NMV : Gestion de la mémoire virtuelle

Gauthier Voron - gauthier.voron@lip6.fr

Gestion de la mémoire : pourquoi

- L'accès à la mémoire est une opération extrèmement fréquente
  - La gestion mémoire a un impact important sur les performances
- Connaître son fonctionnement permet de concevoir
  - des systèmes efficaces $\rightarrow$ avec une bonne gestion mémoire
  - des applications efficaces $\rightarrow$ qui tirent parti de la gestion mémoire
- Exemples de mécanismes basés sur la mémoire virtuelle :
  - La migration *zero-copy*
  - La mémoire distribuée logicielle
  - La mémoire *Copy On Write*

Gestion de la mémoire : quoi

- Gestion de la mémoire : fonction d'un système d'attribuer des adresses ou plages d'adresses à d'autres parties du système.

- Cette attribution des adresses peut suivre certaines règles :
  - Une adresse ne doit pas être attribuée plus d'une fois
  - Une adresse doit pouvoir être remise à disposition par son détenteur
  - Les adresses ne peuvent être attribuées qu'en ordre croissant

- Dans un premier temps
  - Considérons : adresse = position dans la mémoire

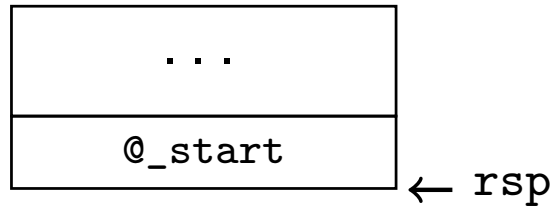
Plan du cours

1. Gestion de la mémoire en espace utilisateur
   - Gestion de la mémoire dynamique
   - Compilation et adressage
   - Isolation mémoire et collisions d'adresses
2. Mémoire virtuelle et *Memory Management Unit*
   - Fonction d'une *Memory Management Unit*
   - Traduction d'adresse et table des pages
   - Performance et *Translation Lookaside Buffer*
3. Mémoire virtuelle dans Linux
   - Traduction d'adresse et faute de page
   - Allocation paresseuse et *First Touch*
   - Appel système et adressage du noyau

Plan du cours

Gestion de la mémoire dynamique : la pile

```c
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

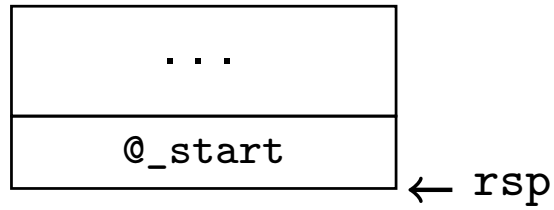



```
sub     $8, %rsp
push    $12
mov     0(%rsp), %rdi
call    f
add     $16, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```c
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

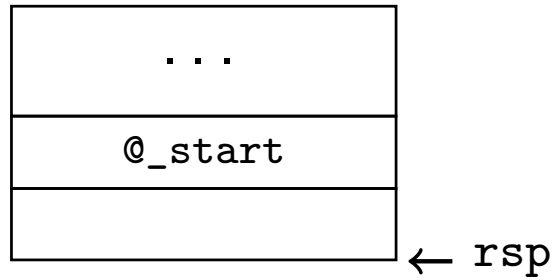



```
sub     $8, %rsp
push    $12
mov     0(%rsp), %rdi
call    f
add     $16, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```c
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

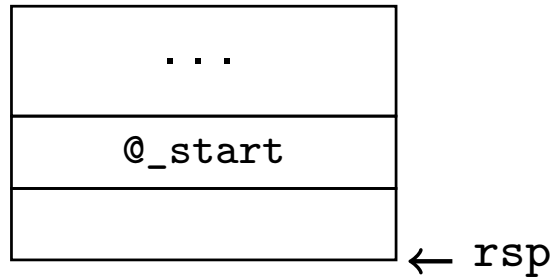



```
sub     $8, %rsp
push    $12
mov     0(%rsp), %rdi
call    f
add     $16, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```c
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

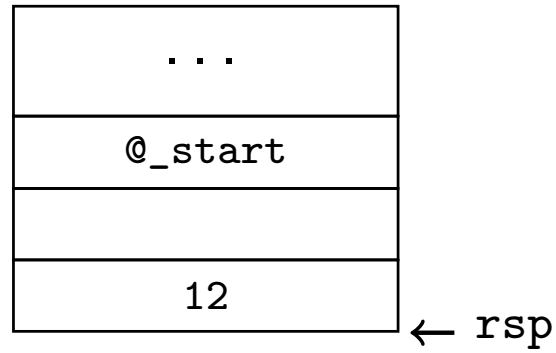



```
sub     $8, %rsp
push    $12
mov     0(%rsp), %rdi
call    f
add     $16, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```c
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

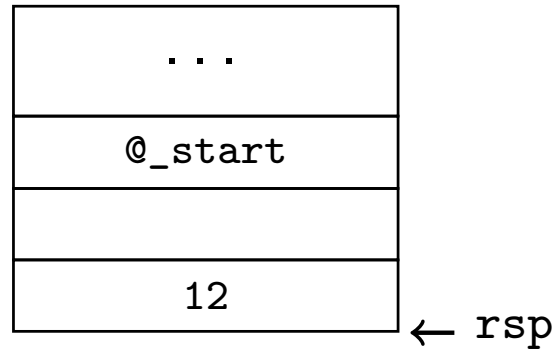



```
sub     $8, %rsp
push    $12
mov     0(%rsp), %rdi
call    f
add     $16, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```c
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

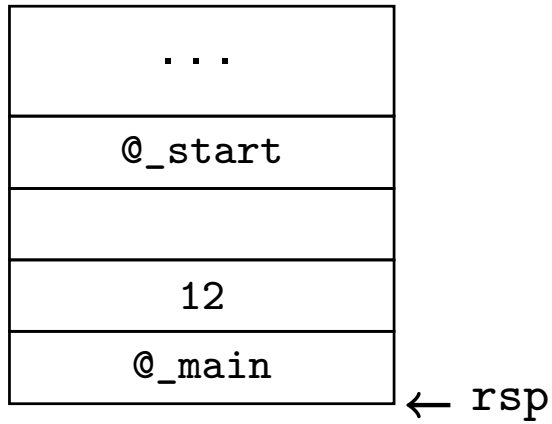



```
sub     $8, %rsp
push    $12
mov     0(%rsp), %rdi
call    f
add     $16, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```c
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

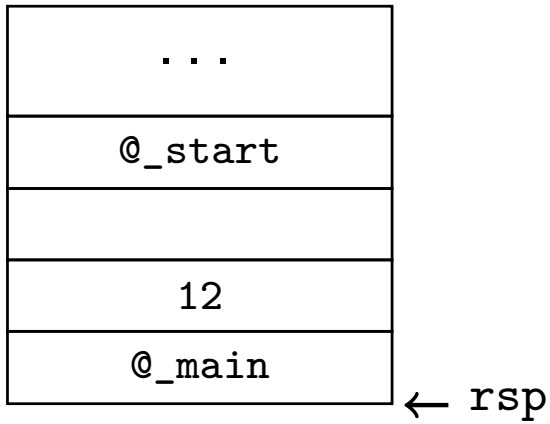



```
sub     $8, %rsp
push    $12
mov     0(%rsp), %rdi
call    f
add     $16, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```c
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

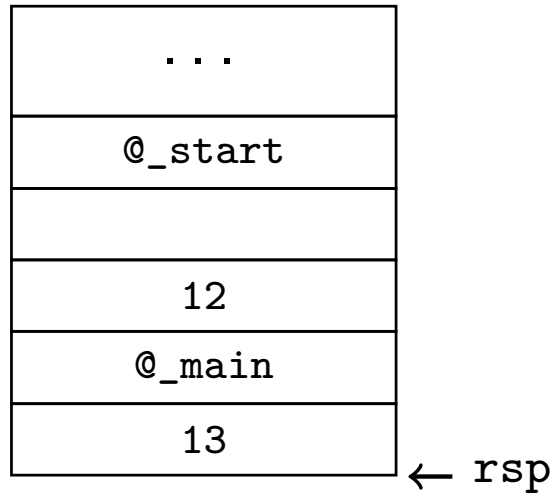



```
push    $13
mov     %rdi, %rax
add     0(%rsp), %rax
add     $8, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

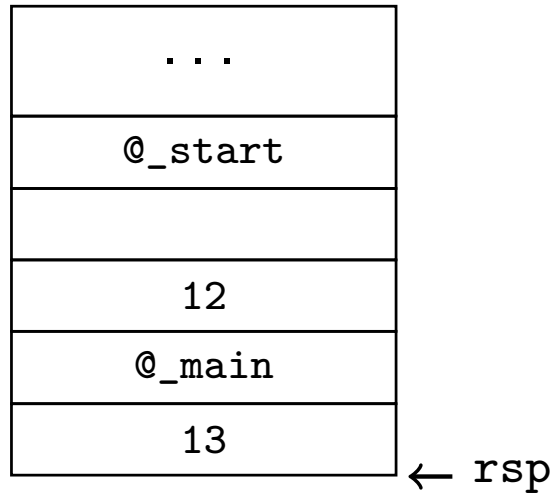



```
push    $13
mov     %rdi, %rax
add     0(%rsp), %rax
add     $8, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

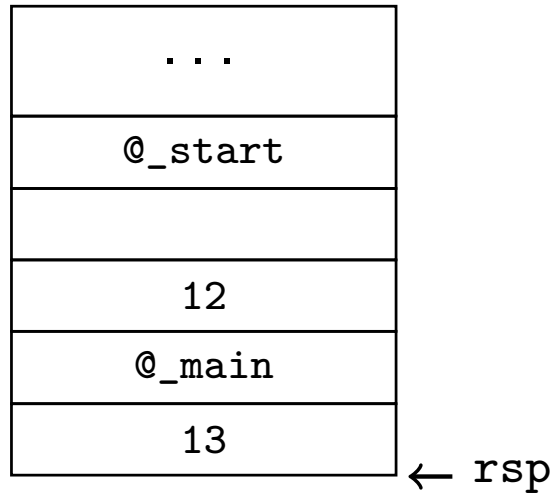



```
push    $13
mov     %rdi, %rax
add     0(%rsp), %rax
add     $8, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

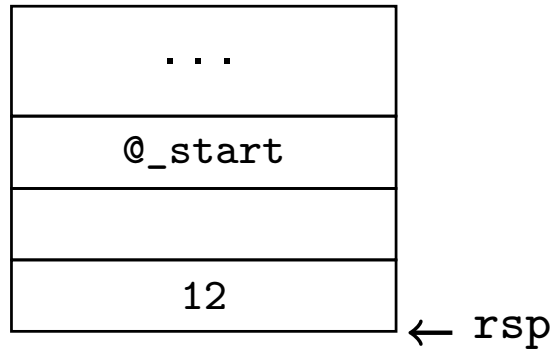



```
push    $13
mov     %rdi, %rax
add     0(%rsp), %rax
add     $8, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```c
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

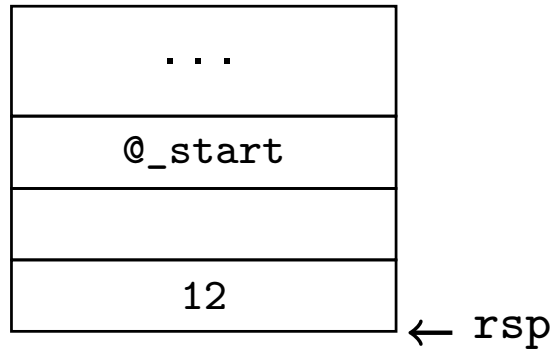



```
push    $13
mov     %rdi, %rax
add     0(%rsp), %rax
add     $8, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```c
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

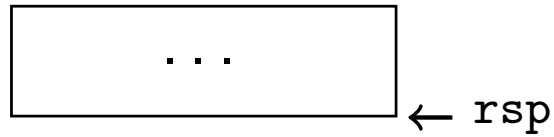



```
sub     $8, %rsp
push    $12
mov     0(%rsp), %rdi
call    f
add     $16, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```c
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

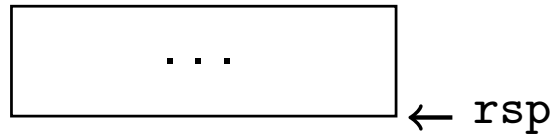



```
sub     $8, %rsp
push    $12
mov     0(%rsp), %rdi
call    f
add     $16, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : la pile

