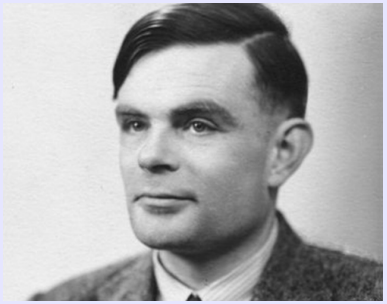
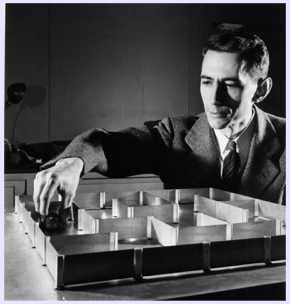
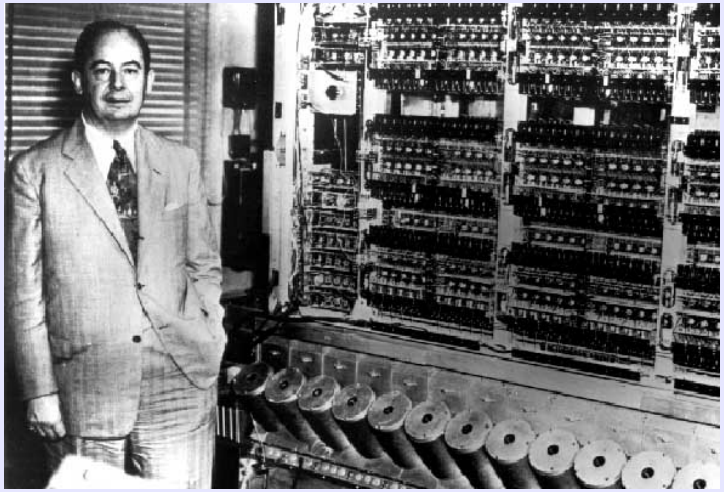




Information, Calcul et Communication

Module 3 : Systèmes

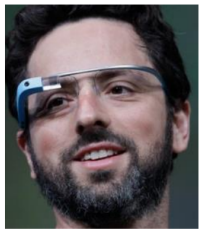
EPFL

Information, Calcul et Communication
Hiérarchies de mémoires

Prs. B. Falsafi, A. Schiper, W. Zwaenepoel, A. Ailamaki,
P. Janson & J.-C. Chappelier

Déluge de données







Divertissements

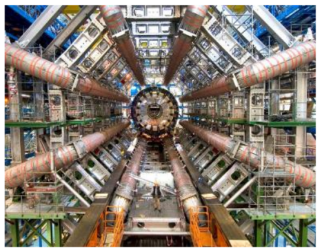
Vie

2 Zo (2012) -- 40? Zo (2020)

(1 Zo = 10^{21} octets)

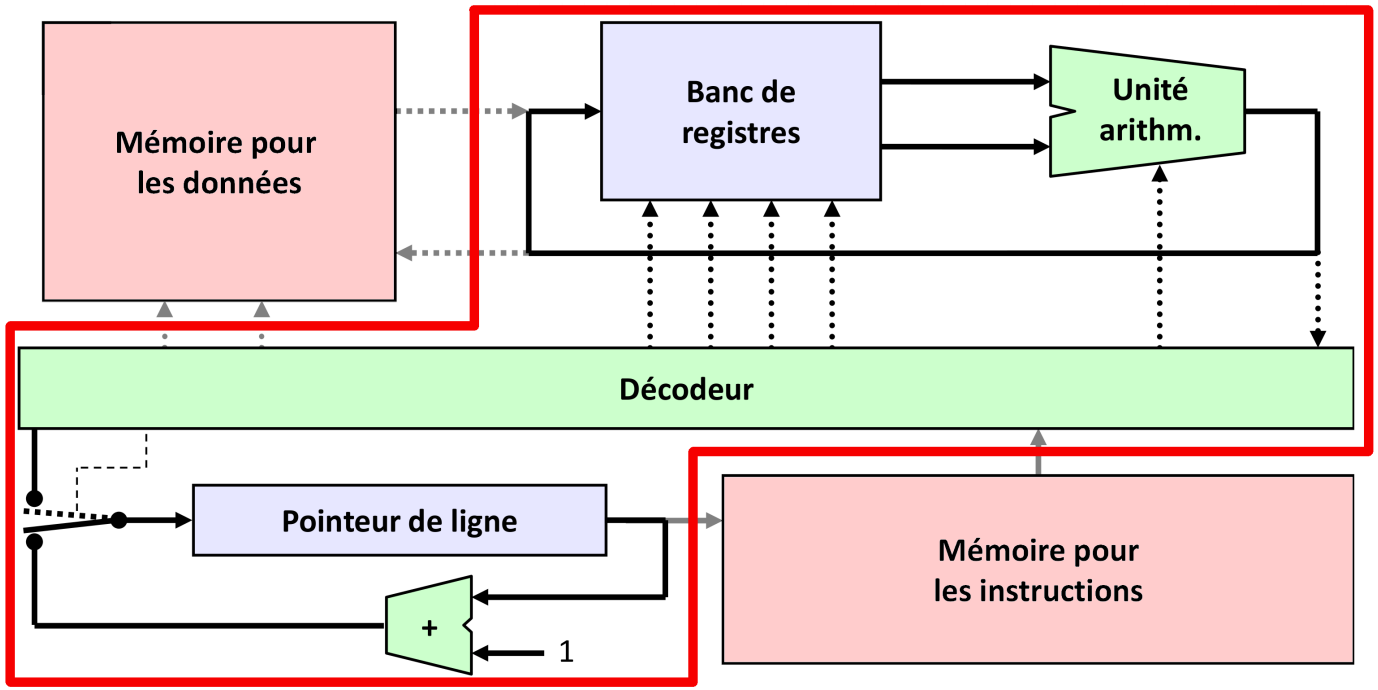
source : International Data Consortium (idc.com)

Commerce


Comment/où stocker tout ça ?

- dans les registres de processeurs ? (leçon semaine passée)
- mémoire de nos ordinateurs ?
- disques durs ?
- ... ?



Objectifs du cours d'aujourd'hui

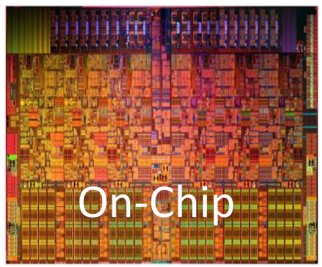
Les objectifs de cette leçon sont de :

- présenter les **technologies** de stockage de données
- présenter le *concept* de « **hiérarchies de mémoires** »
- son *intérêt*, son *fonctionnement*
- expliquer pourquoi l'ordre des **boucles en programmation** peut avoir un impact important sur la performance

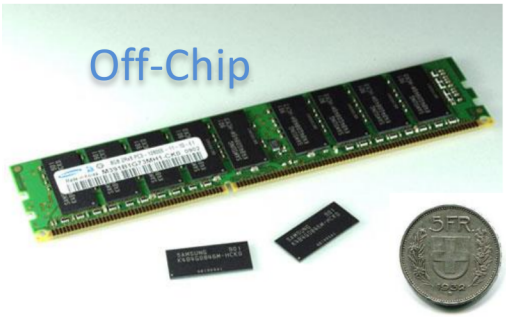
Technologies

Registres du processeur

Mémoire cache

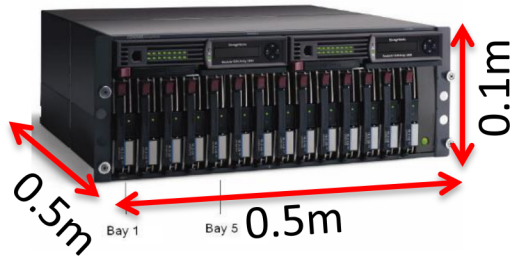


Mémoire primaire



USB FLASH

Hard Disk Array



Robot à bandes magnétiques

Disque dur

SSD

Tablette FLASH

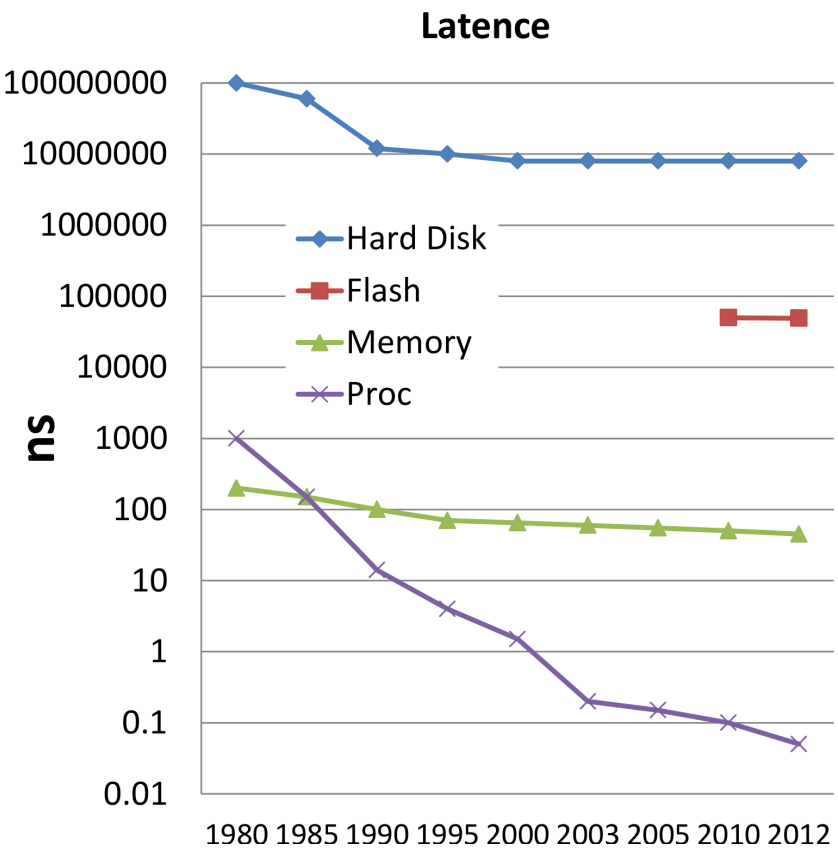
Caractéristiques importantes de chaque technologie

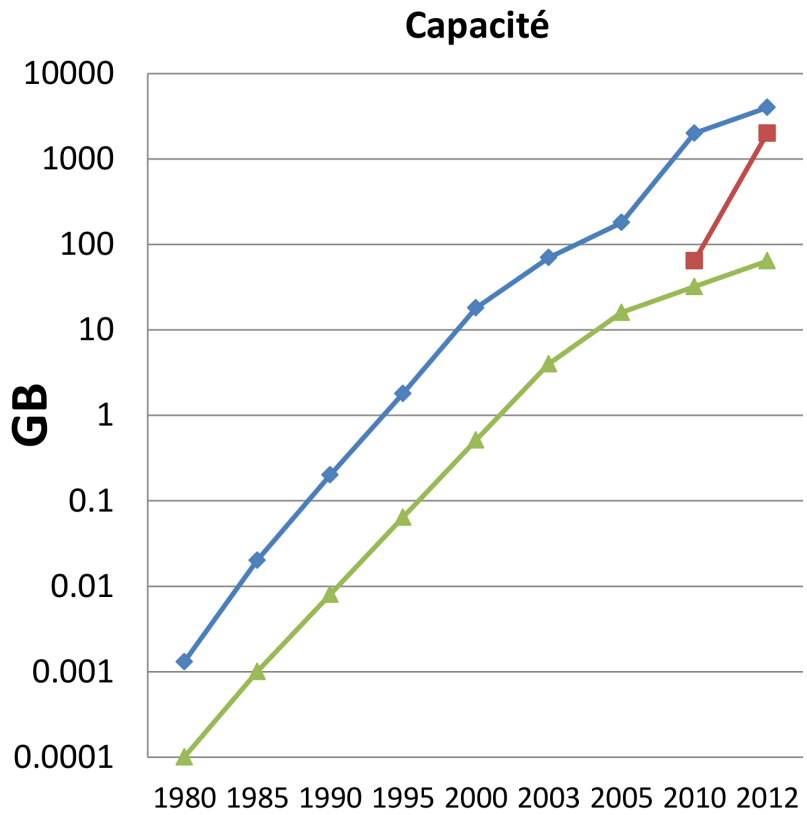
- **Performance**
  - **Latence** (s) : temps qu'il faut pour accéder à un octet donné sur le support
  - **Débit** (o/s) : nombre d'octets consécutifs auxquels on peut alors accéder par seconde
- Stockage (**Taille**)
  - **Capacité** (o) : nombre d'octets que peut contenir le support
  - **Coût** (CHF/o) : le coût unitaire (par octet) du support
- **Rétention** :
  Volatile (supports qui ne retiennent l'information que sous tension) ou **Rémanente** (supports qui retiennent l'information même hors tension)

Capacité

	Support	Texte	Image	Audio	Vidéo
10 Ko		2 pages			
100 Ko			1 photo		
1 Mo	Disquette	1 livre	1 photo HD	1 min. MP3	
10 Mo				1 min. HiFi	
100 Mo	CDs				
1 Go	DVDs			1h HiFi	1h vidéo
10 Go	blueray/RAM				
100 Go	Flash	bibliothèque		1000 CDs	
1 To	disque dur				
10 To	bandes magnétiques	bibliothèque du Congrès (US)			1000 films

Capacité et latence

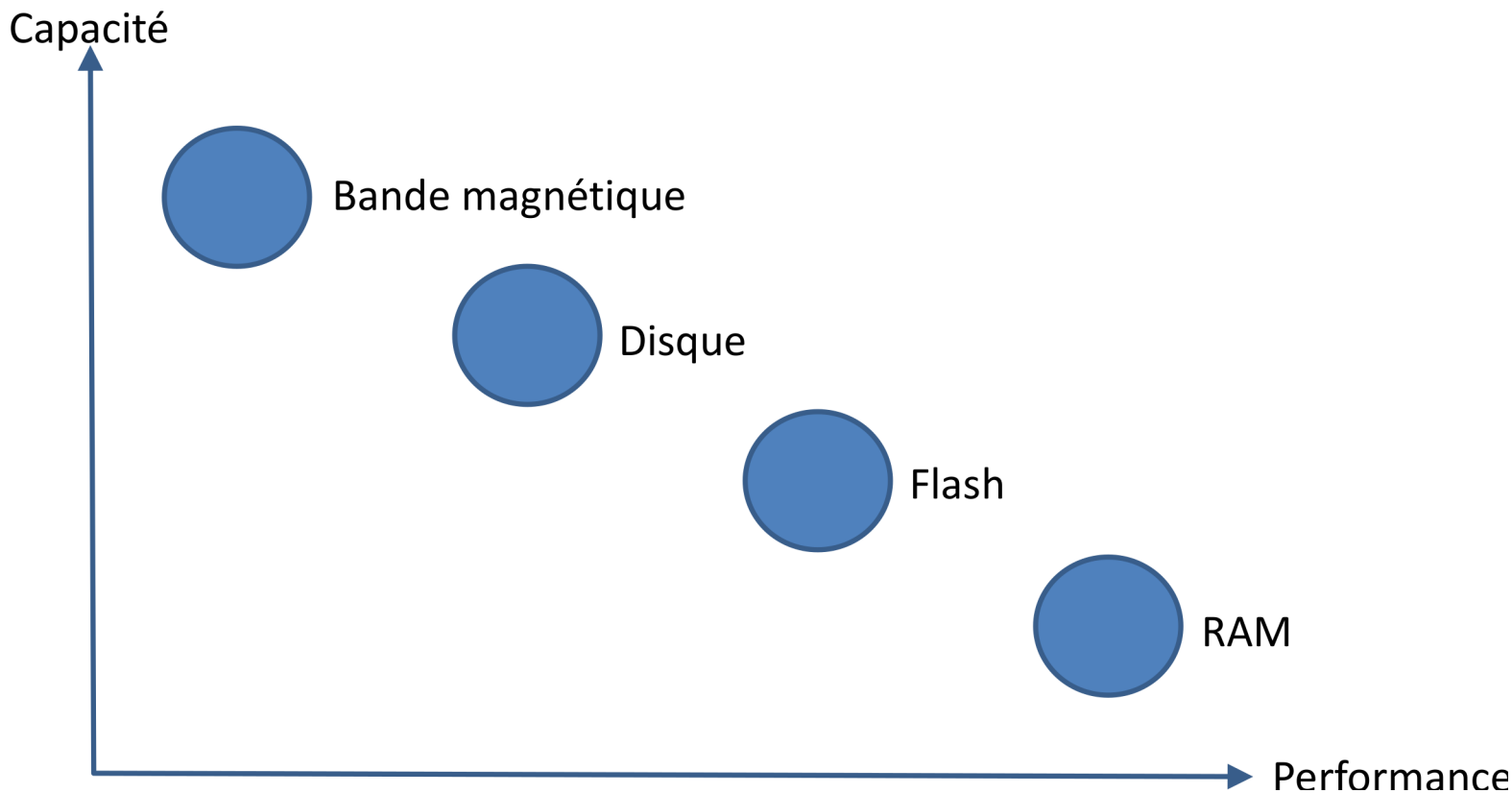




Comparaison des caractéristiques importantes de chaque technologie

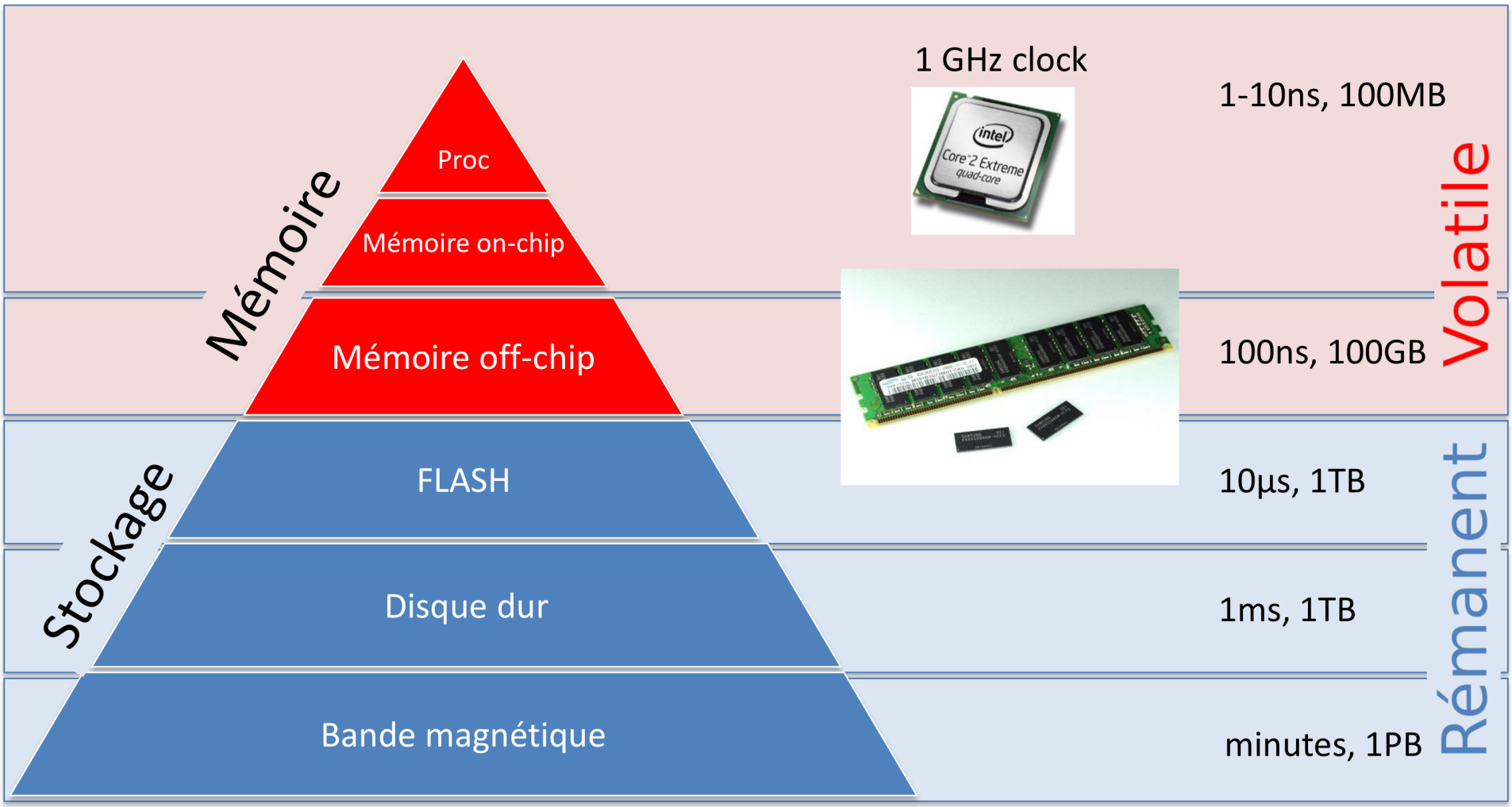
	Latence	Débit	Coût ($/Go)	Capacité	Rétention
Processeurs	ns	GHz	n.a.	n.a.	n.a.
RAM	1 - 100 ns	Go/s	10	Mo - Go	NON
Flash	µs	Go/s	0.5	To	Oui
Disques	ms	100 Mo/s	0.05	> To	Oui
Bandes magnétiques	Encore plus lent !	100 Mo/s	Encore moins cher !	Encore plus Grand !	Oui

Le problème



La mémoire **RAM** est la seule dont le **débit** et la **latence** puissent satisfaire les processeurs ...
MAIS elle présente un **coût prohibitif** et donc une **taille limitée** ...
sans compter qu'elle oublie tout quand on coupe le courant ! (**volatile**)

Le problème : vitesse ou place ?



