Hiérarchie mémoire

N. Louvet

UCB Lyon 1

LIF6 - Automne 2014

Hiérarchie mémoire

Plan

La mémoire centrale permet de stocker à la fois les programmes en cours d'exécution et les données en cours de traitement.

La taille de la mémoire des micro-ordinateurs continue de s'accroître, mais elle sera toujours insuffisante : nous développerons toujours de nouvelles applications capables de la saturer...

De plus, il s'agit d'une mémoire volatile, perdue à l'extinction de la machine...

Cela motive, l'utilisation de *mémoires secondaires*, qui viennent compléter la mémoire centrale :

- les cartouches et bandes magnétiques.
- les disques magnétiques : les antiques disquettes, les disques durs.
- les disques à lecture/écriture optiques : CDROM, CD-RW, DVD (*Digital Versatil Disc*)...
- les supports à base de mémoire flash : clés usb, les disques SSD...

Les mémoires peuvent être caractérisées par :

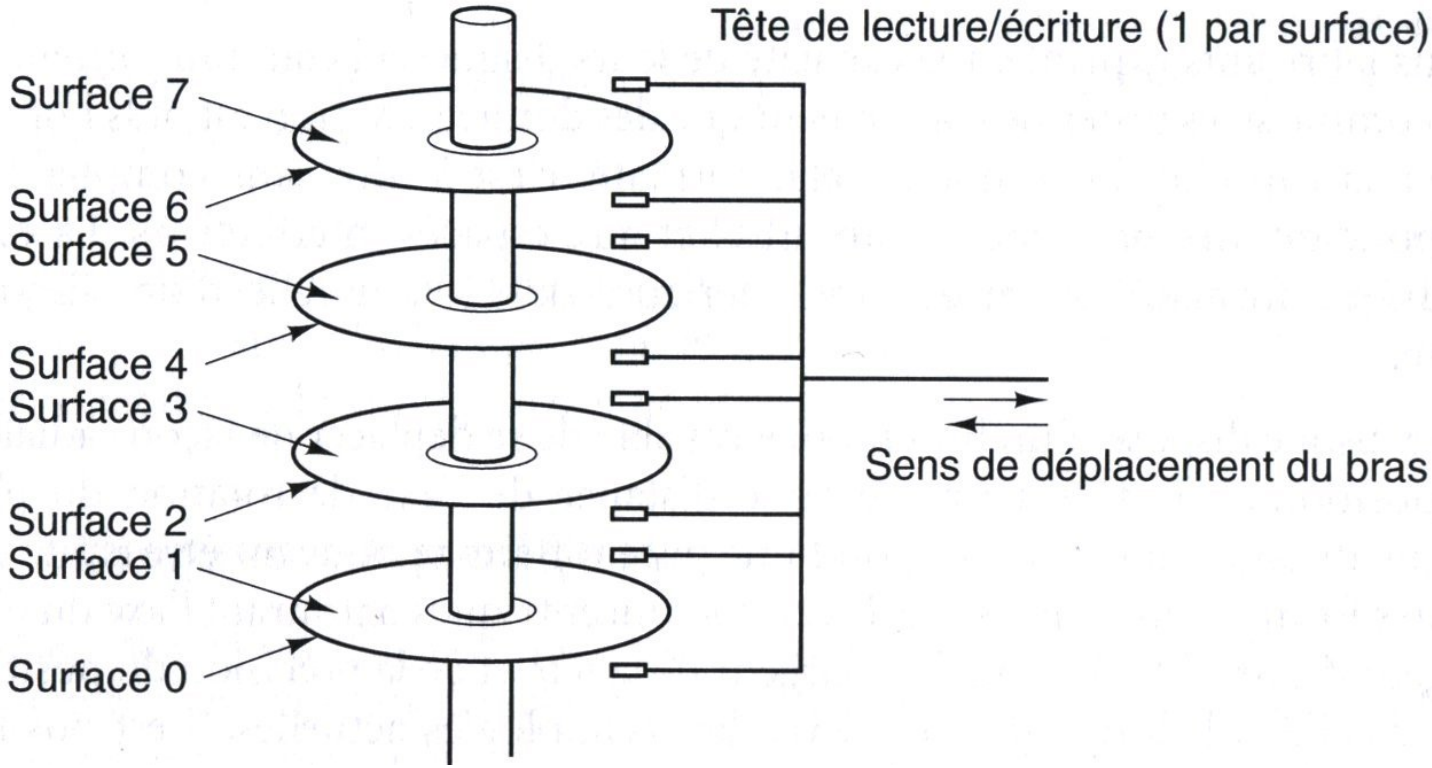
- *le temps d'accès* : c'est le temps nécessaire à l'écriture ou à la lecture d'un octet. C'est donc le temps qui s'écoule entre le lancement d'une opération d'accès et son accomplissement.
- *le débit* : c'est la quantité d'information qu'il est possible de transférer sur, ou de charger depuis le support par unité de temps.

Pour comprendre les différences entre tous ces supports, on considère le tableau suivant (seuls les ordres de grandeurs comptent) :

support	**temps d'accès**	**débit**	**capacité**
registres	1 ns		≈kio
mémoire RAM	5-60 ns	1-20 Gio/s	≈Gio
disques durs	3-20 ms	10-320 Mio/s	≈Tio
CD	120 ms	1-8 Mio/s	650 Mio
DVD	140 ms	2-22 Mio/s	4.6-17 Gio

Rappel : nano $= 10^{-9}$, micro $= 10^{-6}$, milli $= 10^{-3}$.