```c
int f(long x)
{
        long y = 13;
        return x + y;
}

int main(void)
{
        long a, b = 12;

        a = f(b);

        return 0;
}
```

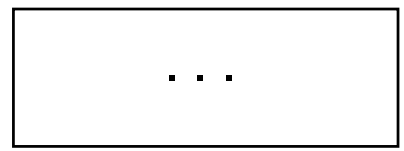



```
sub     $8, %rsp
push    $12
mov     0(%rsp), %rdi
call    f
add     $16, %rsp
ret
```

- Sommet de la pile stocké dans un registre dédié
- Gestion par les prologue / épilogue des fonctions
- Générés à la compilation

Gestion de la mémoire dynamique : le tas

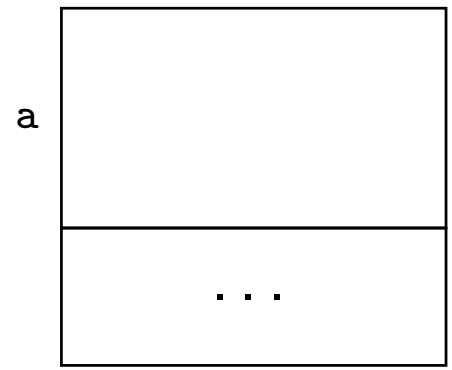
```c
int main(void)
{
        void *a, *b, *c;

        a = malloc(20);
        b = malloc(12);

        free(a);

        c = malloc(18);

        return 0;
}
```



- Ensemble d'objets enregistrés dans des structures logicielles
- Gestion par une librairie dédiée (`glibc` par défault)
- Modifiable au lancement du programme (`LD_PRELOAD`)

Gestion de la mémoire dynamique : le tas

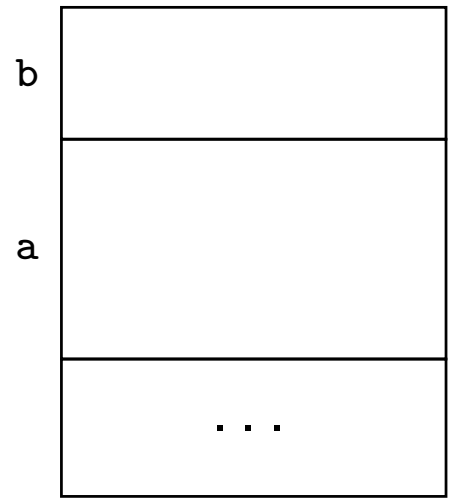
```c
int main(void)
{
        void *a, *b, *c;

        a = malloc(20);
        b = malloc(12);

        free(a);

        c = malloc(18);

        return 0;
}
```



- Ensemble d'objets enregistrés dans des structures logicielles
- Gestion par une librairie dédiée (`glibc` par défault)
- Modifiable au lancement du programme (`LD_PRELOAD`)

Gestion de la mémoire dynamique : le tas

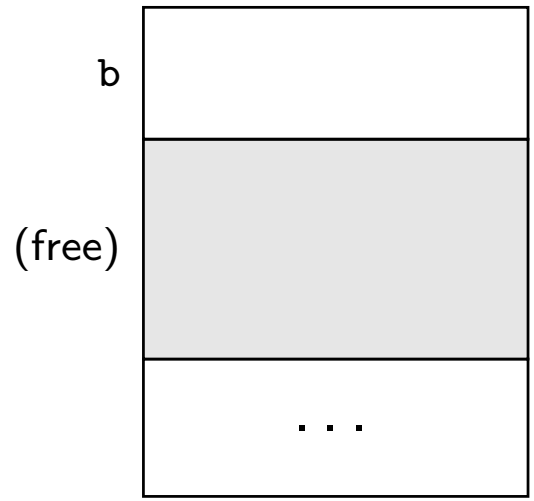
```c
int main(void)
{
        void *a, *b, *c;

        a = malloc(20);
        b = malloc(12);

        free(a);

        c = malloc(18);

        return 0;
}
```



- Ensemble d'objets enregistrés dans des structures logicielles
- Gestion par une librairie dédiée (`glibc` par défault)
- Modifiable au lancement du programme (`LD_PRELOAD`)

Gestion de la mémoire dynamique : le tas

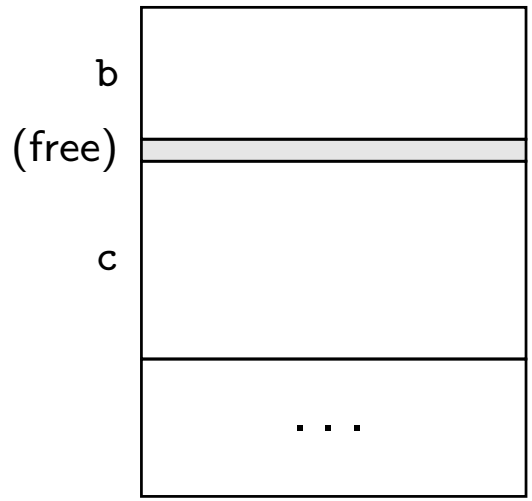
```c
int main(void)
{
        void *a, *b, *c;

        a = malloc(20);
        b = malloc(12);

        free(a);

        c = malloc(18);

        return 0;
}
```



- Ensemble d'objets enregistrés dans des structures logicielles
- Gestion par une librairie dédiée (`glibc` par défault)
- Modifiable au lancement du programme (`LD_PRELOAD`)

Gestion de la mémoire dynamique : le tas

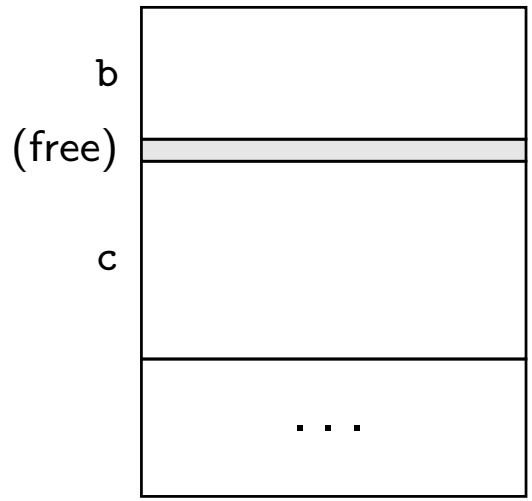
```c
int main(void)
{
        void *a, *b, *c;

        a = malloc(20);
        b = malloc(12);

        free(a);

        c = malloc(18);

        return 0;
}
```



- Ensemble d'objets enregistrés dans des structures logicielles
- Gestion par une librairie dédiée (`glibc` par défault)
- Modifiable au lancement du programme (`LD_PRELOAD`)

Gestion de la mémoire dynamique : le tas

```c
int main(void)
{
        void *a, *b, *c;

        a = malloc(20);
        b = malloc(12);

        free(a);

        c = malloc(18);

        return 0;
}
```



- Ensemble d'objets enregistrés dans des structures logicielles
- Gestion par une librairie dédiée (`glibc` par défault)
- Modifiable au lancement du programme (`LD_PRELOAD`)

Gestion de la mémoire dynamique : le tas

```
int main(void)
{
        void *a, *b, *c;
}
```

- La fonction `malloc` **n'est pas un appel système**
  - Il s'agit d'une fonction userland
  - Alloue de grosses plages d'adresses grâce à l'appel système `mmap`
  - Retourne de petites portions de plages d'adresses
  - La fragmentation / réutilisation des plages d'adresses se fait en espace utilisateur

- Ensemble d'objets enregistrés dans des structures logicielles
- Gestion par une librairie dédiée (`glibc` par défault)
- Modifiable au lancement du programme (`LD_PRELOAD`)

Gestion de la mémoire dynamique : résumé

- Attribution d'adresses dans un *pool* d'adresses données
  - Géré par les prologue / épilogue des fonctions (pile)
  - Géré par des librairies spécialisées (tas)
  - Adresses attribuées à la demande du programme

- Ces *pools* d'adresse sont fournis par le système
  - À l'initialisation du processus (`%rsp = xxx`)
  - Par un appel système (`mmap`)

- D'où viennent les adresses statiques ?
  - Adresses de fonction
  - Variable globales
  - Chaînes de caractère statiques

Compilation et adressage : fichier objet

```c
char *var = "toto";

long n = 0;

int main(void)
{
  func(var[0], n);
  ...
}
```

```
_toto:   "toto\0"
var:     <_toto>
n:       0

func:
  ...

main:
  mov      <var>, %rdi
  mov      0(%rdi), %rdi
  mov      <n>, %rsi
  call     <func>
```

- Le compilateur transforme le code en symboles (`main`, `var`, etc...)
- Les symboles se référencent entre eux
- Les adresses ne sont pas encore attribuées

Compilation et adressage : fichier exécutable

```
_toto:   "toto\0"                  # at 0x400000
var:     0x400000                  # at 0x500000
n:       0                         # at 0x600000

func:                              # at 0x700000
   ...

