NMV : Topologie mémoire

Gauthier Voron - gauthier.voron@lip6.fr

Topologie mémoire : pourquoi

Code source	Temps d'exécution

`for (i = 0; i < WIDTH; i++)`
`  for (j = 0; j < HEIGHT; j++)`
`  array[j * WIDTH + i]++;`
	16.64 secondes

`for (j = 0; j < HEIGHT; j++)`
`  for (i = 0; i < WIDTH; i++)`
`  array[j * WIDTH + i]++;`
	0.35 secondes

Topologie mémoire : pourquoi

Code source	Temps d'exécution

`for (i = 0; i < WIDTH; i++)`
`  for (j = 0; j < HEIGHT; j++)`
`  array[j * WIDTH + i]++;`
	16.64 secondes

`for (j = 0; j < HEIGHT; j++)`
`  for (i = 0; i < WIDTH; i++)`
`  array[j * WIDTH + i]++;`
	0.35 secondes

- Les deux codes sont fontionellement équivalents

Topologie mémoire : pourquoi

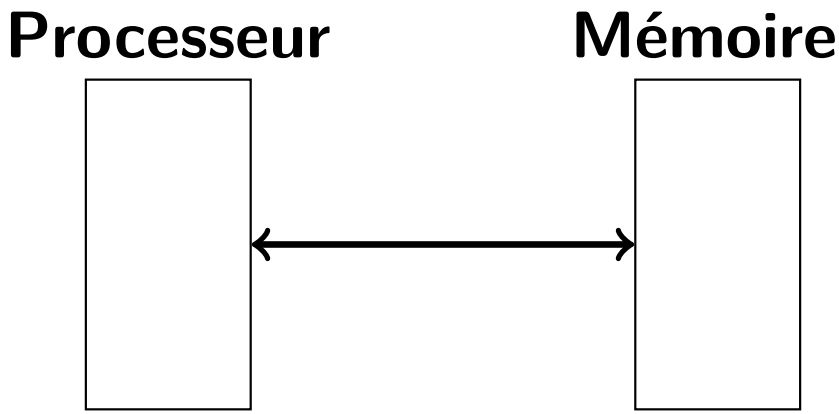
Code source	Temps d'exécution

`for (i = 0; i < WIDTH; i++)`
`  for (j = 0; j < HEIGHT; j++)`
`  array[j * WIDTH + i]++;`
	16.64 secondes

`for (j = 0; j < HEIGHT; j++)`
`  for (i = 0; i < WIDTH; i++)`
`  array[j * WIDTH + i]++;`
	0.35 secondes

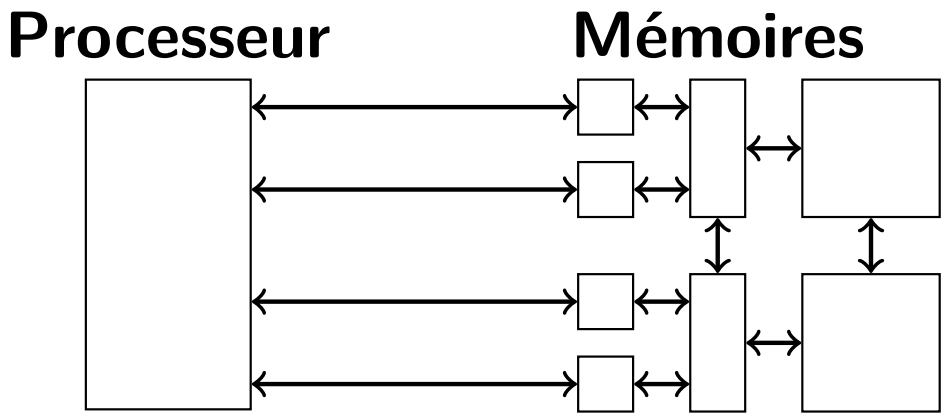
- Les deux codes sont fontionellement équivalents
- La seconde version exploite correctement la topologie mémoire

Topologie mémoire : quoi



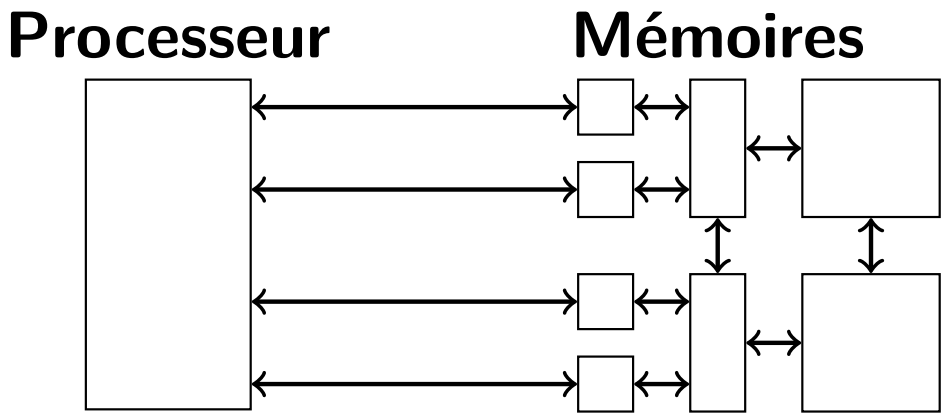
- Représentation simple d'un ordinateur : le processeur accède à la mémoire

Topologie mémoire : quoi



- Représentation simple d'un ordinateur : le processeur accède à la mémoire
- Il n'y a pas une mémoire unique mais plusieurs mémoires qui communiquent entre elles et avec le processeur

Topologie mémoire : quoi



- Représentation simple d'un ordinateur : le processeur accède à la mémoire
- Il n'y a pas une mémoire unique mais plusieurs mémoires qui communiquent entre elles et avec le processeur
- La topologie mémoire est la manière dont les différentes mémoires d'un système sont reliées et communiquent entre elles

Plan du cours

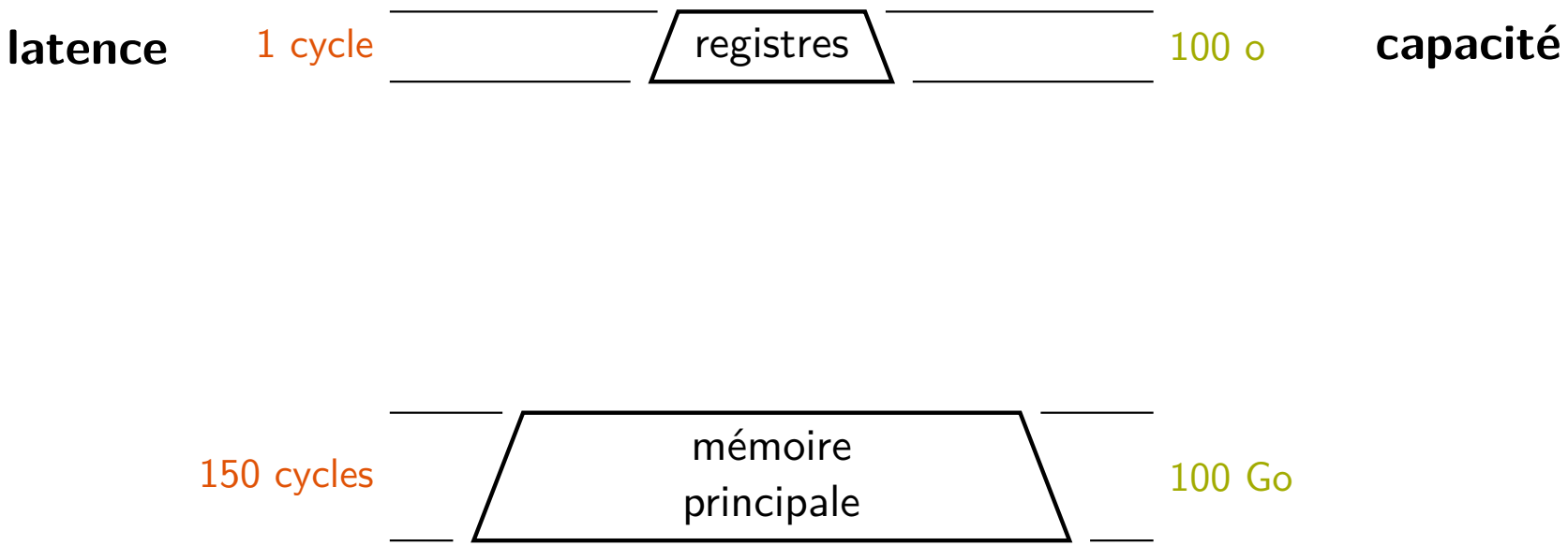
1. Mémoire cache en monocœur
   - Hiérarchie mémoire et mémoire cache
   - Cache direct et collisions d'adresses
   - Cache associatif et stratégies d'éviction
   - Motifs d'accès et *prefetching*
2. Multicœur et cohérence de caches
   - Caches multicœurs et cohérence séquentielle
   - Protocole MESI
   - Topologie de caches et stratégies d'inclusion
   - Contrôle logiciel du cache
3. Architectures *Non Uniform Memory Access*
   - Contention matérielle et architecture NUMA
   - Cohérence NUMA et *cache directory*
   - Stratégies d'allocation mémoire

Plan du cours

Hiérarchie mémoire : latence vs. capacité

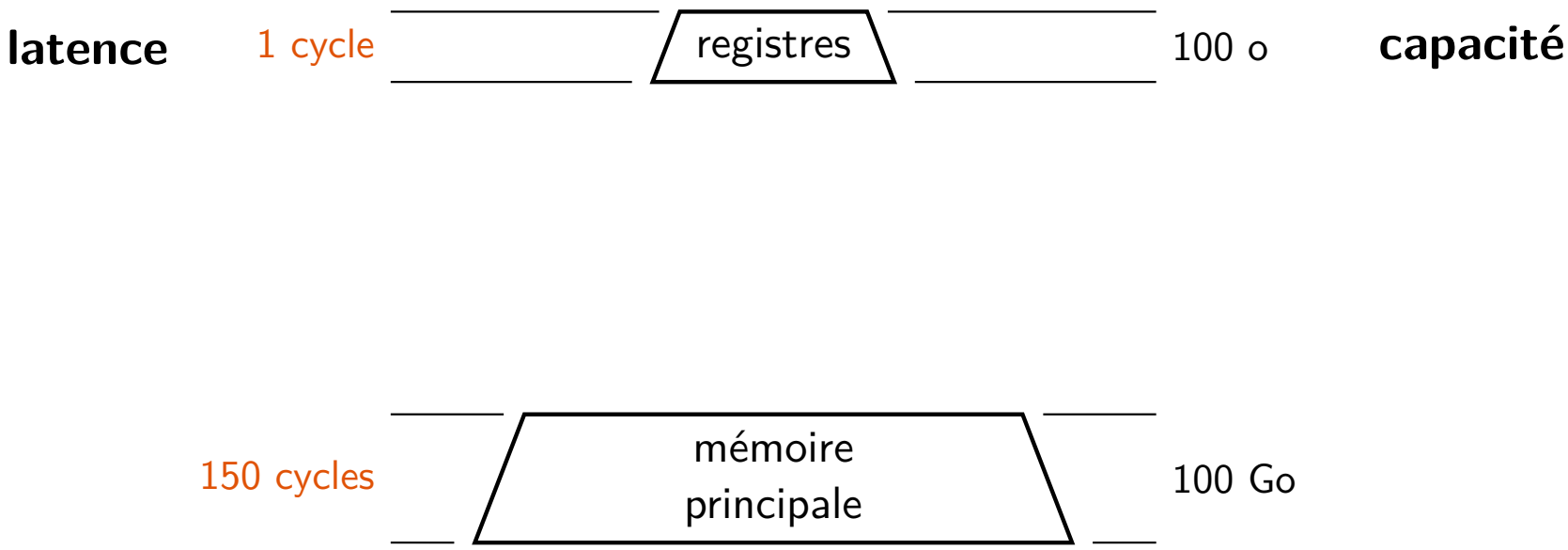
- Le processeur est une unité de traitement
  - Objectif : traiter rapidement de petites données
  - Solution : données placées dans un banc de registres

Hiérarchie mémoire : latence vs. capacité



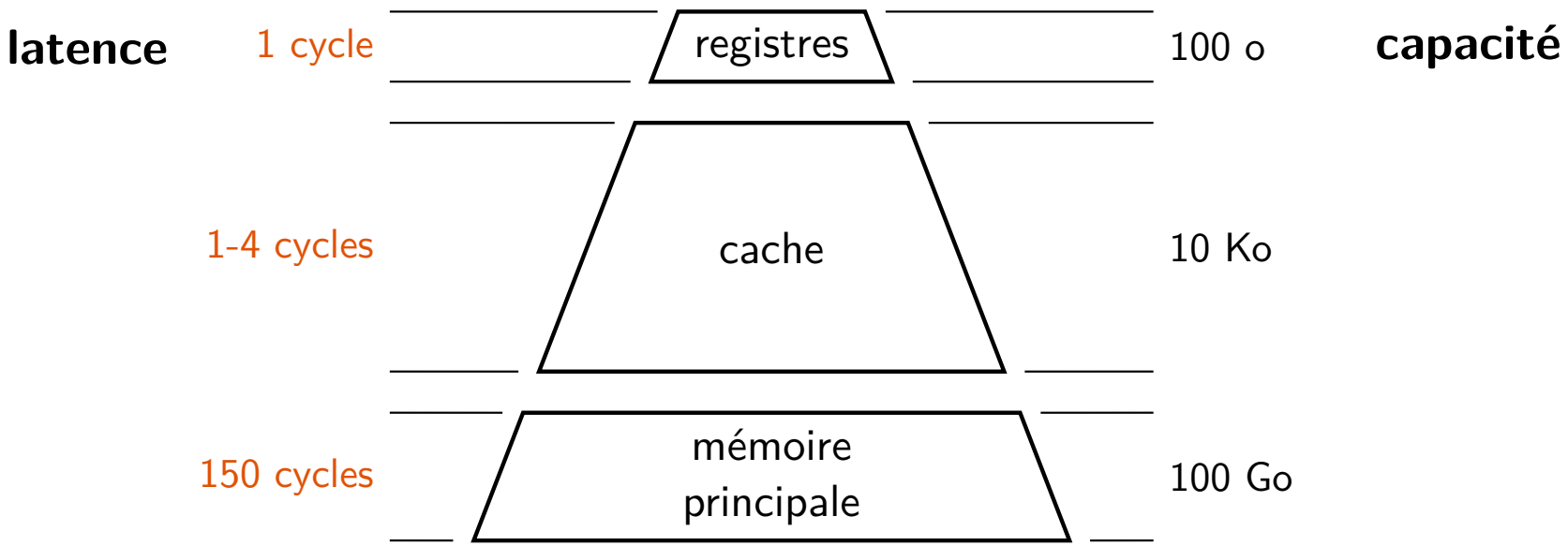
- Le processeur est une unité de traitement
  - Objectif : traiter rapidement de petites données
  - Solution : données placées dans un banc de registres
- La mémoire est une unité de stockage
  - Objectif : stocker longtemps de grosses données
  - Solution : données placées dans des bancs de DRAM

Hiérarchie mémoire : latence vs. capacité



- À chaque accès mémoire, le processeur attend la donnée (*stall*)
  - Rappel : 100% des instructions sont stockées en mémoire
- Si le processeur faisait un accès mémoire par instruction
  - Alors le processeur passerait 99% du temps à attendre

Hiérarchie mémoire : latence vs. capacité



- À chaque accès mémoire, le processeur attend la donnée (*stall*)
  - Rappel : 100% des instructions sont stockées en mémoire
- Si le processeur faisait un accès mémoire par instruction
  - Alors le processeur passerait 99% du temps à attendre
- Il faut une zone de stockage intermédiaire : le cache

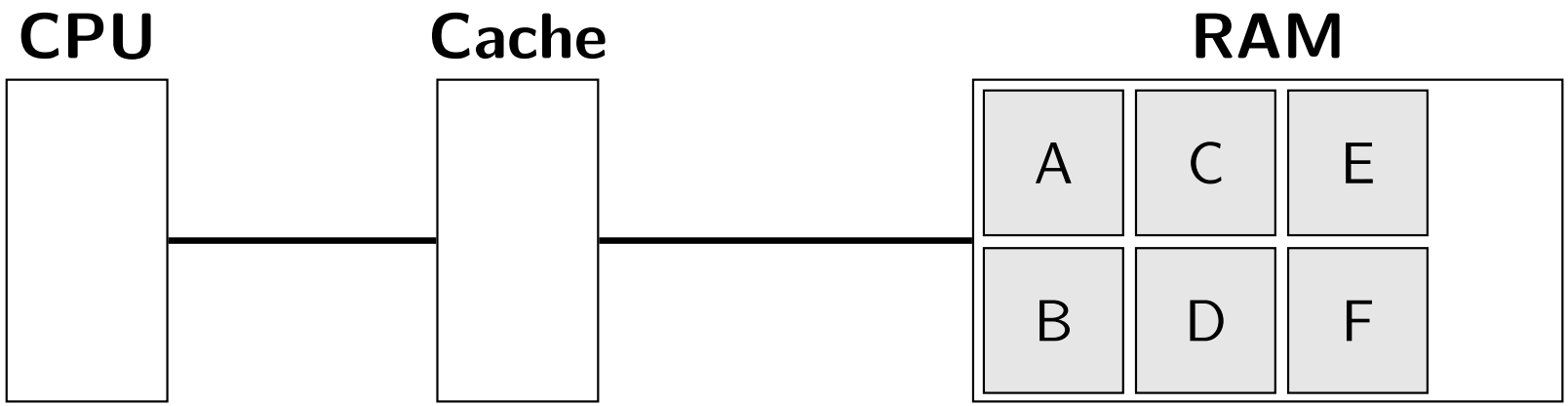
Mémoire cache : définitions

- Une mémoire cache est une zone de stockage intermédiaire entre une zone de stockage principale et son utilisateur
  - Généralement plus rapide d'accès que la zone principale
  - Généralement de plus faible capacité que la zone principale

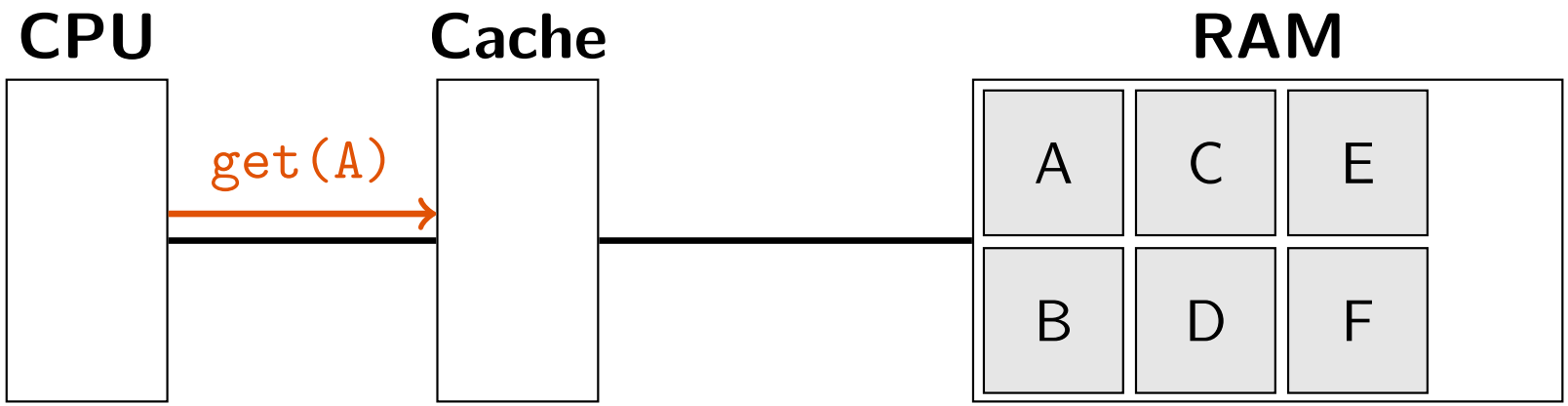
- Principe de localité temporelle : si un programme accède à une adresse A à un temps t, ce programme accède probablement à cette même adresse A au temps $t + \epsilon$

- Principe de localité spatiale : si un programme accède à une adresse A à un temps t, ce programme accède probablement à une adresse proche $A + \delta$ au temps $t + \epsilon$

Mémoire cache : principe de fonctionnement (lecture)

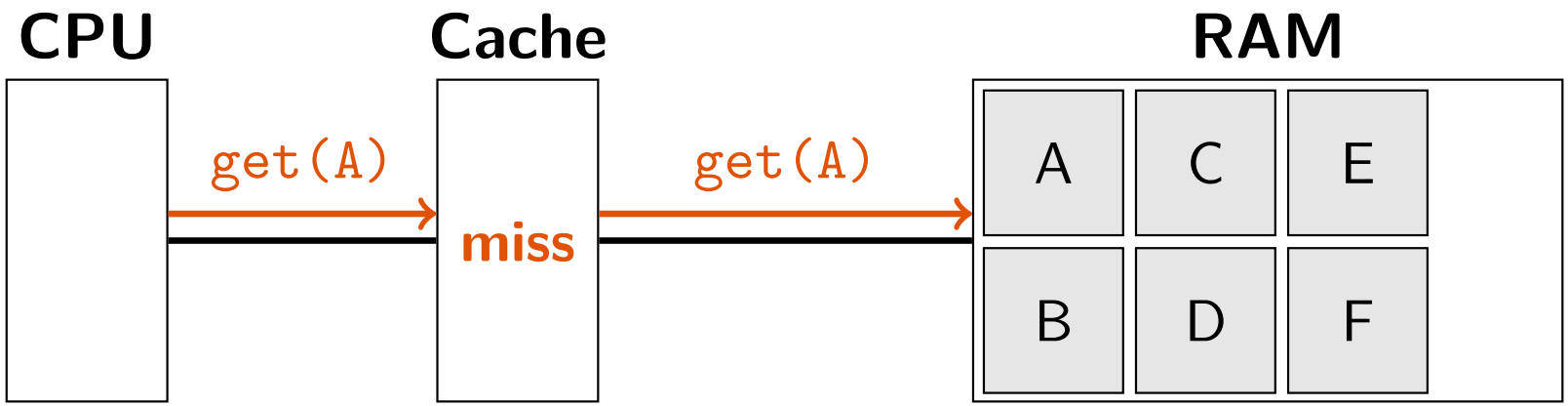


Mémoire cache : principe de fonctionnement (lecture)



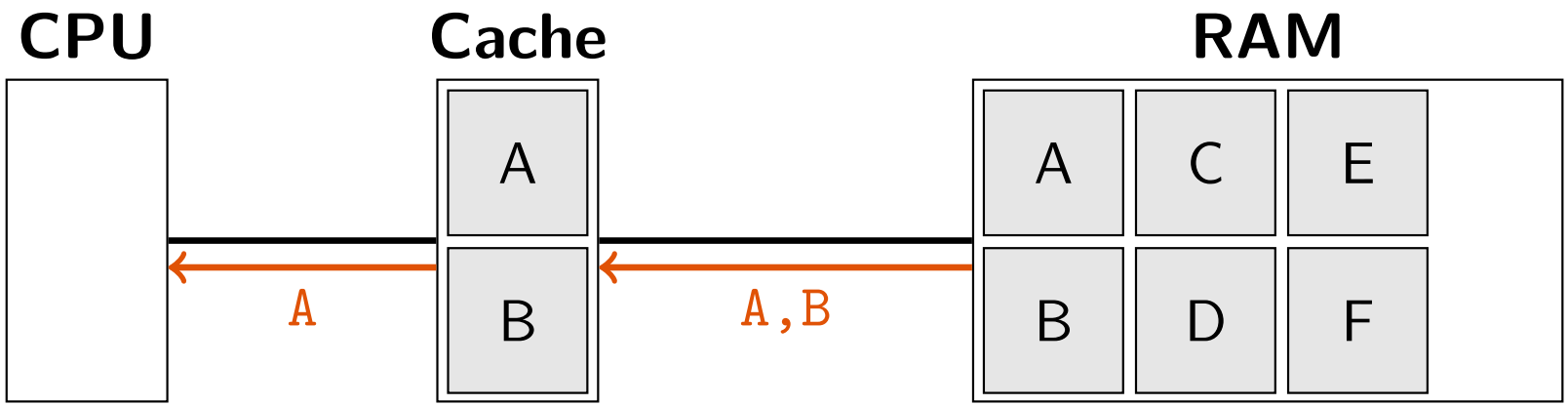
- Le CPU demande la donnée A qui n'est pas dans le cache

Mémoire cache : principe de fonctionnement (lecture)



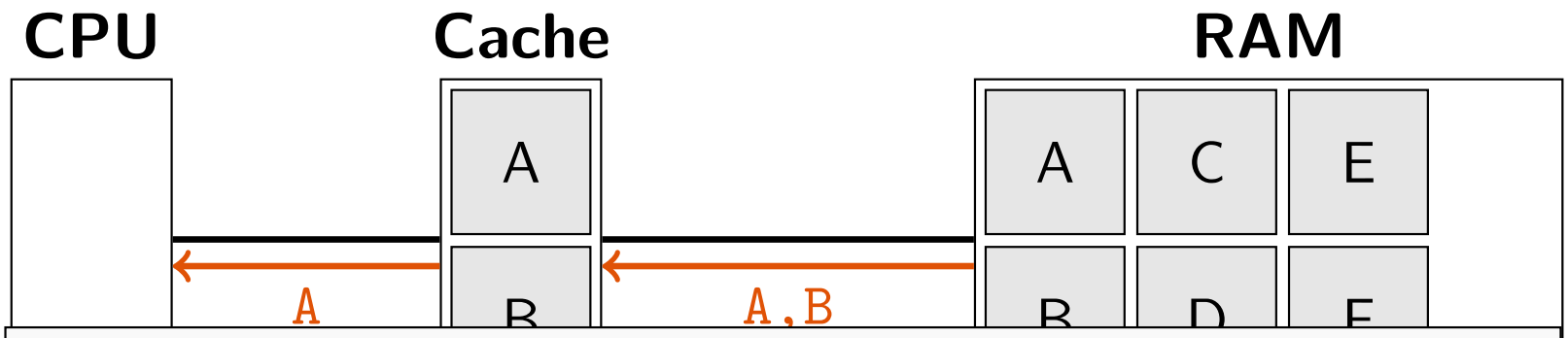
- Le CPU demande la donnée A qui n'est pas dans le cache
  - Cache miss : accès à la mémoire principale obligatoire

Mémoire cache : principe de fonctionnement (lecture)



- Le CPU demande la donnée A qui n'est pas dans le cache
  - Cache miss : accès à la mémoire principale obligatoire
  - Les données A et $A + \delta$ sont stockées en mémoire cache

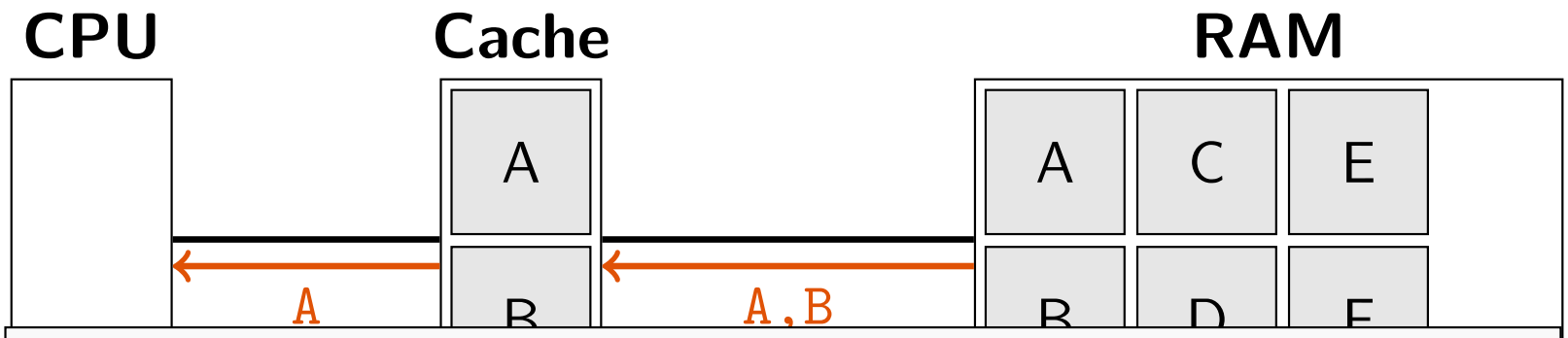
Mémoire cache : principe de fonctionnement (lecture)



- Transférer A et B en cache est-il plus coûteux que transférer seulement A ?

- L
  - Cache miss : accès à la mémoire principale obligatoire
  - Les données A et $A + \delta$ sont stockées en mémoire cache

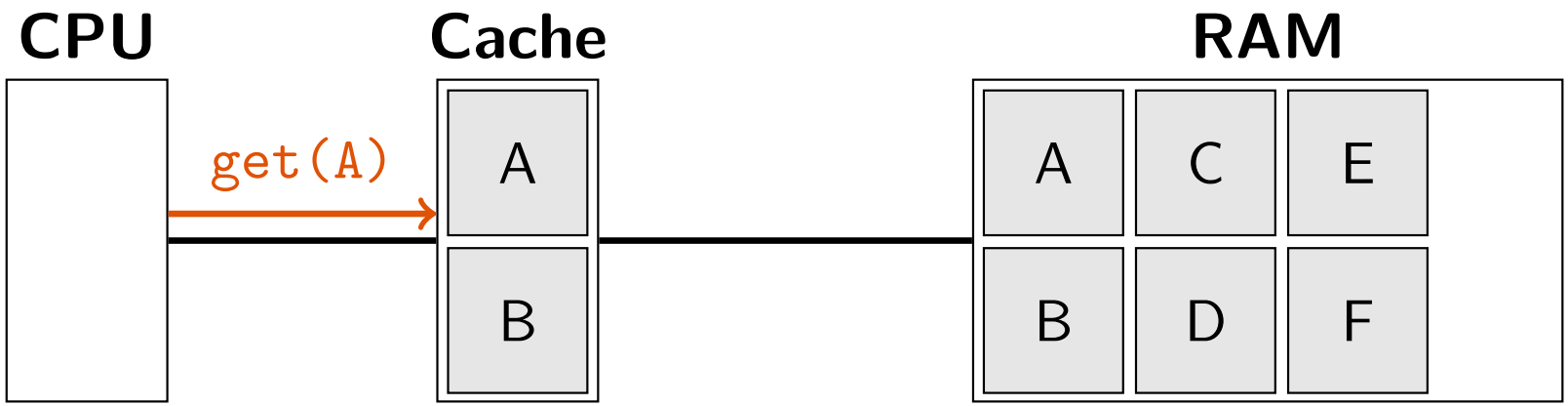
Mémoire cache : principe de fonctionnement (lecture)



- Transférer A et B en cache est-il plus coûteux que transférer seulement A ? $\rightarrow$ non
- N'augmente pas la latence d'accès à la mémoire principale
- Il y a suffisamment de débit pour transférer A et B

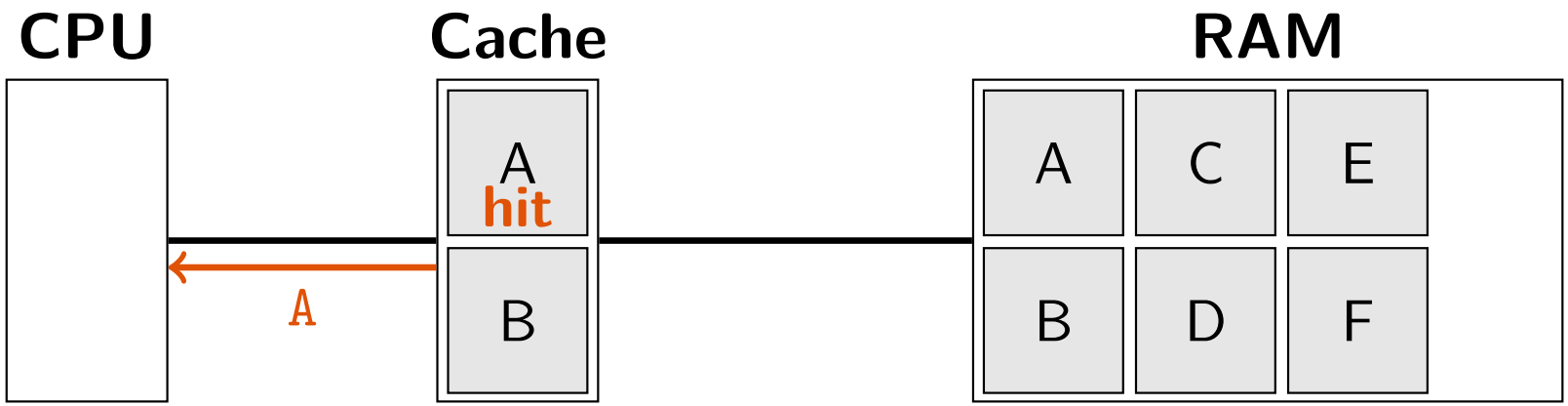
- L
  - Cache miss : accès à la mémoire principale obligatoire
  - Les données A et $A + \delta$ sont stockées en mémoire cache

Mémoire cache : principe de fonctionnement (lecture)



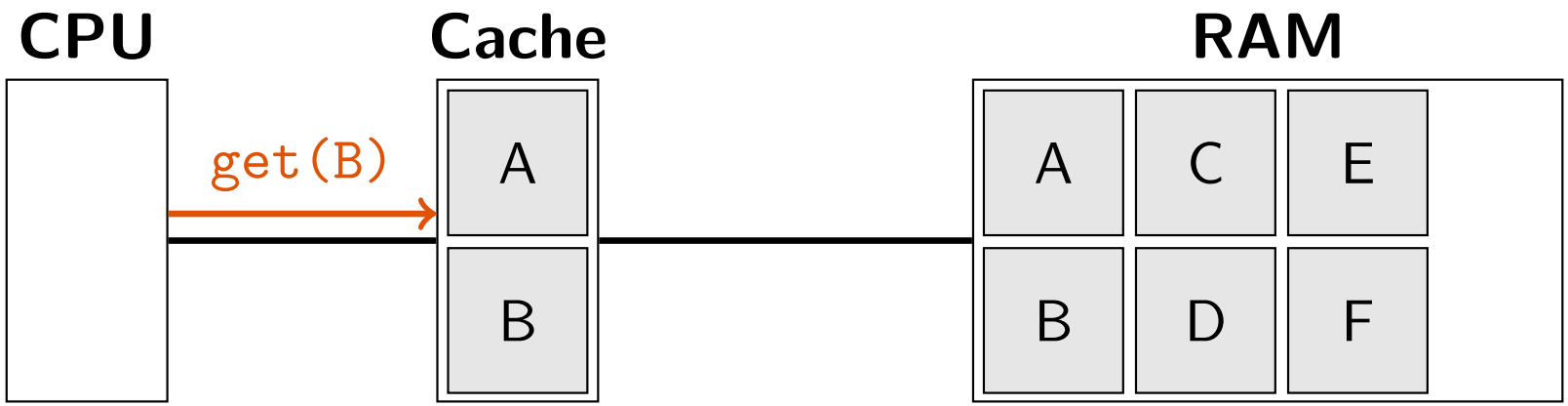
- Le CPU demande la donnée A qui n'est pas dans le cache
  - Cache miss : accès à la mémoire principale obligatoire
  - Les données A et $A + \delta$ sont stockées en mémoire cache
- Le CPU demande à nouveau la donnée $A \rightarrow$ localité temporelle

Mémoire cache : principe de fonctionnement (lecture)



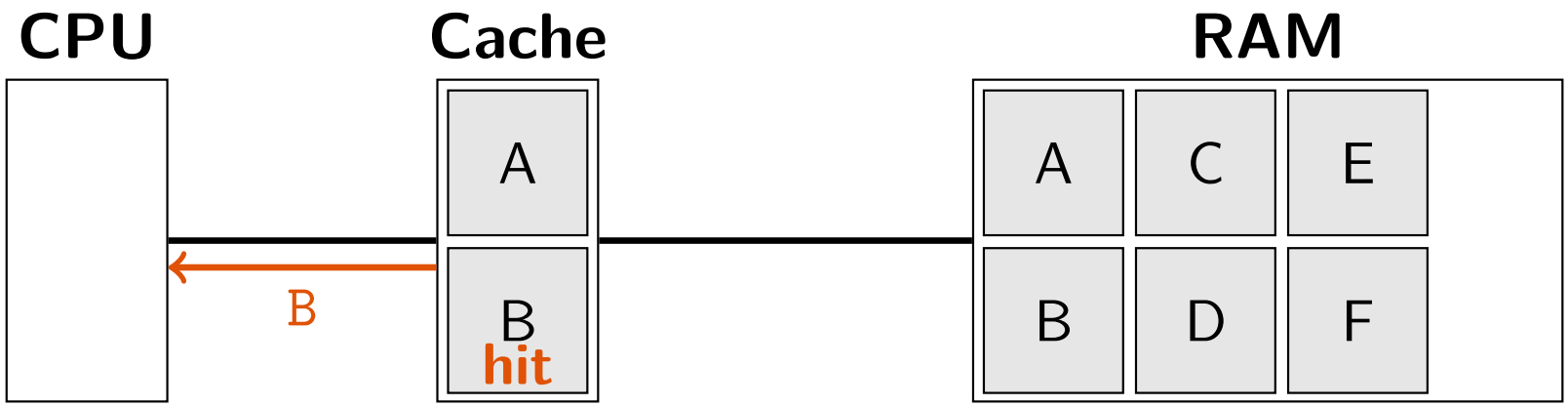
- Le CPU demande la donnée A qui n'est pas dans le cache
  - Cache miss : accès à la mémoire principale obligatoire
  - Les données A et $A + \delta$ sont stockées en mémoire cache
- Le CPU demande à nouveau la donnée $A \rightarrow$ localité temporelle
  - Cache hit : accès à la mémoire cache uniquement

Mémoire cache : principe de fonctionnement (lecture)



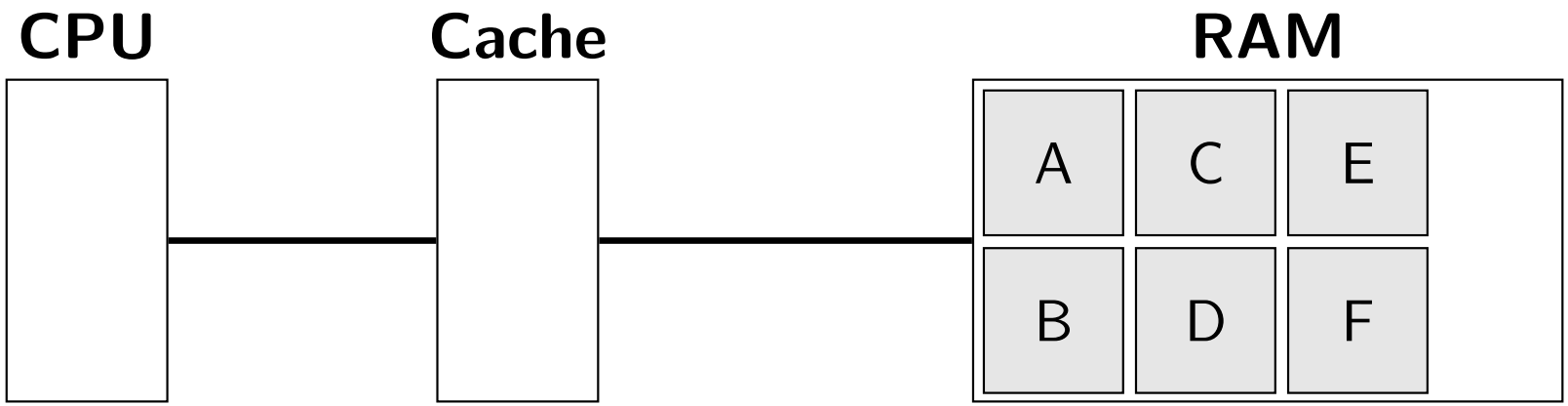
- Le CPU demande la donnée A qui n'est pas dans le cache
  - Cache miss : accès à la mémoire principale obligatoire
  - Les données A et $A + \delta$ sont stockées en mémoire cache
- Le CPU demande à nouveau la donnée $A \rightarrow$ localité temporelle
  - Cache hit : accès à la mémoire cache uniquement
- Le CPU demande la donnée $A + \delta \rightarrow$ localité spatiale

Mémoire cache : principe de fonctionnement (lecture)

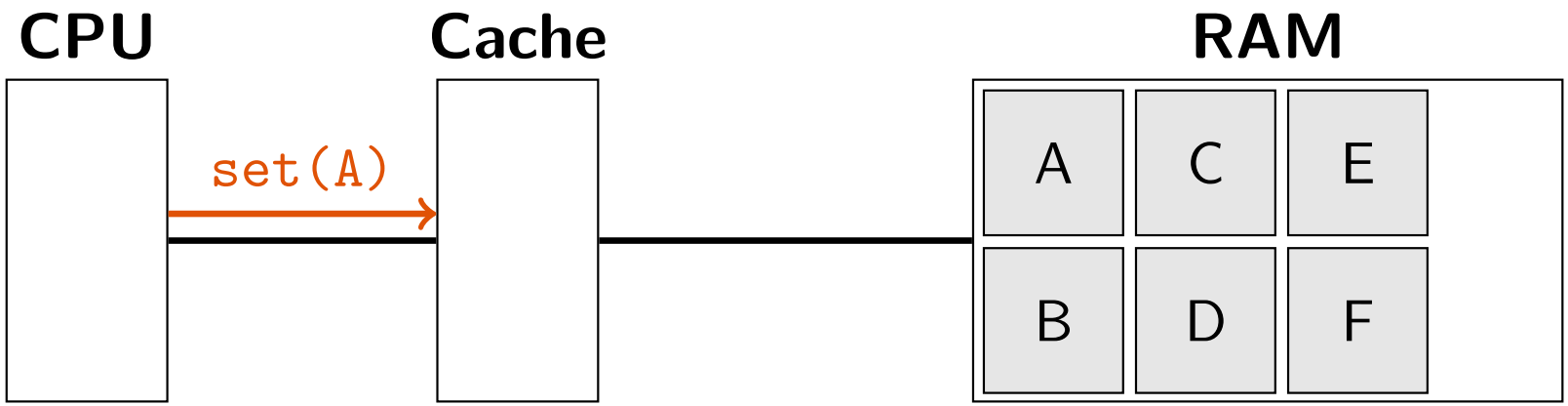


- Le CPU demande la donnée A qui n'est pas dans le cache
  - Cache miss : accès à la mémoire principale obligatoire
  - Les données A et $A + \delta$ sont stockées en mémoire cache
- Le CPU demande à nouveau la donnée $A \rightarrow$ localité temporelle
  - Cache hit : accès à la mémoire cache uniquement
- Le CPU demande la donnée $A + \delta \rightarrow$ localité spatiale
  - Cache hit : accès à la mémoire cache uniquement

Mémoire cache : principe de fonctionnement (écriture)

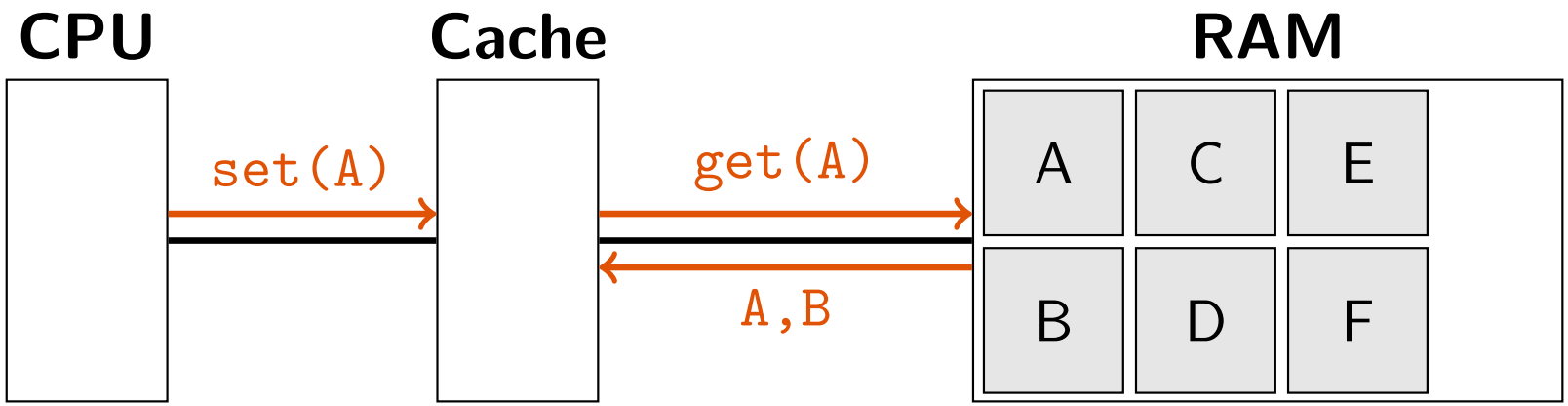


Mémoire cache : principe de fonctionnement (écriture)



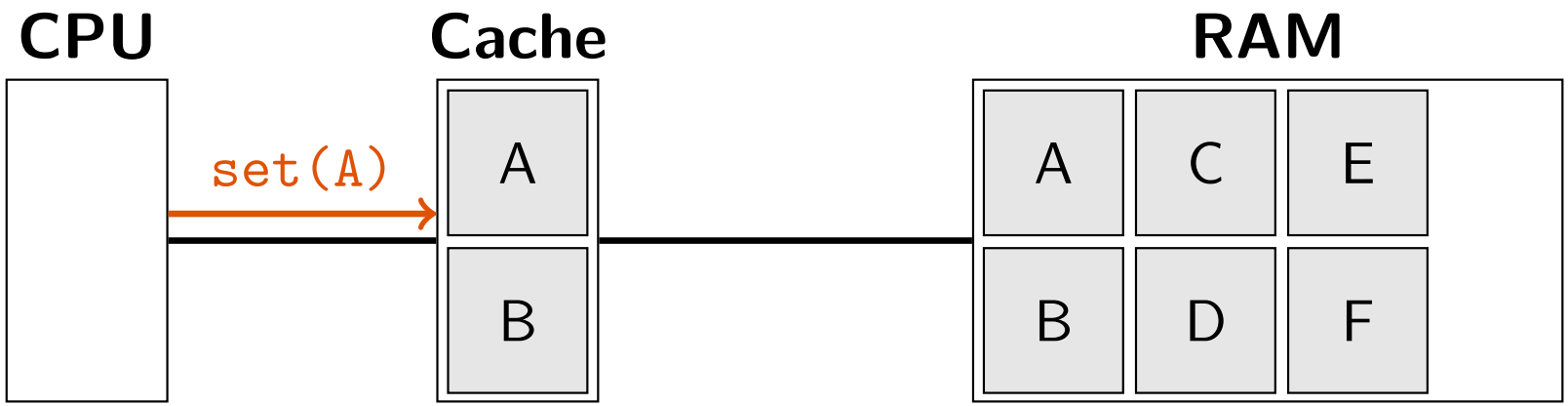
- Le CPU modifie la donnée *A* qui n'est pas dans le cache

Mémoire cache : principe de fonctionnement (écriture)



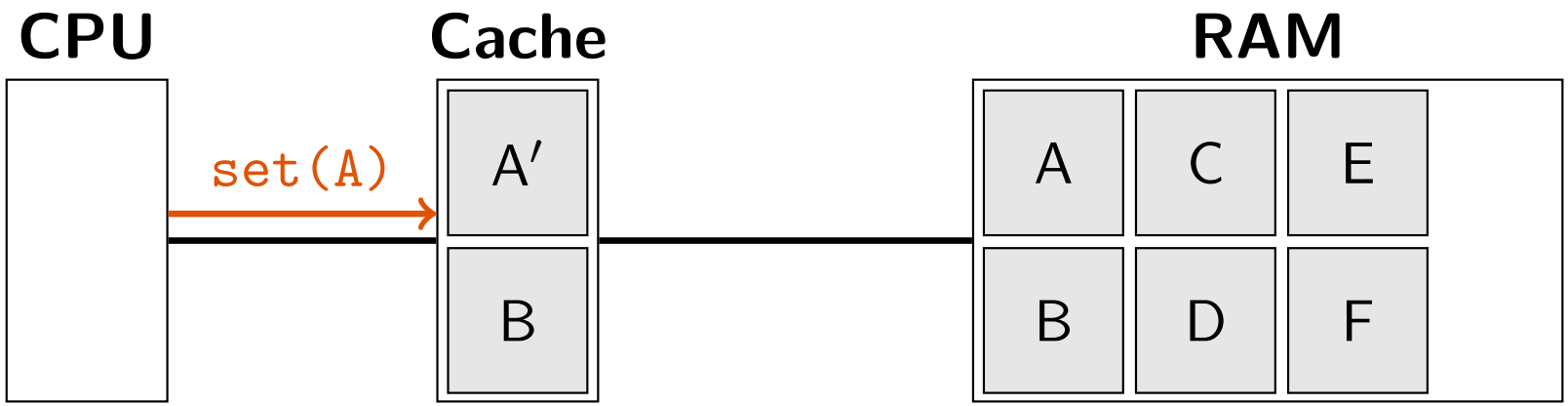
- Le CPU modifie la donnée A qui n'est pas dans le cache
  - Cache miss : la donnée est d'abord ramenée dans le cache

Mémoire cache : principe de fonctionnement (écriture)



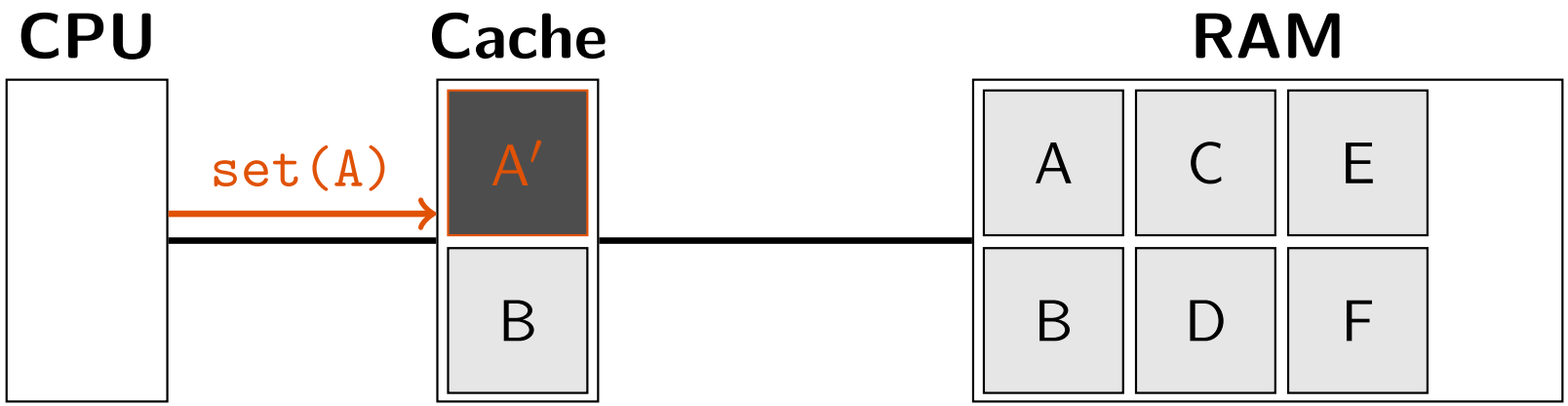
- Le CPU modifie la donnée A qui n'est pas dans le cache
  - Cache miss : la donnée est d'abord ramenée dans le cache
- Le CPU modifie la donnée A qui est dans le cache

Mémoire cache : principe de fonctionnement (écriture)



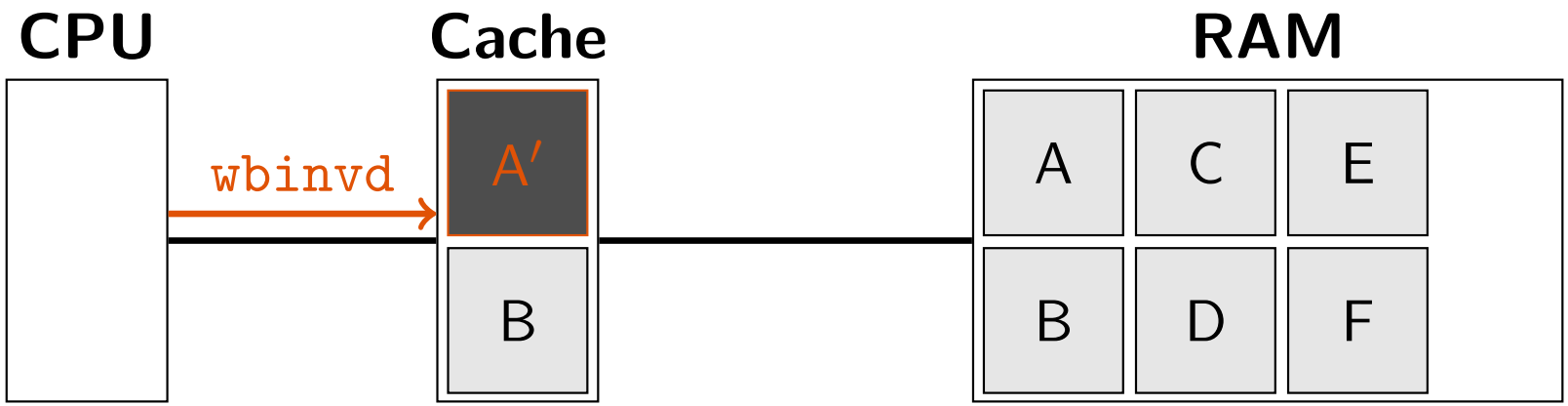
- Le CPU modifie la donnée A qui n'est pas dans le cache
  - Cache miss : la donnée est d'abord ramenée dans le cache
- Le CPU modifie la donnée A qui est dans le cache
  - Cache hit : la donnée A est modifiée dans le cache

Mémoire cache : principe de fonctionnement (écriture)



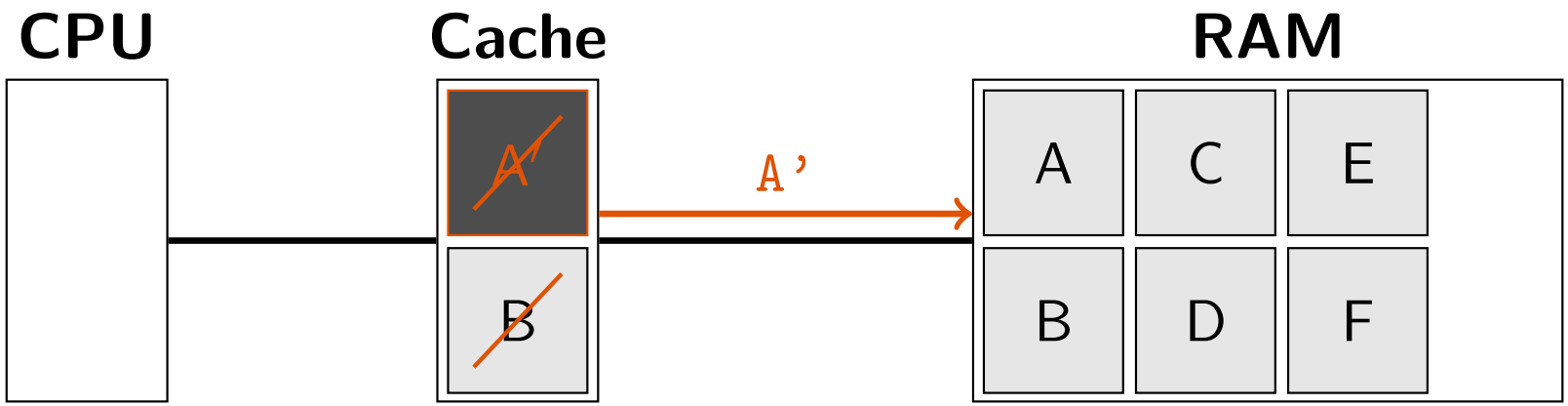
- Le CPU modifie la donnée A qui n'est pas dans le cache
  - Cache miss : la donnée est d'abord ramenée dans le cache
- Le CPU modifie la donnée A qui est dans le cache
  - Cache hit : la donnée A est modifiée dans le cache
  - La donnée A' est marquée *dirty* dans le cache

Mémoire cache : principe de fonctionnement (écriture)



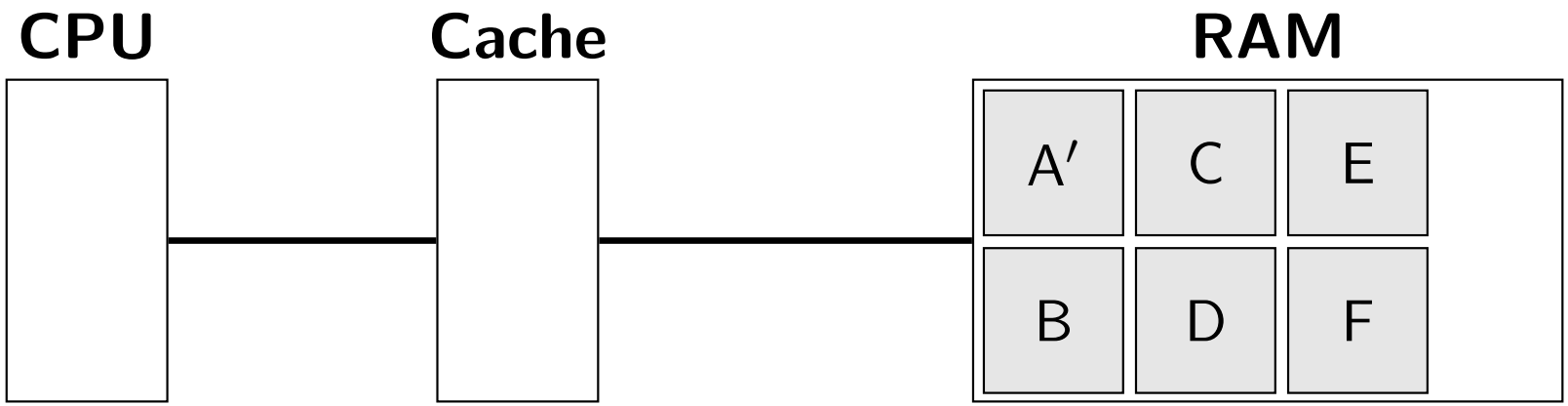
- Le CPU modifie la donnée A qui n'est pas dans le cache
  - Cache miss : la donnée est d'abord ramenée dans le cache
- Le CPU modifie la donnée A qui est dans le cache
  - Cache hit : la donnée A est modifiée dans le cache
  - La donnée A' est marquée *dirty* dans le cache
- Le CPU vide le cache

Mémoire cache : principe de fonctionnement (écriture)



- Le CPU modifie la donnée A qui n'est pas dans le cache
  - Cache miss : la donnée est d'abord ramenée dans le cache
- Le CPU modifie la donnée A qui est dans le cache
  - Cache hit : la donnée A est modifiée dans le cache
  - La donnée A' est marquée *dirty* dans le cache
- Le CPU vide le cache
  - Write-back : les données *dirty* sont propagées en mémoire principale

Mémoire cache : principe de fonctionnement (écriture)



- Le CPU modifie la donnée A qui n'est pas dans le cache
  - Cache miss : la donnée est d'abord ramenée dans le cache
- Le CPU modifie la donnée A qui est dans le cache
  - Cache hit : la donnée A est modifiée dans le cache
  - La donnée A' est marquée *dirty* dans le cache
- Le CPU vide le cache
  - Write-back : les données *dirty* sont propagées en mémoire principale

Performance du cache et *hit rate*

- Quand le CPU accède, en lecture ou en écriture à une donnée
  - Si la donnée est dans le cache : cache hit $\rightarrow$ faible latence
  - Si la donnée n'est pas dans le cache : cache miss $\rightarrow$ forte latence

- Le hit rate d'une exécution est la proportion :

  (nombre de `cache hit`) / (nombre d'accès)

  pendant cette exécution

- Plus le hit rate d'une exécution est élevé
  - moins le processeur passe de temps en *stall*
  - plus le processeur fait d'opérations par seconde

- Le hit rate d'une exécution dépend
  - du programme exécuté $\rightarrow$ responsabilité du programmeur / compilateur
  - du fonctionnement de la mémoire cache $\rightarrow$ responsabilité du matériel

Performance du cache et *hit rate*

- Quand le CPU accède, en lecture ou en écriture à une donnée
  - Si la donnée est dans le cache : cache hit $\rightarrow$ faible latence
  - Si la donnée n'est pas dans le cache : cache miss $\rightarrow$ forte latence

- Le hit rate d'une exécution est la proportion :

  (nombre de `cache hit`) / (nombre d'accès)

  pendant cette exécution

- Plus le hit rate d'une exécution est élevé
  - moins le processeur passe de temps en *stall*
  - plus le processeur fait d'opérations par seconde

- Le hit rate d'une exécution dépend
  - du programme exécuté $\rightarrow$ responsabilité du programmeur / compilateur
  - du fonctionnement de la mémoire cache $\rightarrow$ responsabilité du matériel

Hiérarchie mémoire et mémoire cache : résumé

- Le processeur traite rapidement de petites quantitées de donnée
- La mémoire principale fournit lentement beaucoup de donnée

- La mémoire cache est une zone de stockage intermédiaire entre le CPU et la mémoire principale

- Quand le CPU demande une donnée
  - Si la donnée existe dans le cache, elle est disponible rapidement
  - Sinon, la donnée est fournie lentement depuis la mémoire principale

- La mémoire cache utilise les principes de localité temporelle et spatiale pour avoir les données demandées à disposition

- Le *hit rate* mesure dans quelle proportion la mémoire cache peut servir rapidement le CPU $\rightarrow$ plus le *hit rate* est élevé, plus l'exécution est rapide

Unité d'échange mémoire : la ligne de cache

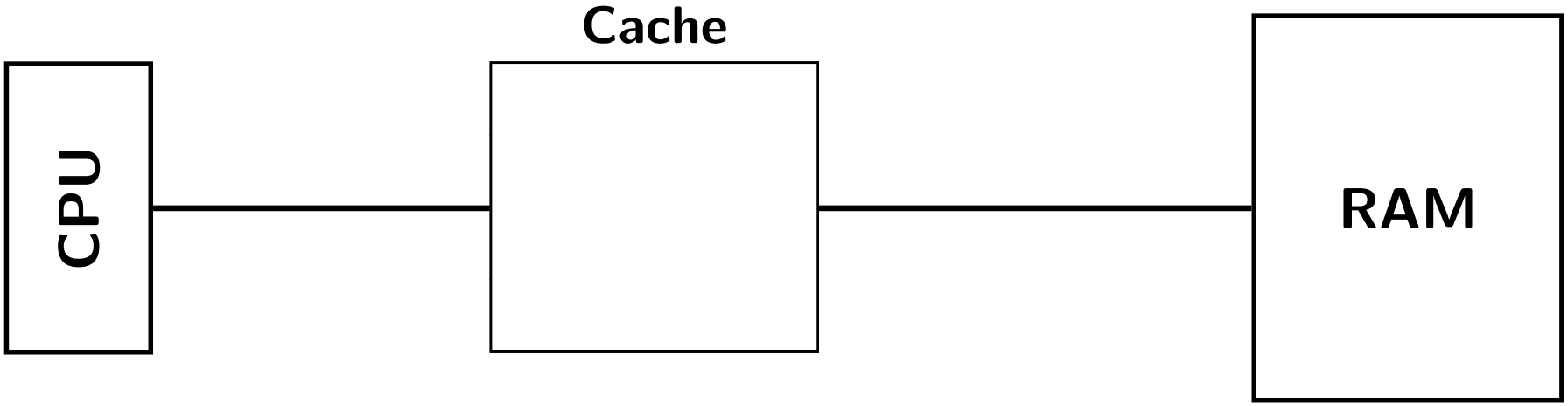
- Une ligne de cache est une donnée de taille fixe. Cette taille est invariable sur un même matériel
- Un emplacement de ligne de cache est un emplacement susceptible de contenir une ligne de cache. Si cet emplacement est en mémoire, il a une adresse alignée sur la taille d'une ligne

- La ligne de cache est l'unité d'échange entre le CPU, la mémoire cache et la mémoire principale

- L'offset est la partie d'une adresse qui désigne la position d'un octet à l'intérieur d'une ligne de cache

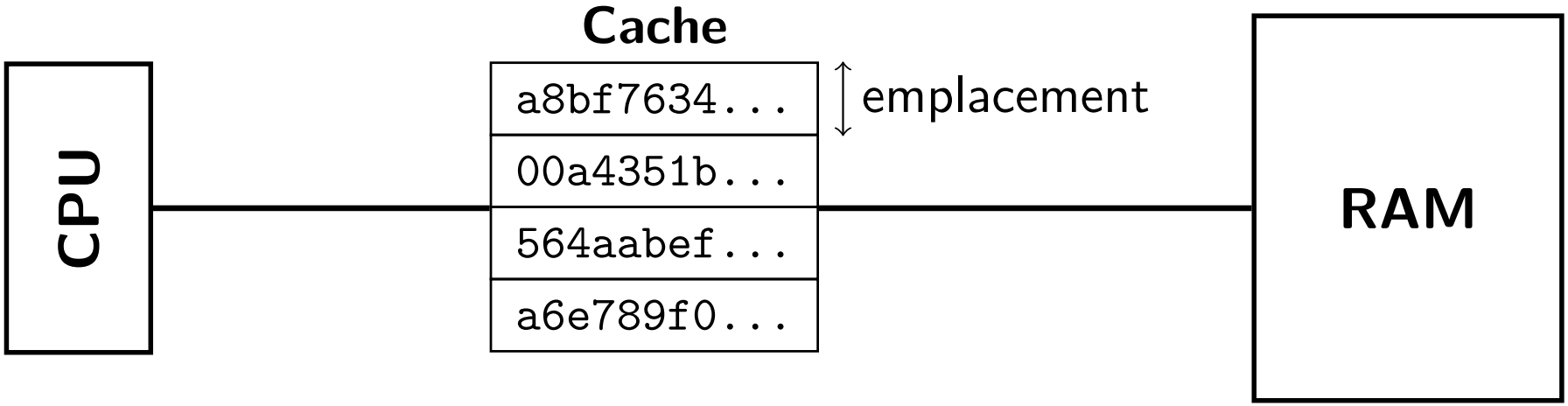
Fonctionnement d'un cache direct : cache hit

- Objectif : trouver rapidement une donnée demandée par le CPU



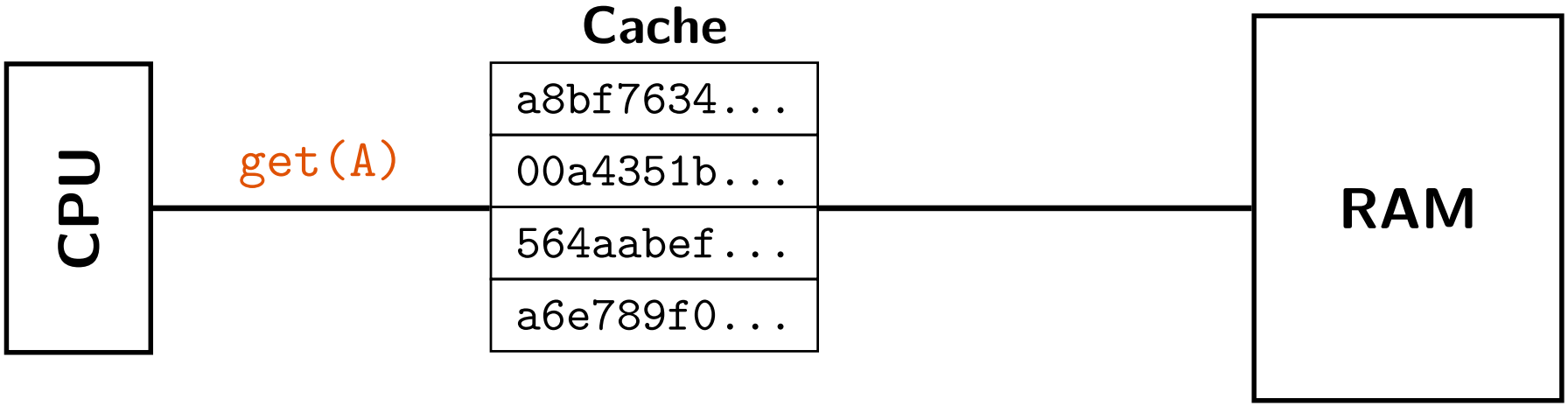
Fonctionnement d'un cache direct : cache hit