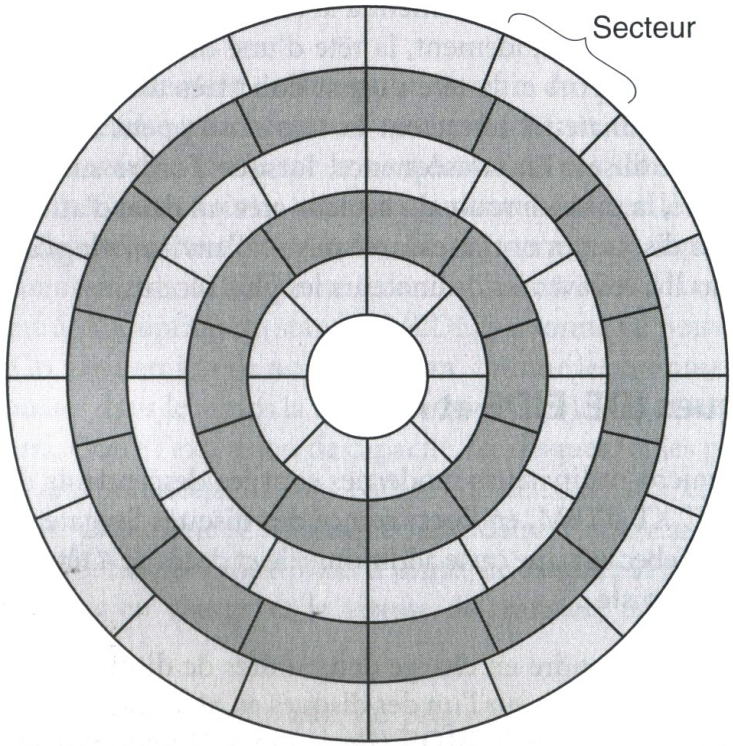
Ex : Un *disque dur* est formé de plateaux couverts d'une couche de matériau magnétisable. Une tête de lecture/écriture comporte une bobine d'induction.



Les plateaux sont maintenus en rotation rapide autour de leur axe.

- Écriture : la bobine crée un champ magnétique qui oriente les particules du disque, dans deux sens opposés en fct du signe du courant dans la bobine.
- Lecture : quand la tête passe au dessus d'une partie magnétisée, un courant est induit dans la bobine, dont le signe dépend du sens du champ.

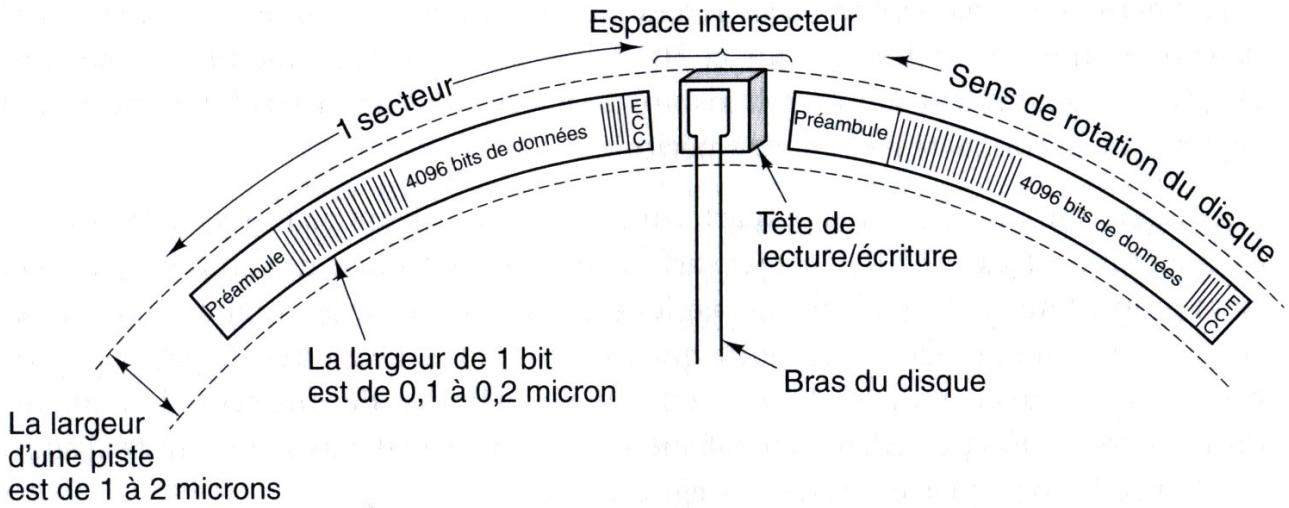
La séquence de bits circulaires pouvant être inscrite sur une surface au cours d'une rotation est appelée *piste*. Chaque piste est divisée en un certain nombre de *secteurs* comportant chacun un même nombre d'octets, par exemple 512 o.



Chaque disque dispose d'un bras mobile capable de déplacer sa tête de lecture/écriture de façon radiale, pour venir la placer sur une piste donnée.

Chaque secteur comporte :

- une zone préambule pour identifier le secteur avant écriture ou lecture ;
- par exemple, 4096 bits (512 o) de données ;
- une zone de données destinées à la correction et détection d'erreurs (ECC).



Le temps d'accès aux données dépend essentiellement :

- du temps nécessaire au positionnement de la tête sur la bonne piste ;
- du délai qui s'écoule avant que le secteur cherché passe sous la tête.