main:                              # at 0x800000
   mov      (0x500000), %rdi
   mov      (%rdi), %rdi
   mov      (0x600000), %rsi
   call     -0x100060               # at 0x800060
```

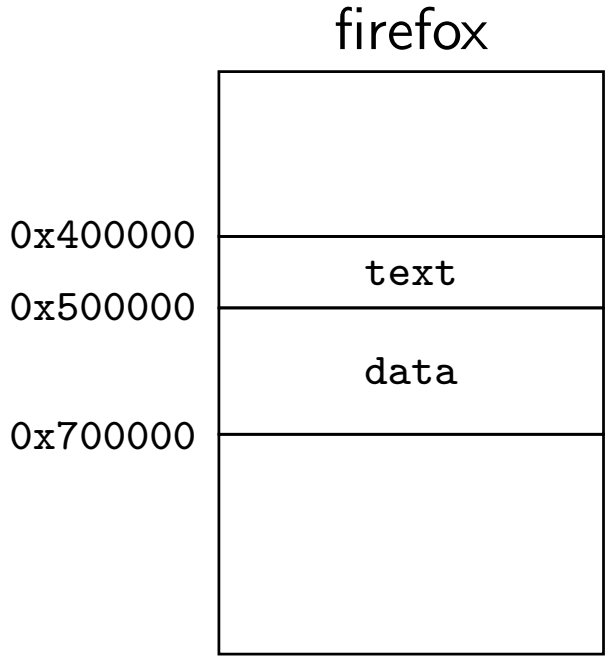
- Le compilateur attribue une adresse à tous les symboles
  - via un script d'édition de lien (`info ld`)
  - via des options dédiées (`gcc -Wl`)
  - via des valeurs prédéfinies
- Les symboles sont remplacés par leurs adresses dans le code
- Les adresses peuvent être absolues ou relatives

Compilation et adressage : résumé

- Attributions des adresses statiques à l'édition des liens
  - La décision revient au programmeur
  - Les valeurs par défault suffisent la plupart du temps
  - Fixé pour un exécutable donné
- Adressage absolu $\rightarrow$ le code doit être chargé à une adresse précise
- Adressage relatif $\rightarrow$ le code doit être chargé dans le bon ordre

Collisions d'adresses

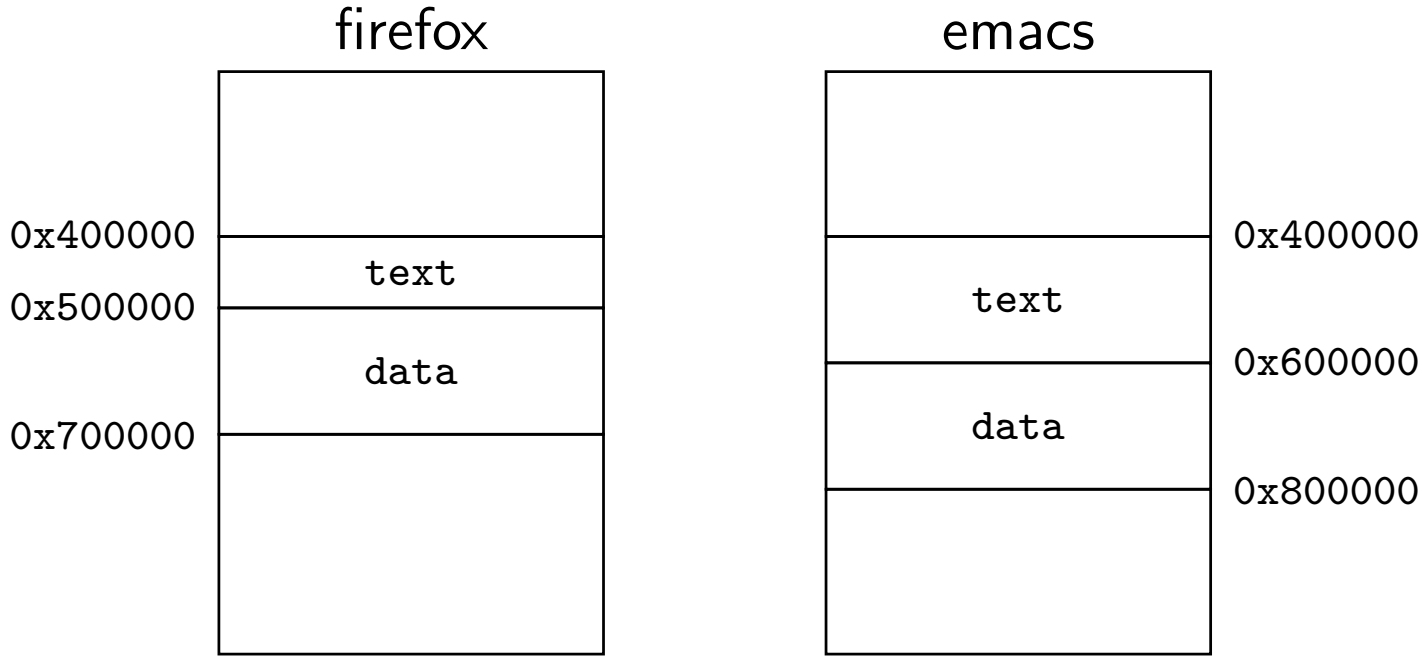
- Rappel : on considère que adresse = position dans la mémoire



- Un programme doit être chargé à une adresse précise
- Cette adresse ne peut être modifiée que par une compilation

Collisions d'adresses

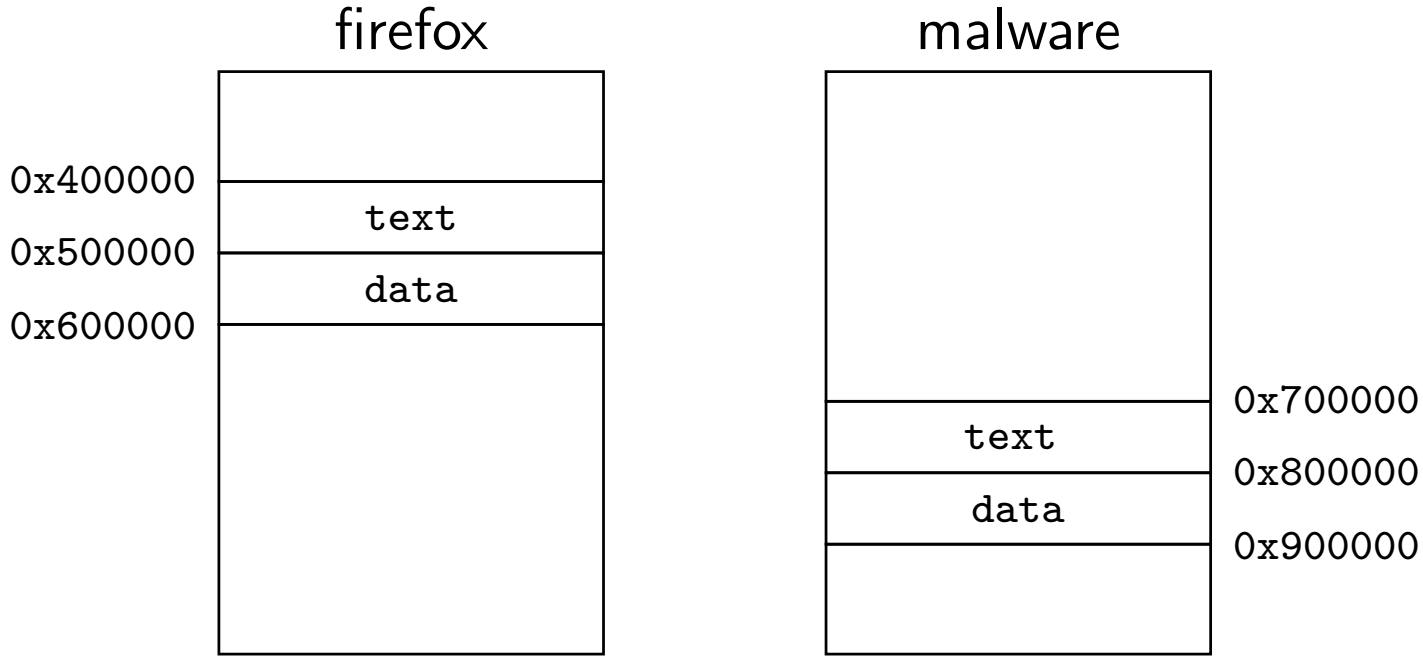
- Rappel : on considère que adresse = position dans la mémoire



- Un programme doit être chargé à une adresse précise
- Cette adresse ne peut être modifiée que par une compilation
- Exécuter deux programmes compilés pour les mêmes adresses ?

Isolation mémoire

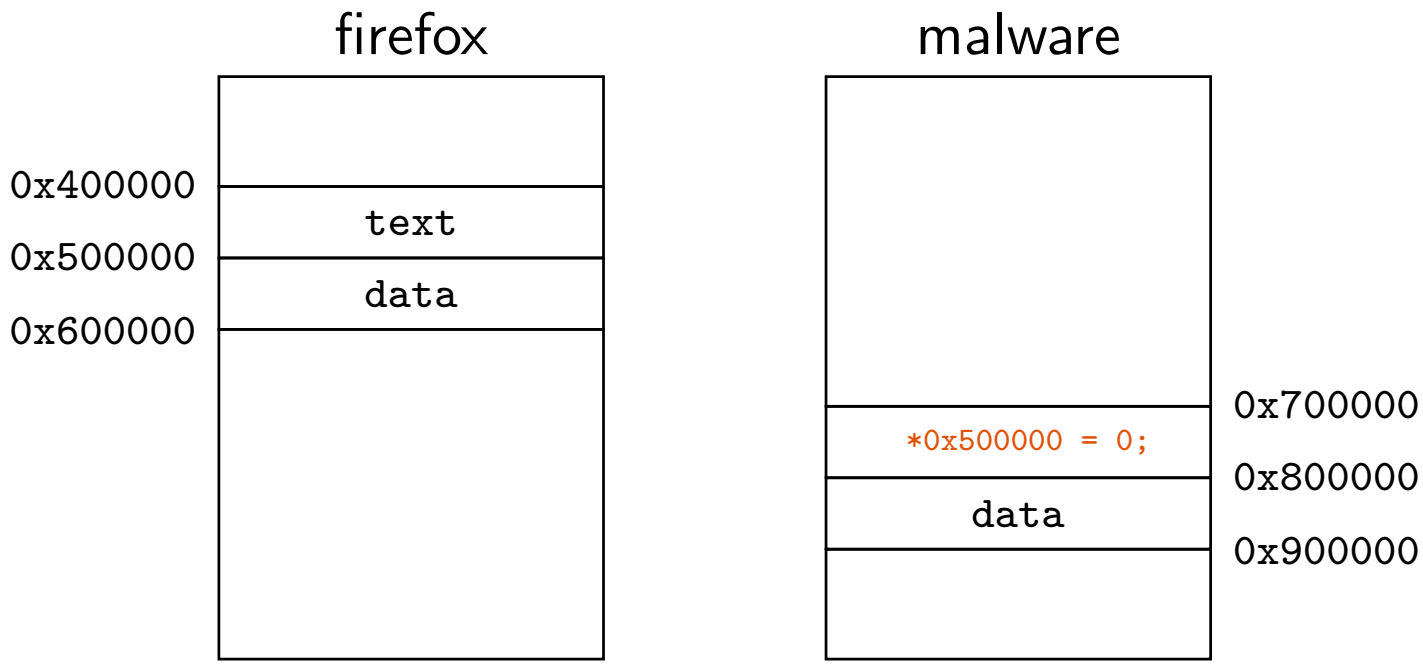
- Rappel : on considère que adresse = position dans la mémoire



- Deux programmes sont chargés à des adresses différentes
- Ils peuvent s'exécuter en même temps

Isolation mémoire

- Rappel : on considère que adresse = position dans la mémoire



- Deux programmes sont chargés à des adresses différentes
- Ils peuvent s'exécuter en même temps
- Comment protéger contre des accès illégaux ?

Gestion de la mémoire en espace utilisateur : résumé

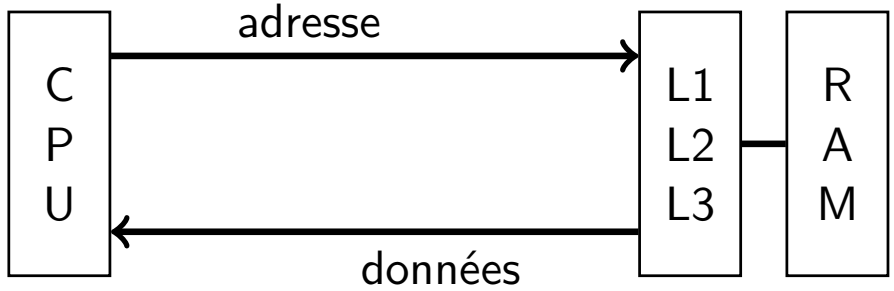
- Gérer la mémoire se résume à attribuer des adresses
- Gestion dynamique (pile/tas)
  - Gestion d'un *pool* d'adresses
  - Le *pool* est alloué par le noyau
- Gestion statique (code/données)
  - Adresses fixées à la compilation
  - Comment gérer les collisions à l'exécution ?
- Comment isoler les adresses des différents processus ?

Gestion de la mémoire en espace utilisateur : résumé

- Gérer la mémoire se résume à attribuer des adresses
- Gestion dynamique (pile/tas)
  - Gestion d'un *pool* d'adresses
  - Le *pool* est alloué par le noyau
- Gestion statique (code/données)
  - Adresses fixées à la compilation
  - Comment gérer les collisions à l'exécution ?
- Comment isoler les adresses des différents processus ?

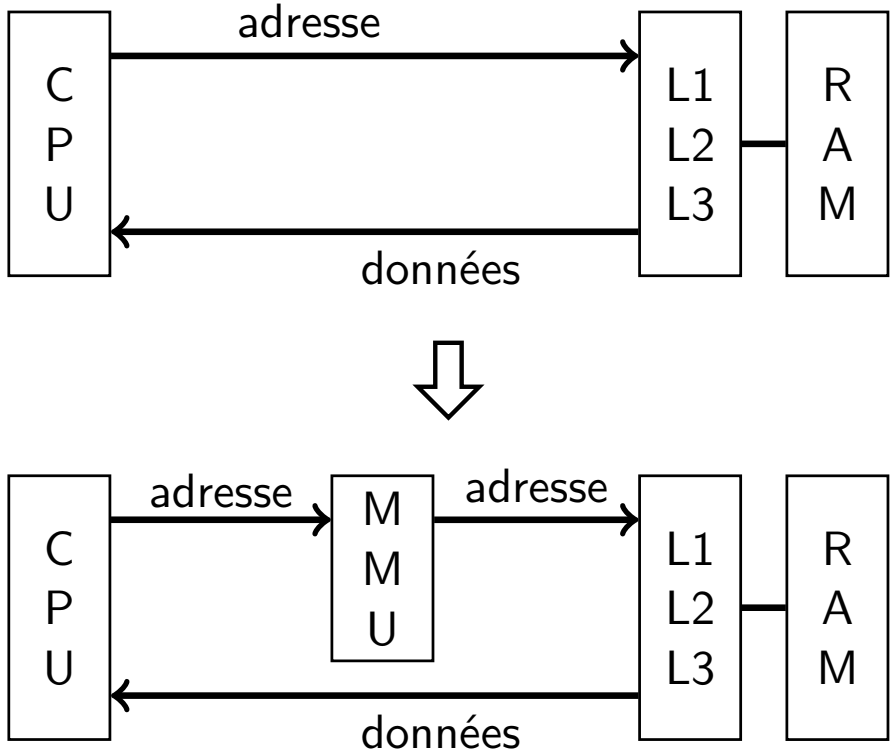
Plan du cours

Fonction d'une *Memory Management Unit*



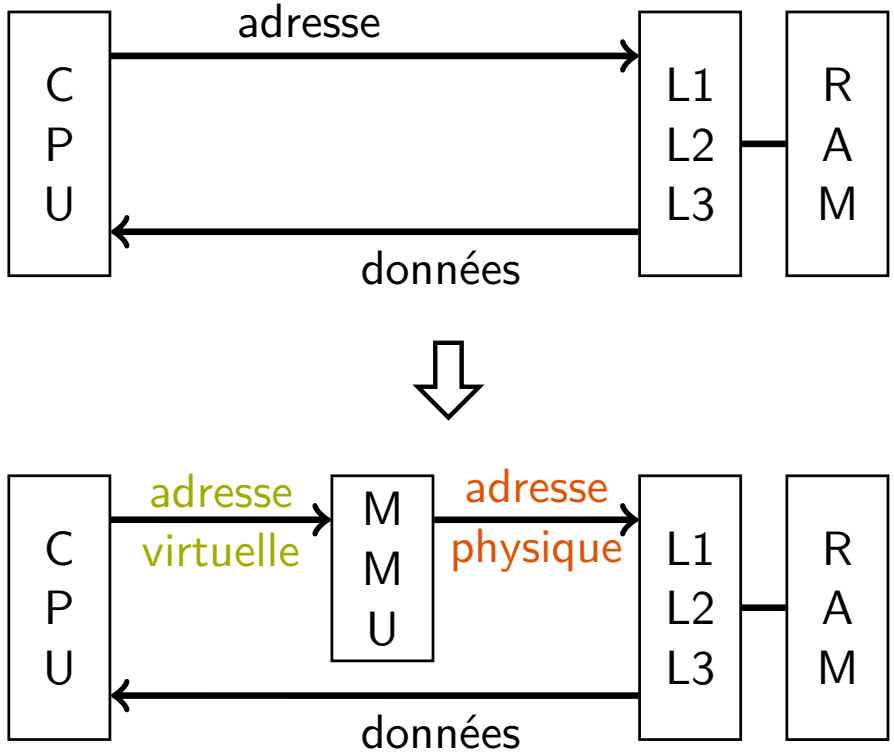
- Le CPU envoie des requêtes au système mémoire
  - Chaque requête contient l'adresse de la donnée à lire / écrire

Fonction d'une *Memory Management Unit*



- Le CPU envoie des requêtes au système mémoire
  - Chaque requête contient l'adresse de la donnée à lire / écrire
- La MMU traduit les adresses émises par le CPU

Fonction d'une *Memory Management Unit*

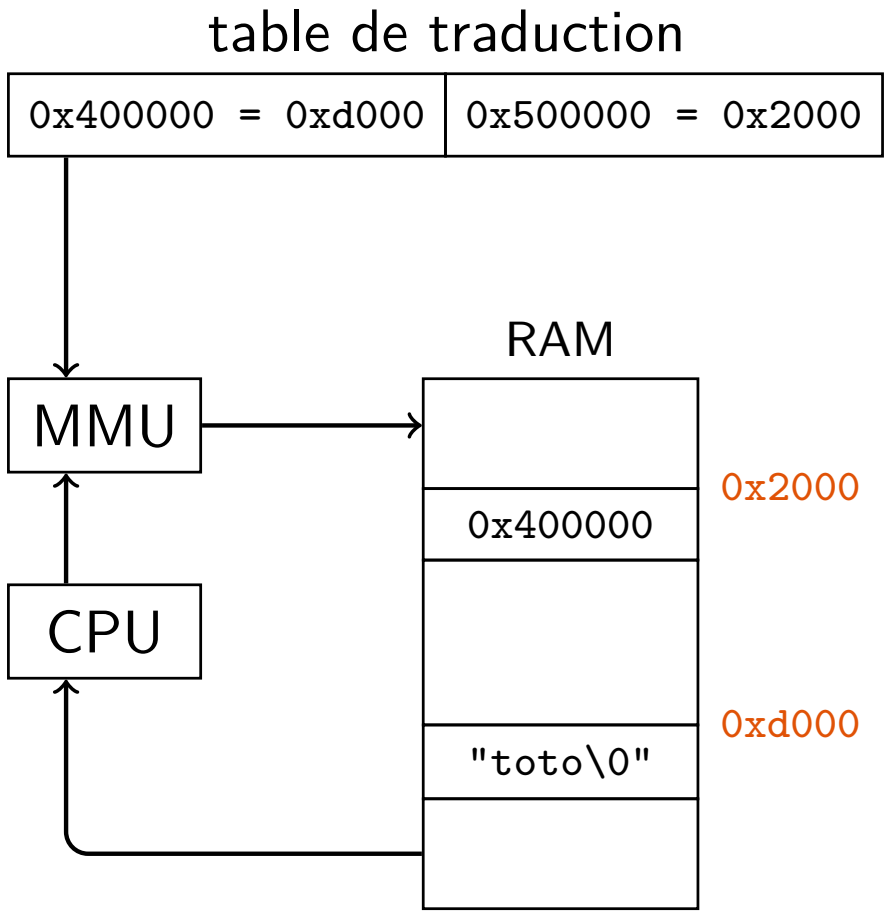


- Le CPU envoie des requêtes au système mémoire
  - Chaque requête contient l'adresse de la donnée à lire / écrire
- La MMU traduit les adresses émises par le CPU
  - Les adresses côté CPU s'appellent adresses virtuelles
  - Les adresses côté mémoire s'appellent adresses physiques

Une *Memory Management Unit* en action