Plan de la leçon

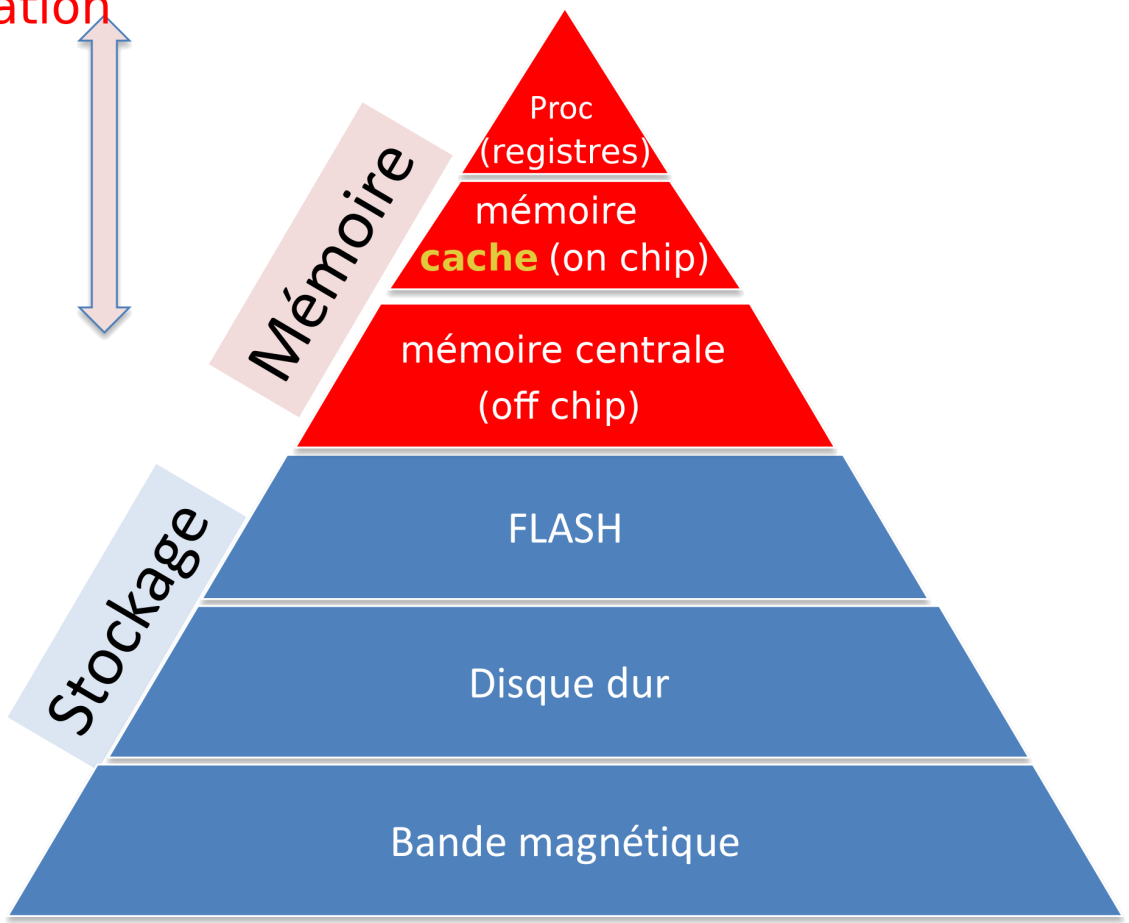
- Technologie des mémoires
- **Hiérarchie des mémoires — concept**
- Hiérarchie des mémoires — principe de fonctionnement
- Hiérarchie des mémoires — réalisation
- Un exemple
- Hiérarchie des mémoires — pourquoi ça marche ? (analyse)
- Attention à vos boucles !

Une solution : mémoires hiérarchiques

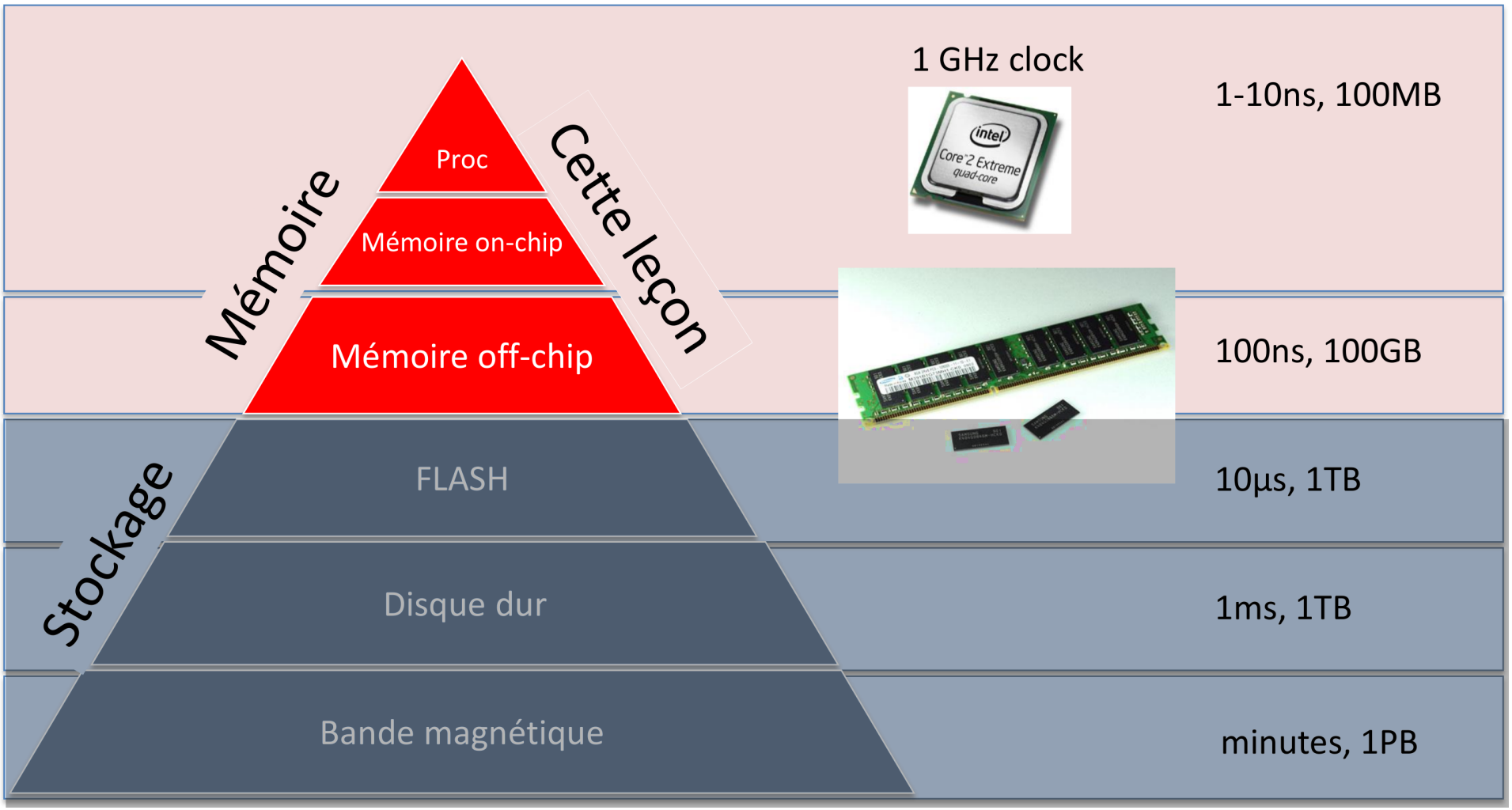
Principe : on garde en **mémoire rapide** ce que l'on utilise **momentanément**

Informations en cours d'utilisation
Gérées automatiquement
(utilisateur/programme
pas impliqué)
Volatile

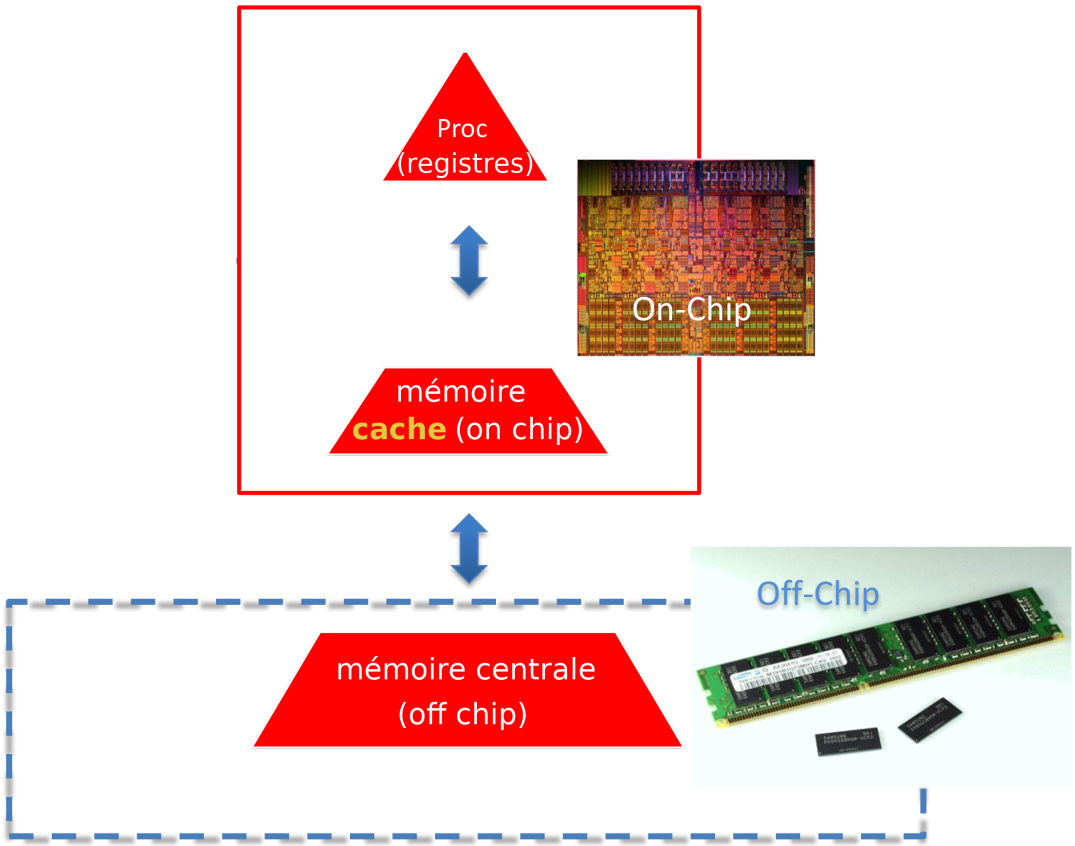
Informations archivées
Gérées "manuellement"
(par utilisateur
ou programme)
Perenne