C'est pourquoi il est inefficace d'effectuer des accès aléatoires sur un disque dur.

Notion de hiérarchie mémoire

Le temps d'accès pour la mémoire RAM est de l'ordre de 10 fois plus long que pour les registres ! Les concepteurs essayent de compenser cette lenteur relative en introduisant une mémoire rapide entre les registres de l'UCT et la mémoire centrale. Il s'agit de la *mémoire cache*, qui sert de zone de stockage temporaire des données : les mots les plus souvent utilisés y sont conservés.

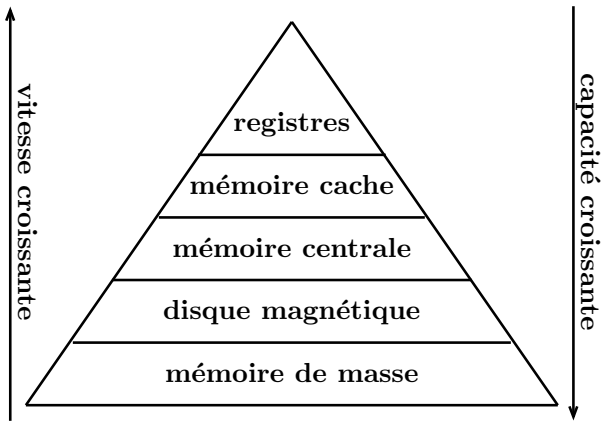
support	**temps d'accès**	**débit**	**capacité**
registres	1 ns		≈kio
cache	2-3ns		≈Mio
mémoire RAM	5-60 ns	1-20 Gio/s	≈Gio
disques durs	3-20 ms	10-320 Mio/s	≈Tio
CD	120 ms	1-8 Mio/s	650 Mio
DVD	140 ms	2-22 Mio/s	4.6-17 Gio

Rappel : nano $= 10^{-9}$, micro $= 10^{-6}$, milli $= 10^{-3}$.

La conception de la mémoire pose les questions suivantes : « combien ? », « à quelle vitesse ? », « à quel prix ? » Il y a un compromis à trouver :

- plus la mémoire est rapide, plus son coût par bit est élevé ;
- plus la capacité est grande,
  - plus son coût par bit est faible ;
  - mais plus les temps d'accès sont importants.

Une solution est de mettre en place une *hiérarchie mémoire* : Les mémoires rapides, de faibles capacité et très coûteuses sont secondées par des mémoires plus lentes, mais de plus grande capacité et à faible coût. On simule ainsi une mémoire de grande capacité, performante en moyenne, pour un coût raisonnable.



Un exemple

Supposons que le processeur ait accès à deux niveaux de mémoire :

- Niveau 1 : temps d'accès de 0.01 μs ;
- Niveau 2 : temps d'accès de 0.1 μs.

Si un mot est accédé alors qu'il est stocké au niveau 1, le processeur y accède directement. S'il est au niveau 2, il doit d'abord être chargé au niveau 1, puis vers le processeur (on suppose pour simplifier que le temps pour déterminer si une donnée se trouve ou non au niveau 1 est nul).

Si par exemple 95% des accès se font au niveau 1, le temps d'accès moyen sera :

$$\begin{aligned} 0.95 \times 0.01\mu\text{s} + 0.05 \times (0.01\mu\text{s} + 0.1\mu\text{s}) &= 0.0095\mu\text{s} + 0.0055\mu\text{s} \\ &= 0.015\mu\text{s}. \end{aligned}$$

Le temps d'accès moyen est donc plus proche de 0.01μs que de 0.1μs.

Exemples de hiérarchies de caches

Les ordinateurs disposent de plusieurs niveaux de caches, afin d'améliorer les accès à la mémoire centrale.

processeur	année	L1 cache	L2 cache	L3 cache
Intel 80486	1989	8 kio	-	-
Intel Pentium	1992	8 kio/8 kio	256-512 kio	-
IBM Power PC G4	1999	32 kio/32 kio	256-1024 kio	2 Mio
Intel Pentium 4	2000	8 kio/8 kio	256-512 kio	-
Intel Itanium	2001	16 kio/16 kio	96 kio	4 Mio
Intel Itanium II	2001	32 kio	256 kio	6 Mio

Deux valeurs séparées par un slash indiquent une séparation entre le cache d'instructions et le cache de données (instructions/données).

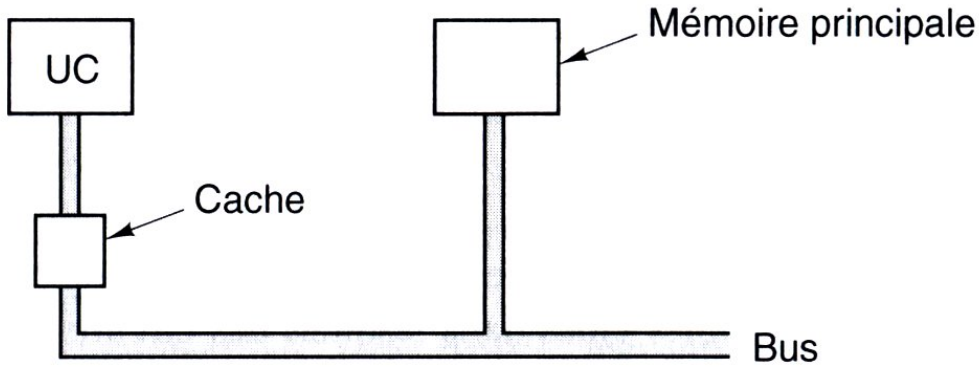
Plan

Nous allons décrire deux organisations possibles d'une mémoire cache

- cache à correspondance directe,
- cache associatif à n voies,

en nous appuyant essentiellement sur un exemple.