```
_toto:   "toto\0"    # at 0x400000
var:     0x400000    # at 0x500000

main:
   mov       (0x500000), %rdi
   mov       (%rdi), %rdi
```

instruction	
adresse	
%rdi	

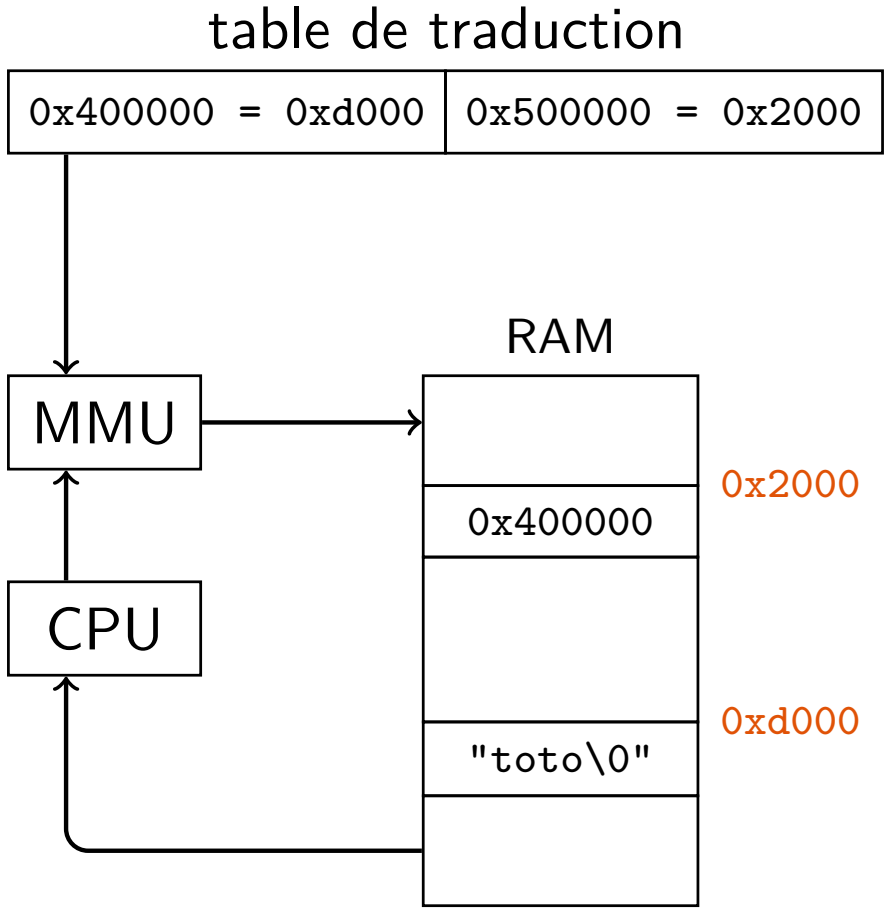


- La MMU utilise une table de traduction

Une *Memory Management Unit* en action

```
_toto:   "toto\0"    # at 0x400000
var:     0x400000    # at 0x500000

main:
   mov       (0x500000), %rdi
   mov       (%rdi), %rdi
```

instruction	mov (0x500000), %rdi
adresse	
%rdi	

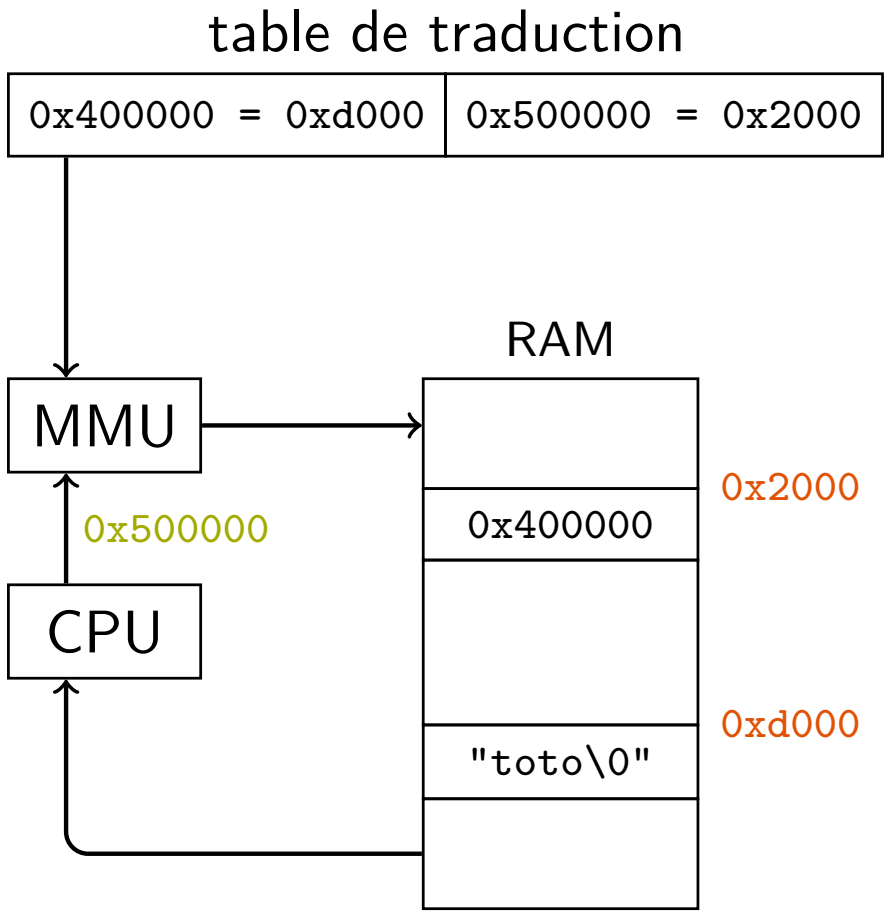


- La MMU utilise une table de traduction

Une *Memory Management Unit* en action

```
_toto:   "toto\0"    # at 0x400000
var:     0x400000    # at 0x500000

main:
   mov       (0x500000), %rdi
   mov       (%rdi), %rdi
```

instruction	mov (0x500000), %rdi
adresse	0x500000
%rdi	

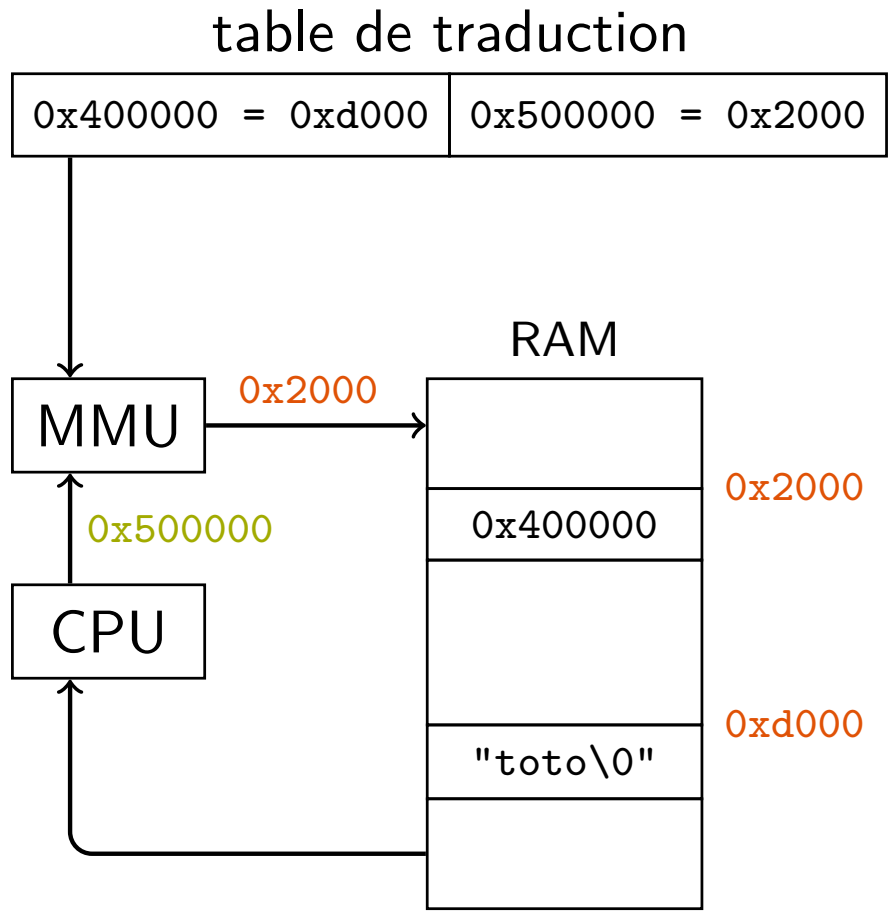


- La MMU utilise une table de traduction

Une *Memory Management Unit* en action

```
_toto:   "toto\0"    # at 0x400000
var:     0x400000    # at 0x500000

main:
   mov      (0x500000), %rdi
   mov      (%rdi), %rdi
```

instruction	mov (0x500000), %rdi
adresse	0x500000
%rdi	

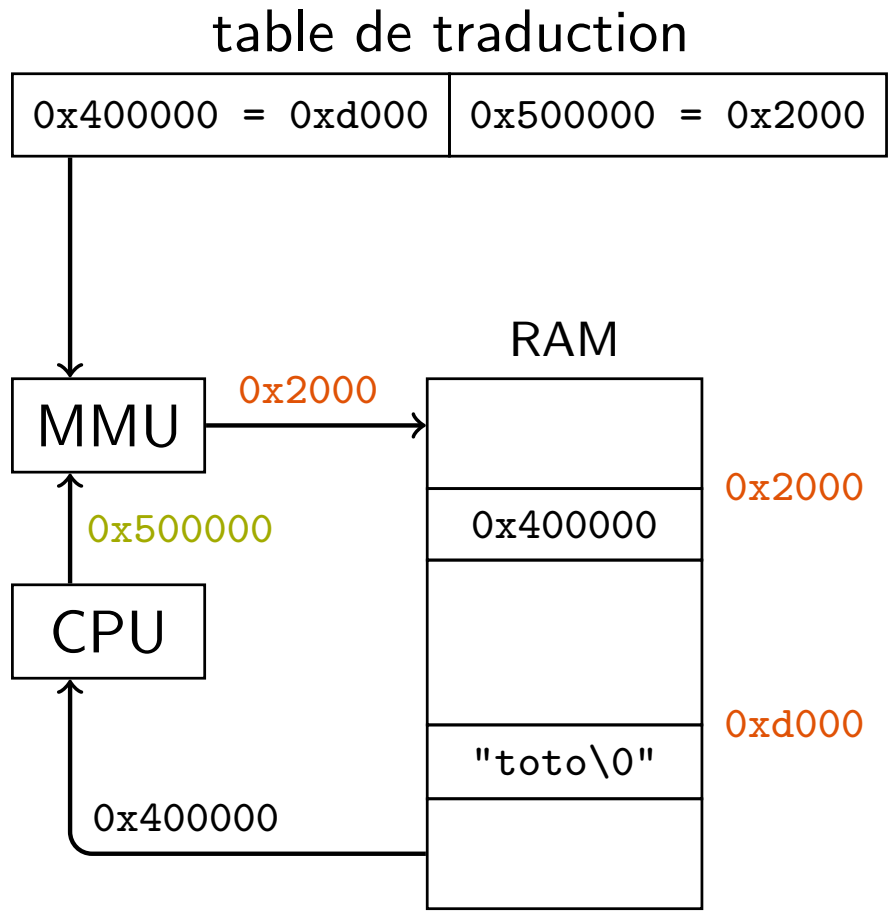


- La MMU utilise une table de traduction

Une *Memory Management Unit* en action

```
_toto:   "toto\0"    # at 0x400000
var:     0x400000    # at 0x500000

main:
   mov      (0x500000), %rdi
   mov      (%rdi), %rdi
```

instruction	mov (0x500000), %rdi
adresse	0x500000
%rdi	0x400000

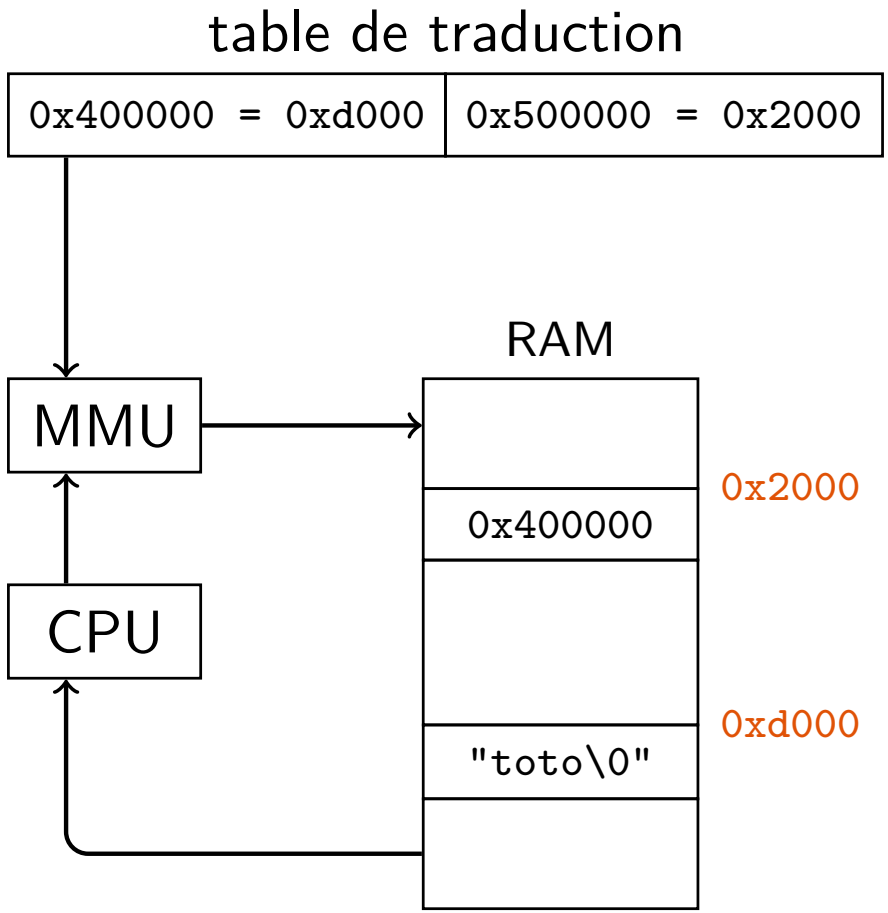


- La MMU utilise une table de traduction
- Les adresses virtuelles d'une requête sont transformées
- Les données ne sont pas altérées

Une *Memory Management Unit* en action