- Objectif : trouver rapidement une donnée demandée par le CPU



Fonctionnement d'un cache direct : cache hit

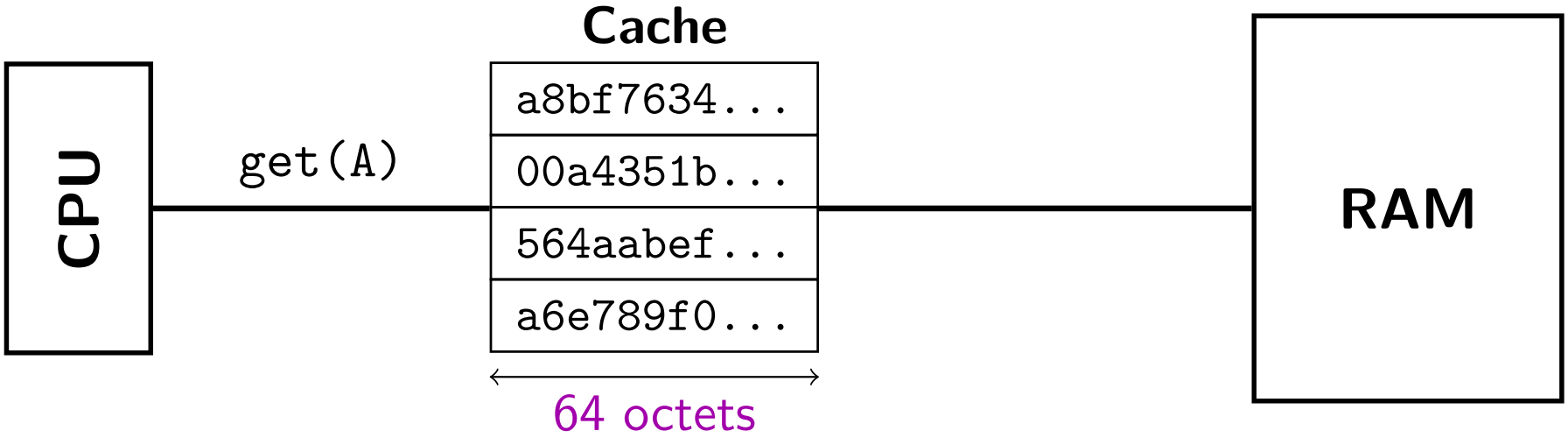
- Objectif : trouver rapidement une donnée demandée par le CPU



`A = 000000000000 00000001101100001101000100001110101001101001 10 000011`

Fonctionnement d'un cache direct : cache hit

- Objectif : trouver rapidement une donnée demandée par le CPU

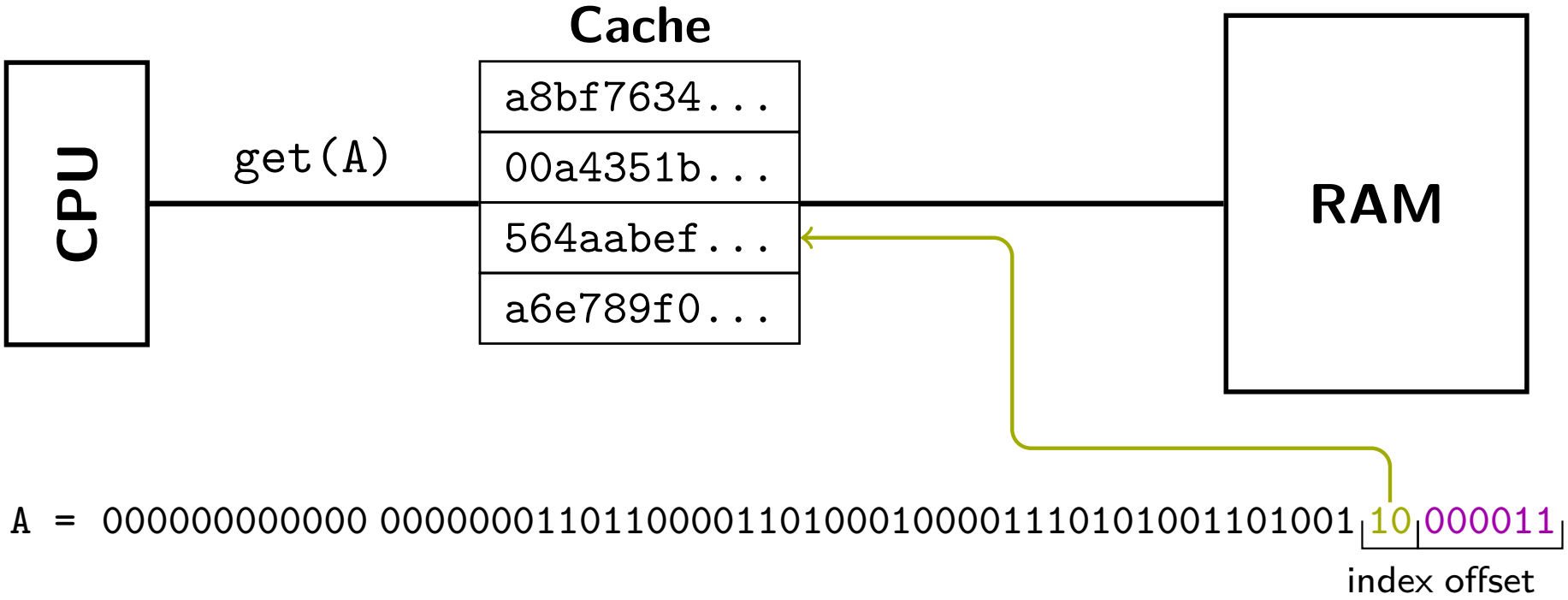


A = 000000000000 00000001101100011010001000011101010011010010 10 000011

offset

Fonctionnement d'un cache direct : cache hit

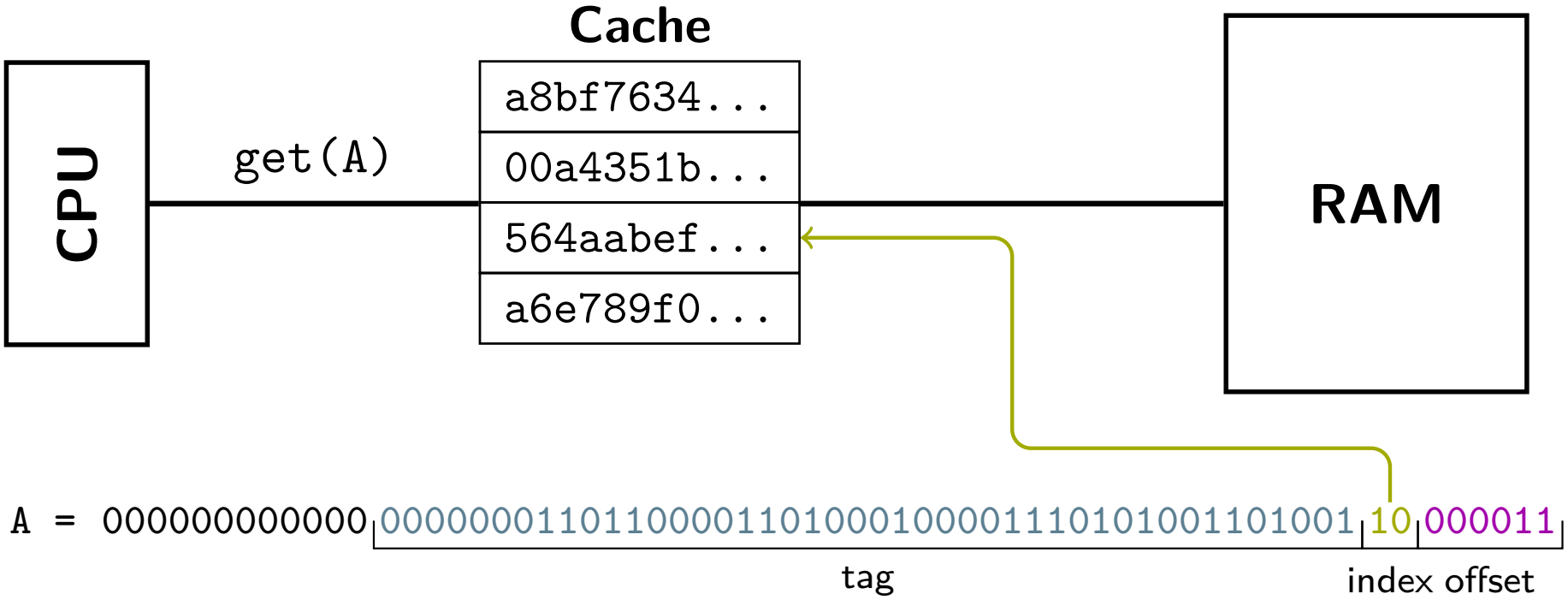
- Objectif : trouver rapidement une donnée demandée par le CPU



- Chaque ligne est repérée par un index
  - L'index indique l'entrée du cache susceptible de contenir la ligne

Fonctionnement d'un cache direct : cache hit

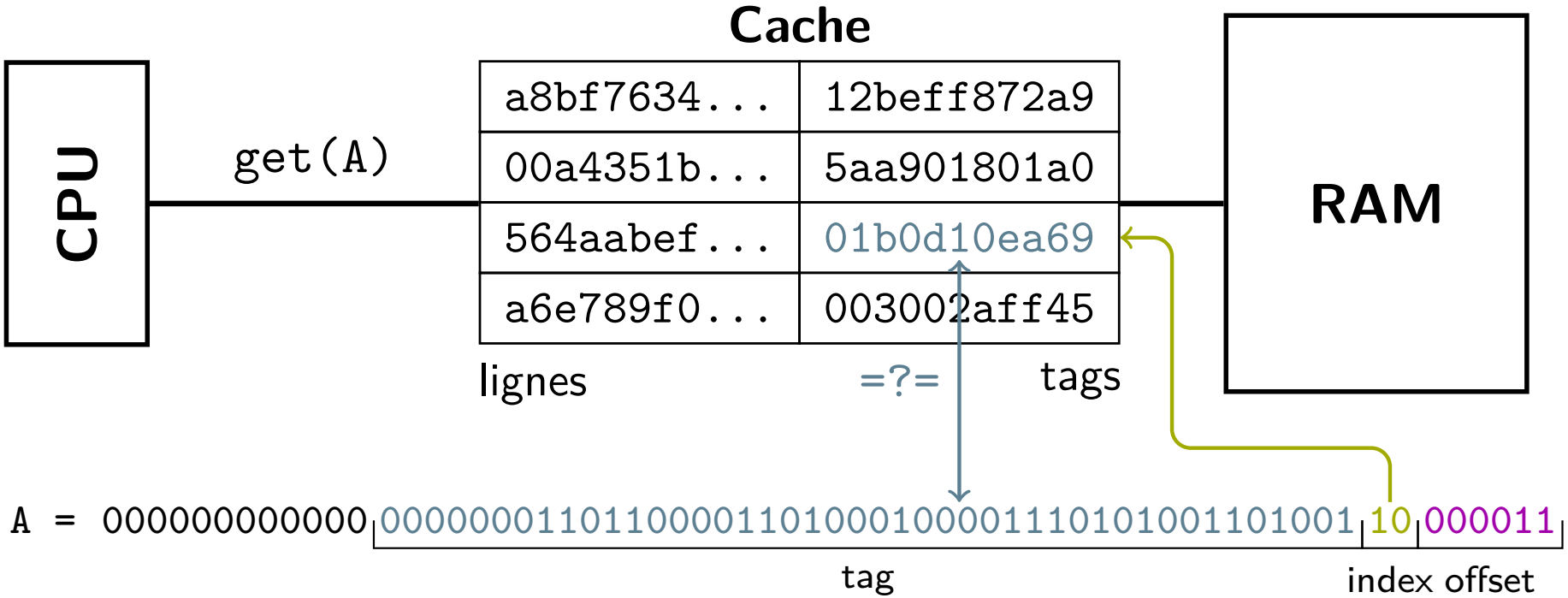
- Objectif : trouver rapidement une donnée demandée par le CPU



- Chaque ligne est repérée par un index et un tag
  - L'index indique l'entrée du cache susceptible de contenir la ligne

Fonctionnement d'un cache direct : cache hit

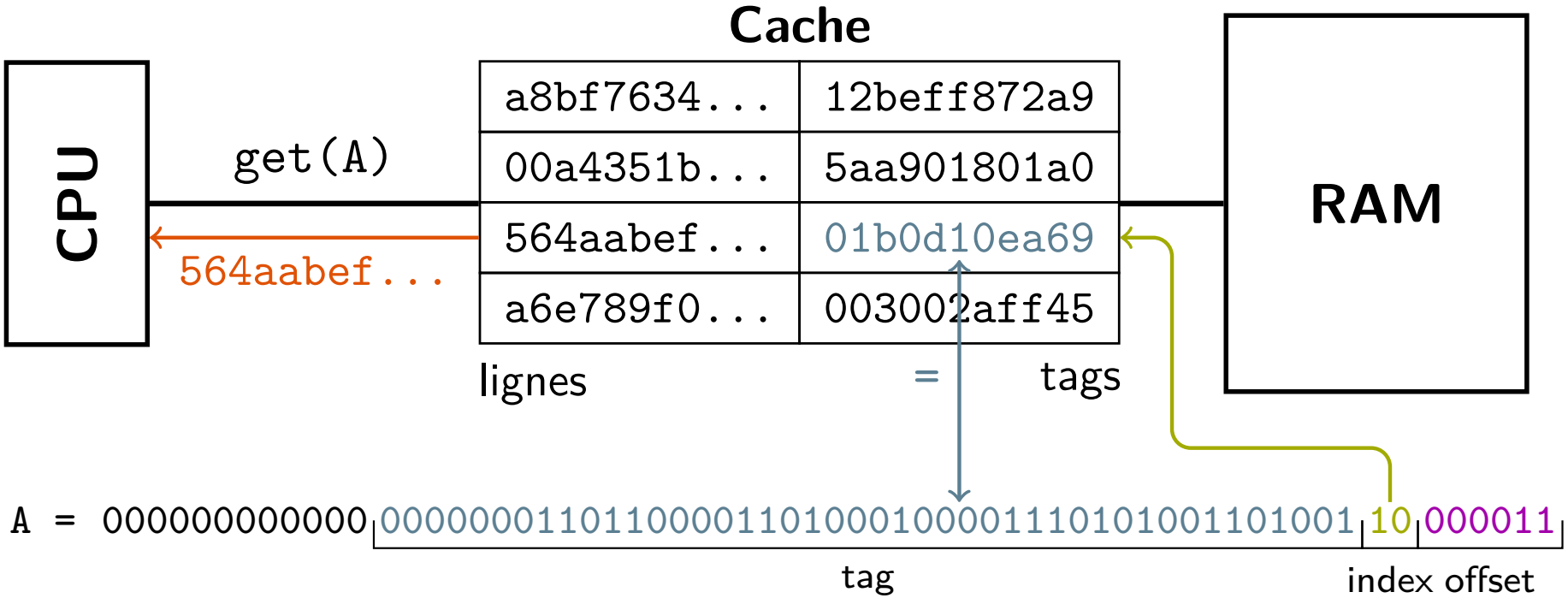
- Objectif : trouver rapidement une donnée demandée par le CPU



- Chaque ligne est repérée par un index et un tag
  - L'index indique l'entrée du cache susceptible de contenir la ligne
  - Le tag est stocké avec la ligne dans le cache correspondante
  - Si les tags demandé et stocké sont les mêmes

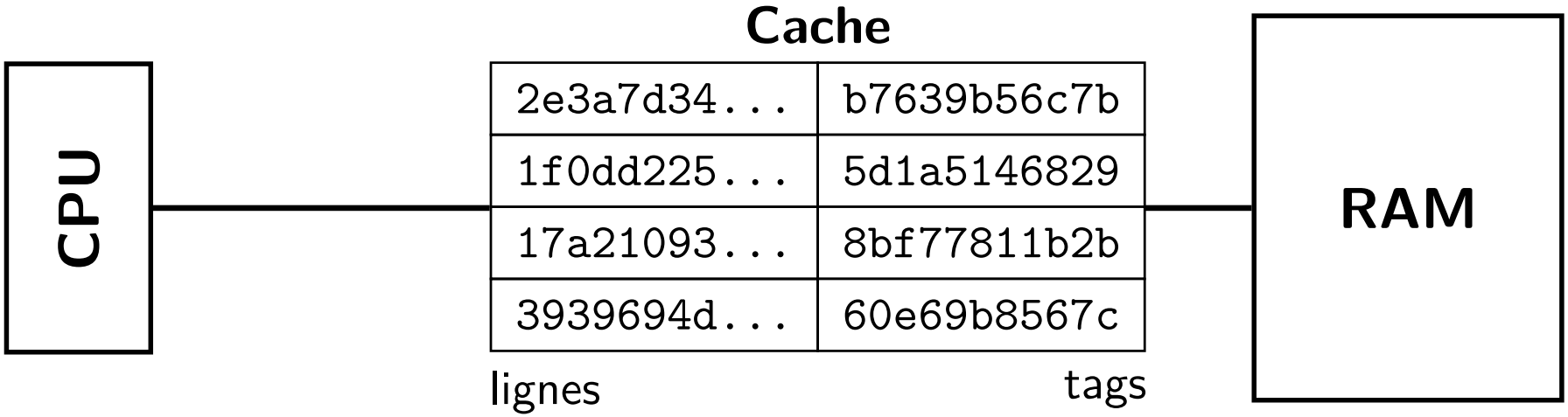
Fonctionnement d'un cache direct : cache hit

- Objectif : trouver rapidement une donnée demandée par le CPU

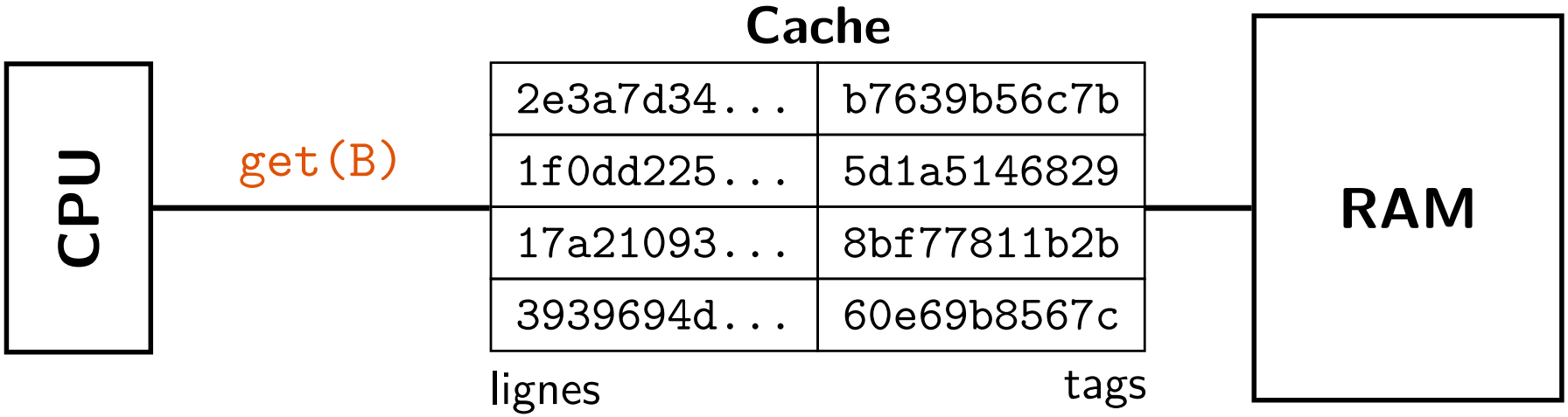


- Chaque ligne est repérée par un index et un tag
  - L'index indique l'entrée du cache susceptible de contenir la ligne
  - Le tag est stocké avec la ligne dans le cache correspondante
  - Si les tags demandé et stocké sont les mêmes → cache hit

Fonctionnement d'un cache direct : cache miss

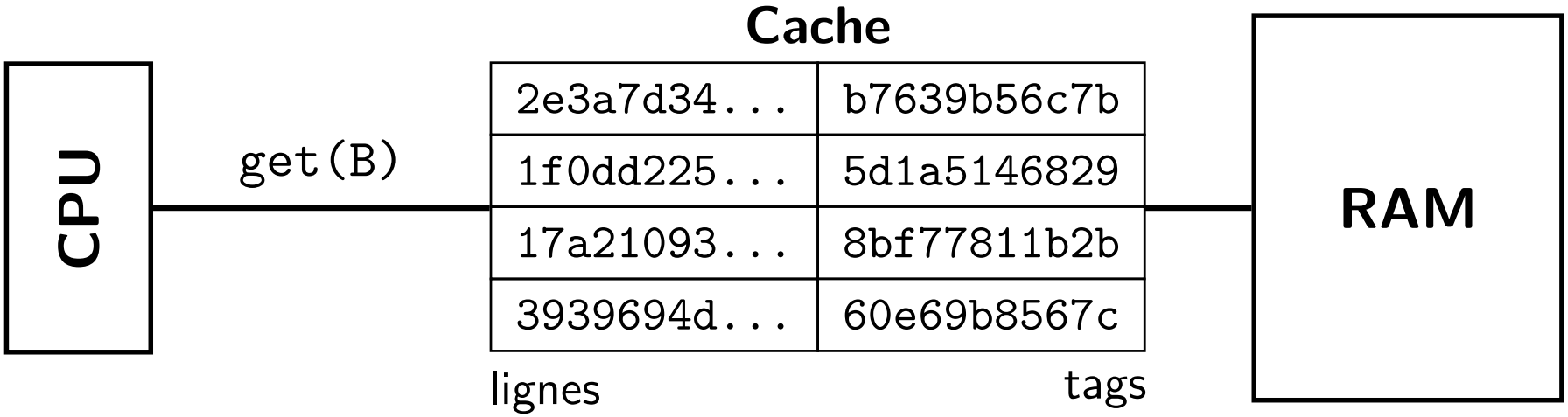


Fonctionnement d'un cache direct : cache miss



B = 000000000000 10110100010100011101111110111010011000010100 10 110110

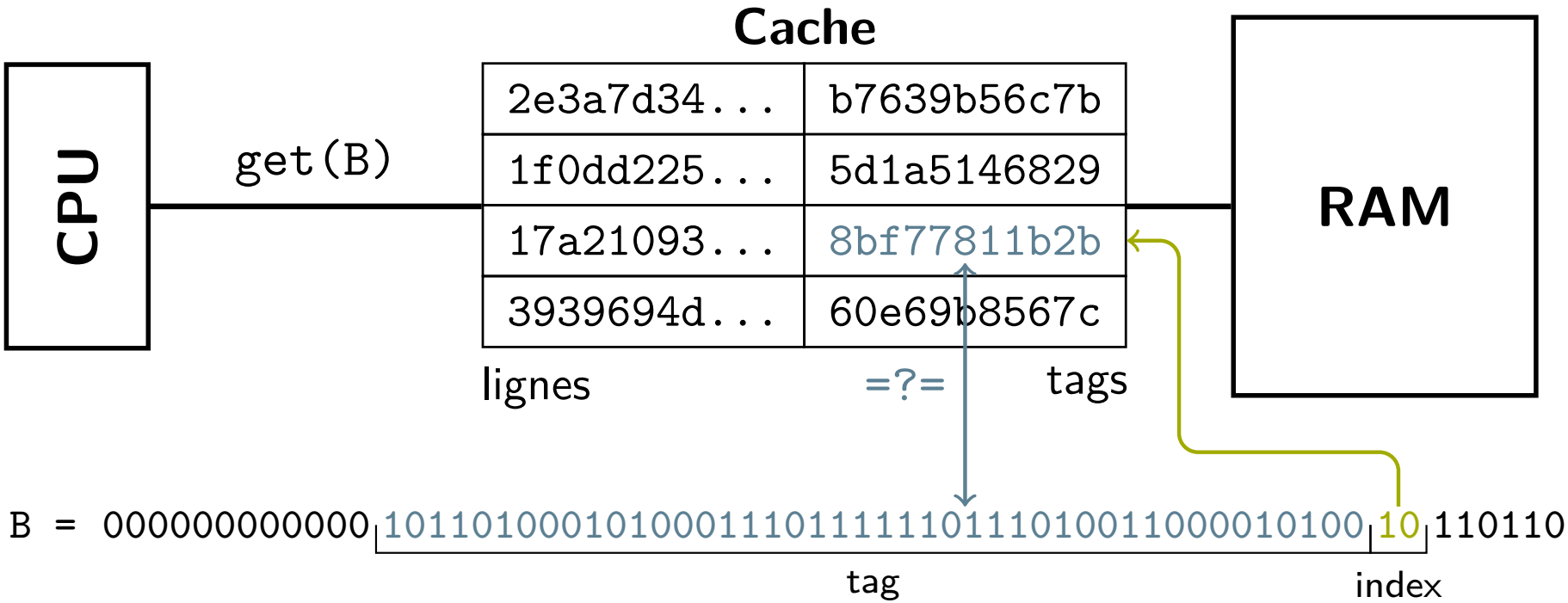
Fonctionnement d'un cache direct : cache miss



B = 000000000000 101101000101000111011111101110100110000010100 10 110110

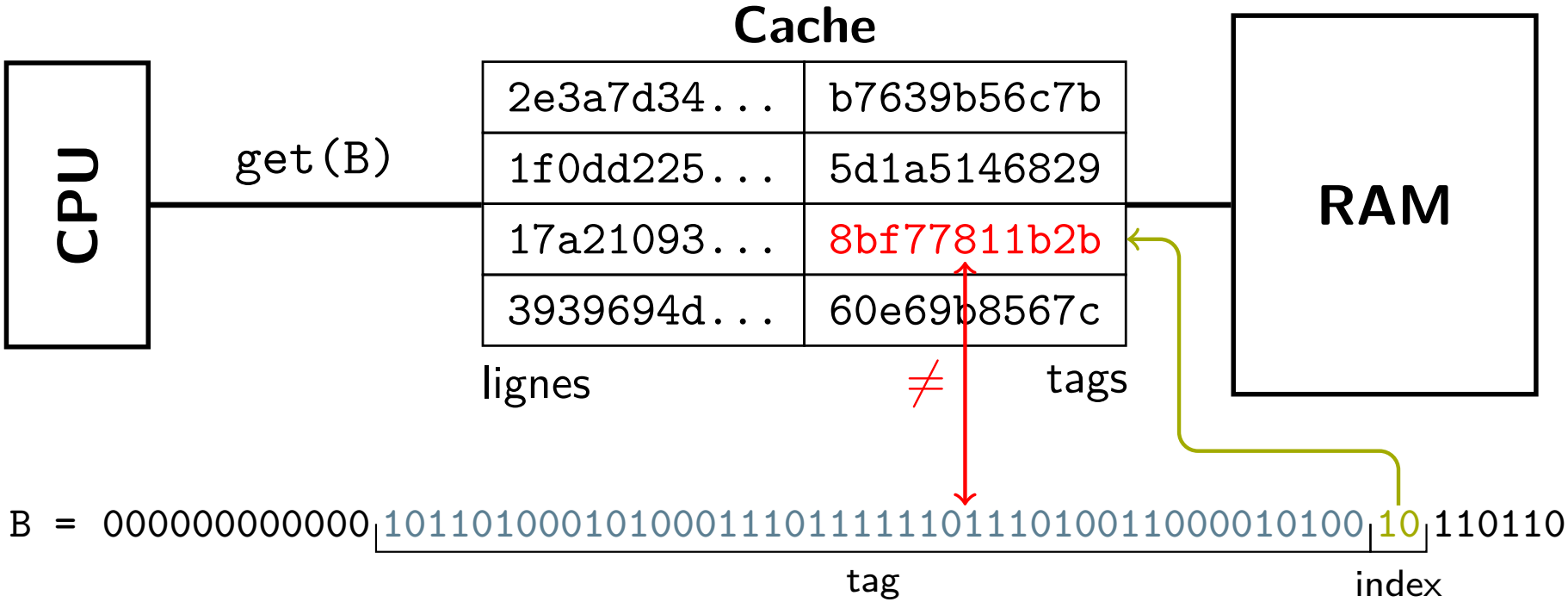
tag index

Fonctionnement d'un cache direct : cache miss



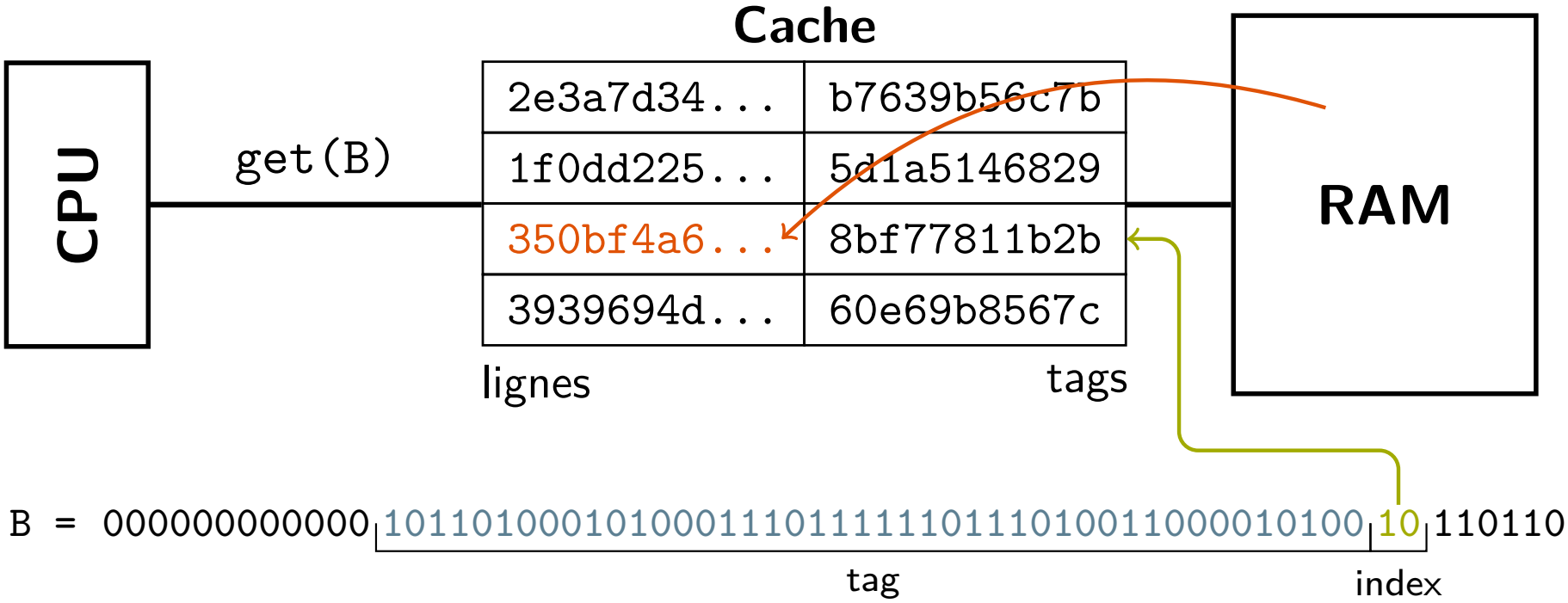
- Si les tags demandé et stocké sont différents

Fonctionnement d'un cache direct : cache miss



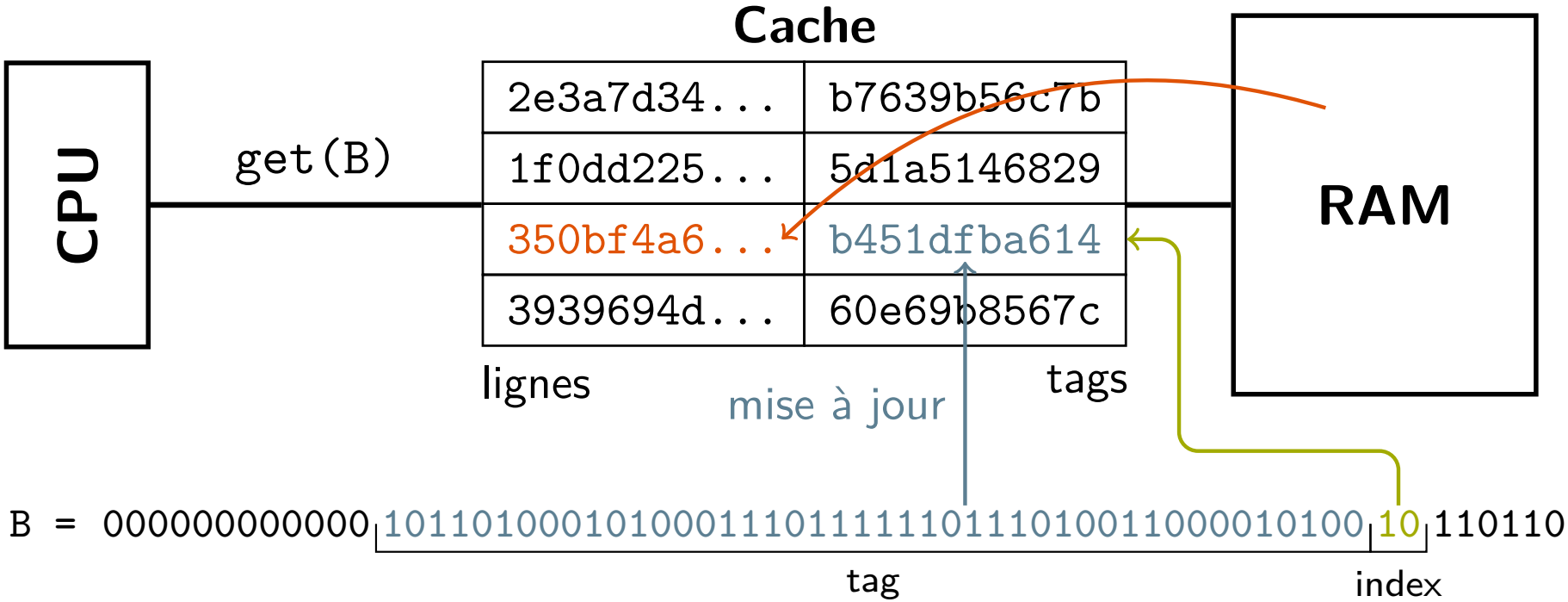
- Si les tags demandé et stocké sont différents → cache miss

Fonctionnement d'un cache direct : cache miss



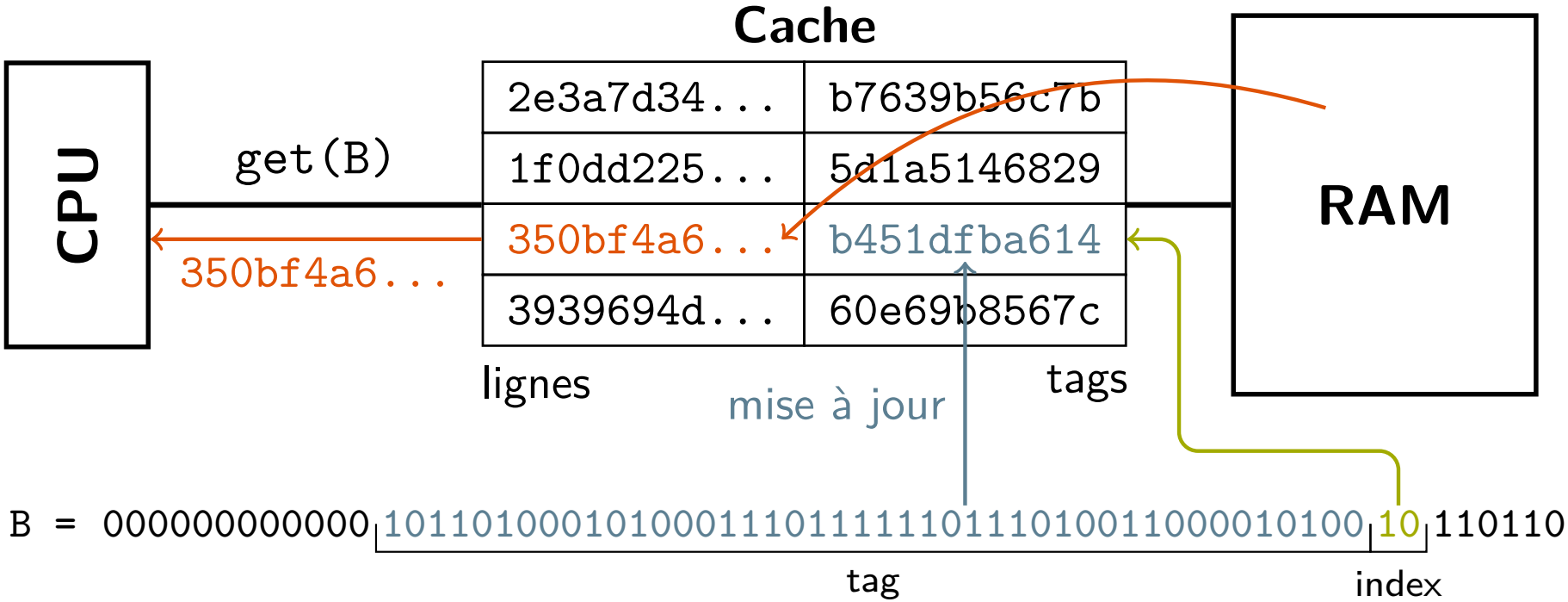
- Si les tags demandé et stocké sont différents → cache miss
  - La nouvelle ligne est chargée depuis la mémoire principale

Fonctionnement d'un cache direct : cache miss



- Si les tags demandé et stocké sont différents → cache miss
  - La nouvelle ligne est chargée depuis la mémoire principale
  - Le tag demandé est chargé dans la mémoire cache

Fonctionnement d'un cache direct : cache miss



- Si les tags demandé et stocké sont différents → cache miss
  - La nouvelle ligne est chargée depuis la mémoire principale
  - Le tag demandé est chargé dans la mémoire cache
  - La nouvelle ligne est envoyée au CPU

Cache direct et collisions d'adresse : exercice

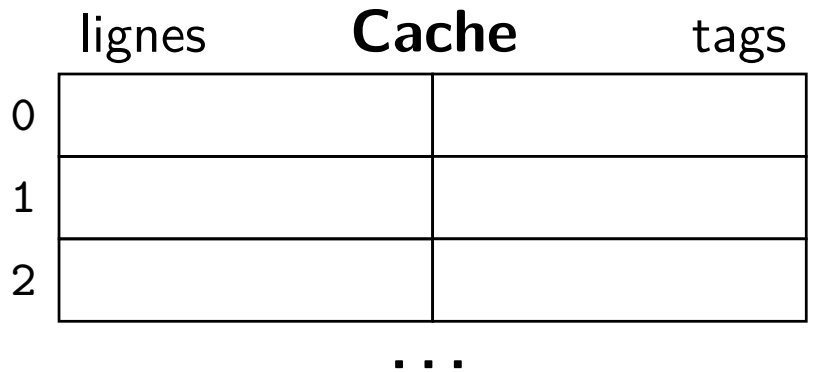
- On considère un cache direct de 32 Kio
  - Une ligne de cache fait 64 octets
- Quel est le *hit rate* du code suivant :

```
char *src = (char *) 0x18000;
char *dest = (char *) 0x10000;
char tmp;

for (i = 0; i < 10; i++) {
    tmp = scr[i];
    dest[i] = tmp;
}
```

Cache direct et collisions d'adresse : exercice

- On considère un cache direct de 32 Kio
  - Une ligne de cache fait 64 octets
- Quel est le *hit rate* du code suivant :

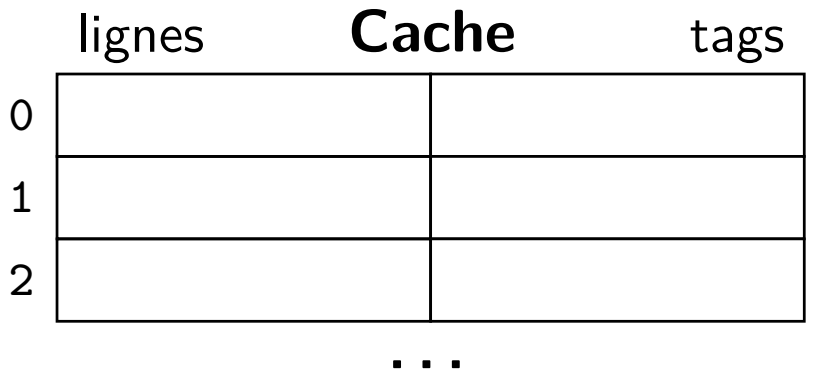
```
char *src = (char *) 0x18000;
char *dest = (char *) 0x10000;
char tmp;

for (i = 0; i < 10; i++) {
    tmp = scr[i];
    dest[i] = tmp;
}
```



Cache direct et collisions d'adresse : exercice

- On considère un cache direct de 32 Kio
  - Une ligne de cache fait 64 octets
- Quel est le *hit rate* du code suivant :

```
char *src = (char *) 0x18000;
char *dest = (char *) 0x10000;
char tmp;

for (i = 0; i < 10; i++) {
    tmp = scr[i];
    dest[i] = tmp;
}
```



A = 0x18000

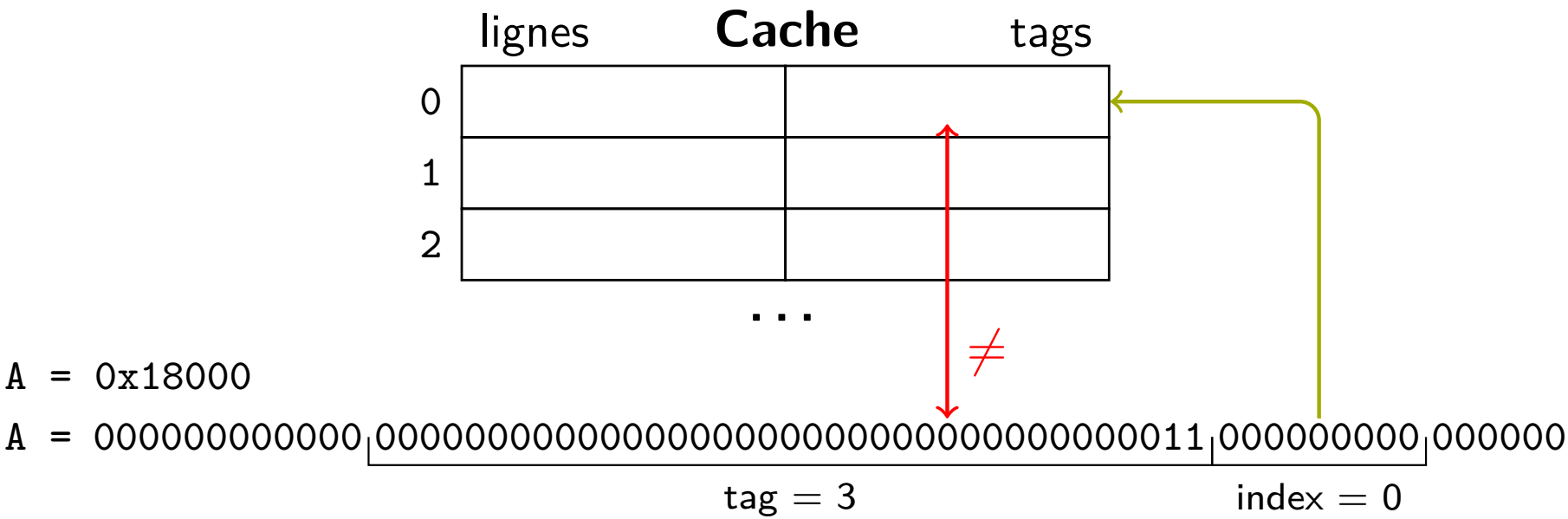
A = 000000000000 | 00000000000000000000000000000011 | 000000000 | 000000

tag = 3 index = 0

Cache direct et collisions d'adresse : exercice

- On considère un cache direct de 32 Kio
  - Une ligne de cache fait 64 octets
- Quel est le *hit rate* du code suivant :

```
char *src = (char *) 0x18000;
char *dest = (char *) 0x10000;
char tmp;

for (i = 0; i < 10; i++) {
    tmp = scr[i];
    dest[i] = tmp;
}
```

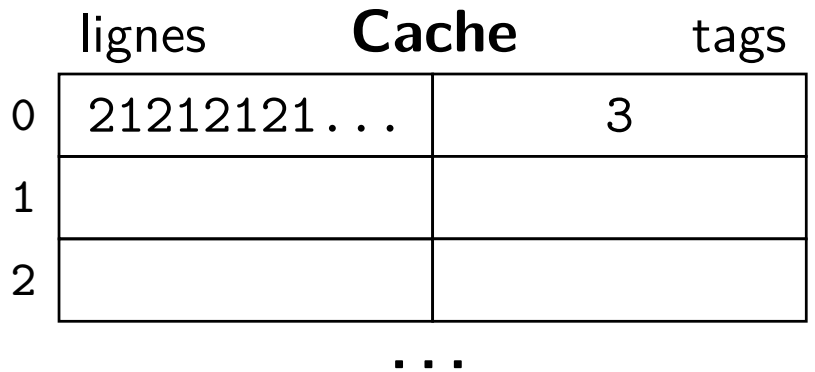


Cache direct et collisions d'adresse : exercice

- On considère un cache direct de 32 Kio
  - Une ligne de cache fait 64 octets
- Quel est le *hit rate* du code suivant :

```
char *src = (char *) 0x18000;
char *dest = (char *) 0x10000;
char tmp;

for (i = 0; i < 10; i++) {
    tmp = scr[i];
    dest[i] = tmp;
}
```

Nombre d'accès	1
Nombre de hit	0



A = 0x18000

A = 000000000000 | 00000000000000000000000000000011 | 000000000 | 000000

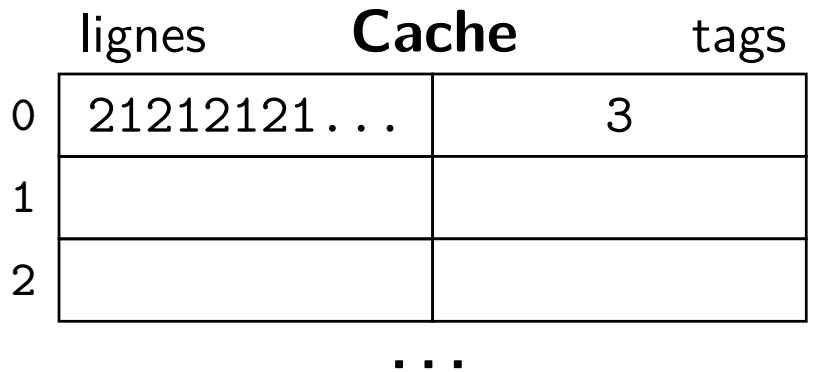
tag = 3 index = 0

Cache direct et collisions d'adresse : exercice

- On considère un cache direct de 32 Kio
  - Une ligne de cache fait 64 octets
- Quel est le *hit rate* du code suivant :

```
char *src = (char *) 0x18000;
char *dest = (char *) 0x10000;
char tmp;

for (i = 0; i < 10; i++) {
    tmp = scr[i];
    dest[i] = tmp;
}
```

Nombre d'accès	1
Nombre de hit	0



A = 0x10000

A = 000000000000 | 00000000000000000000000000000010 | 000000000 | 000000

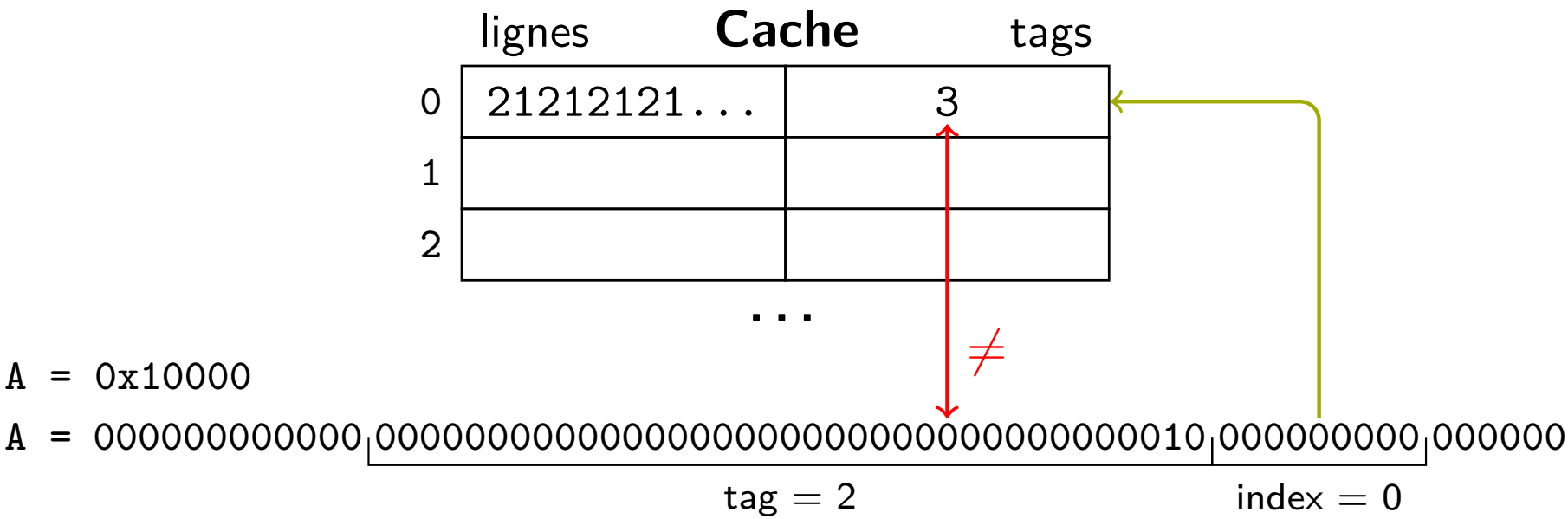
tag = 2 index = 0

Cache direct et collisions d'adresse : exercice

- On considère un cache direct de 32 Kio
  - Une ligne de cache fait 64 octets
- Quel est le *hit rate* du code suivant :

```
char *src = (char *) 0x18000;
char *dest = (char *) 0x10000;
char tmp;

for (i = 0; i < 10; i++) {
    tmp = scr[i];
    dest[i] = tmp;
}
```

Nombre d'accès	1
Nombre de hit	0

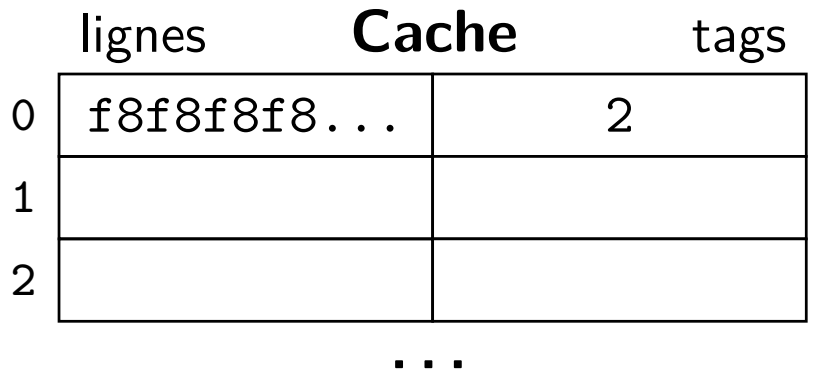


Cache direct et collisions d'adresse : exercice

- On considère un cache direct de 32 Kio
  - Une ligne de cache fait 64 octets
- Quel est le *hit rate* du code suivant :

```
char *src = (char *) 0x18000;
char *dest = (char *) 0x10000;
char tmp;

for (i = 0; i < 10; i++) {
    tmp = scr[i];
    dest[i] = tmp;
}
```

Nombre d'accès	2
Nombre de hit	0



A = 0x10000

A = 000000000000 | 00000000000000000000000000000010 | 000000000 | 000000

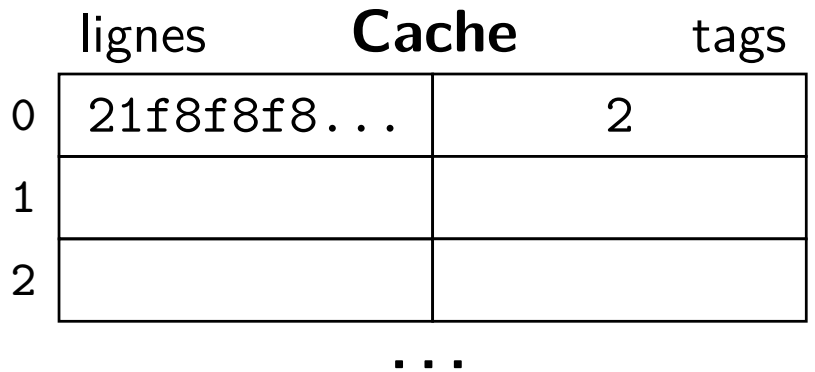
tag = 2 index = 0

Cache direct et collisions d'adresse : exercice

- On considère un cache direct de 32 Kio
  - Une ligne de cache fait 64 octets
- Quel est le *hit rate* du code suivant :

```
char *src = (char *) 0x18000;
char *dest = (char *) 0x10000;
char tmp;

for (i = 0; i < 10; i++) {
    tmp = scr[i];
    dest[i] = tmp;
}
```

Nombre d'accès	2
Nombre de hit	0



A = 0x10000

A = 000000000000 | 00000000000000000000000000000010 | 000000000 | 000000

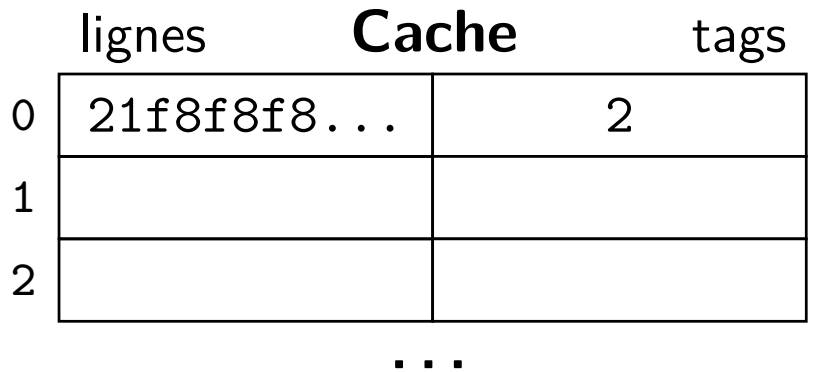
tag = 2 index = 0

Cache direct et collisions d'adresse : exercice

- On considère un cache direct de 32 Kio
  - Une ligne de cache fait 64 octets
- Quel est le *hit rate* du code suivant :

```
char *src = (char *) 0x18000;
char *dest = (char *) 0x10000;
char tmp;

for (i = 0; i < 10; i++) {
    tmp = scr[i];
    dest[i] = tmp;
}
```

Nombre d'accès	2
Nombre de hit	0



A = 0x18001

A = 000000000000 | 00000000000000000000000000000011 | 000000000 | 000001

tag = 3 index = 0

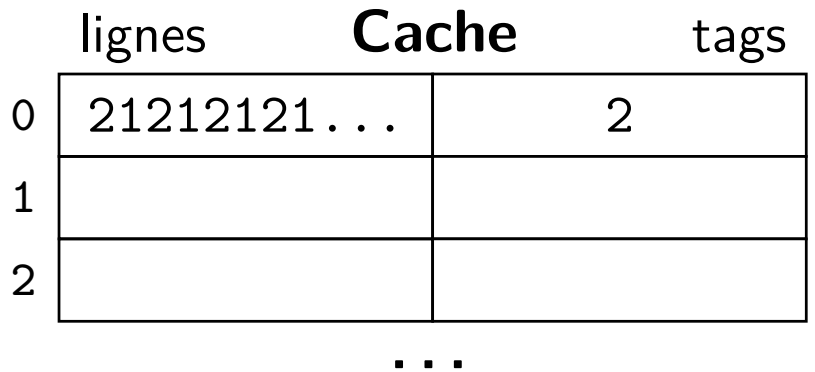
Cache direct et collisions d'adresse : exercice

- On considère un cache direct de 32 Kio
  - Une ligne de cache fait 64 octets
- Quel est le *hit rate* du code suivant :

```
char *src = (char *) 0x18000;
char *dest = (char *) 0x10000;
char tmp;

for (i = 0; i < 10; i++) {
    tmp = scr[i];
    dest[i] = tmp;
}
```

Nombre d'accès	20
Nombre de hit	0

- Hit rate = 0%



A = 0x10009

A = 000000000000 | 00000000000000000000000000000010 | 000000000 | 000009

tag = 2 index = 0

Cache direct : résumé

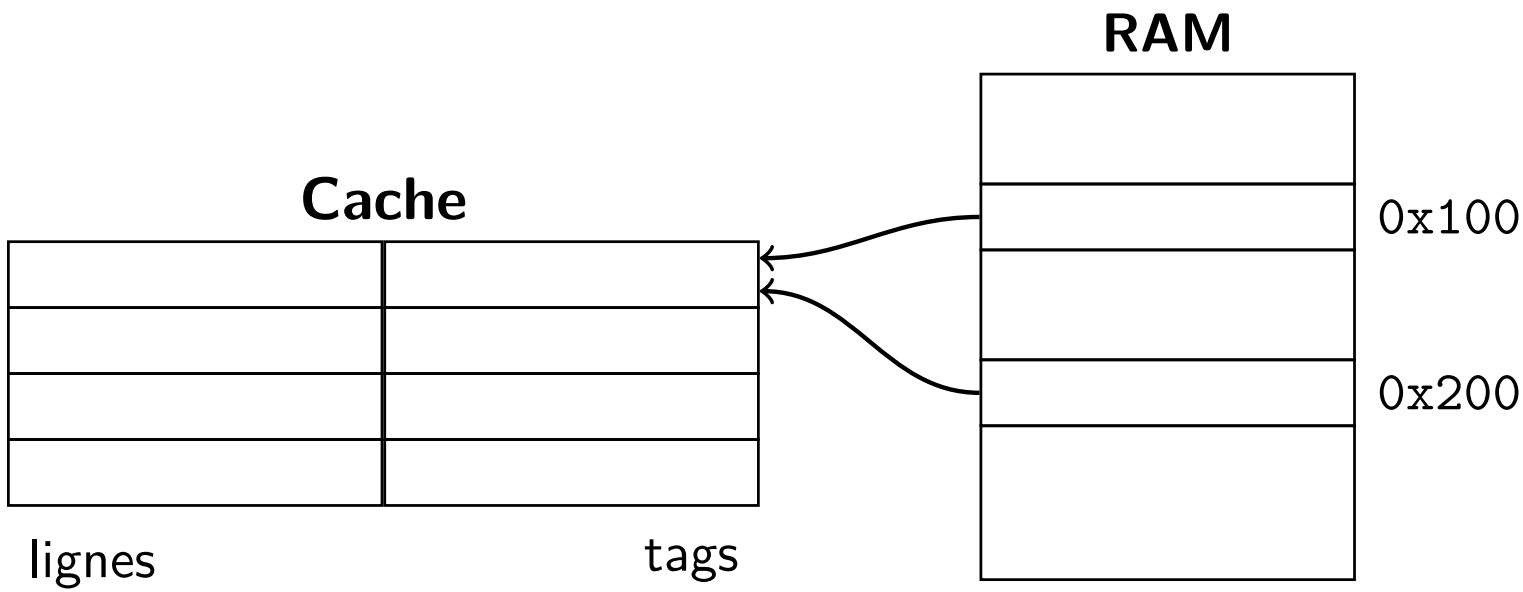
- La ligne de cache est l'unité d'échange entre le CPU et les mémoires
  - Une ligne de cache a toujours la même taille L sur un matériel donné
  - L'offset est la partie d'une adresse qui désigne la position d'un octet à l'intérieur d'une ligne : codé sur $log_2(L)$

- Un cache direct d'une taille de N lignes est un tableau à N entrées
  - L'index est la partie d'un adresse qui désigne l'entrée dans laquelle une ligne peut être stockée : codé sur $log_2(N)$ $\rightarrow$ dépend du cache considéré
  - Toute adresse correspond à exactement un emplacement de ligne dans un cache direct / un emplacement du cache correspond à plusieurs adresses $\rightarrow$ possibilité de collision

- Un cache direct stocke le tag associé à chaque ligne stockée
  - Le tag est la partie de l'adresse qui n'est ni l'index, ni l'offset
  - La correspondance des tags entre l'adresse demandée et l'entrée correspondante du cache indique qu'il y a cache hit

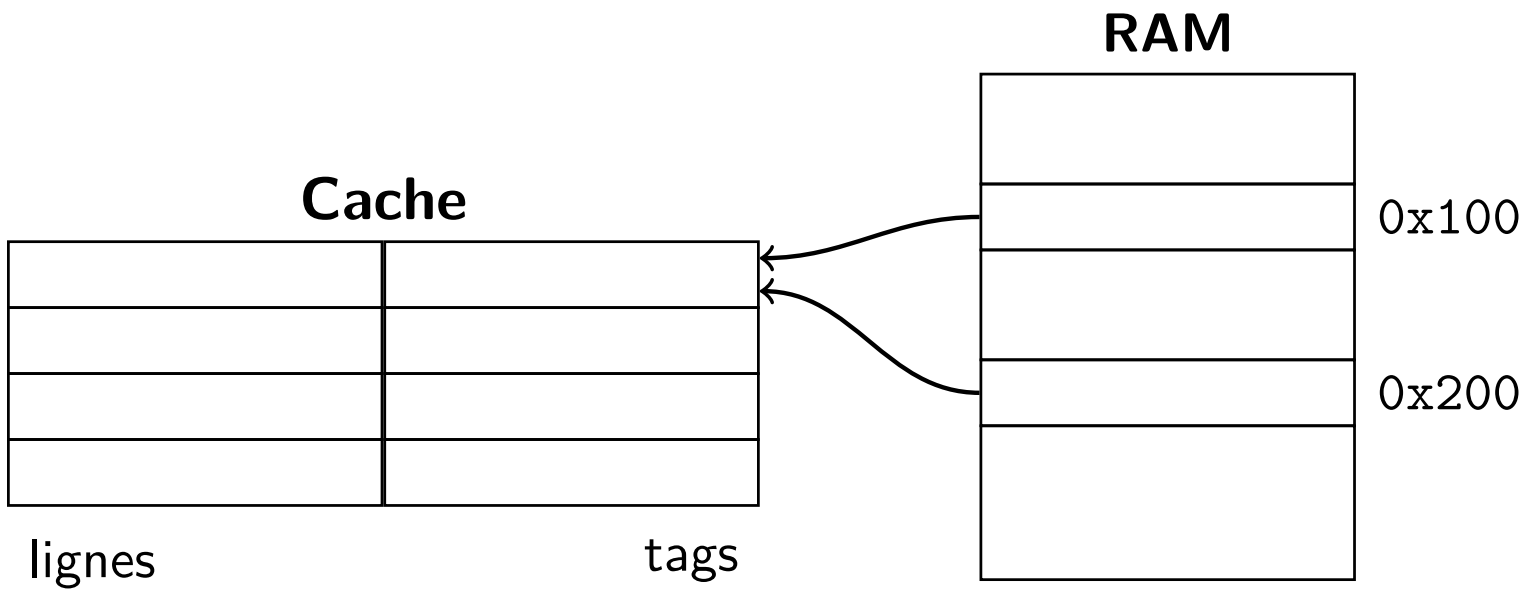
Collisions d'adresse et cache associatif

- Problème du cache direct : collisions d'adresses
  - Pour un cache direct de N lignes, toutes les adresses distantes de N lignes sont en conflit
  - Travailler simultanément sur des lignes en conflit $\rightarrow$ mauvais hit rate



Collisions d'adresse et cache associatif

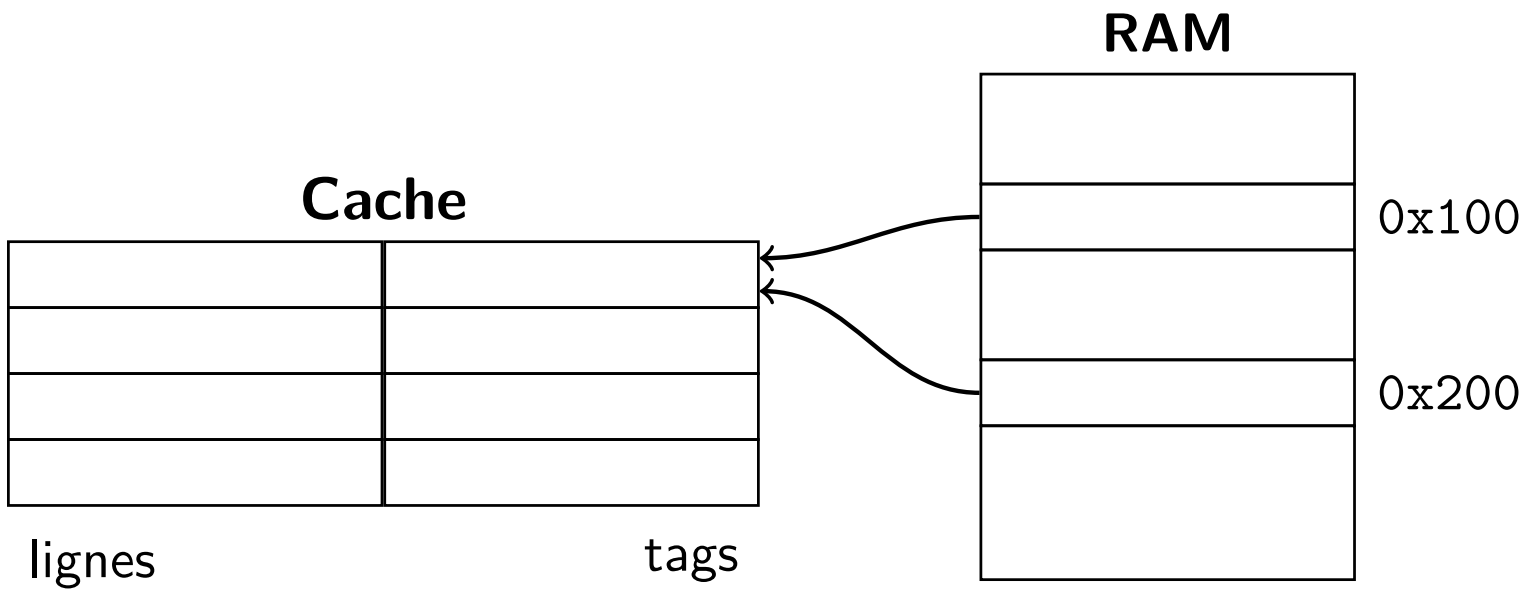
- Problème du cache direct : collisions d'adresses
  - Pour un cache direct de N lignes, toutes les adresses distantes de N lignes sont en conflit
  - Travailler simultanément sur des lignes en conflit $\rightarrow$ mauvais hit rate



- Cache direct : 1 emplacement pour N lignes de même index
  - On ne peut pas stocker plus d'une ligne avec le même index

Collisions d'adresse et cache associatif

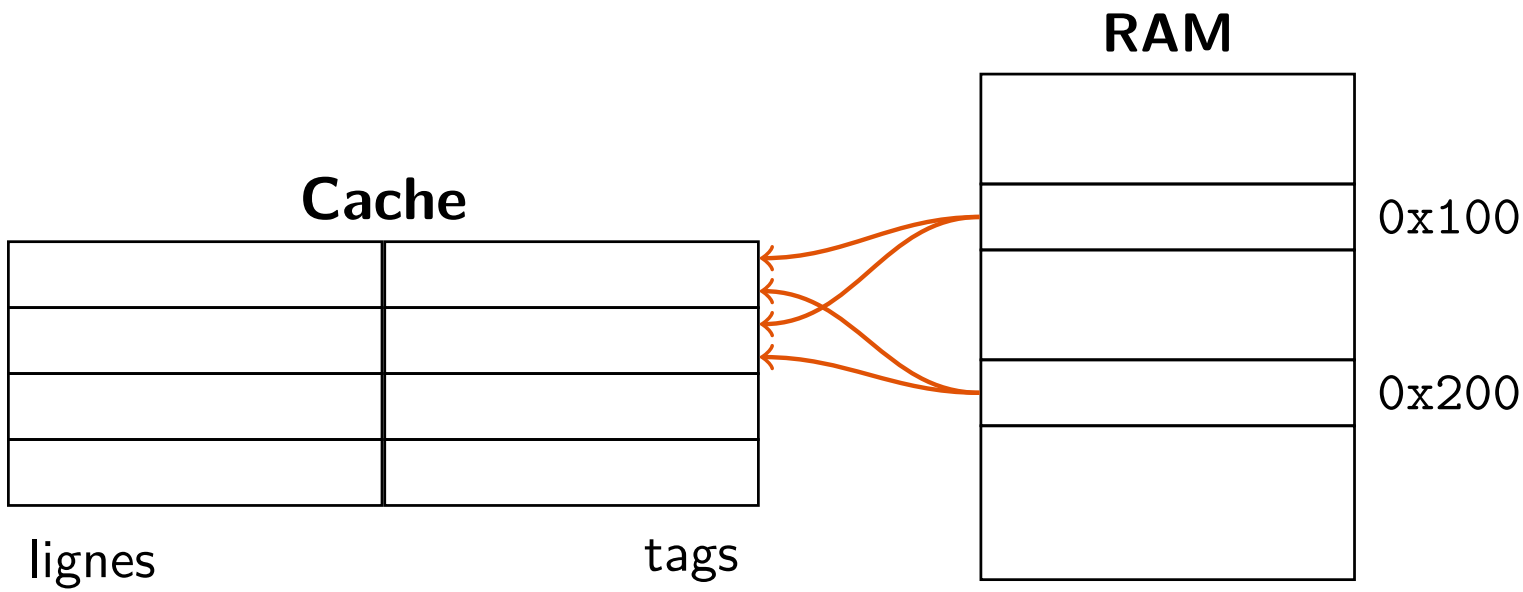
- Problème du cache direct : collisions d'adresses
  - Pour un cache direct de N lignes, toutes les adresses distantes de N lignes sont en conflit
  - Travailler simultanément sur des lignes en conflit $\rightarrow$ mauvais hit rate



- Cache direct : 1 emplacement pour N lignes de même index
  - On ne peut pas stocker plus d'une ligne avec le même index
  - On veut stocker M lignes avec le même index en même temps

Collisions d'adresse et cache associatif

- Problème du cache direct : collisions d'adresses
  - Pour un cache direct de N lignes, toutes les adresses distantes de N lignes sont en conflit
  - Travailler simultanément sur des lignes en conflit $\rightarrow$ mauvais hit rate