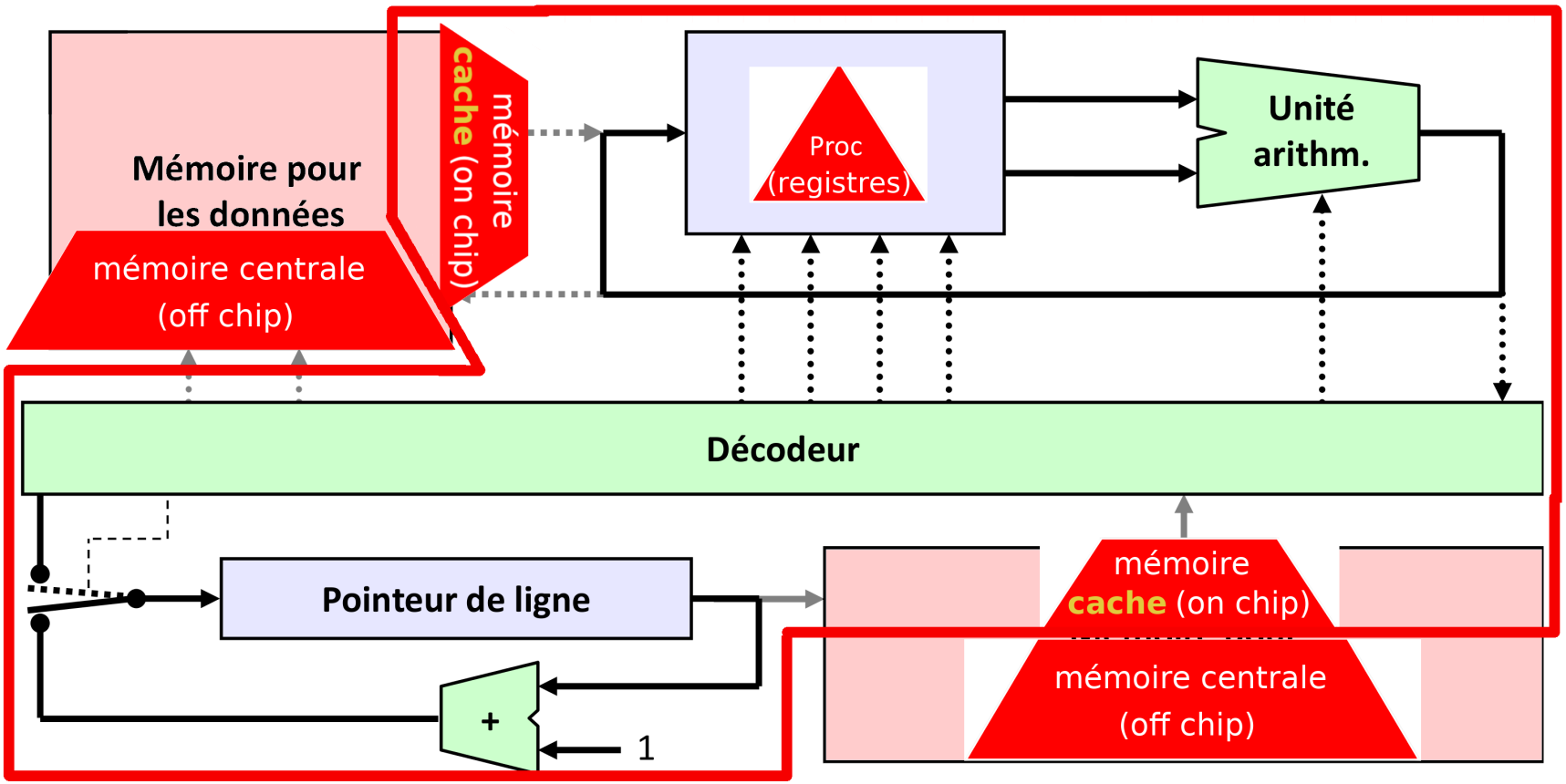
Hiérarchie de mémoires/stockage



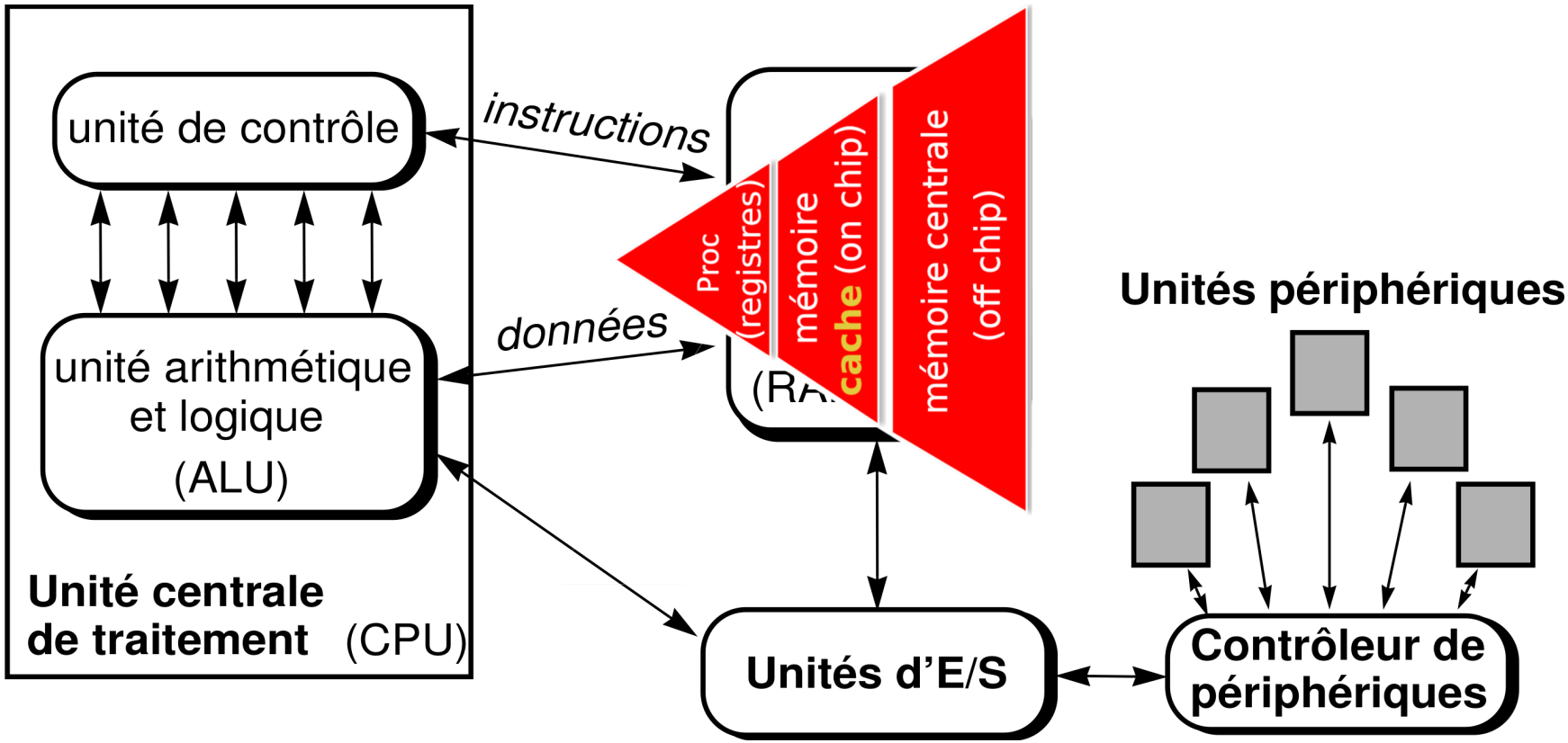
Mémoire cache et mémoire centrale



Lien avec la leçon III.1



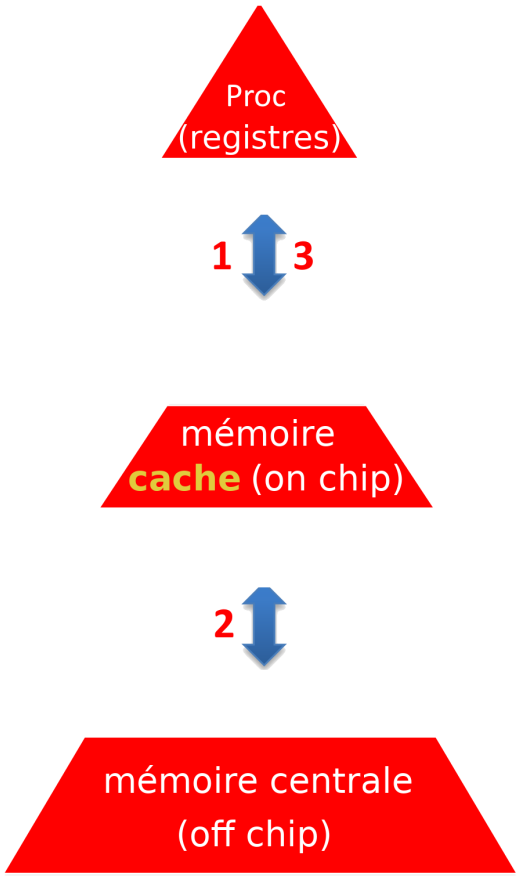
Lien avec l'architecture de von Neumann



Mémoire cache et mémoire centrale

Si l'information requise par le processeur est **hors** cache

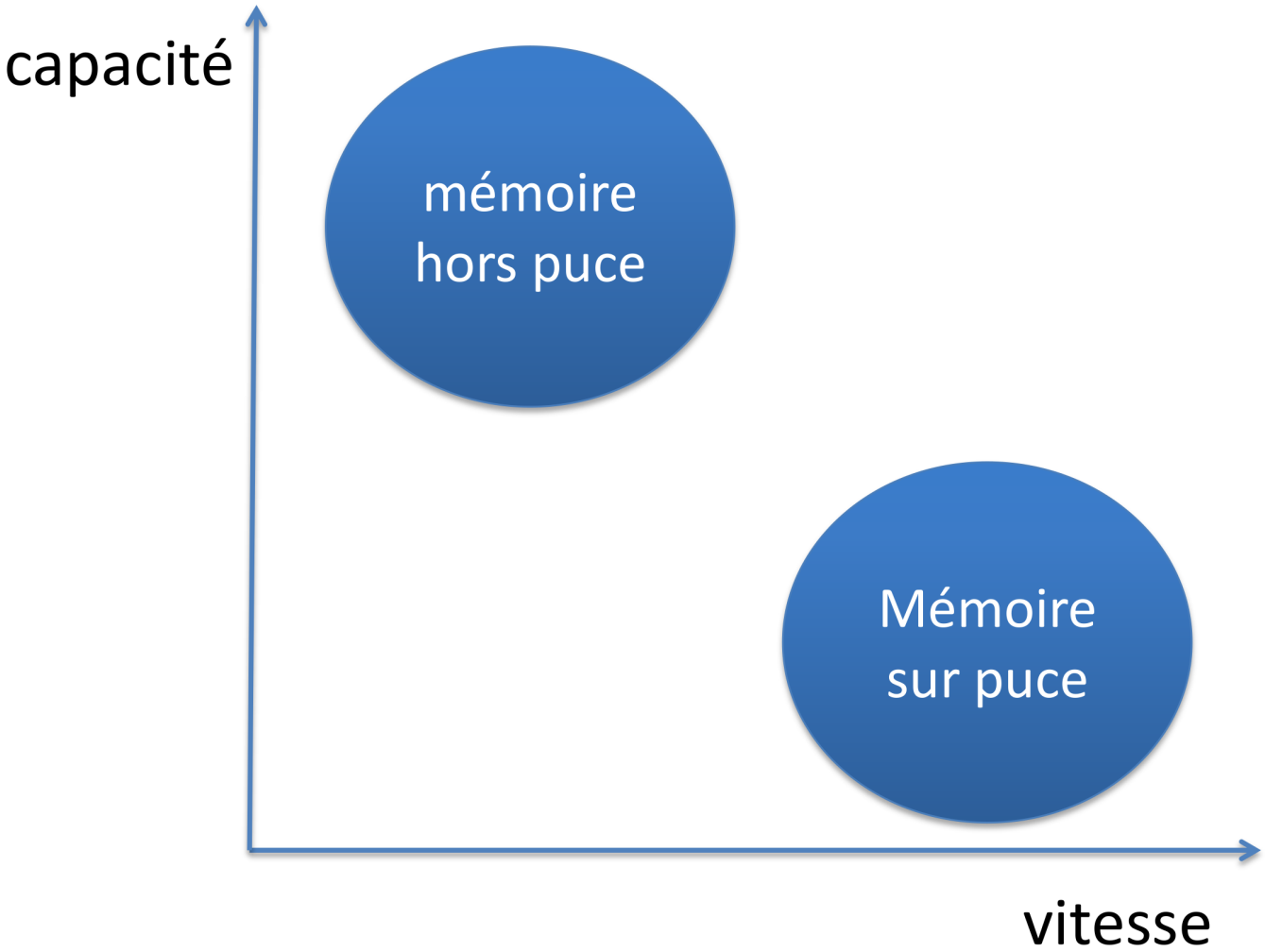
pas en cache



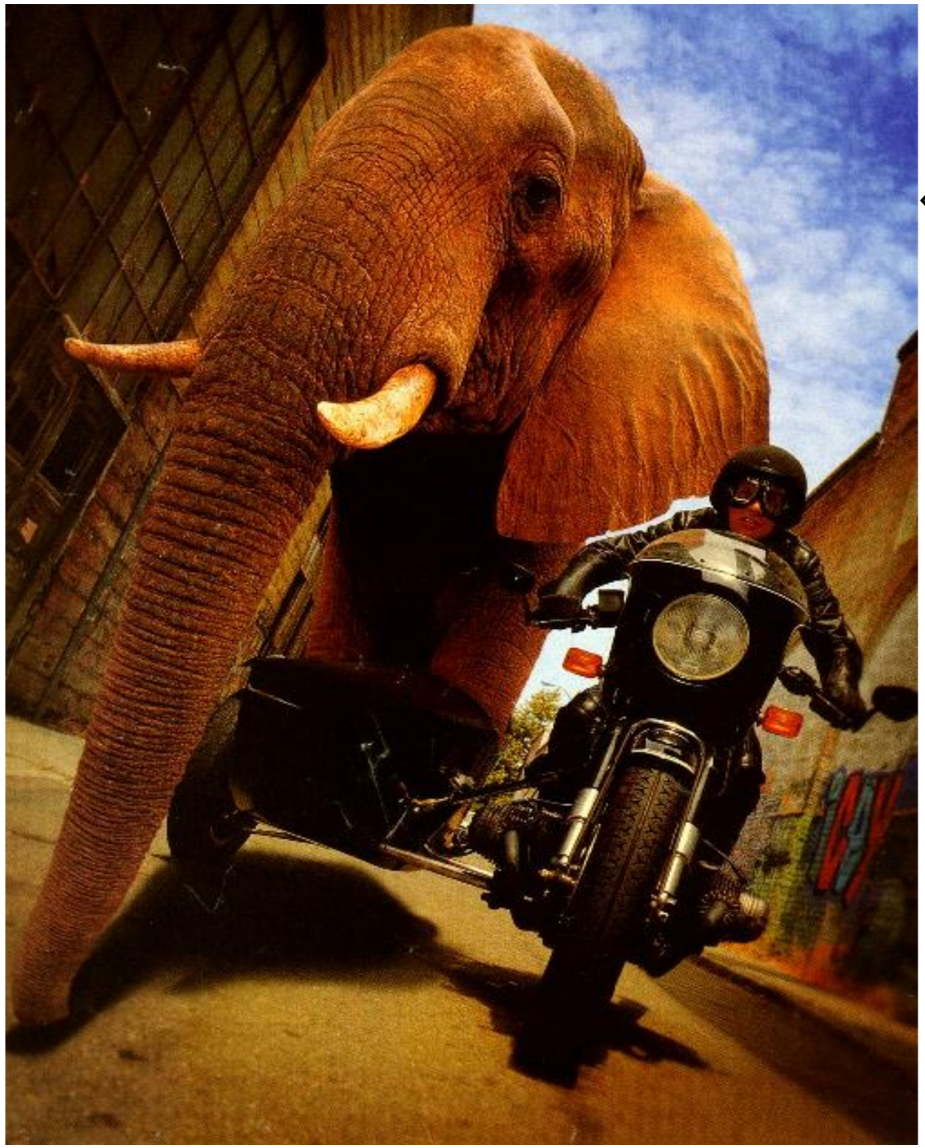
Ordre d'accès passé à la mémoire Centrale (2)

Le processeur attend la la réponse (3)

Deux besoins... ...mais deux contraintes



Analogie



Votre mémoire/stockage
Le plus grand possible

Votre processeur
Le plus rapide possible

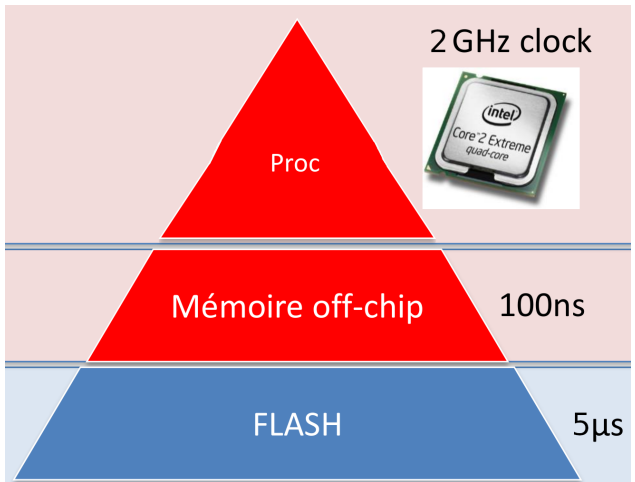
Exemple/Exercice

Q : Votre smartphone a un processeur à 2 GHz (un tic de l'horloge toutes les 0.5ns) et une mémoire Flash de latence minimum de *5 µs*. Combien faut-il de tics d'horloge pour lire un mot de la mémoire ?

R :

Q : Votre smartphone dispose également d'une **mémoire volatile (hors puce / off-chip)** beaucoup plus rapide (latence de *100 ns*). Combien faut-il de tics d'horloge pour lire un mot de la mémoire ?

R :



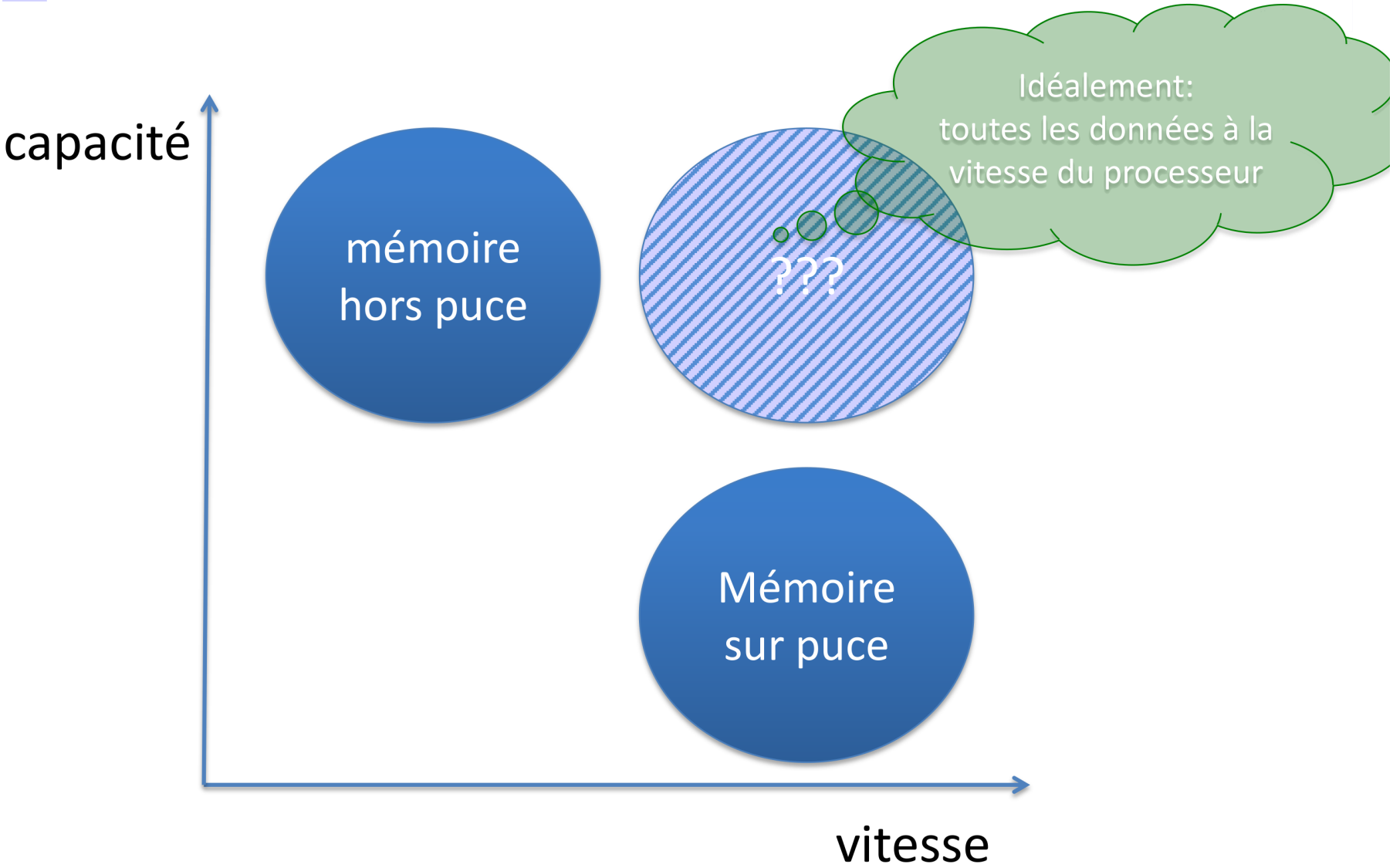
Exemple de la vie courante 1

- **La valise :**
  - Quelques habits bien choisis pour le voyage
  - Usage temporaire
  - Espace limité
  - Proche/Avec vous
- **Dans l'armoire (/à la maison) :**
  - Tous vos habits
  - Pour toute l'année
  - Pas toujours à proximité

Exemple de la vie courante 2 (mais tend à disparaître avec le cloud)

- **Smartphone :**
  - Usage temporaire (quoique...)
  - Espace limité
  - Proche/Avec vous
- **Ordi fixe (/à la maison) :** (quoique...[cloud])
  - Toutes vos données
  - Grande capactié
  - Pas toujours disponible

Création d'une abstration



Mémoire rapide et de grande taille ?

Souhait : mémoire de stockage rapide et de grande taille

- Idéalement, donner au processeur tout ce dont il a besoin en 1 ns (1 GHz)
  P. ex., ne voudriez-vous pas avoir tous vos habits lorsque vous voyagez ?
- Mais toutes les données ne peuvent être gardées proches (**place limitée**)

Solution :

☞ Garder **proches** les données qui seront **vraisemblablement utilisées** :

- Données utilisées *récemment* ☞ localité temporelle
- Données accédées *qui sont liées* ☞ localité spatiale

Comment le réaliser ?

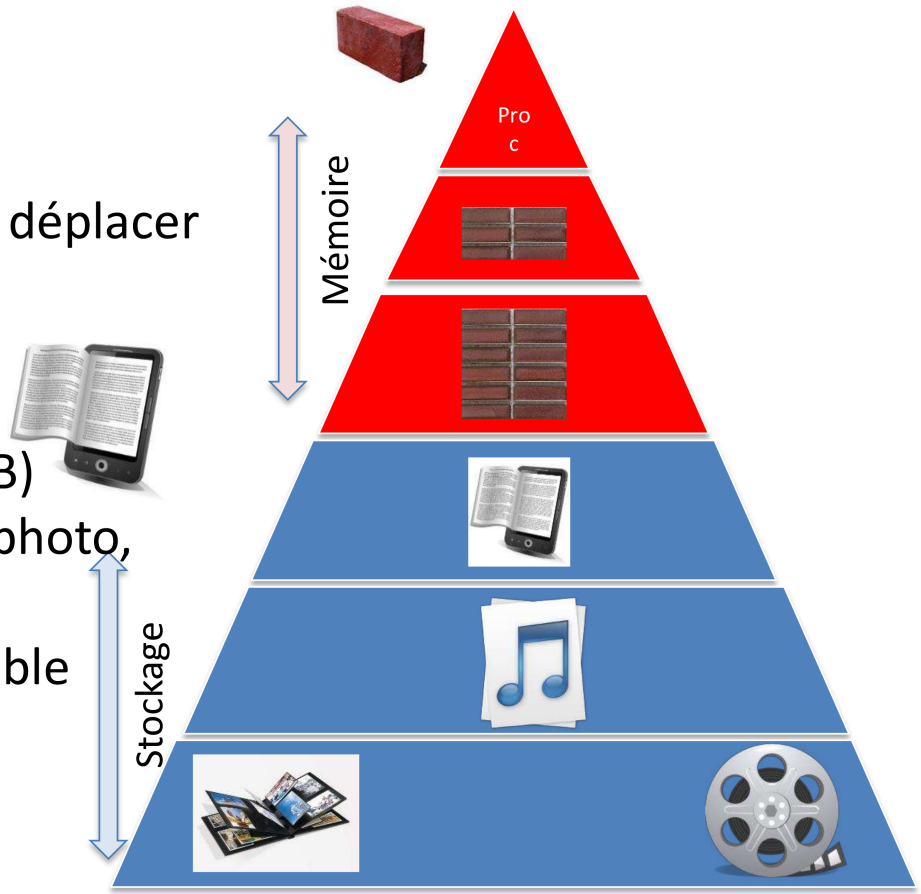
☞ En déplaçant les données par **morceaux/blocs** !

Morceaux de mémoire (4 – 4KB)

- Décomposer les fichiers en morceaux
- Taille fixe
- Immuable
- Plus rapide à déplacer

Fichiers (< GB)

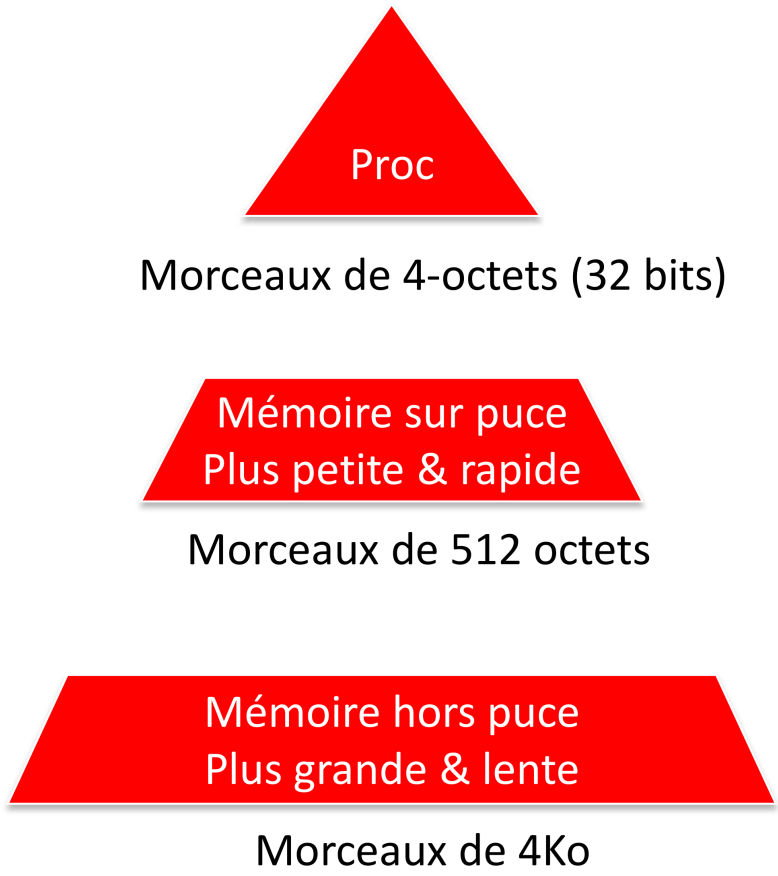
- Musique, photo, film, livre
- Taille variable
- Mutable



Petit et rapide vs. grand et lent

morceaux plus grands : plus de capacité, mais...
morceaux plus grands : plus lents (plus d'informations déplacées par unité de temps)

- ☞ Taille des morceaux adaptée à la vitesse :
- Processeur : les « blocs » s'appelent « **mots** »
  Typiquement : 4 à 8 octets
- Mémoire cache (on-chip) :
  « blocs » entre 1 mot et quelques centaines d'octets
- Mémoire centrale (off-chip) :
  « blocs » de quelques milliers d'octets



Latence et Débit

(**Note :** le terme « *bande passante* » est aussi utilisé pour désigner le débit maximal ; il ne faut pas confondre cette « bande passante » avec celle des signaux !)