```
_toto:   "toto\0"    # at 0x400000
var:     0x400000    # at 0x500000

main:
   mov       (0x500000), %rdi
   mov       (%rdi), %rdi
```

instruction	mov (%rdi), %rdi
adresse	
%rdi	0x400000

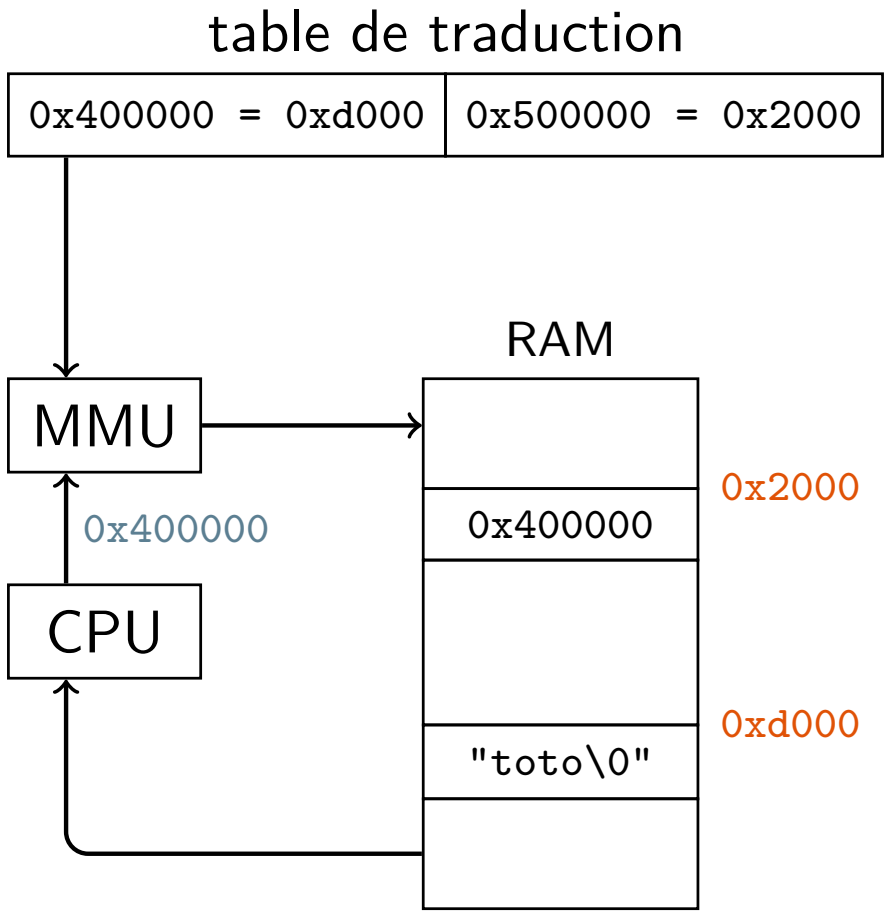


- La MMU utilise une table de traduction
- Les adresses virtuelles d'une requête sont transformées
- Les données ne sont pas altérées

Une *Memory Management Unit* en action

```
_toto:   "toto\0"    # at 0x400000
var:     0x400000    # at 0x500000

main:
   mov       (0x500000), %rdi
   mov       (%rdi), %rdi
```

instruction	mov (%rdi), %rdi
adresse	0x400000
%rdi	0x400000

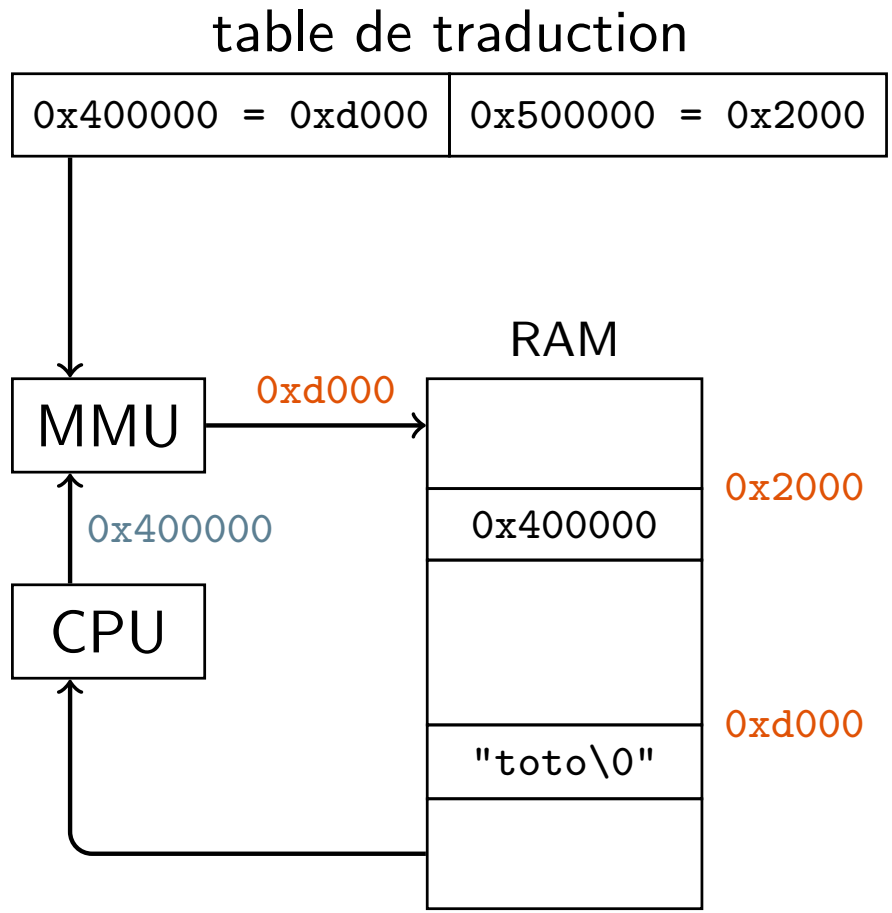


- La MMU utilise une table de traduction
- Les adresses virtuelles d'une requête sont transformées
- Les données ne sont pas altérées

Une *Memory Management Unit* en action

```
_toto:   "toto\0"    # at 0x400000
var:     0x400000    # at 0x500000

main:
   mov      (0x500000), %rdi
   mov      (%rdi), %rdi
```

instruction	mov (%rdi), %rdi
adresse	0x400000
%rdi	0x400000

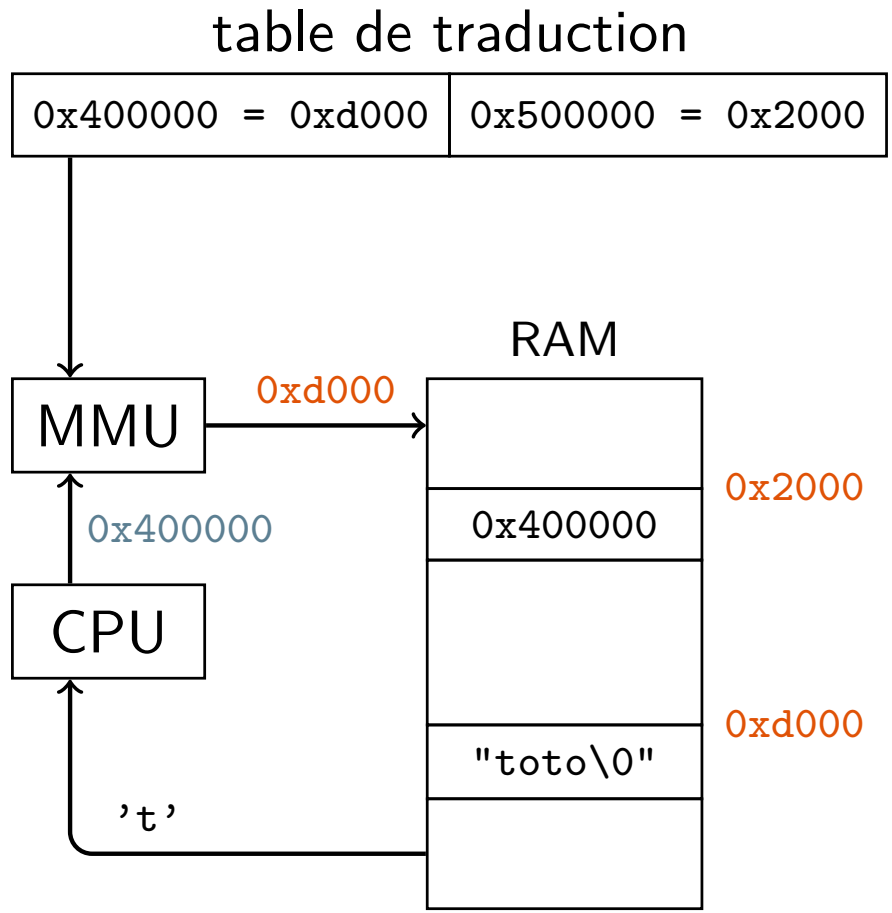


- La MMU utilise une table de traduction
- Les adresses virtuelles d'une requête sont transformées
- Les données ne sont pas altérées

Une *Memory Management Unit* en action

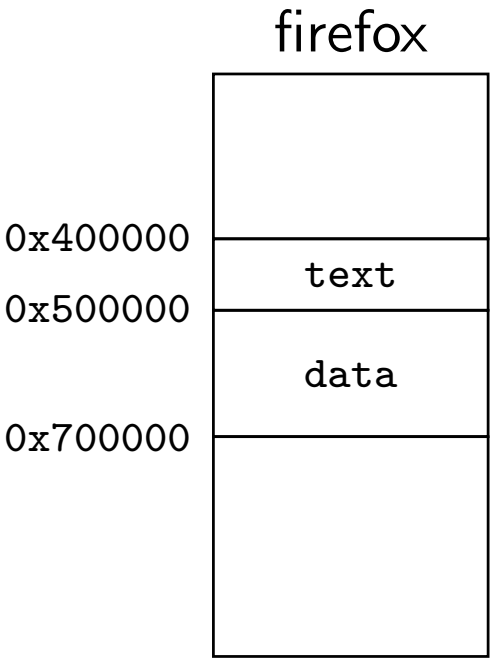
```
_toto:   "toto\0"    # at 0x400000
var:     0x400000    # at 0x500000