- Cache direct : 1 emplacement pour N lignes de même index
  - On ne peut pas stocker plus d'une ligne avec le même index
- Cache M-associatif : M emplacements pour N lignes de même index
  - On stocke M lignes avec le même index en même temps

Fonctionnement d'un cache associatif : cache hit

Cache 2-associative

1f621...	b4494
f92d8...	e1840
14578...	1850f
94ad1...	3ec24
161e4...	19e5b
3ee0e...	0a597
ec7d2...	1620c
20a3f...	d0f86

lignes tags

Fonctionnement d'un cache associatif : cache hit

- Un cache associatif est composé d'ensembles (*set*) d'emplacement

Cache 2-associative

1f621...	b4494
f92d8...	e1840
14578...	1850f
94ad1...	3ec24
161e4...	19e5b
3ee0e...	0a597
ec7d2...	1620c
20a3f...	d0f86

lignes tags

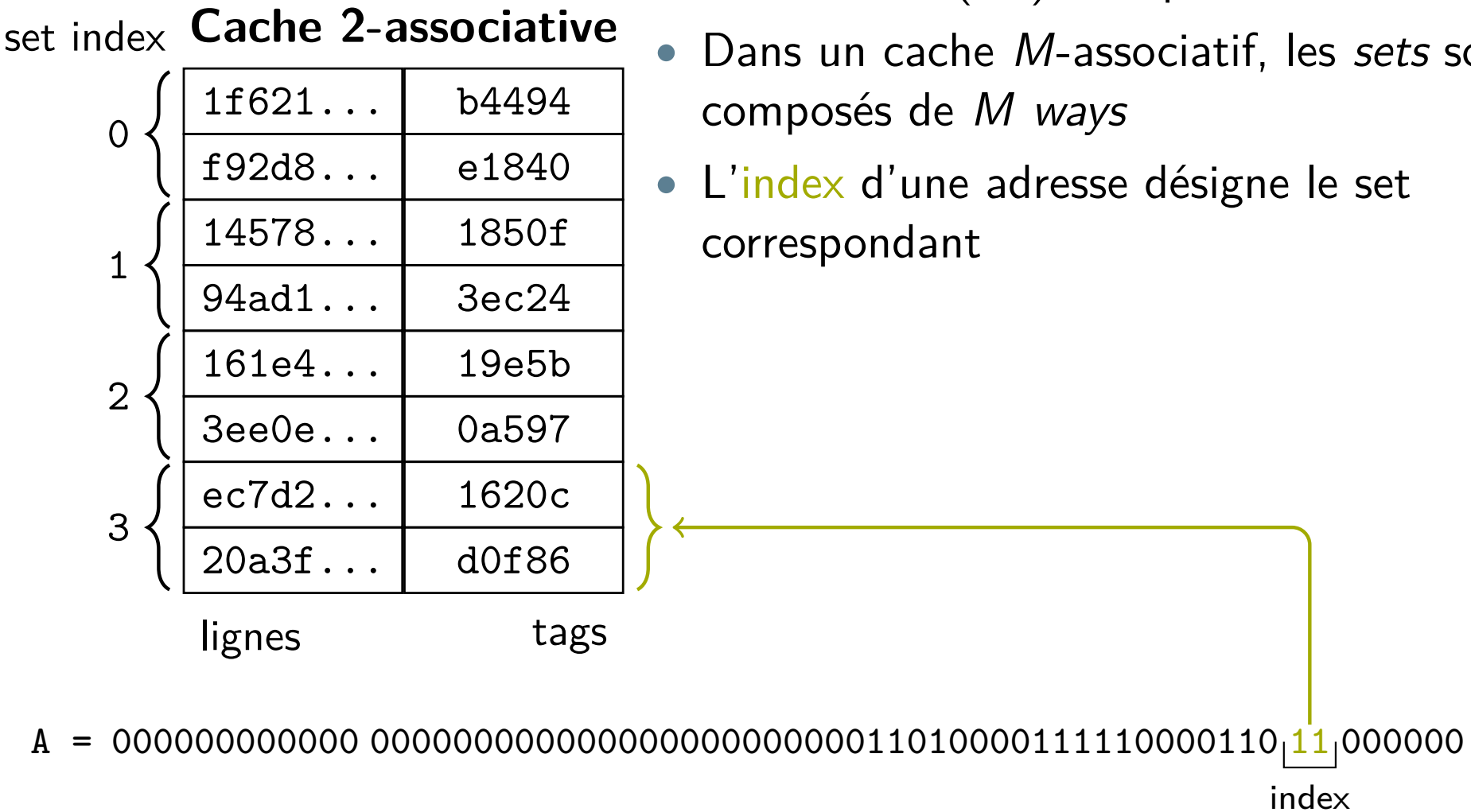
Fonctionnement d'un cache associatif : cache hit

Cache 2-associative

lignes	tags
1f621...	b4494
f92d8...	e1840
14578...	1850f
94ad1...	3ec24
161e4...	19e5b
3ee0e...	0a597
ec7d2...	1620c
20a3f...	d0f86

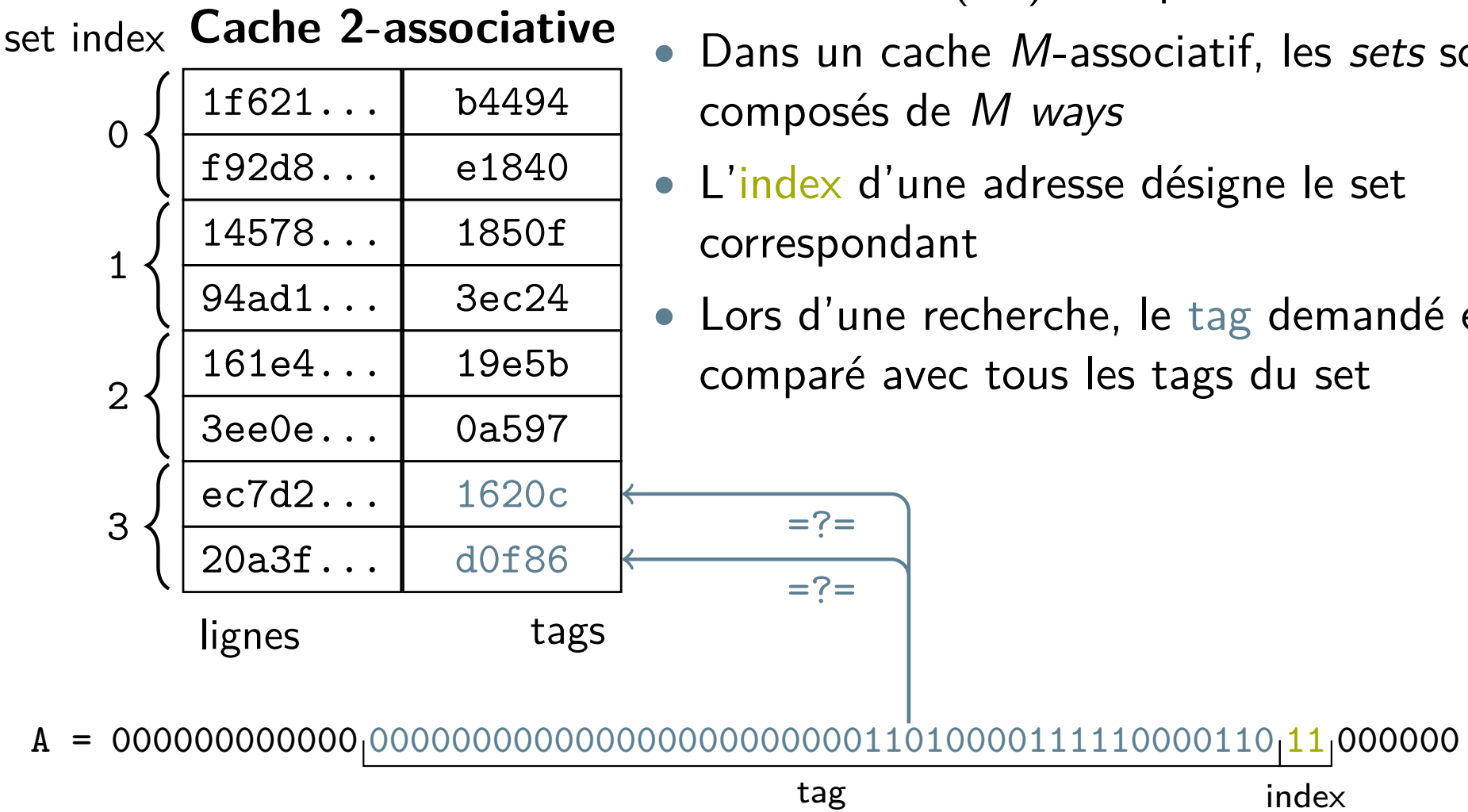
- Un cache associatif est composé d'ensembles (*set*) d'emplacement
- Dans un cache M-associatif, les *sets* sont composés de M *ways*

Fonctionnement d'un cache associatif : cache hit



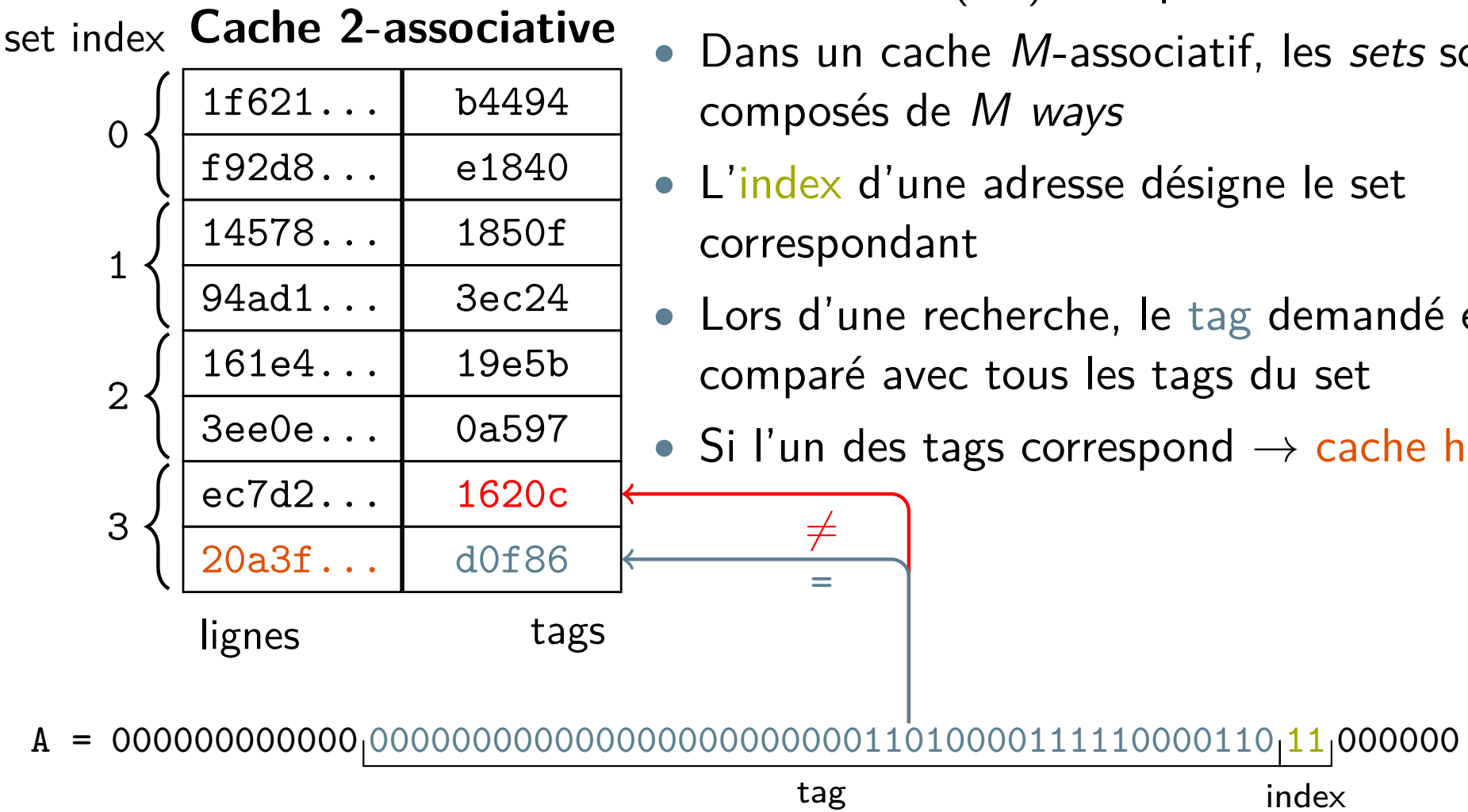
- Un cache associatif est composé d'ensembles (*set*) d'emplacement
- Dans un cache *M*-associatif, les *sets* sont composés de *M ways*
- L'index d'une adresse désigne le set correspondant

Fonctionnement d'un cache associatif : cache hit



- Un cache associatif est composé d'ensembles (*set*) d'emplacement
- Dans un cache *M*-associatif, les *sets* sont composés de *M ways*
- L'index d'une adresse désigne le set correspondant
- Lors d'une recherche, le tag demandé est comparé avec tous les tags du set

Fonctionnement d'un cache associatif : cache hit



- Un cache associatif est composé d'ensembles (*set*) d'emplacement
- Dans un cache *M*-associatif, les *sets* sont composés de *M ways*
- L'index d'une adresse désigne le set correspondant
- Lors d'une recherche, le tag demandé est comparé avec tous les tags du set
- Si l'un des tags correspond → cache hit

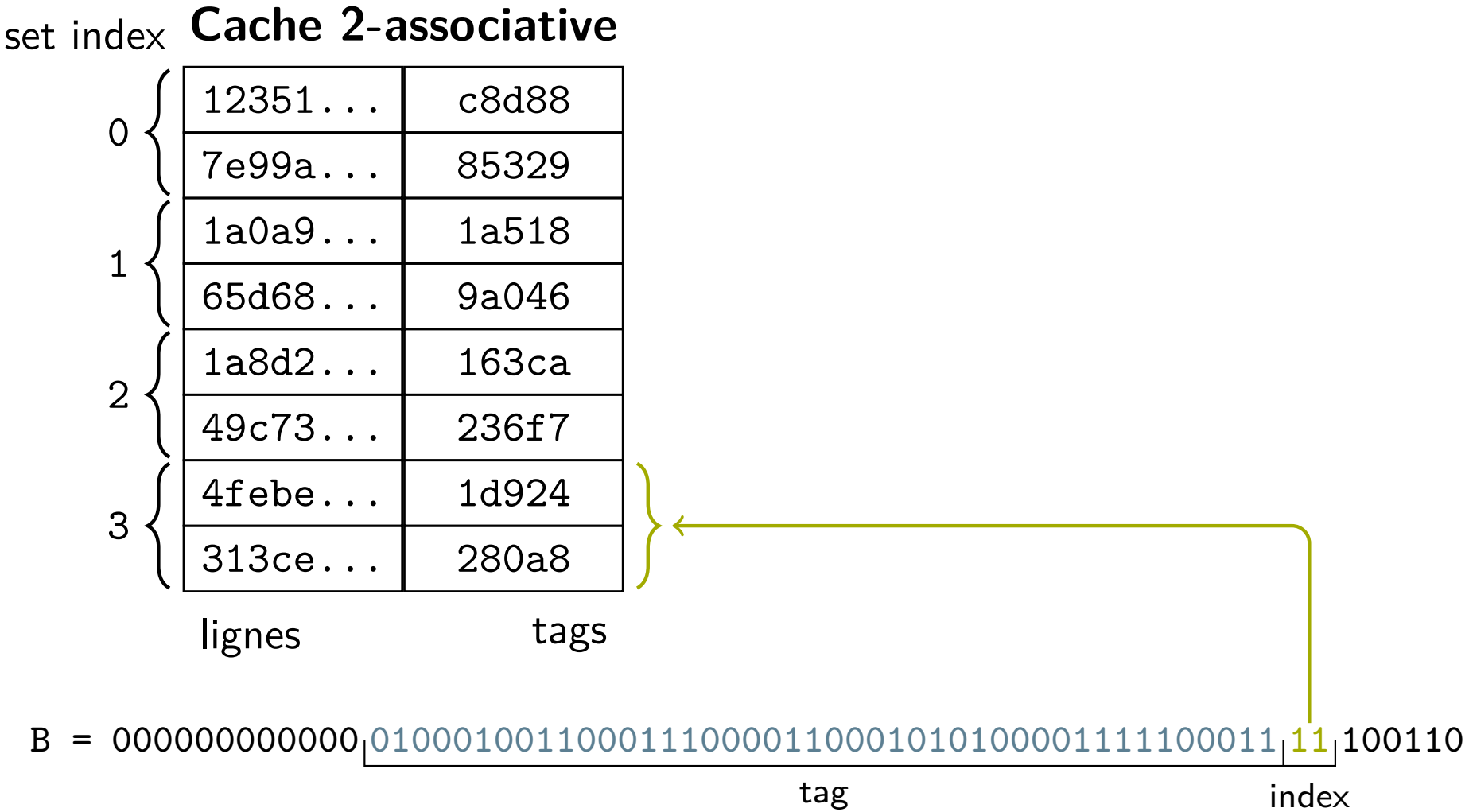
Fonctionnement d'un cache associatif : cache miss

Cache 2-associative

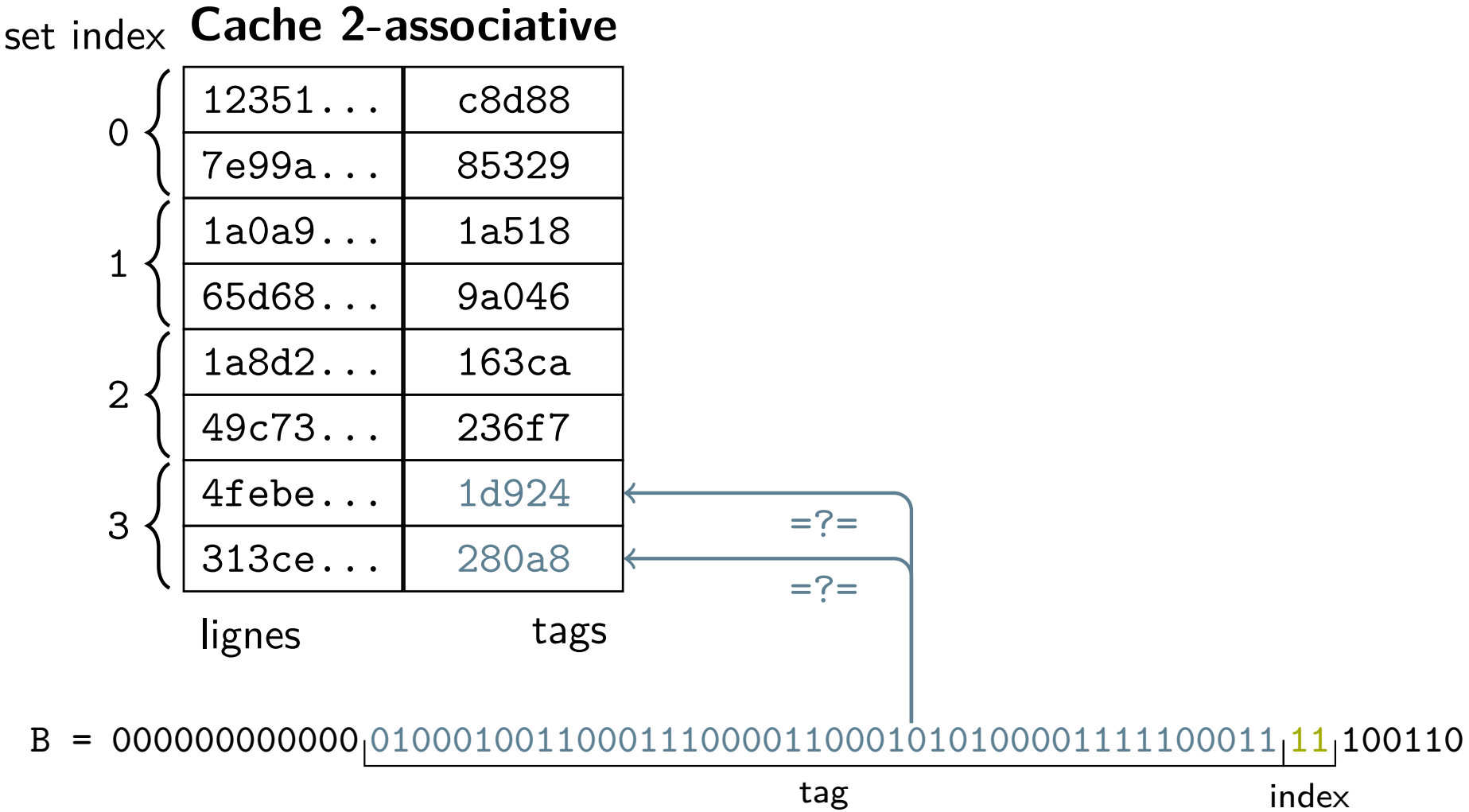
set index	lignes	tags
0	12351...	c8d88
	7e99a...	85329
1	1a0a9...	1a518
	65d68...	9a046
2	1a8d2...	163ca
	49c73...	236f7
3	4febe...	1d924
	313ce...	280a8

`B = 000000000000 01000100110001110000110001010100001111100011 11 100110`

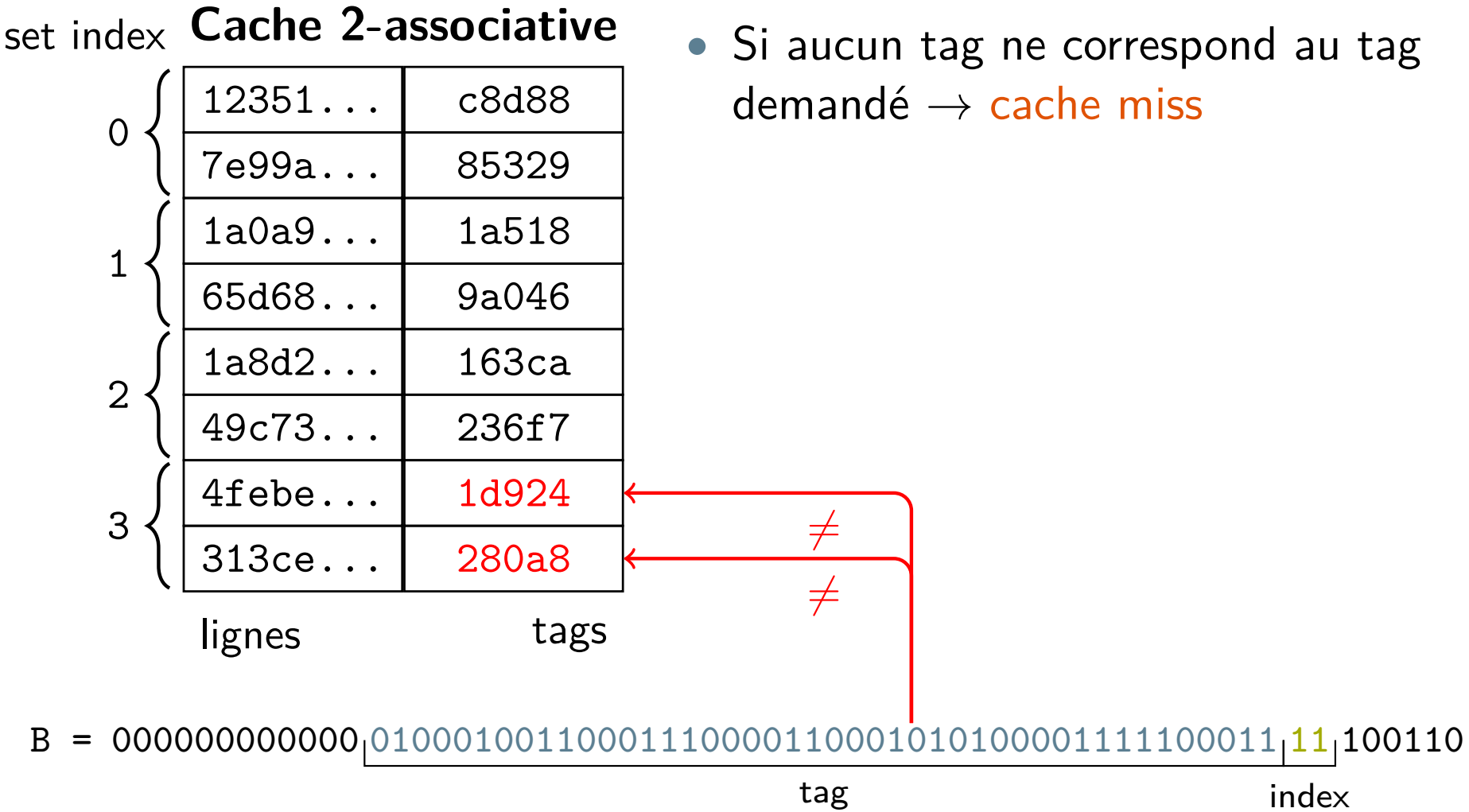
Fonctionnement d'un cache associatif : cache miss



Fonctionnement d'un cache associatif : cache miss

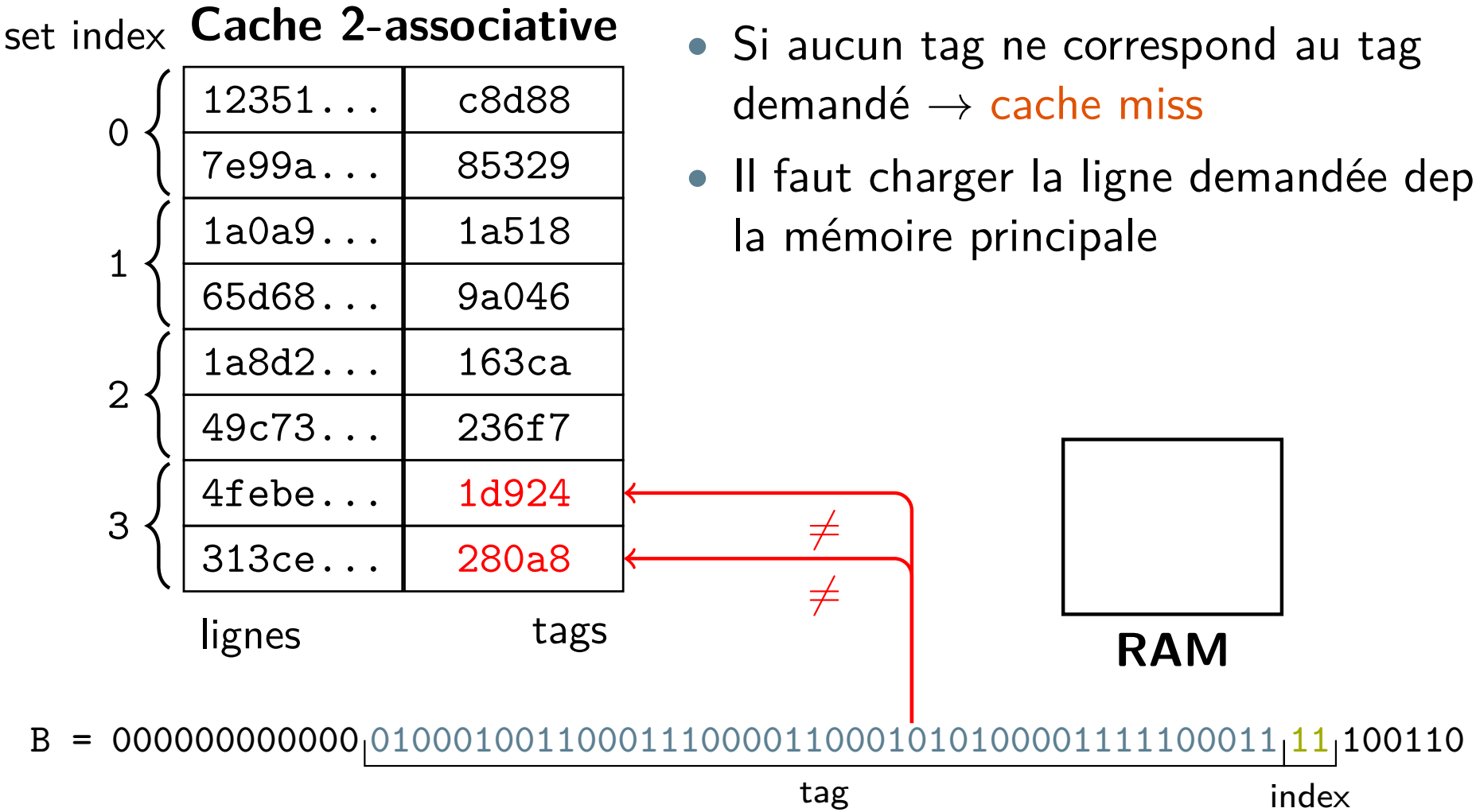


Fonctionnement d'un cache associatif : cache miss



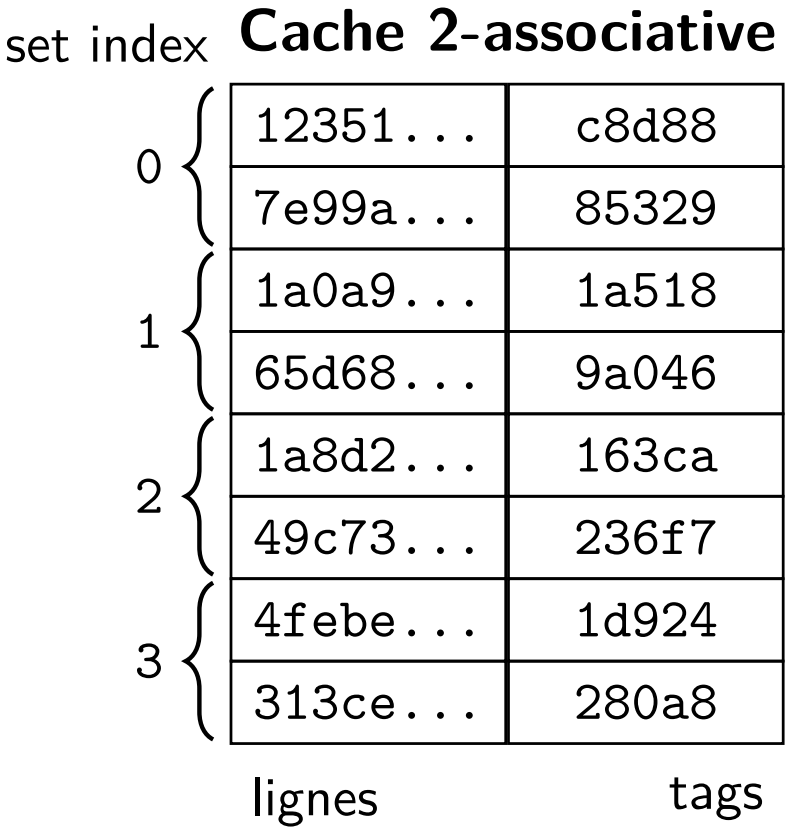
- Si aucun tag ne correspond au tag demandé → cache miss

Fonctionnement d'un cache associatif : cache miss

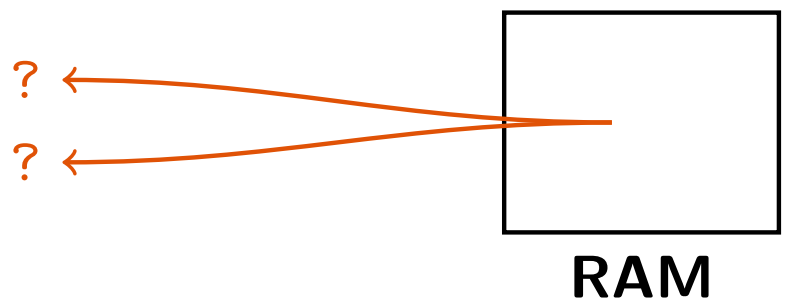


- Si aucun tag ne correspond au tag demandé → cache miss
- Il faut charger la ligne demandée depuis la mémoire principale

Fonctionnement d'un cache associatif : cache miss



- Si aucun tag ne correspond au tag demandé → cache miss
- Il faut charger la ligne demandée depuis la mémoire principale
- Dans quelle way stocker la nouvelle ligne → quelle ligne évincer ?



B = 000000000000 01000100110001110000110001010100001111100011 11 100110

tag index

Cache associatif : stratégie d'éviction

- Au moment d'un cache miss dans un cache *M*-associatif
  - La ligne demandée est chargée depuis la mémoire
  - Il faut décider dans quel *way* stocker la nouvelle ligne
  - Il faut décider quelle ligne évincer du cache

- Il existe plusieurs stratégies d'éviction
  - Least Recently Used $\rightarrow$ profite de la localité temporelle
  - Not Recently Used $\rightarrow$ approximation peu coûteuse de LRU
  - Most Recently Used $\rightarrow$ activée quand le *hit rate* devient trop faible

- La stratégie utilisée varie selon les implémentations
  - Les contrôleurs de cache sont des circuits complexes
  - Peuvent choisir la stratégie en fonction des accès observés

Cache associatif : résumé

- Un cache M-associatif d'une taille de N lignes est un tableau à N/M entrées indexées
  - Chaque entrée d'un cache M-associatif s'appelle un set
  - Chaque set d'un cache M-associatif contient M ways

- L'index est la partie d'un adresse qui désigne le set associé à l'adresse
  - Une ligne à l'adresse A peut être stockée dans n'importe quelle way du set associé à l'adresse A
  - Un cache M-associatif stocke jusqu'à M lignes avec le même index sans qu'il n'y ait de collision

- Lors d'un cache miss, le contrôleur doit charger la ligne demandée depuis la mémoire dans le set associé
  - Le contrôleur doit décider quelle ligne évincer du set
  - Il existe plusieurs stratégies d'éviction $\rightarrow$ objectif : maximiser le hit rate
  - Chaque contrôleur de cache M-associatif implémente une ou plusieurs de ces stratégies

Motifs d'accès mémoire

- Les programmes accèdent souvent à la mémoire de la même manière

```
for (i = 0; i < len; i++) {
    elm = array[i];
    ...
}
```

```
while (cursor != NULL) {
    ...
    cursor = cursor->next;
}
```

Motifs d'accès mémoire

- Les programmes accèdent souvent à la mémoire de la même manière

```
for (i = 0; i < len; i++) {
    elm = array[i];
    ...
}
```

- Au temps $t \rightarrow$ accès à A
- Au temps $t + 1 \rightarrow$ accès à $A + N$
- Au temps $t + 2 \rightarrow$ accès à $A + 2 \cdot N$

```
while (cursor != NULL) {
    ...
    cursor = cursor->next;
}
```

Motifs d'accès mémoire

- Les programmes accèdent souvent à la mémoire de la même manière

```
for (i = 0; i < len; i++) {
    elm = array[i];
    ...
}
```

- Au temps $t \rightarrow$ accès à A
- Au temps $t+1 \rightarrow$ accès à $A+N$
- Au temps $t+2 \rightarrow$ accès à $A+2 \cdot N$

```
while (cursor != NULL) {
    ...
    cursor = cursor->next;
}
```

- Au temps $t \rightarrow$ accès à A
- Au temps $t+1 \rightarrow$ accès à ${}^*A = B$
- Au temps $t+2 \rightarrow$ accès à ${}^*B = C$

Motifs d'accès mémoire

- Les programmes accèdent souvent à la mémoire de la même manière

```
for (i = 0; i < len; i++) {
    elm = array[i];
    ...
}
```

- Au temps $t \rightarrow$ accès à A
- Au temps $t+1 \rightarrow$ accès à $A+N$
- Au temps $t+2 \rightarrow$ accès à $A+2 \cdot N$

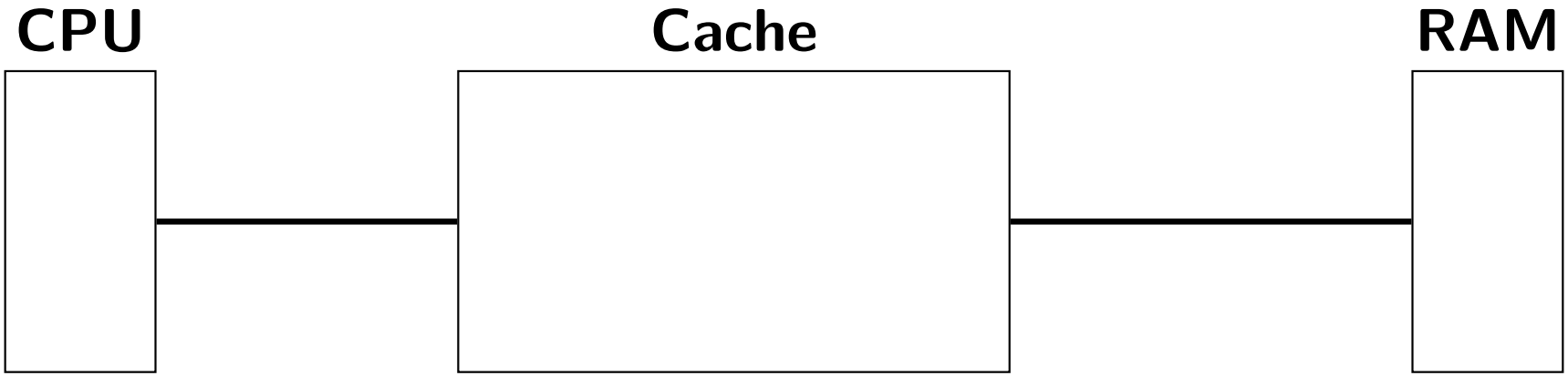
```
while (cursor != NULL) {
    ...
    cursor = cursor->next;
}
```

- Au temps $t \rightarrow$ accès à A
- Au temps $t+1 \rightarrow$ accès à ${}^*A = B$
- Au temps $t+2 \rightarrow$ accès à ${}^*B = C$

- La manière dont un processus accède à la mémoire s'appelle un motif d'accès

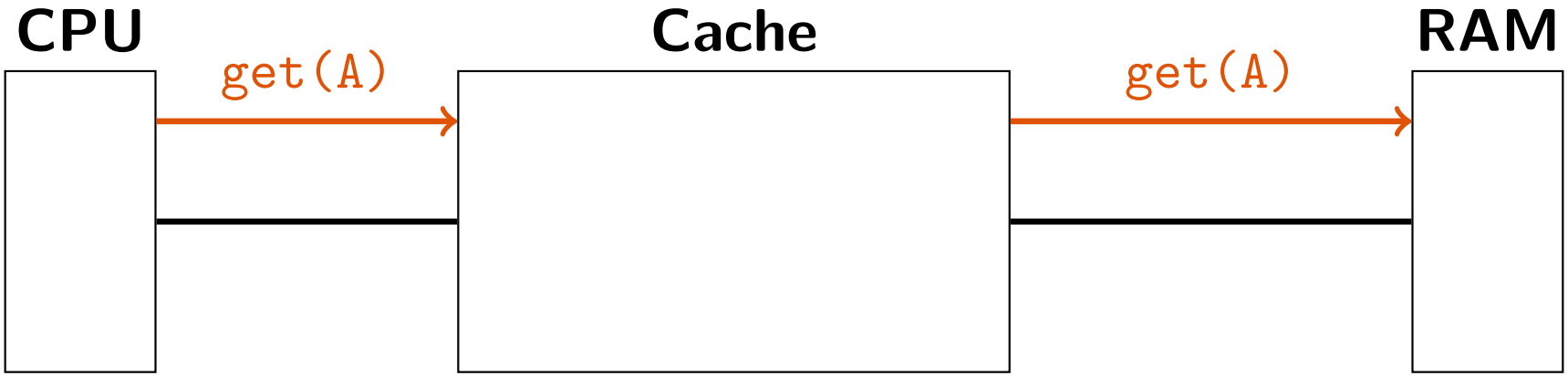
Prefetching automatique

- Les contrôleurs de cache tentent de prédire les accès futurs en détectant les motifs d'accès



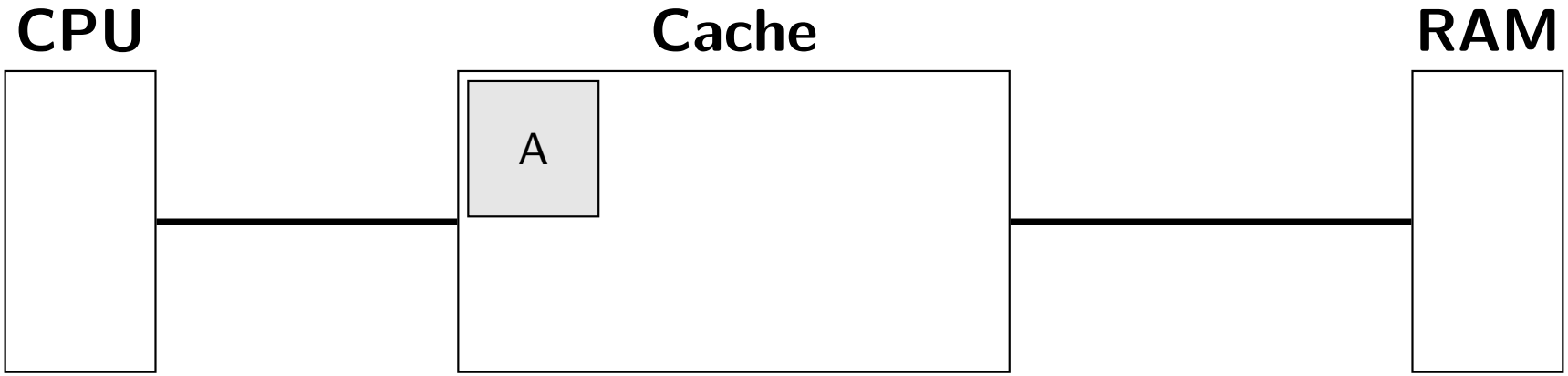
Prefetching automatique

- Les contrôleurs de cache tentent de prédire les accès futurs en détectant les motifs d'accès



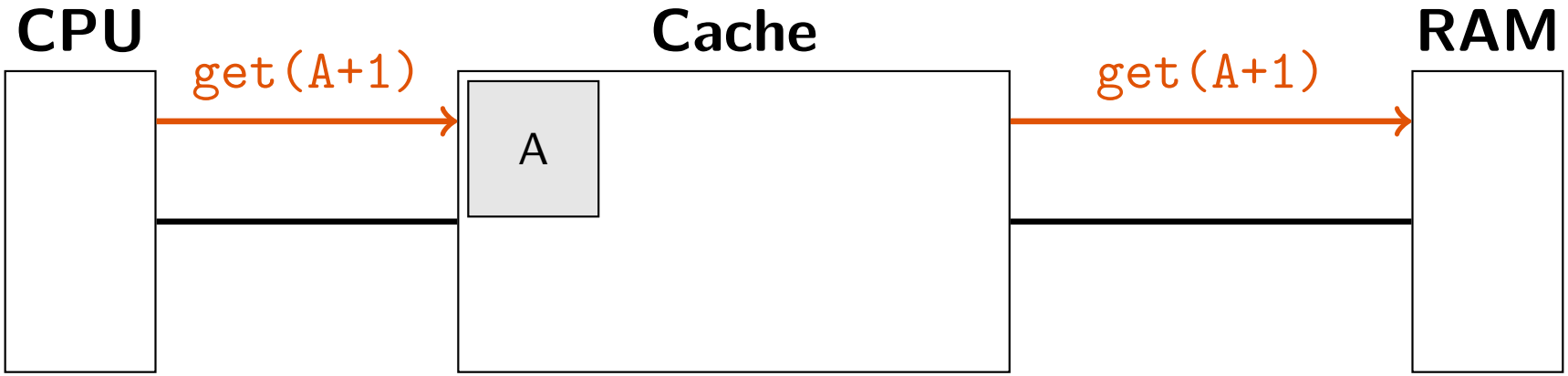
Prefetching automatique

- Les contrôleurs de cache tentent de prédire les accès futurs en détectant les motifs d'accès



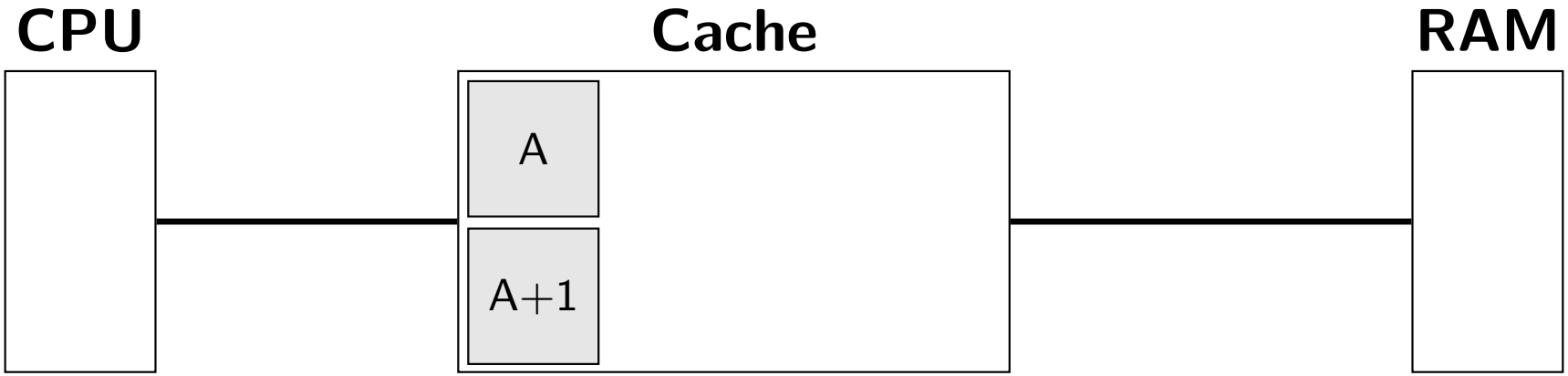
Prefetching automatique

- Les contrôleurs de cache tentent de prédire les accès futurs en détectant les motifs d'accès



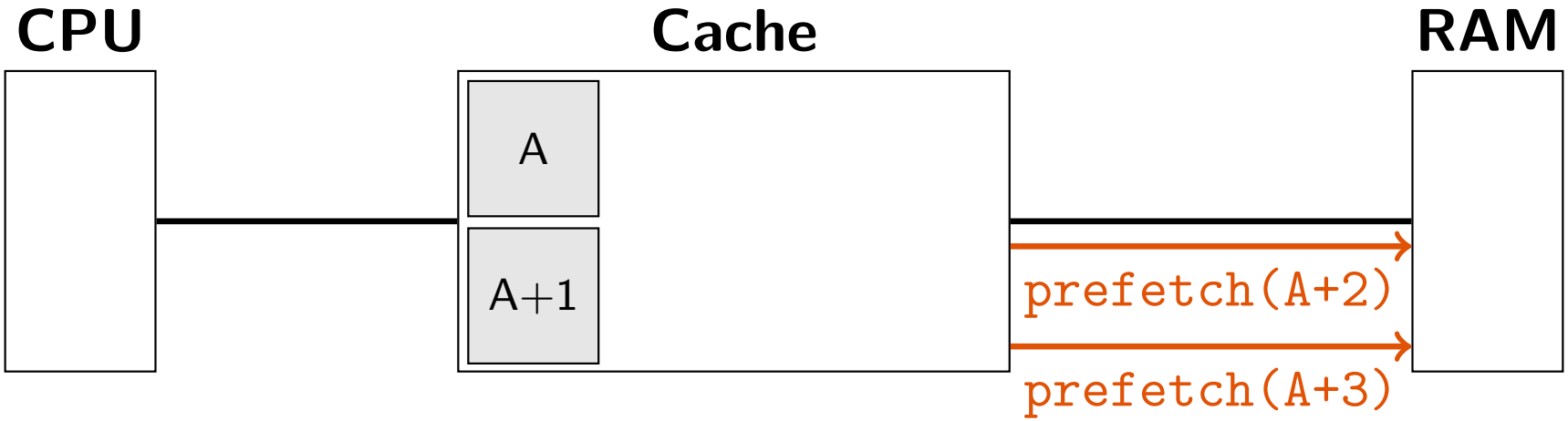
Prefetching automatique

- Les contrôleurs de cache tentent de prédire les accès futurs en détectant les motifs d'accès



Prefetching automatique

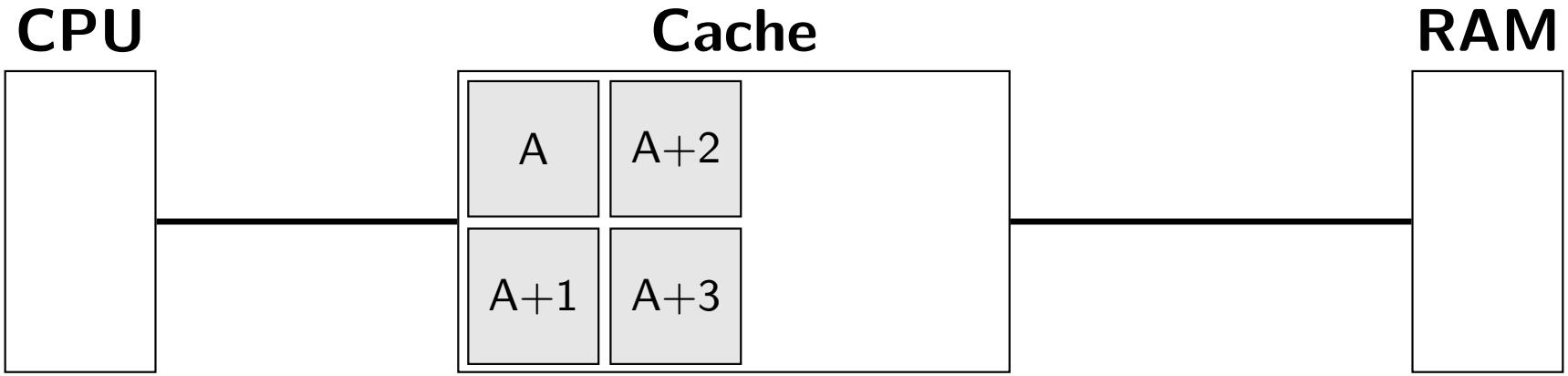
- Les contrôleurs de cache tentent de prédire les accès futurs en détectant les motifs d'accès



- Quand le contrôleur de cache détecte un motif, il *prefetch* automatiquement les prochaines données

Prefetching automatique

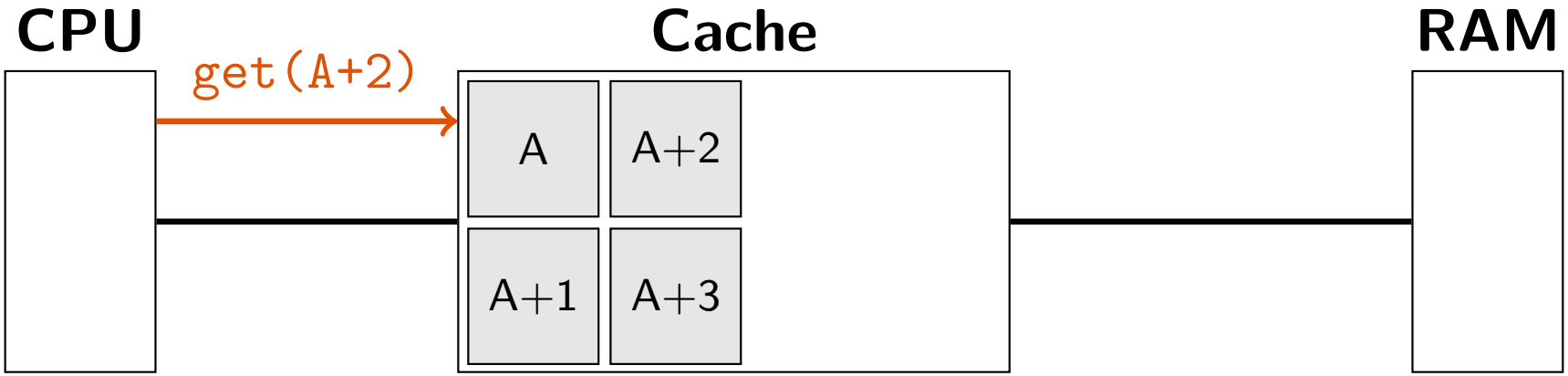
- Les contrôleurs de cache tentent de prédire les accès futurs en détectant les motifs d'accès



- Quand le contrôleur de cache détecte un motif, il *prefetch* automatiquement les prochaines données
- Le *prefetcher* et le CPU fonctionnent en parallèle $\rightarrow$ pas de stall

Prefetching automatique

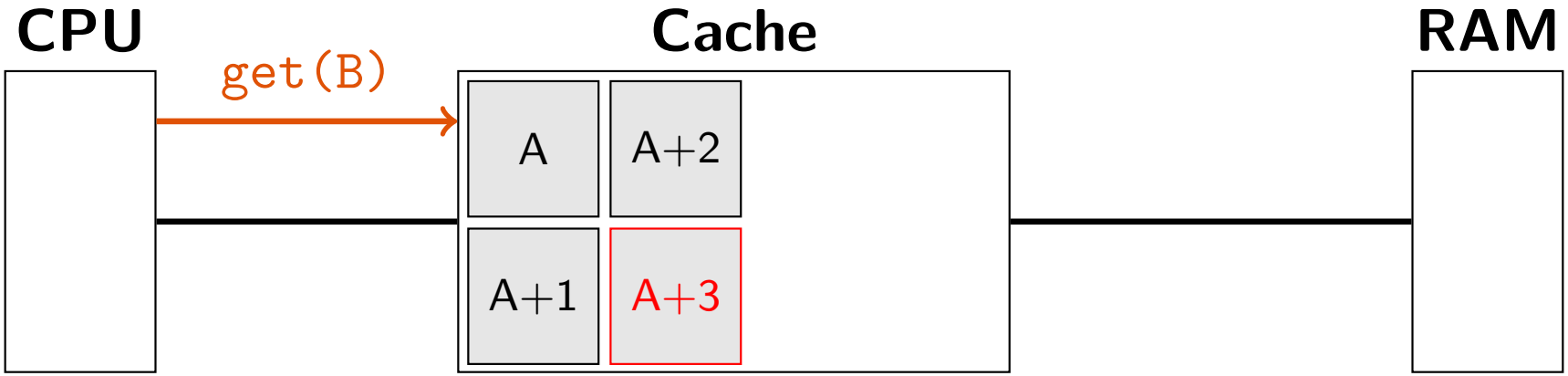
- Les contrôleurs de cache tentent de prédire les accès futurs en détectant les motifs d'accès



- Quand le contrôleur de cache détecte un motif, il *prefetch* automatiquement les prochaines données
- Le *prefetcher* et le CPU fonctionnent en parallèle → pas de stall
- Si la prédiction est correcte, le prefetching augmente le hit rate

Prefetching automatique

- Les contrôleurs de cache tentent de prédire les accès futurs en détectant les motifs d'accès



- Quand le contrôleur de cache détecte un motif, il *prefetch* automatiquement les prochaines données
- Le *prefetcher* et le CPU fonctionnent en parallèle → pas de stall
- Si la prédiction est correcte, le prefetching augmente le hit rate
- Si la prédiction est érronée, consommation inutile de ressources
  - Ligne de cache évincée sans contrepartie
  - Consommation de bande passante mémoire

Mémoire cache : résumé

- La mémoire cache est une mémoire secondaire rapide de plus faible capacité que la mémoire principale
  - La mémoire cache tire parti des principes de localité pour avoir à disposition les données demandées par le CPU
  - Les données sont échangées sous forme de ligne de cache

- Le cache direct associe à chaque adresse un slot de cache unique
  - Le slot d'une ligne est déterminé par l'index de son adresse
  - Si deux lignes ont le même index, elles sont en conflit

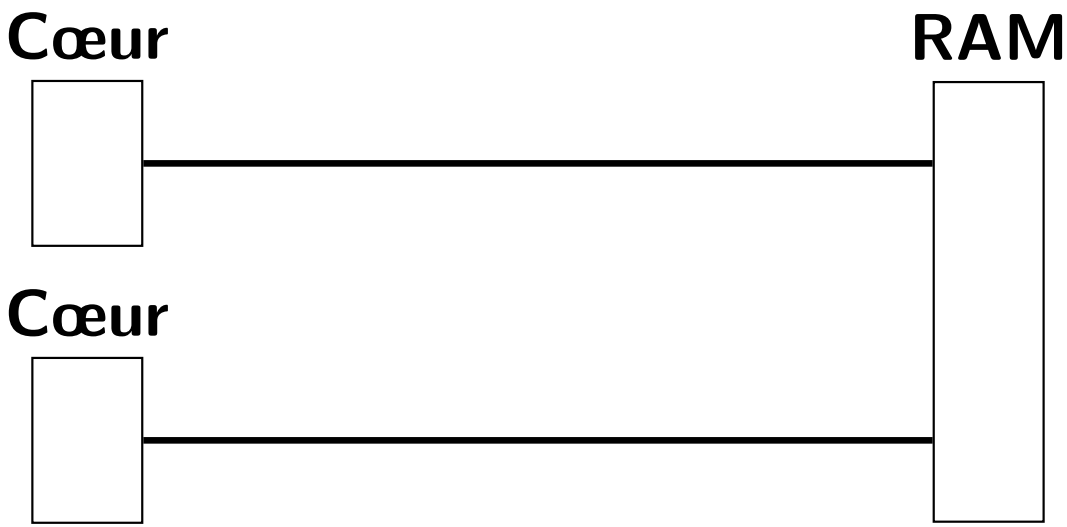
- Le cache *M*-associatif associe à chaque adresse un slot parmis *M*
  - Un ensemble de *M* slots possibles (les *ways*) s'appelle un *set*
  - Le set d'une ligne est déterminé par l'index de son adresse
  - Le cache décide quelle ligne évincer d'un set avec une stratégie d'éviction

- Le *prefetcher* est la partie du cache qui détecte les motifs d'accès
  - En fonction du motif observé, le prefetcher va chercher en mémoire principale les lignes prochainement accédées

Plan du cours

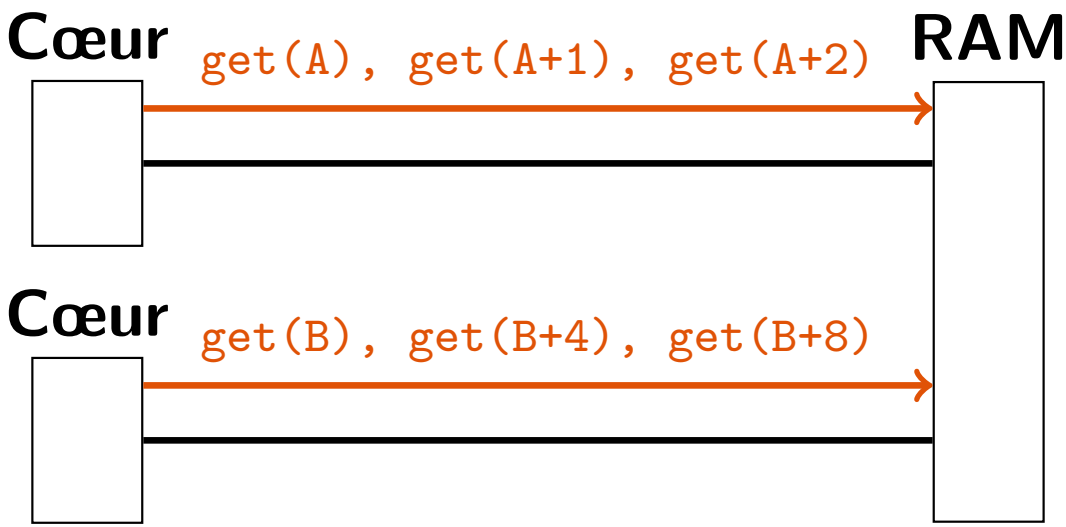
Architectures multicœurs et mémoire cache

- Les machines actuelles ont une architecture multicœur
- Chaque cœur accède à la mémoire indépendamment des autres cœurs



Architectures multicœurs et mémoire cache

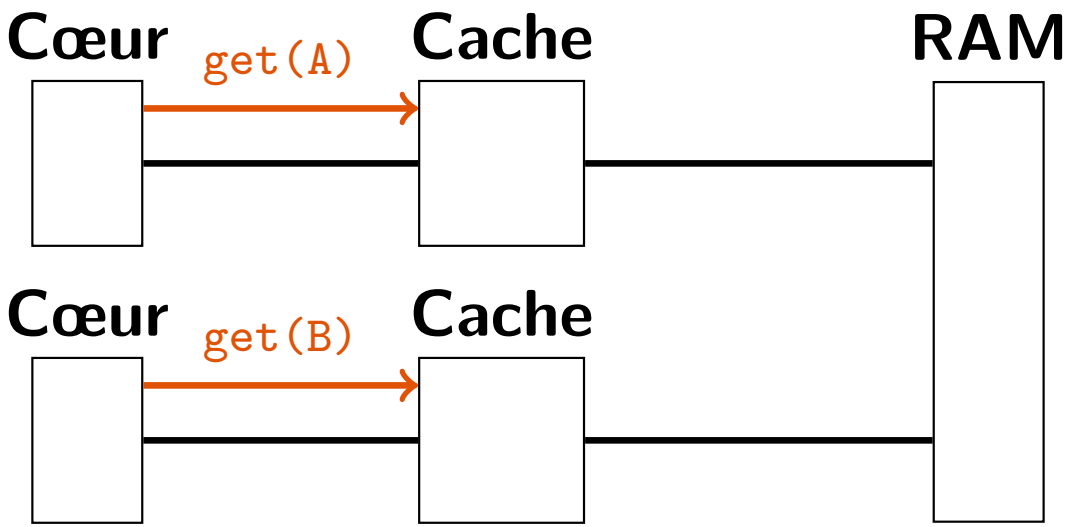
- Les machines actuelles ont une architecture multicœur
- Chaque cœur accède à la mémoire indépendamment des autres cœurs



- Chaque cœur suit son propre motif d'accès $\rightarrow$ peu de localité entre les cœurs

Architectures multicœurs et mémoire cache

- Les machines actuelles ont une architecture multicœur
- Chaque cœur accède à la mémoire indépendamment des autres cœurs



- Chaque cœur suit son propre motif d'accès → peu de localité entre les cœurs
- Chaque cœur dispose de son propre cache → bonne localité par cache

Cohérence séquentielle

- Les fonctions `on_core_0()` et `on_core_1()` s'exécutent en parallèle

```
A = 0;
B = 0;
```

```
void on_core_0(void) {
    A = 1;
    B = 1;
}
```

```
void on_core_1(void) {
    b = B;
    a = A;
}
```

- États possibles de `(a,b)` :

Cohérence séquentielle

- Les fonctions `on_core_0()` et `on_core_1()` s'exécutent en parallèle

```
A = 0;
B = 0;
```

```
void on_core_0(void) {
    A = 1;
    B = 1;
}
```

```
void on_core_1(void) {
    b = B;
    a = A;
}
```

- États possibles de `(a,b)` : `(0,0)`, `(1,0)`, `(1,1)`

Cohérence séquentielle

- Les fonctions `on_core_0()` et `on_core_1()` s'exécutent en parallèle

```
A = 0;
B = 0;
```

```
void on_core_0(void) {
    A = 1;
    B = 1;
}
```

```
void on_core_1(void) {
    b = B;
    a = A;
}
```

- États possibles de `(a,b)` : `(0,0)`, `(1,0)`, `(1,1)`
- L'état `(0,1)` est impossible avec un modèle de cohérence séquentielle

Cohérence séquentielle : définition

- Les fonctions `on_core_0()` et `on_core_1()` s'exécutent en parallèle

```
A = 0;
B = 0;
```

```
void on_core_0(void) {
    A = 1;
    B = 1;
}
```

```
void on_core_1(void) {
    b = B;
    a = A;
}
```

- États possibles de `(a,b)` : `(0,0)`, `(1,0)`, `(1,1)`
- L'état `(0,1)` est impossible avec un modèle de cohérence séquentielle

Une exécution est **cohérente séquentiellement** si
il existe un ordre total des opérations qui préserve l'ordre des instructions de chaque cœur et qui conduit au même résultat

Cohérence séquentielle : définition

- Les fonctions `on_core_0()` et `on_core_1()` s'exécutent en parallèle

```
A = 0;
B = 0;
```

```
void on_core_0(void) {
    A = 1;
    B = 1;
}
```

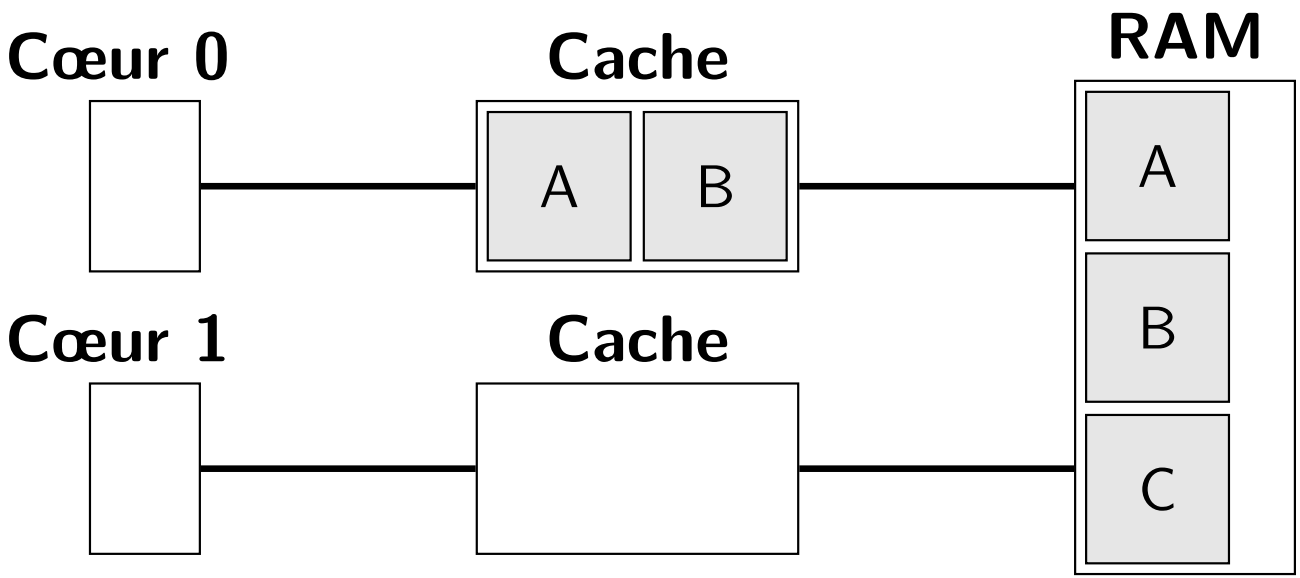
```
void on_core_1(void) {
    b = B;
    a = A;
}
```

- États possibles de `(a,b)` : `(0,0)`, `(1,0)`, `(1,1)`
- L'état `(0,1)` est impossible avec un modèle de cohérence séquentielle

> Une exécution est **cohérente séquentiellement** si il existe un ordre total des opérations qui préserve l'ordre des instructions de chaque cœur et qui conduit au même résultat

- La cohérence séquentielle paraît naturelle pour le programmeur

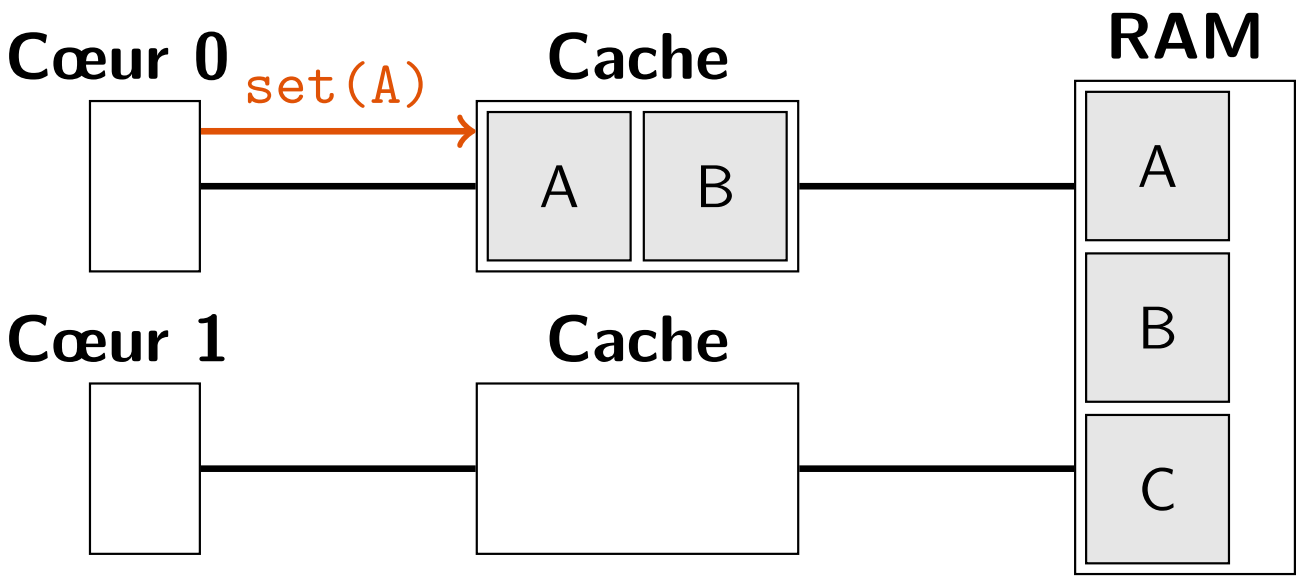
Cohérence séquentielle et cache multicœur