Pour fixer les idées, on suppose qu'un seul niveau de cache vient se placer entre la mémoire centrale et l'UCT :



Quand l'UCT réalise un accès à une donnée en mémoire :

- soit la donnée est présente dans le cache, et l'UCT y accède rapidement.
- soit la donnée est absente du cache, et elle doit d'abord y être chargée.

Cache à correspondance directe

Un ordinateur dispose d'une mémoire centrale de 2^{32} o, adressable par octets.

La mémoire principale est divisée en blocs de taille fixe, appelés *lignes de cache*.

- Chaque ligne comporte 2^5 o $= 32$ o.
- La mémoire se décompose en 2^{27} lignes de caches.

Chaque ligne peut être identifiée par un entier sur 27 bits.

Supposons que l'on dispose d'une *mémoire cache* de 64 kio composée de

- $2048 = 2^{11}$ entrées de 32 o.

Comment utiliser ces 2^{11} entrées comme espace de stockage temporaire pour les 2^{27} lignes de cache de la mémoire centrale ?

D'après l'adresse d'une donnée en mémoire centrale, on doit pouvoir :

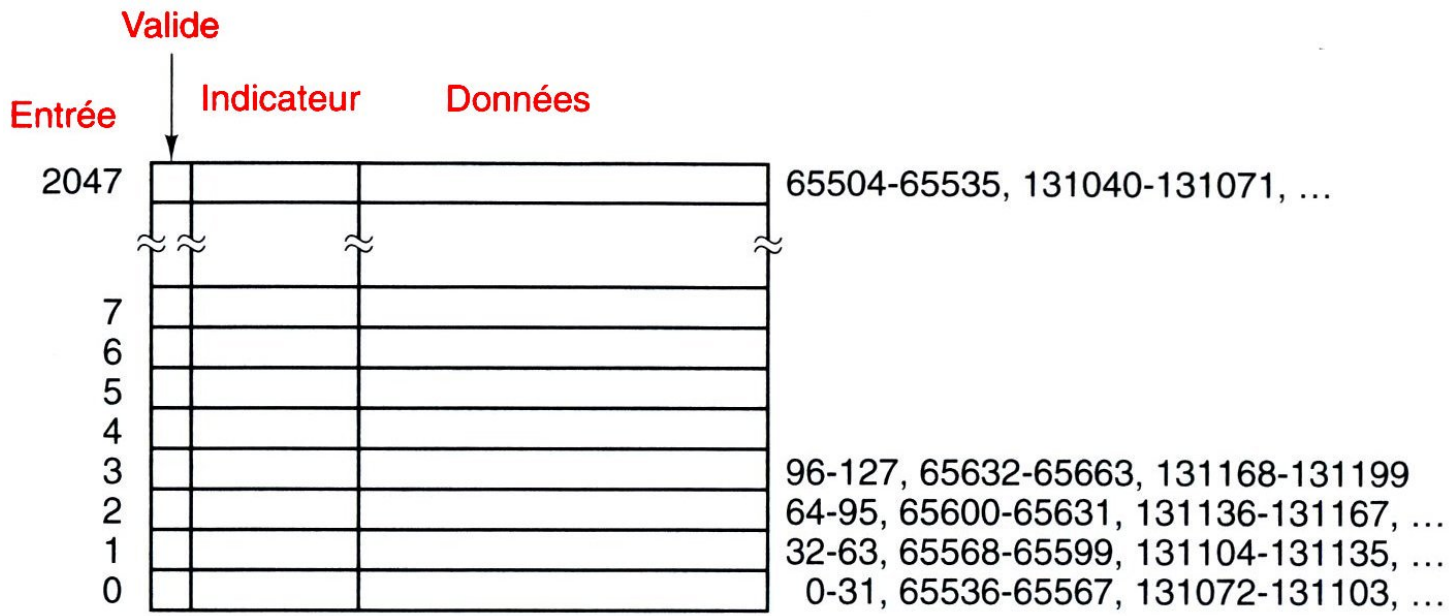
- déterminer à quelle ligne de cache elle appartient ;
- dire dans quelle entrée elle doit être rangée dans la mémoire cache.

On a 2^{11} entrées et 2^{27} lignes de cache. On décompose les adresses ainsi :

16 bits (31-16)	11 bits (15-5)	5 bits (4-0)
INDICATEUR	ENTREE	OCTET

- Les 27 bits formés de INDICATEUR et ENTREE identifient chaque ligne.
- Les 11 bits du champ ENTREE identifient une entrée du cache.
- Les 5 bits d'OCTET identifient l'un des 32 o d'une ligne ou d'un entrée.
- Toutes les adresses ayant même INDICATEUR et ENTREE appartiennent à la même ligne de cache.
- Toutes les lignes
  - ayant même ENTREE seront rangées dans la même entrée du cache.
  - ayant même INDICATEUR sont consécutives en mémoire centrale.

La mémoire cache est organisée comme ceci :



- Une ligne est stockée dans l'entrée correspondant à son champ ENTREE.
- Le champ Indicateur correspond au champ INDICATEUR des adresses, et permet d'identifier la ligne de cache d'où proviennent les données.
- Le bit Valide indique si l'entrée contient des données valides (initialisé à 0).
- Le champ Données contient la copie d'une ligne de cache.