main:
   mov      (0x500000), %rdi
   mov      (%rdi), %rdi
```

instruction	mov (%rdi), %rdi
adresse	0x400000
%rdi	't'



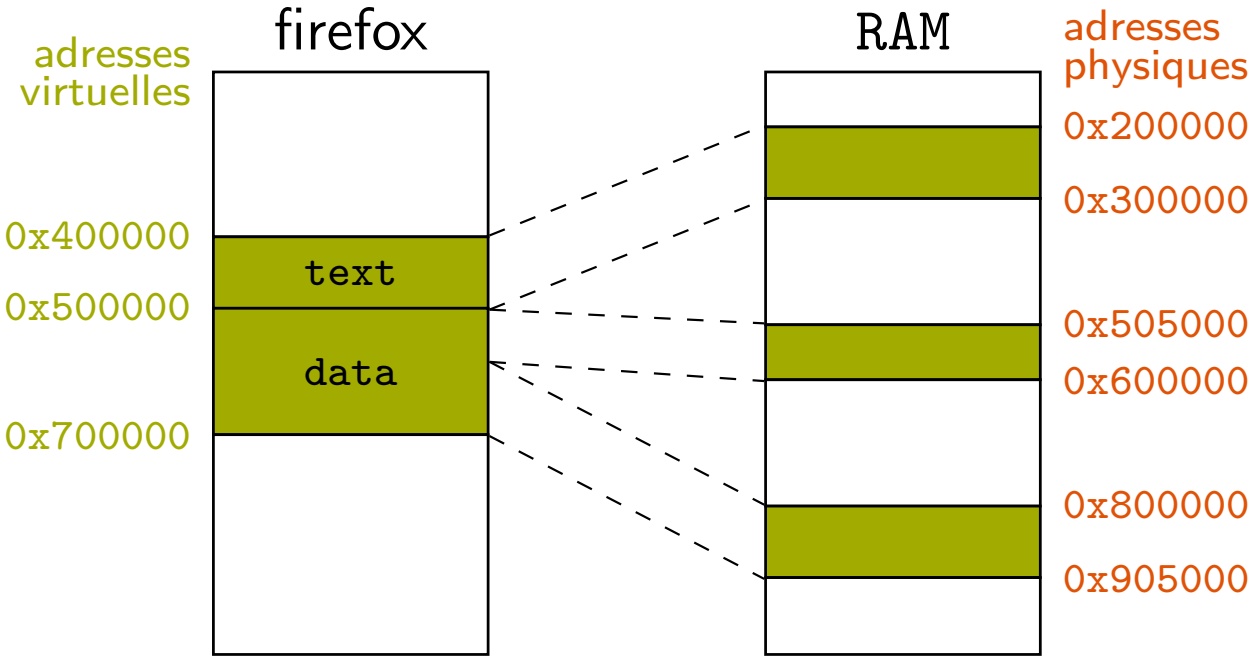
- La MMU utilise une table de traduction
- Les adresses virtuelles d'une requête sont transformées
- Les données ne sont pas altérées

Memory Management Unit et collisions d'adresses



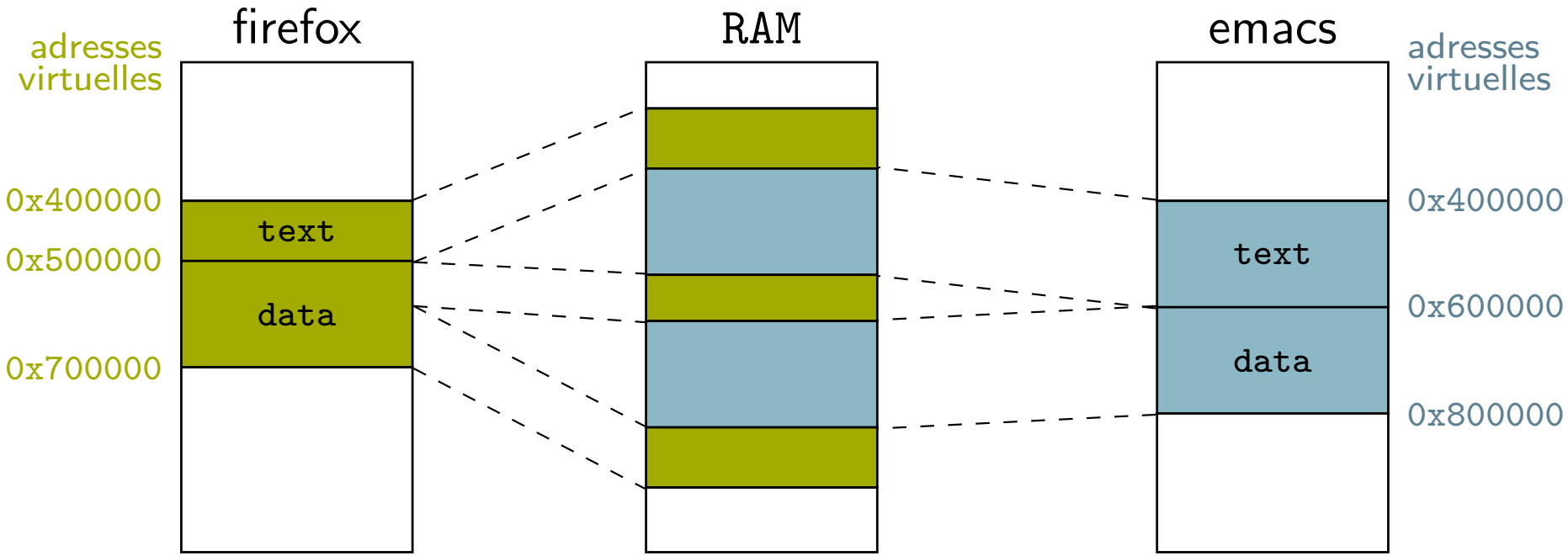
- Un programme doit être chargé à une adresse précise

Memory Management Unit et collisions d'adresses



- Un programme doit être chargé à une adresse virtuelle précise
- La MMU définie le placement en mémoire physique
  - Les adresses physiques sont choisies à l'exécution

Memory Management Unit et collisions d'adresses



- Un programme doit être chargé à une adresse virtuelle précise
- La MMU définie le placement en mémoire physique
  - Les adresses physiques sont choisies à l'exécution
- La MMU peut-être reconfigurée à la commutation
  - Les adresses virtuelles d'un autre processus sont traduites différemment

Fonction d'une *Memory Management Unit* : résumé

- La *Memory Management Unit* est un composant du processeur
- Elle intercepte les requêtes du CPU vers le système mémoire
  - Elle transforme les adresses virtuelles en adresses physiques
  - Elle ne modifie pas les données transmises au CPU
- Cette traduction permet de modifier l'emplacement en mémoire des données
  - Sans devoir recompiler le programme
  - Sans devoir en informer le processus

Traduction d'adresse et table des pages : définitions

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

Traduction d'adresse et table des pages : définitions

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

- Une adresse peut être décomposée en deux parties : l'index, qui indique l'emplacement de page, et l'offset qui indique une position dans la page.

  Exemple : dans l'adresse `0x54bd18ef`, la valeur `0x54bd1` est l'index et `0x8ef` est l'offset.