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

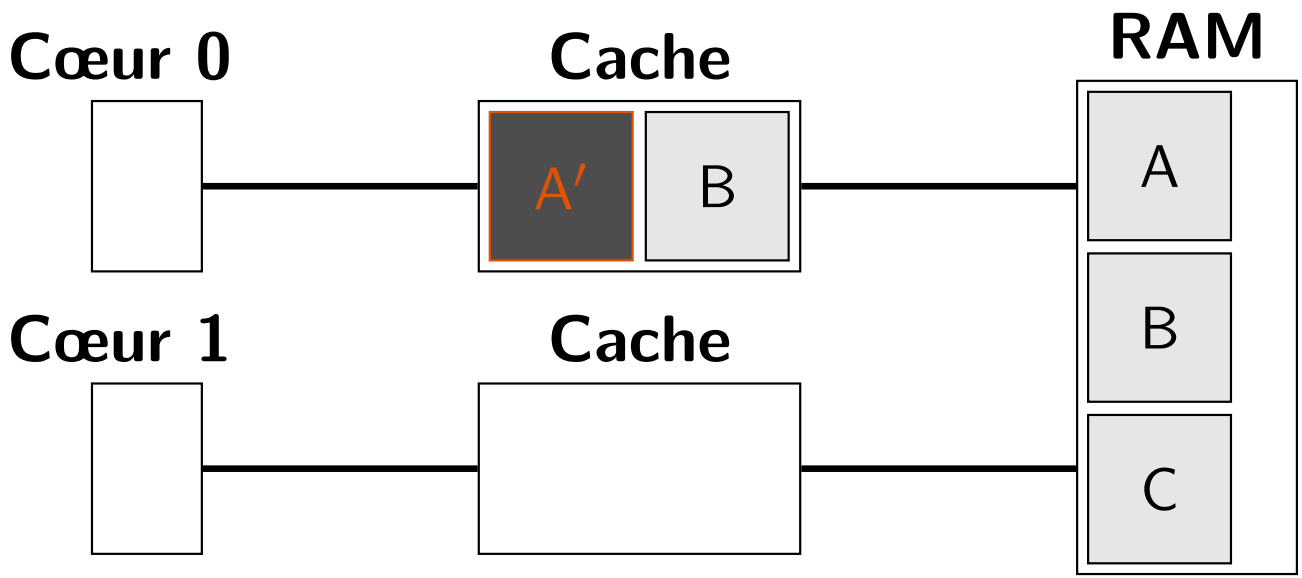
Cohérence séquentielle et cache multicœur




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

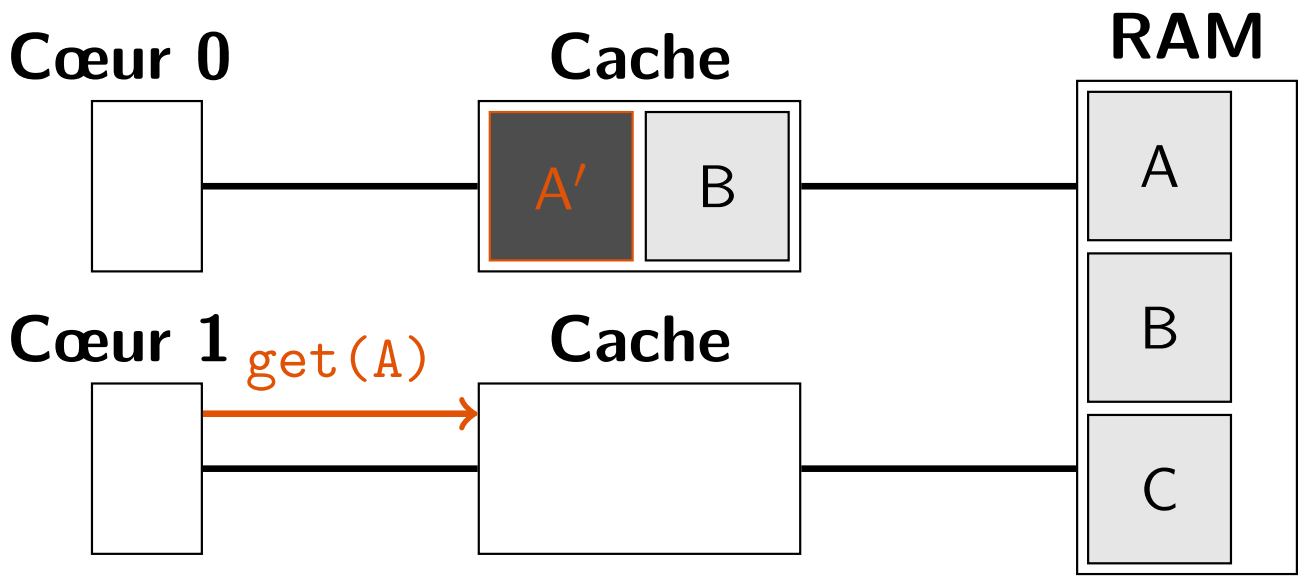
Cohérence séquentielle et cache multicœur




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

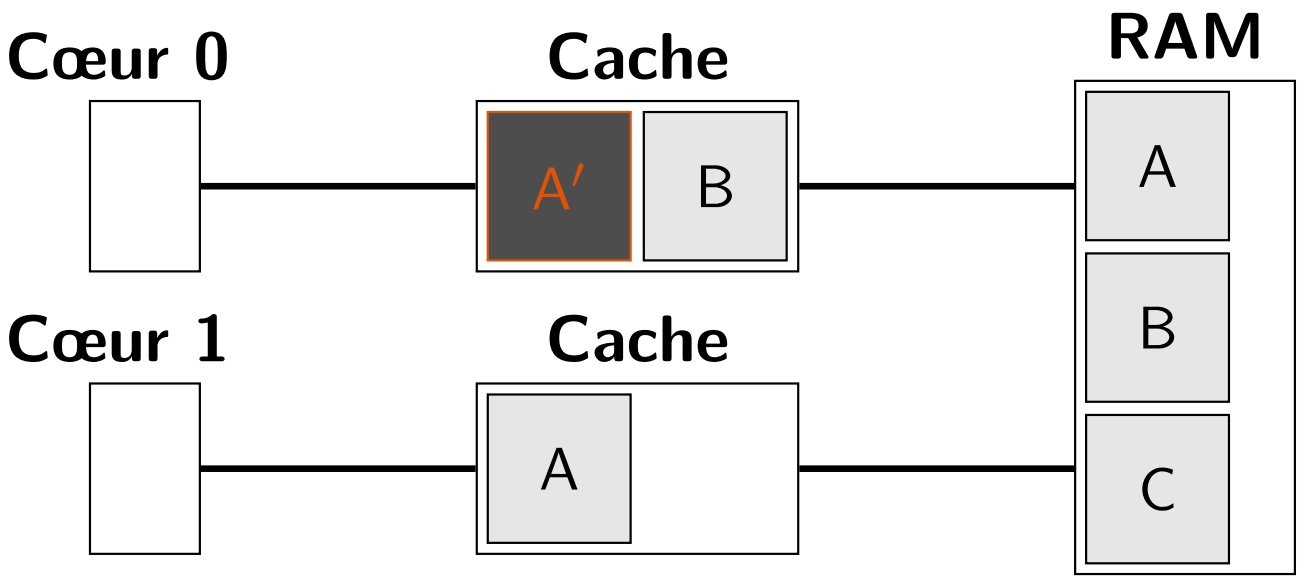



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée

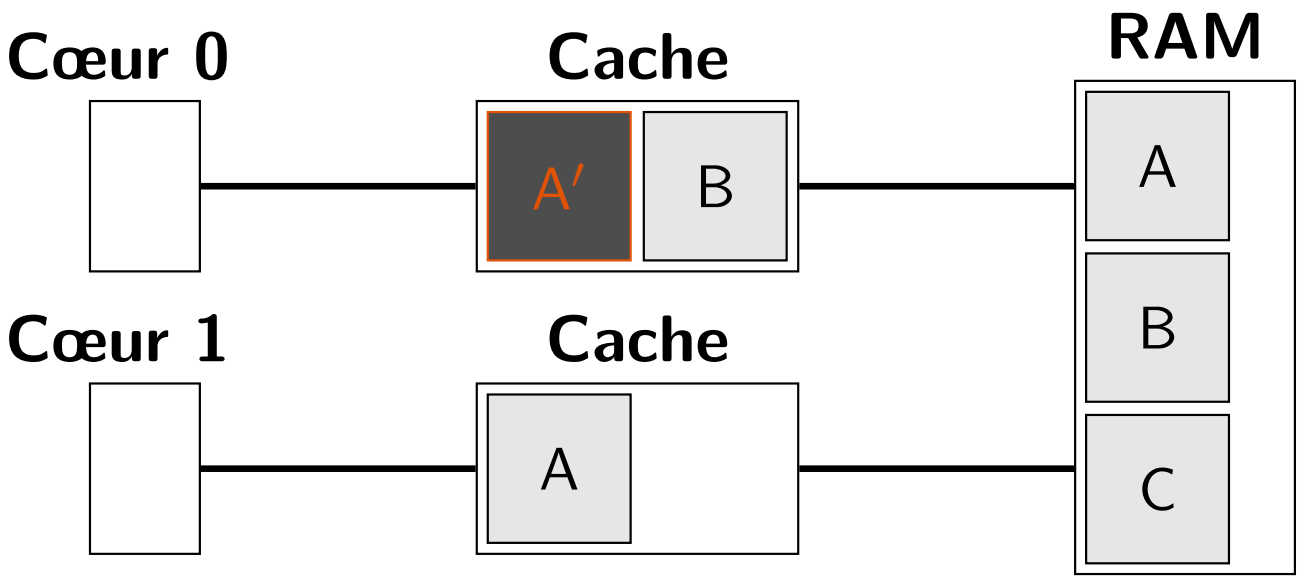



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

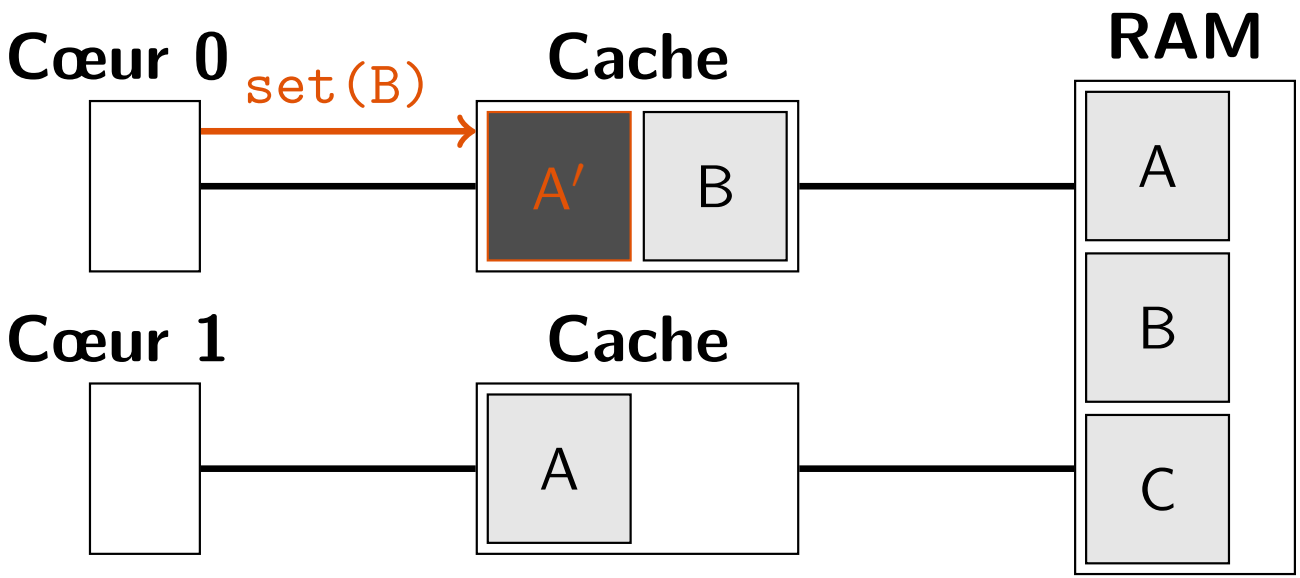



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

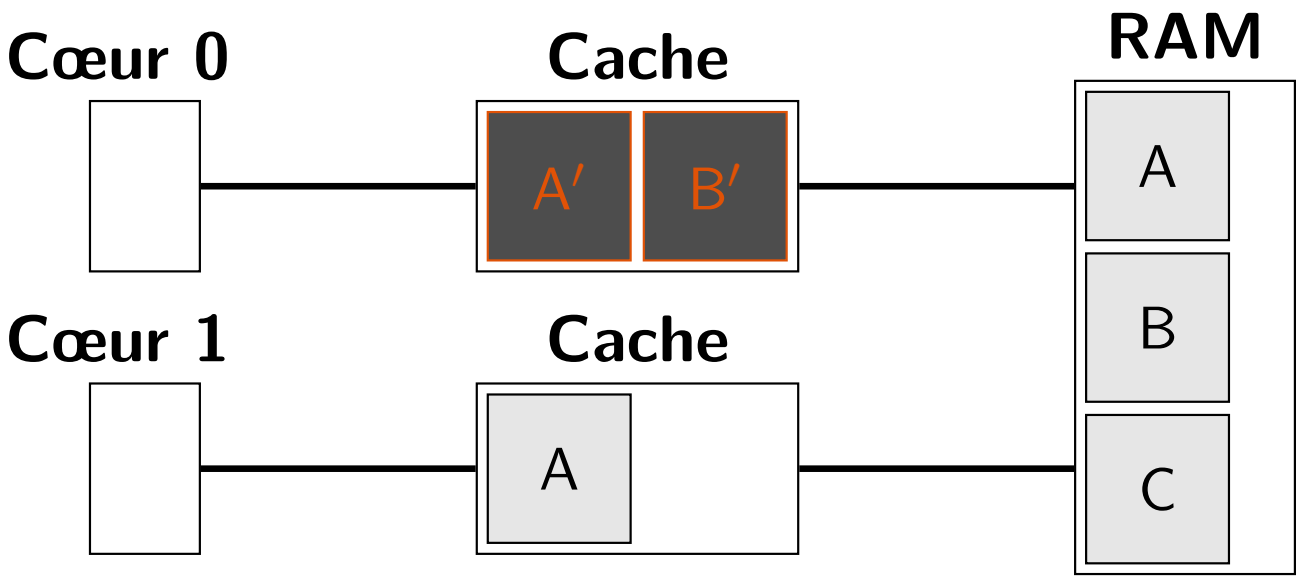



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

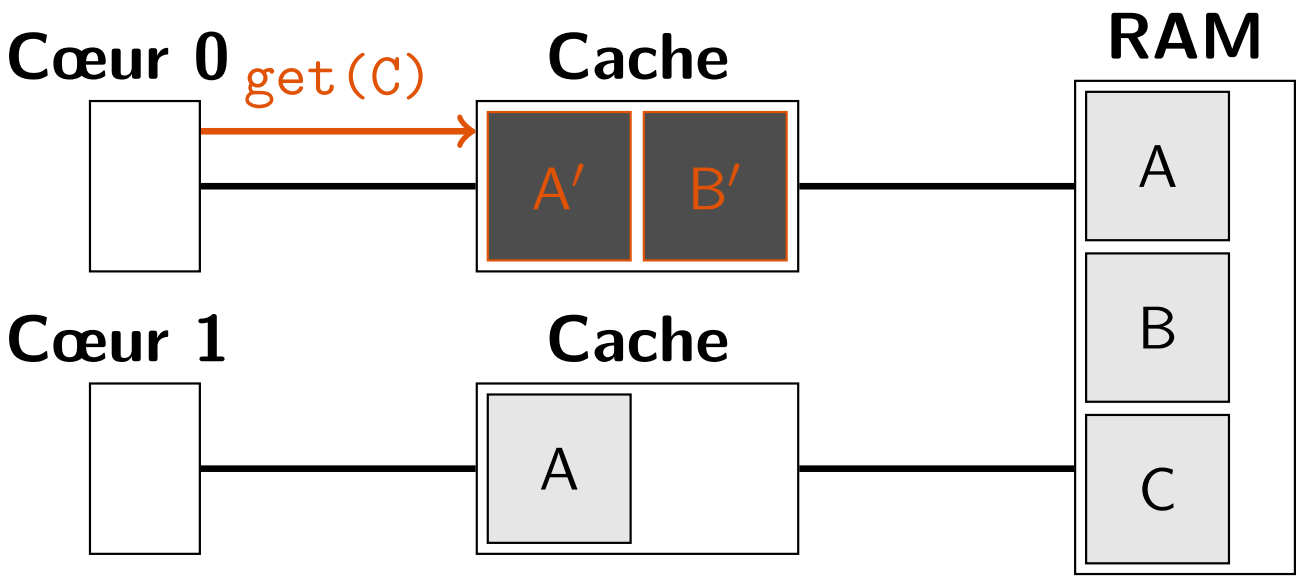



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

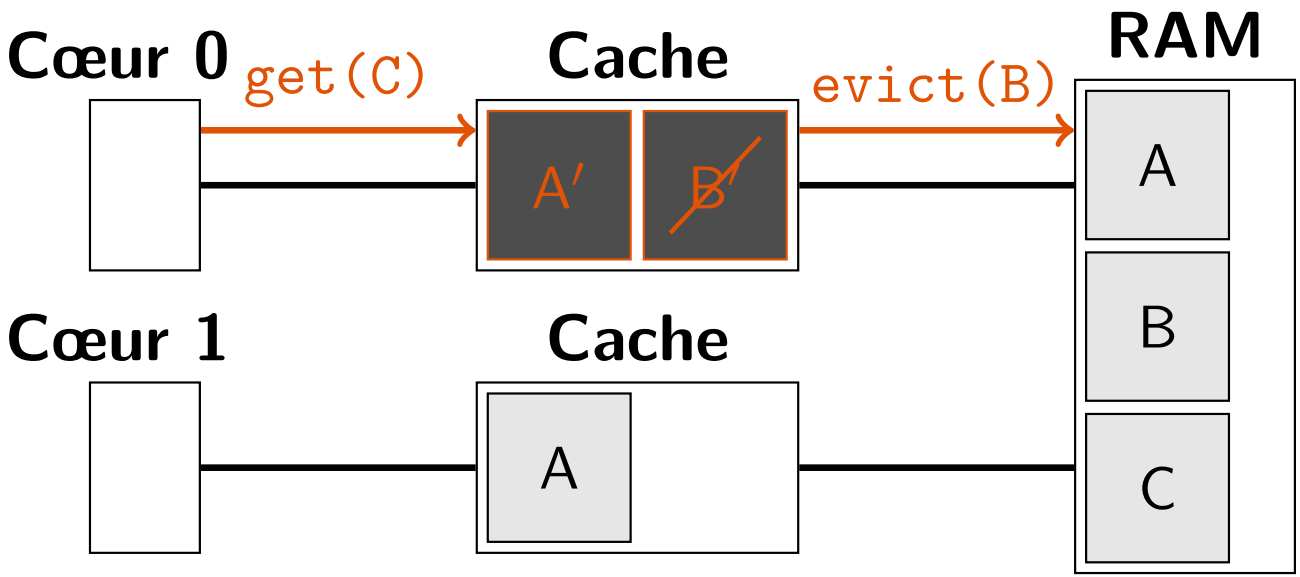



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

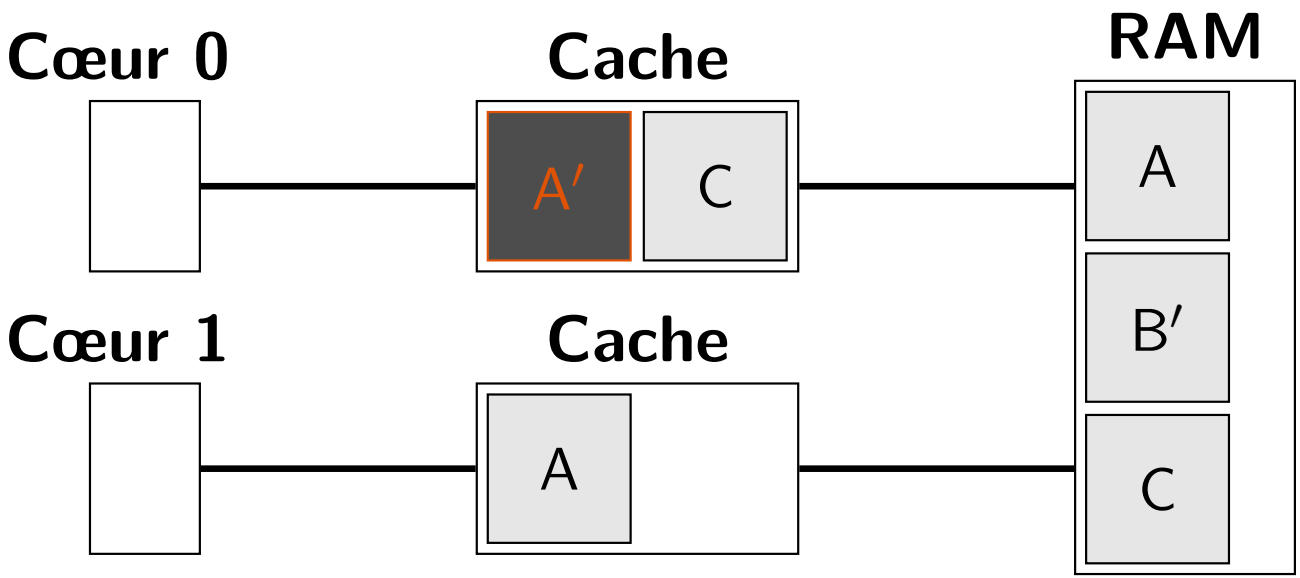



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

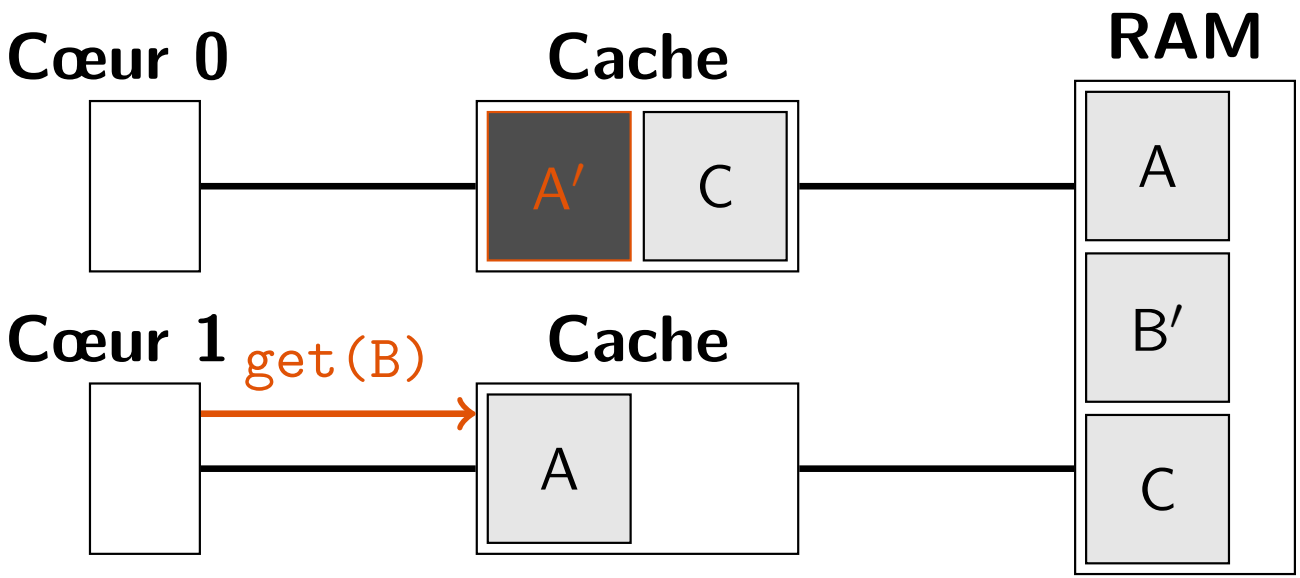



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

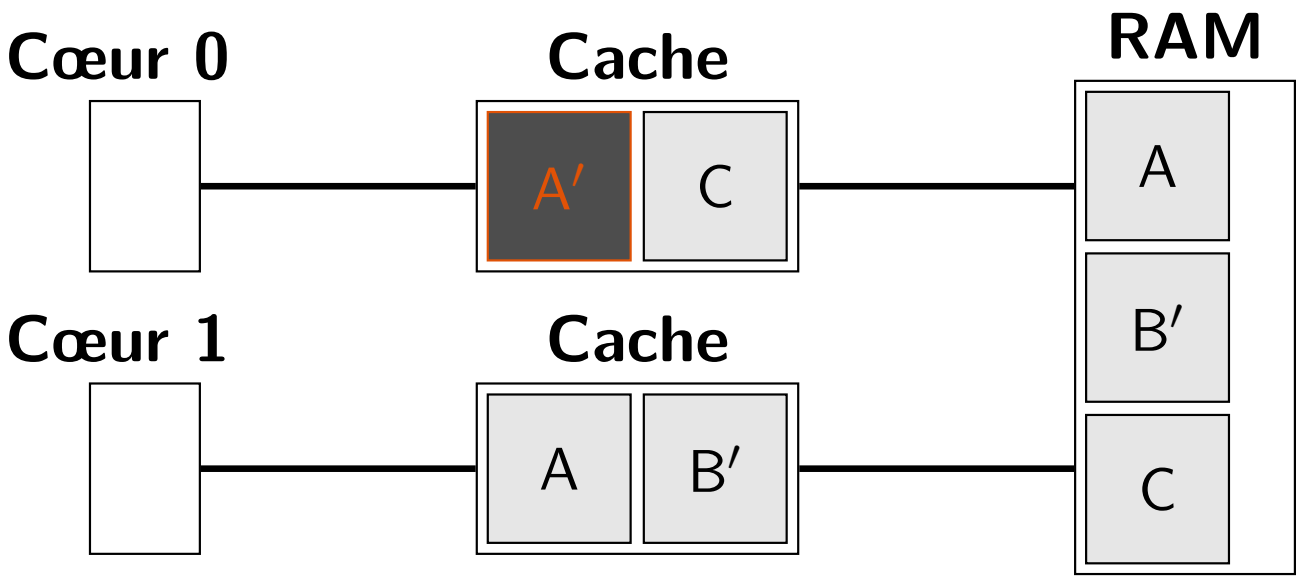



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

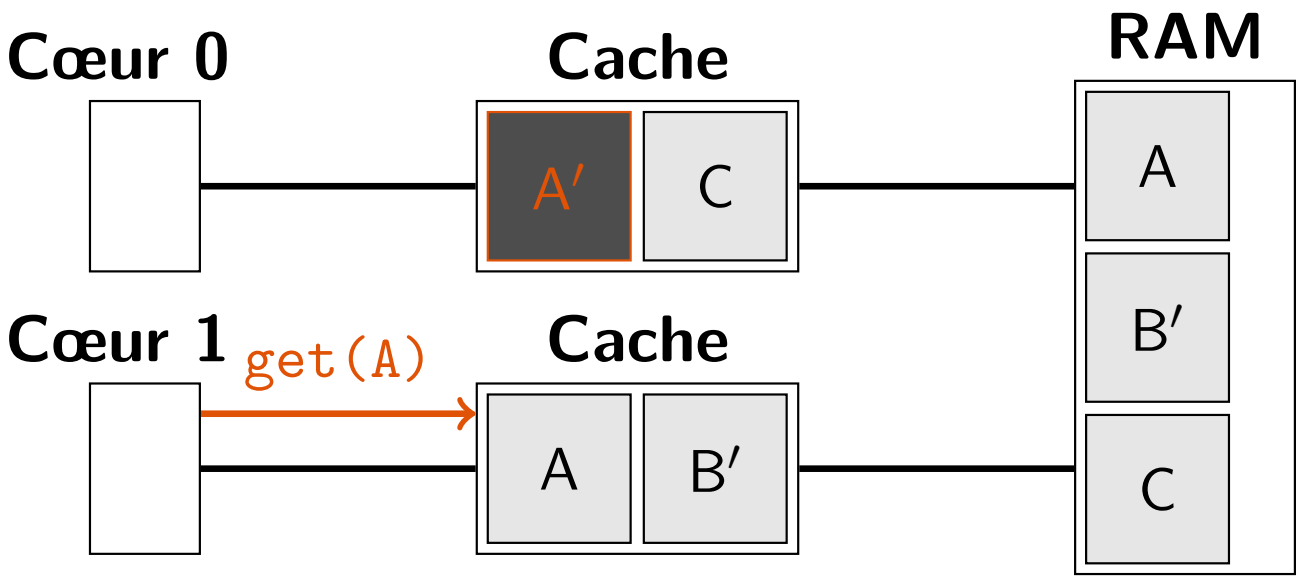



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

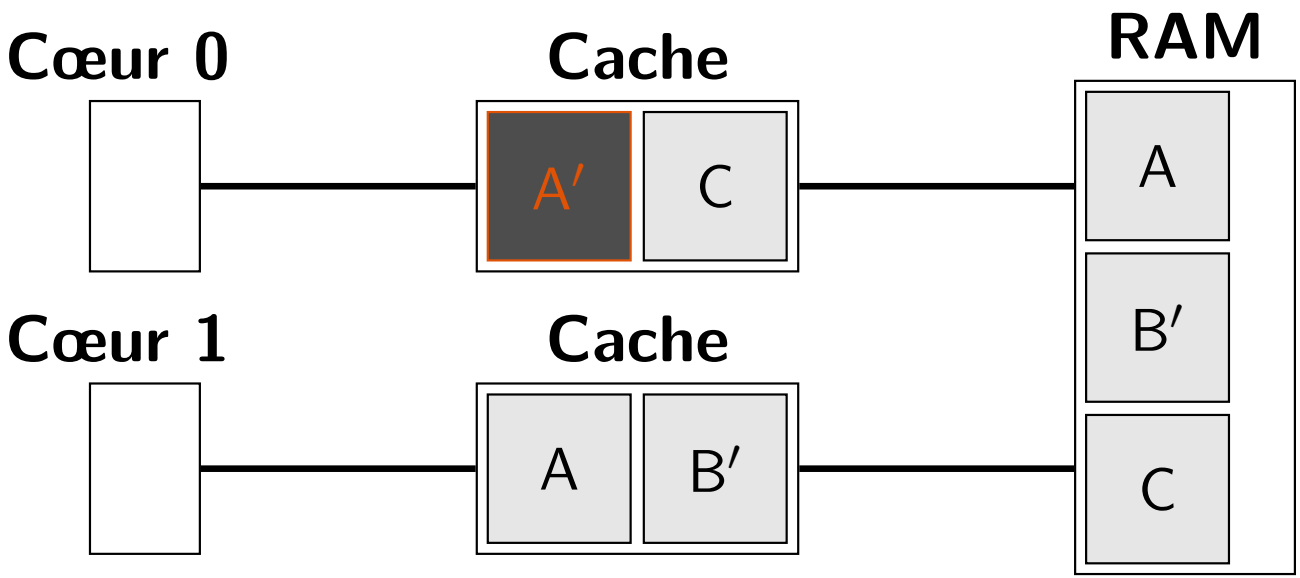



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici
- Le cœur 1 lit `(a,b) = (0,1)`

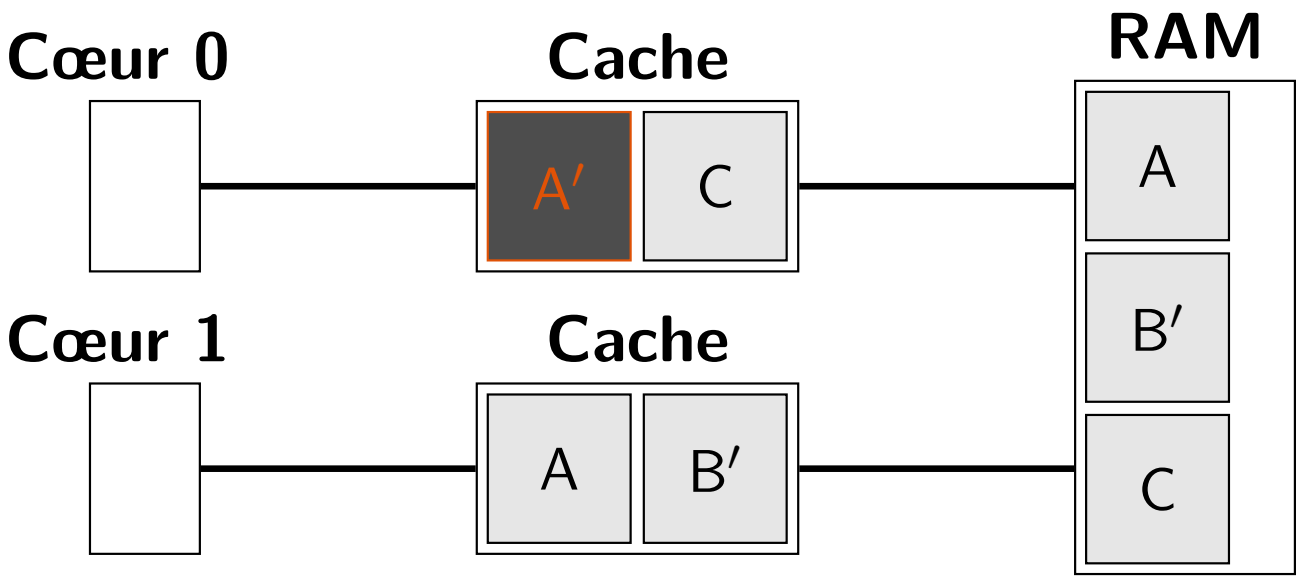



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici
- Le cœur 1 lit `(a,b) = (0,1)` → incohérence séquentielle




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Caches multicœurs et cohérence séquentielle : résumé

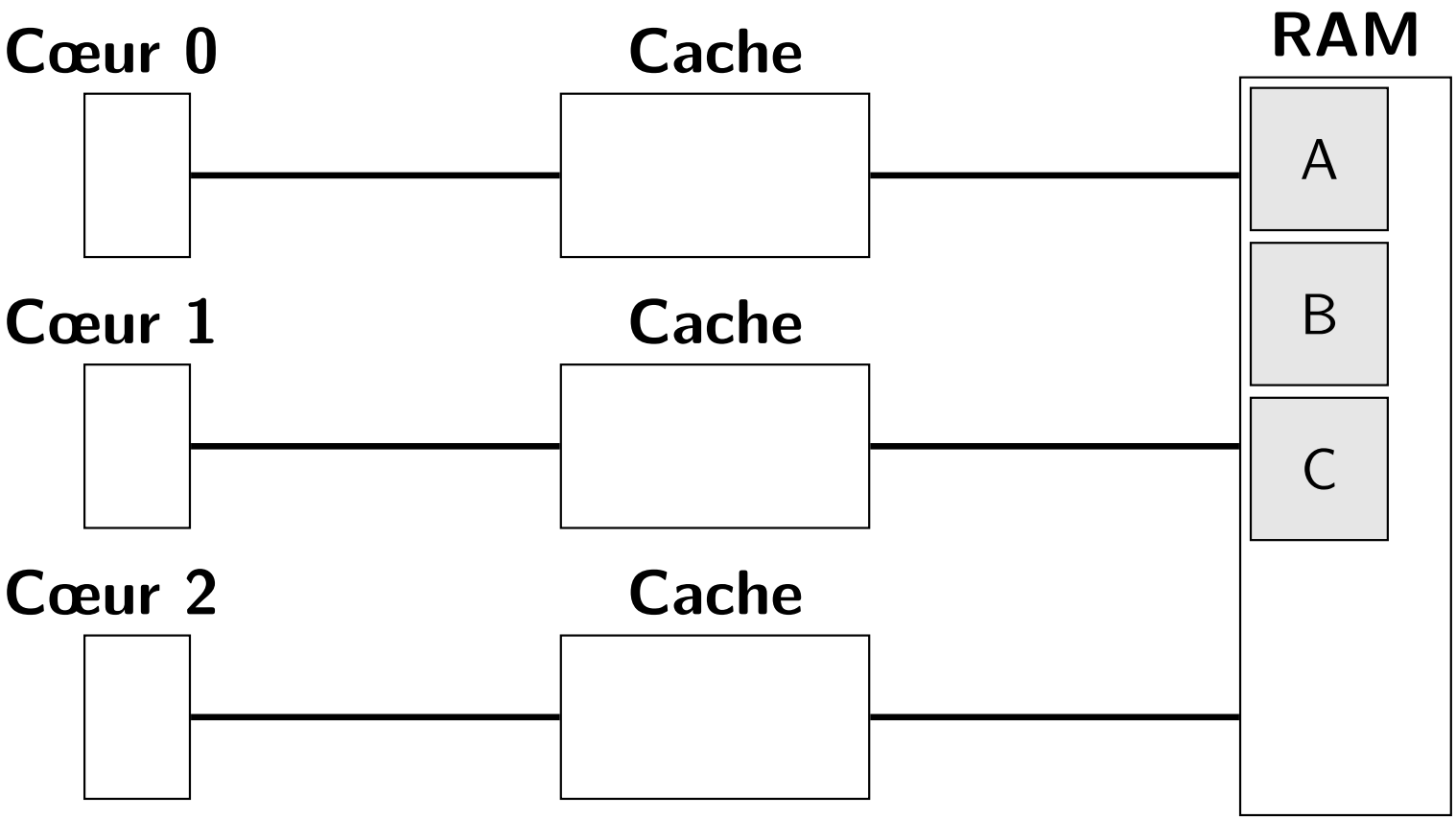
- Les architectures actuelles cherchent à guarantir la cohérence séquentielle des exécutions
  - Fonctionnement le plus naturel pour les programmeurs $\rightarrow$ incite les développeurs à utiliser l'architecture
  - Les instructions sur chaque cœur doivent "avoir l'air" de s'exécuter dans l'ordre
- Dans une architecture multicœur, chaque cœur a son propre cache
  - Les localités temporelle et spatiales sont spécifiques à chaque cœur
  - Pas de problème de multiplexage des ressources
- L'utilisation de mémoires cache totalement indépendantes brise la cohérence séquentielle

Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain

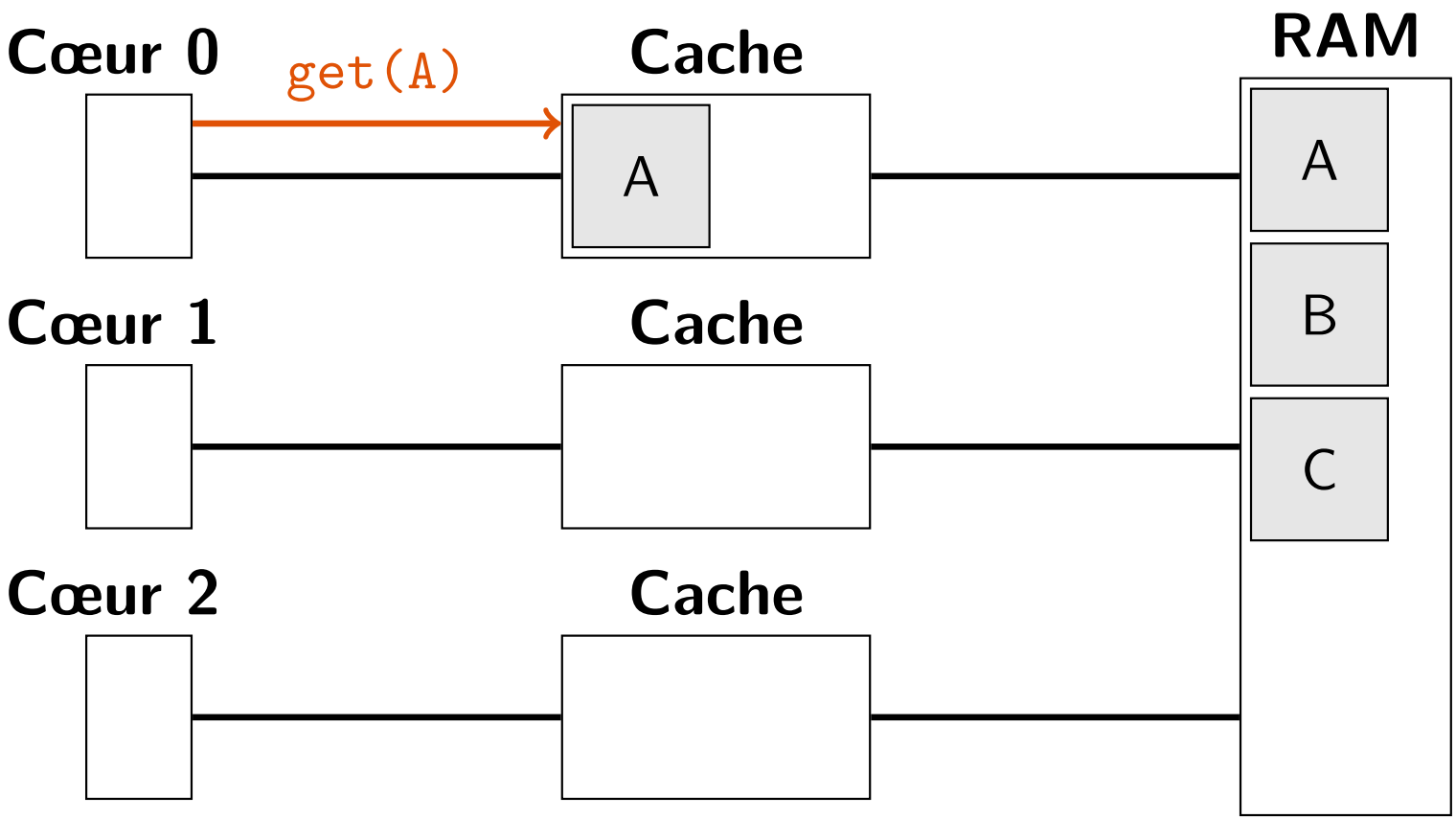
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : $\varnothing$



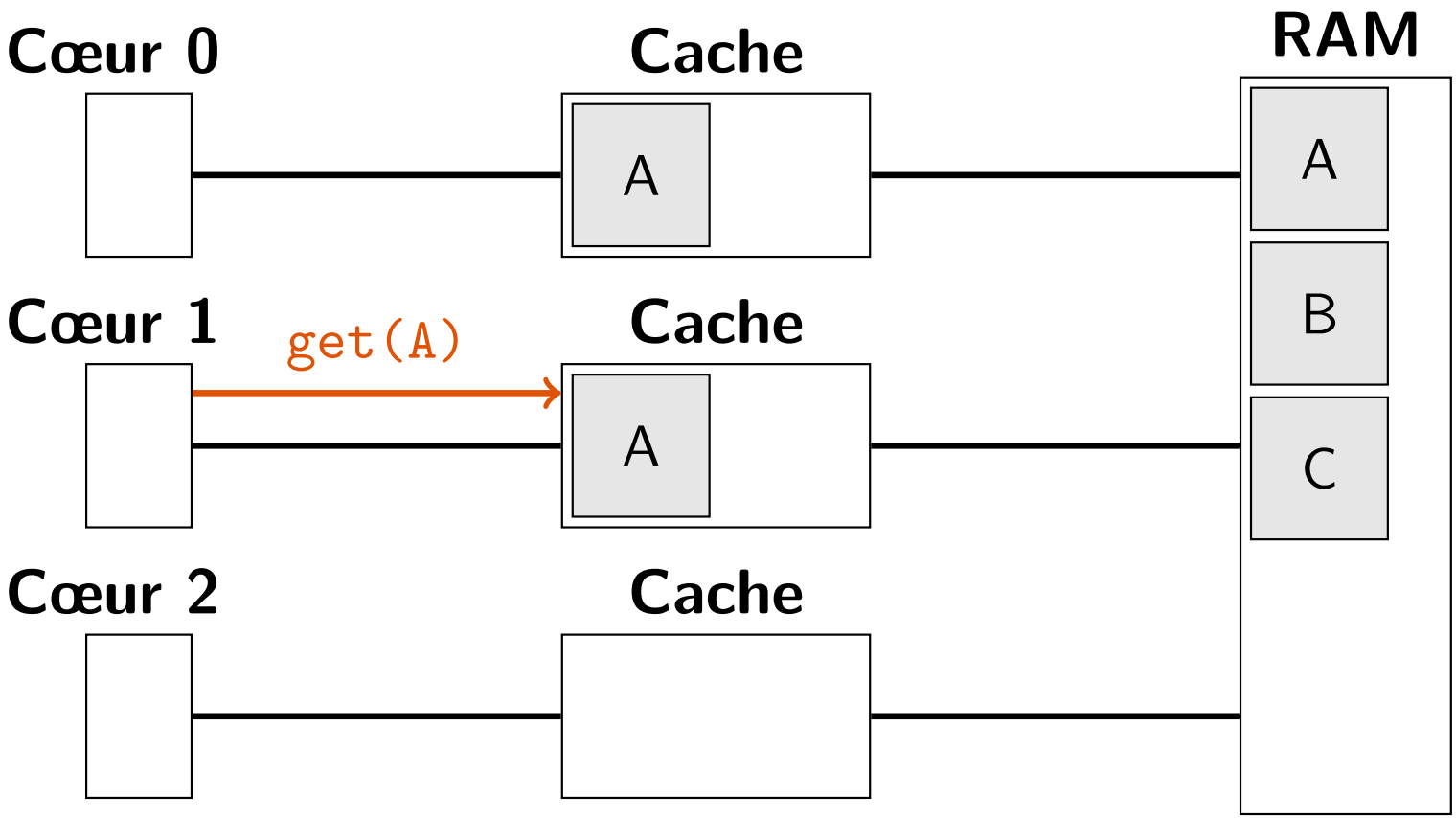
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : READ



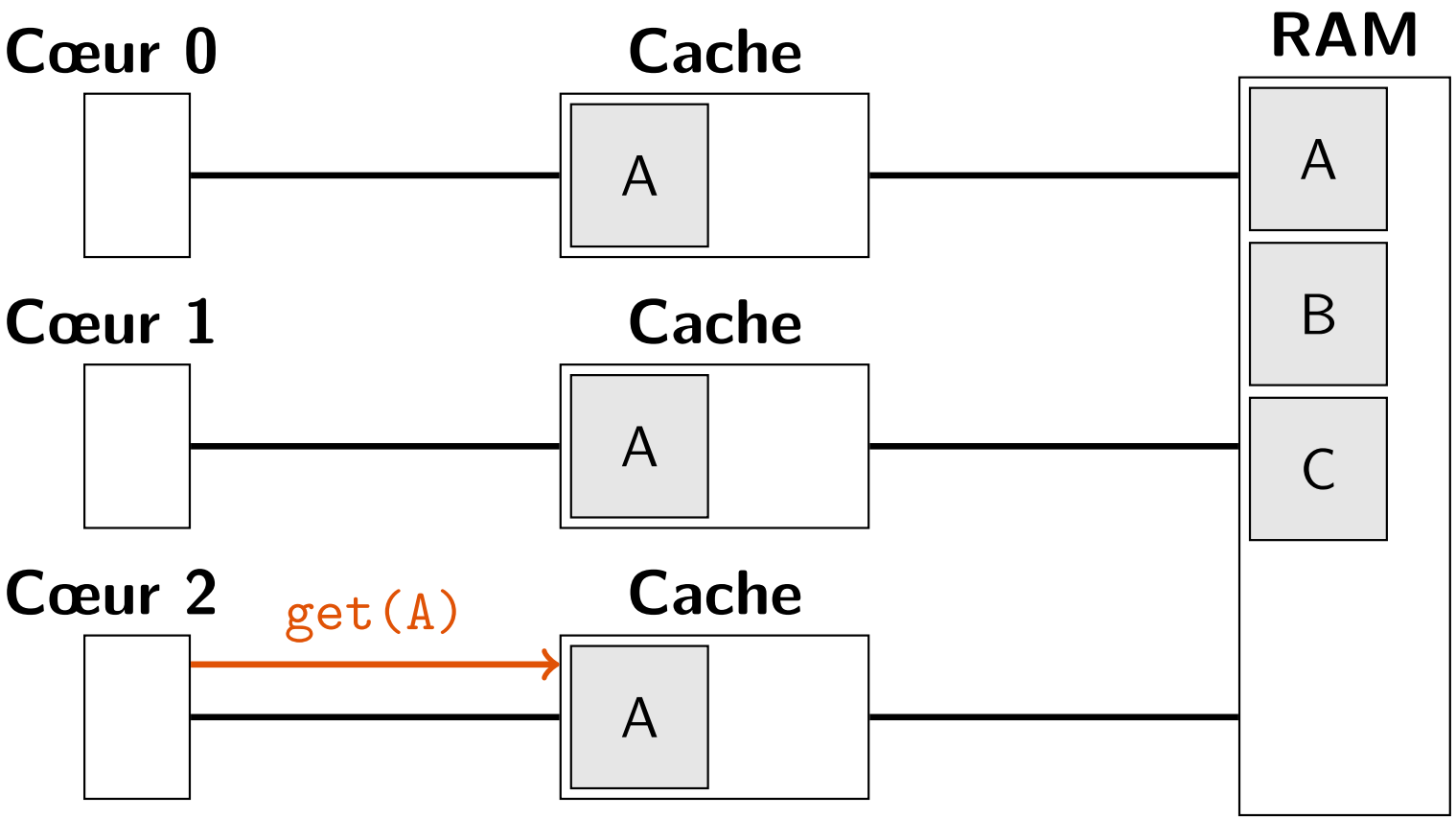
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : READ



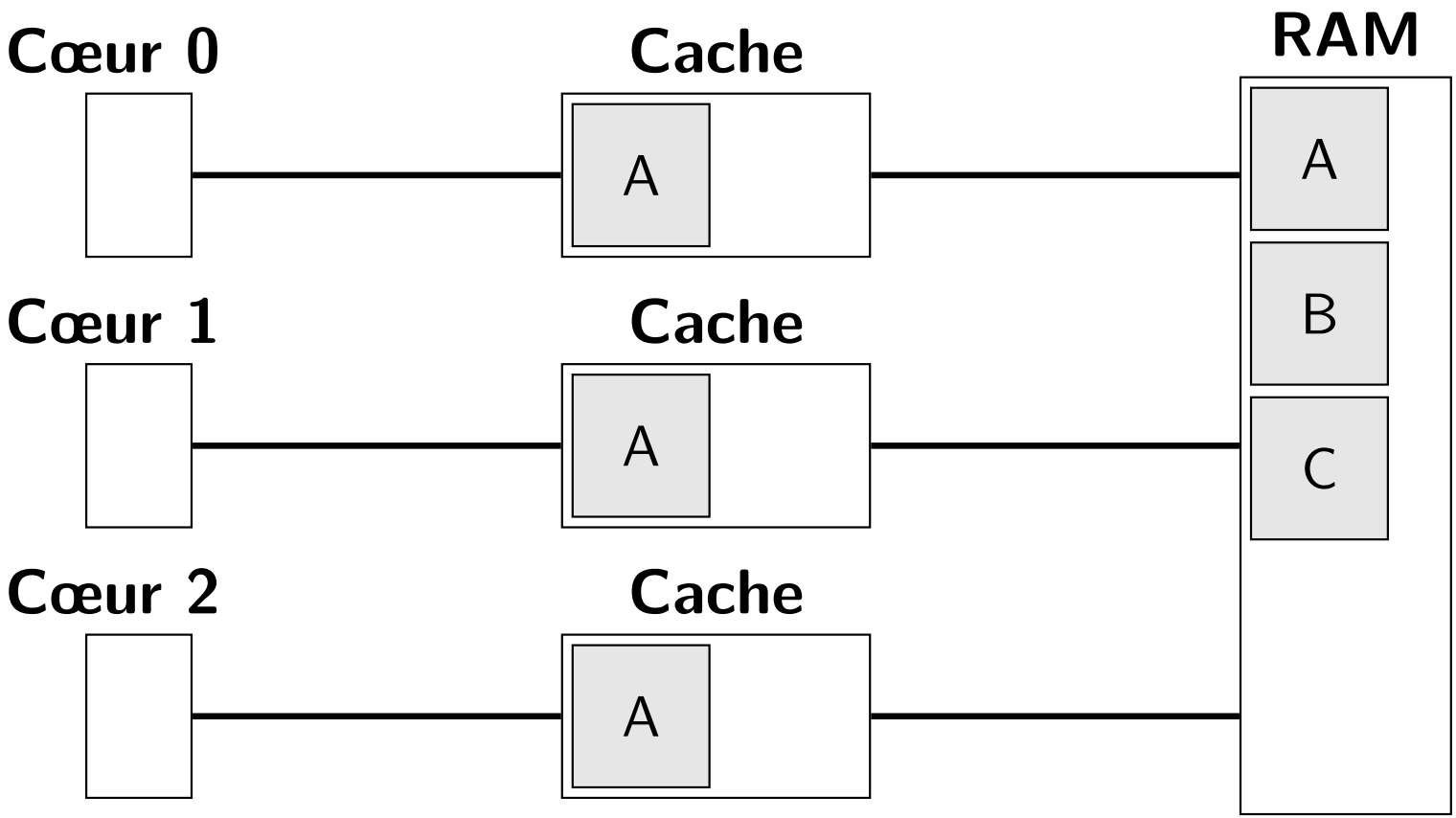
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : READ



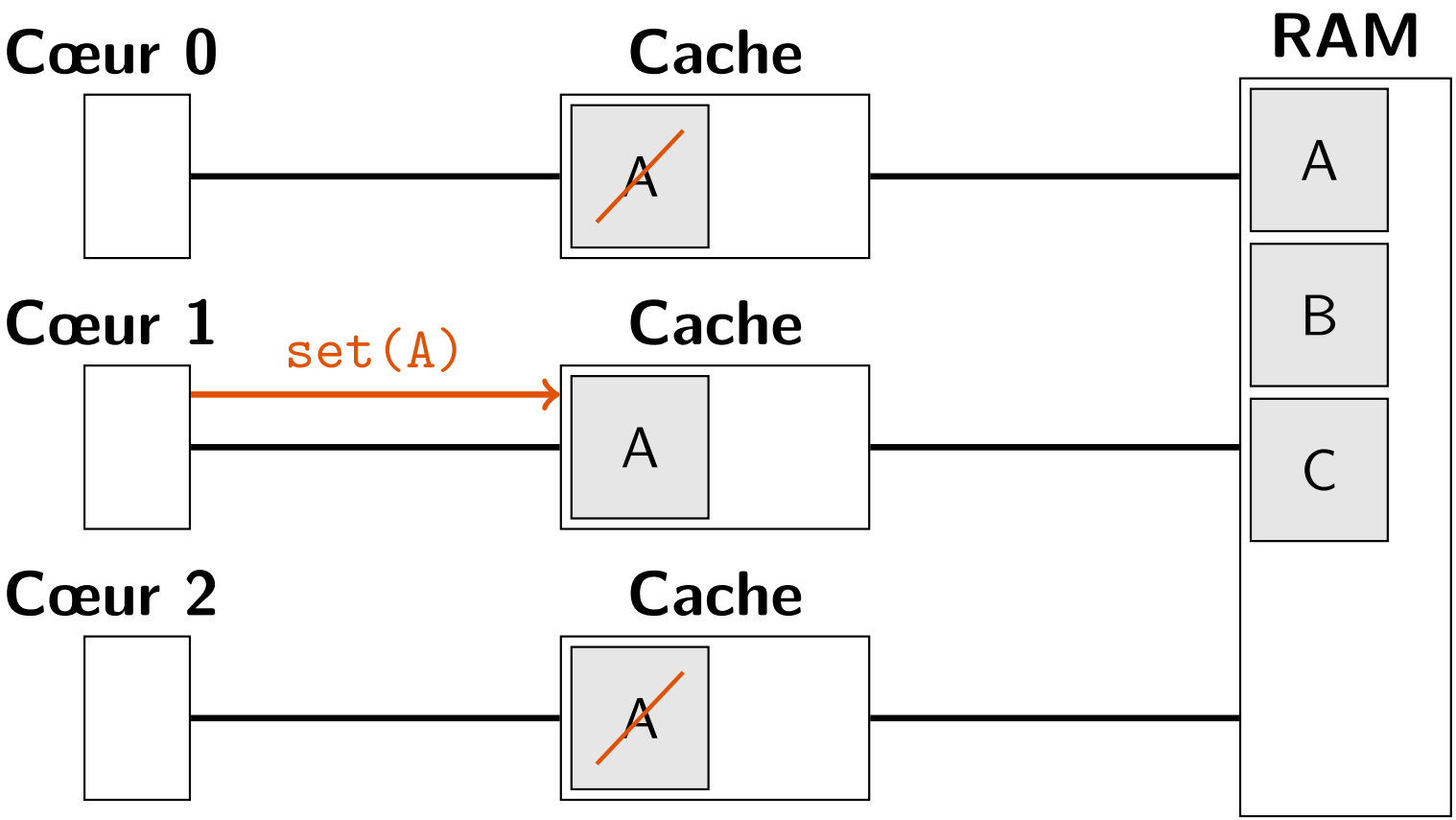
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : READ



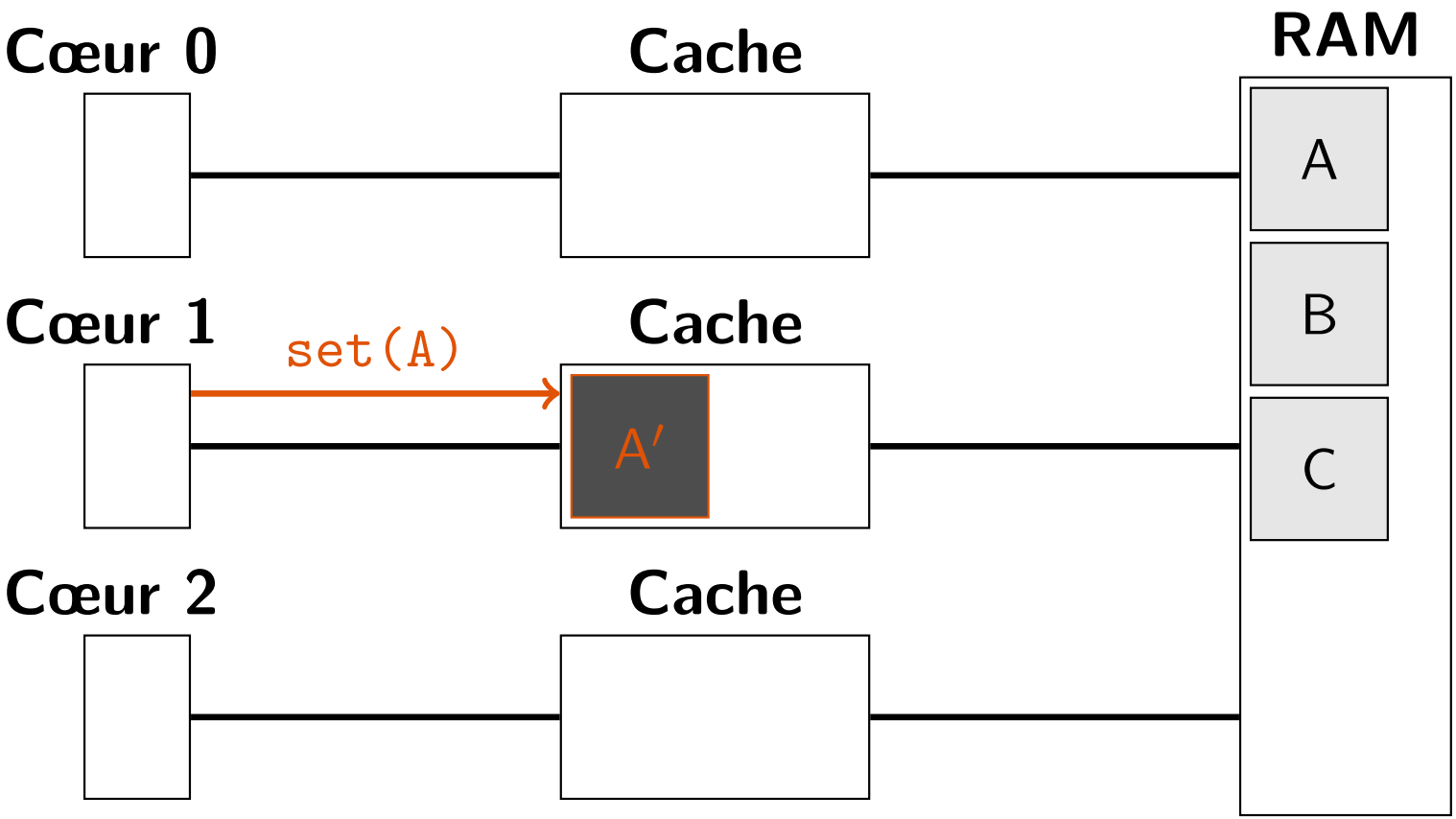
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : READ $\rightarrow$ WRITE



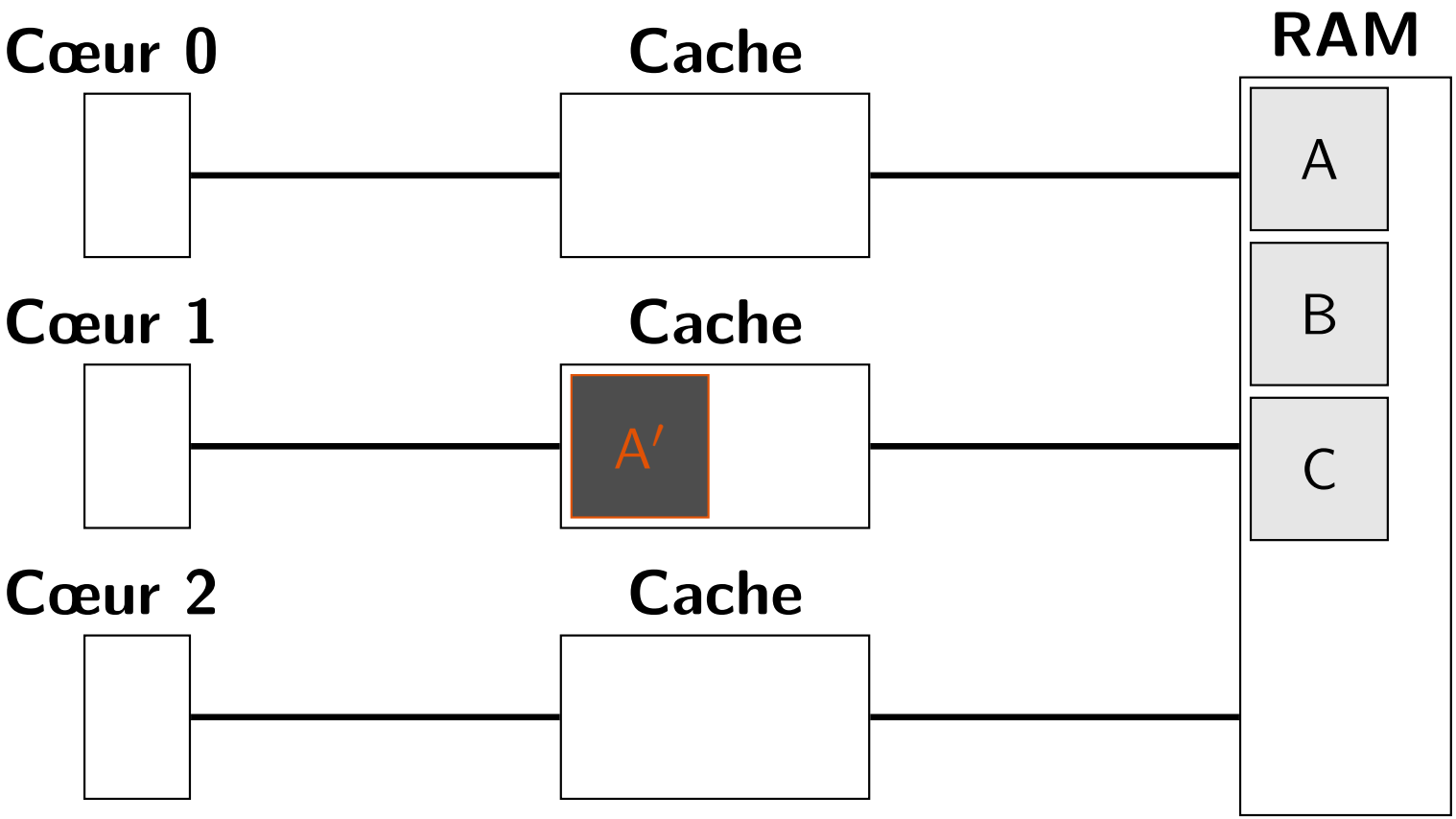
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : WRITE



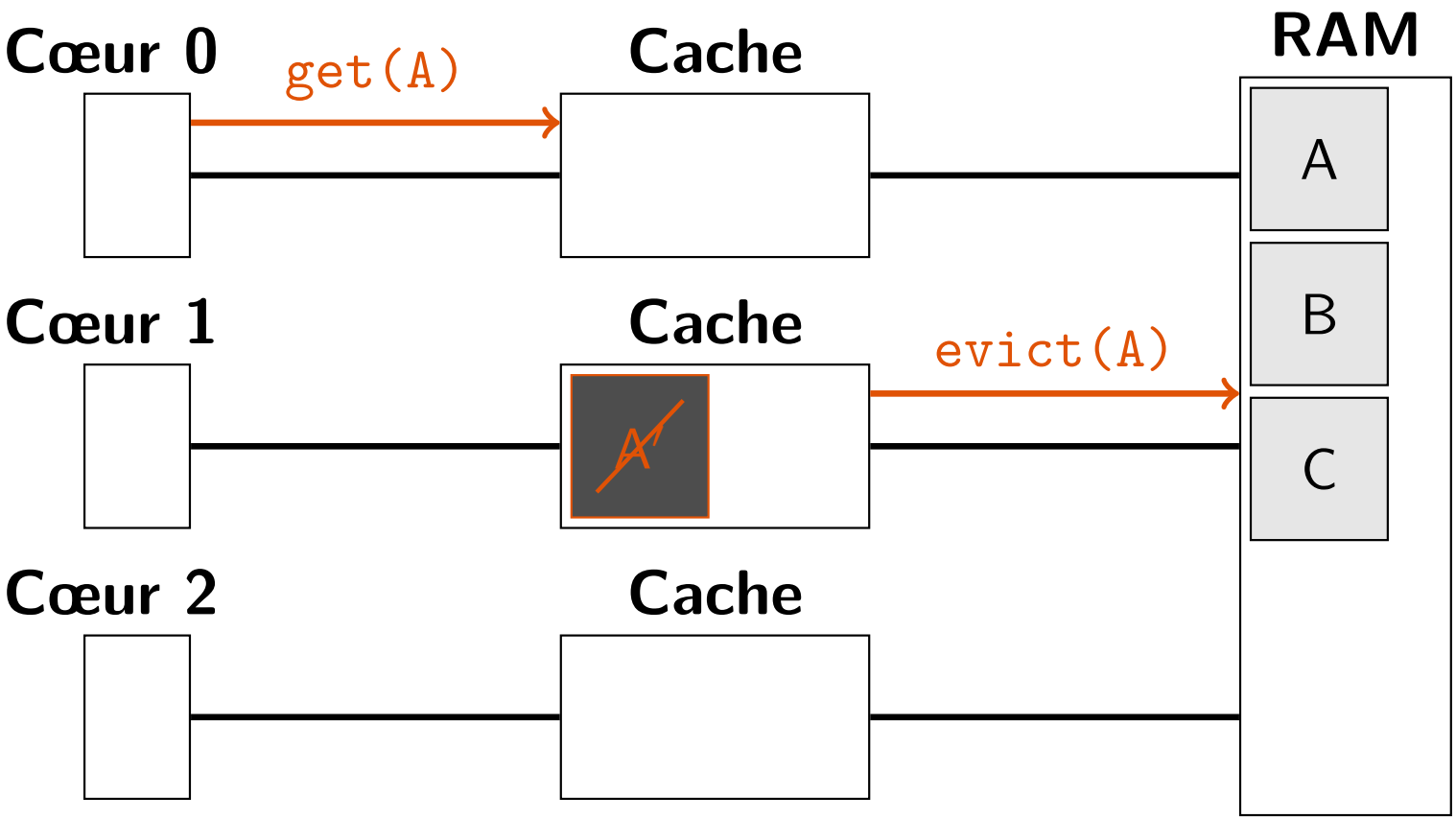
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : WRITE