Quand le processeur doit *lire* un octet à une adresse donnée en mémoire :

- le champ ENTREE de l'adresse indique une entrée du cache ;
- si Valide=1, et INDICATEUR=Indicateur, il y a *cache hit* ; l'octet peut être lu dans l'entrée du cache d'indice ENTREE, et sa position dans l'entrée est indiquée par OCTET. *Cela épargne un accès à la mémoire centrale.*
- si Valide=0 ou INDICATEUR≠Indicateur, il y a *cache miss* ; la ligne accédée est copiée de la mémoire vers le cache.

Les choses se compliquent pour les écritures : *comment assurer la cohérence entre les données du cache et celles de la mémoire ?* Il existe deux statégies :

- *écriture immédiate* : mise à jour simultanée du mot dans le cache et la mémoire principale ; on perd l'intérêt du cache pour les écritures...
- *écriture différée* : lorsque la ligne concernée par l'écriture est présente dans le cache, seul le cache est mis à jour. La mémoire ne sera mise à jour que lorsque la ligne sera évincée du cache.

Pour mettre en place une *écriture différée*, on peut utiliser le champ Valide pour indiquer si la ligne a été écrite depuis son chargement dans le cache.

Quand le processeur doit *lire* un octet à une adresse en mémoire :

- si INDICATEUR=Indicateur, il y a *cache hit*, la lecture est réalisée.
- sinon :
  - si Valide=1, la ligne présente dans le cache est cohérente, on se contente donc de l'évincer au profit de la ligne qui doit être lue ; Valide←1.
  - si Valide=0, la ligne présente dans le cache a été écrite : il faut la copier en mémoire avant de l'évincer au profit de celle qui doit être lue, et Valide←1.

Quand le processeur doit *écrire* un octet à une adresse en mémoire :

- si INDICATEUR=Indicateur, il y a *cache hit*, l'écriture est effectuée uniquement dans le cache, et Valide←0.
- sinon :
  - si Valide=1, la ligne présente dans le cache est cohérente, on l'évince au profit de la ligne qui doit être écrite. L'écriture est effectuée, et Valide←0.
  - si Valide=0, la ligne a été écrite, il faut la sauvegarder avant de l'évincer au profit de celle qui doit être écrite. L'écriture a lieu, et Valide←0.

Le système de cache à correspondance directe que nous venons de décrire place des lignes de caches consécutives dans des entrées consécutives du cache : 64 kio de données contiguës peuvent être stockées dans le cache.

Si on effectue des accès à tous les éléments d'un tableau de 65536 entiers sur 8 bits, on aura juste un *cache miss* par ligne de cache occupée par le tableau.

Par contre, deux lignes dont les adresses différent d'un multiple de 64 kio ne peuvent être stockée en même temps dans le cache, car elles ont le même champs ENTREE.

Si un programme effectue un accès à une adresse X, puis aux adresses $X + 65536$, $X + 2 \times 65536, \ldots$ alors il provoquera à chaque fois un *cache miss*, d'où de moins bonnes performances (cela revient à ne pas avoir de cache).

Les concepteurs des caches sont partis du principe que, souvent, le programmeur respectera le *principe de localité*, qui se décompose en deux parties :

- *localité spatiale :* les adresses mémoires « proches » d'une adresse qui vient d'être utilisée sont susceptibles d'être référencées dans un futur proche.

- *localité temporelle :* une adresse mémoire ayant fait d'objet d'un accès récent sera de nouveau utilisée prochainement.

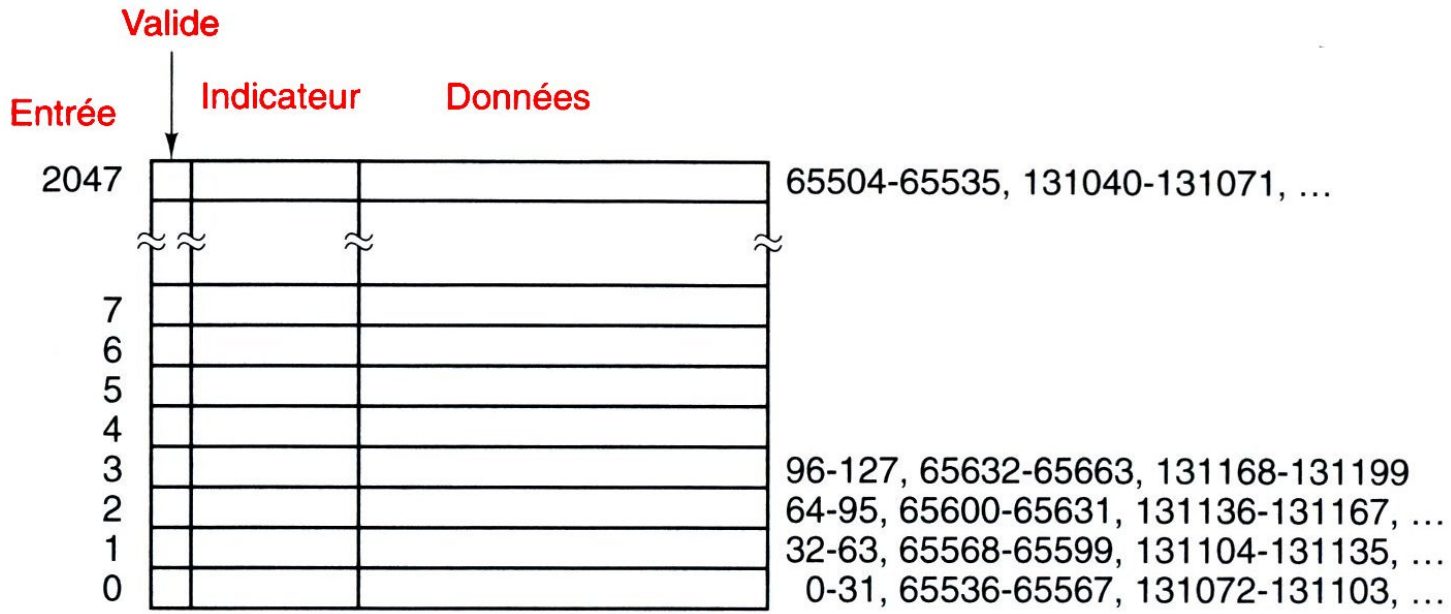
La tâche du programmeur est donc :