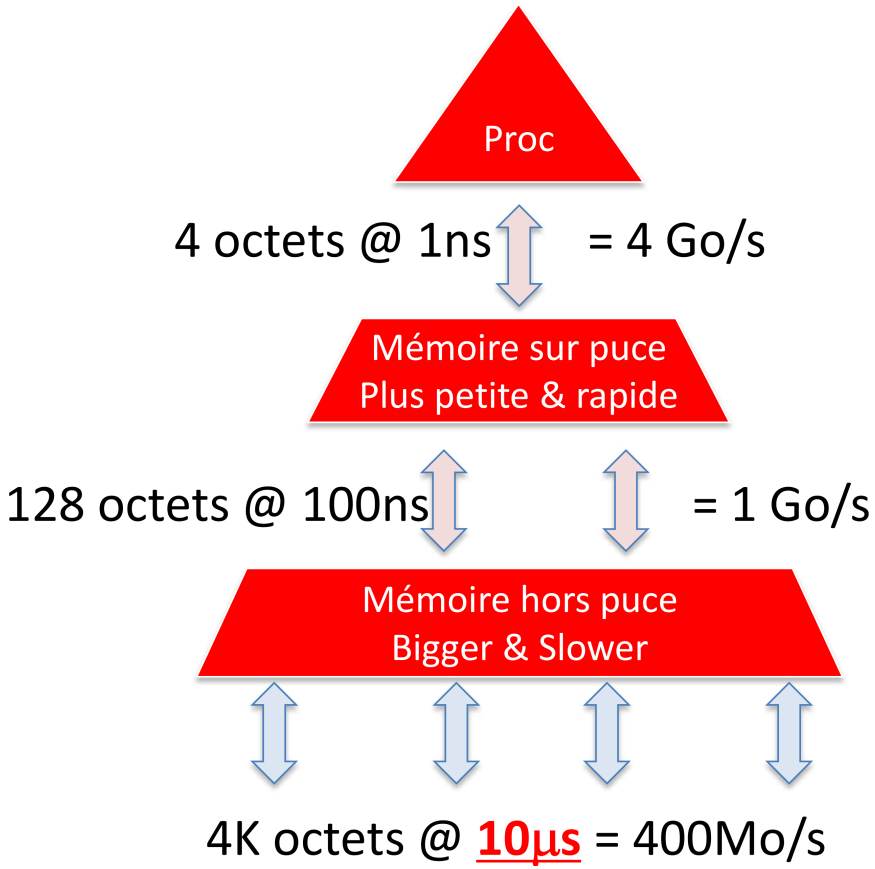
Latences (temps d'accès à 1 « bloc ») :
1 ns .vs. 100 ns .vs. 10 μs

Mais les niveaux inférieurs échangent des « blocs » plus gros !

Qu'en devient-il du débit ?

Débits (nombres d'octets par seconde) :
4 Go/s .vs. 1 Go/s .vs. 0.5 Go/s

☞ En pratique, les tailles de blocs ne compensent pas les différences de latences



Exemple/Exercice

Q : Vous avez un fichier avec un film de 8 Go. Combien de temps cela prendrait-il pour lire le fichier si l'on lisait un seul mot (4 octets) à la fois depuis la mémoire Flash (latence : 5 μs) ?

R :

Q : Votre processeur peut accéder la mémoire Flash avec un débit de 400 Mo/s. Combien de temps est nécéssaire pour transférer le fichier ?

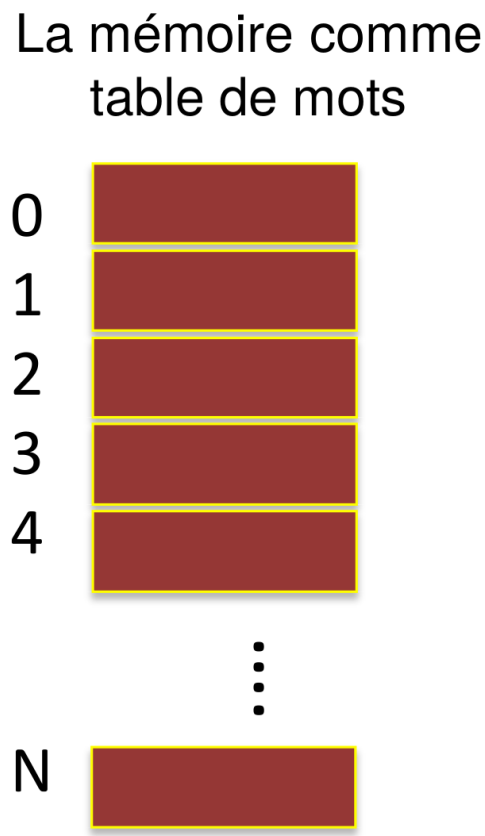
R :

☞ Le débit fait toute la différence !

Mémoire = table de mots

La mémoire est composée de mots.

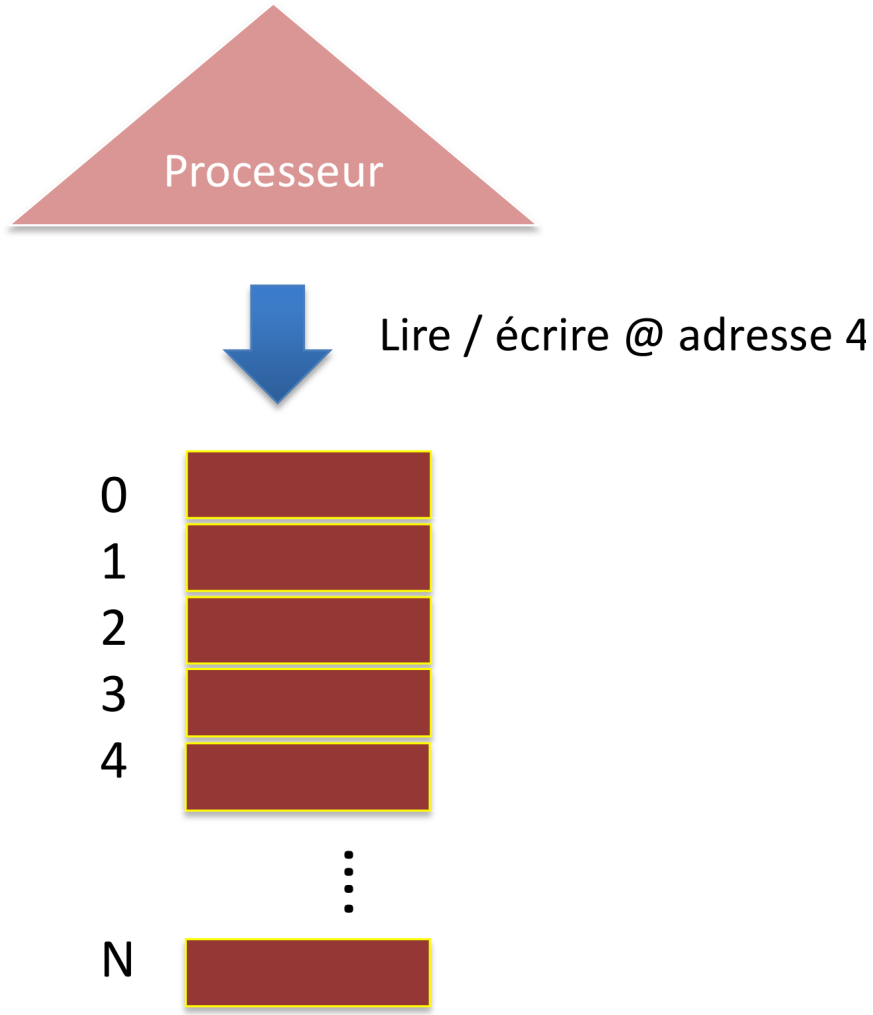
Chaque mot a une adresse.



Accès logique à la mémoire

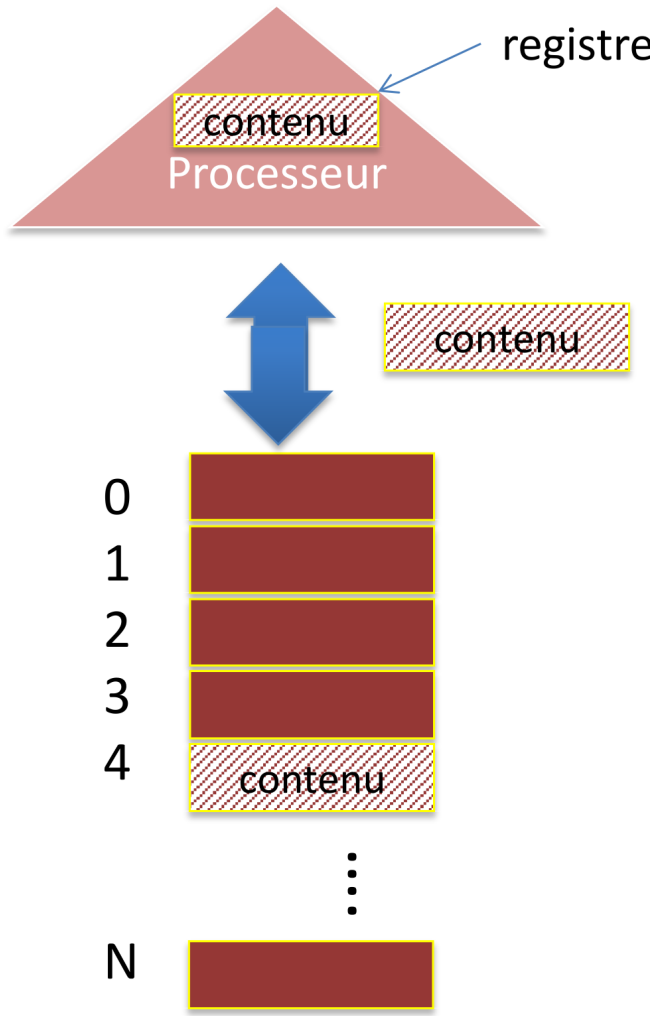
Le processeur donne à la mémoire :

- un ordre de lecture ou d'écriture
- l'adresse concernée
- en cas d'écriture : la valeur à écrire (contenu d'un registre)



Accès logique à la mémoire (2)

La mémoire répond en conséquence

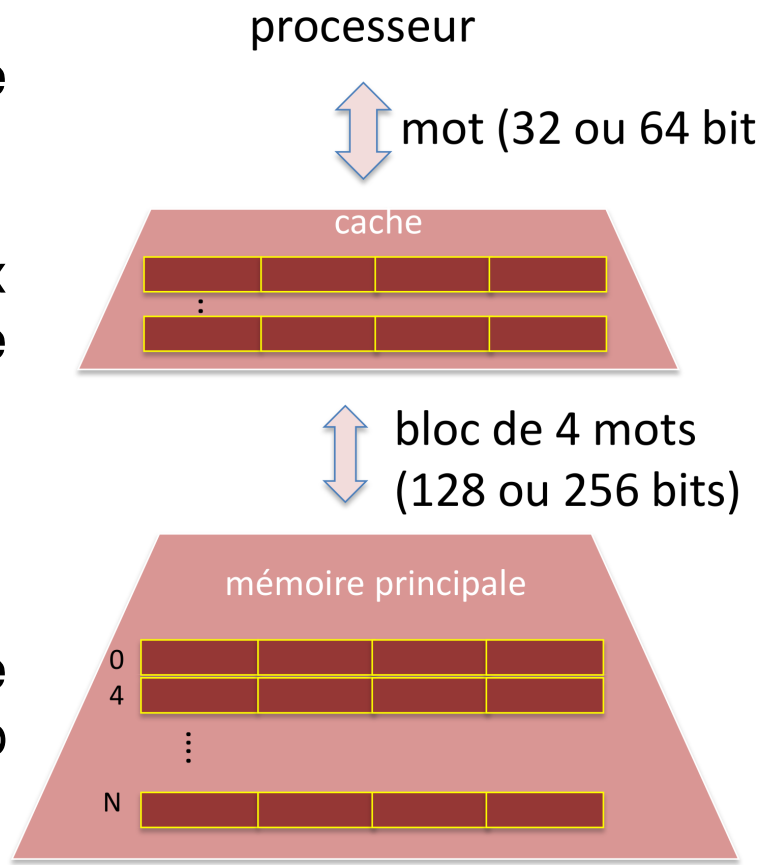


Implémentation physique de la mémoire

Mémoire cache et mémoire centrale sont organisées en blocs.

Le transfert des données entre les deux mémoires se fait par bloc de la mémoire cache.

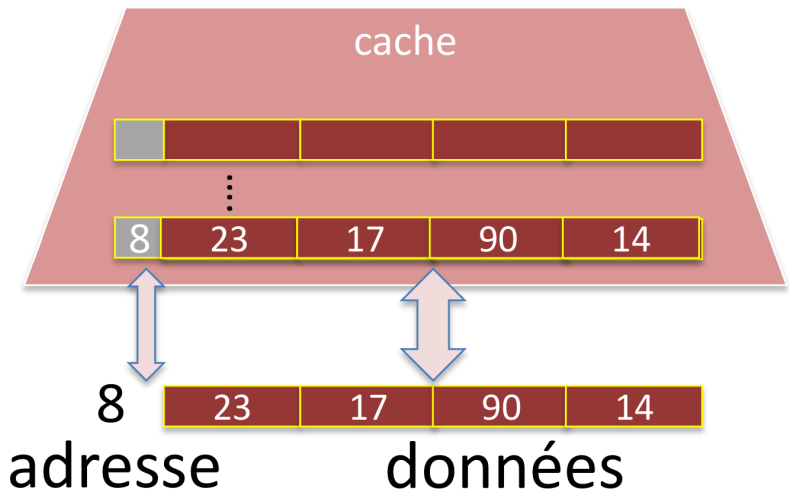
La taille des blocs (échange entre mémoire cache et mémoire centrale) a pour but de réduire la latence globale (échange de plus de mots d'un coup avec la mémoire centrale, plus lente).



Implémentation physique de la mémoire (2)

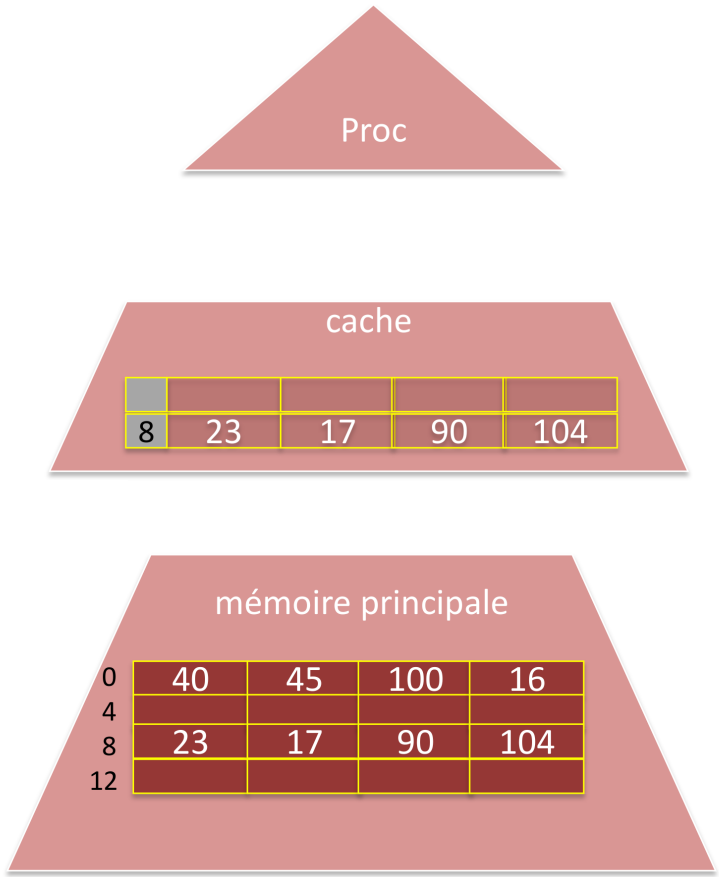
Comment savoir quels blocs de la mémoire centrale se trouvent en cache ?

- ☞ Chaque bloc stocké en mémoire cache garde trace de son adresse correspondante en mémoire centrale



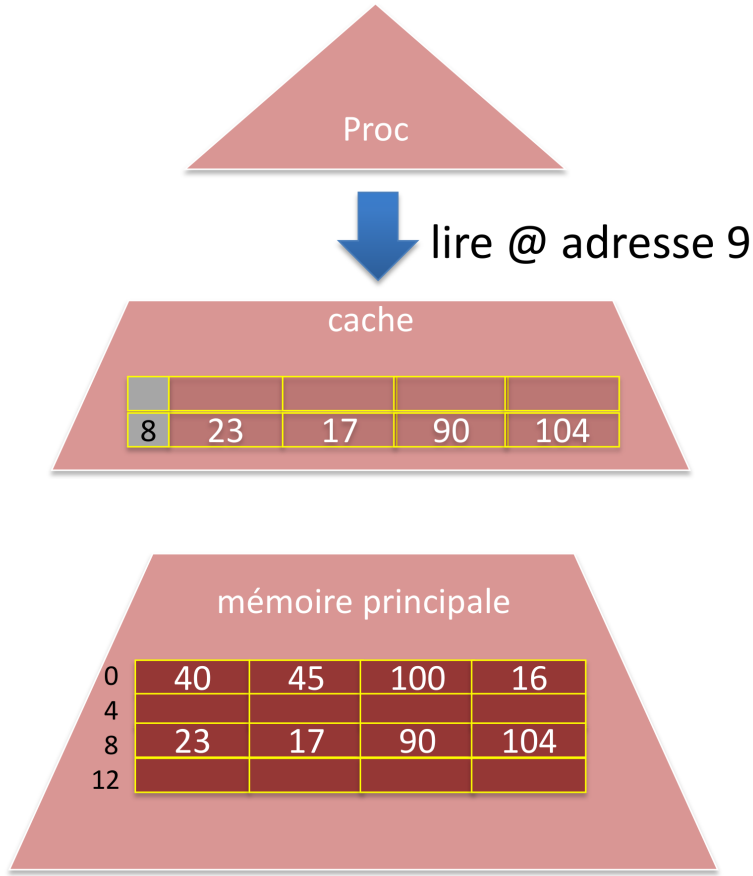
Six questions

- Comment le processeur *lit*-il un mot...
  1. ...quand il est *en* cache ?
  2. ...quand il est *hors* cache ?
- Comment le processeur *écrit*-il un mot...
  3. ...quand il est *en* cache ?
  4. ...quand il est *hors* cache ?
5. Quid lorsque la cache est pleine ?
6. Quid quand le bloc supprimé a été modifié ?