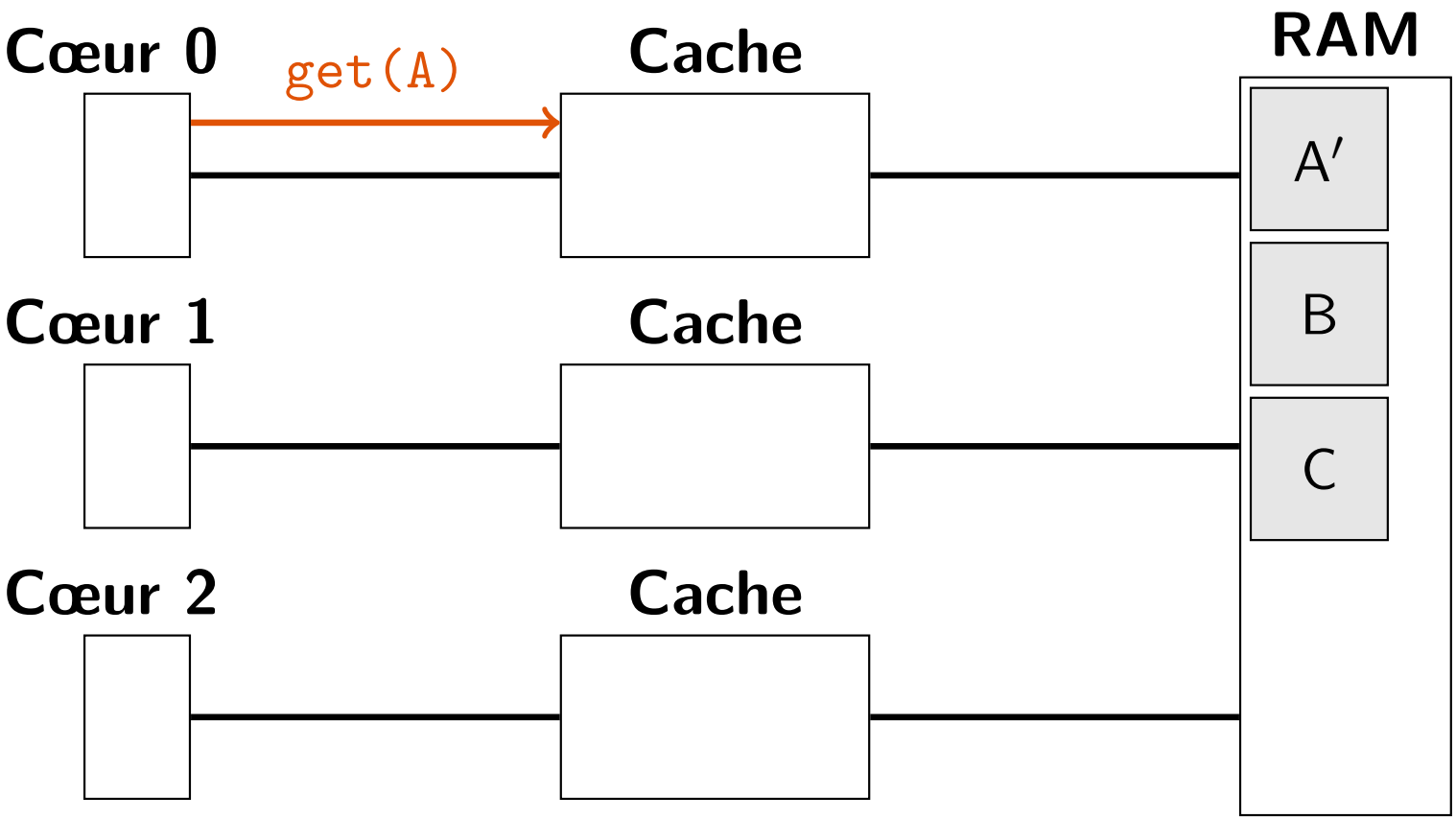
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : WRITE $\rightarrow$ READ



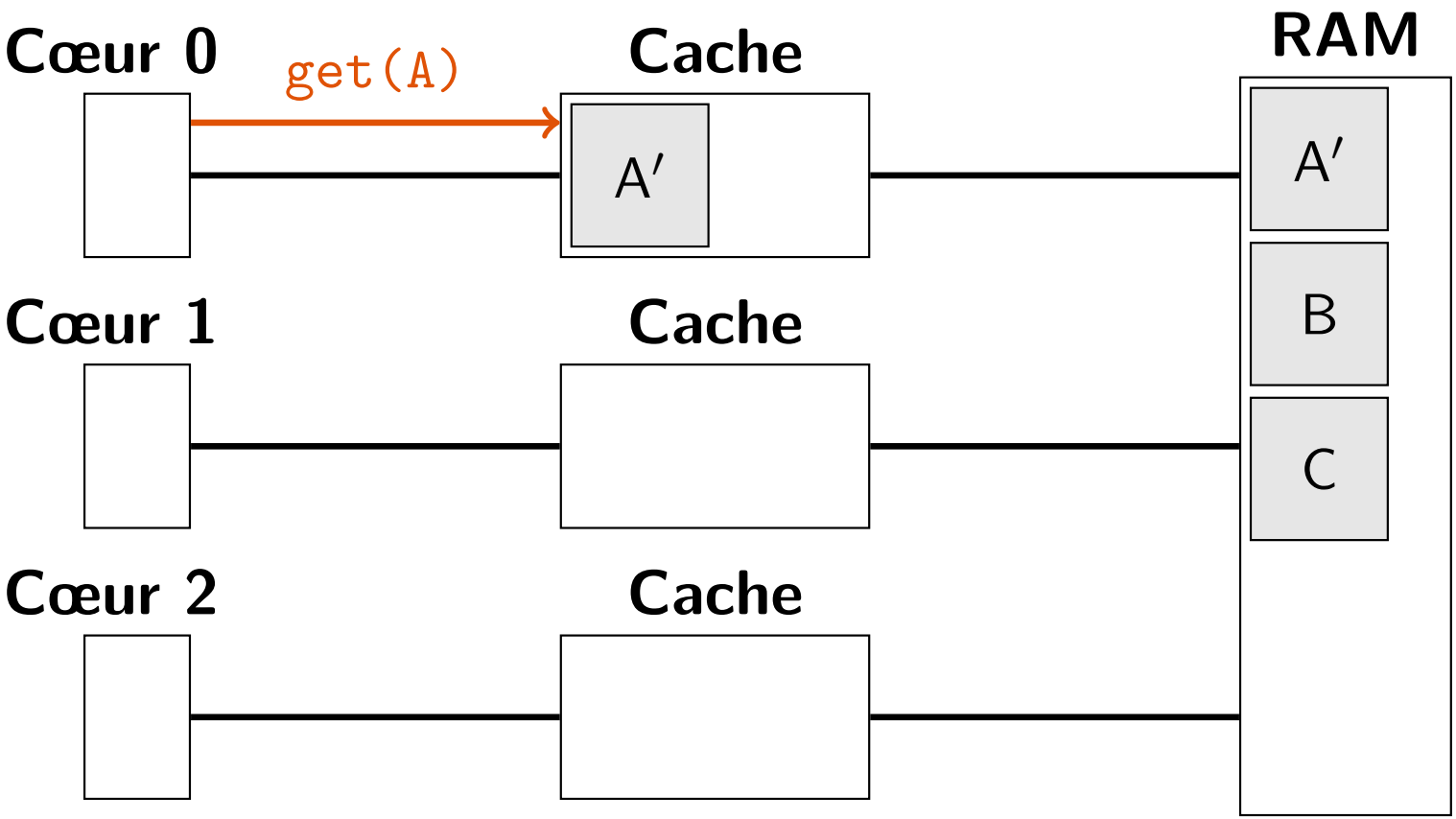
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : WRITE $\rightarrow$ READ



Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : READ

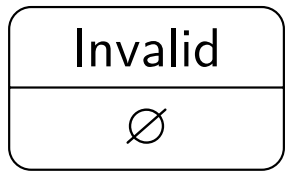


Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne

Le protocole MESI : une implémentation du RW-lock

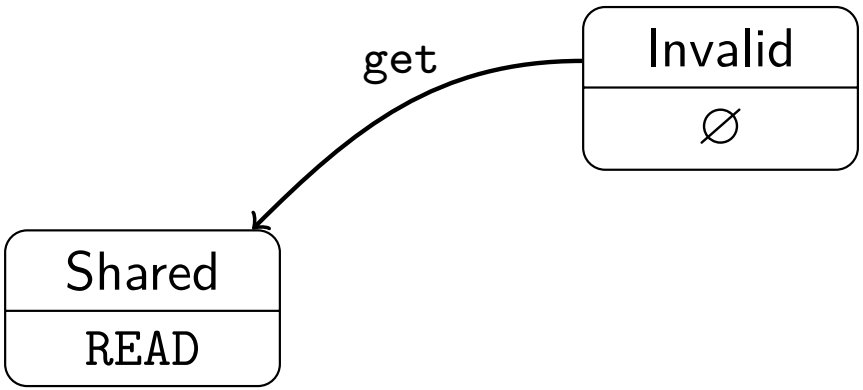
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock



- Invalid : la ligne n'est pas stockée dans le cache

Le protocole MESI : une implémentation du RW-lock

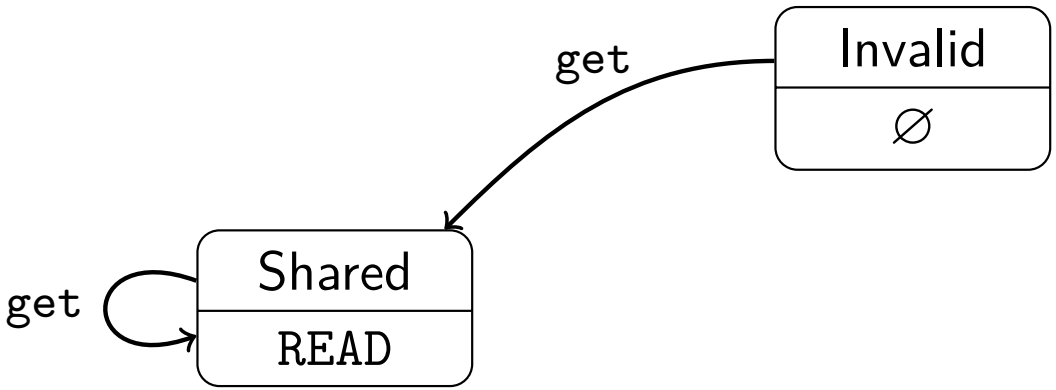
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches

Le protocole MESI : une implémentation du RW-lock

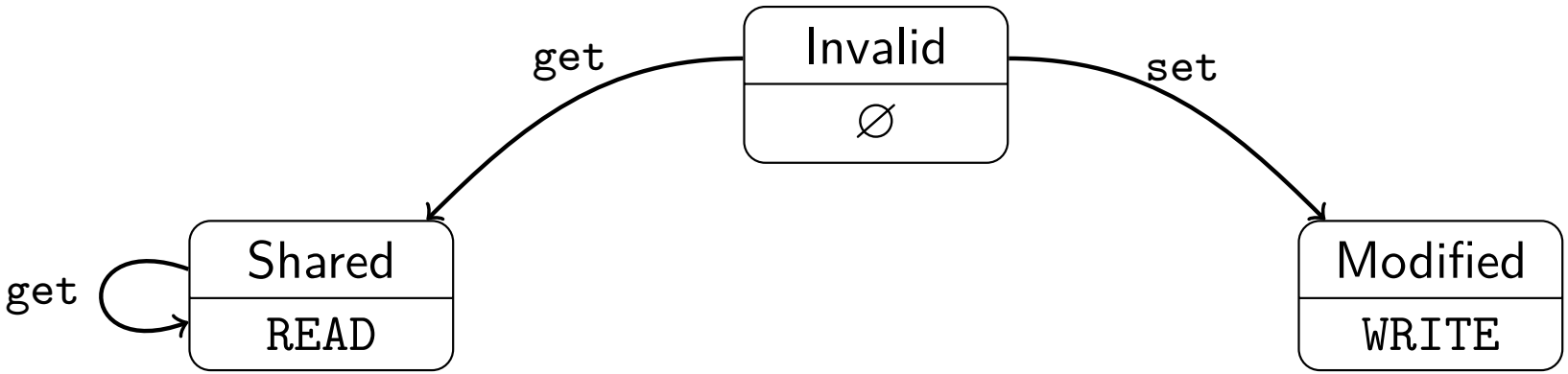
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches

Le protocole MESI : une implémentation du RW-lock

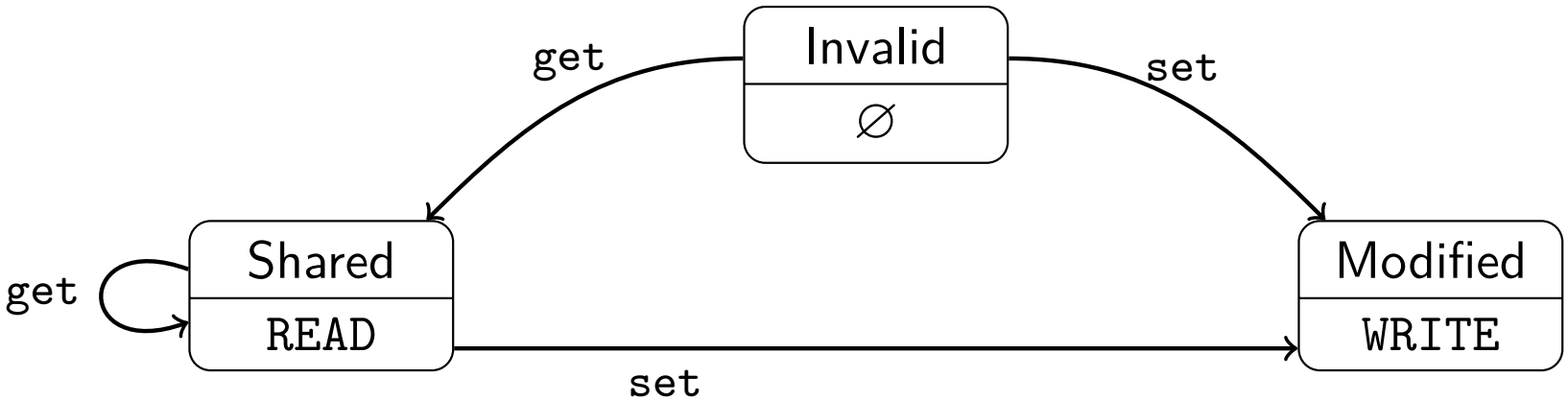
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache

Le protocole MESI : une implémentation du RW-lock

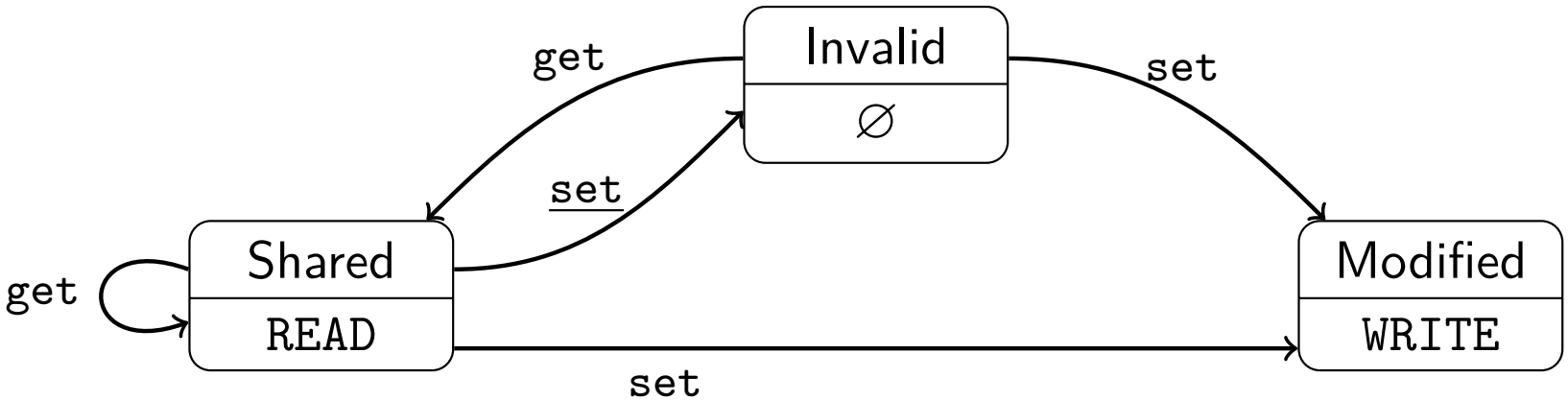
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache

Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour

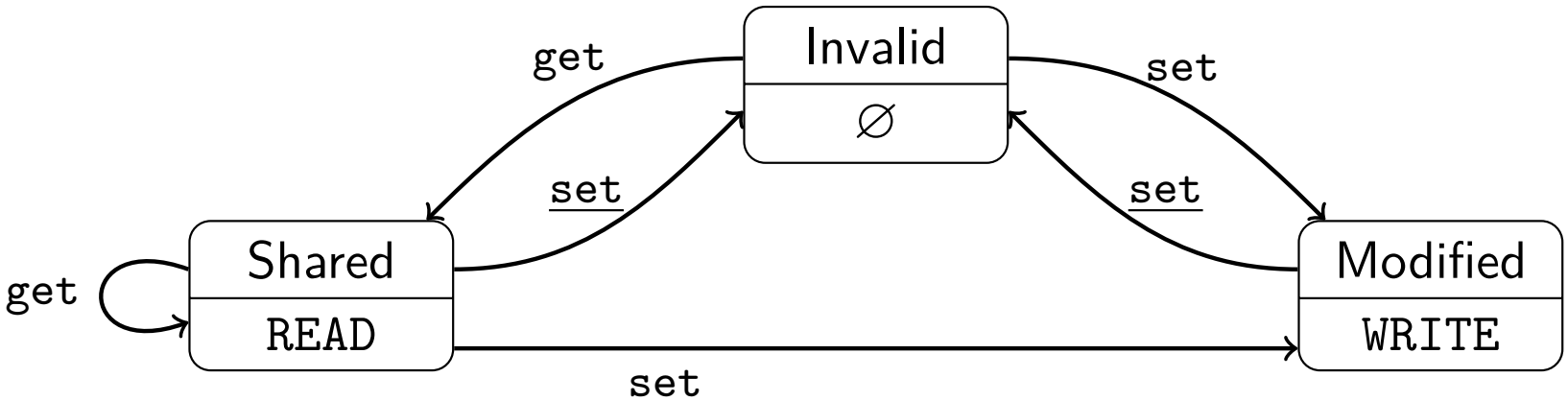


`opération locale`
`opération distante`

- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache

Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour

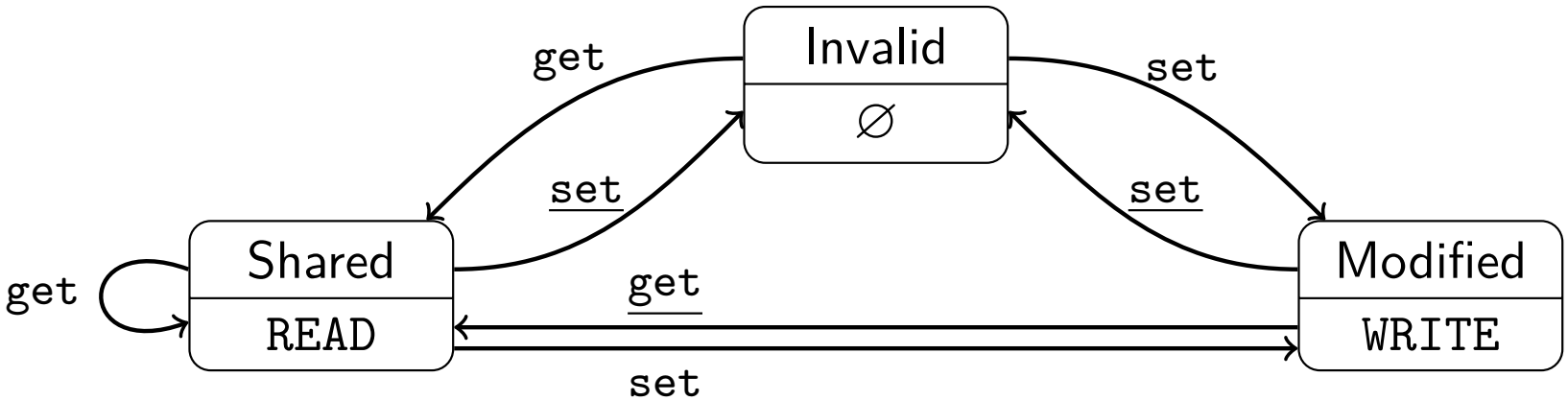


`opération locale`
`opération distante`

- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache

Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour

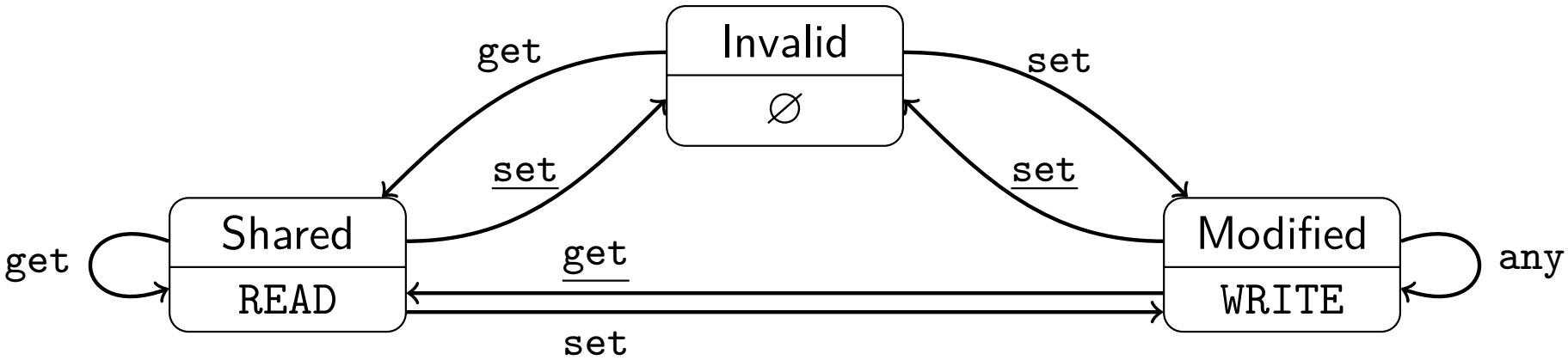


`opération locale`
`opération distante`

- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache

Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour

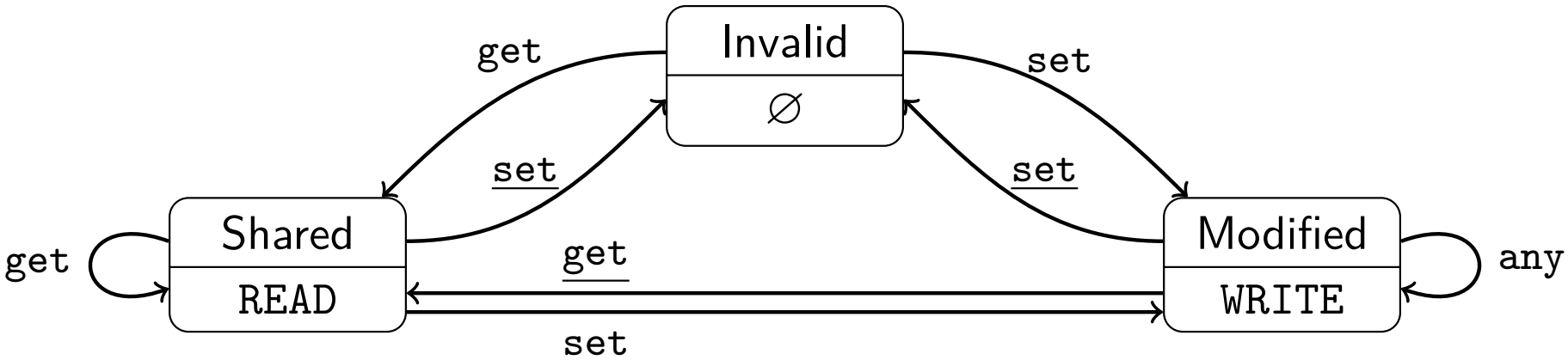


`opération locale`
`opération distante`

- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache

Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour

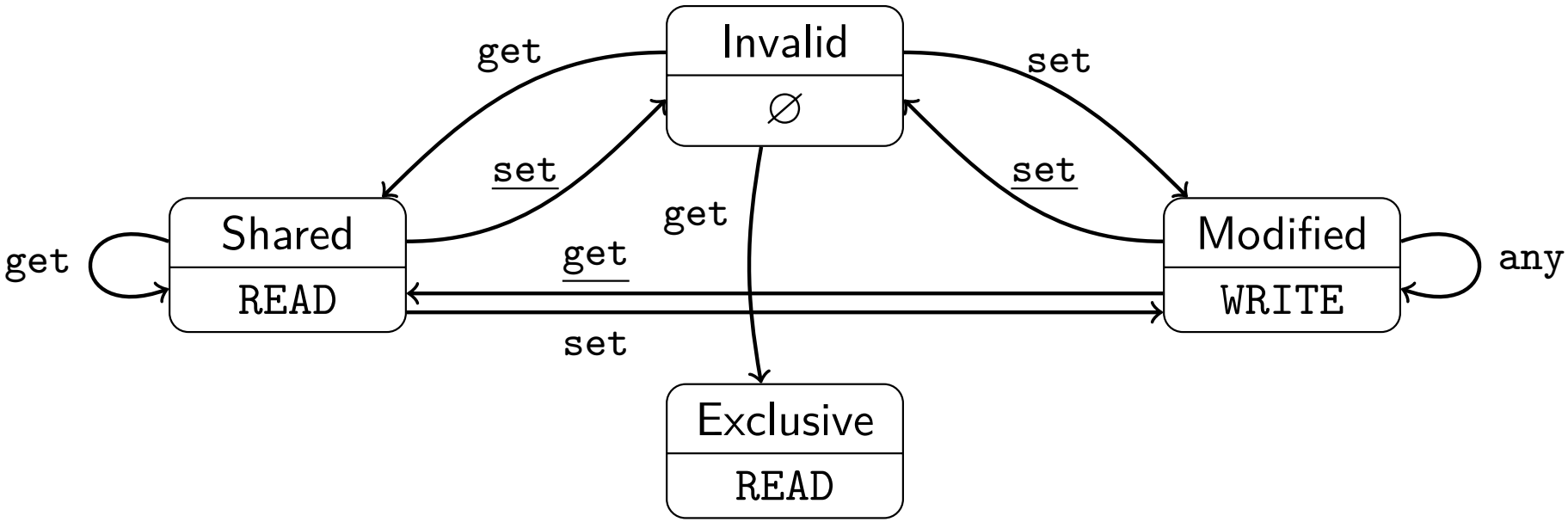


`opération locale`
`opération distante`

- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache
  - La ligne peut être *dirty* uniquement dans cet état

Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour

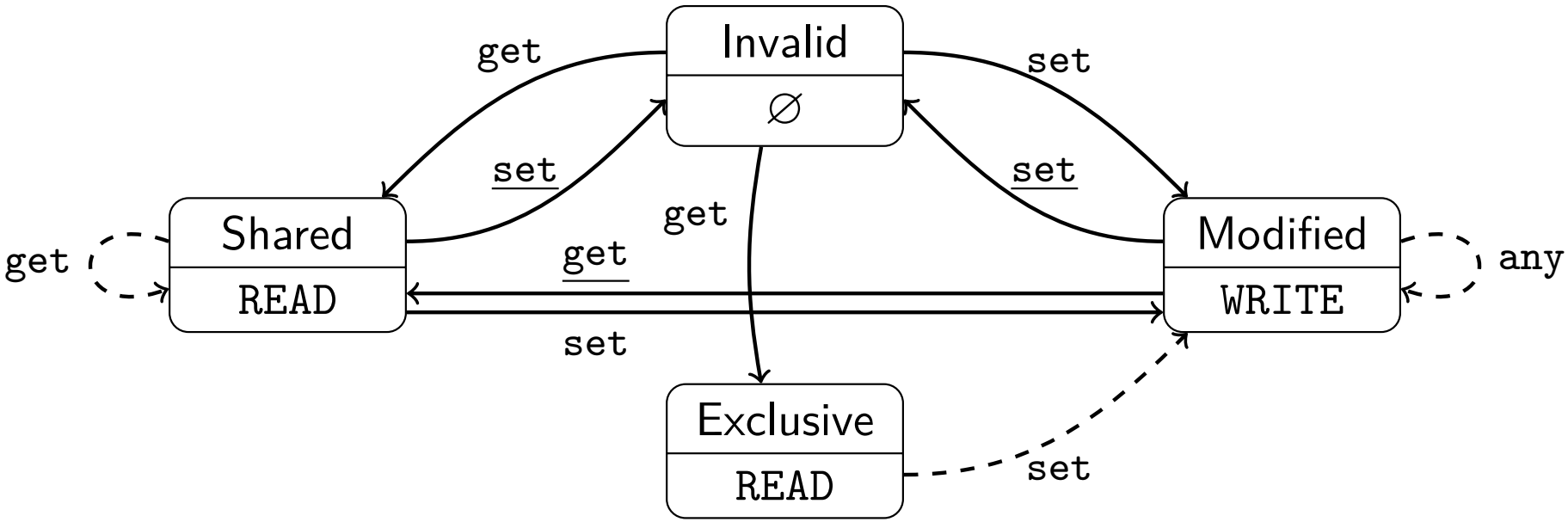


`opération locale`
`opération distante`

- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache
  - La ligne peut être *dirty* uniquement dans cet état
- Exclusive : la ligne est lue par un unique cache

Le protocole MESI : une implémentation du RW-lock

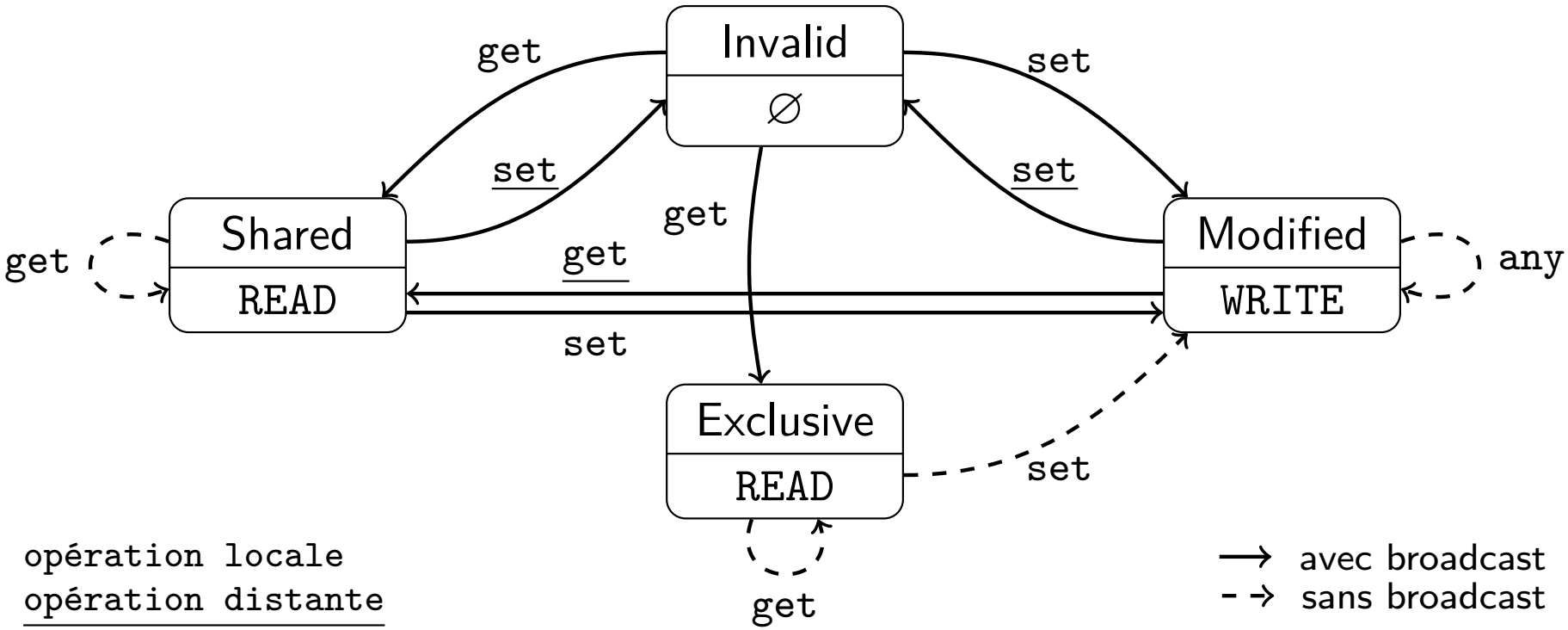
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache
  - La ligne peut être *dirty* uniquement dans cet état
- Exclusive : la ligne est lue par un unique cache

Le protocole MESI : une implémentation du RW-lock

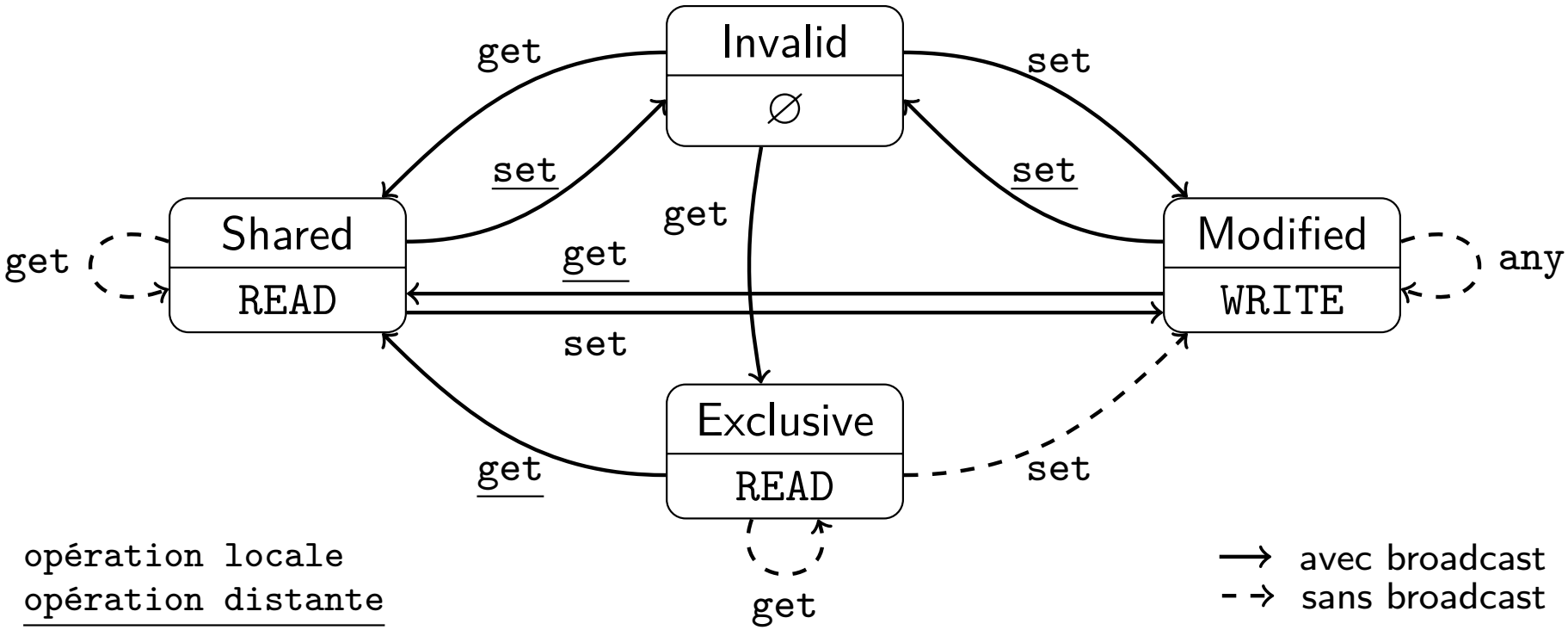
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache
  - La ligne peut être *dirty* uniquement dans cet état
- Exclusive : la ligne est lue par un unique cache

Le protocole MESI : une implémentation du RW-lock

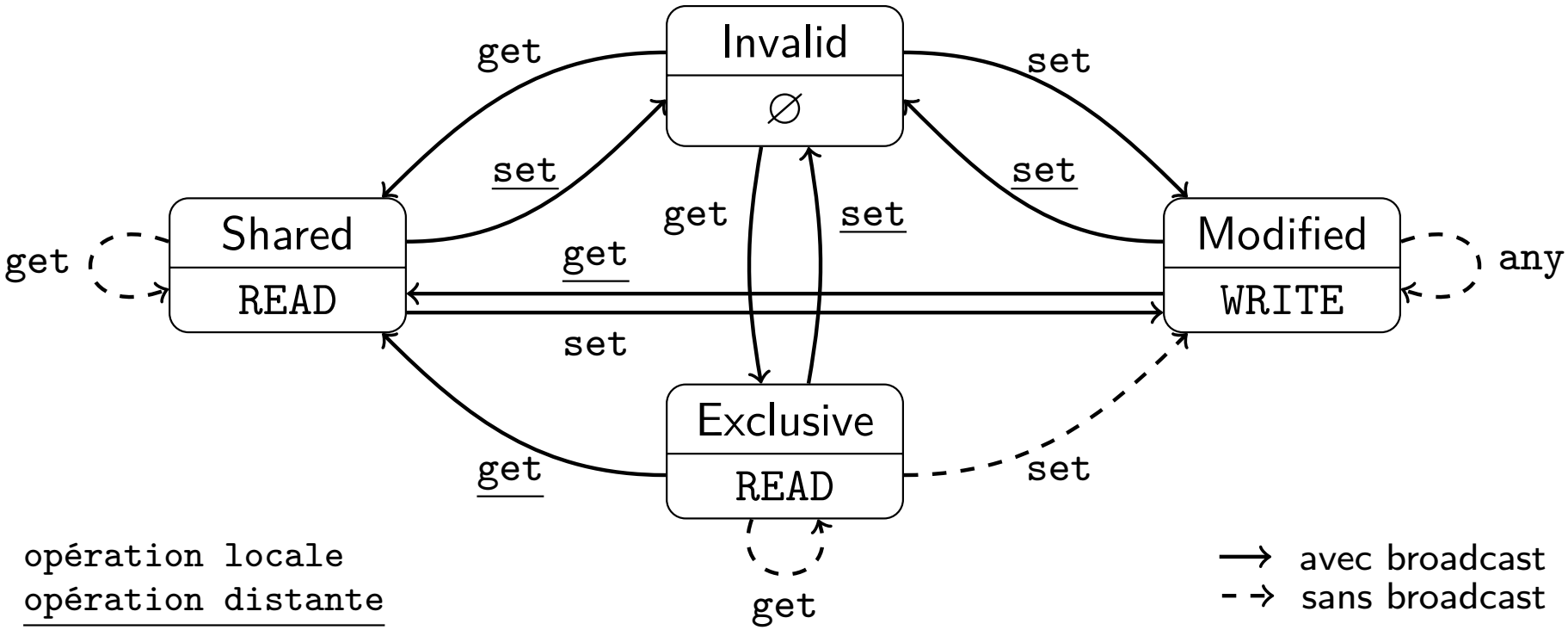
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache
  - La ligne peut être *dirty* uniquement dans cet état
- Exclusive : la ligne est lue par un unique cache

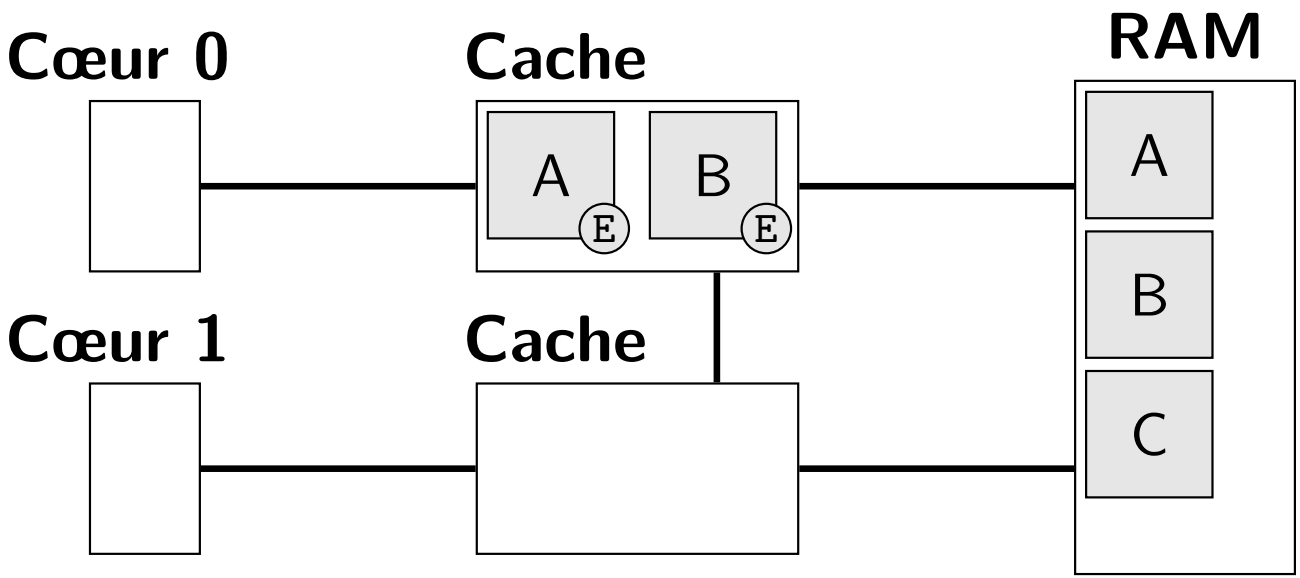
Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache
  - La ligne peut être *dirty* uniquement dans cet état
- Exclusive : la ligne est lue par un unique cache