Traduction d'adresse et table des pages : définitions

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

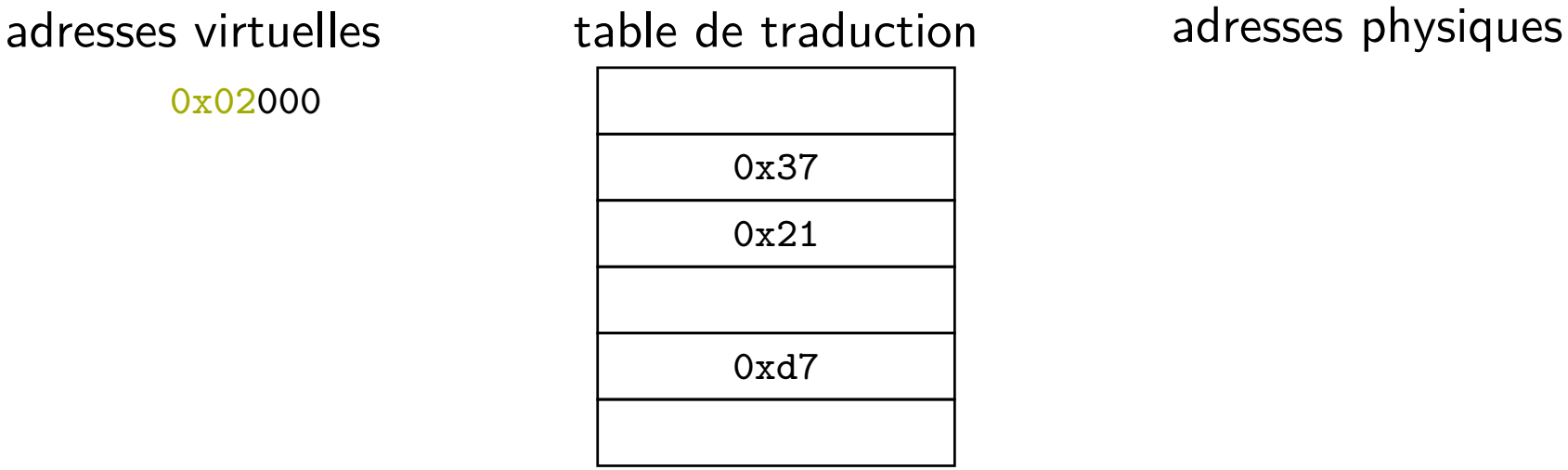
- Une adresse peut être décomposée en deux parties : l'index, qui indique l'emplacement de page, et l'offset qui indique une position dans la page.

  Exemple : dans l'adresse `0x54bd18ef`, la valeur `0x54bd1` est l'index et `0x8ef` est l'offset.

- Les tailles de l'index et de l'offset dépendent de la taille d'une page et de la taille maximum de la mémoire.

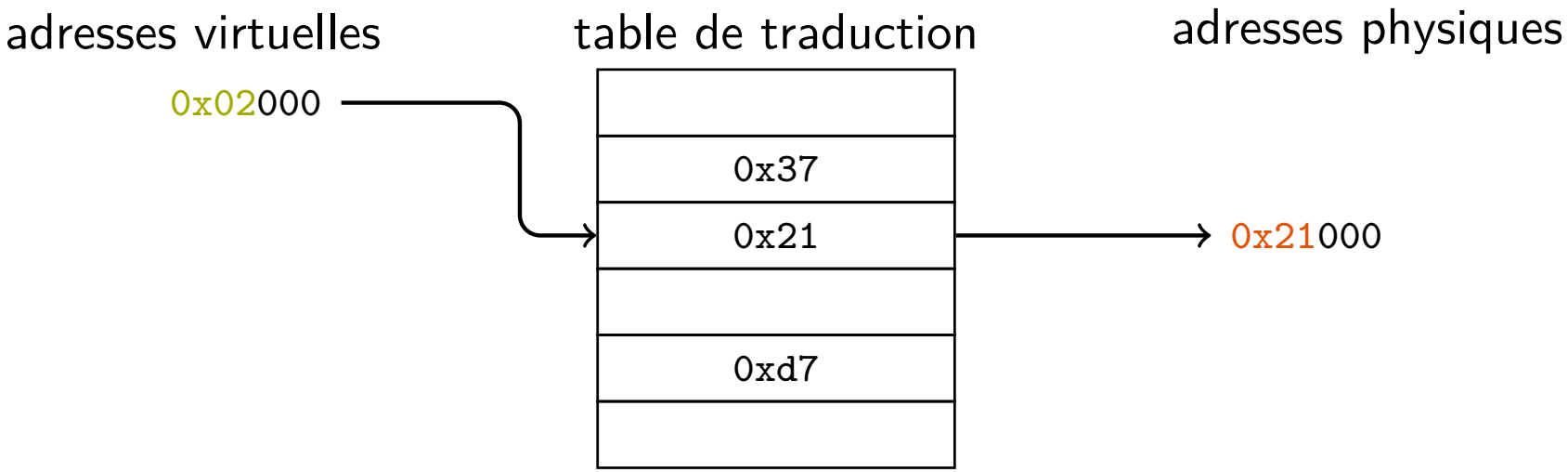
Traduction d'adresse et table des pages : aperçu

- Les traductions de la MMU dépendent de la table de traduction
  - Est une structure de données stockée en RAM
  - Est modifiée par le système d'exploitation
  - Est lue par la MMU
- Au cours d'une traduction d'adresse, seul l'index est modifié.
  - Seul l'emplacement de page est tranformé
  - On parle de table des pages
- On peut représenter la table des pages par un tableau



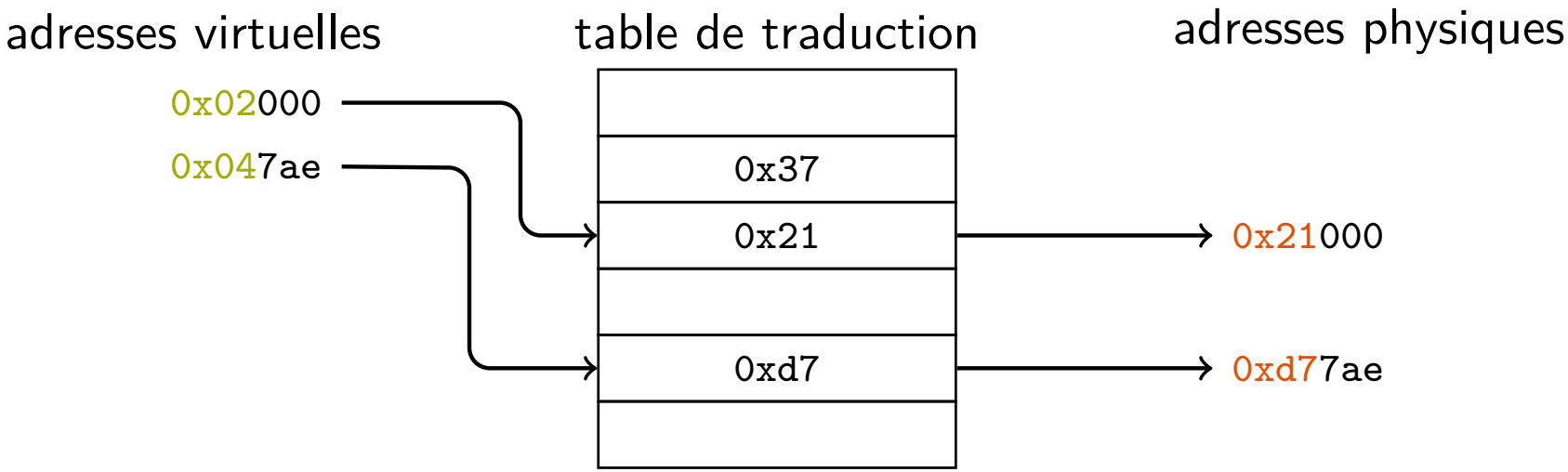
Traduction d'adresse et table des pages : aperçu

- Les traductions de la MMU dépendent de la table de traduction
  - Est une structure de données stockée en RAM
  - Est modifiée par le système d'exploitation
  - Est lue par la MMU
- Au cours d'une traduction d'adresse, seul l'index est modifié.
  - Seul l'emplacement de page est tranformé
  - On parle de table des pages
- On peut représenter la table des pages par un tableau



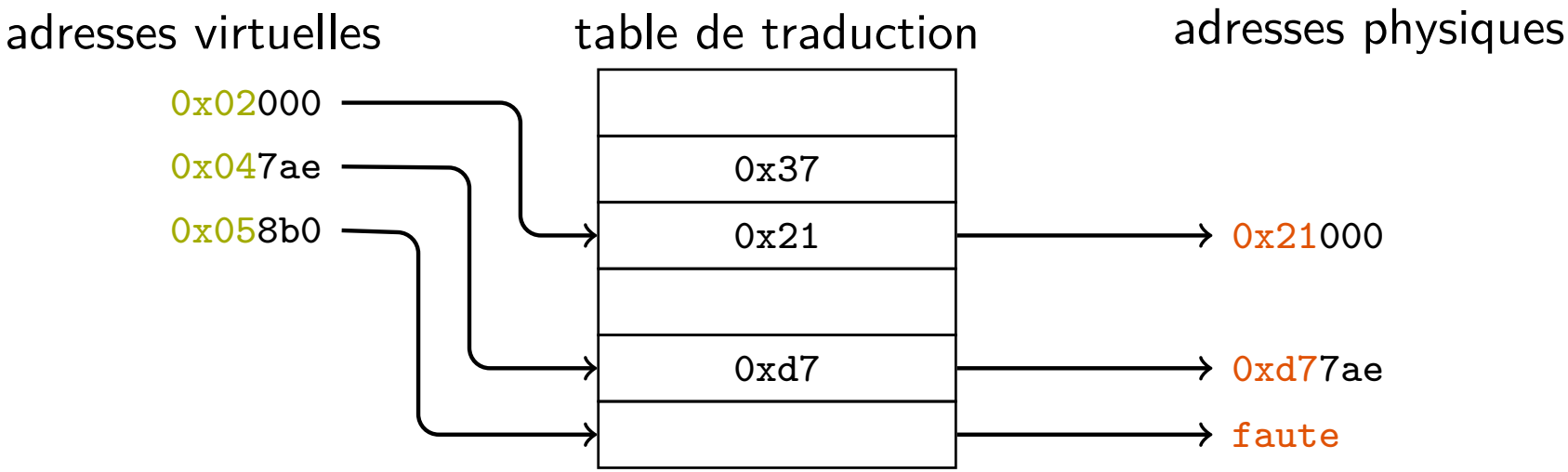
Traduction d'adresse et table des pages : aperçu

- Les traductions de la MMU dépendent de la table de traduction
  - Est une structure de données stockée en RAM
  - Est modifiée par le système d'exploitation
  - Est lue par la MMU
- Au cours d'une traduction d'adresse, seul l'index est modifié.
  - Seul l'emplacement de page est tranformé
  - On parle de table des pages
- On peut représenter la table des pages par un tableau



Traduction d'adresse et table des pages : aperçu