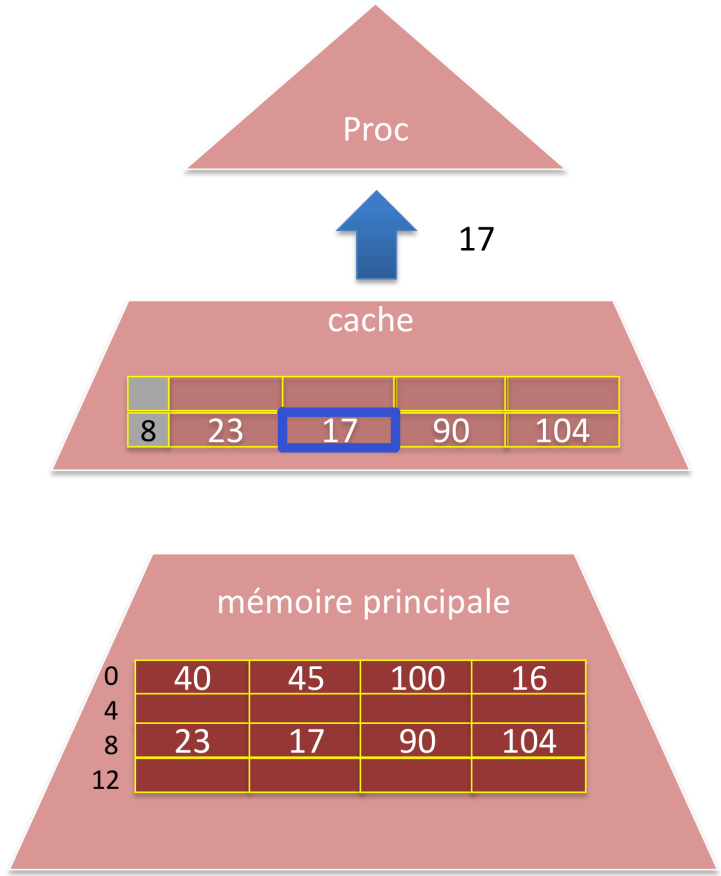
1. Lecture en cache

1. Le processeur envoit l'ordre de lecture



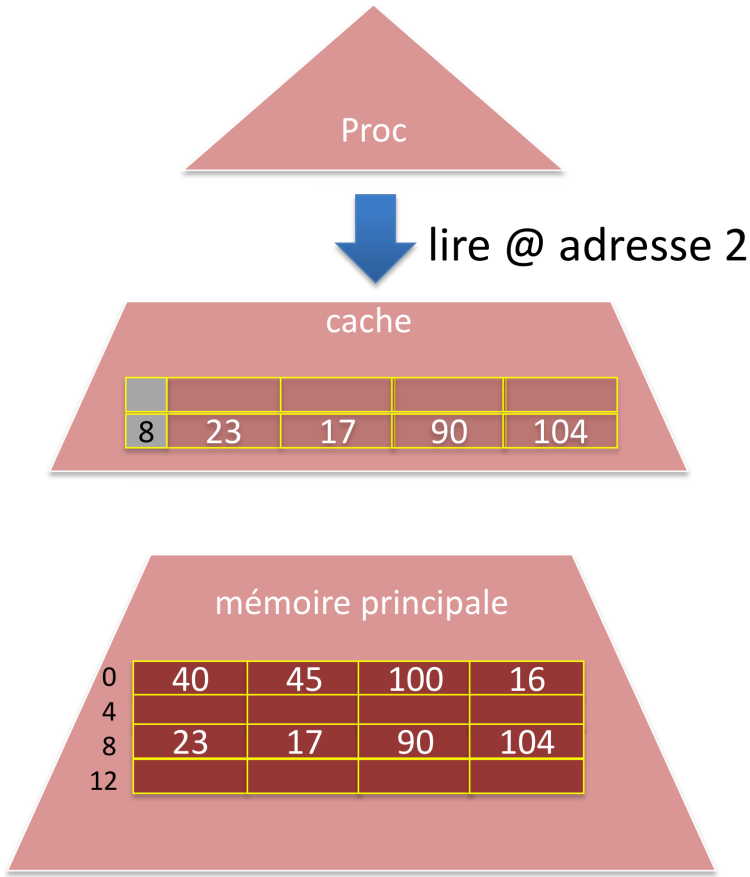
1. Lecture en cache

1. Le processeur envoit l'ordre de lecture
2. La cache vérifie si le mot demandé est présent
3. Et le retourne



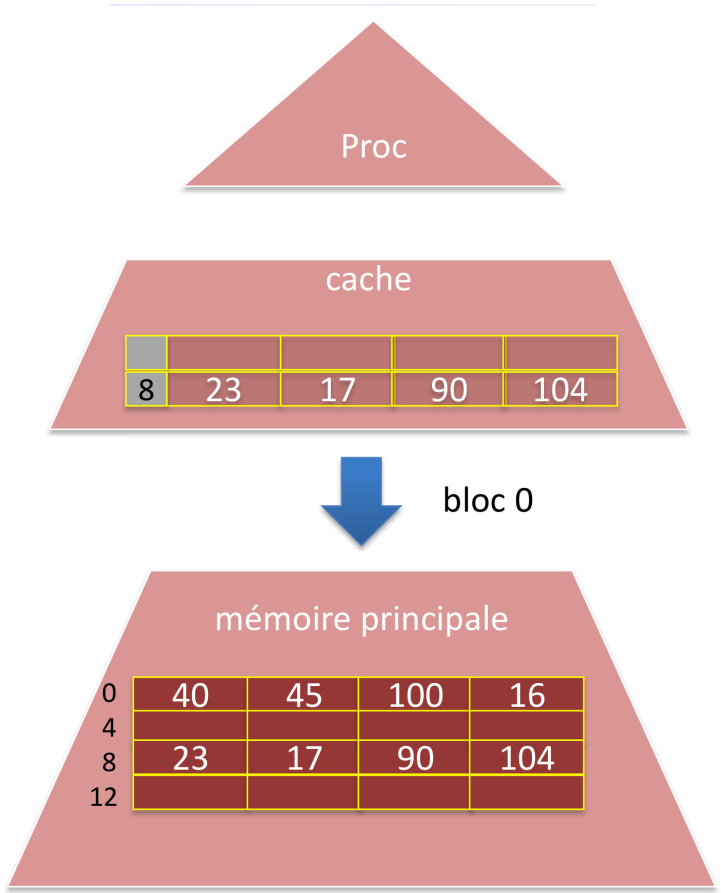
2. Lecture hors cache

1. Le processeur envoit l'ordre de lecture



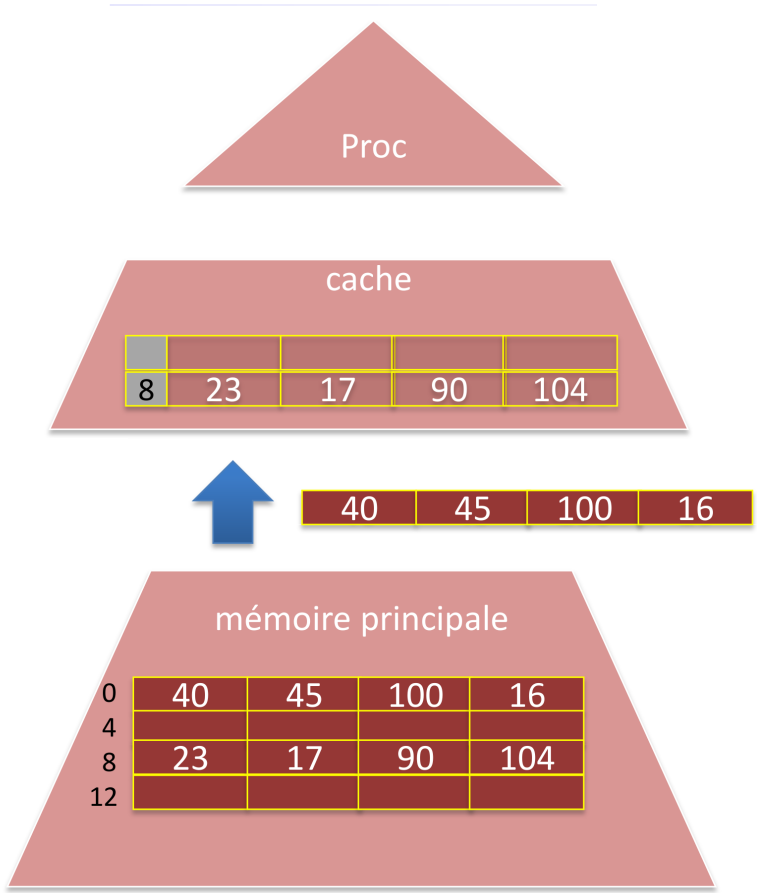
2. Lecture hors cache

1. Le processeur envoit l'ordre de lecture
2. La cache vérifie si le mot demandé est présent
   L'absence du mot cause un **défaut de cache**
3. La cache demande le **bloc** à la mémoire centrale



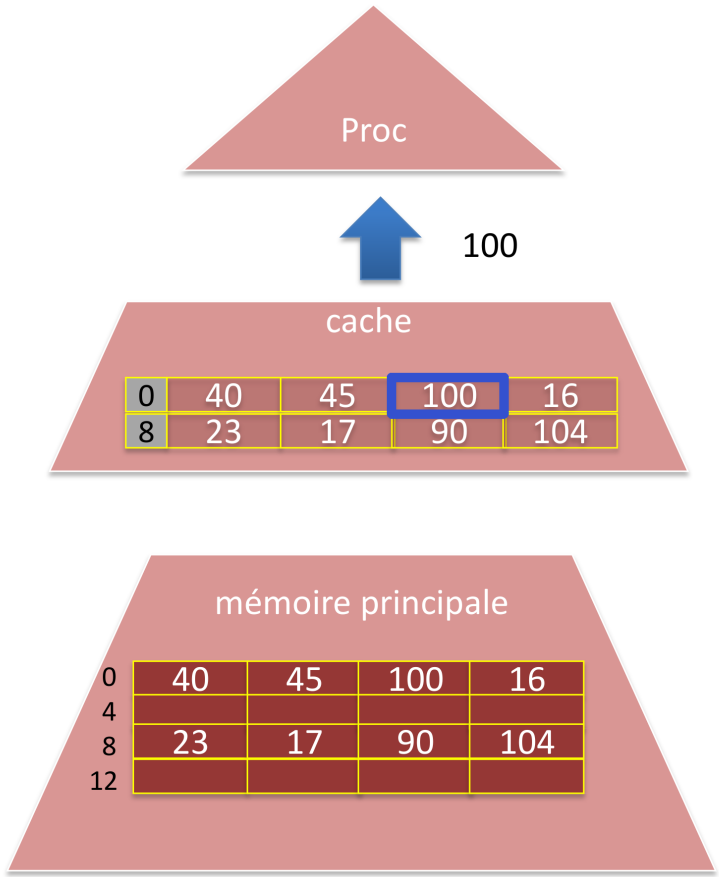
2. Lecture hors cache

1. Le processeur envoit l'ordre de lecture
2. La cache vérifie si le mot demandé est présent
   L'absence du mot cause un **défaut de cache**
3. La cache demande le **bloc** à la mémoire centrale
4. La mémoire centrale retourne le bloc correspondant



2. Lecture hors cache

1. Le processeur envoit l'ordre de lecture
2. La cache vérifie si le mot demandé est présent
   L'absence du mot cause un **défaut de cache**
3. La cache demande le **bloc** à la mémoire centrale
4. La mémoire centrale retourne le bloc correspondant
5. La cache enregistre le bloc (et son *adresse*)
6. La cache retourne le mot demandé au processeur



3. et 4. Ecriture

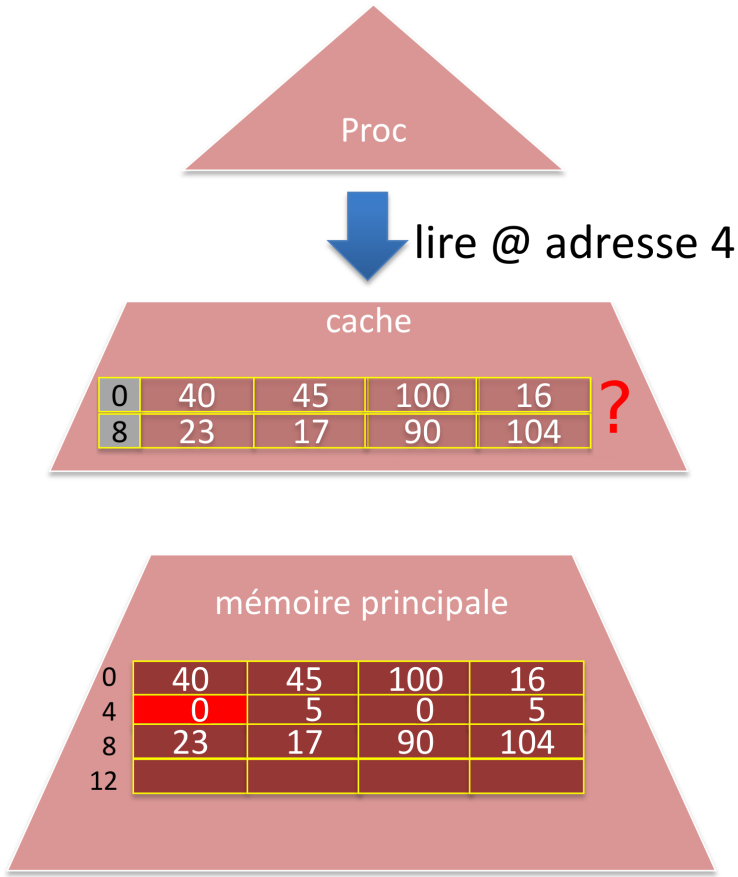
- L'écriture se déroule de façon similaire à la lecture
- La cache retourne à chaque fois une confirmation
- Lorsque l'écriture se fait en cache uniquement
  la cache ne remet pas nécessairement la mémoire centrale à jour
  (pas nécessaire tant que ça reste dans le cache :
  personne d'autre que le processeur n'accède à la mémoire)

5. Cache pleine

1. Le processeur envoit l'ordre de lecture
   La cache vérifie si le mot demandé est présent : *défaut de cache*

   Mais : quel bloc remplacer ? ?

   Il existe plusieurs stratégies de choix. Présentons ici la **LRU : Least Recently Used** : remplace le bloc **utilisé le moins récemment**



5. Cache pleine

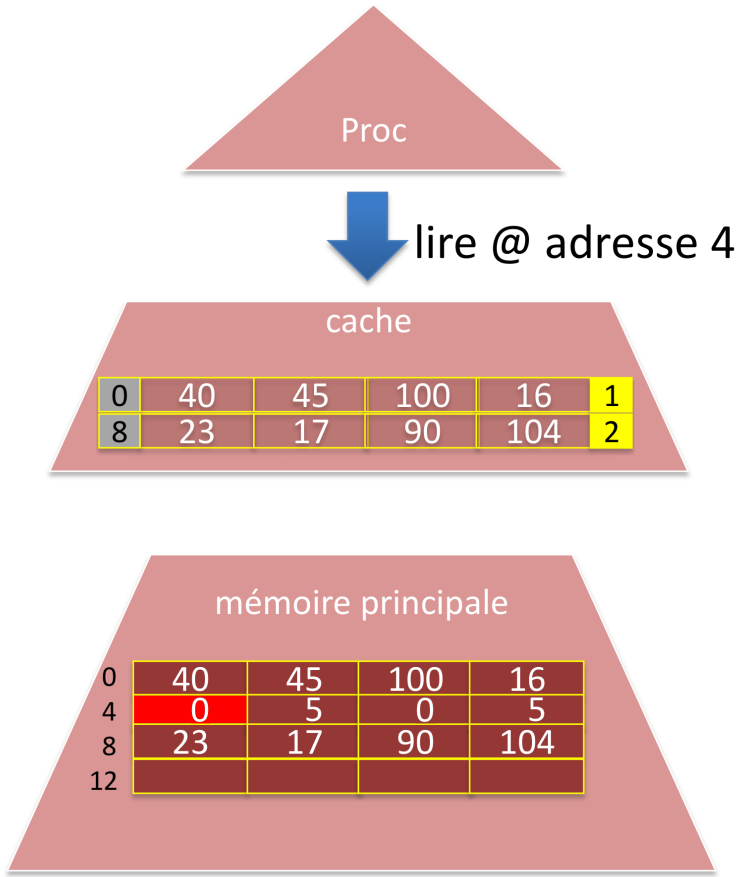
1. Le processeur envoit l'ordre de lecture : *défaut de cache*

   Mais : quel bloc remplacer ? ?

   Il existe plusieurs stratégies de choix. Présentons ici la **LRU : Least Recently Used** : remplace le bloc **utilisé le moins récemment**

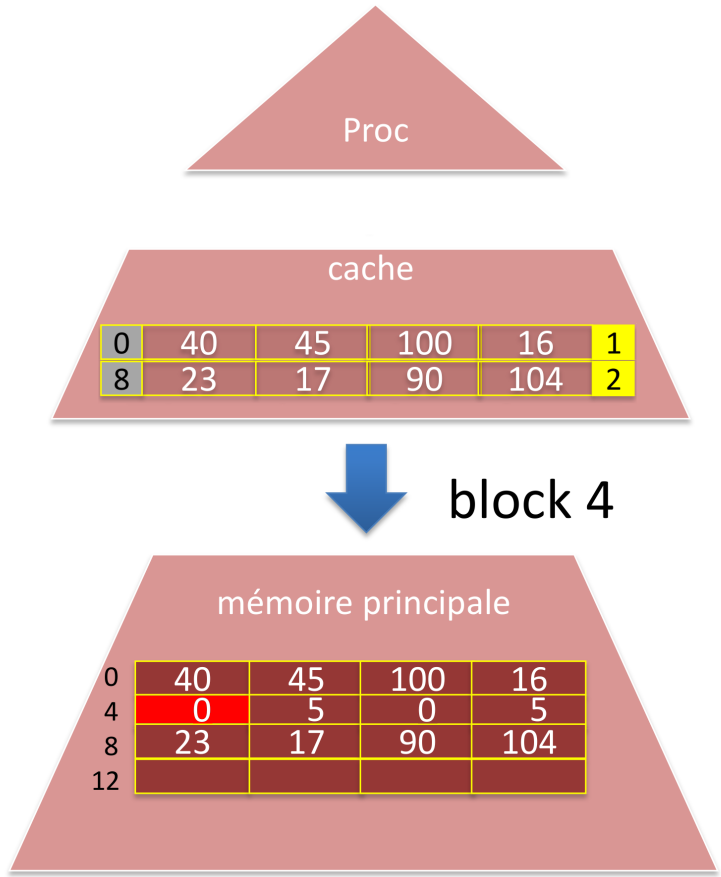
   Pour cela, on ajoute à chaque bloc un compteur.
   Le plus grand compteur indique le bloc le plus anciennement utilisé.
   Les compteurs sont remis à jour à chaque utilisation :

   - $+1$ pour les blocs non utilisés
   - remise à 1 pour le bloc utilisé



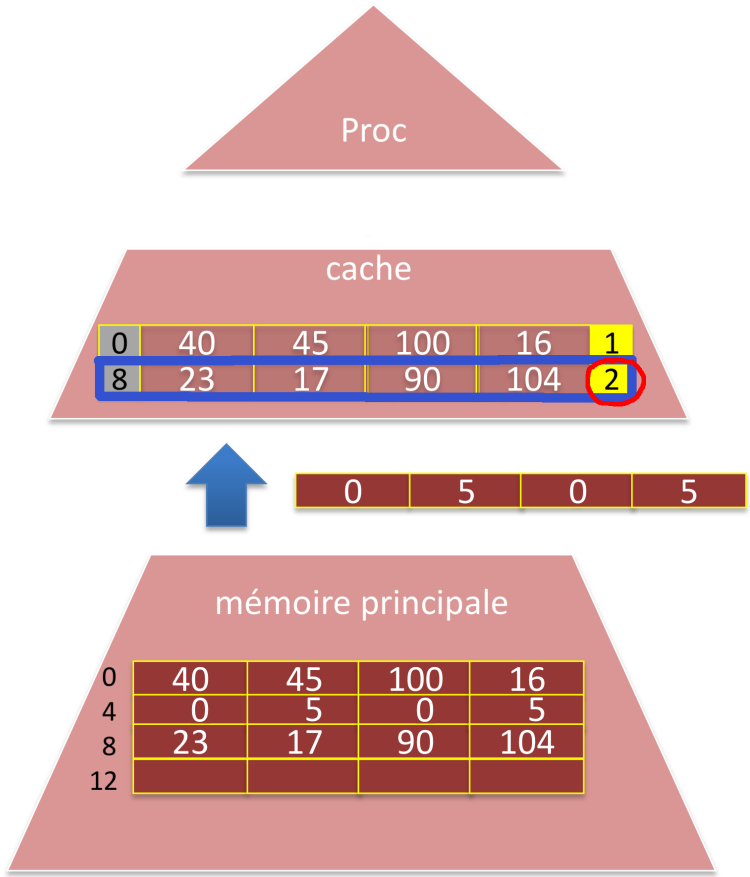
5. Cache pleine

1. Le processeur envoit l'ordre de lecture : *défaut de cache*
2. La cache demande le bloc à la mémoire centrale



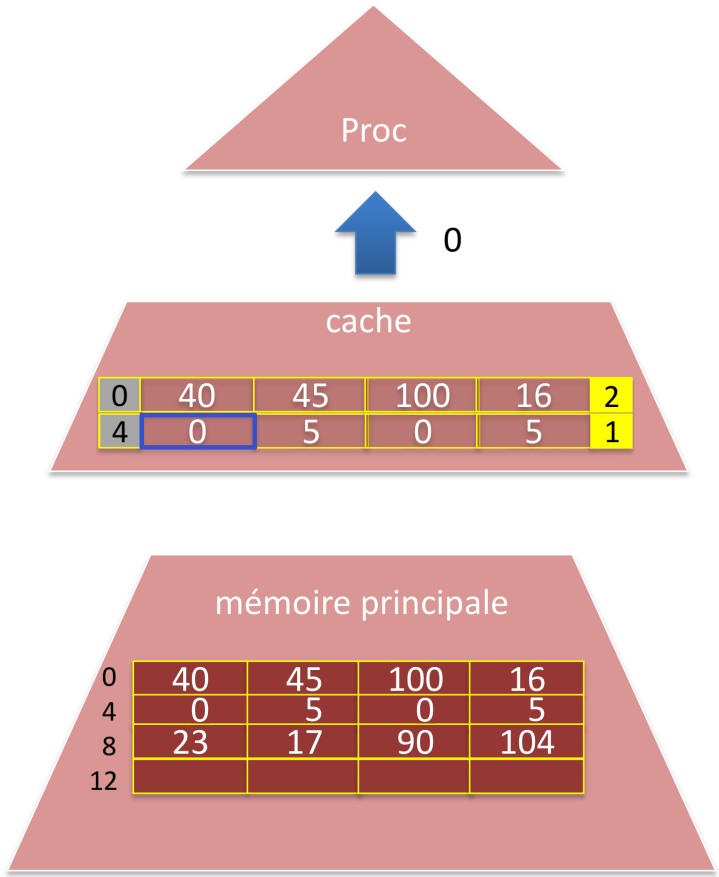
5. Cache pleine

1. Le processeur envoit l'ordre de lecture : *défaut de cache*
2. La cache demande le bloc à la mémoire centrale
3. La mémoire centrale retourne le bloc correspondant
4. Algorithme **LRU** : la cache remplace le bloc **utilisé le moins récemment**
   et remet les compteurs à jour