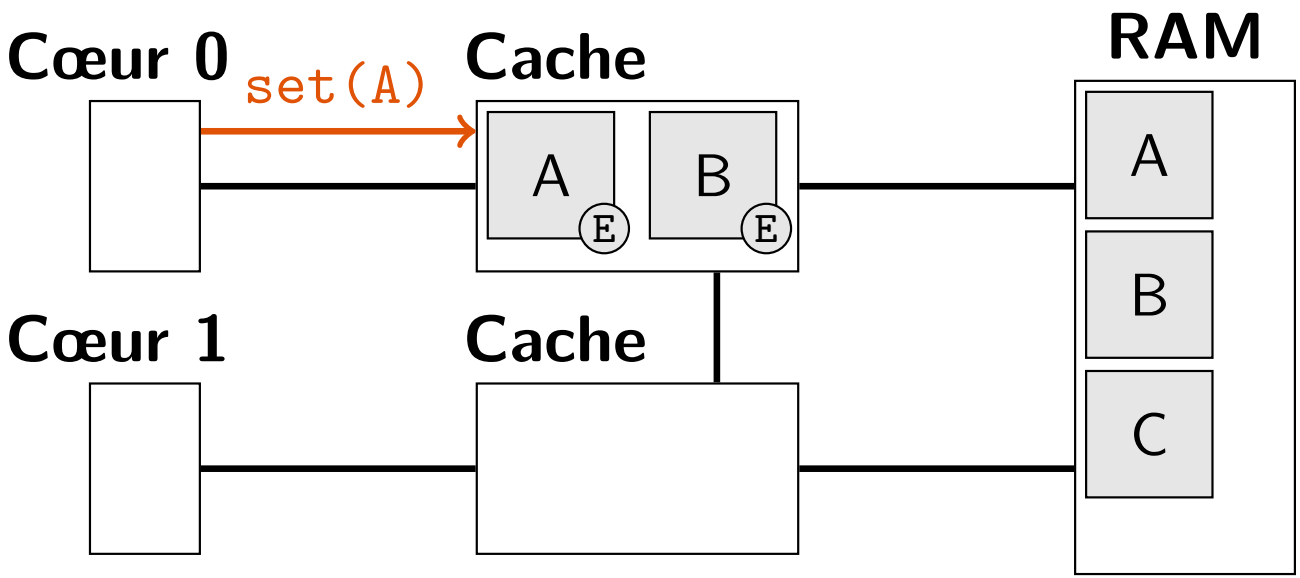
Protocol MESI : exemple




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

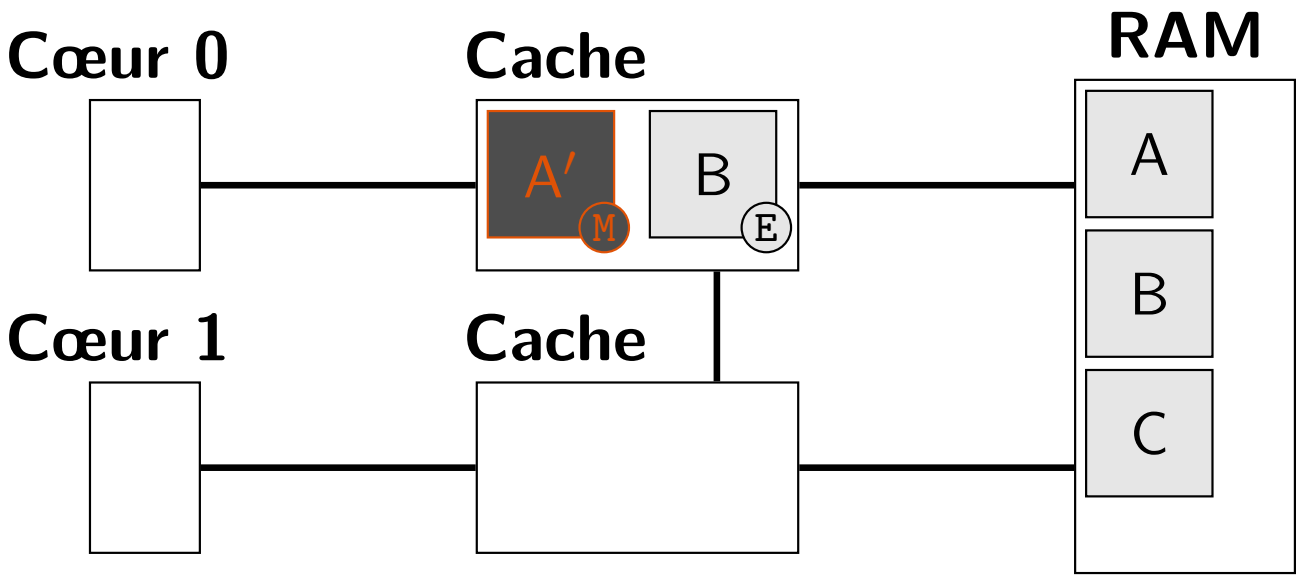
Protocol MESI : exemple




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

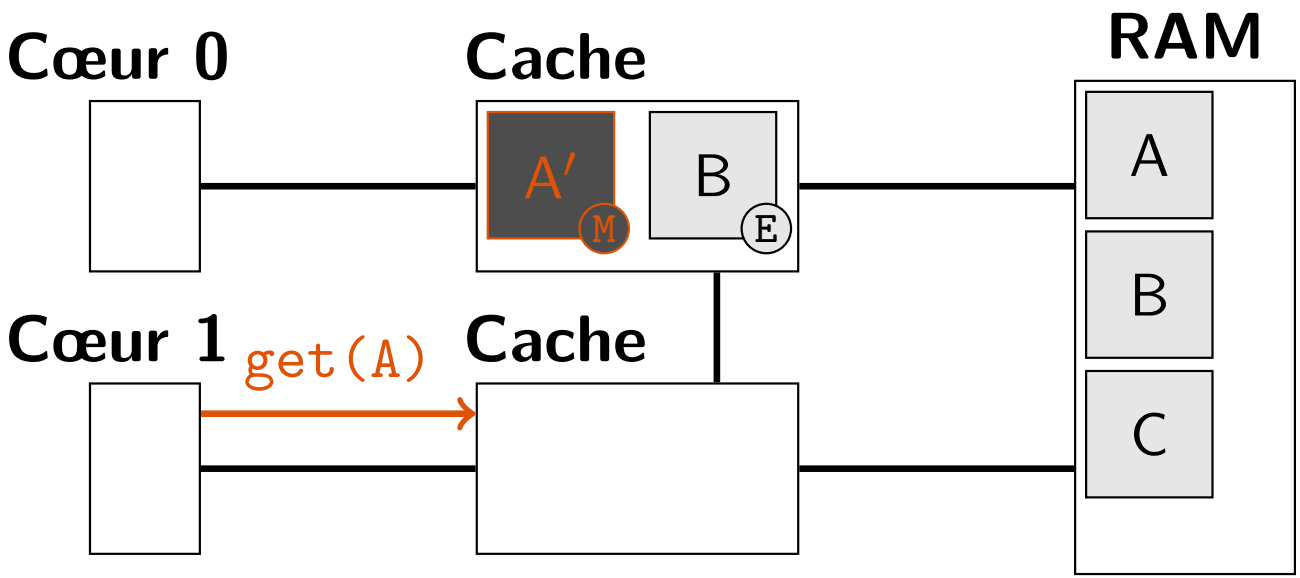
Protocol MESI : exemple




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

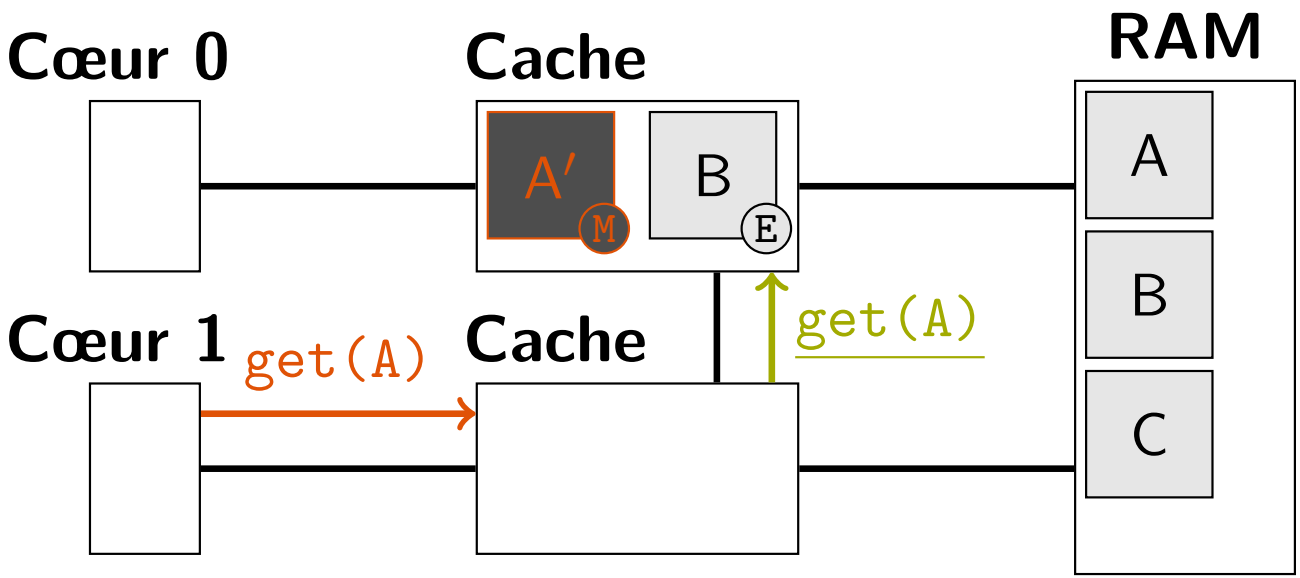
Protocol MESI : exemple




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

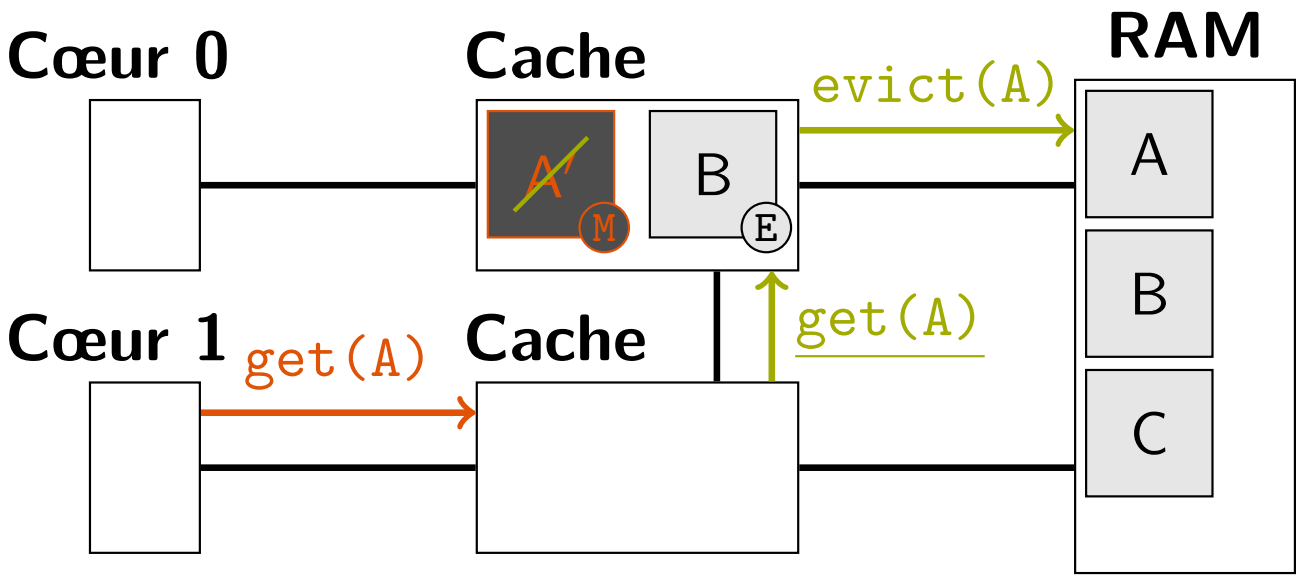
Protocol MESI : exemple




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

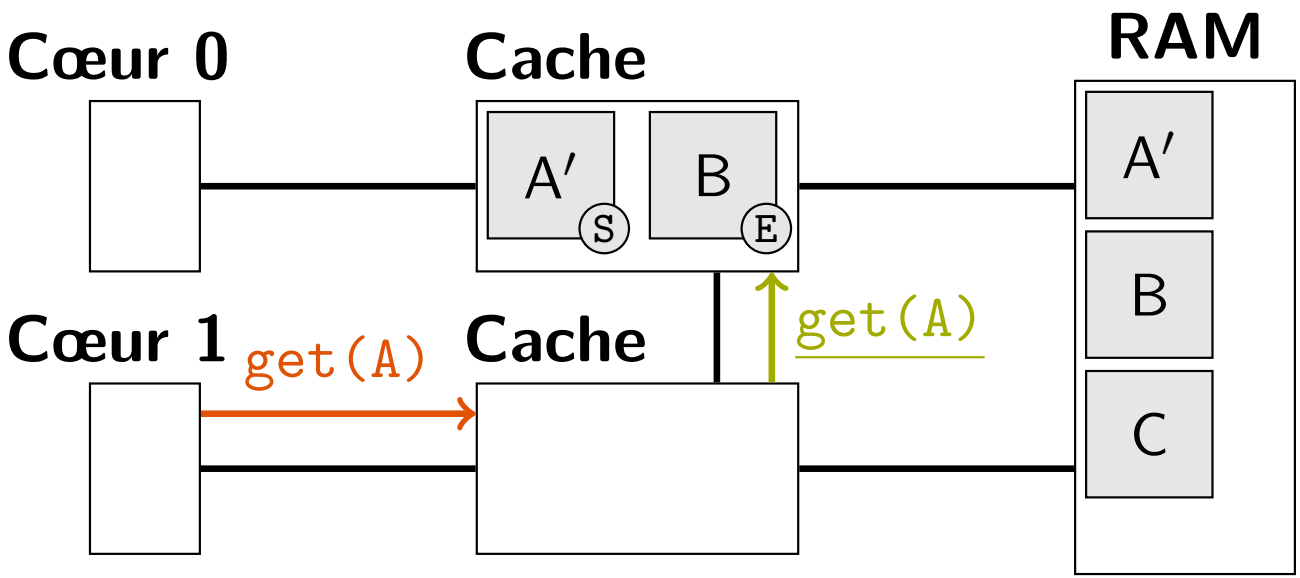
Protocol MESI : exemple




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

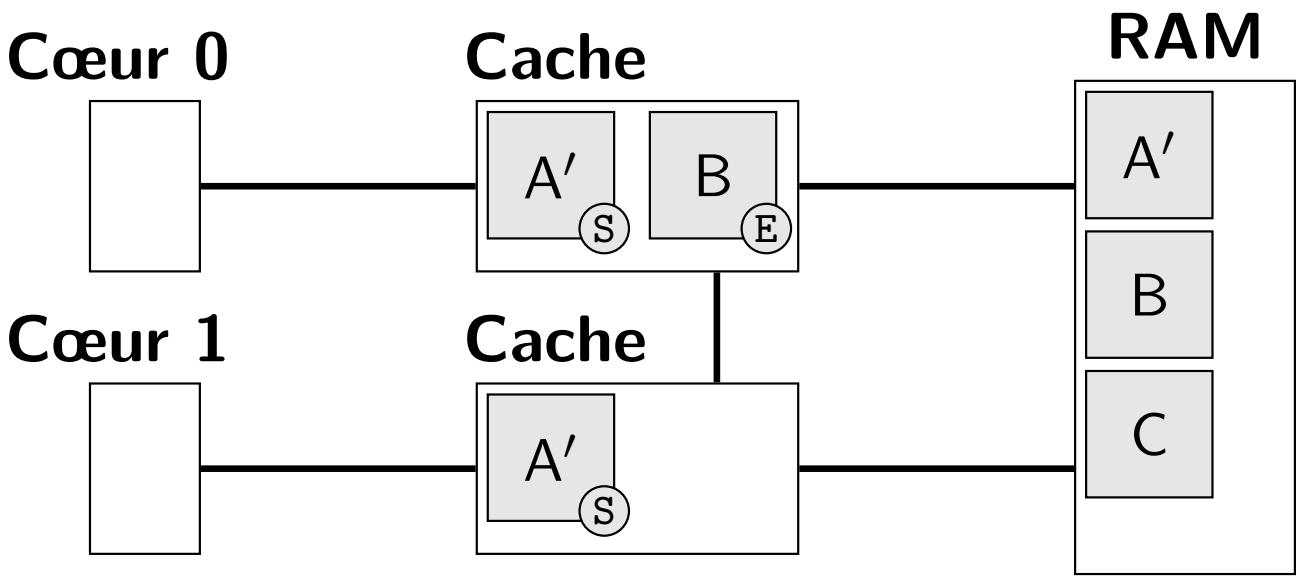



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

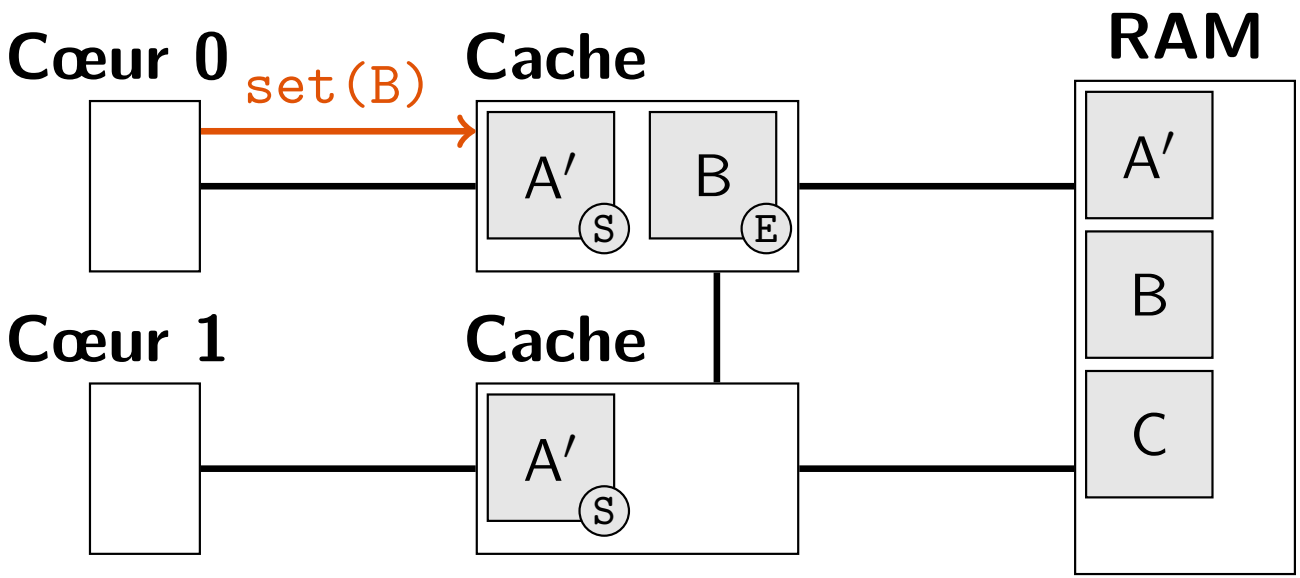



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

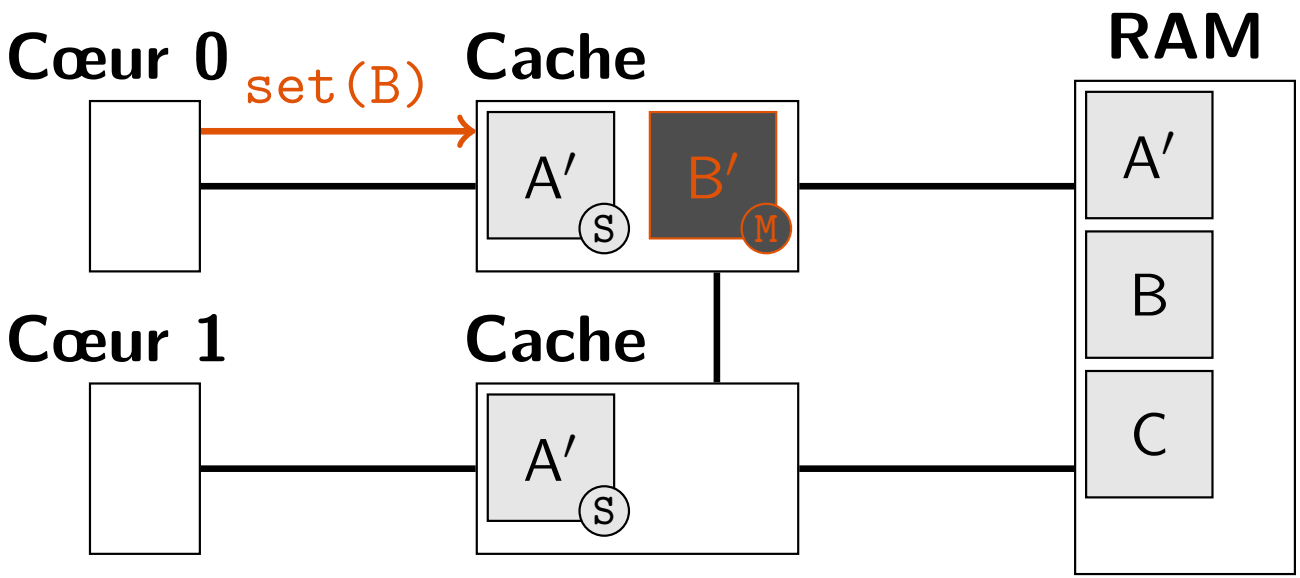



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

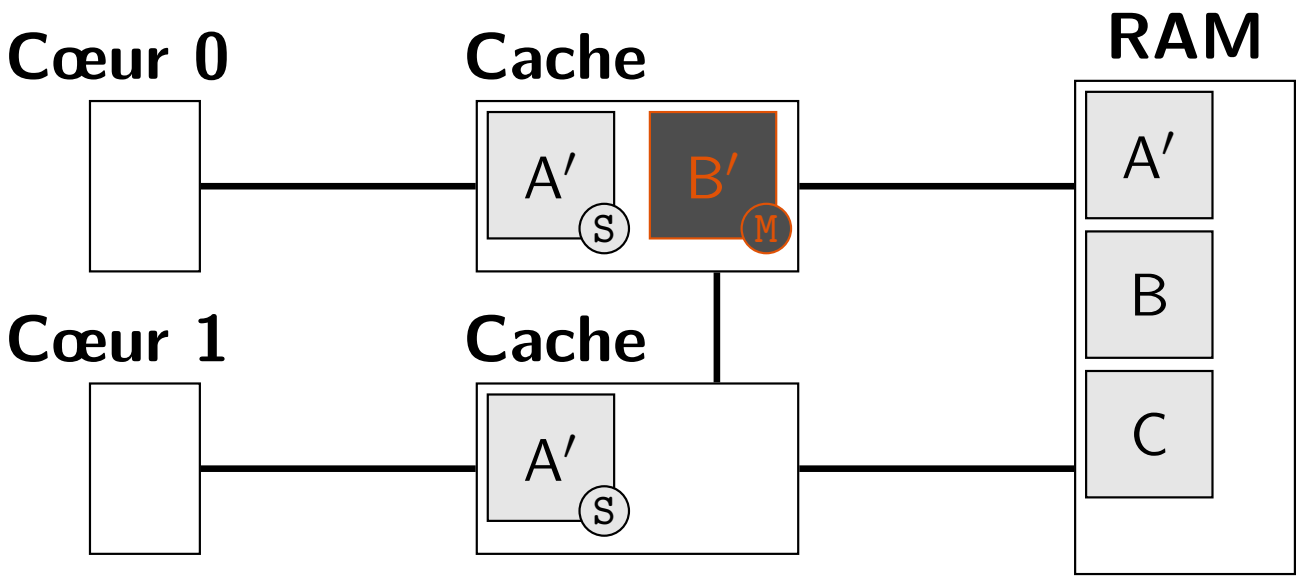



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

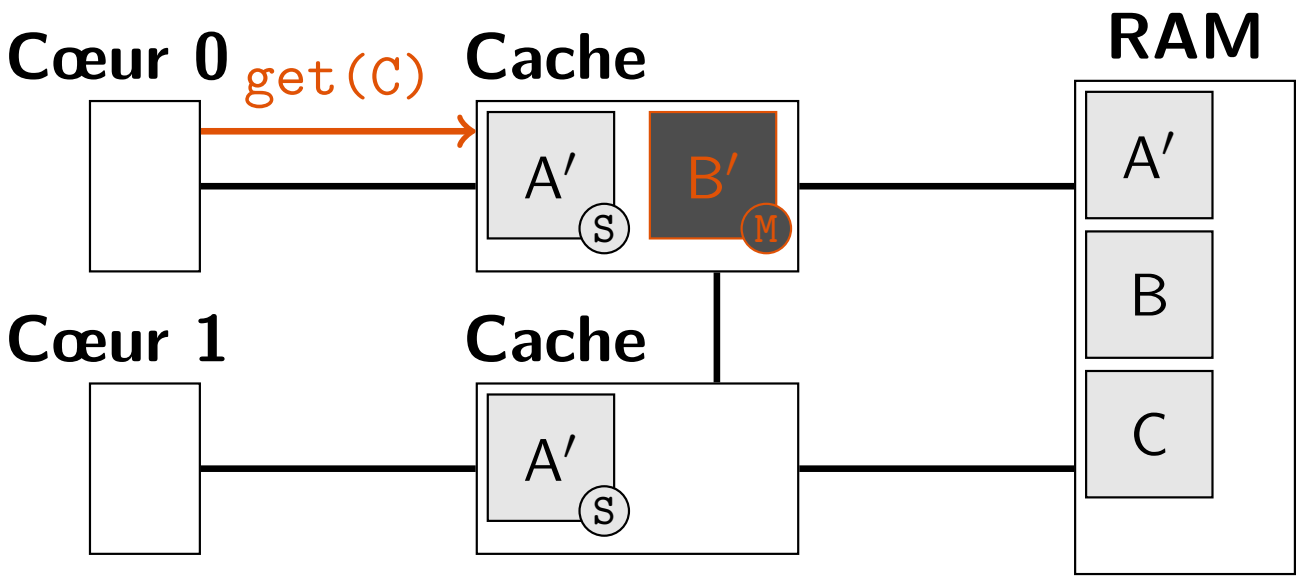



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

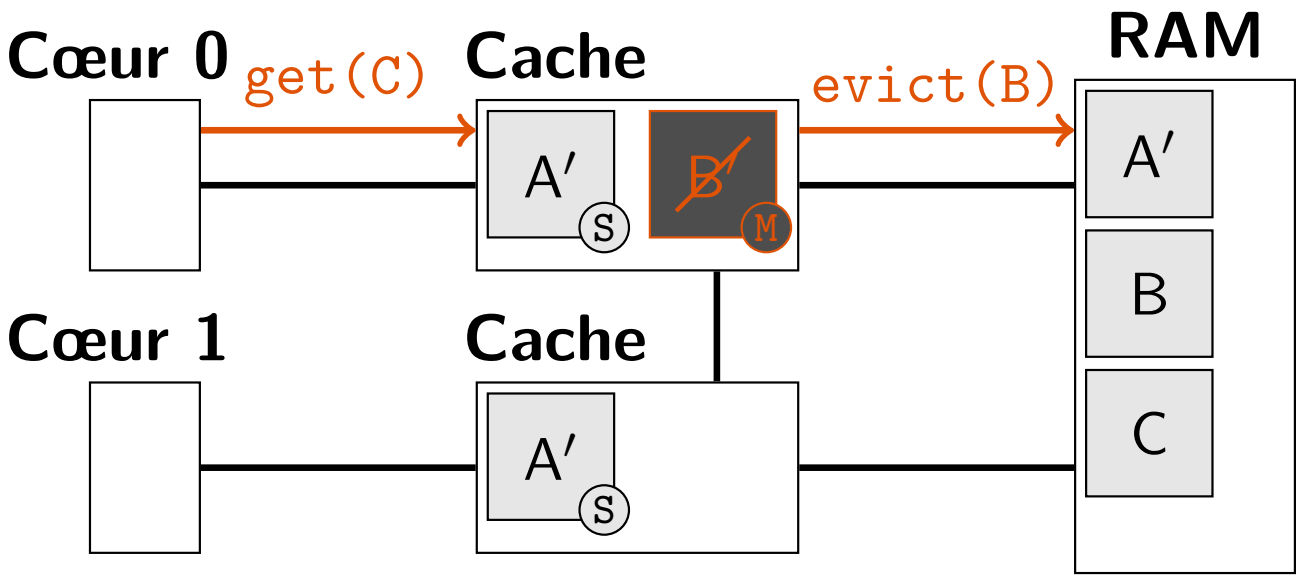



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

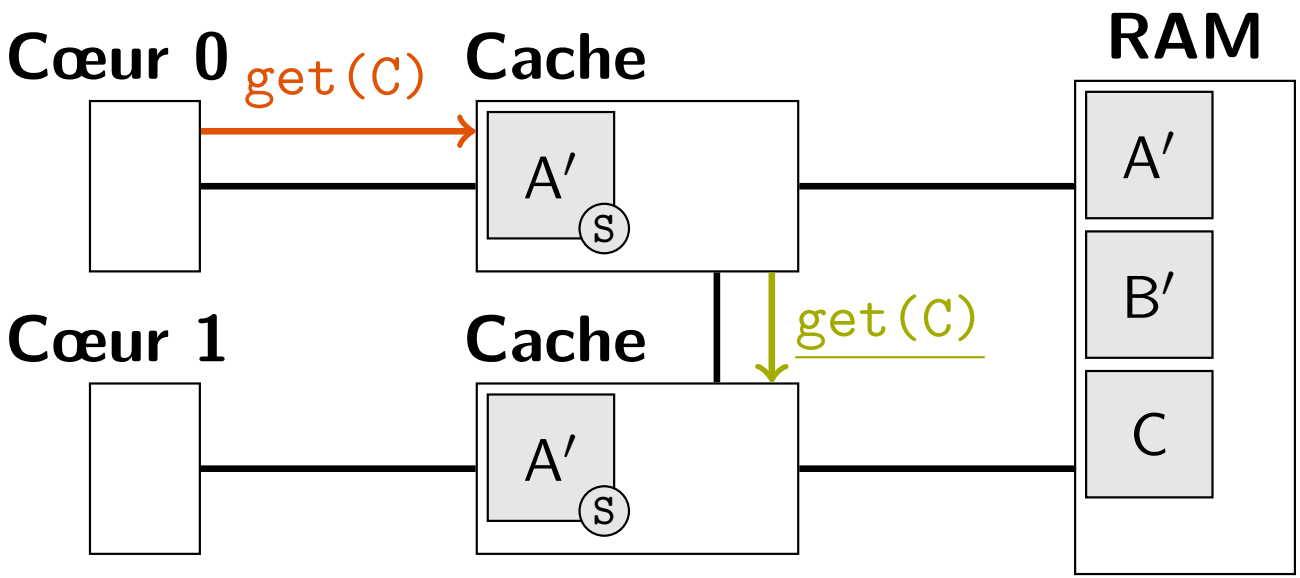



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

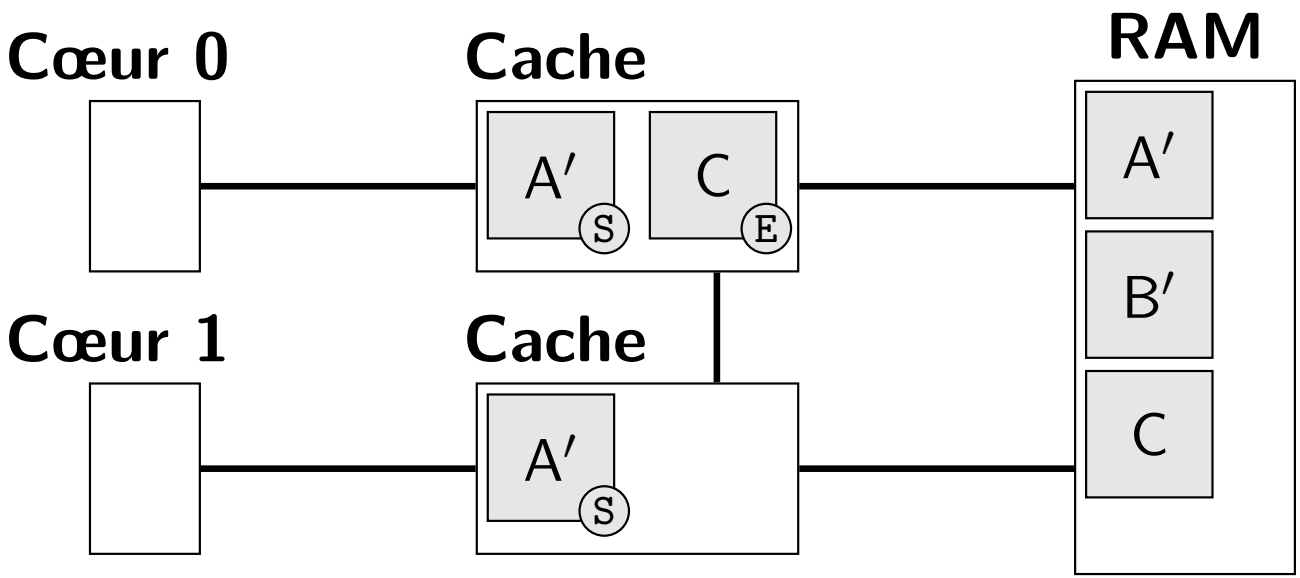



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

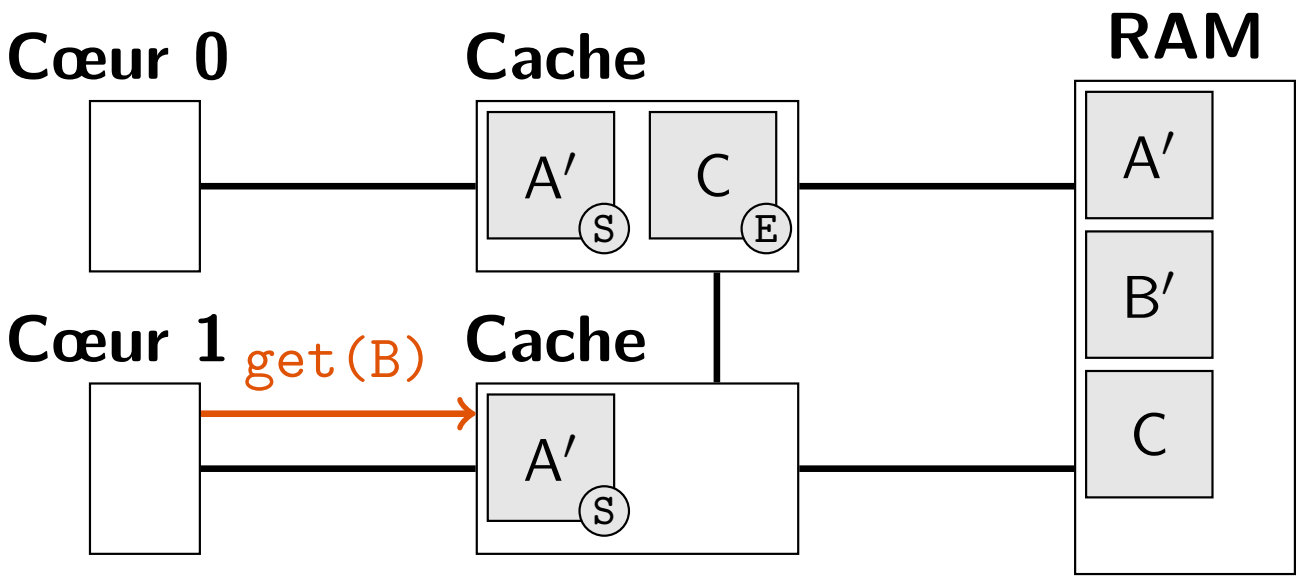



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

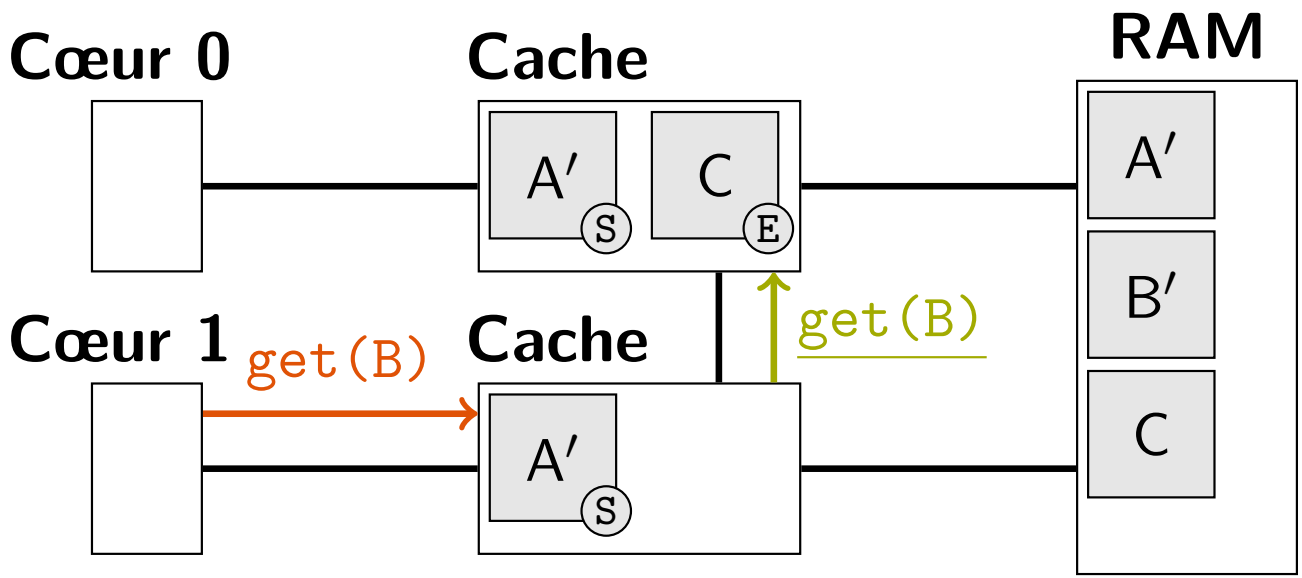



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

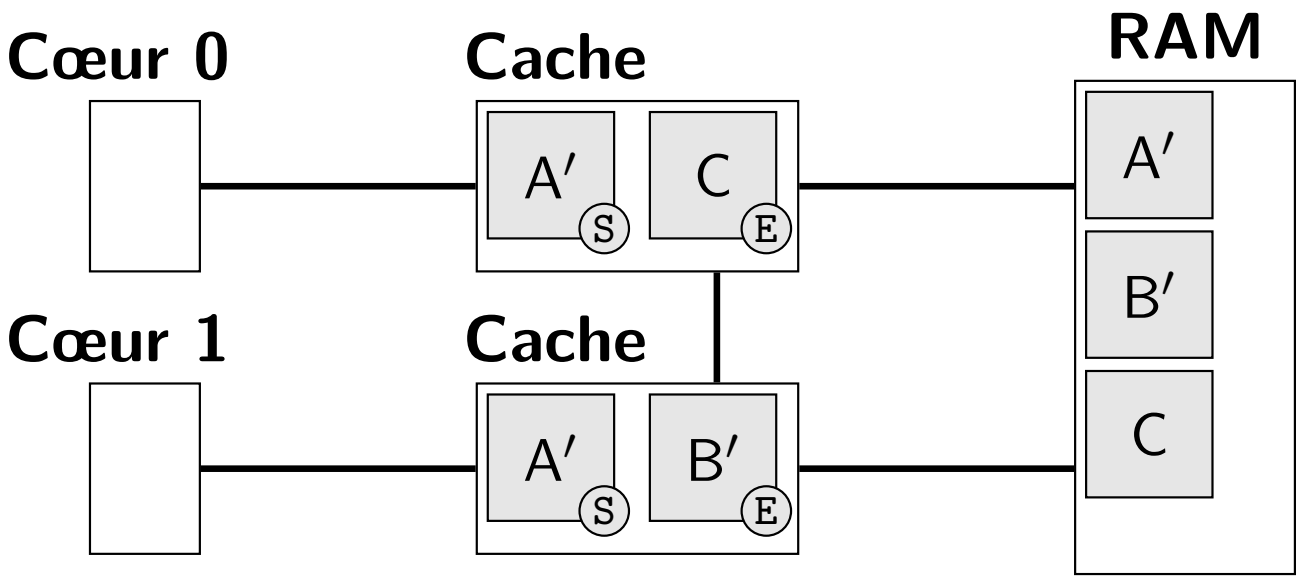



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

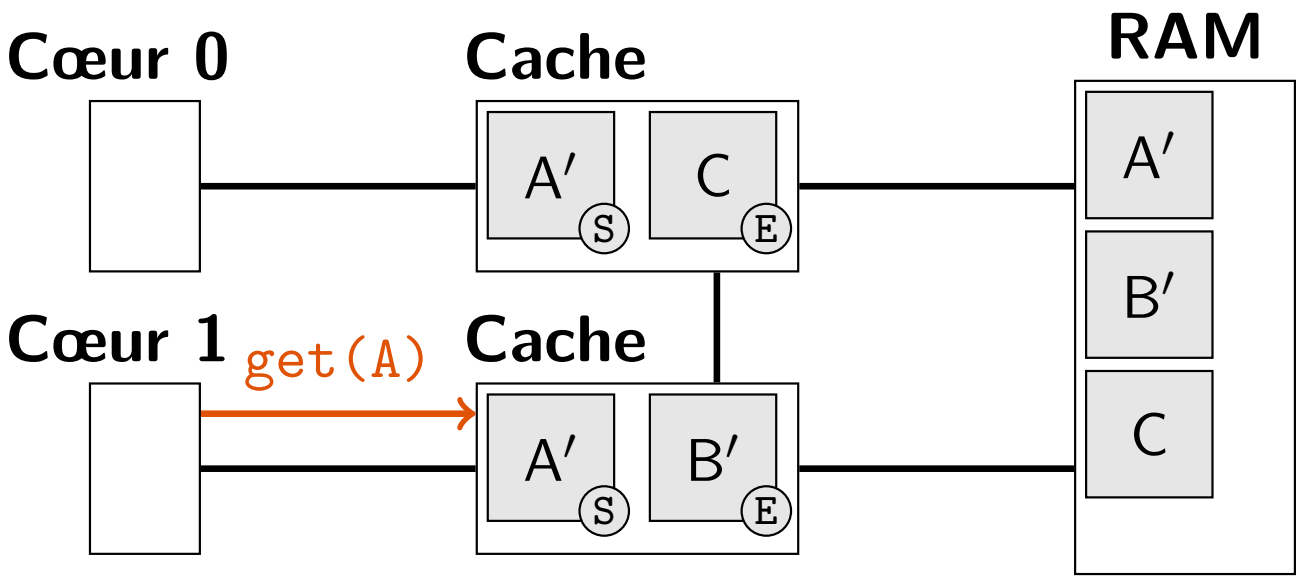



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

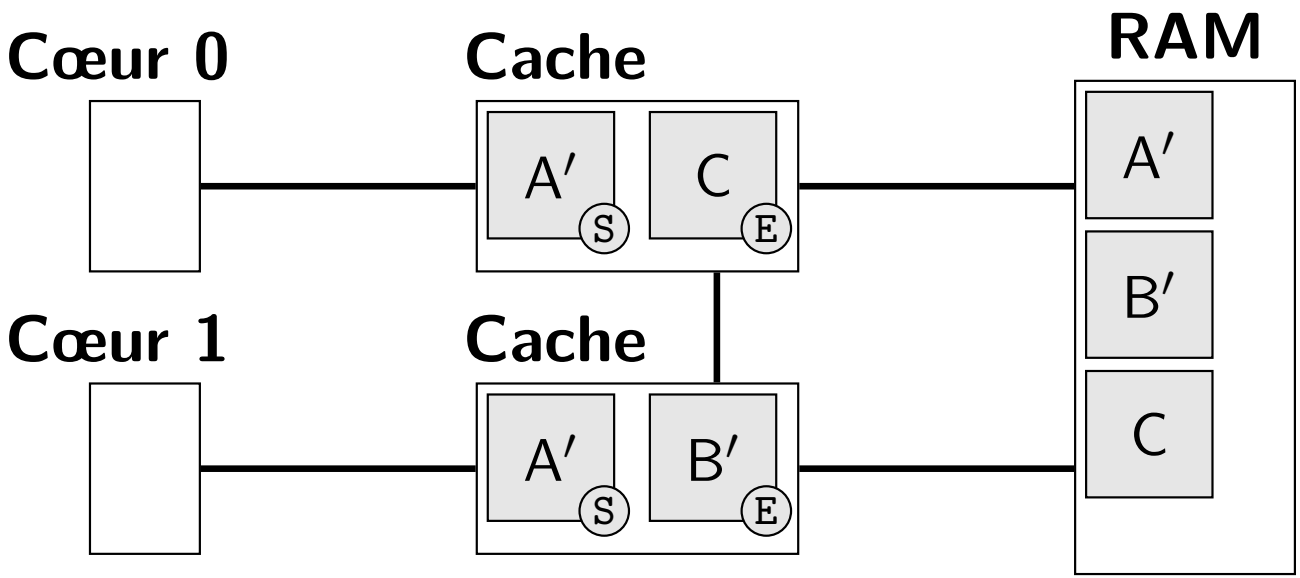



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

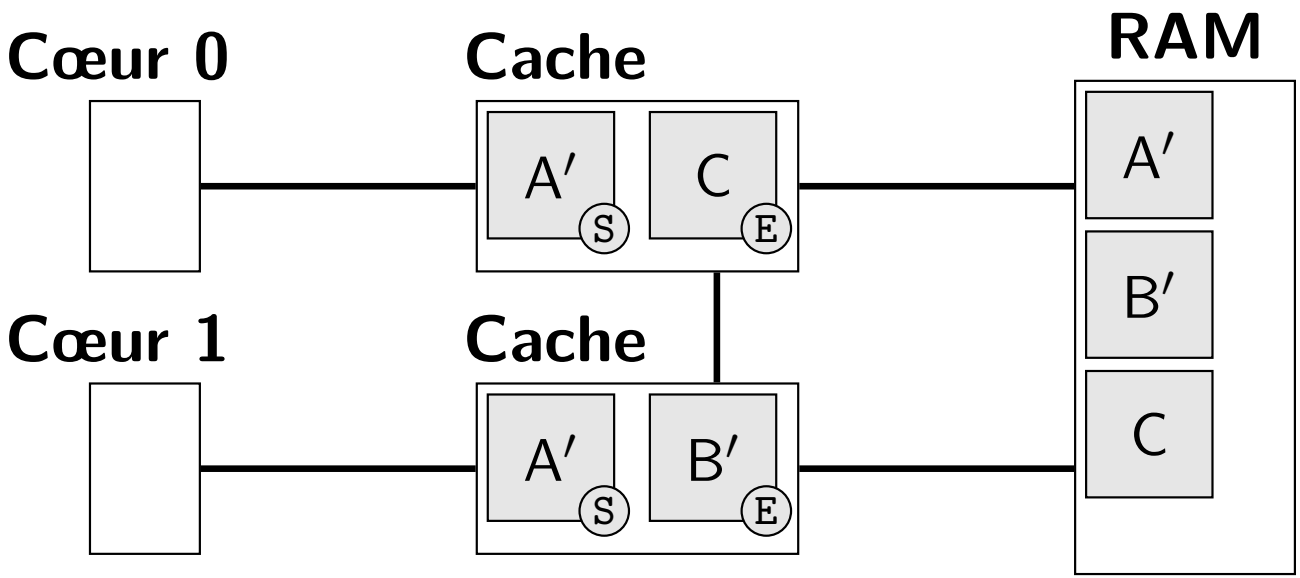



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne
- Par conséquent, la cohérence séquentielle est assurée




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

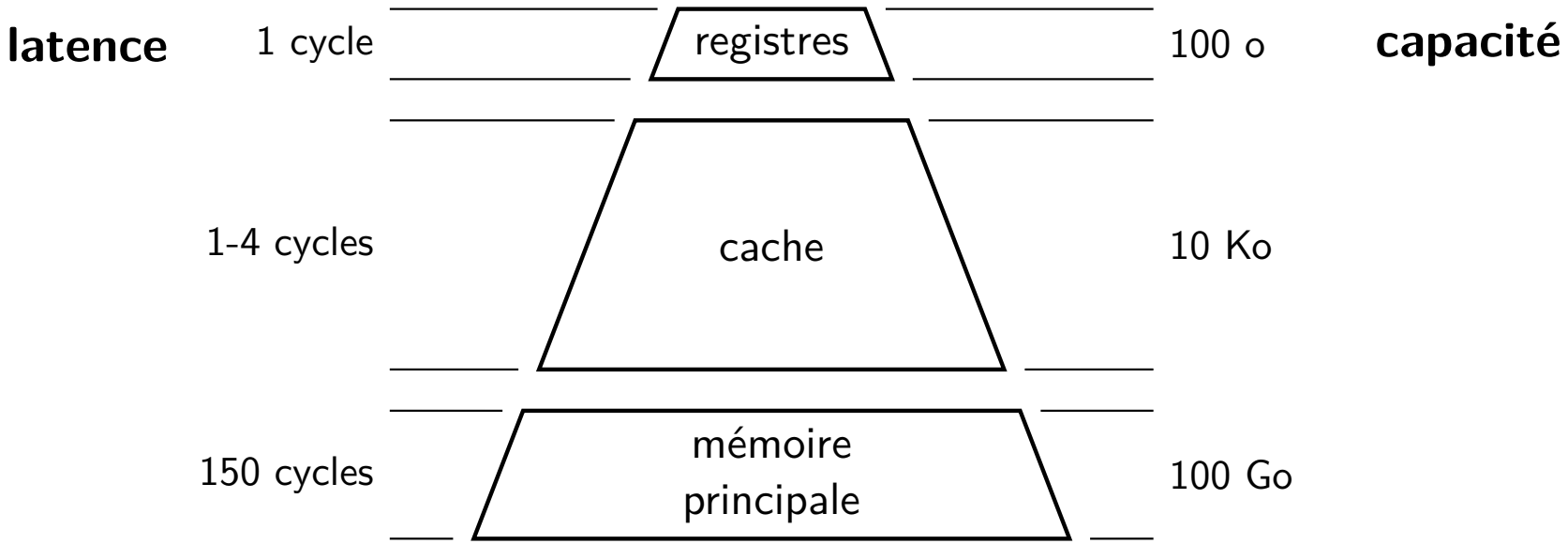
Protocole MESI : résumé

- Les mémoires cache actuelles assurent la cohérence séquentielle
  - Tout cœur travaille toujours sur la dernière version de chaque ligne
  - Chaque ligne est protégée par un verrou lecteur/écrivain distribué

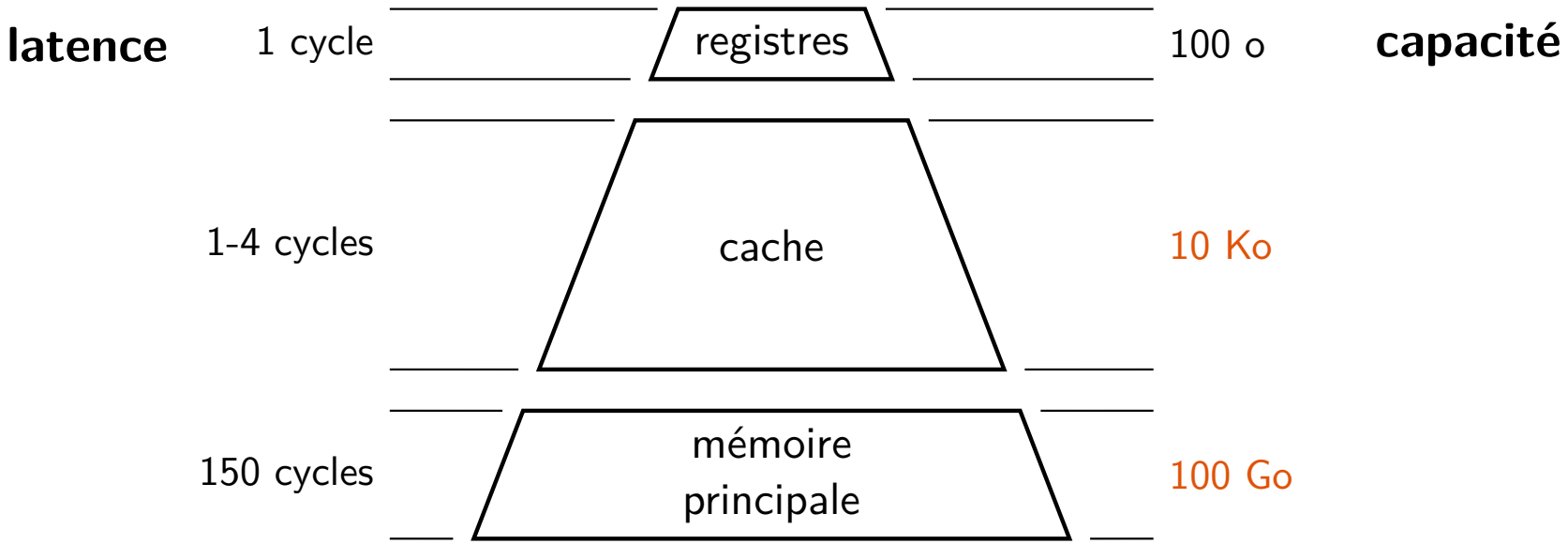
- Une implémentation du verrou lecteur/écrivain est le protocole MESI
  - Le protocole MESI est implémenté par les caches Intel
  - Chaque ligne stockée en cache a un état
  - Invalid $= \varnothing$, Shared $=$ `READ`, Modified $=$ `WRITE`
  - Exclusive $=$ `READ` $\rightarrow$ optimisation en cas d'accesseur unique
  - Les caches maintiennent les états de chaque ligne à jour en communiquant

- Il existe plusieurs variantes de MESI $\rightarrow$ MOESI
  - Le protocole MOESI est implémenté par les caches AMD
  - Shared $=$ `READ` mais peut-être *dirty*
  - Owned $=$ `READ` mais *dirty* et responsable de la propagation en RAM
  - Une ligne Modified passe à Owned en cas de get distant

Topologie de caches : retour sur la hiérarchie mémoire

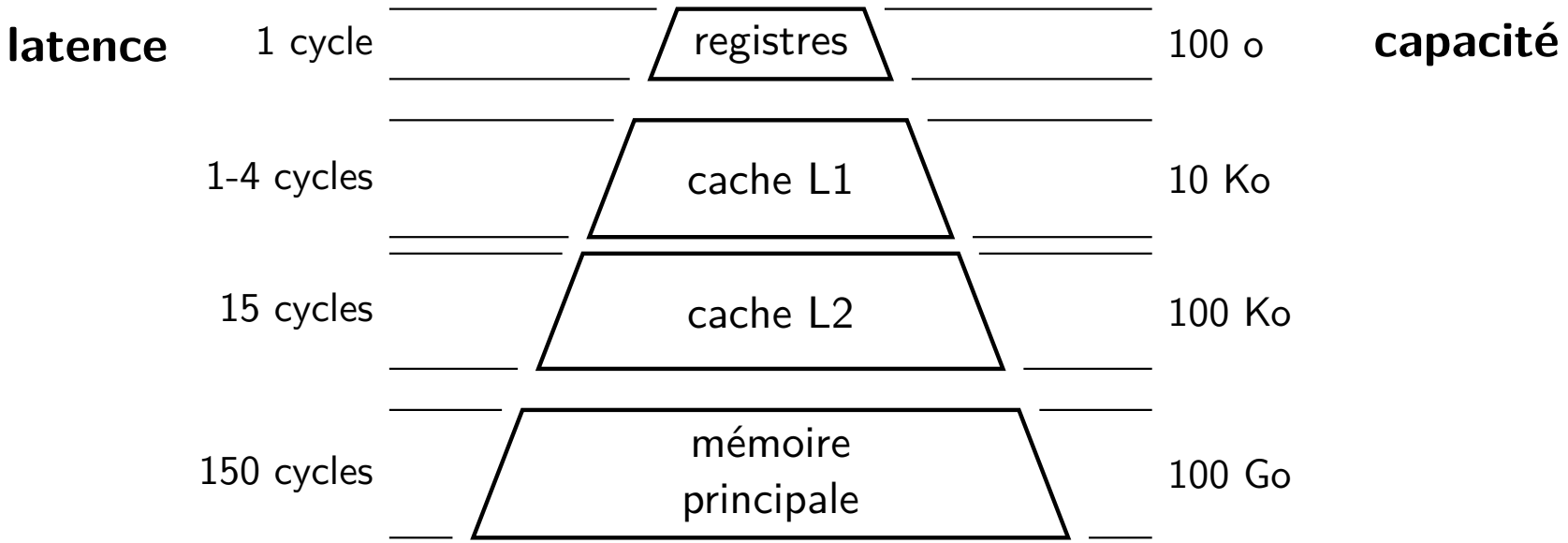


Topologie de caches : retour sur la hiérarchie mémoire



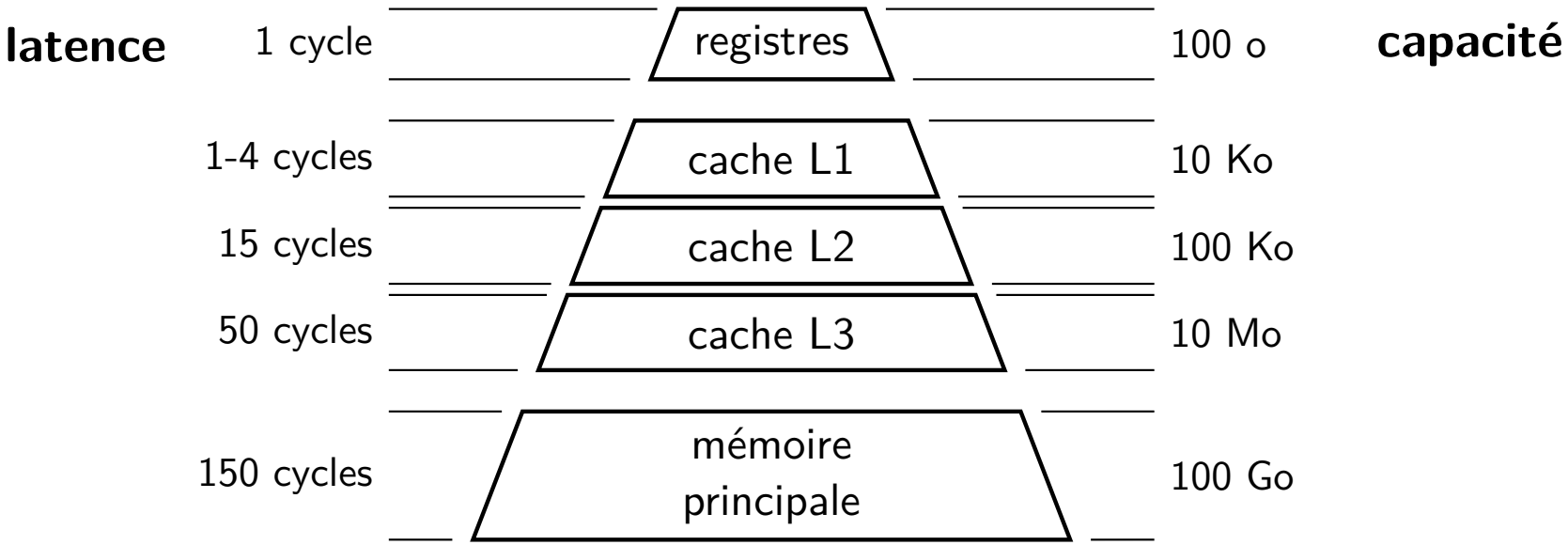
- Un cache de 10 Ko est très petit devant la mémoire (100 Go)
  - Beaucoup de cache miss, même avec une bonne localité
- Augmenter la taille du cache $\rightarrow$ impossible sans augmenter la latence

Topologie de caches : retour sur la hiérarchie mémoire



- Un cache de 10 Ko est très petit devant la mémoire (100 Go)
  - Beaucoup de cache miss, même avec une bonne localité
- Augmenter la taille du cache $\rightarrow$ impossible sans augmenter la latence
- Ajouter un second cache entre le premier cache et la mémoire

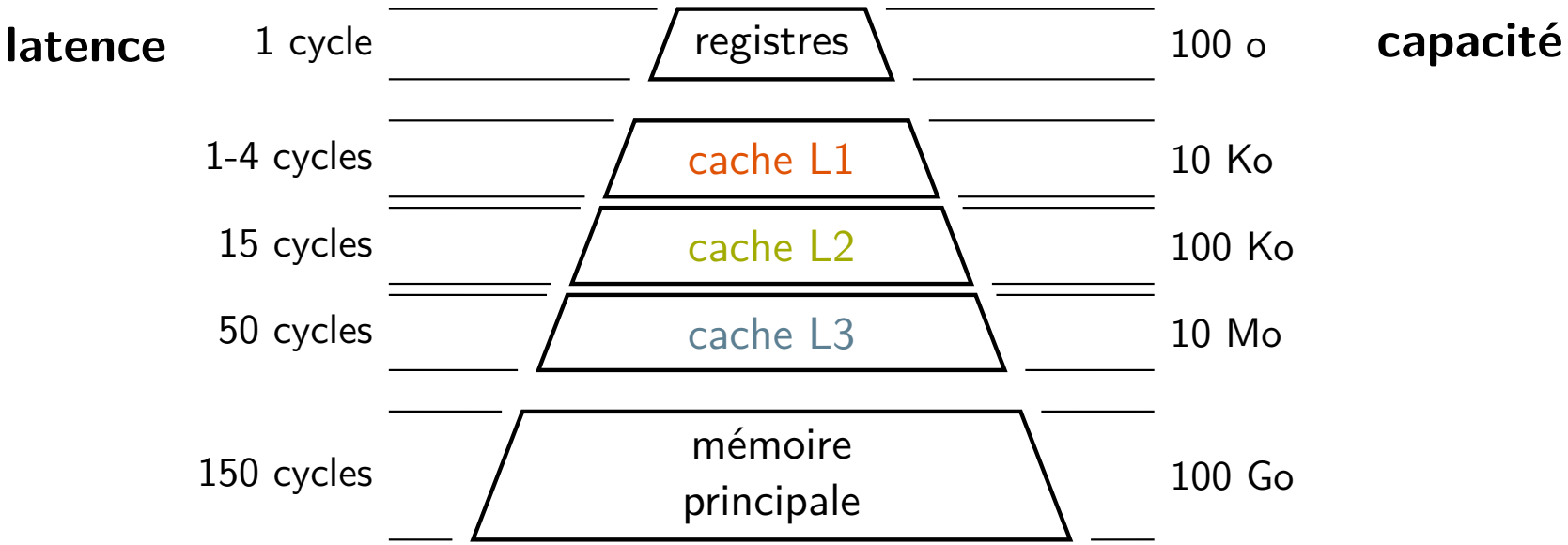
Topologie de caches : retour sur la hiérarchie mémoire



- Un cache de 10 Ko est très petit devant la mémoire (100 Go)
  - Beaucoup de cache miss, même avec une bonne localité
- Augmenter la taille du cache $\rightarrow$ impossible sans augmenter la latence
- Ajouter un second cache entre le premier cache et la mémoire

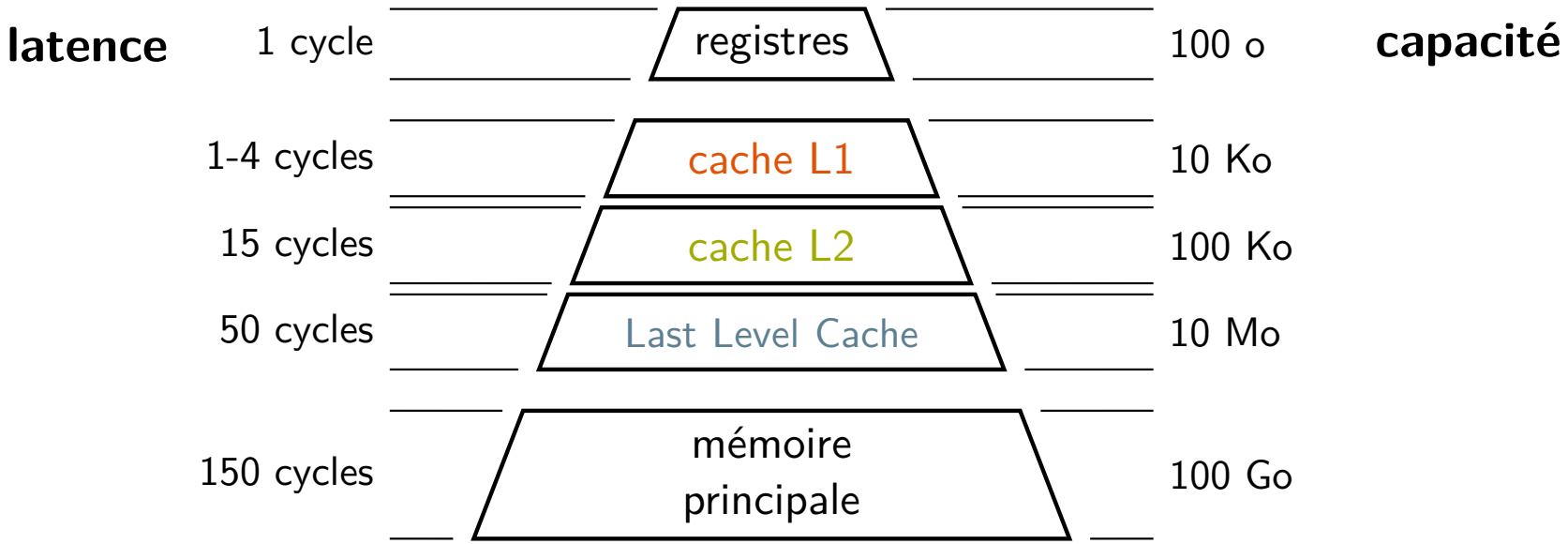
- Ajouter un troisième cache entre le second cache et la mémoire

Topologie de caches : retour sur la hiérarchie mémoire



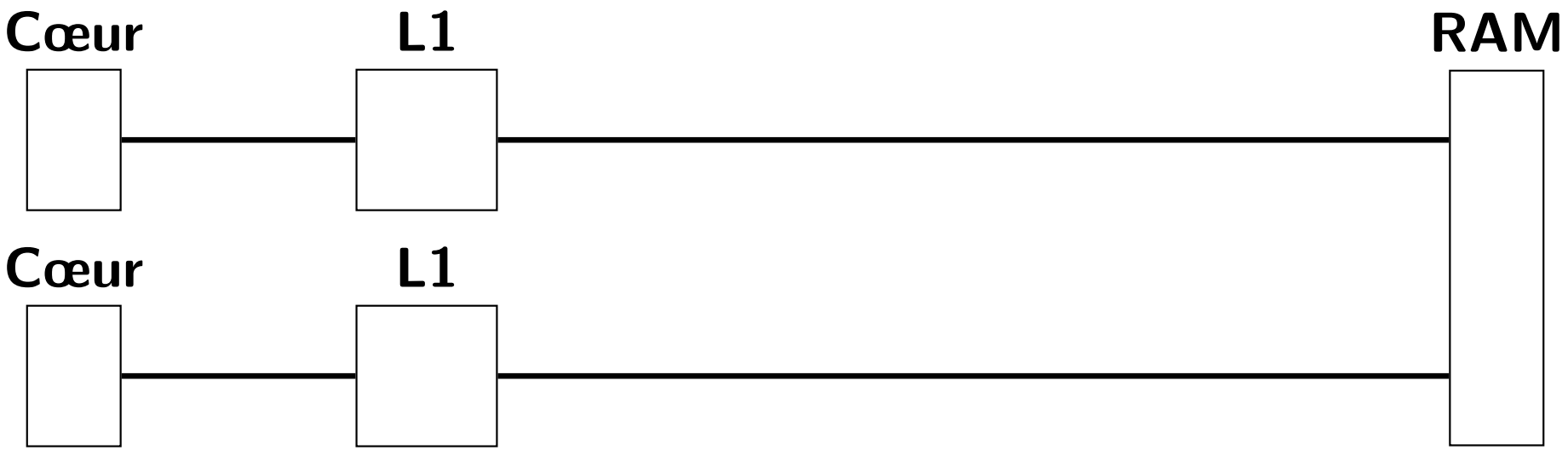
- Un cache de 10 Ko est très petit devant la mémoire (100 Go)
  - Beaucoup de cache miss, même avec une bonne localité
- Augmenter la taille du cache $\rightarrow$ impossible sans augmenter la latence
- Ajouter un second cache entre le premier cache et la mémoire
  - Appelé cache L2
- Ajouter un troisième cache entre le second cache et la mémoire
  - Appelé cache L3

Topologie de caches : retour sur la hiérarchie mémoire



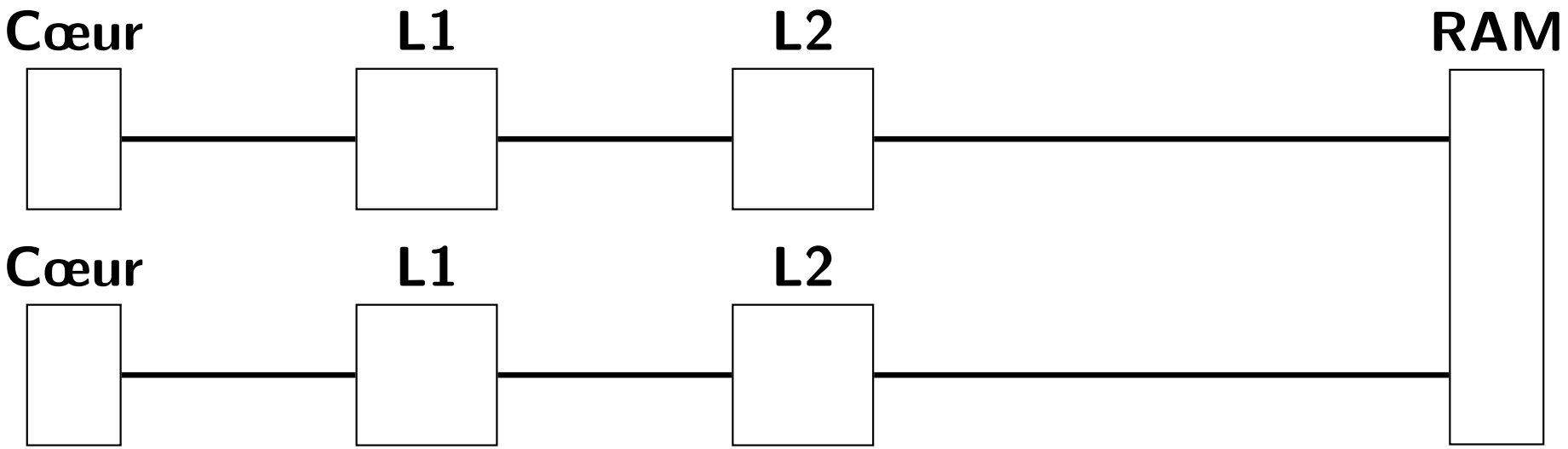
- Un cache de 10 Ko est très petit devant la mémoire (100 Go)
  - Beaucoup de cache miss, même avec une bonne localité
- Augmenter la taille du cache $\rightarrow$ impossible sans augmenter la latence
- Ajouter un second cache entre le premier cache et la mémoire
  - Appelé cache L2
- Ajouter un troisième cache entre le second cache et la mémoire
  - Appelé cache L3
  - Appelé Last Level Cache (quand c'est le cas)

Topologie de caches et chaînage de requêtes



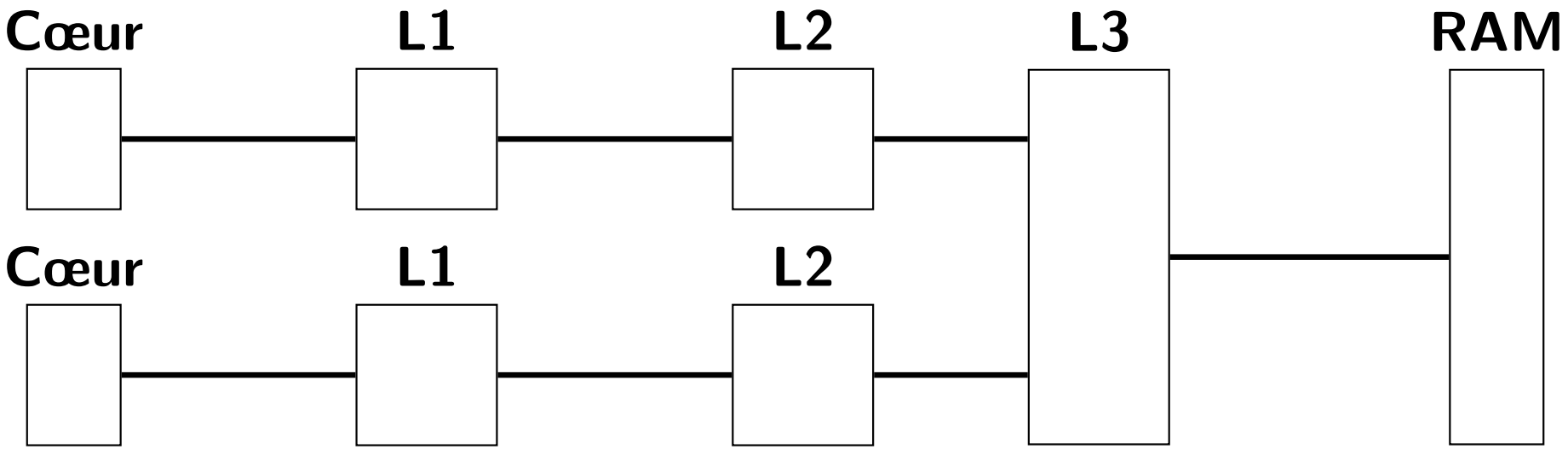
Topologie de caches et chaînage de requêtes

- Un cache L2 privé par cœur $\rightarrow$ meilleure localité



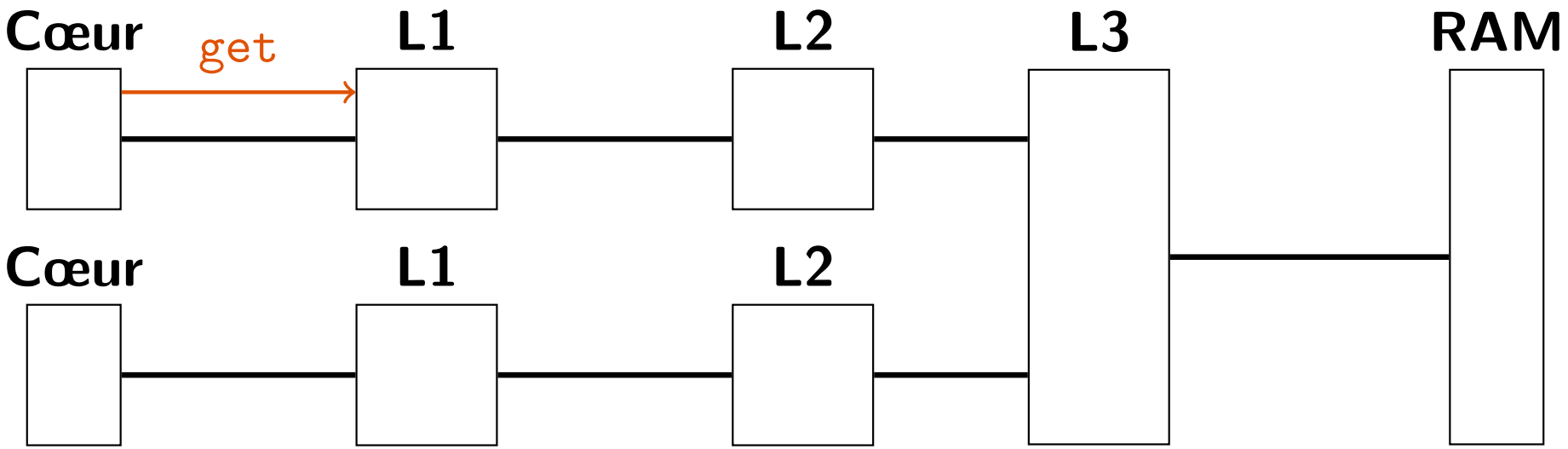
Topologie de caches et chaînage de requêtes

- Un cache L2 privé par cœur $\rightarrow$ meilleure localité
- Un unique cache L3 pour tous les cœurs $\rightarrow$ moins de communications



Topologie de caches et chaînage de requêtes

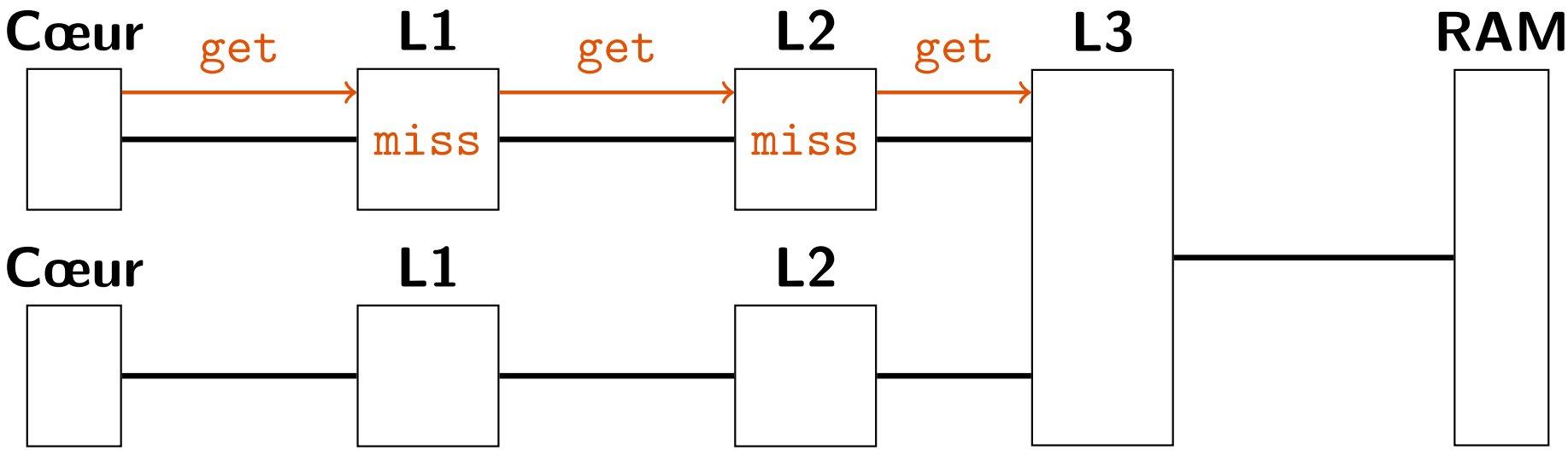
- Un cache L2 privé par cœur → meilleure localité
- Un unique cache L3 pour tous les cœurs → moins de communications



- Le cœur fait systématiquement une requête au cache L1

Topologie de caches et chaînage de requêtes

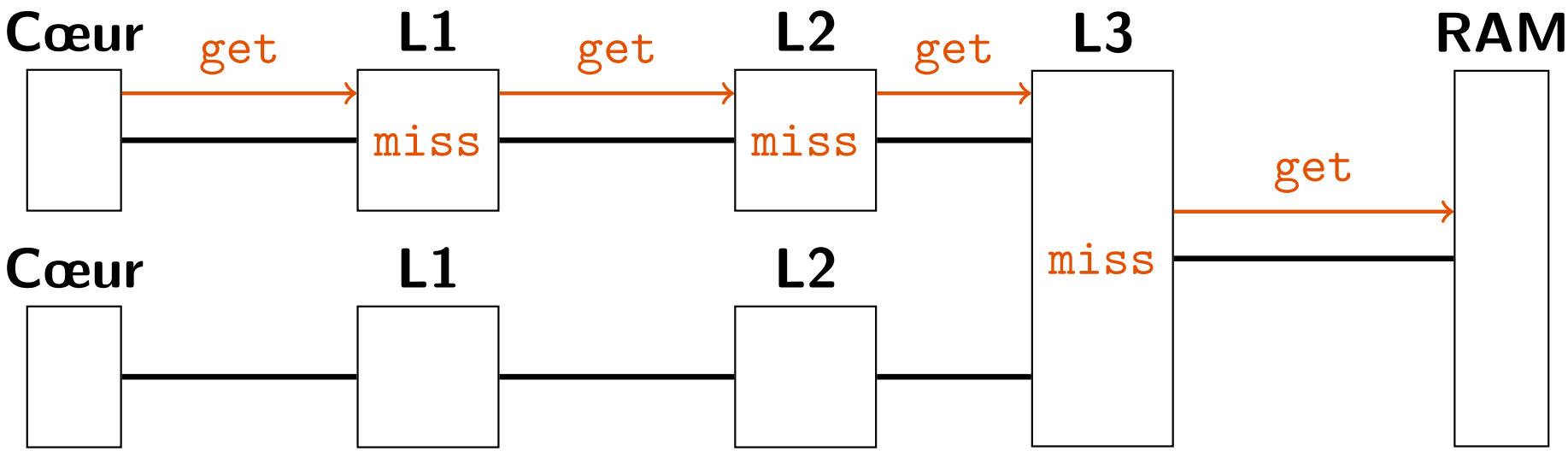
- Un cache L2 privé par cœur → meilleure localité
- Un unique cache L3 pour tous les cœurs → moins de communications



- Le cœur fait systématiquement une requête au cache L1
- Si un niveau de cache miss, il transmet la requête au niveau suivant

Topologie de caches et chaînage de requêtes

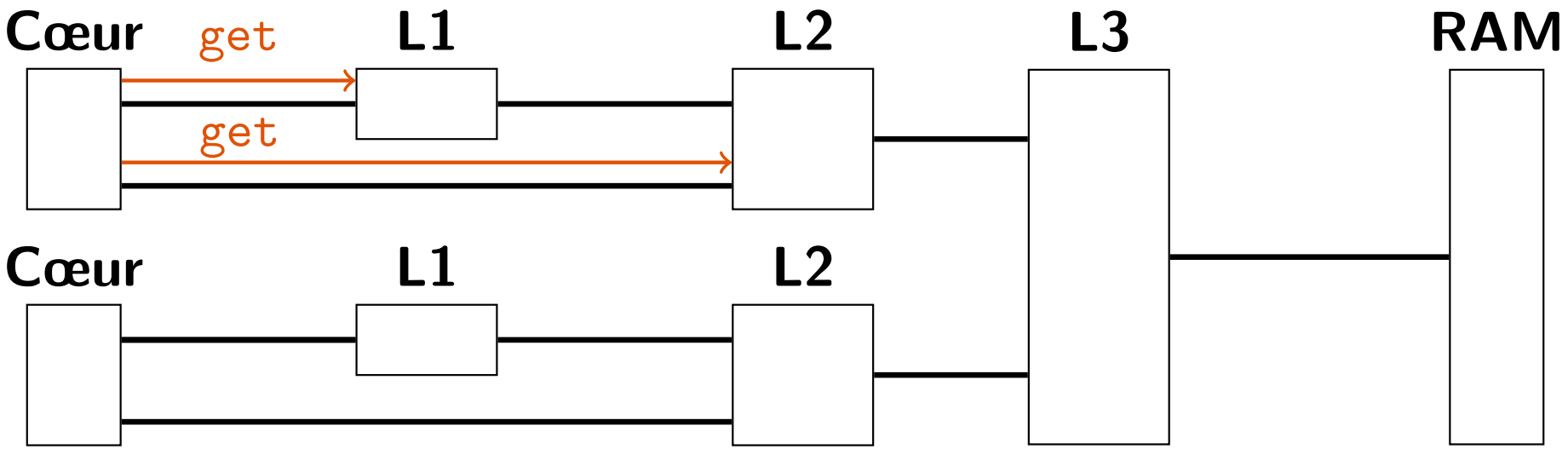
- Un cache L2 privé par cœur → meilleure localité
- Un unique cache L3 pour tous les cœurs → moins de communications



- Le cœur fait systématiquement une requête au cache L1
- Si un niveau de cache miss, il transmet la requête au niveau suivant
- Si le LLC miss, il transmet à la mémoire principale

Topologie de caches et chaînage de requêtes

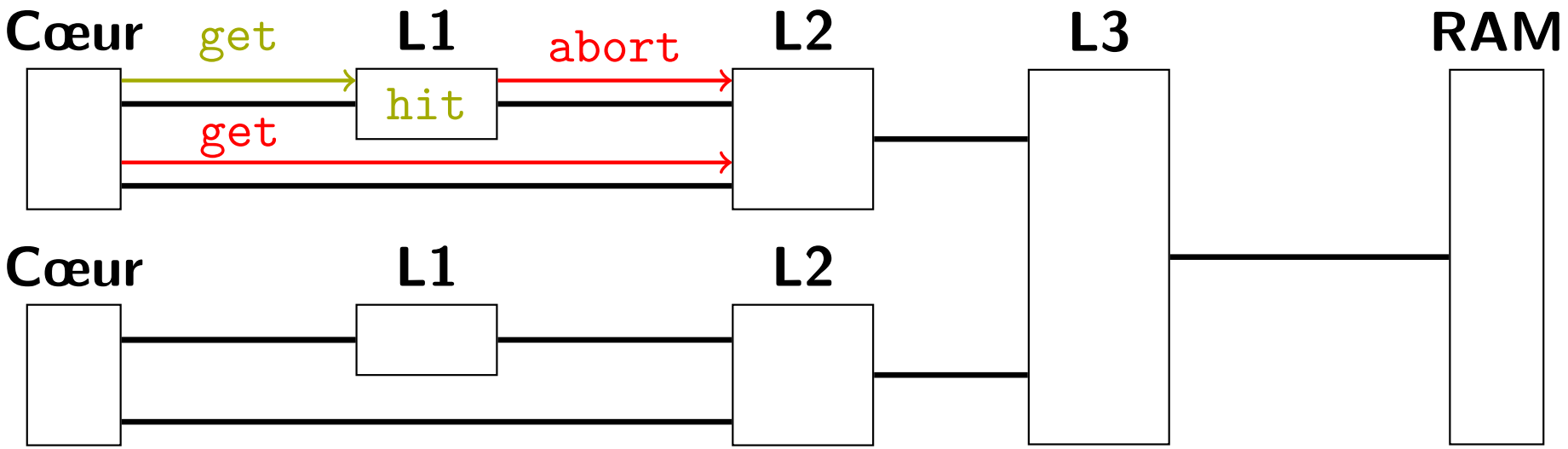
- Un cache L2 privé par cœur $\rightarrow$ meilleure localité
- Un unique cache L3 pour tous les cœurs $\rightarrow$ moins de communications



- Le cœur fait systématiquement une requête au cache L1
- Si un niveau de cache miss, il transmet la requête au niveau suivant
- Si le LLC miss, il transmet à la mémoire principale
- Plusieurs requêtes peuvent être émises en parallèle

Topologie de caches et chaînage de requêtes

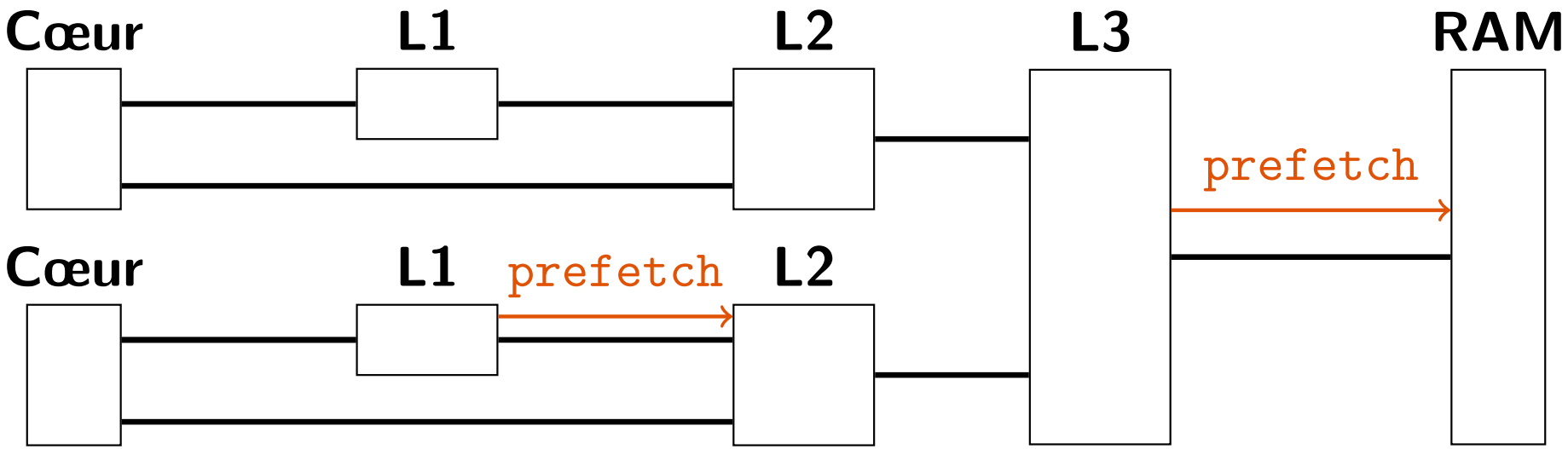
- Un cache L2 privé par cœur → meilleure localité
- Un unique cache L3 pour tous les cœurs → moins de communications



- Le cœur fait systématiquement une requête au cache L1
- Si un niveau de cache miss, il transmet la requête au niveau suivant
- Si le LLC miss, il transmet à la mémoire principale
- Plusieurs requêtes peuvent être émises en parallèle
  - Si une des requêtes est servie, les autres requêtes sont annulées

Topologie de caches et chaînage de requêtes

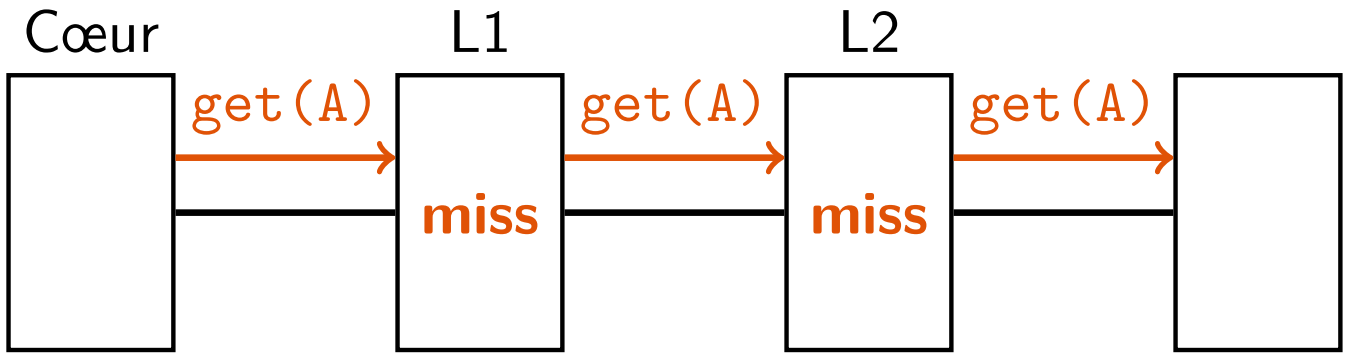
- Un cache L2 privé par cœur → meilleure localité
- Un unique cache L3 pour tous les cœurs → moins de communications



- Le cœur fait systématiquement une requête au cache L1
- Si un niveau de cache miss, il transmet la requête au niveau suivant
- Si le LLC miss, il transmet à la mémoire principale
- Plusieurs requêtes peuvent être émises en parallèle
  - Si une des requêtes est servie, les autres requêtes sont annulées
- Plusieurs niveaux de prefetching

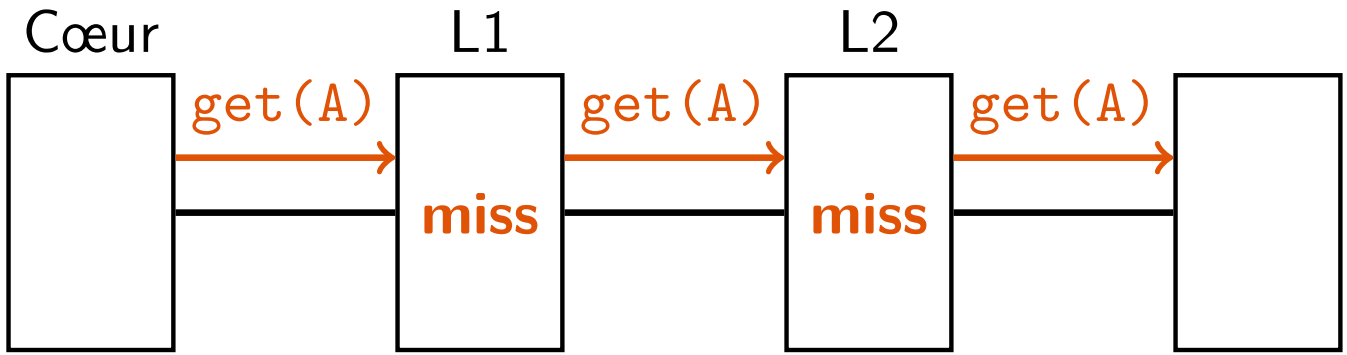
Stratégies d'inclusion

- En cas de *cache miss* sur plusieurs niveaux de cache → où est stockée la nouvelle ligne ? Tous les niveaux ? L1 ?



