs.

L'idée de Von Neuman et d'Alan Turing de faire travailler une unité centrale de façon séquentielle , porte en elle une notion de limites : des performances (vitesse de calcul) , tailles , etc.

Il est envisageable de se dire : si un processeur exécute un travail en un temps t , peut-on extrapoler et dire que x processeurs travaillant tous en même temps mettront $\frac{t}{x}$ temps ?

On aimerait bien trouver une organisation qui permettrait d'avoir la courbe d'évolution suivante.




En réalité les performances obtenues ne croissent pas linéairement , à cause des phénomènes de communication d'informations entre les processeurs , qui ralentissent les vitesses de traitement.

Il existe plusieurs voies de recherche pour paralléliser les ordinateurs. Elles passent toutes par une augmentation du nombre de processeurs. Physiquement il y a 2 grandes familles d'architectures.

## Systèmes SIMD

Dans les systèmes SIMD[22] tous les processeurs exécutent la même instruction , mais sur autant de data qu'il y a de processeurs. Il n'y a qu'une mémoire vive générale pour tous les processeurs. La Connexion Machine a par exemple 16 000 processeurs qui travaillent en même temps , cela lui permet d'effectuer des calculs avec une puissance de l'ordre de 2 Gflops[23]. A titre de comparaison un 486DX 33 possède une puissance d'environ 3 Mflops , c'est 1000 fois moins.

[22] SIMD : Single Instruction Multiple Data.

[23] Gflops : Giga flops = giga floating operations par seconde (giga = milliard).

# Systèmes MIMD

Une autre alternative , c'est les systèmes MIMD[24]. Dans cette architecture chaque processeur possède sa propre zone mémoire et échange avec les autres processeurs des informations par l'intermédiaire d'un ou plusieurs bus à haute vitesse. Beaucoup de constructeurs travaillent sur cette architecture. Intel a construit avec la DARPA (Defence Advanced Research Projet Agency) un ordinateur contenant 2 000 processeurs i860 , qui peut effectuer des opérations à environ 100 Gflops.

D'autres utilisent le nouveau processeur de DEC , l'Alpha , et ils pensent arriver à une puissance d'environ 500 Gflops avec 2 000 processeurs.

## Paralléliser les logiciels

Quelle que soit l'architecture , il faut savoir qu'écrire des programmes pour une architecture parallèle est un affaire longue et compliquée. L'idée est de faire faire le travail par l'ordinateur lui-même. On écrit un programme normalement , comme si c'était un ordinateur classique , et c'est la machine sur laquelle il doit tourner qui le parallélise. Cela suppose que la parallélisation prenne en compte tous les aspects de gestion de ressources communes , d'allocation mémoire , de synchronisation entre les processus , etc. qui sont très loin d'être simples et opérationnels.

Mach de l'université du MIT , et Chorus de Chorus Système , qui sont des systèmes à base de micro-noyaux , permettent d'envisager avec espoir la parallélisation des systèmes.

## Conclusion

Les domaines utilisant le calcul massif sont aussi divers que : la simulation , la CAO , l'intelligence artificielle , etc. Cela touche tous les domaines des sciences , de la mathématique à l'ingénierie en passant par la biologie et l'économie. Il faut donc de plus en plus de performances de calcul.

Des recherches nombreuses sont effectuées en vue d'arriver en l'an 2 000 à avoir des ordinateurs , qui soient capables de tourner en affichant une puissance d'un Tflops (un téra flops = mille milliards d'opérations flottantes à la seconde).

Des recherches tout aussi nombreuses se font pour savoir comment parralléliser les programmes et les algorithmes afin de pouvoir les porter sur ces nouvelles machines.

C'est LE domaine en "ébullition" dans l'informatique d'aujourd'hui. Tant que l'on n'aura pas trouvé une (ou des) architecture et une (ou des) programmation permettant la parallélisation massive , on sera bloqué , et il faudra attendre avant d'aller plus loin.

Le joyeux désordre des débuts de cette nouvelle voie,  a fait place à des enjeux considérables de la part des grosses entreprises. Tous les constructeurs sérieux ont , sinon commercialisé , du moins dans leurs cartons , un ordinateur parallèle. Qui contrôlera le calcul massif contrôlera les techniques futures et par là l'économie.

[24] MIMD : Multiple Instruction Multiple Data

# Bibliographie

## Mise en garde

La rédaction a été faite pour servir de support de cours dans le cadre de la formation pour adultes. Il est bien évident que ce texte est imparfait sur autant de points que le lecteur voudra, son seul mérite étant d'exister ; quelques brillants esprits pouvant même reprocher ce dernier point. Cependant il faut savoir que le rédacteur accepte avec joie et humilité toutes suggestions constructives , permettant d'améliorer ce texte.

Le nombres d'ouvrages consacrés aux systèmes d'exploitation est considérable , il n'est donné ici que la bibliographie ayant servi à la constitution de ce texte.


| | | |
|---|---|---|
| J.Beauquier<br>B. Bérard | **Systèmes d'exploitation**<br>**Concepts et algorithmes** | Mc Graw Hill |
| M.J. Bach | **Conception du système UNIX** | Masson |
| CI. Lhermitte | **Les systèmes d'exploitations**<br>**Structure et concepts fondamentaux** | Masson |
| A. Tanenbaum | **Les systèmes d'exploitations**<br>**Conception et mise en oeuvre** | InterEditions |


Les lecteurs avides d'approfondir la question sont invités à rechercher en bibliothèque les références qui leurs manqueraient , ou parcourir les rayons des librairies spécialisées en informatique.

# Table des matières

# Cours de Système

Arno LLOP