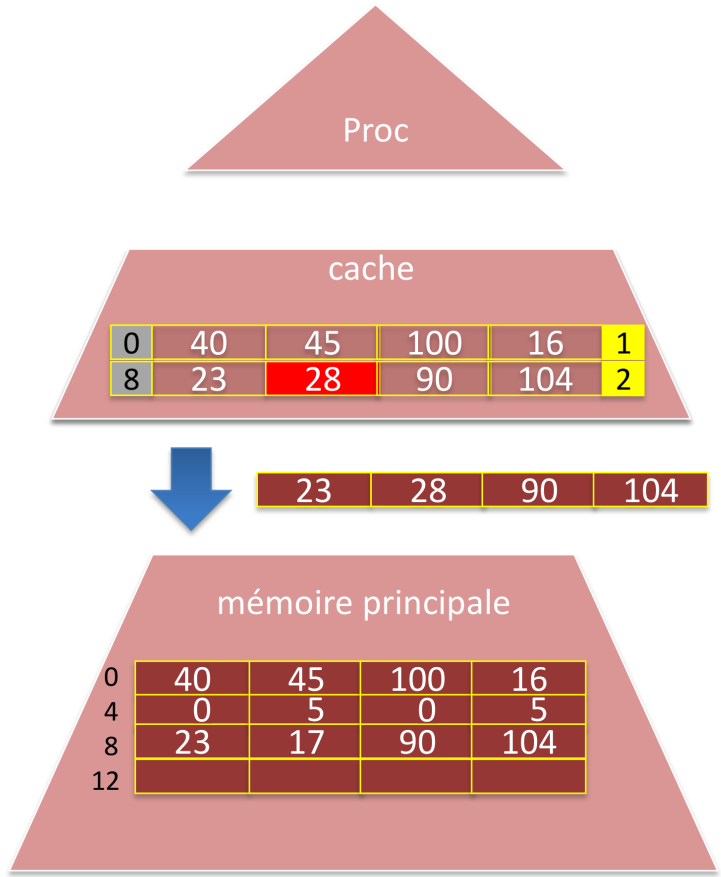
5. Cache pleine

1. Le processeur envoit l'ordre de lecture : *défaut de cache*
2. La cache demande le bloc à la mémoire centrale
3. La mémoire centrale retourne le bloc correspondant
4. Algorithme **LRU** : la cache remplace le bloc **utilisé le moins récemment**
   et remet les compteurs à jour
5. La cache retourne le mot demandé au processeur



6. Et si le bloc remplacé a été modifié ?

avant de demander le nouveau bloc
la cache demande l'écriture du bloc le plus ancien en mémoire centrale



Un exemple compte

Reprennons (à nouveau) l'exemple de la somme des n premiers entiers

somme

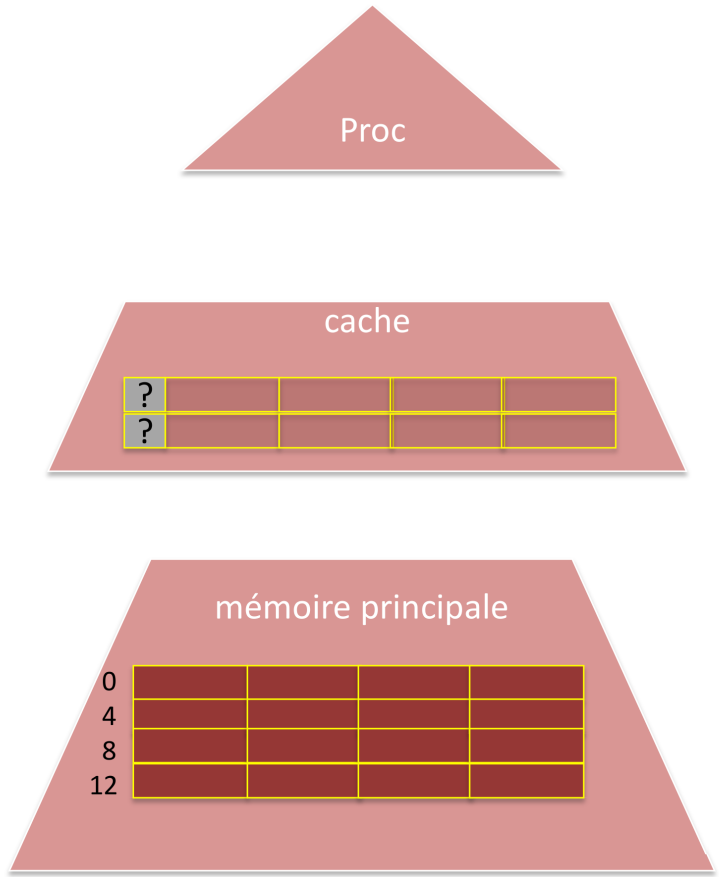
entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s + n$
  $n \leftarrow n - 1$
sortir : s

et concentrons nous sur la boucle

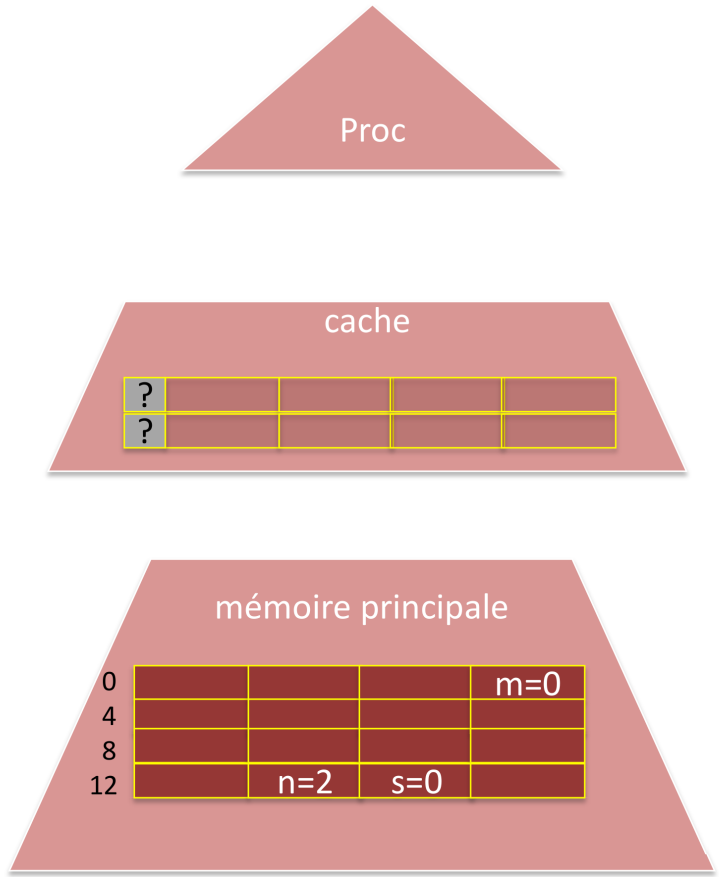
Notre ordinateur hypothétique

- Cache avec 2 blocs
- Blocs de 4 mots
- Mémoire de 4 blocs
- 2 registres dans le processeur
  (et aucune optimisation de mémorisation (caching) de ce coté là)



Emplacement hypothétique en mémoire

- *m* : adresse 3
- *n* : adresse 13
- *s* : adresse 14
- *s* et *m* initialement à 0
- *n* initialement à 2



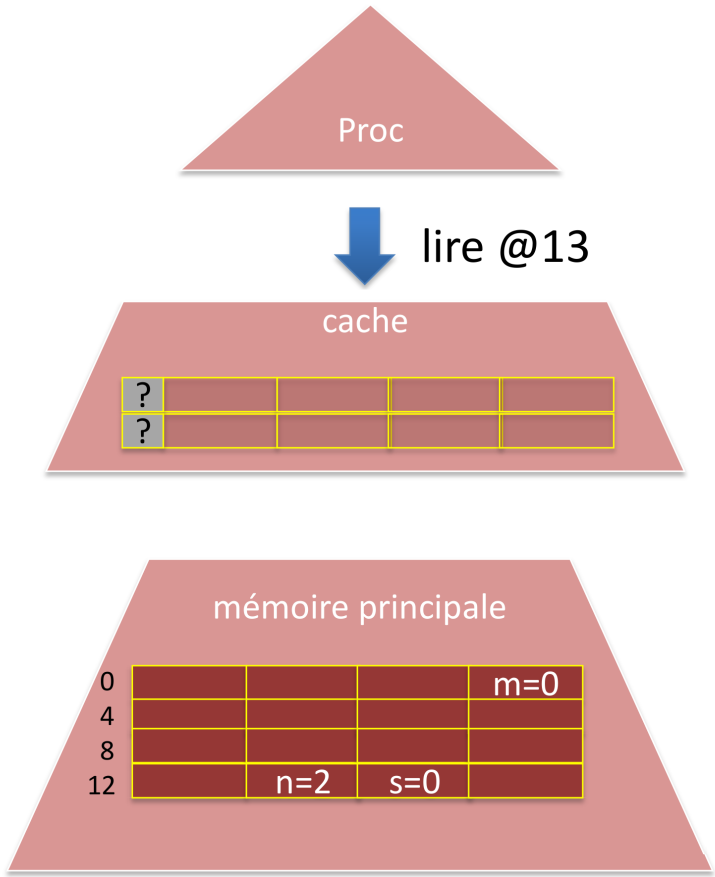
Exécution de la boucle

somme

entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s+n$
  $n \leftarrow n-1$
sortir : s

```
charge r1 @13
```



Défaut(s) de cache : 011
Accès mémoire : 123456

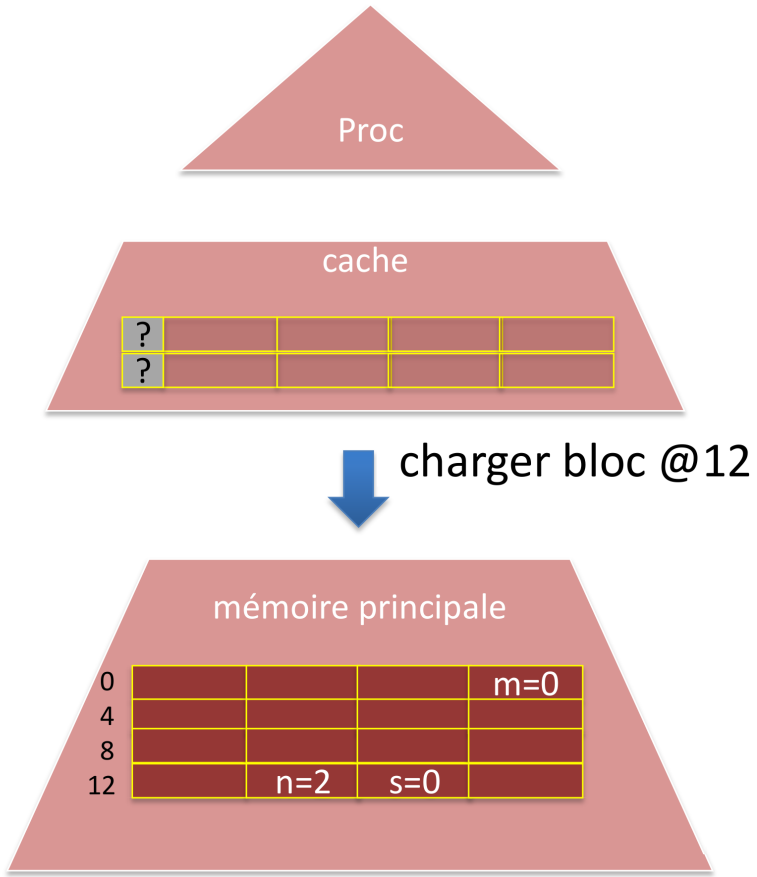
Exécution de la boucle

somme

entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s+n$
  $n \leftarrow n-1$
sortir : s

```
charge r1 @13
```



Défaut(s) de cache : 011
Accès mémoire : 123456

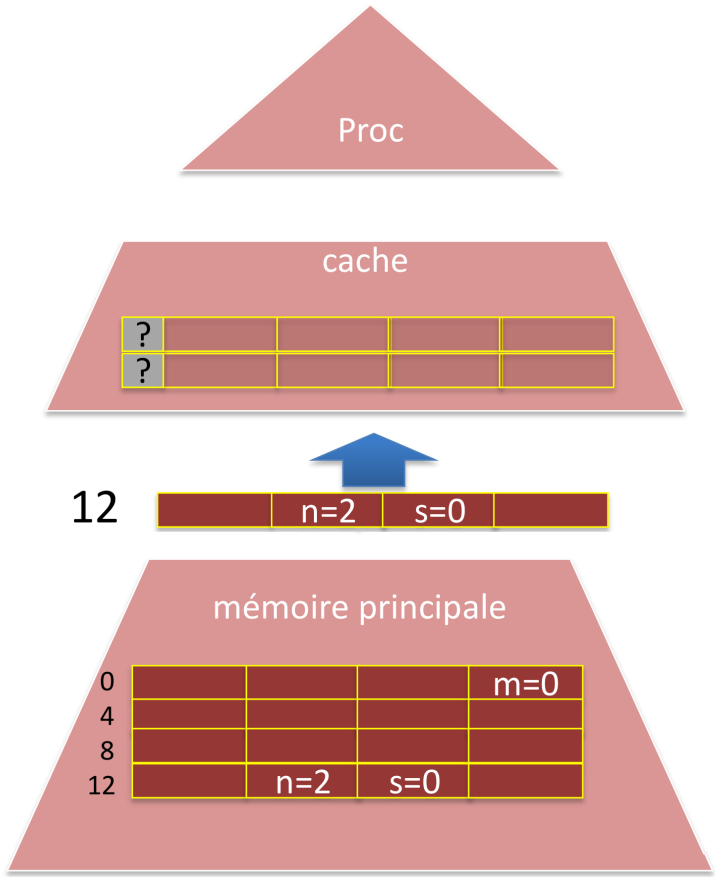
Exécution de la boucle

somme

entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s+n$
  $n \leftarrow n-1$
sortir : s

```
charge r1 @13
```



Défaut(s) de cache : 011
Accès mémoire : 123456

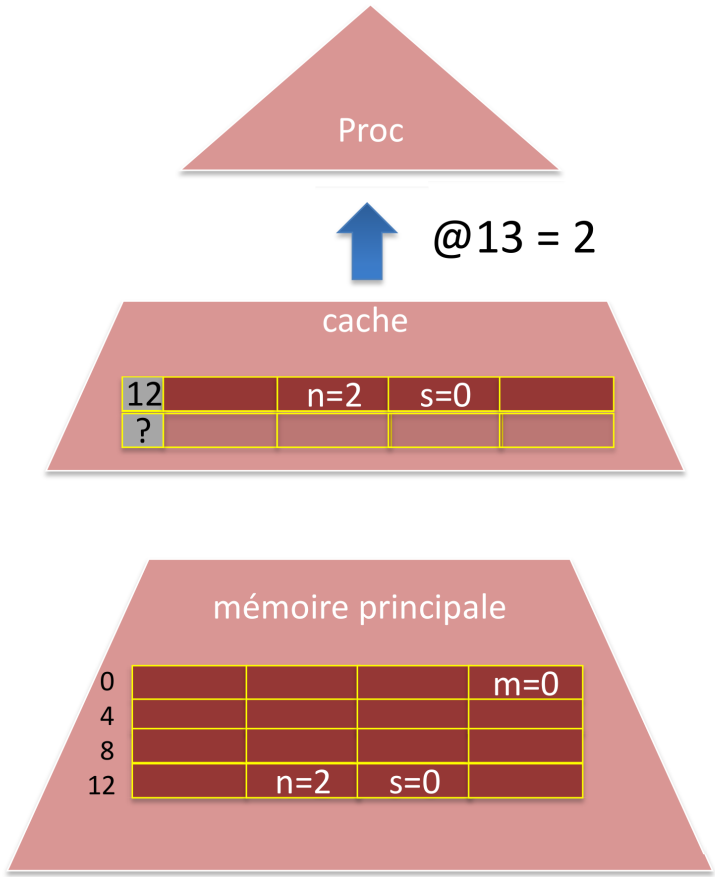
Exécution de la boucle

somme

entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s+n$
  $n \leftarrow n-1$
sortir : s

```
charge r1 @13
cont_ppe r1 0 6
```



Défaut(s) de cache : 011
Accès mémoire : 123456

Exécution de la boucle

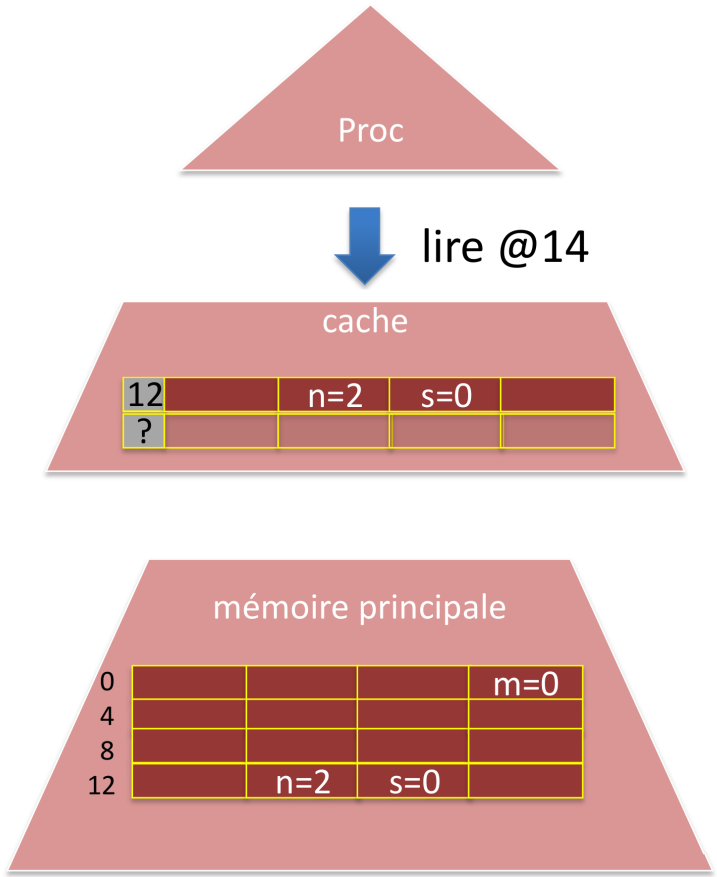
somme

entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s+n$
  $n \leftarrow n-1$
sortir : s

```
charge r1 @13
cont_ppe r1 0 6
charge r1 @14
```