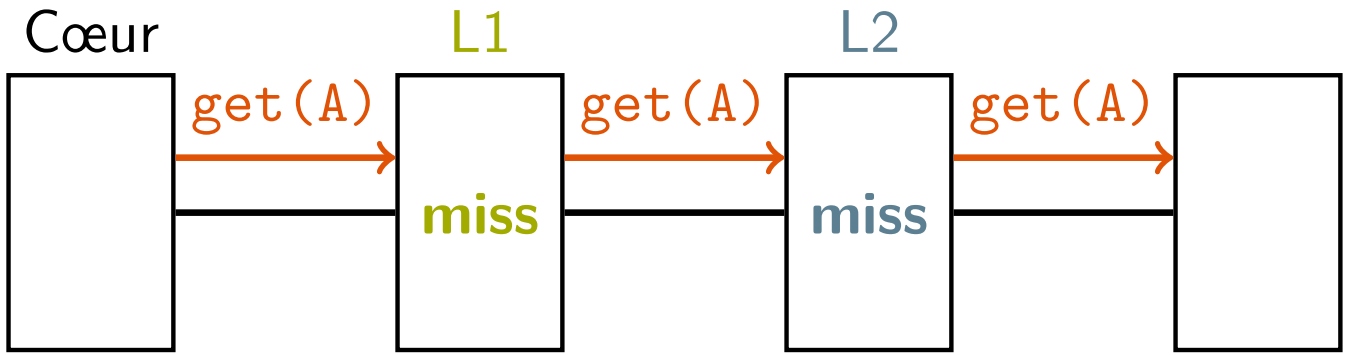
Stratégies d'inclusion

- Toute ligne transmise au cœur est stockée au moins en cache L1



Stratégies d'inclusion : cache inclusif

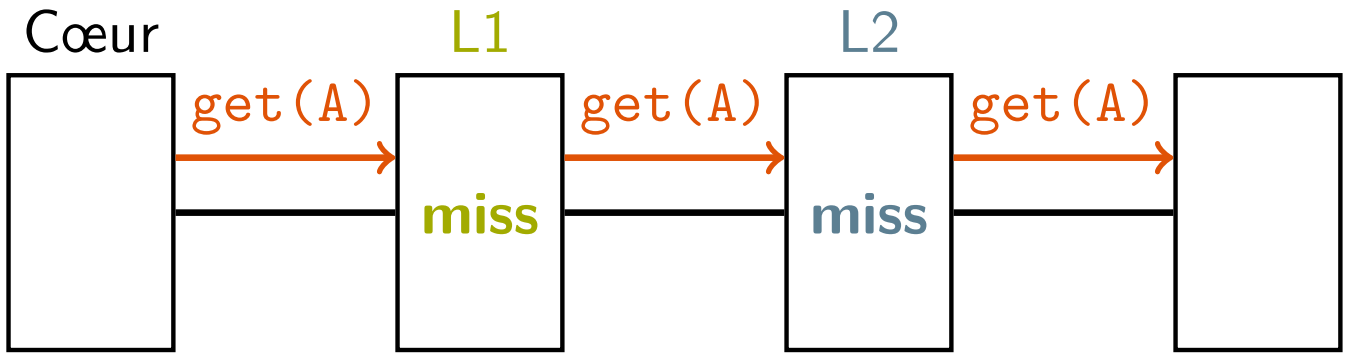
- Toute ligne transmise au cœur est stockée au moins en cache L1



- Le cache LX est inclusif du cache LY ($Y < X$)
  - Si une ligne A est présente dans LY
  - Alors la ligne A est présente dans LX

Stratégies d'inclusion : cache inclusif

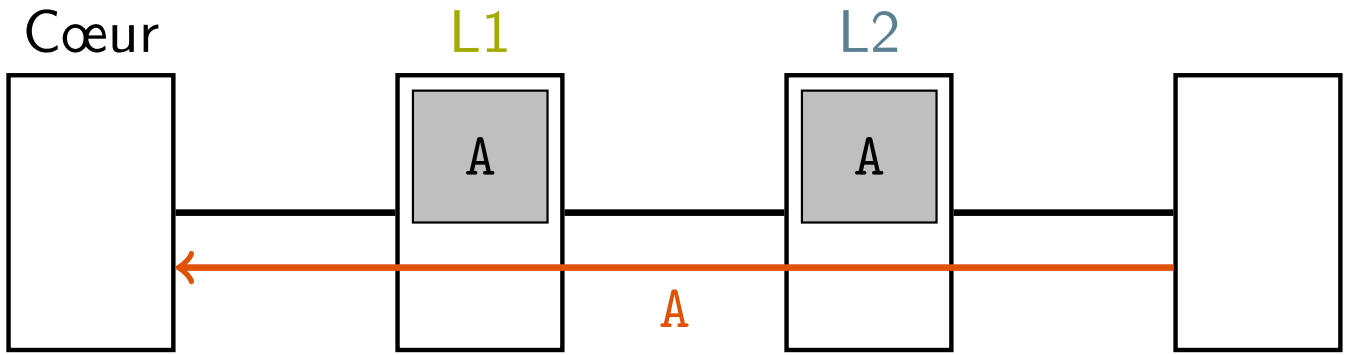
- Toute ligne transmise au cœur est stockée au moins en cache L1



- Le cache LX est inclusif du cache LY ($Y < X$)
  - Si une ligne A est présente dans LY
  - Alors la ligne A est présente dans LX
- Quand miss LY et miss LX

Stratégies d'inclusion : cache inclusif

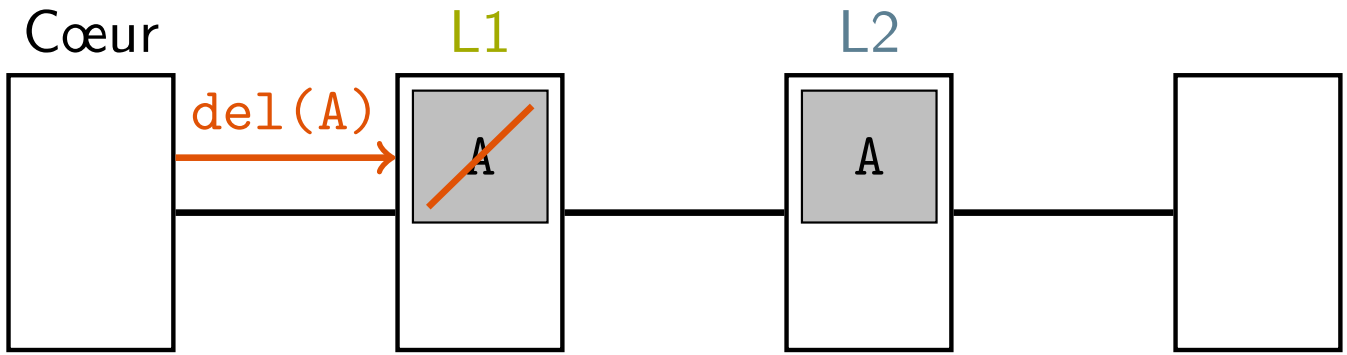
- Toute ligne transmise au cœur est stockée au moins en cache L1



- Le cache LX est inclusif du cache LY ($Y < X$)
  - Si une ligne A est présente dans LY
  - Alors la ligne A est présente dans LX
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LX et LY

Stratégies d'inclusion : cache inclusif

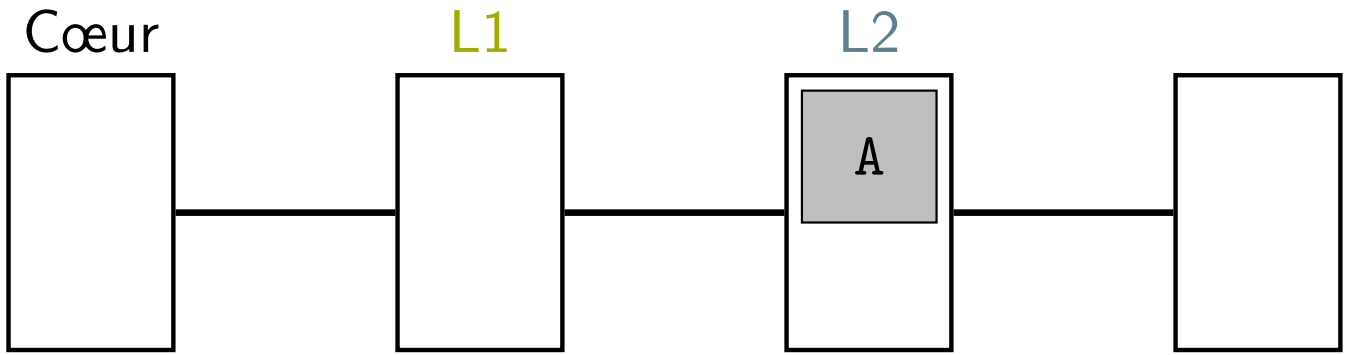
- Toute ligne transmise au cœur est stockée au moins en cache L1



- Le cache LX est inclusif du cache LY ($Y < X$)
  - Si une ligne A est présente dans LY
  - Alors la ligne A est présente dans LX
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LX et LY
- Quand la ligne A est évincée de LY

Stratégies d'inclusion : cache inclusif

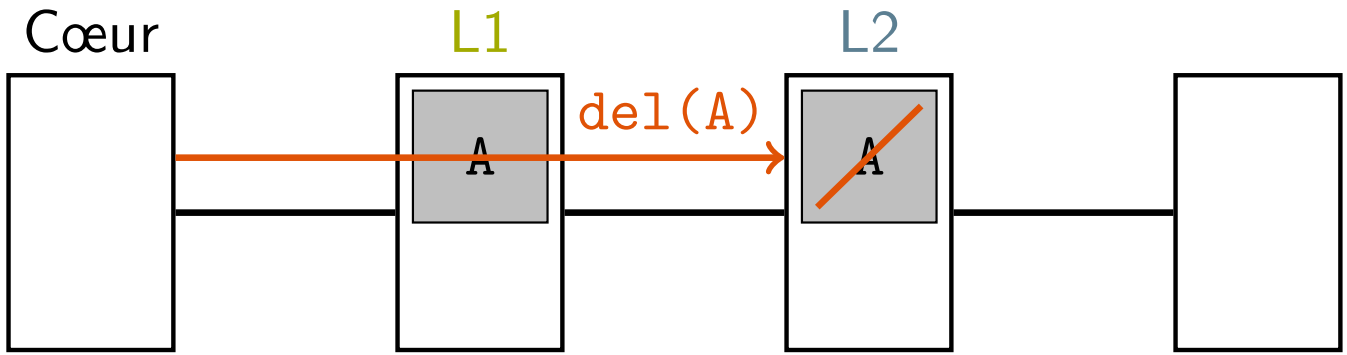
- Toute ligne transmise au cœur est stockée au moins en cache L1



- Le cache LX est inclusif du cache LY ($Y < X$)
  - Si une ligne A est présente dans LY
  - Alors la ligne A est présente dans LX
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LX et LY
- Quand la ligne A est évincée de LY $\rightarrow$ pas d'effet sur LX

Stratégies d'inclusion : cache inclusif

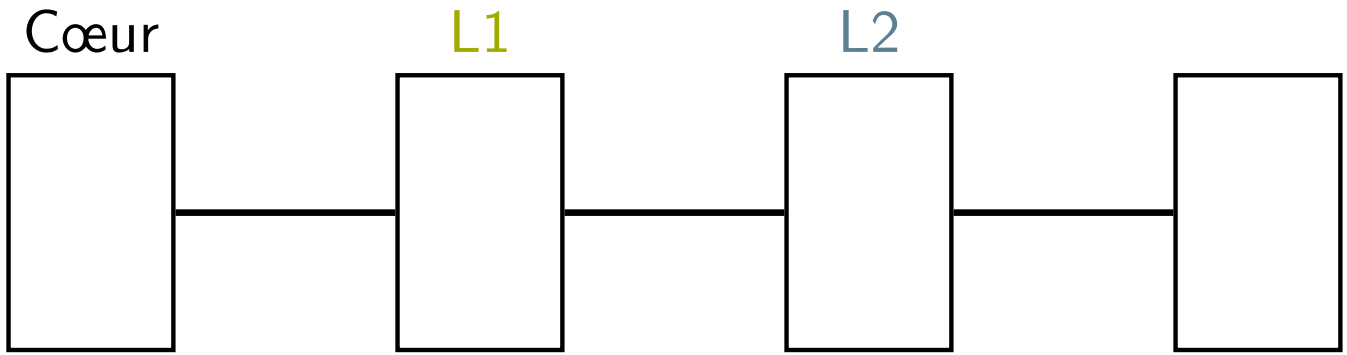
- Toute ligne transmise au cœur est stockée au moins en cache L1



- Le cache LX est inclusif du cache LY ($Y < X$)
  - Si une ligne A est présente dans LY
  - Alors la ligne A est présente dans LX
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LX et LY
- Quand la ligne A est évincée de LY $\rightarrow$ pas d'effet sur LX
- Quand la ligne A est évincée de LX

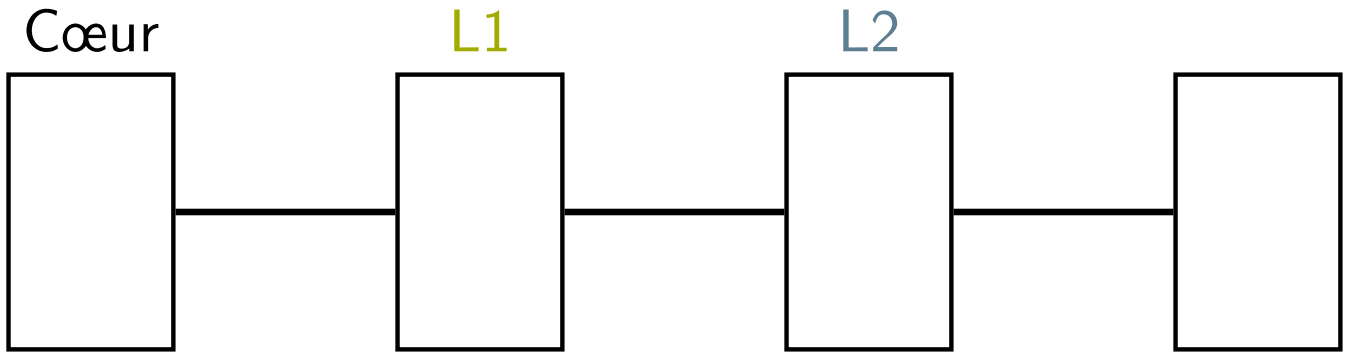
Stratégies d'inclusion : cache inclusif

- Toute ligne transmise au cœur est stockée au moins en cache L1



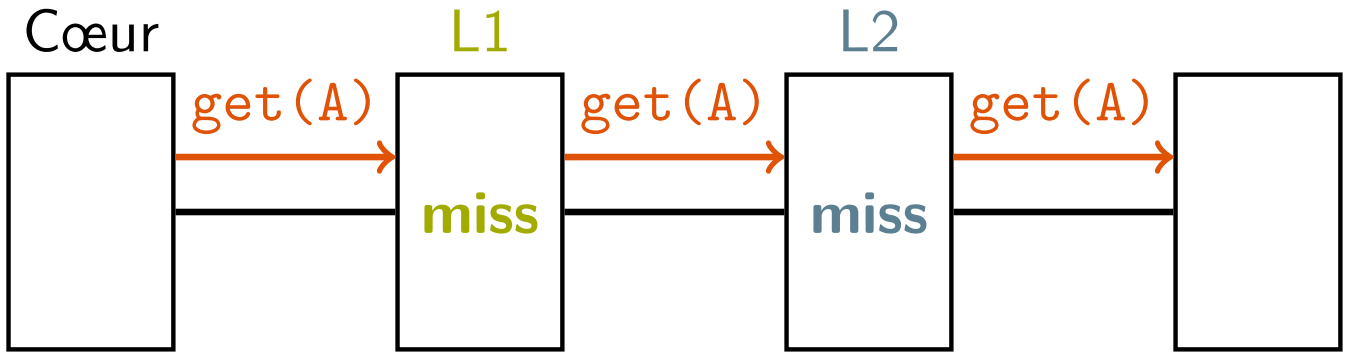
- Le cache LX est inclusif du cache LY ($Y < X$)
  - Si une ligne A est présente dans LY
  - Alors la ligne A est présente dans LX
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LX et LY
- Quand la ligne A est évincée de LY $\rightarrow$ pas d'effet sur LX
- Quand la ligne A est évincée de LX $\rightarrow$ eviction de LY

Stratégies d'inclusion : cache exclusif



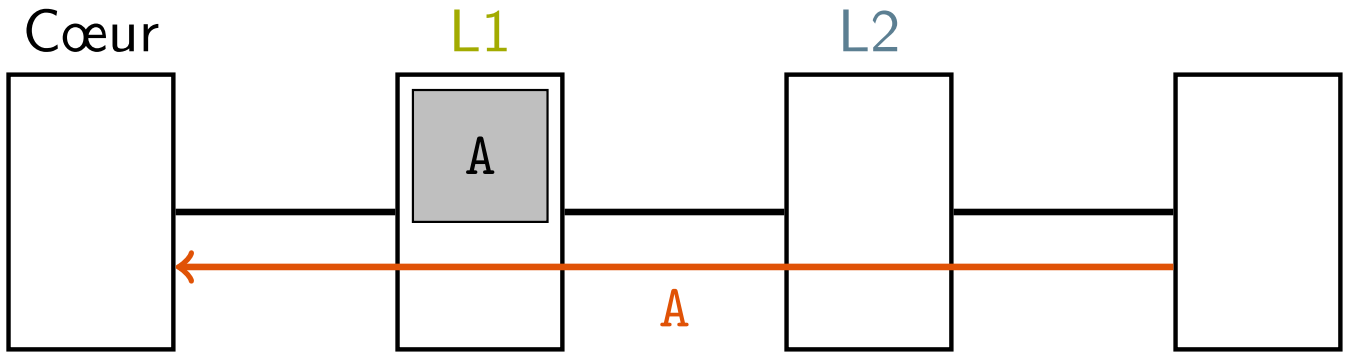
- Les caches LX et LY sont exclusifs
  - Une ligne A est présente dans au plus un des caches
  - Quand $Y = X - 1$, on dit que LX est le *victim cache* de LY

Stratégies d'inclusion : cache exclusif



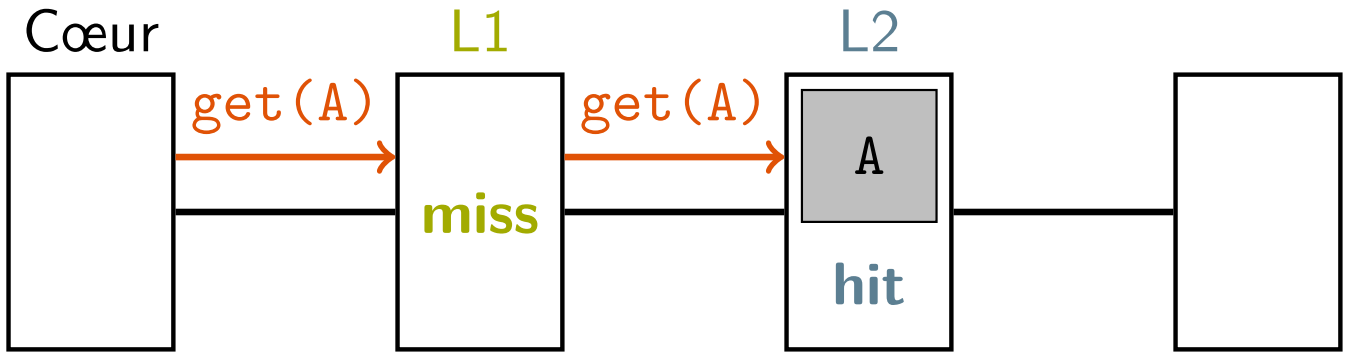
- Les caches LX et LY sont exclusifs
  - Une ligne A est présente dans au plus un des caches
  - Quand $Y = X - 1$, on dit que LX est le *victim cache* de LY
- Quand miss LY et miss LX

Stratégies d'inclusion : cache exclusif



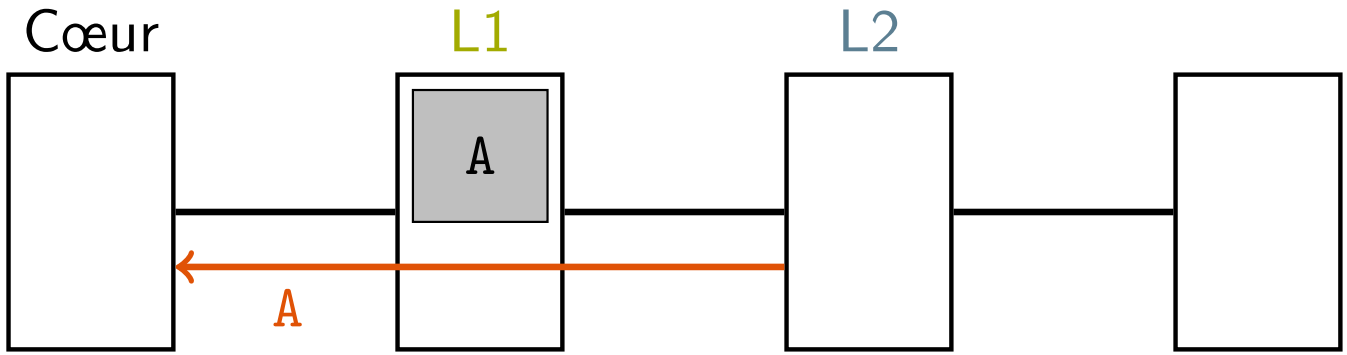
- Les caches LX et LY sont exclusifs
  - Une ligne A est présente dans au plus un des caches
  - Quand $Y = X - 1$, on dit que LX est le *victim cache* de LY
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LY

Stratégies d'inclusion : cache exclusif



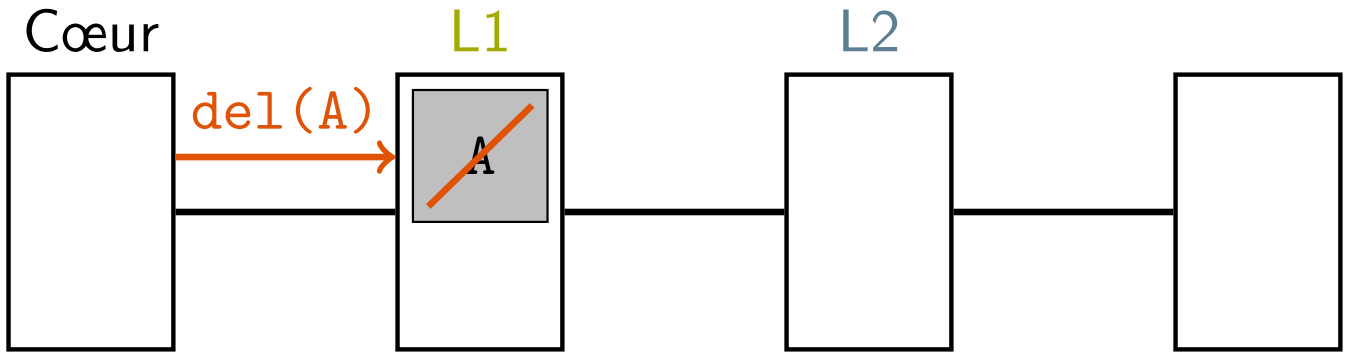
- Les caches LX et LY sont exclusifs
  - Une ligne A est présente dans au plus un des caches
  - Quand $Y = X - 1$, on dit que LX est le *victim cache* de LY
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LY
- Quand miss LY et hit LX

Stratégies d'inclusion : cache exclusif



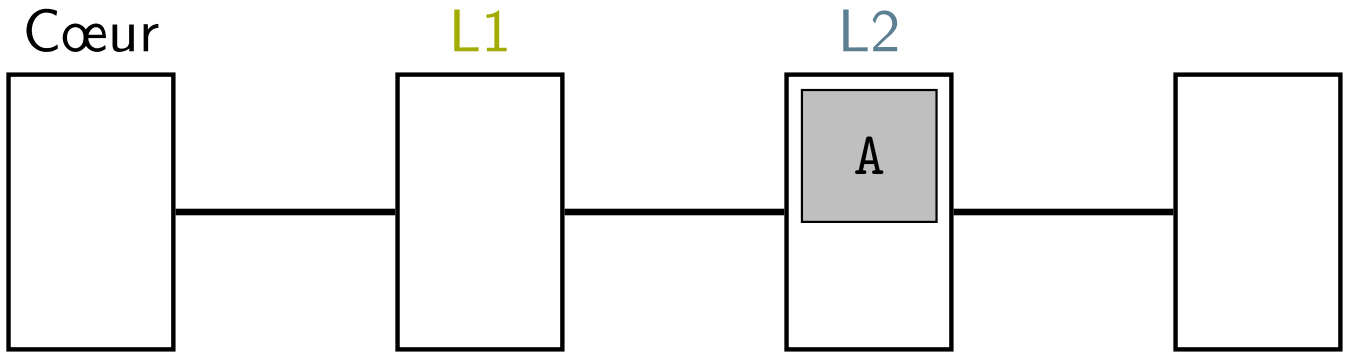
- Les caches LX et LY sont exclusifs
  - Une ligne A est présente dans au plus un des caches
  - Quand $Y = X - 1$, on dit que LX est le *victim cache* de LY
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LY
- Quand miss LY et hit LX $\rightarrow$ la ligne est évincée de LX

Stratégies d'inclusion : cache exclusif



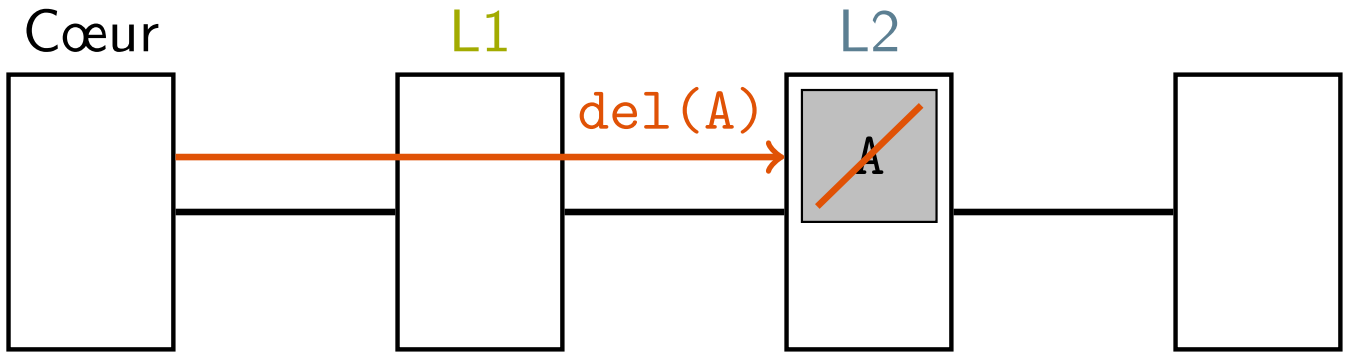
- Les caches LX et LY sont exclusifs
  - Une ligne A est présente dans au plus un des caches
  - Quand $Y = X - 1$, on dit que LX est le *victim cache* de LY
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LY
- Quand miss LY et hit LX $\rightarrow$ la ligne est évincée de LX
- Quand la ligne A est évincée de LY

Stratégies d'inclusion : cache exclusif



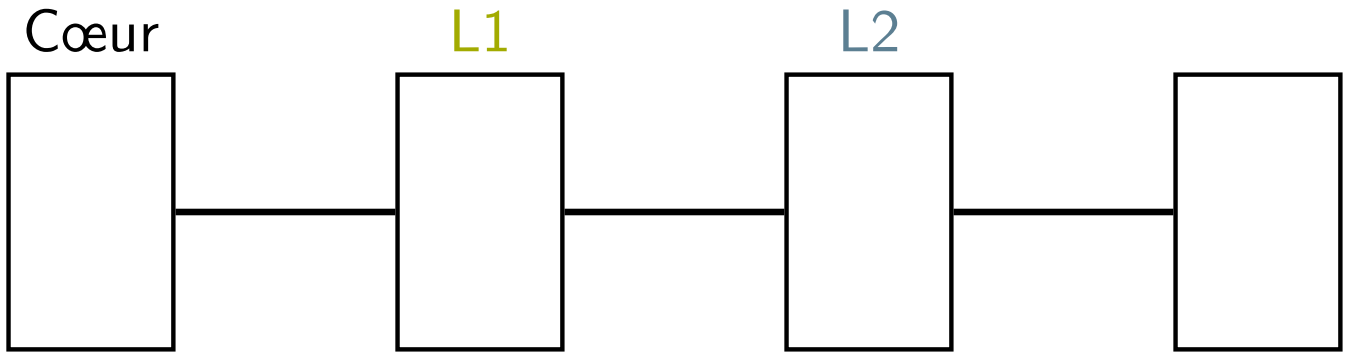
- Les caches LX et LY sont exclusifs
  - Une ligne A est présente dans au plus un des caches
  - Quand $Y = X - 1$, on dit que LX est le *victim cache* de LY
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LY
- Quand miss LY et hit LX $\rightarrow$ la ligne est évincée de LX
- Quand la ligne A est évincée de LY $\rightarrow$ elle va dans LX

Stratégies d'inclusion : cache exclusif



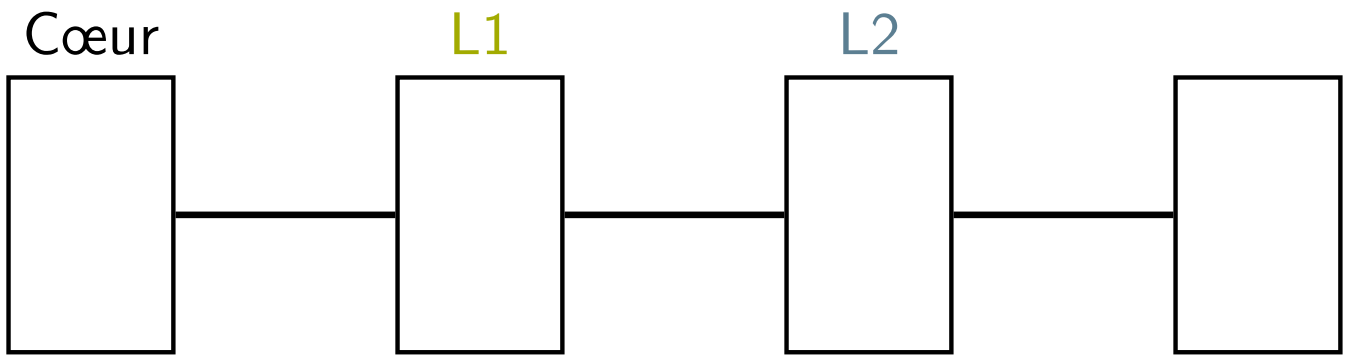
- Les caches LX et LY sont exclusifs
  - Une ligne A est présente dans au plus un des caches
  - Quand $Y = X - 1$, on dit que LX est le *victim cache* de LY
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LY
- Quand miss LY et hit LX $\rightarrow$ la ligne est évincée de LX
- Quand la ligne A est évincée de LY $\rightarrow$ elle va dans LX
- Quand la ligne A est évincée de LX

Stratégies d'inclusion : cache exclusif



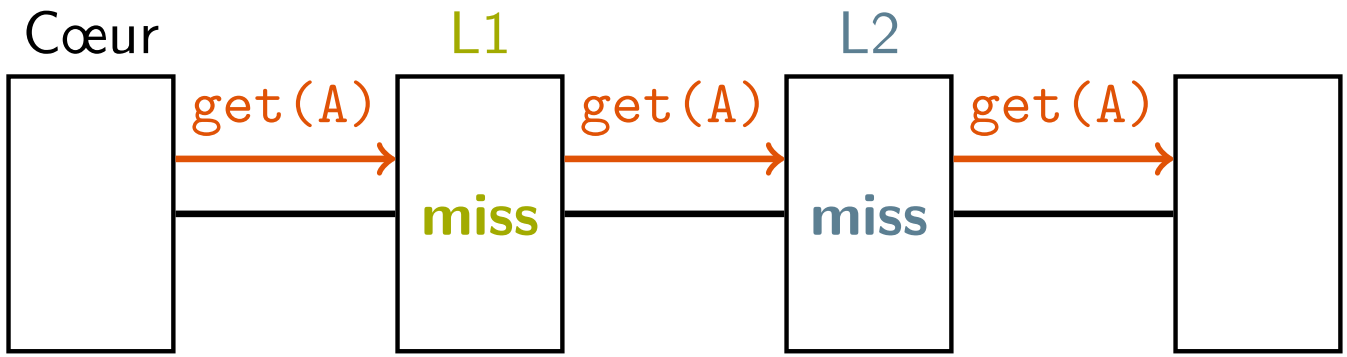
- Les caches LX et LY sont exclusifs
  - Une ligne A est présente dans au plus un des caches
  - Quand $Y = X - 1$, on dit que LX est le *victim cache* de LY
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LY
- Quand miss LY et hit LX $\rightarrow$ la ligne est évincée de LX
- Quand la ligne A est évincée de LY $\rightarrow$ elle va dans LX
- Quand la ligne A est évincée de LX $\rightarrow$ pas d'effet sur LY

Stratégies d'inclusion : cache NINE



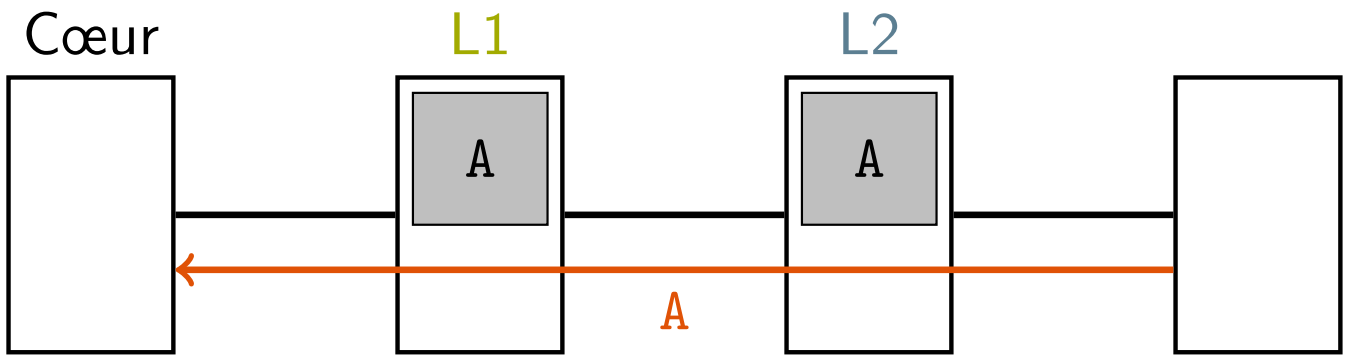
- Les caches LX et LY sont Ni Inclusif Ni Exclusif
  - Pas d'éviction forcée

Stratégies d'inclusion : cache NINE



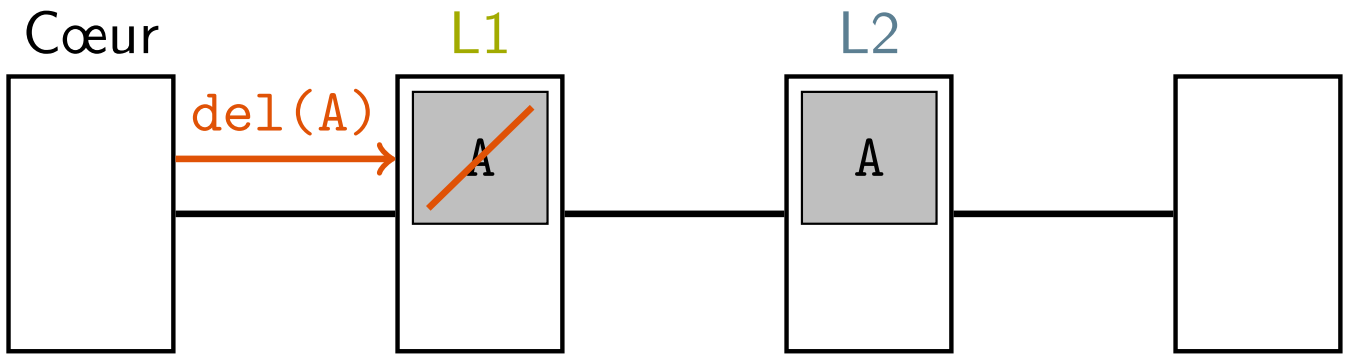
- Les caches LX et LY sont Ni Inclusif Ni Exclusif
  - Pas d'éviction forcée
- Quand miss LY et miss LX

Stratégies d'inclusion : cache NINE



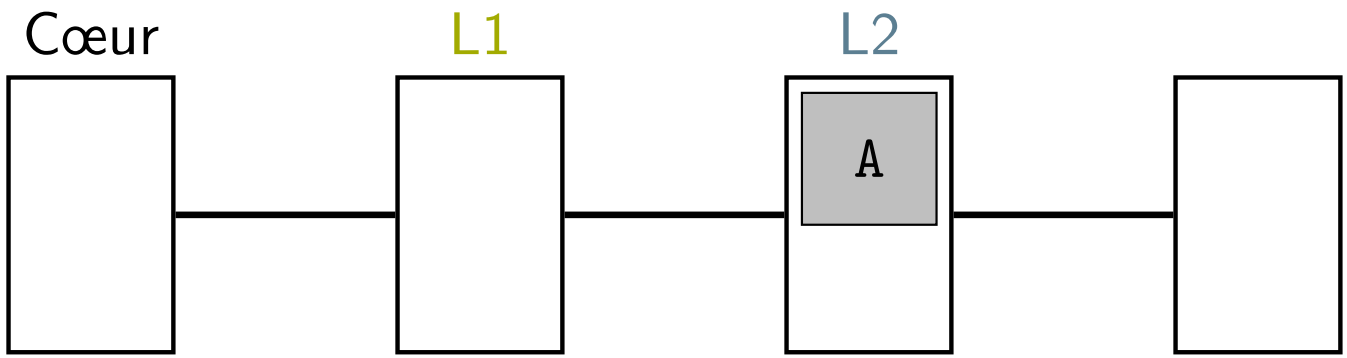
- Les caches LX et LY sont Ni Inclusif Ni Exclusif
  - Pas d'éviction forcée
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LX et LY

Stratégies d'inclusion : cache NINE



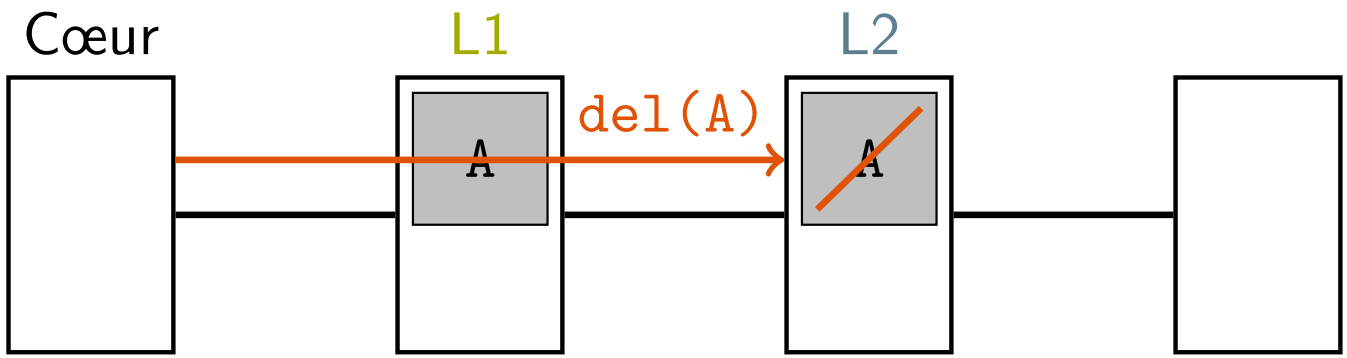
- Les caches LX et LY sont Ni Inclusif Ni Exclusif
  - Pas d'éviction forcée
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LX et LY
- Quand la ligne A est évincée de LY

Stratégies d'inclusion : cache NINE



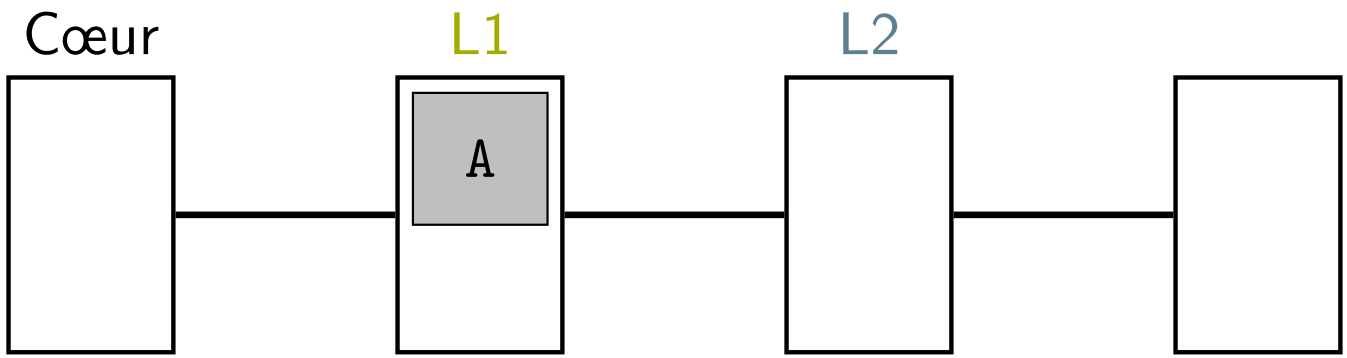
- Les caches LX et LY sont Ni Inclusif Ni Exclusif
  - Pas d'éviction forcée
- Quand miss LY et miss LX $\rightarrow$ la nouvelle ligne va dans LX et LY
- Quand la ligne A est évincée de LY $\rightarrow$ pas d'effet sur LX

Stratégies d'inclusion : cache NINE



- Les caches LX et LY sont Ni Inclusif Ni Exclusif
  - Pas d'éviction forcée
- Quand miss LY et miss LX → la nouvelle ligne va dans LX et LY
- Quand la ligne A est évincée de LY → pas d'effet sur LX
- Quand la ligne A est évincée de LX

Stratégies d'inclusion : cache NINE



- Les caches LX et LY sont Ni Inclusif Ni Exclusif
  - Pas d'éviction forcée
- Quand miss LY et miss LX → la nouvelle ligne va dans LX et LY
- Quand la ligne A est évincée de LY → pas d'effet sur LX
- Quand la ligne A est évincée de LX → pas d'effet sur LY

Stratégies d'inclusion : résumé

- Les différents niveaux de cache peuvent se combiner différemment
  - Combinaison via des stratégies d'inclusion
  - Comment une ligne A s'échange entre les niveaux LX et LY ($Y < X$)

- Stratégie inclusive : si A est dans LY, alors A est dans LX
  - Gain de temps $\rightarrow$ si miss LX alors pas de requête à LY (cache partagé)
  - Perte de place $\rightarrow$ même ligne stockée dans deux caches en même temps

- Stratégie exclusive : si A est dans un cache, alors A n'est pas dans l'autre
  - Gain de place $\rightarrow$ capacité totale $=$ somme de LX et LY
  - Consommation de bande passante $\rightarrow$ transferts entre LX et LY

- Stratégie NINE : pas de contrainte
  - Gain de place $\rightarrow$ capacité totale $\simeq$ somme de LX et LY
  - Pas de gain ni de perte de temps

Contrôle logiciel du cache : prefetching

- Le prefetching est fait automatiquement par le contrôleur de cache
  - Tente de prédire le prochain accès en se basant sur les accès passés
  - Mauvaise prédiction $\rightarrow$ consommation de ressources inutile

- Le logiciel peut demander un prefetch $\rightarrow$ instructions dédiées
  - Le programmeur connaît le programme exécuté $\rightarrow$ meilleure prédiction
  - Requête traitée en parallèle des instructions suivantes $\rightarrow$ pas de stall

Contrôle logiciel du cache : prefetching

- Le prefetching est fait automatiquement par le contrôleur de cache
  - Tente de prédire le prochain accès en se basant sur les accès passés
  - Mauvaise prédiction $\rightarrow$ consommation de ressources inutile

- Le logiciel peut demander un prefetch $\rightarrow$ instructions dédiées
  - Le programmeur connaît le programme exécuté $\rightarrow$ meilleure prédiction
  - Requête traitée en parallèle des instructions suivantes $\rightarrow$ pas de stall

- `prefetch` $\Rightarrow$ prefetch simple en lecture $\simeq$ lecture mémoire standard

Contrôle logiciel du cache : prefetching

- Le prefetching est fait automatiquement par le contrôleur de cache
  - Tente de prédire le prochain accès en se basant sur les accès passés
  - Mauvaise prédiction $\rightarrow$ consommation de ressources inutile

- Le logiciel peut demander un prefetch $\rightarrow$ instructions dédiées
  - Le programmeur connaît le programme exécuté $\rightarrow$ meilleure prédiction
  - Requête traitée en parallèle des instructions suivantes $\rightarrow$ pas de stall

- `prefetch` $\Rightarrow$ prefetch simple en lecture $\simeq$ lecture mémoire standard
- `prefecthw` $\Rightarrow$ prefetch simple en écriture $\rightarrow$ état Modified

Contrôle logiciel du cache : prefetching

- Le prefetching est fait automatiquement par le contrôleur de cache
  - Tente de prédire le prochain accès en se basant sur les accès passés
  - Mauvaise prédiction $\rightarrow$ consommation de ressources inutile

- Le logiciel peut demander un prefetch $\rightarrow$ instructions dédiées
  - Le programmeur connaît le programme exécuté $\rightarrow$ meilleure prédiction
  - Requête traitée en parallèle des instructions suivantes $\rightarrow$ pas de stall

- `prefetch` $\Rightarrow$ prefetch simple en lecture $\simeq$ lecture mémoire standard
- `prefecthw` $\Rightarrow$ prefetch simple en écriture $\rightarrow$ état Modified
- `prefetchnta` $\Rightarrow$ prefetch en minimisant la pollution de cache
  - Charger en cache L1 uniquement
  - Interdire l'éviction en niveau de cache supérieur

Contrôle logiciel du cache : prefetching

- Le prefetching est fait automatiquement par le contrôleur de cache
  - Tente de prédire le prochain accès en se basant sur les accès passés
  - Mauvaise prédiction $\rightarrow$ consommation de ressources inutile

- Le logiciel peut demander un prefetch $\rightarrow$ instructions dédiées
  - Le programmeur connaît le programme exécuté $\rightarrow$ meilleure prédiction
  - Requête traitée en parallèle des instructions suivantes $\rightarrow$ pas de stall

- `prefetch` $\Rightarrow$ prefetch simple en lecture $\simeq$ lecture mémoire standard
- `prefecthw` $\Rightarrow$ prefetch simple en écriture $\rightarrow$ état Modified
- `prefetchnta` $\Rightarrow$ prefetch en minimisant la pollution de cache
  - Charger en cache L1 uniquement
  - Interdire l'éviction en niveau de cache supérieur

- Ne peut pas déclencher de faute de page

Contrôle logiciel du cache : prefetching

- Le prefetching est fait automatiquement par le contrôleur de cache
  - Tente de prédire le prochain accès en se basant sur les accès passés
  - Mauvaise prédiction → consommation de ressources inutile
- L
- p
- p
- `prefetchnta` ⇒ prefetch en minimisant la pollution de cache
  - Charger en cache L1 uniquement
  - Interdire l'éviction en niveau de cache supérieur
- Ne peut pas déclencher de faute de page

Premature optimization is the root of all evil (or at least most of it) in programming

Donald Knuth

Contrôle logiciel du cache : prefetching

- Le prefetching est fait automatiquement par le contrôleur de cache
  - Tente de prédire le prochain accès en se basant sur les accès passés
  - Mauvaise prédiction → consommation de ressources inutile

- L
- p
- p
- `prefetchnta` ⇒ prefetch en minimisant la pollution de cache
  - Charger en cache L1 uniquement
  - Interdire l'éviction en niveau de cache supérieur

- Ne peut pas déclencher de faute de page

Pas d'optimisation sans **profiling**

Contrôle logiciel du cache : éviction

- Les lignes sont automatiquement évincées du cache
  - Évènement non prédictible pour le logiciel
  - Stratégies d'éviction complexes : pas toujours la meilleure ligne choisie

- Le logiciel peut demander une éviction $\rightarrow$ instructions dédiées
  - Contrôle fin des ways d'un cache associatif $\rightarrow$ performance
  - Synchronisation avec les périphériques $\rightarrow$ validité

Contrôle logiciel du cache : éviction

- Les lignes sont automatiquement évincées du cache
  - Évènement non prédictible pour le logiciel
  - Stratégies d'éviction complexes : pas toujours la meilleure ligne choisie

- Le logiciel peut demander une éviction $\rightarrow$ instructions dédiées
  - Contrôle fin des ways d'un cache associatif $\rightarrow$ performance
  - Synchronisation avec les périphériques $\rightarrow$ validité

- `clflush` $\Rightarrow$ évince une ligne donnée de tous les niveaux de cache
  - Si la ligne est *dirty*, elle est propagée en mémoire

Contrôle logiciel du cache : éviction

- Les lignes sont automatiquement évincées du cache
  - Évènement non prédictible pour le logiciel
  - Stratégies d'éviction complexes : pas toujours la meilleure ligne choisie

- Le logiciel peut demander une éviction → instructions dédiées
  - Contrôle fin des ways d'un cache associatif → performance
  - Synchronisation avec les périphériques → validité

- `clflush` ⇒ évince une ligne donnée de tous les niveaux de cache
  - Si la ligne est *dirty*, elle est propagée en mémoire

- `wbinvd` ⇒ invalide tout le cache (instruction privilégiée)
  - Les lignes *dirty* sont propagées en mémoire
  - Exemple d'utilisation : effectuer toutes les modifications en mémoire principale avant de lancer un transfert DMA

Contrôle logiciel du cache : table des pages

- Le bit **P**age **C**ache **D**isable indique si les lignes de la page peuvent être mises en cache
- Le bit **P**age **W**rite **T**hrough indique si les lignes de la page peuvent être *dirty*

- Quand un cœur modifie une ligne marquée write through
  - La ligne est modifiée dans le cache
  - La ligne est modifiée en mémoire principale

Contrôle logiciel du cache : table des pages

- Le bit **P**age **C**ache **D**isable indique si les lignes de la page peuvent être mises en cache
- Le bit **P**age **W**rite **T**hrough indique si les lignes de la page peuvent être *dirty*

- Quand un cœur modifie une ligne marquée write through
  - La ligne est modifiée dans le cache
  - La ligne est modifiée en mémoire principale

- Utile pour les périphériques mappés en mémoire
  - Exemple : la mémoire CGA (vidéo)

Contrôle logiciel du cache : résumé

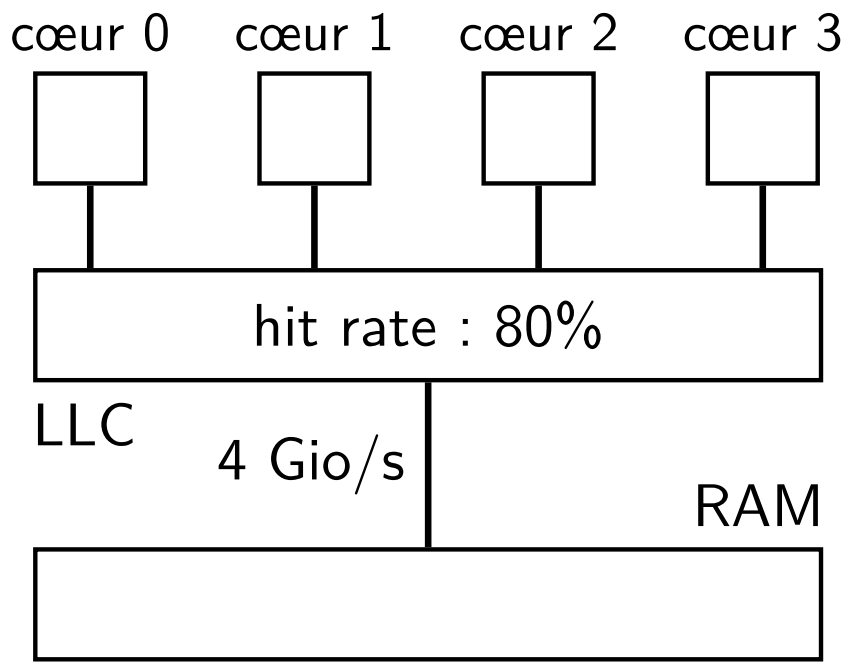
- En espace utilisateur, le fonctionnement des caches est transparent
  - Affecte uniquement les performances
  - Un contrôle fin est possible avec `prefetch*` et `clflush`

- En noyau, le fonctionnement des caches est souvent transparent
  - Affecte également les performances
  - Peut affecter le fonctionnement du système $\rightarrow$ DMA, CGA, ...
  - Un contrôle explicite est possible avec `wbinvd` et la pagination

Plan du cours

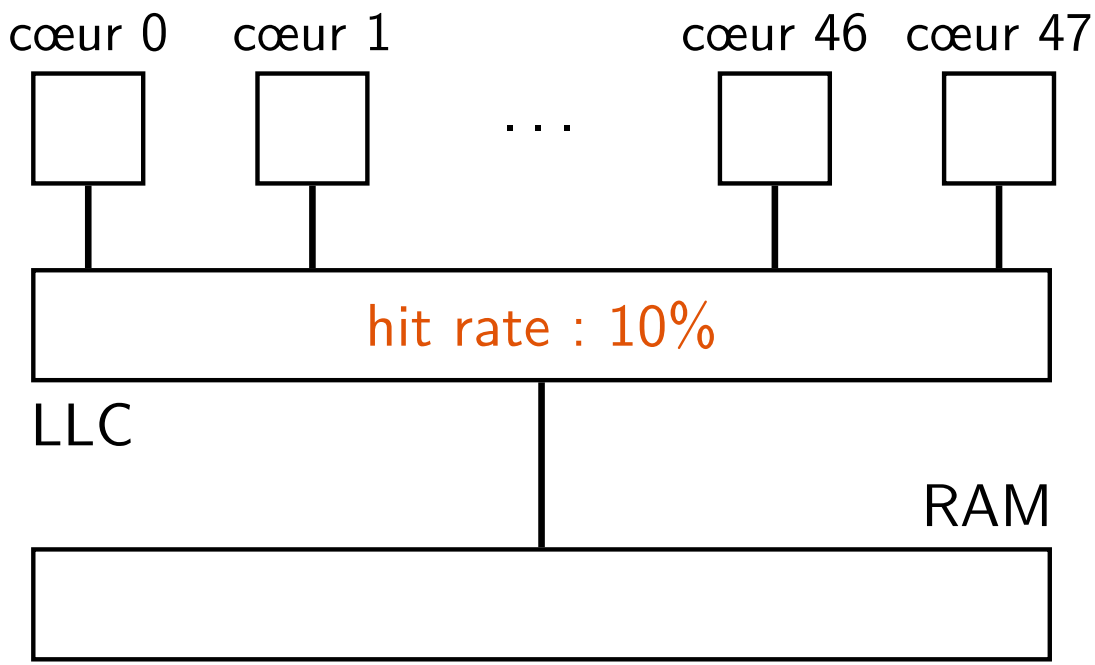
Multicœur et contention matérielle

- Entre 1 et $\sim$ 16 cœurs $\rightarrow$ problème de latence mémoire
  - Solution : mémoire cache $\rightarrow$ réduit la latence mémoire
  - Bonus : réduit le nombre d'accès à la mémoire principale



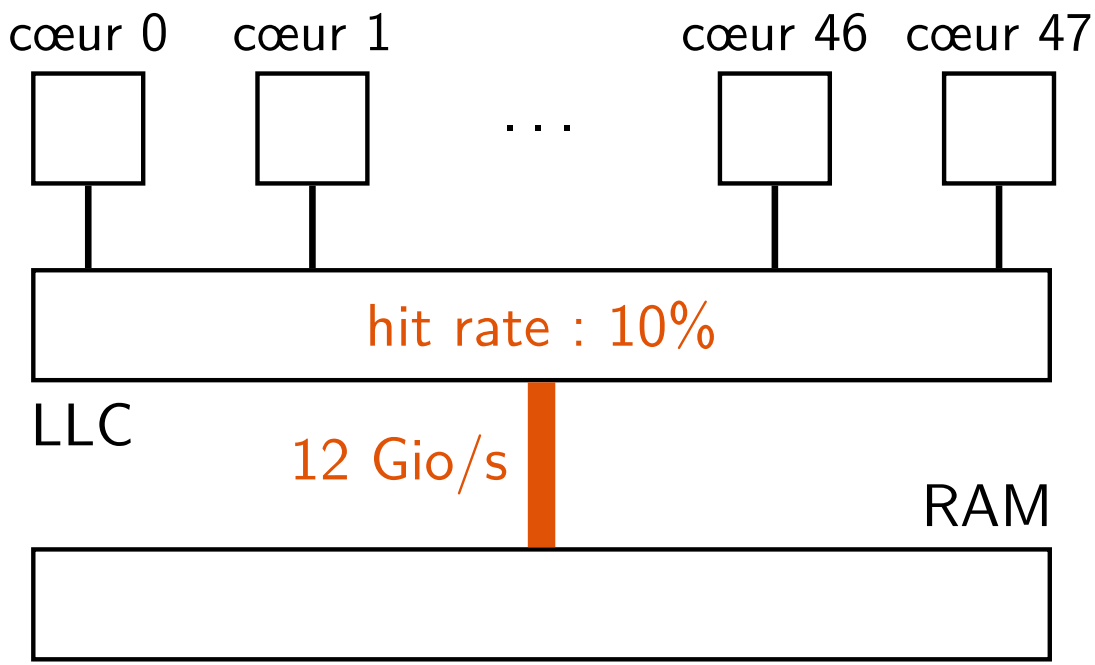
Multicœur et contention matérielle

- Entre 1 et $\sim$ 16 cœurs $\rightarrow$ problème de latence mémoire
  - Solution : mémoire cache $\rightarrow$ réduit la latence mémoire
  - Bonus : réduit le nombre d'accès à la mémoire principale
- Au delà de 16 cœurs
  - Faible localité dans les caches partagés



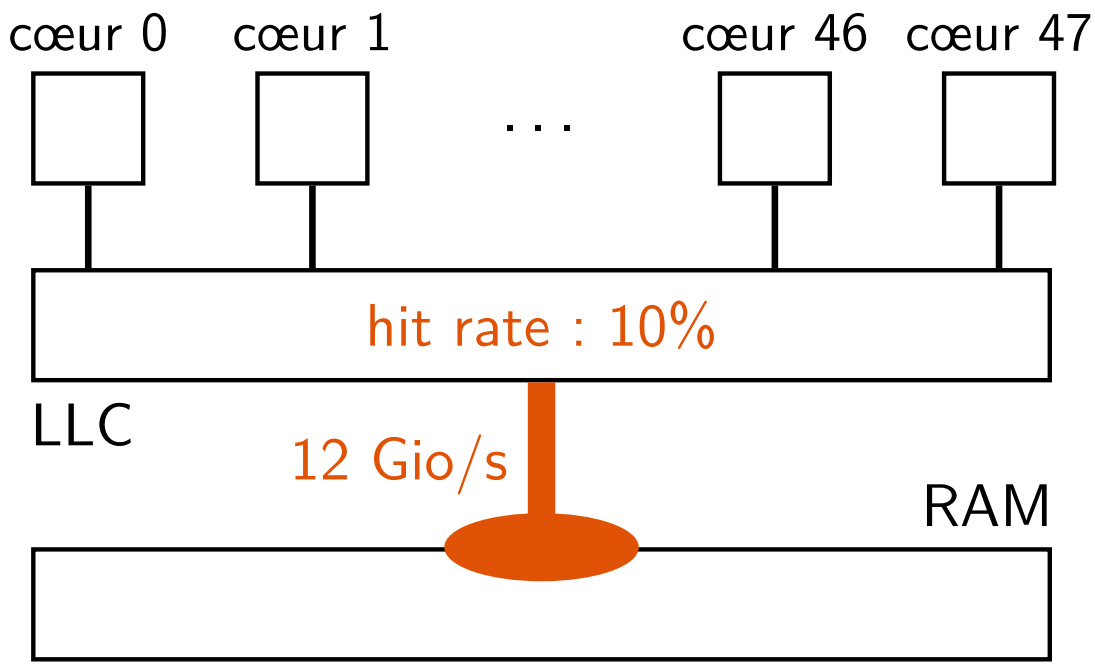
Multicœur et contention matérielle

- Entre 1 et $\sim$ 16 cœurs $\rightarrow$ problème de latence mémoire
  - Solution : mémoire cache $\rightarrow$ réduit la latence mémoire
  - Bonus : réduit le nombre d'accès à la mémoire principale
- Au delà de 16 cœurs
  - Faible localité dans les caches partagés
  - Augmente le nombre d'accès à la mémoire principale



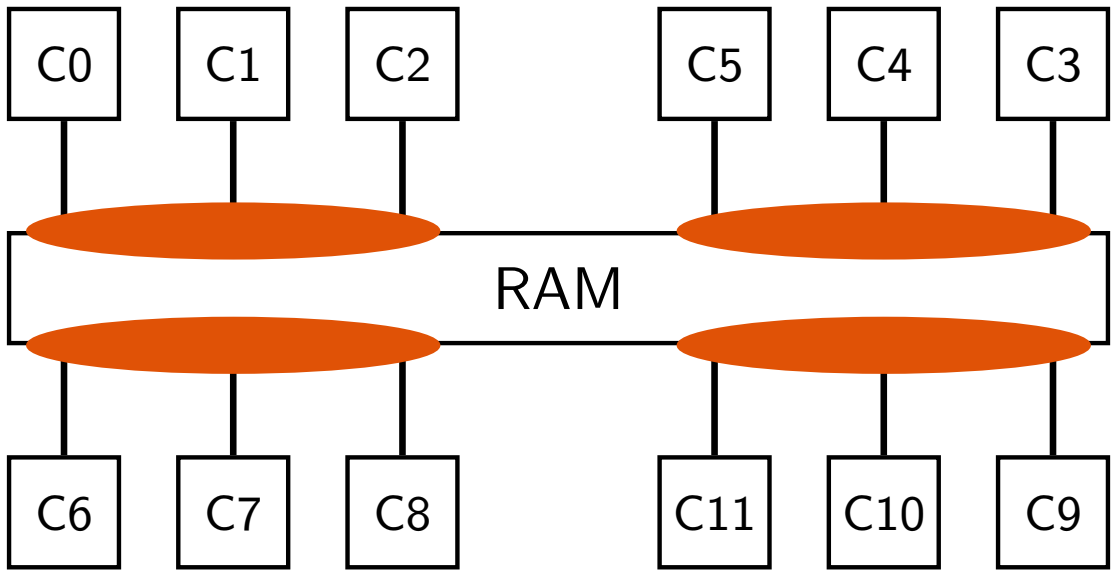
Multicœur et contention matérielle

- Entre 1 et $\sim$ 16 cœurs $\rightarrow$ problème de latence mémoire
  - Solution : mémoire cache $\rightarrow$ réduit la latence mémoire
  - Bonus : réduit le nombre d'accès à la mémoire principale
- Au delà de 16 cœurs $\rightarrow$ problème de débit mémoire
  - Faible localité dans les caches partagés
  - Augmente le nombre d'accès à la mémoire principale
  - Saturation de la mémoire principale



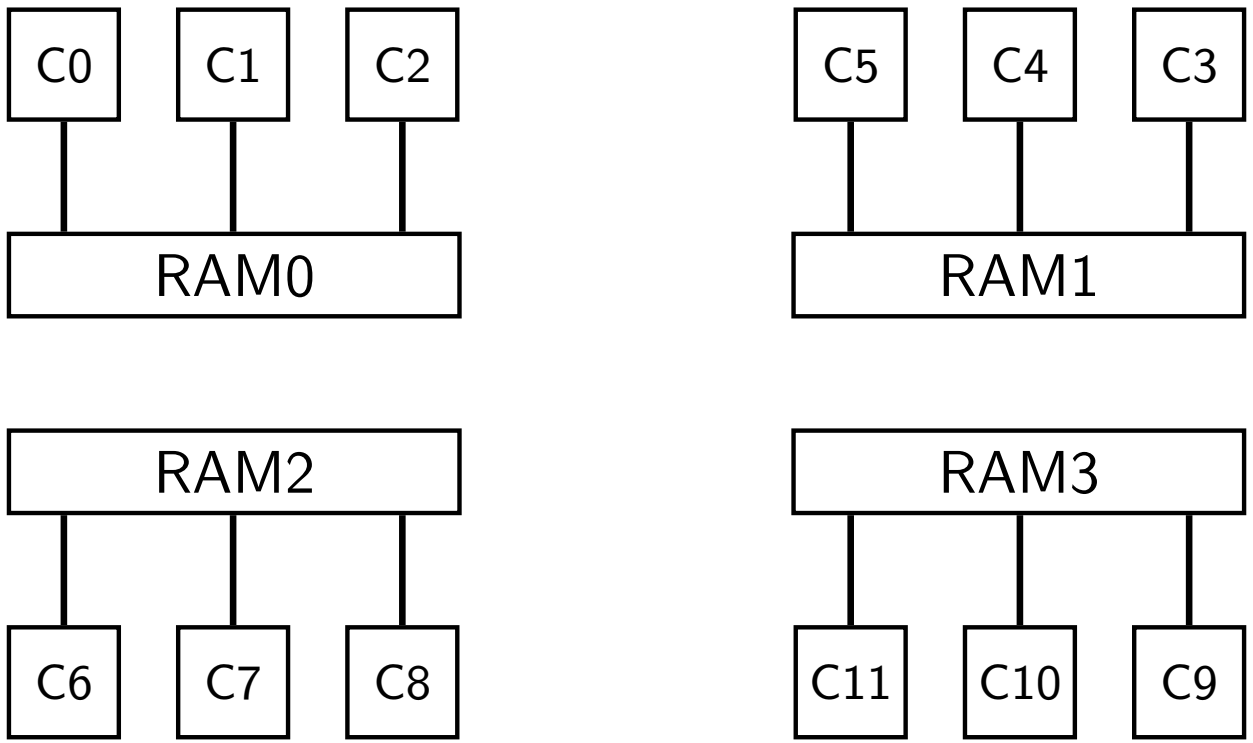
Architecture *Non Uniform Memory Access*

- Problème classique : trop de clients pour une seule ressource



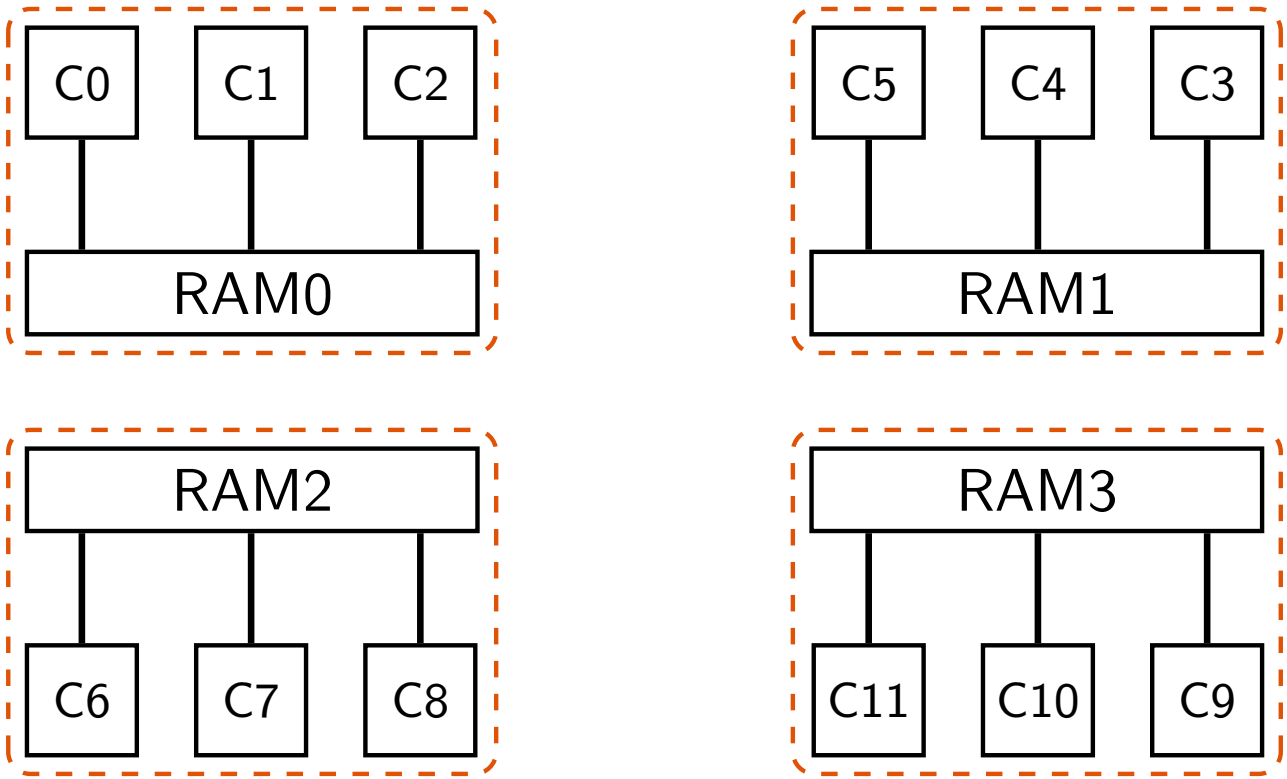
Architecture *Non Uniform Memory Access*

- Problème classique : trop de clients pour une seule ressource
- Solution classique : fragmenter / distribuer la ressource



Architecture *Non Uniform Memory Access*

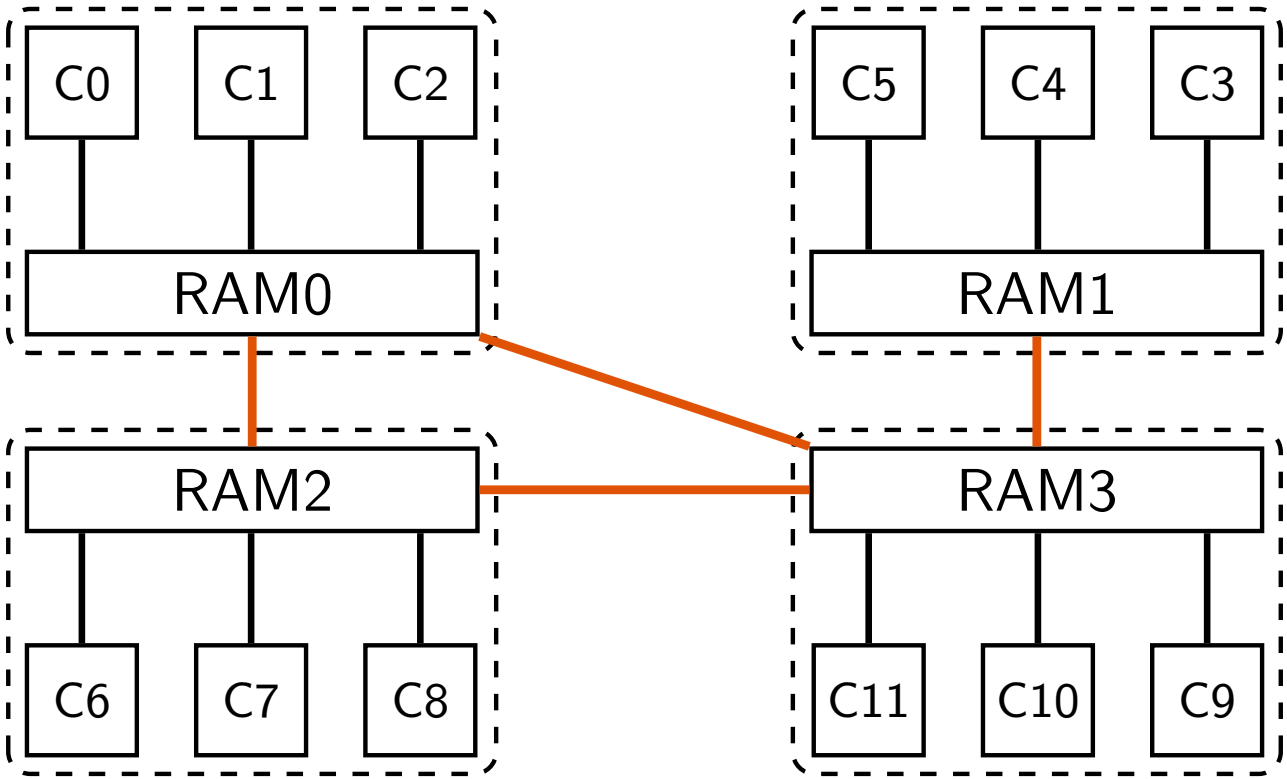
- Problème classique : trop de clients pour une seule ressource
- Solution classique : fragmenter / distribuer la ressource