- de choisir des algorithmes favorisant le principe de localité,
- de programmer ces algorithmes selon le principe de localité,

afin d'améliorer les performances de ces programmes.

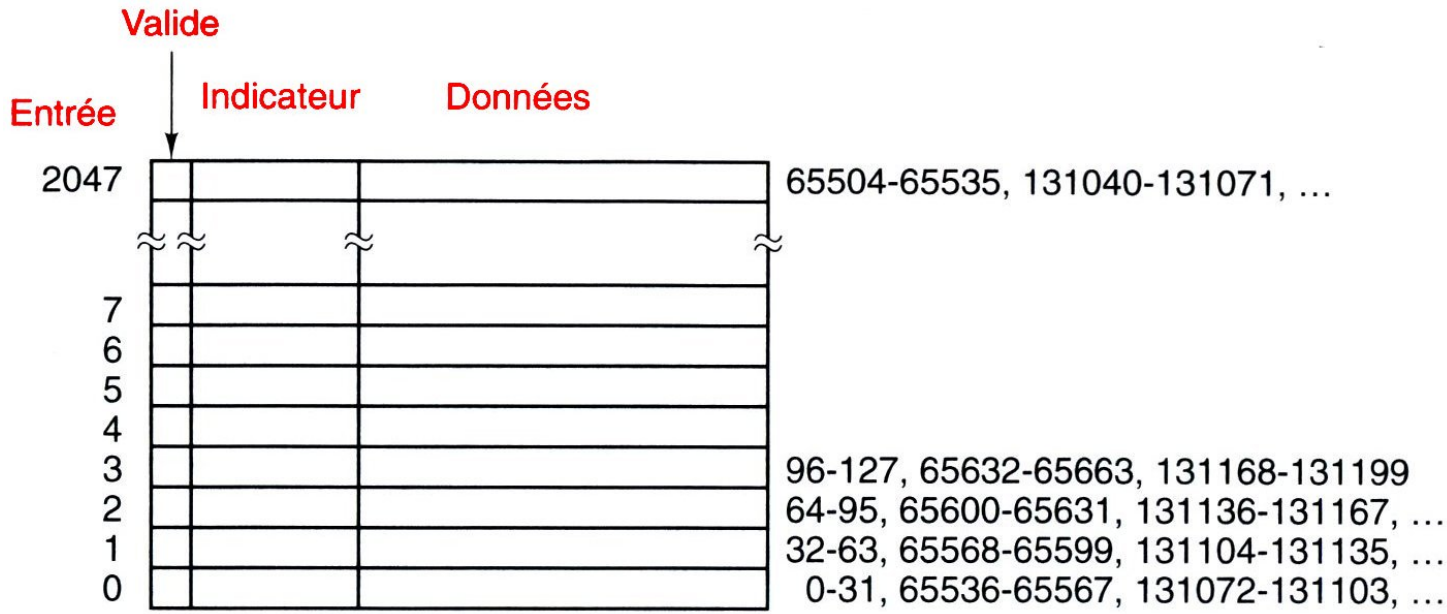
Cache associatif à n entrées

Dans un cache direct figuré ci-dessous, si un programme effectuent des accès intensifs aux adresses 0 et 65536, il y aura constamment un conflit sur l'entrée 0.