Défaut(s) de cache : 011
Accès mémoire : 123456



Exécution de la boucle

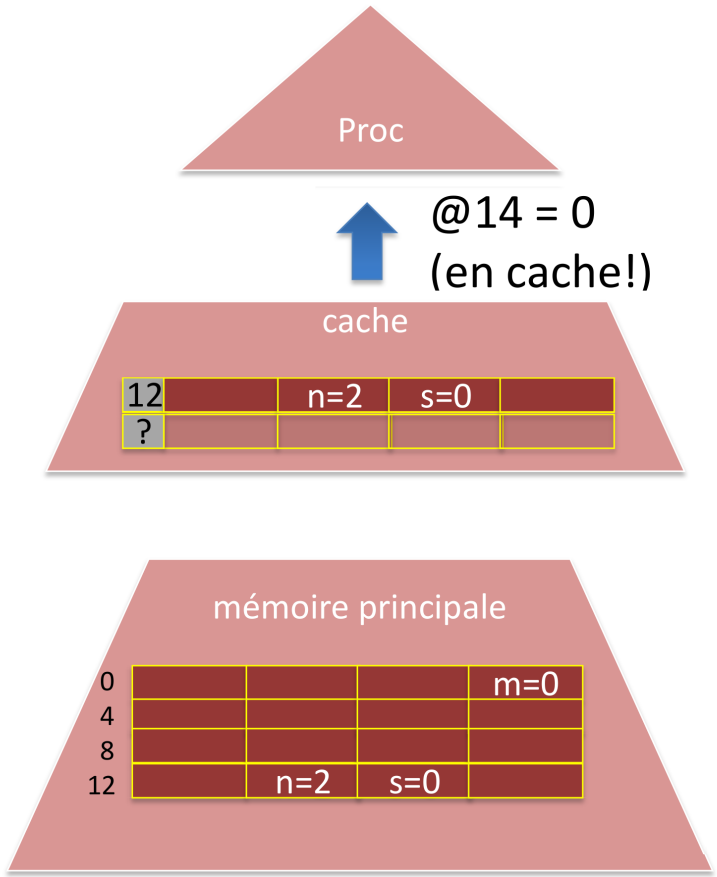
somme

entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s+n$
  $n \leftarrow n-1$
sortir : s

```
charge r1 @13
cont_ppe r1 0 6
charge r1 @14
```

Défaut(s) de cache : 011
Accès mémoire : 123456



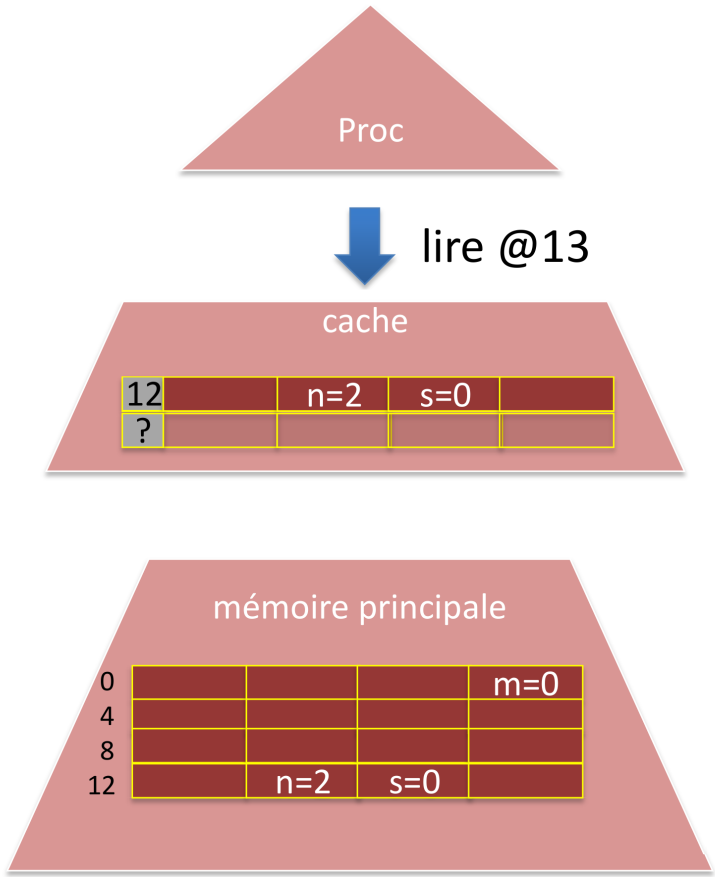
Exécution de la boucle

somme

entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s+n$
  $n \leftarrow n-1$
sortir : s

```
charge r1 @13
cont_ppe r1 0 6
charge r1 @14
charge r2 @13
```



Défaut(s) de cache : 011
Accès mémoire : 123456

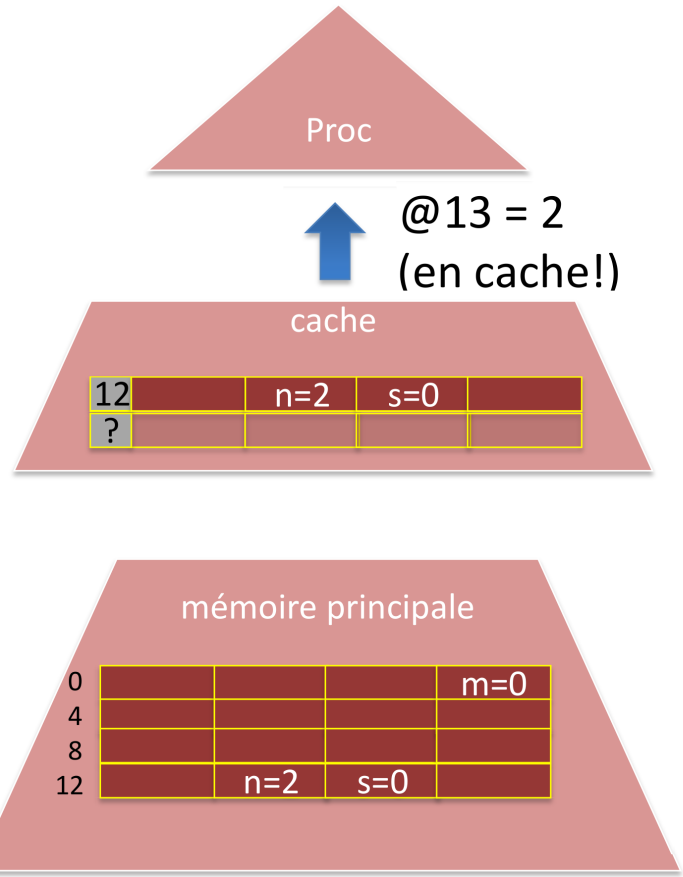
Exécution de la boucle

somme

entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s + n$
  $n \leftarrow n - 1$
sortir : s

```
charge r1 @13
cont_ppe r1 0 6
charge r1 @14
charge r2 @13
somme r1 r1 r2
```



Défaut(s) de cache : 011
Accès mémoire : 123456

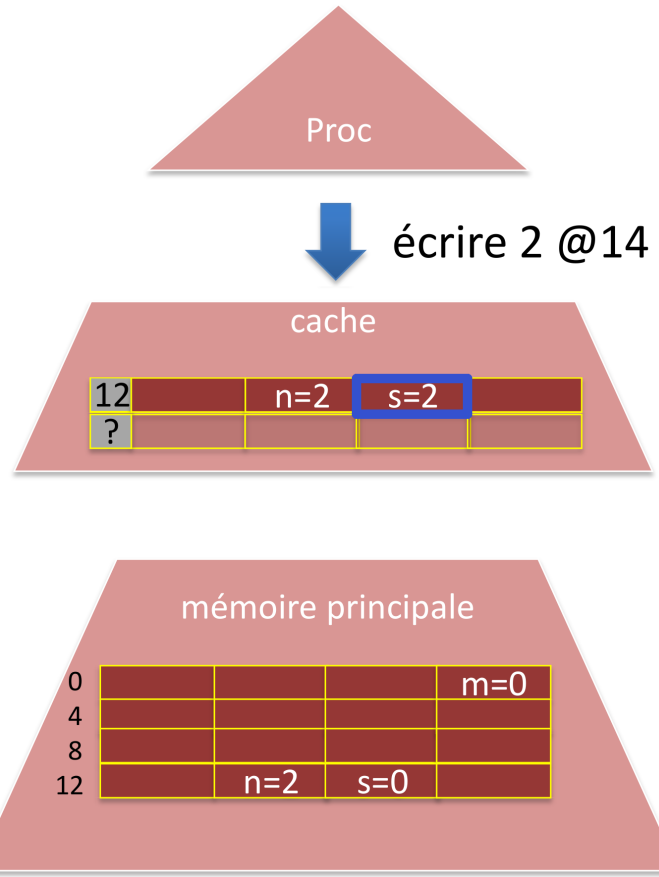
Exécution de la boucle

somme

entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s+n$
  $n \leftarrow n-1$
sortir : s

```
charge r1 @13
cont_ppe r1 0 6
charge r1 @14
charge r2 @13
somme r1 r1 r2
écris r1 @14
```



Défaut(s) de cache : 011
Accès mémoire : 123456

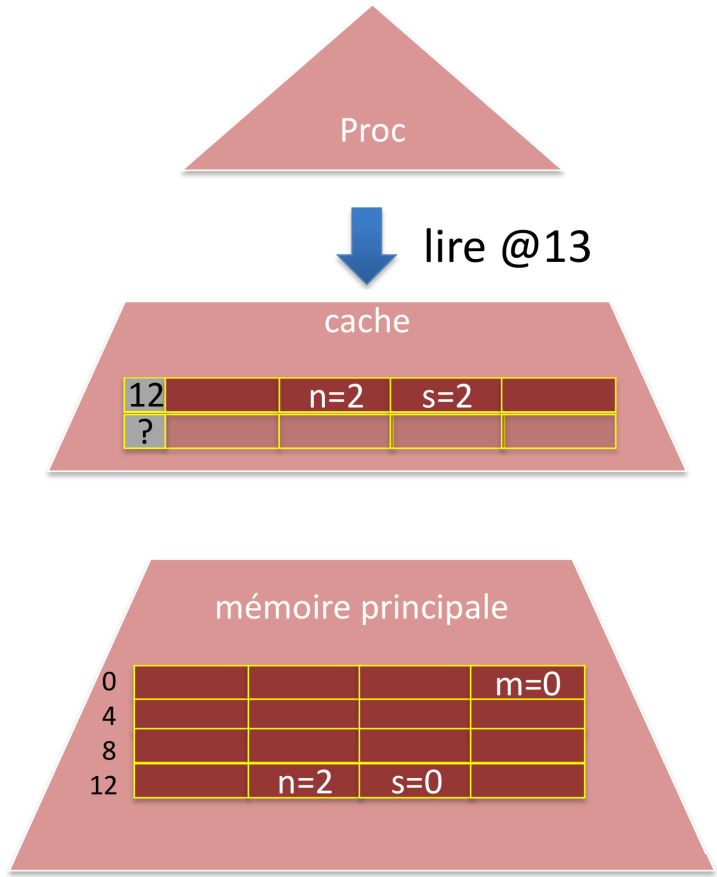
Exécution de la boucle

somme

entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s+n$
  $n \leftarrow n-1$
sortir : s

```
charge r1 @13
cont_ppe r1 0 6
charge r1 @14
charge r2 @13
somme r1 r1 r2
écris r1 @14
charge r1 @13
```



Défaut(s) de cache : 011
Accès mémoire : 123456

Exécution de la boucle

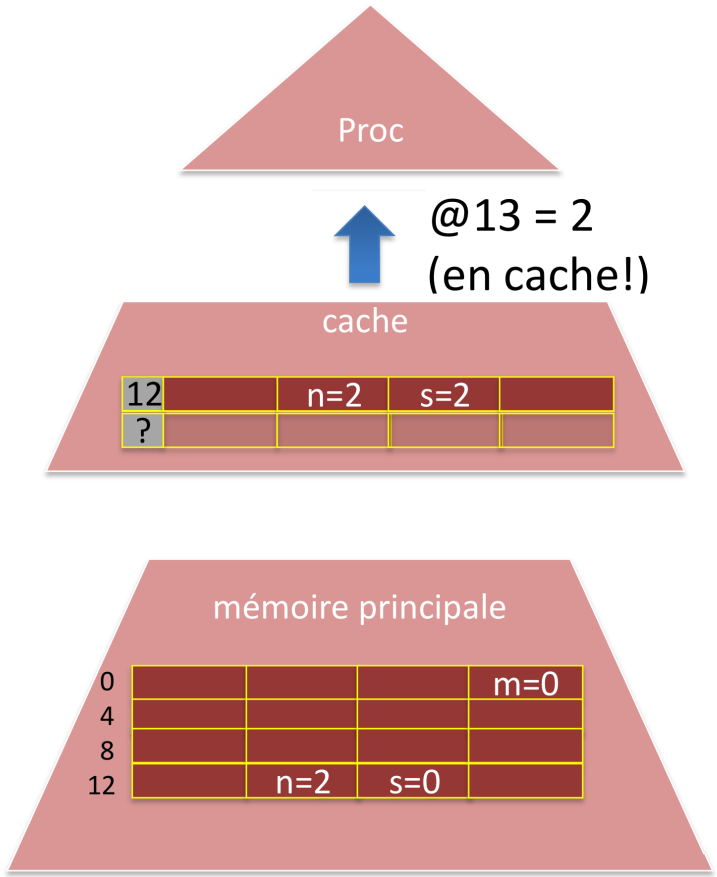
somme

entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s+n$
  $n \leftarrow n-1$
sortir : s

```
charge r1 @13
cont_ppe r1 0 6
charge r1 @14
charge r2 @13
somme r1 r1 r2
écris r1 @14
charge r1 @13
somme r1 r1 −1
```

Défaut(s) de cache : 011
Accès mémoire : 123456



Exécution de la boucle

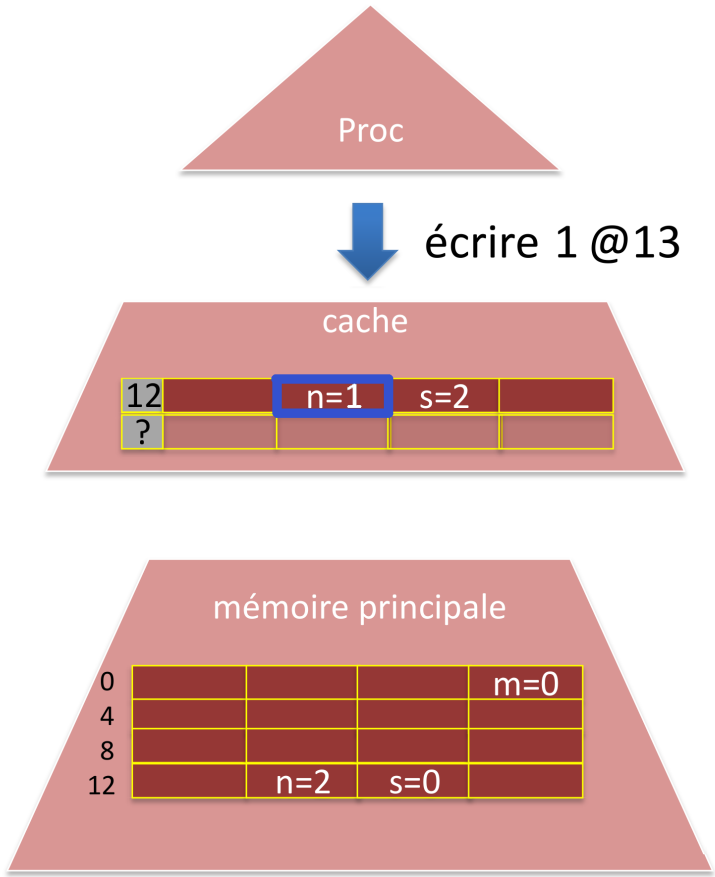
somme

entrée : n
sortie : $S(n)$

$s \leftarrow 0$
Tant que $n > 0$
  $s \leftarrow s+n$
  $n \leftarrow n-1$
sortir : s

```
charge r1 @13
cont_ppe r1 0 6
charge r1 @14
charge r2 @13
somme r1 r1 r2
écris r1 @14
charge r1 @13
somme r1 r1 −1
écris r1 @13
```

Défaut(s) de cache : 011
Accès mémoire : 123456



Temps d'accès au total

- première itération :
  5 accès au cache,
  1 accès mémoire centrale (1 défaut de cache)
- toutes les autres itérations :
  6 accès au cache

Au total :

- $6(n-1)+5$ accès au cache (à 1 ns/accès)
- 1 accès à la mémoire centrale (à 100 ns)

☞ $(6n+99)$ ns au total
au lieu de $600n$ ns si l'on n'avait pas de cache !

Pourquoi la cache aide-t-elle ?

Il y a deux cas où la cache est utile :

- ré-accès à une même adresse dans un cours laps de temps
  ☞ **localité temporelle**

  Exemple : *n* plusieurs fois de suite dans l'exemple précédent

- ré-accès à des adresses dans un même bloc
  ☞ **localité spatiale**

  Exemple : *n* et *s* dans l'exemple précédent

Localité temporelle

En cache parce qu'il y a eu des accès à
des **adresses identiques rapprochés dans le temps**.

☞ réutilisation d'un mot du cache *déjà utilisé*

Analogie : pendant une semaine à la neige, on réutilise son snowboard tous les jours.

Réaliste ?

Tous les algorithmes « intéressants » comportent des boucles accédant aux mêmes variables

Localité spatiale

En cache parce qu'il y a eu des accès à
des **adresses différentes mais proche dans l'espace**

- ☞ utilisation d'un mot présent dans le cache en raison d'un accès antérieur à un *autre mot du même bloc*

Analogie : pendant une semaine à la neige, on a besoin de ses ski *et* de ses chaussures *et* de sa combinaison *et* de ses gants, etc.

Réaliste ?

Tous les algorithmes « intéressants » travaillent avec une série de variables liées entre elles

Au compilateur de les organiser assez proche en mémoire

Localité spatiale/temporelle : illustration

Considérons deux situations extrêmes :

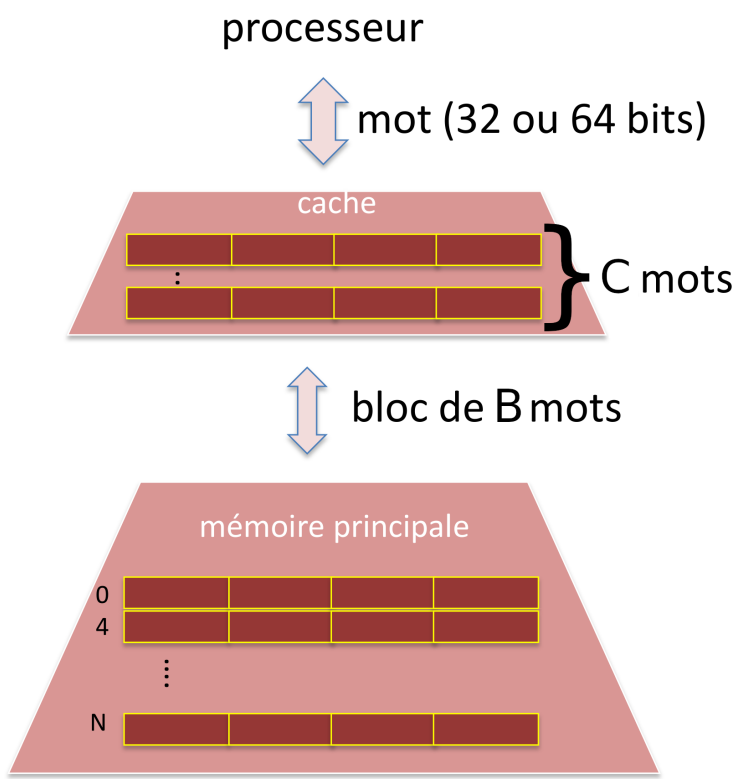
- un programme **P1**
  - qui utilise peu de variables
  - lesquelles sont stockées proches les unes des autres (forte localité **spatiale**)
  - mais sont utilisées une fois de temps en temps (faible localité temporelle)

- un programme **P2**
  - qui utilise tellement de variables qu'elles sont éparpillées ici et là (elles ne peuvent pas tenir dans un même bloc)
  - qui utilise 2 de ces variables, n et m, situés de part et d'autre de la mémoire
  - mais qui fait toujours l'opération $n + m$
  - ☞ n et m ont alors une grande localité **temporelle** (toujours utilisées en même temps), mais pas de localité spatiale.

Localité spatiale/temporelle : illustration

Considérons trois architectures de cache.

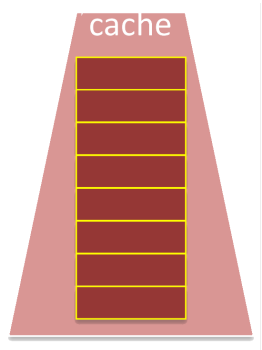
Soient *C* la taille de la cache,
et *B* la taille d'un bloc (mesurées en mots ici)



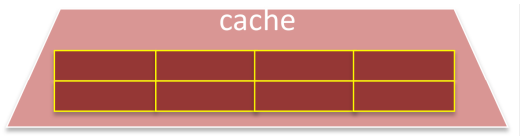
Localité spatiale/temporelle : illustration

Considérons trois architectures de cache.