- Chaque ensemble (cœurs + RAM) s'appelle un nœud

Architecture *Non Uniform Memory Access*

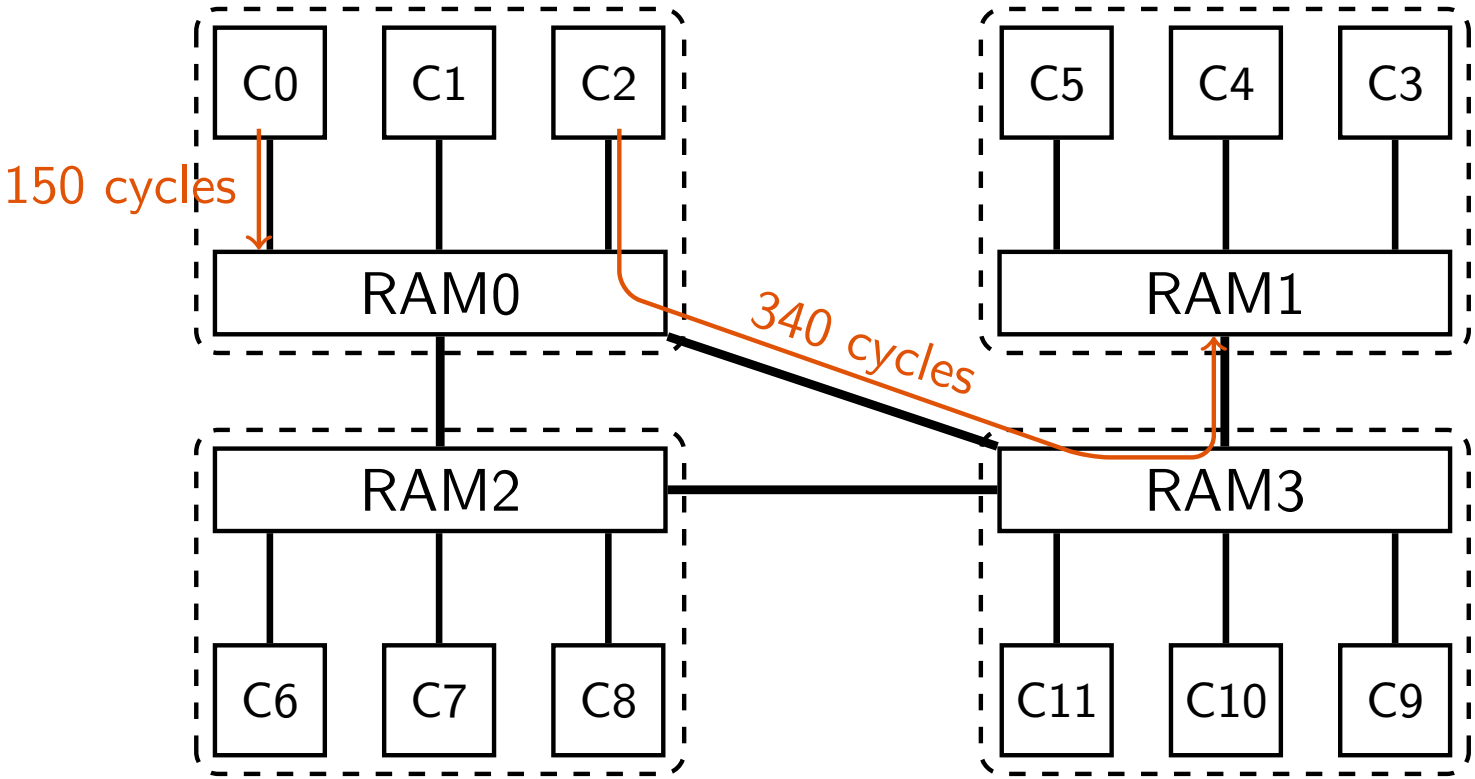
- Problème classique : trop de clients pour une seule ressource
- Solution classique : fragmenter / distribuer la ressource



- Chaque ensemble (cœurs + RAM) s'appelle un nœud
- Tout cœur peut accéder à n'importe quelle RAM via l'interconnect

Architecture *Non Uniform Memory Access*

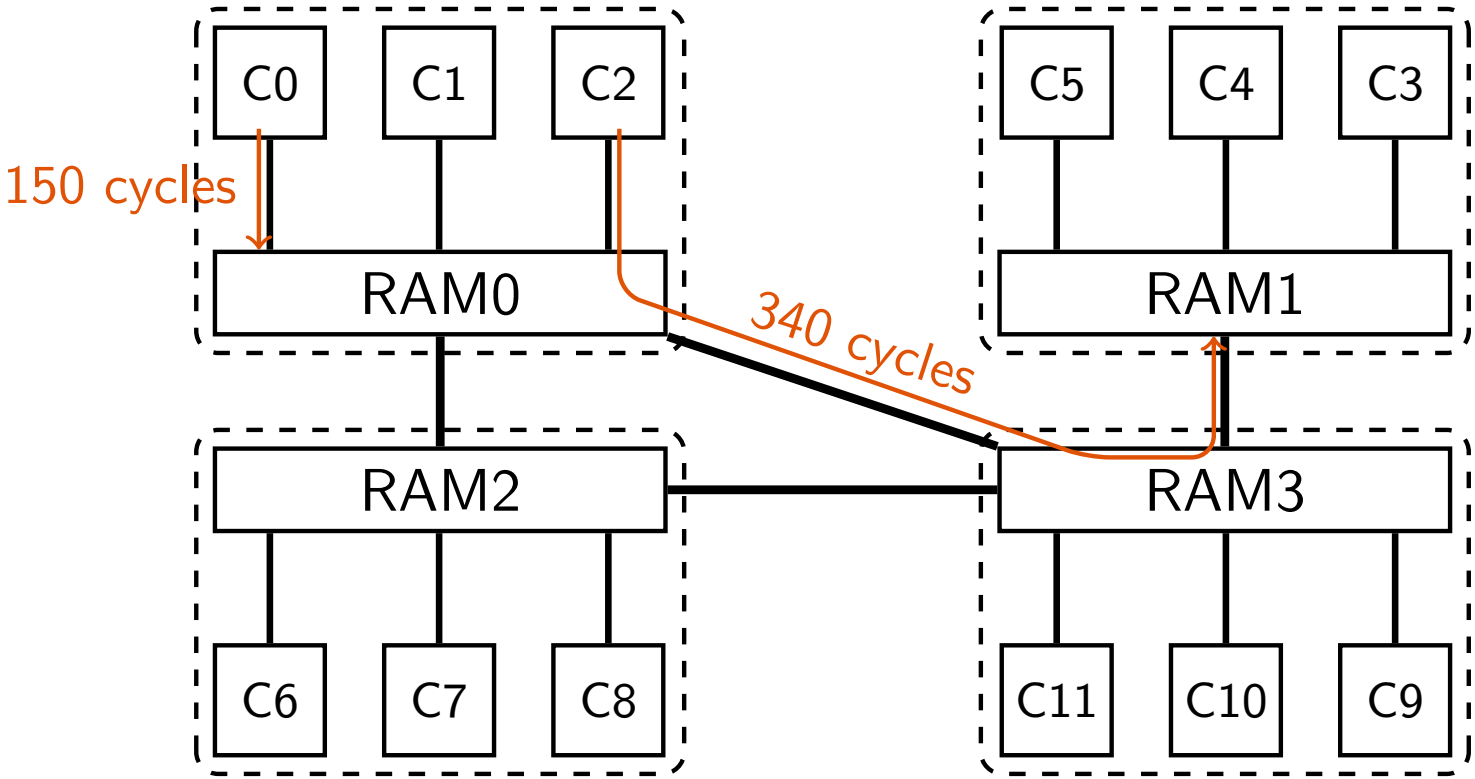
- Problème classique : trop de clients pour une seule ressource
- Solution classique : fragmenter / distribuer la ressource



- Chaque ensemble (cœurs + RAM) s'appelle un nœud
- Tout cœur peut accéder à n'importe quelle RAM via l'interconnect
- Accéder à la RAM locale est plus rapide qu'à une RAM distante

Architecture *Non Uniform Memory Access*

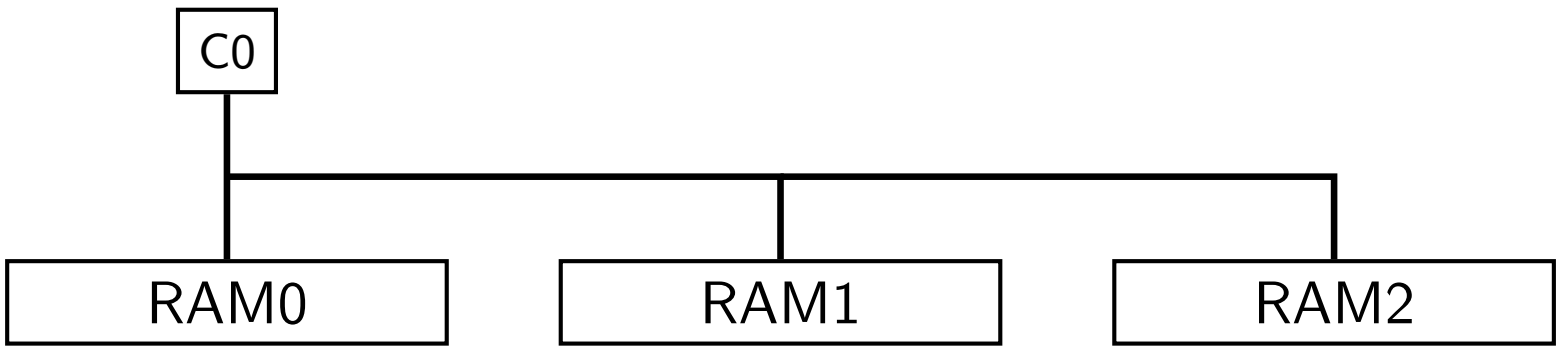
- Problème classique : trop de clients pour une seule ressource
- Solution classique : fragmenter / distribuer la ressource



- Chaque ensemble (cœurs + RAM) s'appelle un nœud NUMA
- Tout cœur peut accéder à n'importe quelle RAM via l'interconnect
- Accéder à la RAM locale est plus rapide qu'à une RAM distante
  - Les temps d'accès à la mémoire principale ne sont plus uniformes

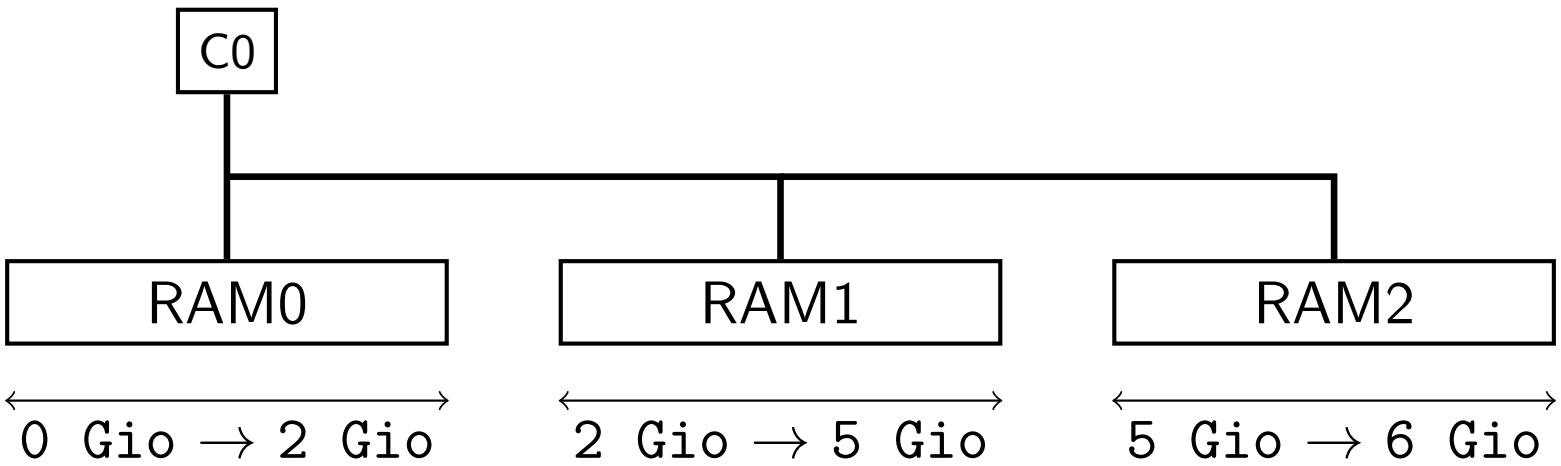
Architecture NUMA et adressage physique

- Objectif pour les constructueurs : compatibilité ascendante
  - Exécution possible du code legacy sur les nouvelles machines
- Le matériel doit présenter un espace d'adressage unique au logiciel



Architecture NUMA et adressage physique

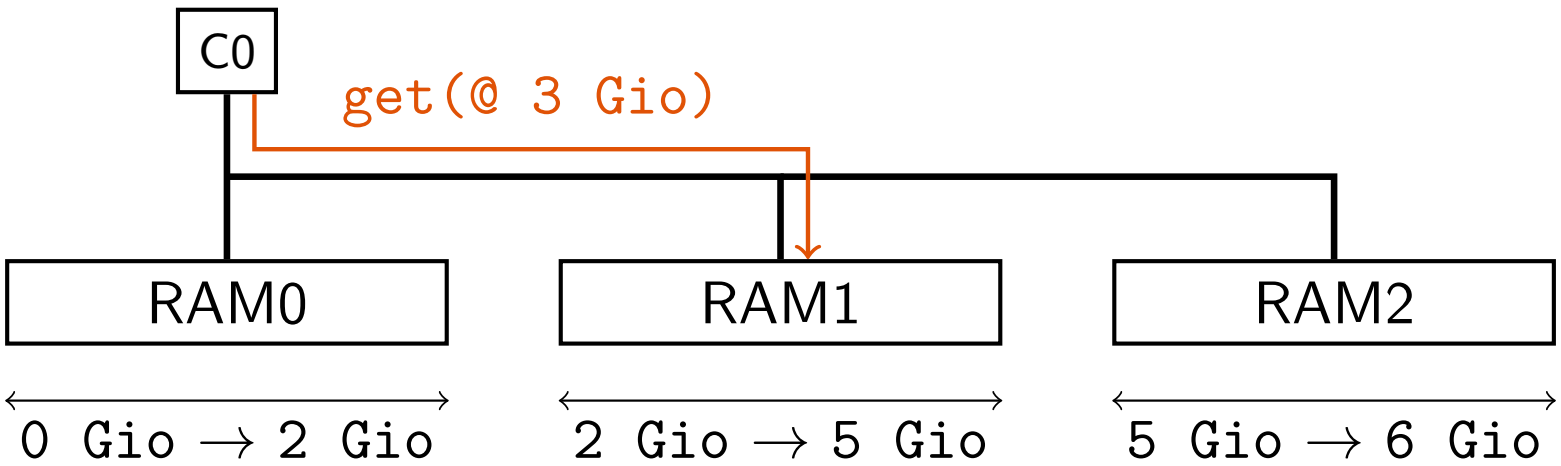
- Objectif pour les constructueurs : compatibilité ascendante
  - Exécution possible du code legacy sur les nouvelles machines
- Le matériel doit présenter un espace d'adressage unique au logiciel



- Les mémoires principales forment une partition de l'espace d'adressage physique

Architecture NUMA et adressage physique

- Objectif pour les constructueurs : compatibilité ascendante
  - Exécution possible du code legacy sur les nouvelles machines
- Le matériel doit présenter un espace d'adressage unique au logiciel



- Les mémoires principales forment une partition de l'espace d'adressage physique
- Le matériel route automatiquement les requêtes mémoire vers la bonne mémoire principale

Contention matérielle et architecture NUMA : résumé

- Augmentation du nombre de cœurs = augmentation de la pression mémoire → problème de débit mémoire
  - Au delà de ∼ 16 cœurs, la mémoire principale ne peut plus servir toutes les requêtes à temps → contention mémoire

- Solution : séparer le mémoire principale en plusieurs unités de mémoire indépendantes → architecture NUMA
  - Répartition de la charge mémoire
  - Le matériel (cœurs + mémoires) sont regroupés en nœuds NUMA
  - Les nœuds sont connectés entre eux par l'interconnect

- Une machine NUMA peut exécuter du code legacy
  - Chaque cœur peut accéder à l'ensemble de la mémoire
  - Le matériel présente un espace d'adressage physique unique au logiciel

- Le temps d'accès à la mémoire principale dépend du cœur et de l'adresse physique considérée

Cohérence de cache NUMA

- Un protocole de cohérence de cache assure la cohérence séquentielle
  - Les protocoles classiques (MESI, MOESI) utilise un RW-lock distribué
  - Le RW-lock est implémenté en associant un état à chaque ligne
  - Les caches maintiennent les états à jour avec des broadcasts

Cohérence de cache NUMA

- Un protocole de cohérence de cache assure la cohérence séquentielle
  - Les protocoles classiques (MESI, MOESI) utilise un RW-lock distribué
  - Le RW-lock est implémenté en associant un état à chaque ligne
  - Les caches maintiennent les états à jour avec des broadcasts

Cohérence de cache NUMA

- Un protocole de cohérence de cache assure la cohérence séquentielle
  - Les protocoles classiques (MESI, MOESI) utilise un RW-lock distribué
  - Le RW-lock est implémenté en associant un état à chaque ligne
  - Les caches maintiennent les états à jour avec des broadcasts

- Chaque broadcast passe par l'interconnect
  - Ajout de latence lors de l'acquisition d'une nouvelle ligne ($I \rightarrow *$)
  - Ajout de latence lors d'une mise à jour de l'état (sauf $E \rightarrow M$)
  - Consommation de bande passante

Cohérence de cache NUMA

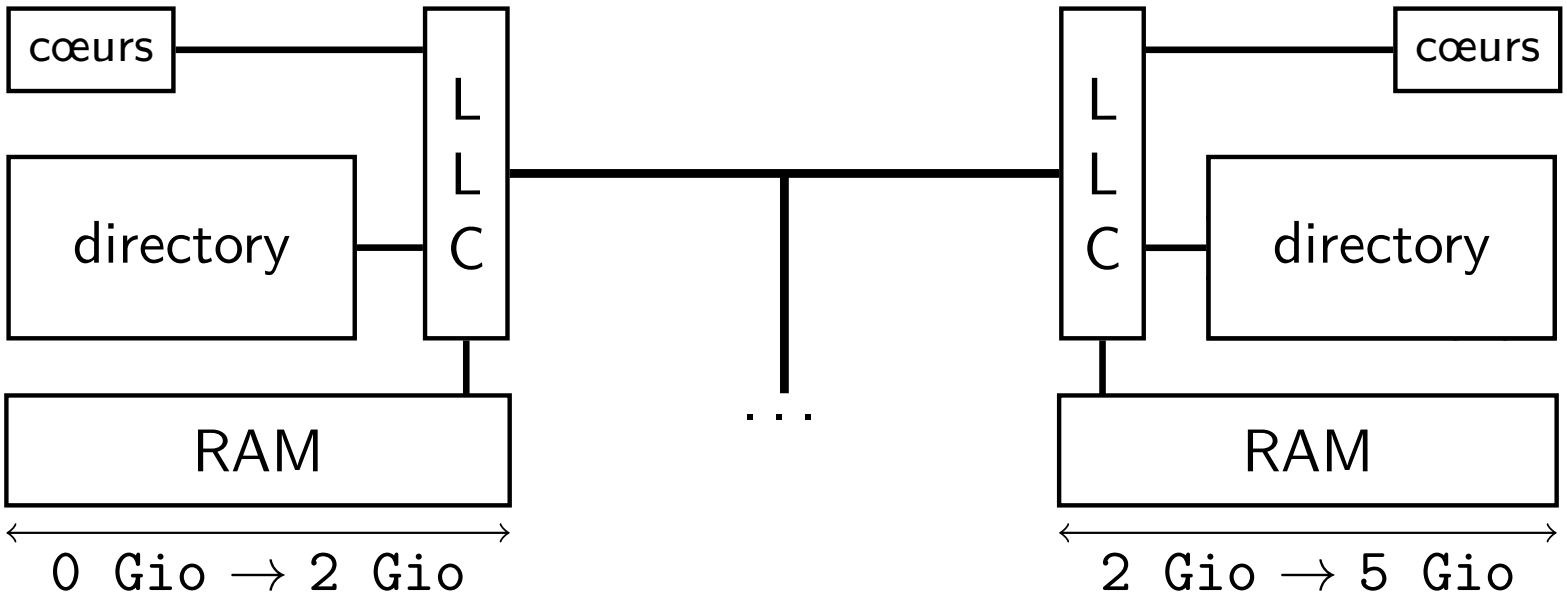
- Un protocole de cohérence de cache assure la cohérence séquentielle
  - Les protocoles classiques (MESI, MOESI) utilise un RW-lock distribué
  - Le RW-lock est implémenté en associant un état à chaque ligne
  - Les caches maintiennent les états à jour avec des broadcasts

- Chaque broadcast passe par l'interconnect
  - Ajout de latence lors de l'acquisition d'une nouvelle ligne (I $\rightarrow$ *)
  - Ajout de latence lors d'une mise à jour de l'état (sauf E $\rightarrow$ M)
  - Consommation de bande passante

- Simple à implémenter $\rightarrow$ conservation du protocole existant
  - Encore présent sur les vieilles machines AMD
  - Encore présent sur les machines Intel QPI

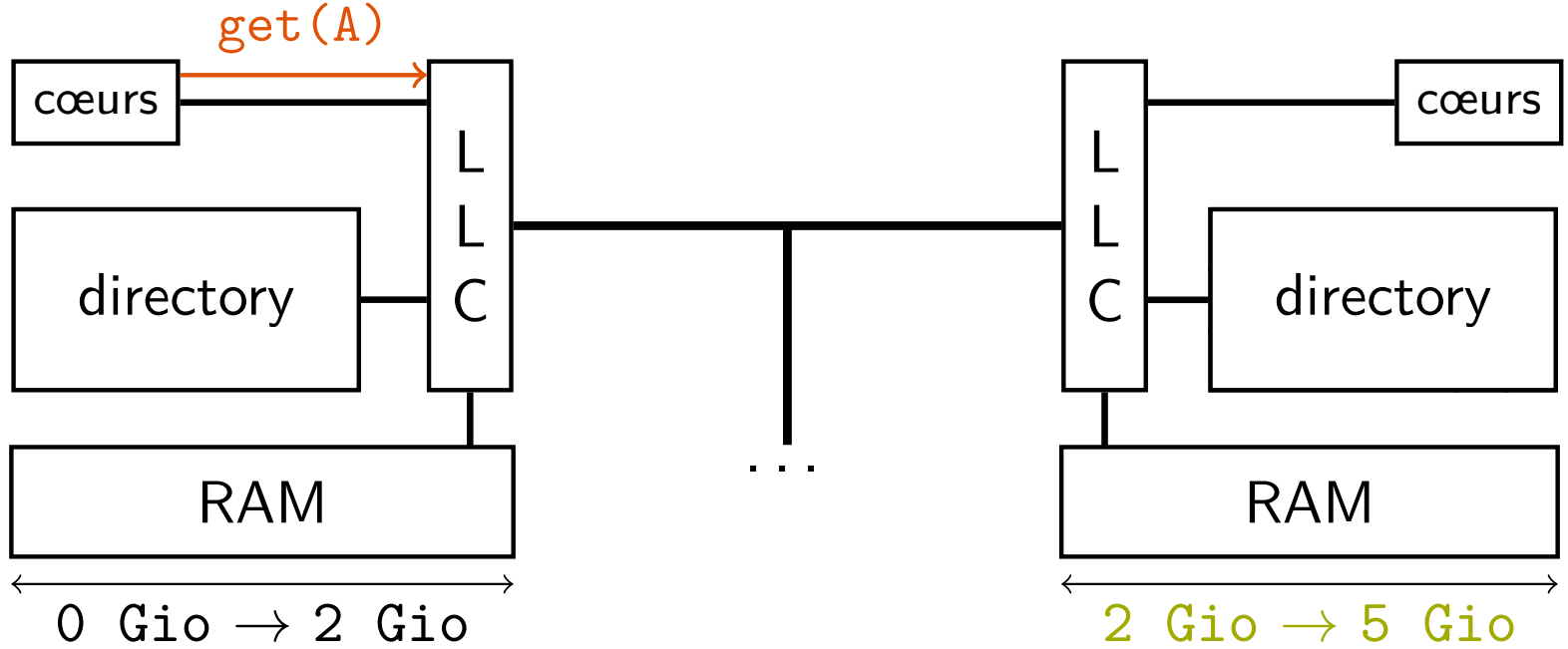
Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*



Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*

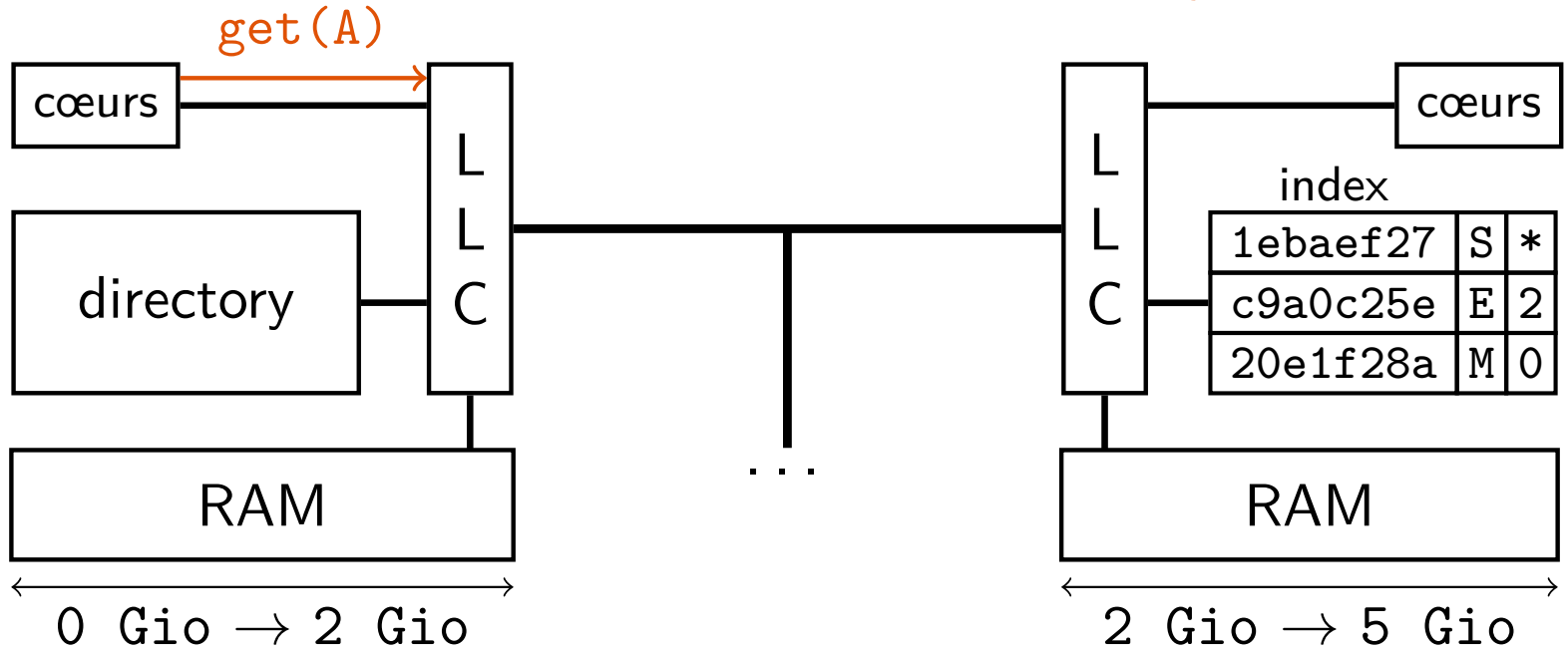


$A\varphi$ = 0x00000c9a0c25e $\simeq$ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique

Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*

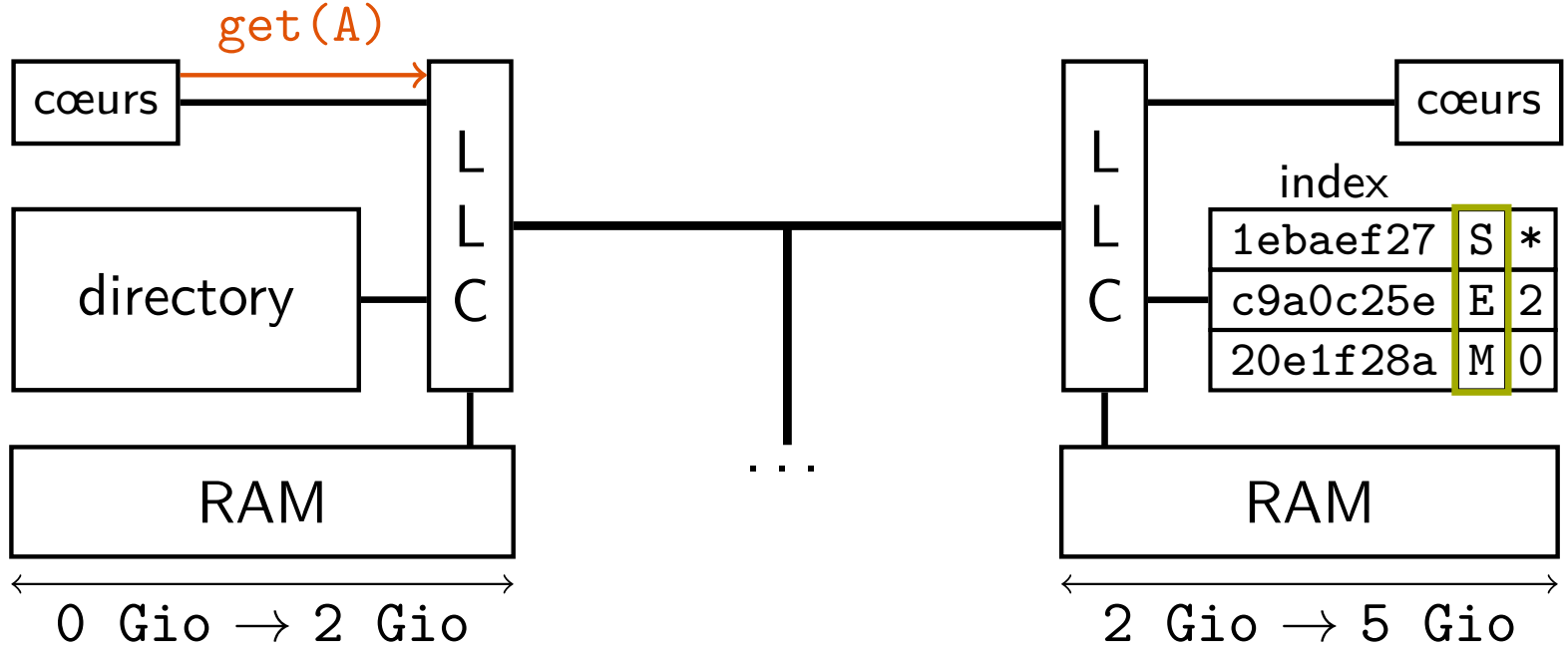


$A\varphi$ = 0x00000c9a0c25e ≃ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique
- Le *home node* stocke pour chacune des lignes associées

Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*

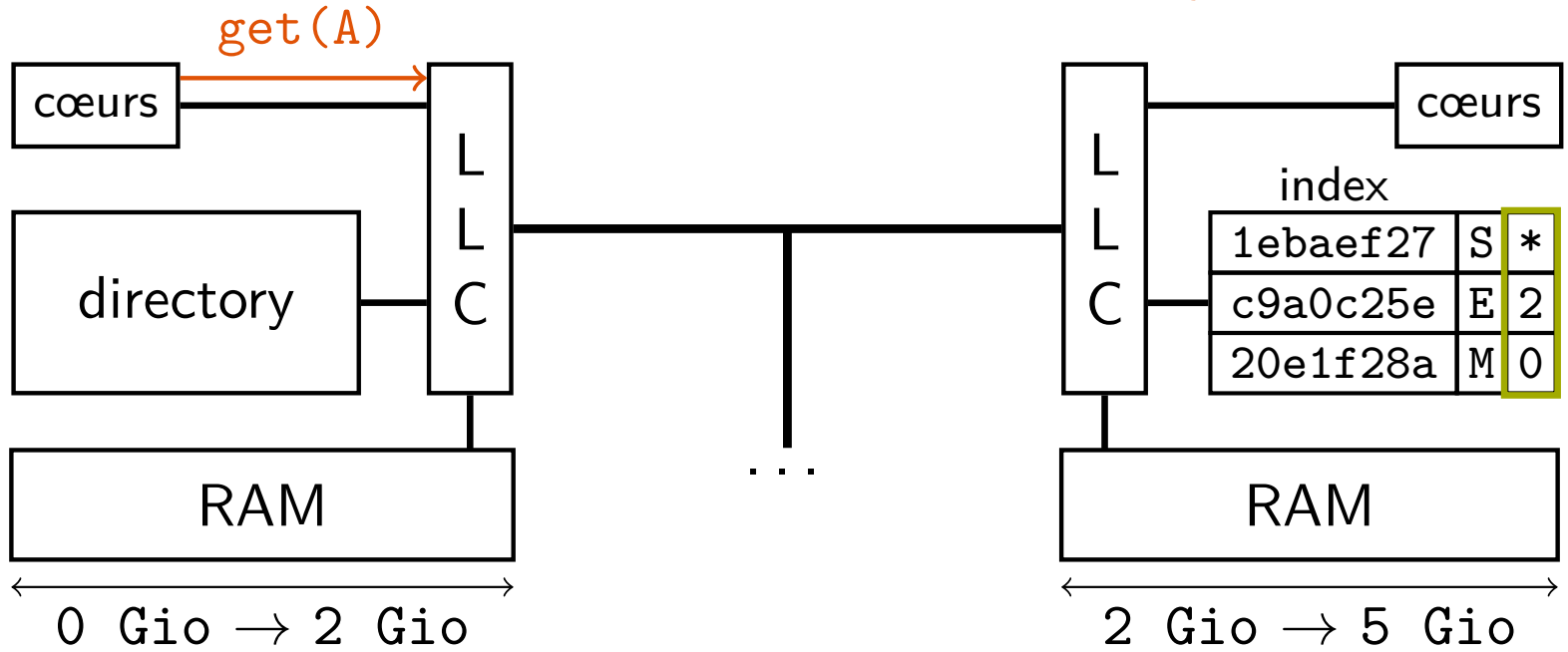


$A\varphi$ = 0x00000c9a0c25e ≃ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique
- Le *home node* stocke pour chacune des lignes associées
  - L'état de la ligne → MESI, MOESI, états supplémentaires

Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*

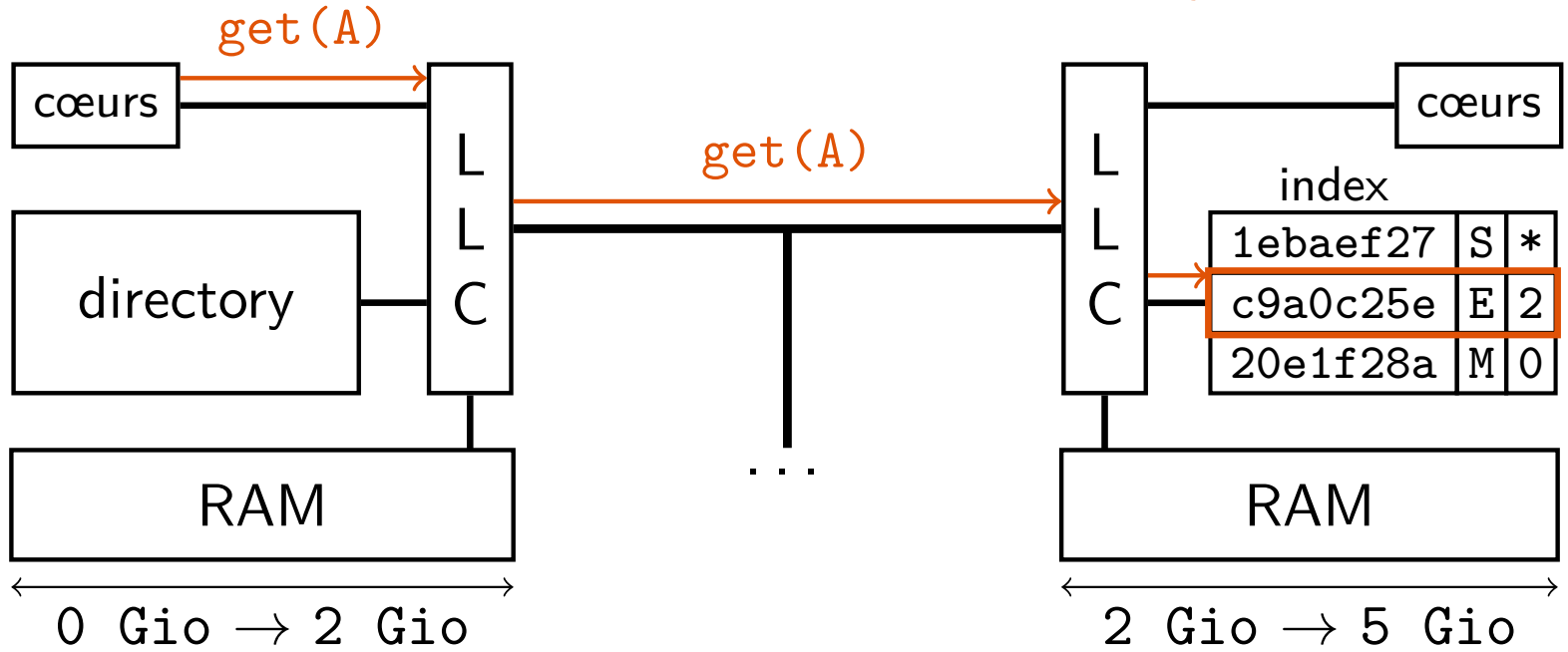


$A\varphi$ = 0x00000c9a0c25e ≃ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique
- Le *home node* stocke pour chacune des lignes associées
  - L'état de la ligne → MESI, MOESI, états supplémentaires
  - Les nœuds propriétaires → qui détient la ligne en cache

Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*

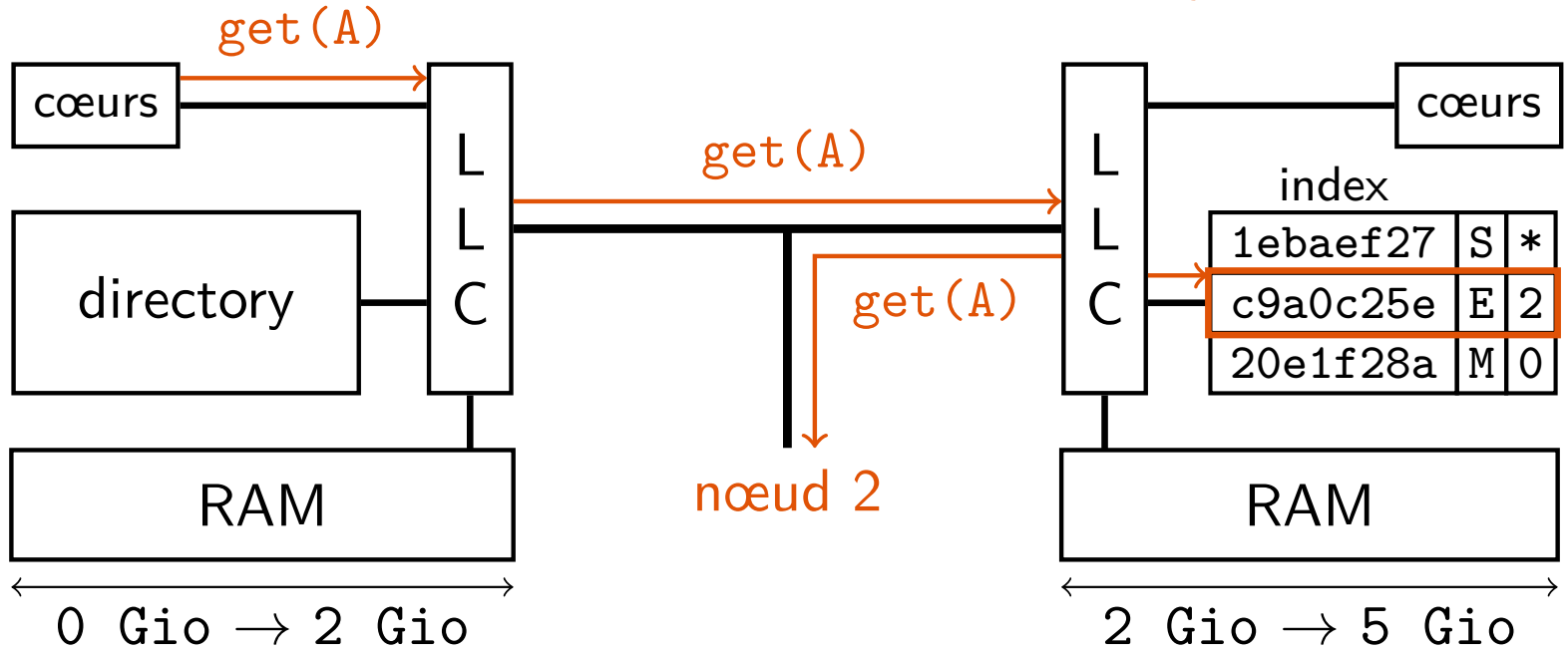


$A\varphi$ = 0x00000c9a0c25e ≃ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique
- Le *home node* stocke pour chacune des lignes associées
  - L'état de la ligne → MESI, MOESI, états supplémentaires
  - Les nœuds propriétaires → qui détient la ligne en cache

Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*

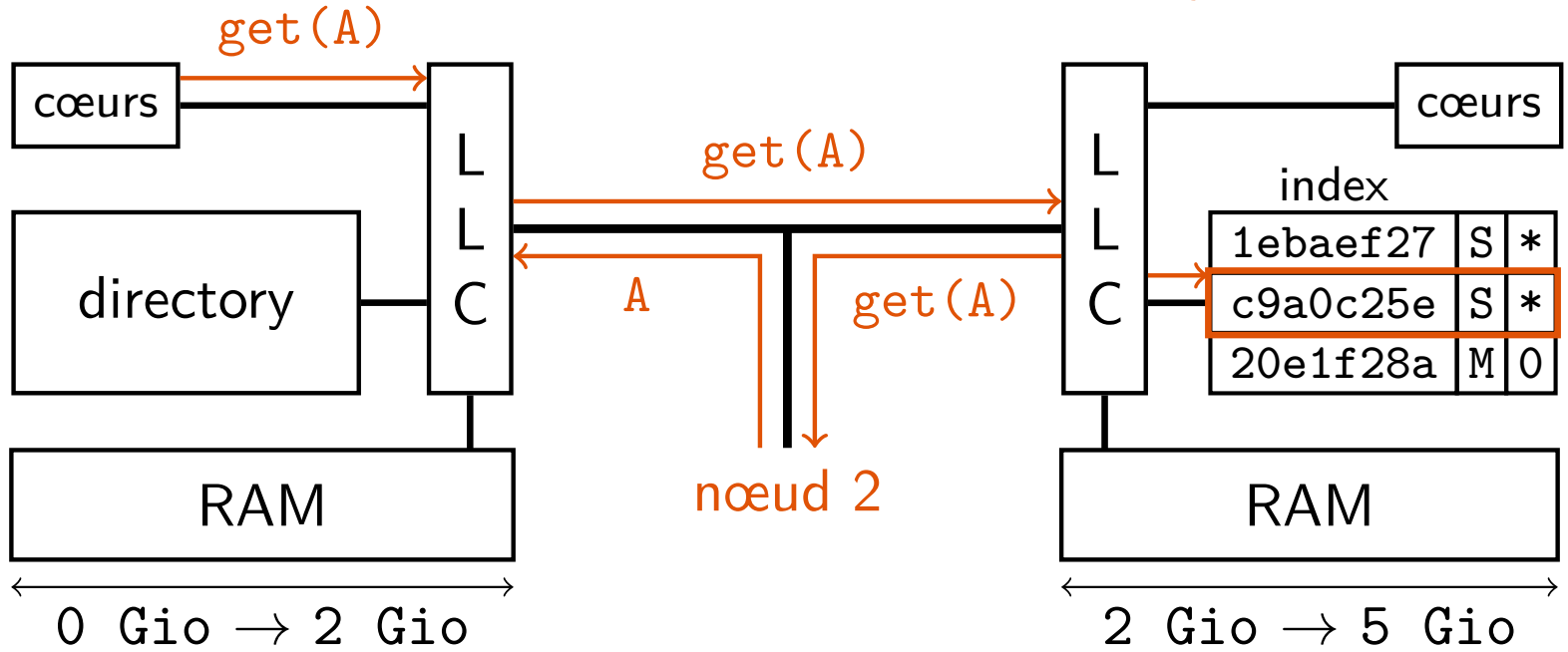


$A\varphi$ = 0x00000c9a0c25e ≃ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique
- Le *home node* stocke pour chacune des lignes associées
  - L'état de la ligne → MESI, MOESI, états supplémentaires
  - Les nœuds propriétaires → qui détient la ligne en cache

Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*



$A\varphi$ = 0x00000c9a0c25e $\simeq$ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique
- Le *home node* stocke pour chacune des lignes associées
  - L'état de la ligne → MESI, MOESI, états supplémentaires
  - Les nœuds propriétaires → qui détient la ligne en cache

Home node et cache directory : complément

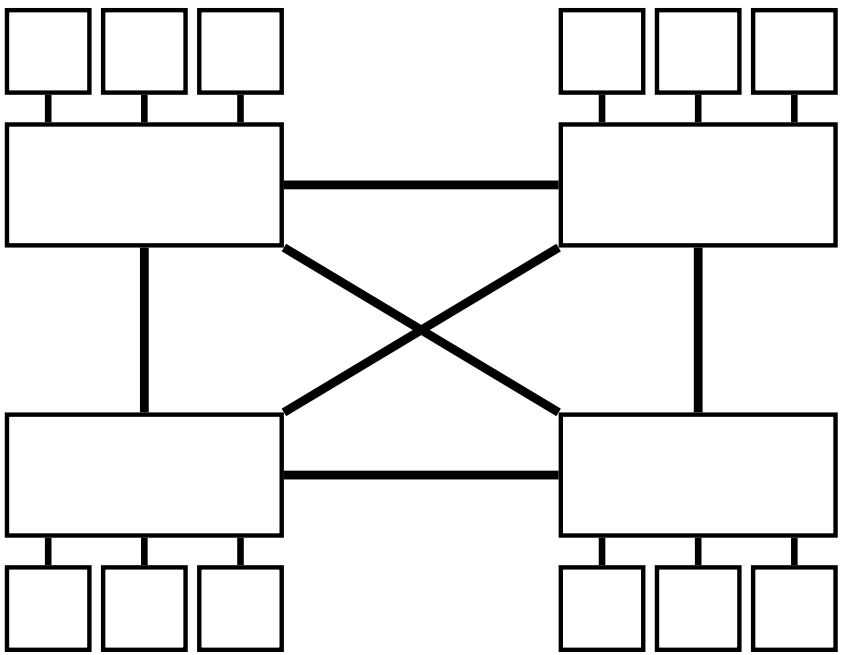
- Un protocole *directory based* réduit le nombre de broadcast entre les différents nœuds NUMA
  - Libère de la bande passante sur l'interconnect
  - Ne change rien aux broadcasts à l'intérieur d'un nœud NUMA
  - Le broadcast entre les nœuds reste utile (exemple : S $\rightarrow$ M)

- Un protocole *directory based* augmente le nombre de *hop* nécessaire à l'obtention d'une nouvelle ligne
  - Si la donnée n'est pas présente dans les caches locaux, il faut toujours passer par le home node
  - Ajoute de la latence pour la communication entre nœuds NUMA

- Plus complexe à implémenter mais généralement plus performant
  - Présent sur les machines AMD récentes (HT-Assist 3)
  - Présent sur les machines Intel récentes (Intel UPI)

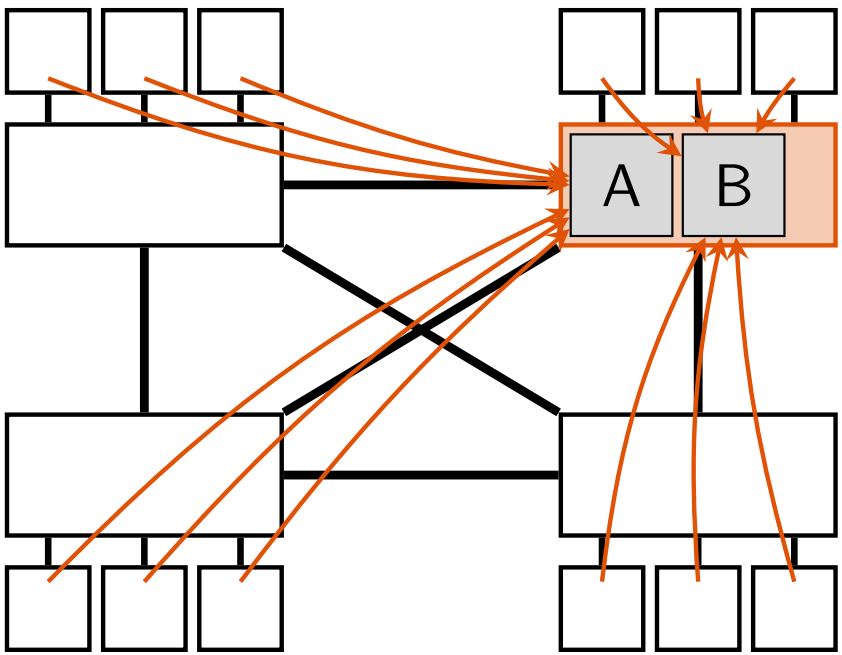
Architecture *Non Uniform Memory Access* et performance

- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique



Architecture *Non Uniform Memory Access* et performance

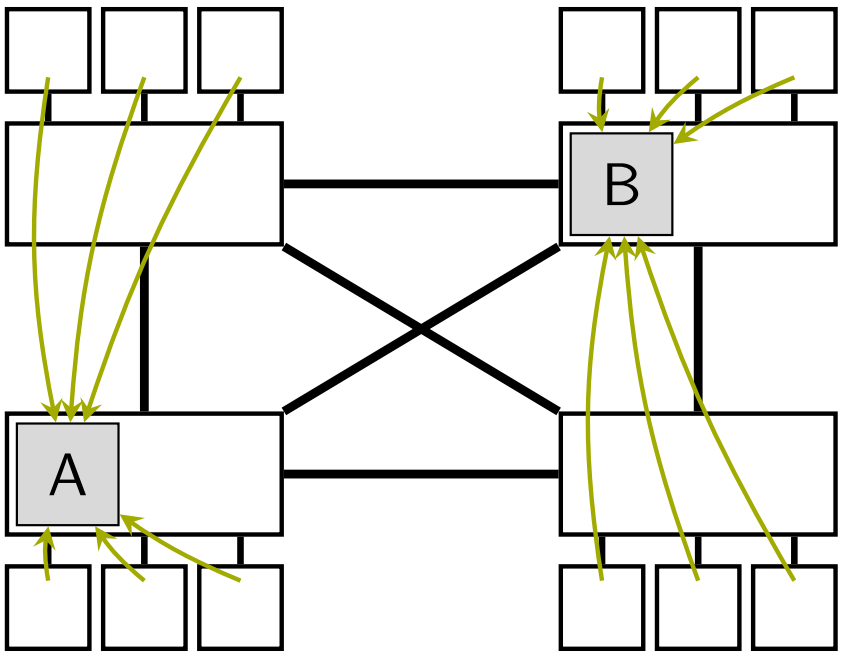
- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique
  - Répartition de la charge mémoire



- Si tous les cœurs accèdent au même nœud
  - Saturation d'une mémoire principale
  - Problème de débit mémoire
  - Effondrement des performances

Architecture *Non Uniform Memory Access* et performance

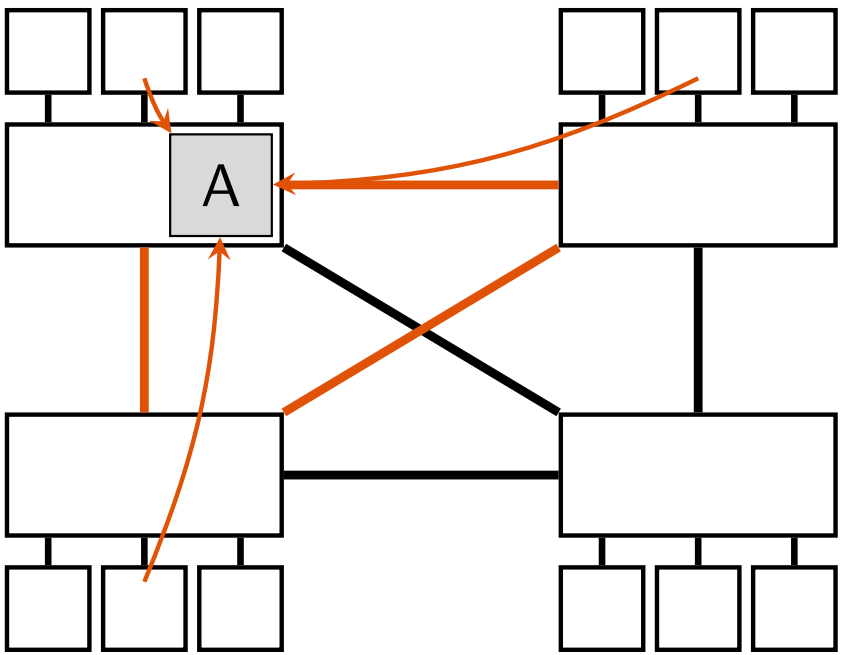
- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique
  - Répartition de la charge mémoire



- Si tous les cœurs accèdent au même nœud
  - Saturation d'une mémoire principale
  - Problème de débit mémoire
  - Effondrement des performances
- Répartir les données dans différents nœuds
  - Changer l'emplacement physique des données

Architecture *Non Uniform Memory Access* et performance

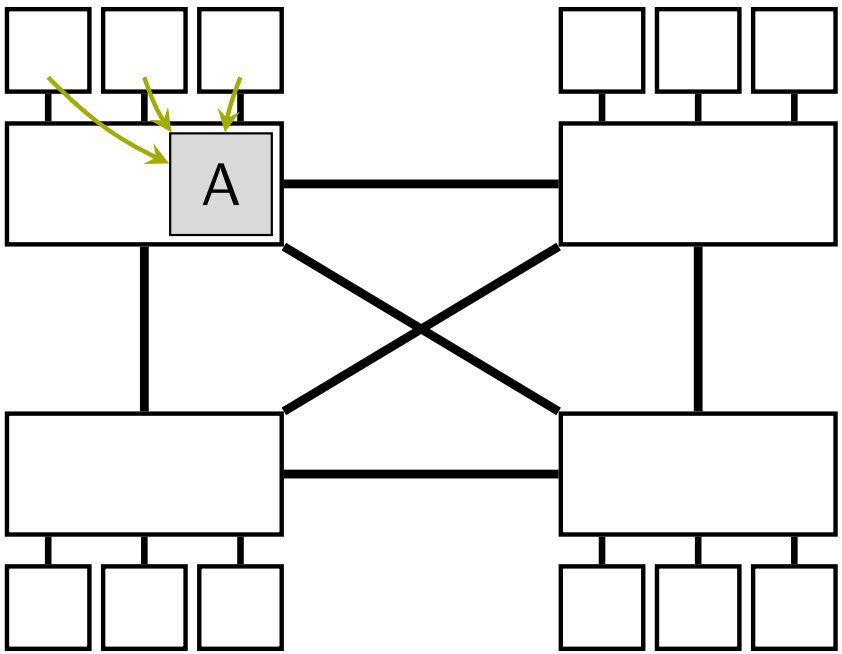
- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique
  - Répartition de la charge mémoire
  - Limitation des transferts entre nœuds



- Transfert de ligne entre nœud coûteux
  - Latence d'accès mémoire cache supplémentaire
  - Consommation de bande passante interconnect
  - Effondrement des performances

Architecture *Non Uniform Memory Access* et performance

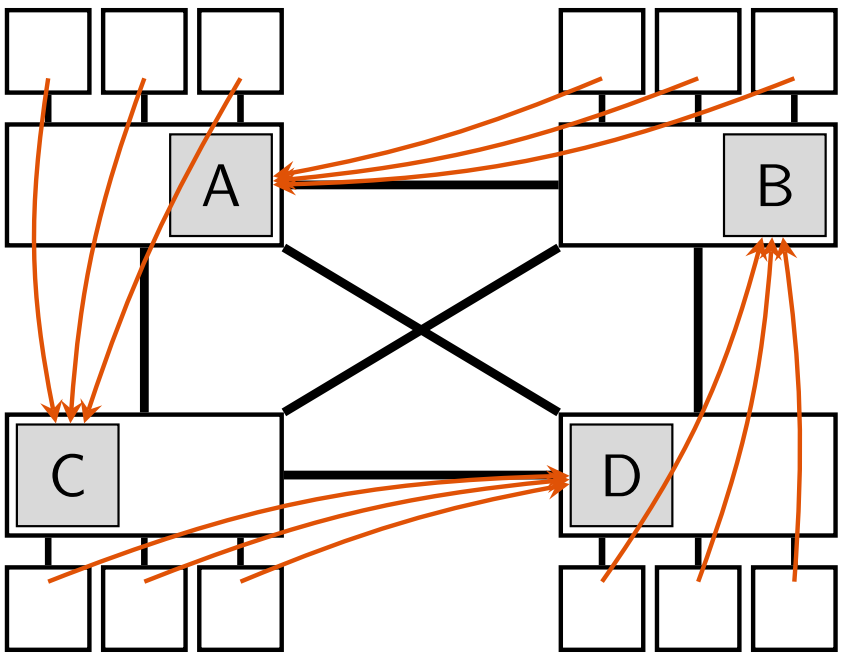
- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique
  - Répartition de la charge mémoire
  - Limitation des transferts entre nœuds



- Transfert de ligne entre nœud coûteux
  - Latence d'accès mémoire cache supplémentaire
  - Consommation de bande passante interconnect
  - Effondrement des performances
- Communiquer localement
  - Identifier les tâches qui communiquent (données partagées)
  - Migrer ces tâches sur un même nœud

Architecture *Non Uniform Memory Access* et performance

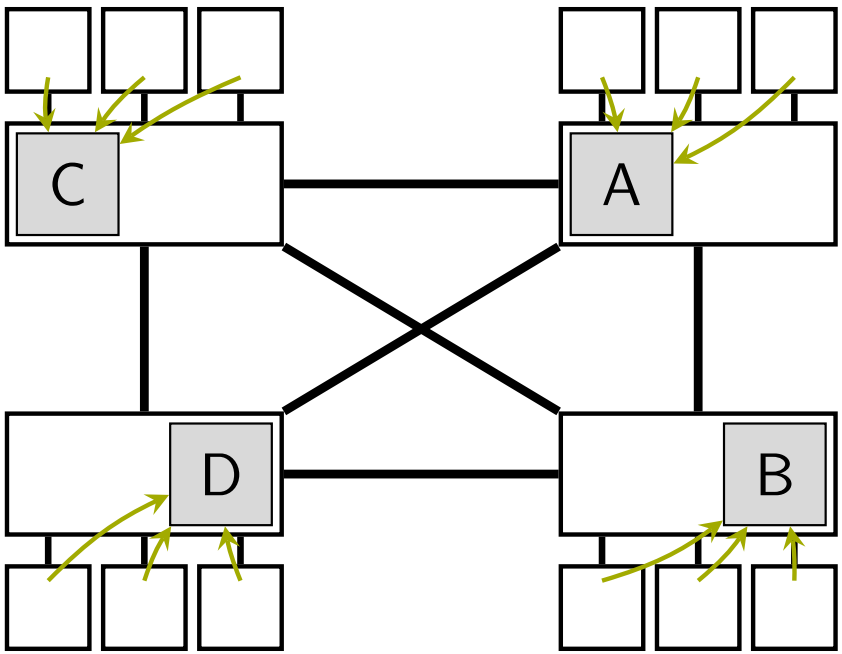
- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique
  - Répartition de la charge mémoire
  - Limitation des transferts entre nœuds
  - Localité des accès mémoire



- Accès distant plus lent qu'accès local
  - Latence RAM supplémentaire
  - Diminution des performances

Architecture *Non Uniform Memory Access* et performance

- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique
  - Répartition de la charge mémoire
  - Limitation des transferts entre nœuds
  - Localité des accès mémoire



- Accès distant plus lent qu'accès local
  - Latence RAM supplémentaire
  - Diminution des performances
- Accéder aux données locales
  - Changer l'emplacement des données
  - Migrer les tâches vers les données

Placement de ressources et performance

- Les performances dépendent du placement des tâches et des données entre elles et par rapport à la topologie mémoire

Placement de ressources et performance

- Les performances dépendent du placement des tâches et des données entre elles et par rapport à la topologie mémoire

- Créer les nouvelles tâches sur le bon nœud (au moment du `clone`)
  - Pas encore d'informations sur la nouvelle tâche
- Migrer périodiquement les tâches sur les bons nœuds
  - Changement de cache $\rightarrow$ perte de localité

Placement de ressources et performance

- Les performances dépendent du placement des tâches et des données entre elles et par rapport à la topologie mémoire

- Créer les nouvelles tâches sur le bon nœud (au moment du `clone`)
  - Pas encore d'informations sur la nouvelle tâche
- Migrer périodiquement les tâches sur les bons nœuds
  - Changement de cache $\rightarrow$ perte de localité

- Allouer les nouvelles données sur le bon nœud (au moment du

Placement de ressources et performance

- Les performances dépendent du placement des tâches et des données entre elles et par rapport à la topologie mémoire

- Créer les nouvelles tâches sur le bon nœud (au moment du `clone`)
  - Pas encore d'informations sur la nouvelle tâche
- Migrer périodiquement les tâches sur les bons nœuds
  - Changement de cache $\rightarrow$ perte de localité

- Allouer les nouvelles données sur le bon nœud (au moment du *first touch*)
  - Pas encore d'informations sur la nouvelle donnée
- Migrer périodiquement les données sur les bons nœuds
  - Coût du `memcpy` $\rightarrow$ bande passante, pollution de cache

Placement de ressources et performance

- Les performances dépendent du placement des tâches et des données entre elles et par rapport à la topologie mémoire

- Créer les nouvelles tâches sur le bon nœud (au moment du `clone`)
  - Pas encore d'informations sur la nouvelle tâche
- Migrer périodiquement les tâches sur les bons nœuds
  - Changement de cache $\rightarrow$ perte de localité

- Allouer les nouvelles données sur le bon nœud (au moment du *first touch*)
  - Pas encore d'informations sur la nouvelle donnée
- Migrer périodiquement les données sur les bons nœuds
  - Coût du `memcpy` $\rightarrow$ bande passante, pollution de cache

Placement de données : le *first touch*

- Intuition : la tâche qui alloue un espace en mémoire est souvent la tâche qui utilise cet espace → exemple : la pile
- Principe : allouer l'espace mémoire sur le nœud allocateur → le nœud qui touche l'espace mémoire en premier

```
void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

        if (!tree_contains(vaddr))
                segfault();

        current = current_node();
        paddr = allocate_page_on_node(current);
        map_page(vaddr, paddr);
}
```

Placement de données : le *first touch*

- Intuition : la tâche qui alloue un espace en mémoire est souvent la tâche qui utilise cet espace → exemple : la pile
- Principe : allouer l'espace mémoire sur le nœud allocateur → le nœud qui touche l'espace mémoire en premier

```
void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

        if (!tree_contains(vaddr))
                segfault();

        current = current_node();
        paddr = allocate_page_on_node(current);
        map_page(vaddr, paddr);
}
```

Placement de données : le *first touch*

- Intuition : la tâche qui alloue un espace en mémoire est souvent la tâche qui utilise cet espace → exemple : la pile
- Principe : allouer l'espace mémoire sur le nœud allocateur → le nœud qui touche l'espace mémoire en premier

```
void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

        if (!tree_contains(vaddr))
                segfault();

        current = current_node();
        paddr = allocate_page_on_node(current);
        map_page(vaddr, paddr);
}
```

- Politique par défault sous Linux
  - Configuration explicite avec `numactl --localalloc`

Placement de données : l'*interleaving*

- Observation : la répartition de la charge mémoire est le facteur le plus important pour les performances sur NUMA
- Principe : répartir équitablement les données sur tous les nœuds pour équilibrer la charge

```
void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

        if (!tree_contains(vaddr))
                segfault();

        random = choose_random_node();
        paddr = allocate_page_on_node(random);
        map_page(vaddr, paddr);
}
```

Placement de données : l'*interleaving*

- Observation : la répartition de la charge mémoire est le facteur le plus important pour les performances sur NUMA
- Principe : répartir équitablement les données sur tous les nœuds pour équilibrer la charge

```
void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

        if (!tree_contains(vaddr))
                segfault();

        random = choose_random_node();
        paddr = allocate_page_on_node(random);
        map_page(vaddr, paddr);
}
```

Placement de données : l'*interleaving*

- Observation : la répartition de la charge mémoire est le facteur le plus important pour les performances sur NUMA
- Principe : répartir équitablement les données sur tous les nœuds pour équilibrer la charge

```
void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

        if (!tree_contains(vaddr))
                segfault();

        random = choose_random_node();
        paddr = allocate_page_on_node(random);
        map_page(vaddr, paddr);
}
```

- Politique la moins risquée
  - Configuration avec `numactl --interleave`

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

- Exécution avec *first touch* :
- Exécution avec *interleaving* :

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

- Exécution avec *first touch* : rapide
- Exécution avec *interleaving* : lente → mauvaise localité

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

- Temps d'exécution avec *first touch* :
- Temps d'exécution avec *interleaving* :

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

- Temps d'exécution avec *first touch* : très lente → contention mémoire
- Temps d'exécution avec *interleaving* : rapide

Stratégies d'allocation mémoire : résumé

- Pour s'exécuter efficacement sur une architecture NUMA, le logiciel doit maximiser plusieurs critères
  - Une bonne répartition de la charge mémoire
  - La limitation des transferts entre nœuds NUMA
  - Une bonne localité des accès mémoire

- Le logiciel peut agir sur le placement des données et des tâches → ici, on s'intéresse uniquement aux données
  - Placement initial des données → sur quel nœud allouer la mémoire
  - Migration des données après allocation → pas vu dans ce cours

- Stratégie d'allocation *first touch*
  - Allouer sur le même nœud que la tâche qui alloue
  - Performant quand chaque tâche alloue sa mémoire

- Stratégie d'allocation *interleaving*
  - Répartir équitablement les données entre les nœuds
  - Évite la contention mémoire → stratégie du moindre mal

Conclusion : mémoire cache en monocœur

- La mémoire principale fournit les données trop lentement par rapport à la consommation du processeur
  - Problème de latence d'accès aux données et aux instructions
  - La mémoire cache est une zone de stockage intermédiaire
  - Échange des lignes de cache avec la mémoire principale et le CPU
  - Réduit la latence d'accès grâce aux principes de localité
- Il existe plusieurs types de cache $\rightarrow$ cache direct, cache associatif
  - Utilise l'adresse des lignes de cache pour choisir où les stocker
  - M lignes sont en conflit si elles ne peuvent pas être stockées en même temps dans le cache
  - Les caches associatifs utilisent des stratégies d'éviction pour minimiser l'impact des conflits
- Les mémoires cache tentent de prédire les accès futurs pour charger les lignes en avance
  - Un chargement de ligne qui résulte d'une prédiction est un *prefetch*
  - Le *prefetching* est fait automatiquement par les contrôleurs de cache
  - Le *prefetching* fonctionne bien pour les motifs d'accès simples

Conclusion : multicœur et cohérence de caches

- Dans une architecture multicœur, chaque cœur a son propre cache
  - Du point de vue du logiciel, la mémoire doit garantir la cohérence séquentielle des accès
  - Les évictions de lignes de cache sont imprévisibles $\rightarrow$ incohérence séquentielle
  - Pour garantir la cohérence, le matériel assure que tout cache fournit toujours la dernière version connue de chaque ligne
  - Les caches maintiennent un RW-lock sur chaque ligne grâce à un protocole de cohérence de cache $\rightarrow$ MESI, MOESI

- Il existe plusieurs niveaux de cache $\rightarrow$ L1, L2, ..., LLC
  - Si une ligne est absente du niveau LX, elle est recherchée dans le niveau $L(X+1)$
  - Les différents niveaux de cache communiquent entre eux selon des stratégies d'inclusion

- Le logiciel peut contrôler finement le cache
  - Obtenir de meilleurs performances
  - Guarantir que des données sont bien en mémoire principale

Conclusion : architectures *Non Uniform Memory Access*

- Quand le nombre de cœur augmente, la mémoire ne peut plus servir rapidement toutes les requêtes mémoire
  - Problème de débit d'accès à la mémoire principale
  - Dans les architectures NUMA, la mémoire principale est fragmentée en unités indépendantes
  - Une unité indépendante et les cœurs associés s'appelle un nœud NUMA
  - Le matériel présente un espace d'adressage physique unifié au logiciel
- Les protocoles de cohérence de cache classiques génèrent beaucoup de traffic quand le nombre de cœur est grand
  - Les protocoles modernes sont basés sur des *cache directories*
  - Chaque ligne a un *home node* associé $\rightarrow$ le *home node* indique quel nœud contacter pour obtenir le verrou d'une ligne
- La performance d'un logiciel sur un matériel NUMA dépend du placement des tâches et des données
  - Linux utilise des stratégies d'allocation pour placer les données
  - *First touch* : allouer au même endroit que la tâche qui alloue
  - *Interleaving* : allouer au hasard $\rightarrow$ équilibrage de charge