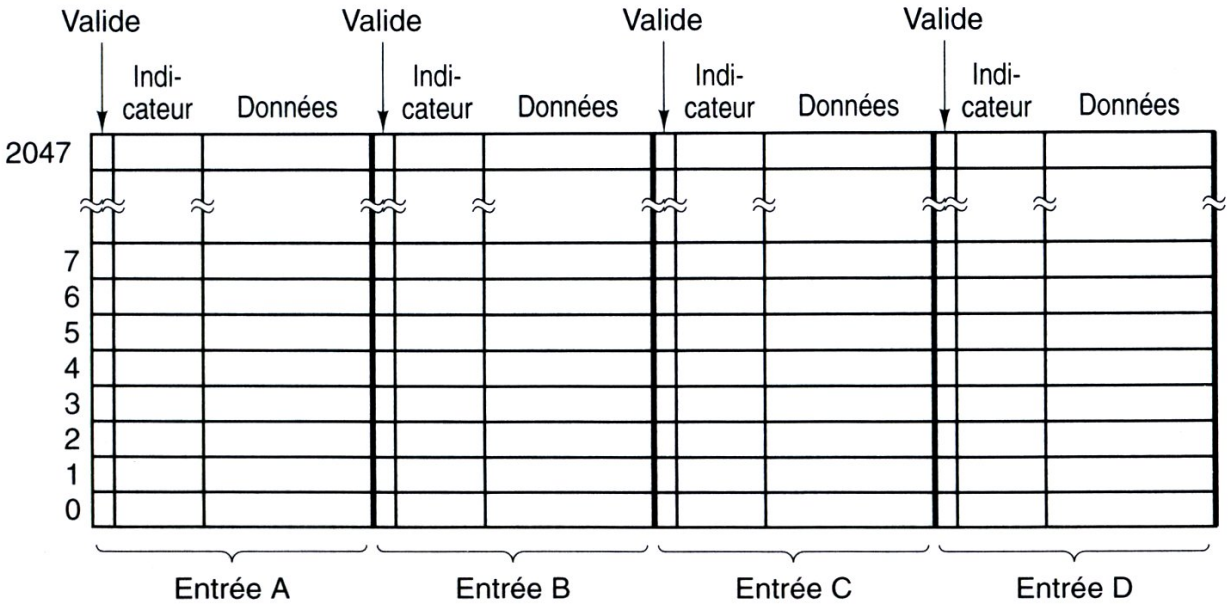
Cache associatif à n entrées

Dans un cache direct figuré ci-dessous, si un programme effectuent des accès intensifs aux adresses 0 et 65536, il y aura constamment un conflit sur l'entrée 0.



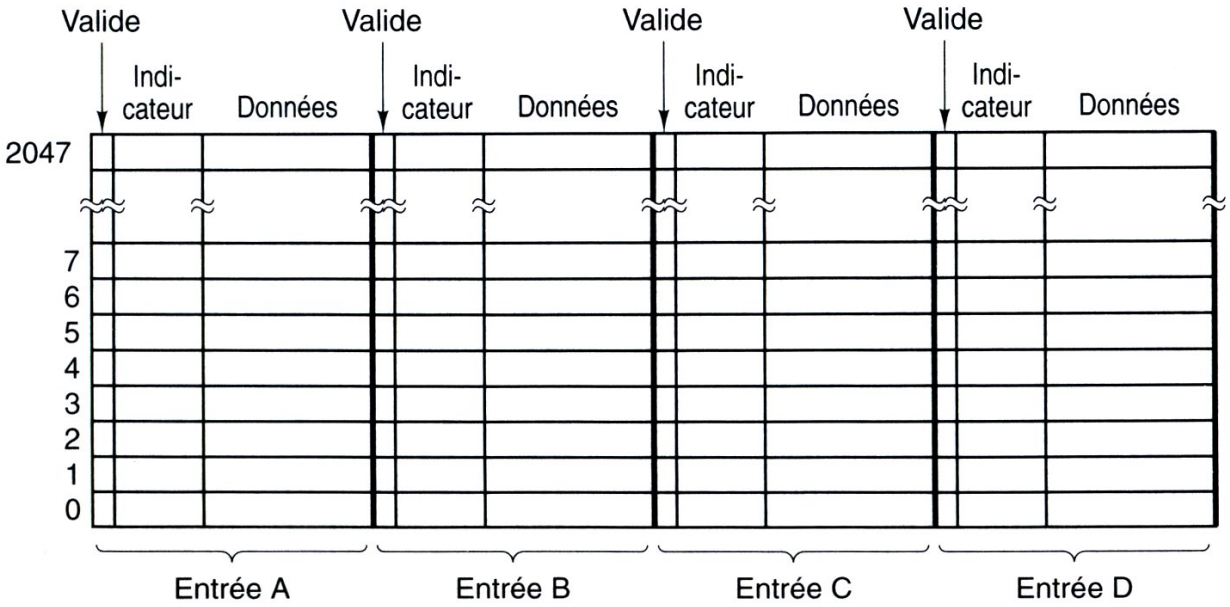
Une solution consiste à autoriser $n \geq 2$ lignes pour chaque entrées du cache.

On obtient un cache associatif à n voies (ici $n = 4$) :



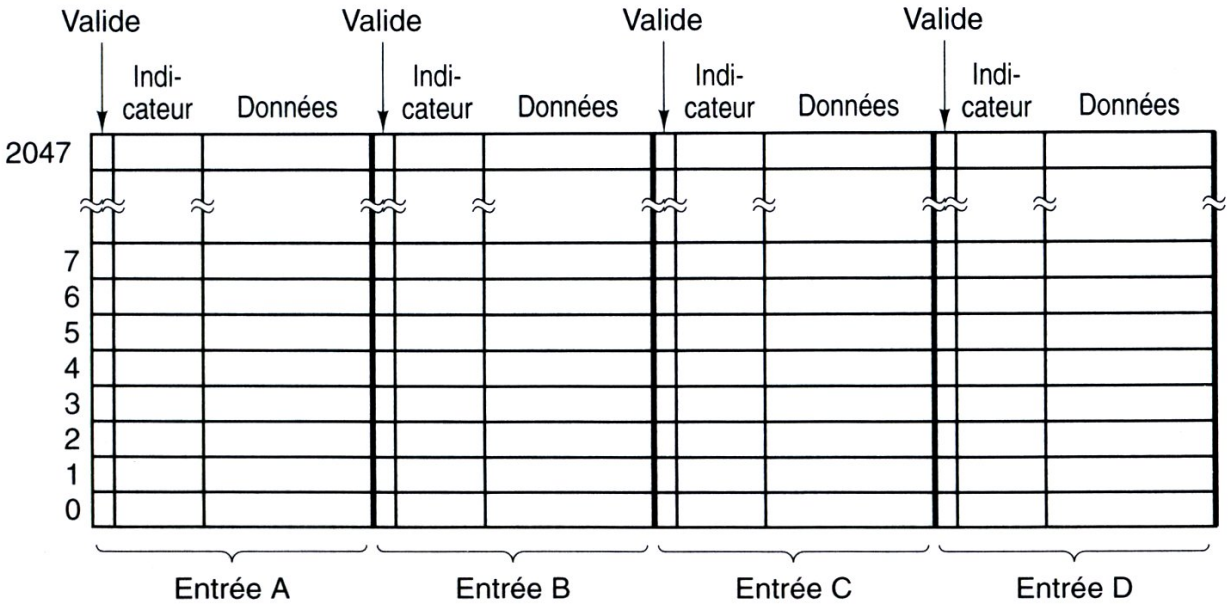
Cela semble régler une partie du problème...