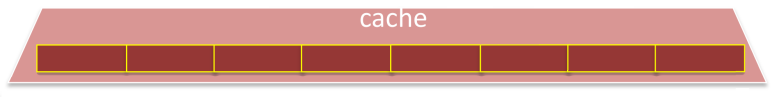
Soient C la taille de la cache,
et B la taille d'un bloc (mesurées en mots ici)



$B = 1$



$B = C/2$



$B = C$

Impact de la taille des blocs : blocs plus petits

À un extrême, on peut construire un ordinateur tel que $B = 1$.

Sur P1, il ferait plein de défauts de cache
(en tout cas au début pour charger mot à mot toute cette mémoire) ;

alors que sur P2, il serait efficace (typiquement 2 défauts de cache, uniquement sur n et m).

☞ **MOINS de localité spatiale**
(charge moins de variables différentes à chaque fois)

Impact de la taille des blocs : blocs plus grands

À l'autre extrême, on peut construire un autre ordinateur tel que $B = C$ (la cache n'a qu'un seul bloc)

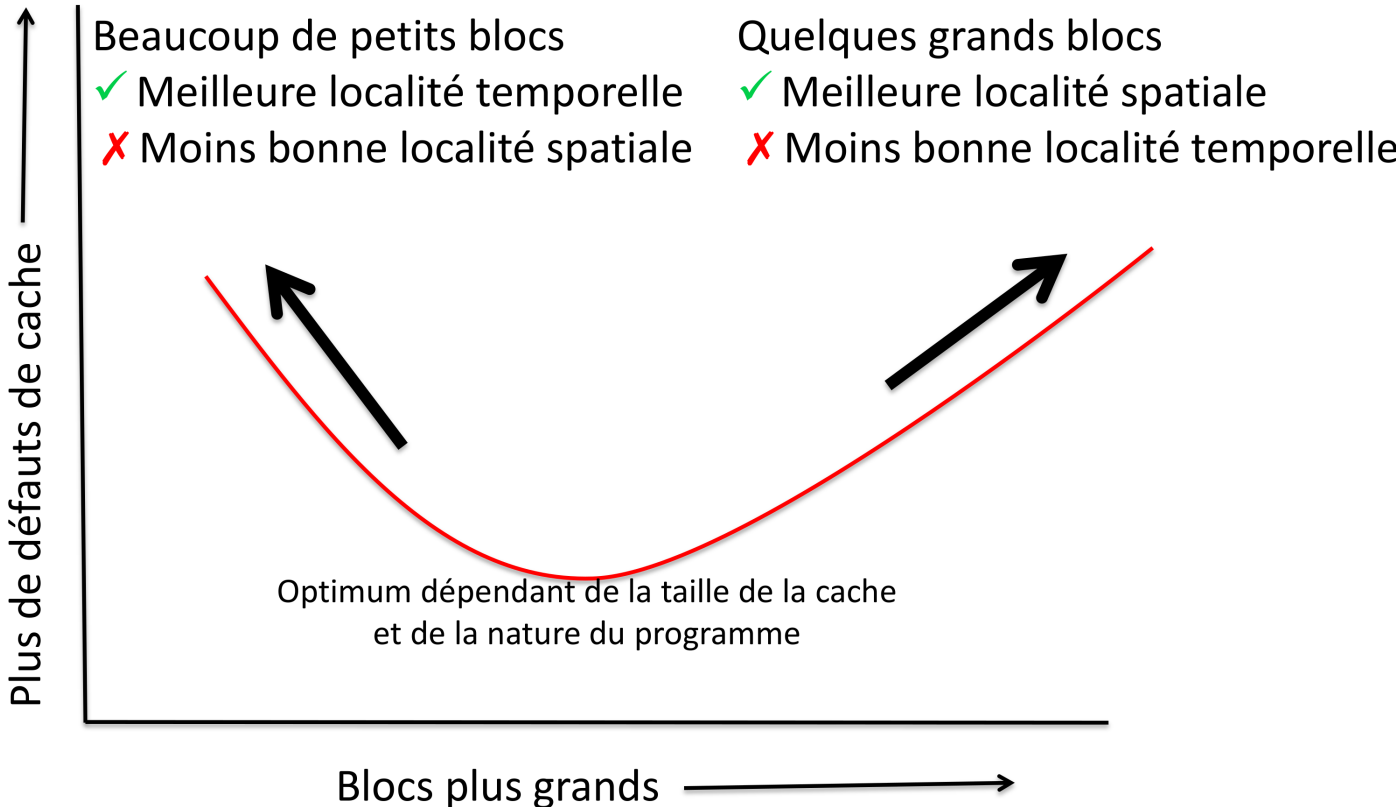
Il serait bien pour P1 : 1 seul défaut de cache (au tout début),

mais très mauvais pour P2 : sans arrêt des défauts de cache.

☞ **MOINS de localité temporele**
(pour des variables espacées en mémoire)

Impact de la taille des blocs

En réalité, on ne fait pas tourner que P1 ou P2, mais plein de programmes différents et la courbe « *nombre de défauts de cache par rapport à la taille de B* » est donc une *moyenne* sur tous ces programmes, situés entre ces deux extrêmes (P1 et P2) :



Exemple : addition des éléments d'une matrice

Addition des éléments de M

$$s \leftarrow \sum_{i=0}^{3}\sum_{j=0}^{3} M(i,j)$$

La somme ne dépend pas de la façon dont les éléments sont ajoutés

$$s \leftarrow \sum_{j=0}^{3}\sum_{i=0}^{3} M(i,j)$$

Mais la façon de faire la somme peut influencer le temps de calcul

Matrix M(i,j)

M(0,0)	M(0,1)	M(0,2)	M(0,3)
M(1,0)	M(1,1)	M(1,2)	M(1,3)
M(2,0)	M(2,1)	M(2,2)	M(2,3)
M(3,0)	M(3,1)	M(3,2)	M(3,3)

Addition par lignes

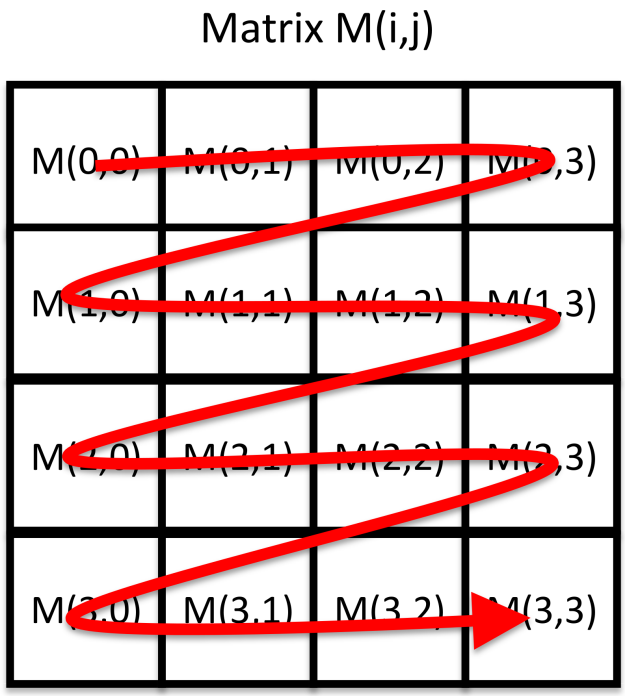
Première option :

- additionner les éléments d'une ligne
- puis passer à la ligne suivante

somme

entrée : *matrice M 4x4*
sortie : *somme des éléments de M*

$s \leftarrow 0$
Pour i *de* 0 *à* 3
  Pour j *de* 0 *à* 3
    $s \leftarrow s + M(i,j)$
sortir : s



$$\sum_{i=0}^{3} \sum_{j=0}^{3} M(i,j)$$

Addition par colonnes

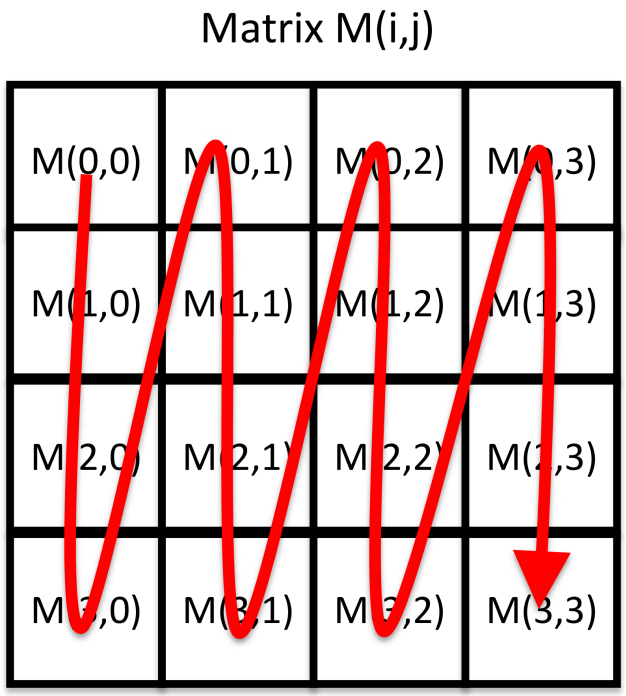
Deuxième option :

- additionner les éléments d'une colonne
- puis passer à la colonne suivante

somme

entrée : *matrice M 4x4*
sortie : *somme des éléments de M*

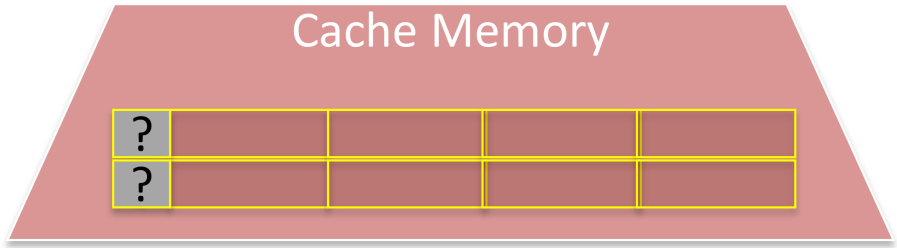
$s \leftarrow 0$
Pour j *de* 0 *à* 3
  Pour i *de* 0 *à* 3
    $s \leftarrow s + M(i,j)$
sortir : s

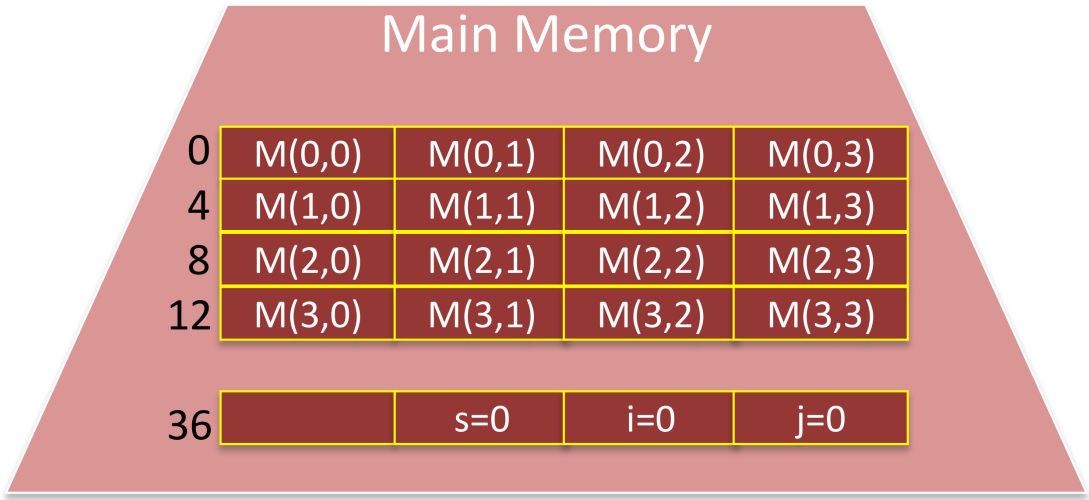


$$\sum_{j=0}^{3} \sum_{i=0}^{3} M(i,j)$$

Exemple de disposition mémoire

Supposons que l'on ait une cache de 2 blocs et que la matrice soit organisée de la façon suivante en mémoire :



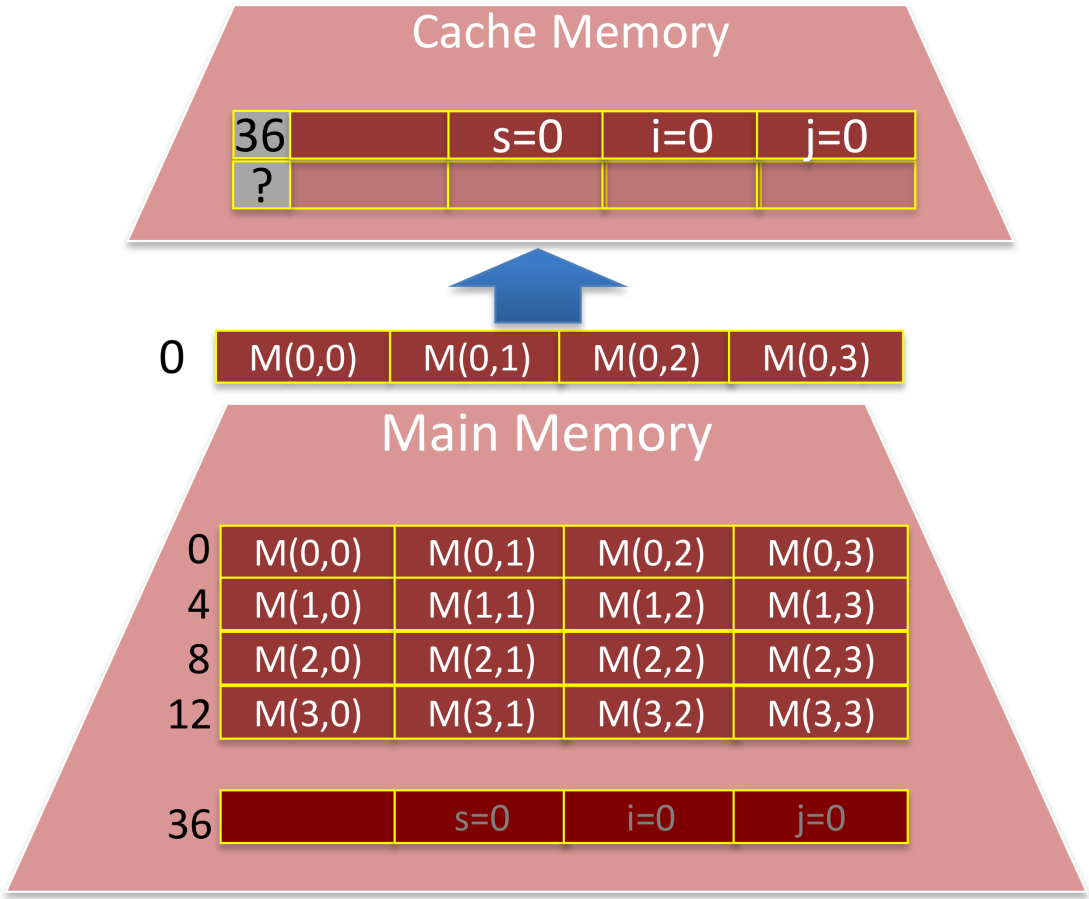


Exercice

- combien de défauts de cache dans l'addition par lignes ?
- combien de défauts de cache dans l'addition par colonnes ?
- laquelle est la plus performante ?
  (par exemple, si l'accès la cache prend 1 ns et l'accès à la mémoire centrale prend 100 ns)

Analyse

Une fois *i*, *j* et *s* en cache (qui y restent par localité temporelle) :



☞ 1 défaut de cache à chaque fois qu'on accède à une nouvelle ligne

Accès mémoire – addition par lignes

- Par boucle :
  1 seul défaut de cache,
  une dizaine (17 exactement) d'accès à la cache

- Au total :
  5 défauts de cache (1 initial + 4 lignes)
  4×17 accès à la cache

Accès mémoire – addition par colonnes

- Par boucle :
  4 défauts de cache (1 par ligne),
  une dizaine (17 exactement) d'accès à la cache

- Au total :
  17 défauts de cache (1 initial + 4x4 lignes)
  4×17 accès à la cache

Bilan

- par lignes : $4 \times 17 + 5 \times 100 \simeq 600$ ns
- par colonnes : $4 \times 17 + 17 \times 100 \simeq 1800$ ns

☞ 3 fois plus lent !

Exemple concret : 150 répétitions sur une matrice 1000x1000 en C++ sur ma machine :

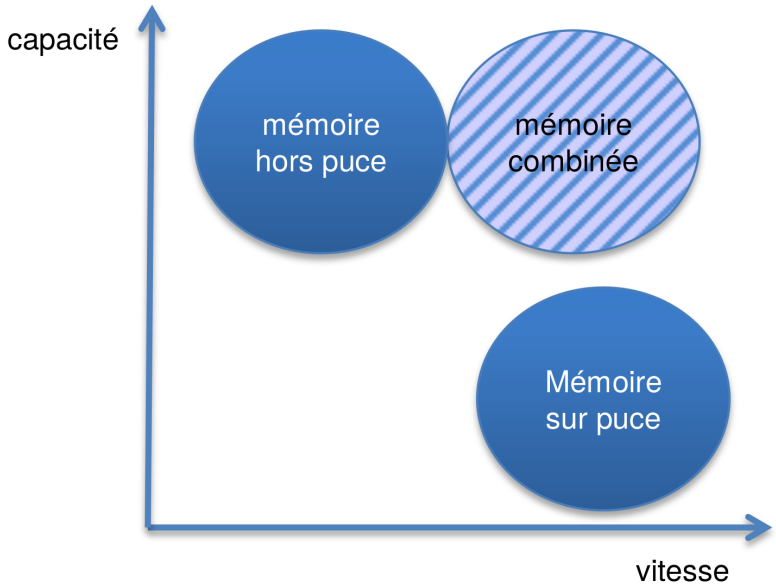
```
time ./lines
real    0m1.086s

time ./columns
real    0m2.090s
```

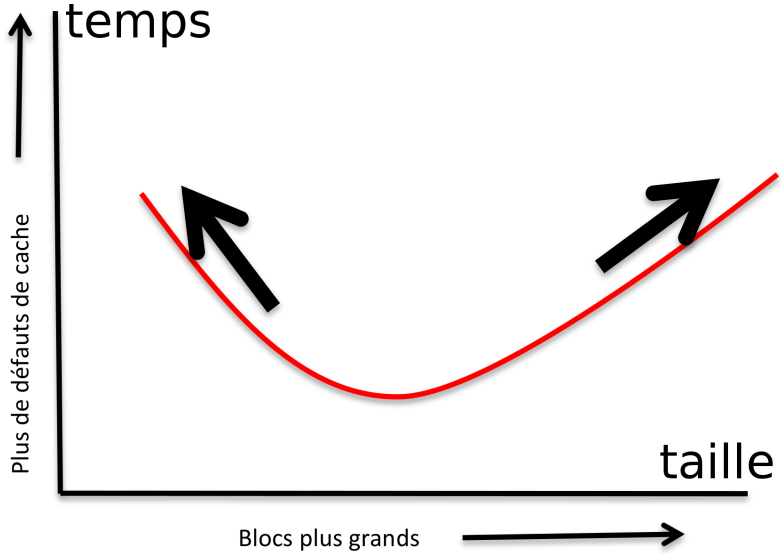
☞ **2 fois plus lent !**

☞ Conclusion : **attention à l'ordre de vos boucles** en programmation ! !

Ce que j'ai appris aujourd'hui

- Différence de technologie : latence, débit, capacité, volatilité
  ☞ vitesse vs. capacité
- Deux types de stockage
  - Mémoire pour le calcul
  - Mémoire de stockage pour l'archivage de données
- Hiérarchies : rendre la mémoire à la fois grande et rapide
- Localité spatiale et temporelle
- Conséquence : programmation : attention à l'ordre de vos boucles !





Pour ceux que ça intéresse

Le livre « *Penser en algorithmes* »
(B. Christian & T. Griffiths ; PPUR 2019 ;
trad. de l'anglais « *Algorithms to Live By* »)
présente ce sujet dans son
chapitre 4 : « *Mettre en cache* »