On obtient un cache associatif à n voies (ici $n = 4$) :



Cela semble régler une partie du problème... Un nouveau problème : lorsqu'une nouvelle entrée doit être insérée dans le cache, quelle ligne doit en être évincée ?

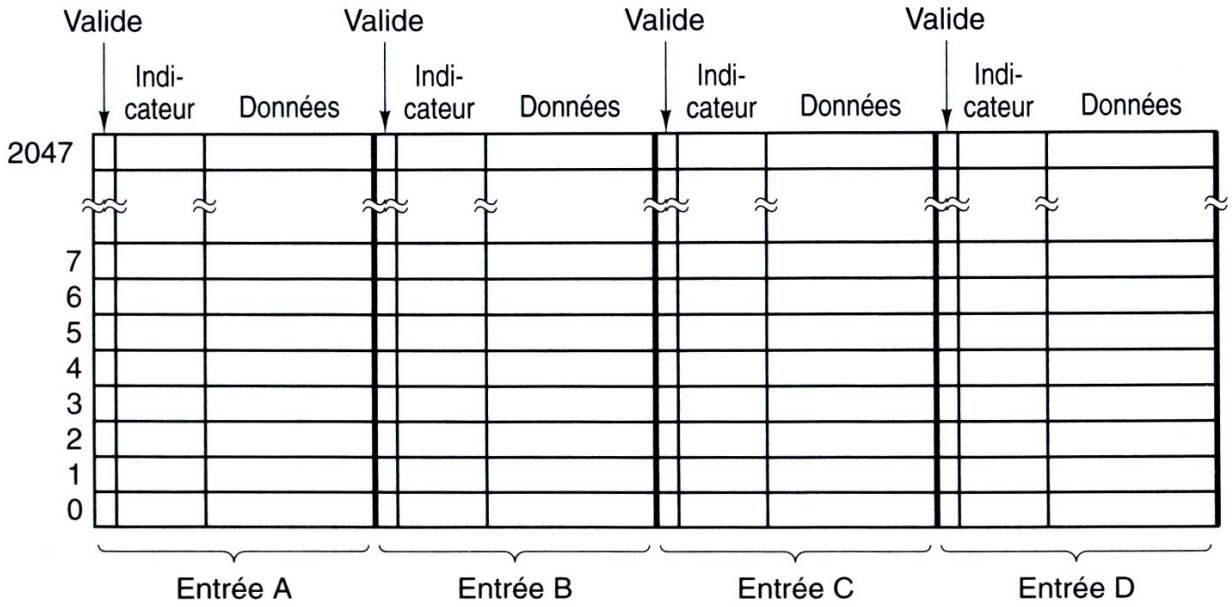
On obtient un cache associatif à n voies (ici $n = 4$) :



Cela semble régler une partie du problème... Un nouveau problème : lorsqu'une nouvelle entrée doit être insérée dans le cache, quelle ligne doit en être évincée ?

Un algorithme assez efficace est l'algorithme *LRU* (*Least Recently Used*) : on évince la ligne qui n'a pas été utilisée depuis le plus longtemps.

On obtient un cache associatif à n voies (ici $n = 4$) :



Cela semble régler une partie du problème... Un nouveau problème : lorsqu'une nouvelle entrée doit être insérée dans le cache, quelle ligne doit en être évincée ?

Un algorithme assez efficace est l'algorithme *LRU* (*Least Recently Used*) : on évince la ligne qui n'a pas été utilisée depuis le plus longtemps.

D'autres algos existent, mais pas de solution miracle : on arrive toujours à trouver un programme provoquant « beaucoup » de *cache miss*. Mais on obtient nécessairement une amélioration des performances moyennes du cache.

Plan

- Nous avons donné une idée de l'organisation de la *mémoire centrale* d'un ordinateur, et montré comment elle peut être secondée par des mémoires secondaires via une organisation hiérarchique.

- Nous avons décrit le principe des *niveaux de cache*, qui permettent de réduire le temps d'accès moyen aux données stockées en mémoire.

- On retiendra en particulier le *principe de localité* :
  - *localité spatiale :* les adresses mémoires proches d'une adresse qui vient d'être utilisée sont susceptibles d'être utilisées dans un futur proche.
  - *localité temporelle :* une adresse mémoire qui vient d'être utilisée le sera de nouveau prochainement.