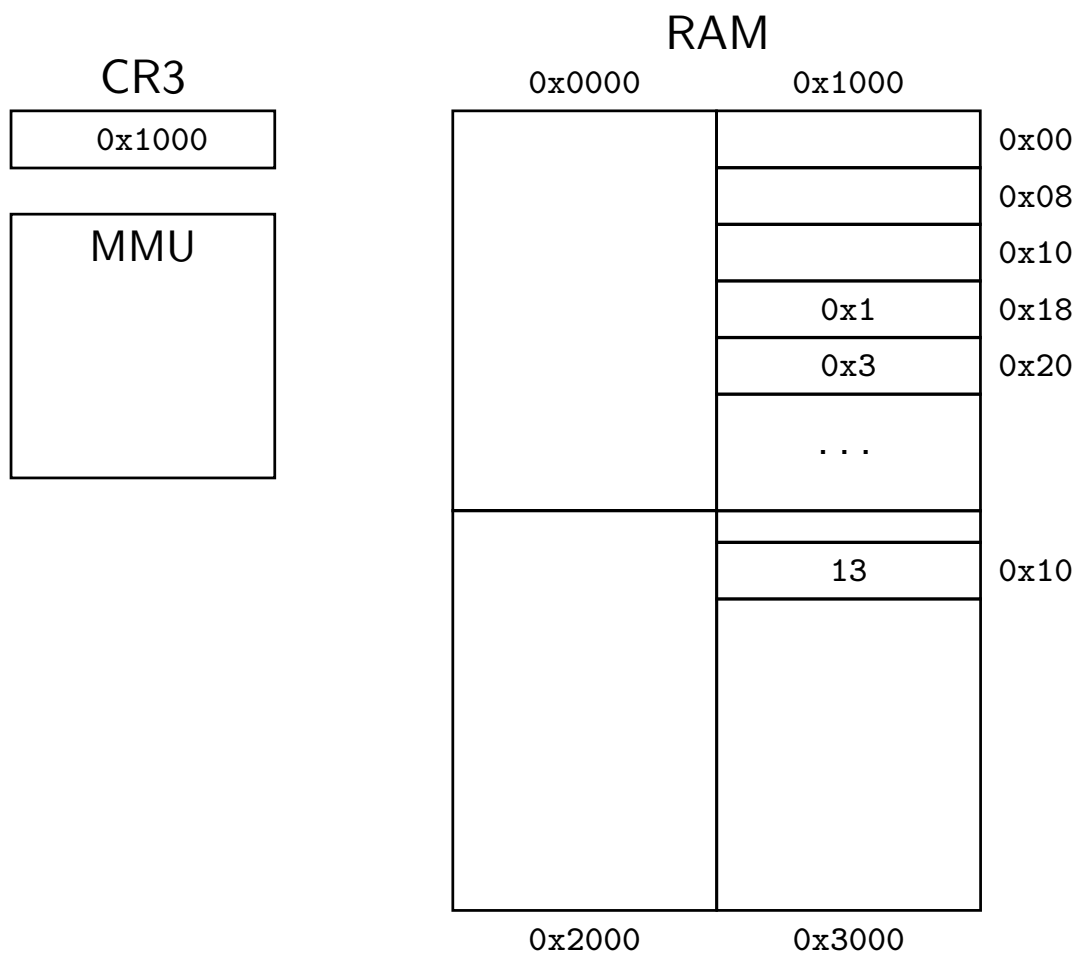
- Les traductions de la MMU dépendent de la table de traduction
  - Est une structure de données stockée en RAM
  - Est modifiée par le système d'exploitation
  - Est lue par la MMU
- Au cours d'une traduction d'adresse, seul l'index est modifié.
  - Seul l'emplacement de page est tranformé
  - On parle de table des pages
- On peut représenter la table des pages par un tableau



Traduction d'adresse et table des pages : exemple

- L'adresse physique de la table des pages est stockée dans un registre dédié du processeur : le CR3.

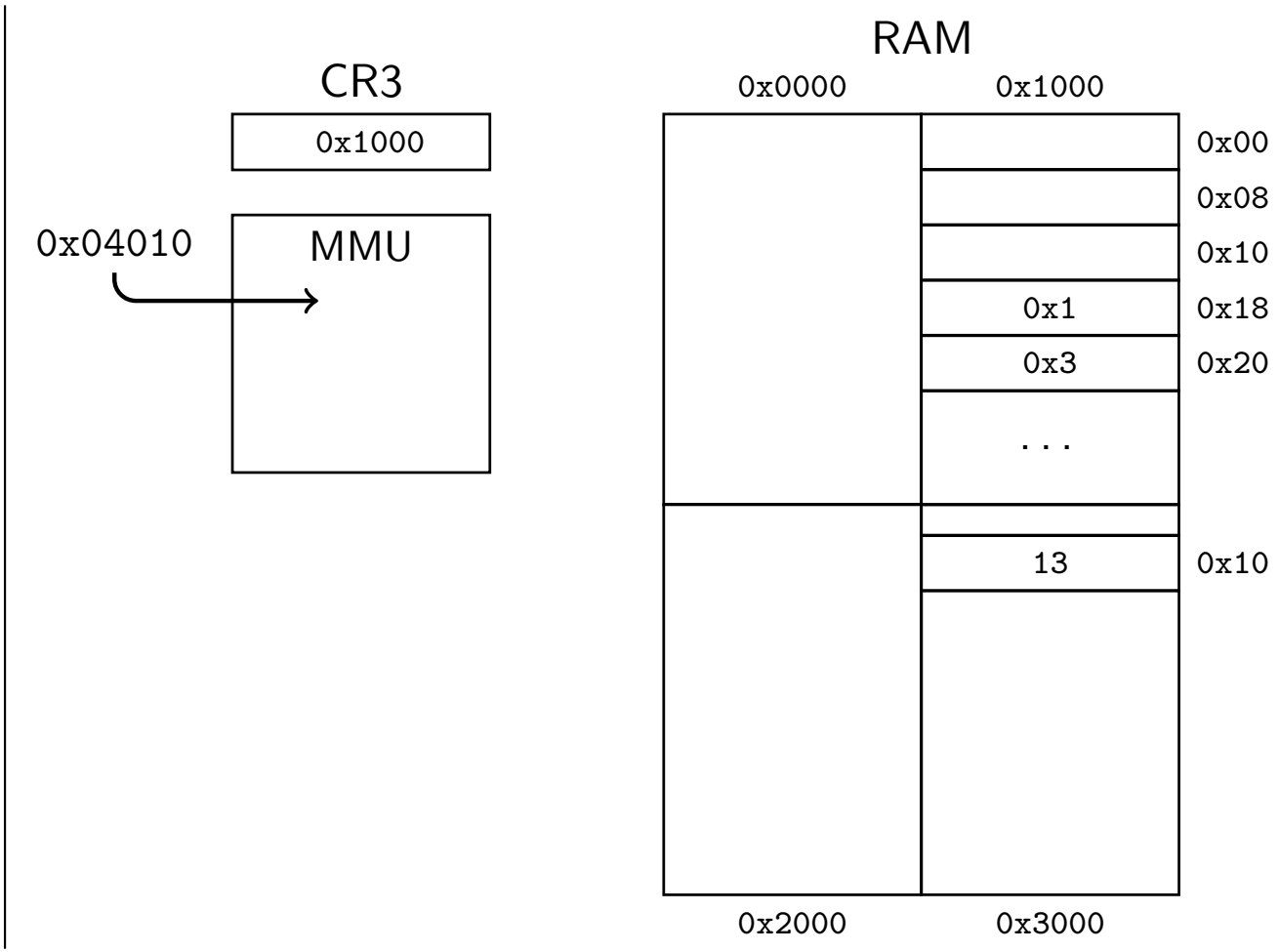
- lire à 0x04010



Traduction d'adresse et table des pages : exemple

- L'adresse physique de la table des pages est stockée dans un registre dédié du processeur : le CR3.

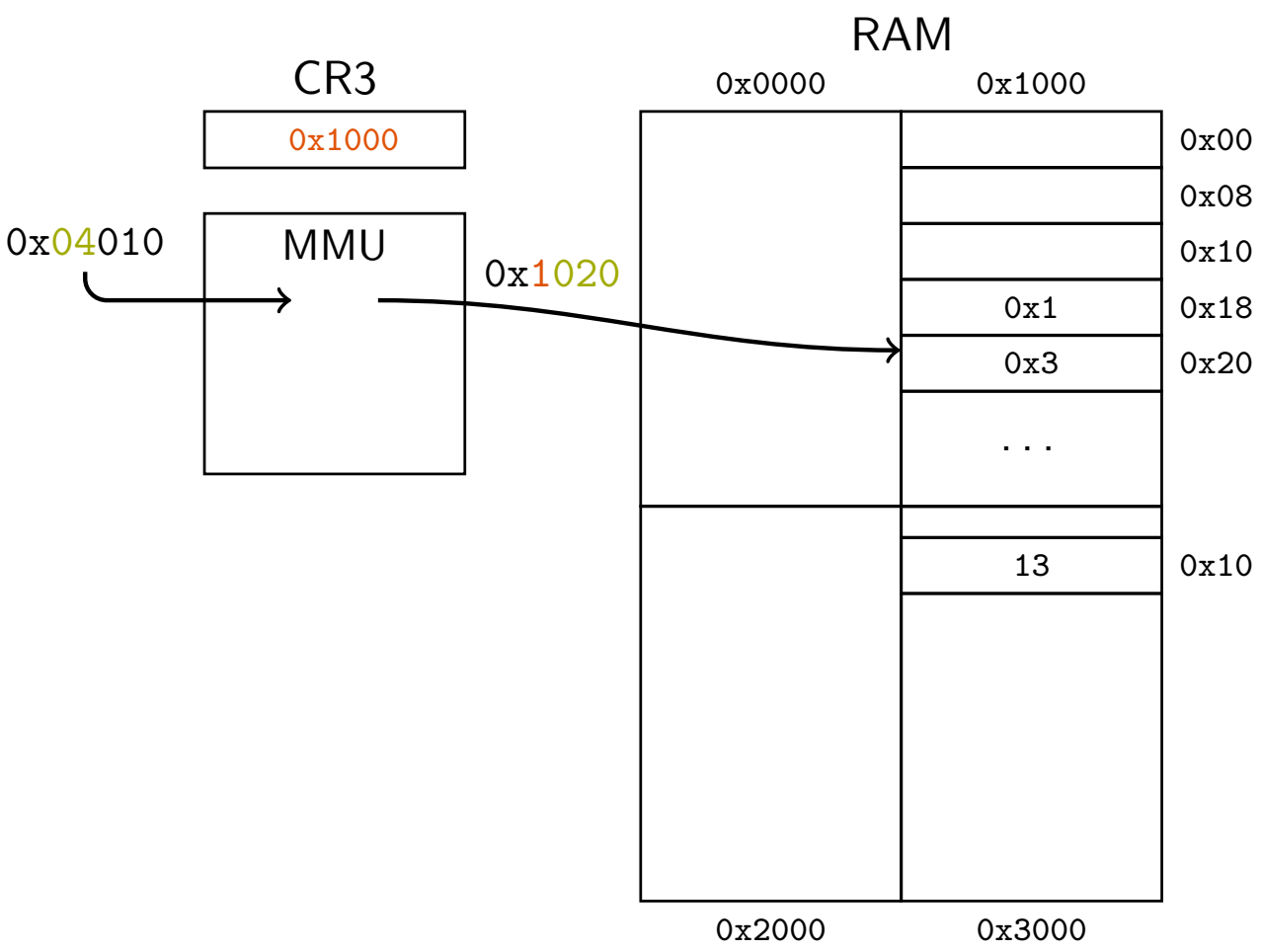
- lire à 0x04010



Traduction d'adresse et table des pages : exemple

- L'adresse physique de la table des pages est stockée dans un registre dédié du processeur : le CR3.

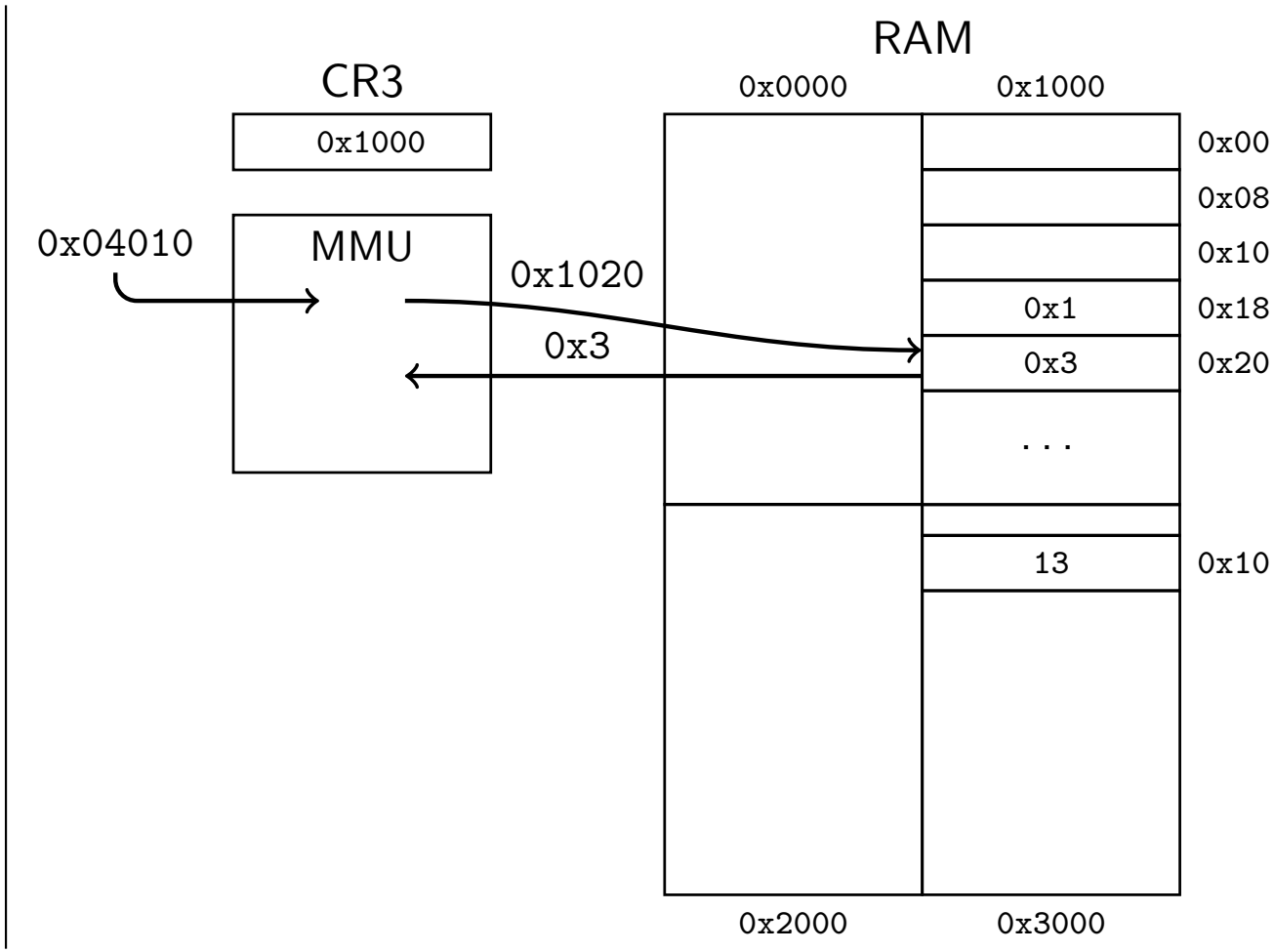
- lire à 0x04010
  - lire l'entrée 0x04
  - lire l'offset 0x020 (0x04×8)



Traduction d'adresse et table des pages : exemple

- L'adresse physique de la table des pages est stockée dans un registre dédié du processeur : le CR3.

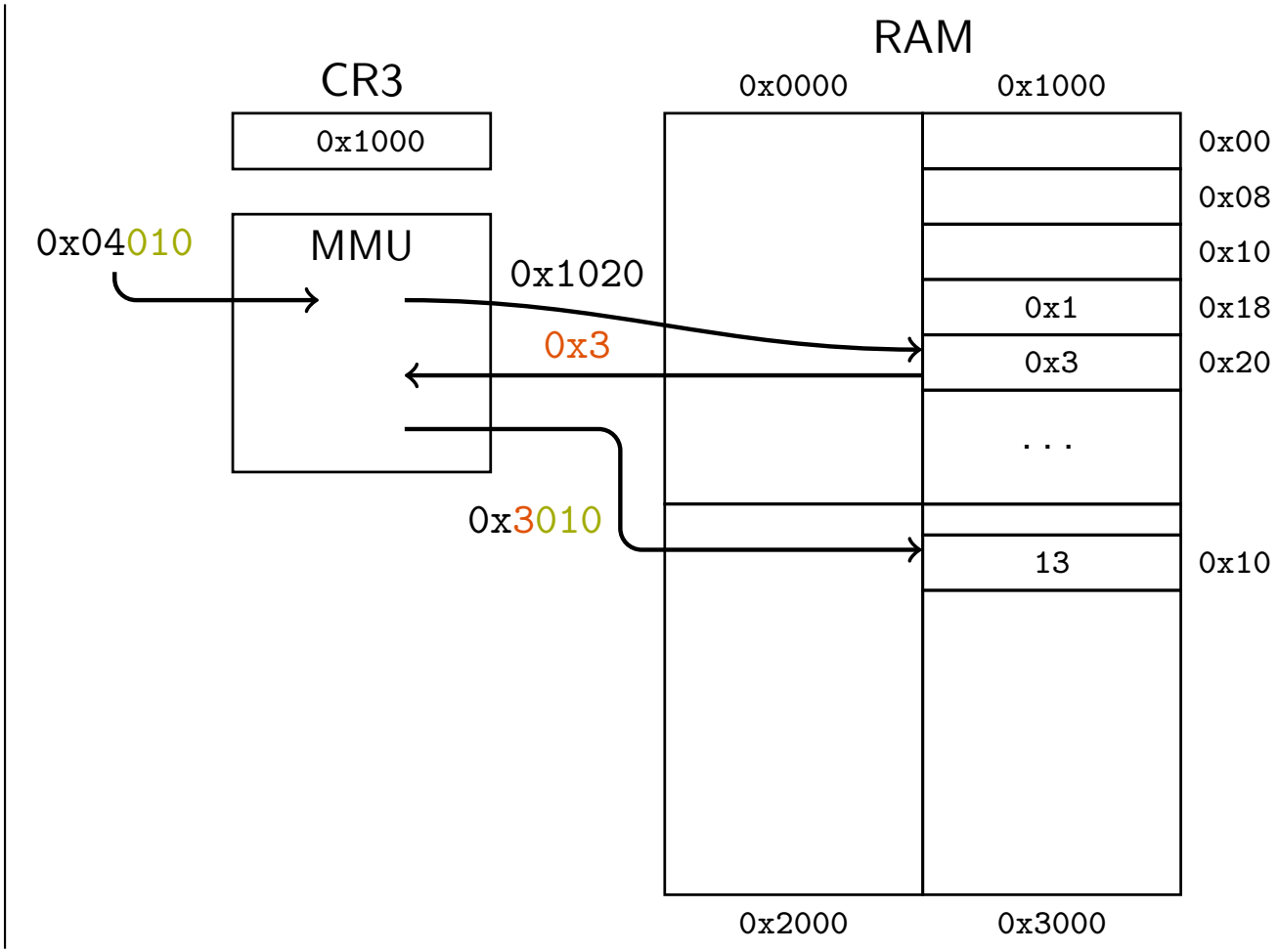
- lire à 0x04010



Traduction d'adresse et table des pages : exemple

- L'adresse physique de la table des pages est stockée dans un registre dédié du processeur : le CR3.

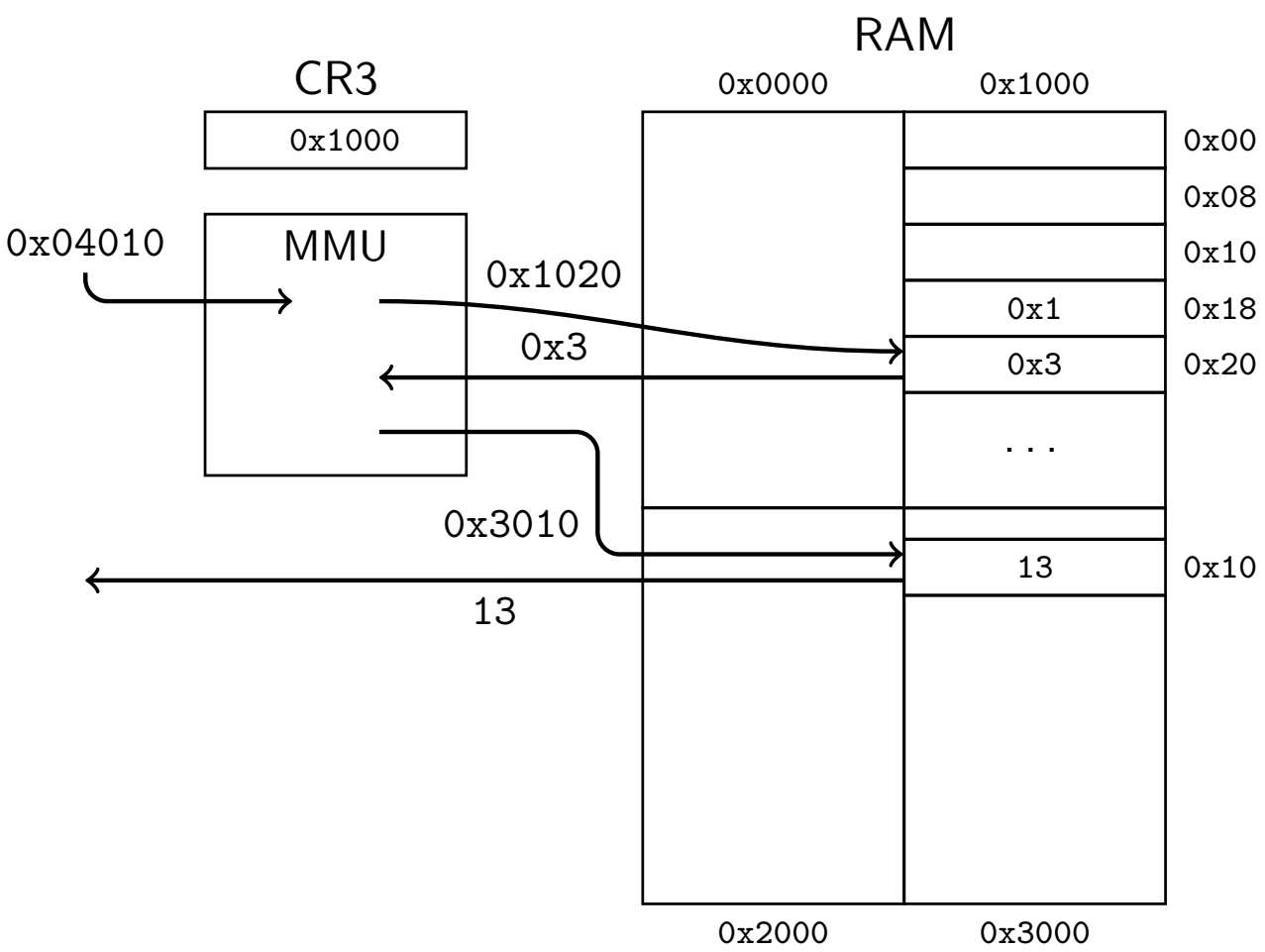
- lire à 0x04010



Traduction d'adresse et table des pages : exemple

- L'adresse physique de la table des pages est stockée dans un registre dédié du processeur : le CR3.

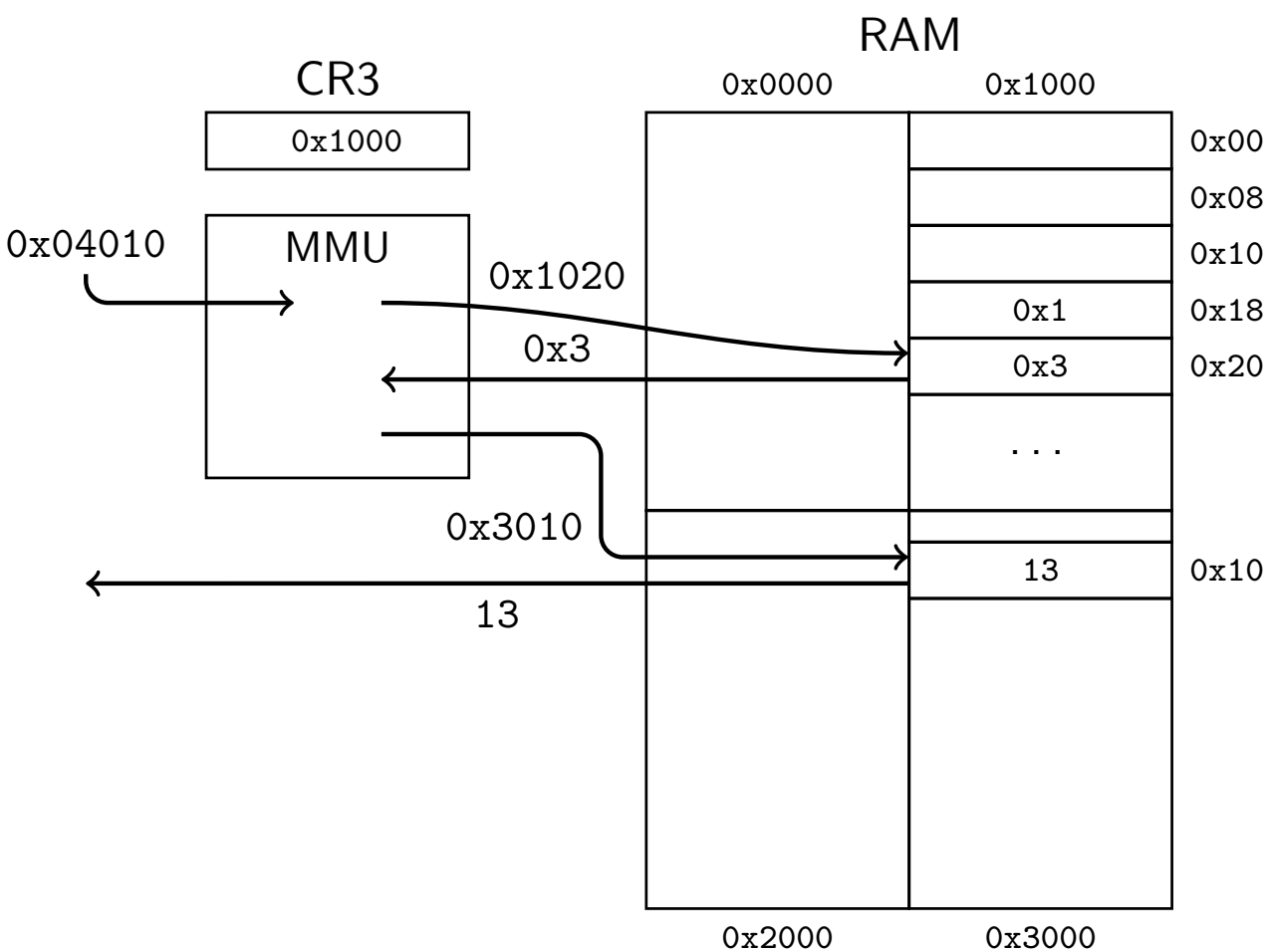
- lire à 0x04010
  - valeur : 13



Traduction d'adresse et table des pages : exercice

- L'adresse physique de la table des pages est stockée dans un registre dédié du processeur : le CR3.

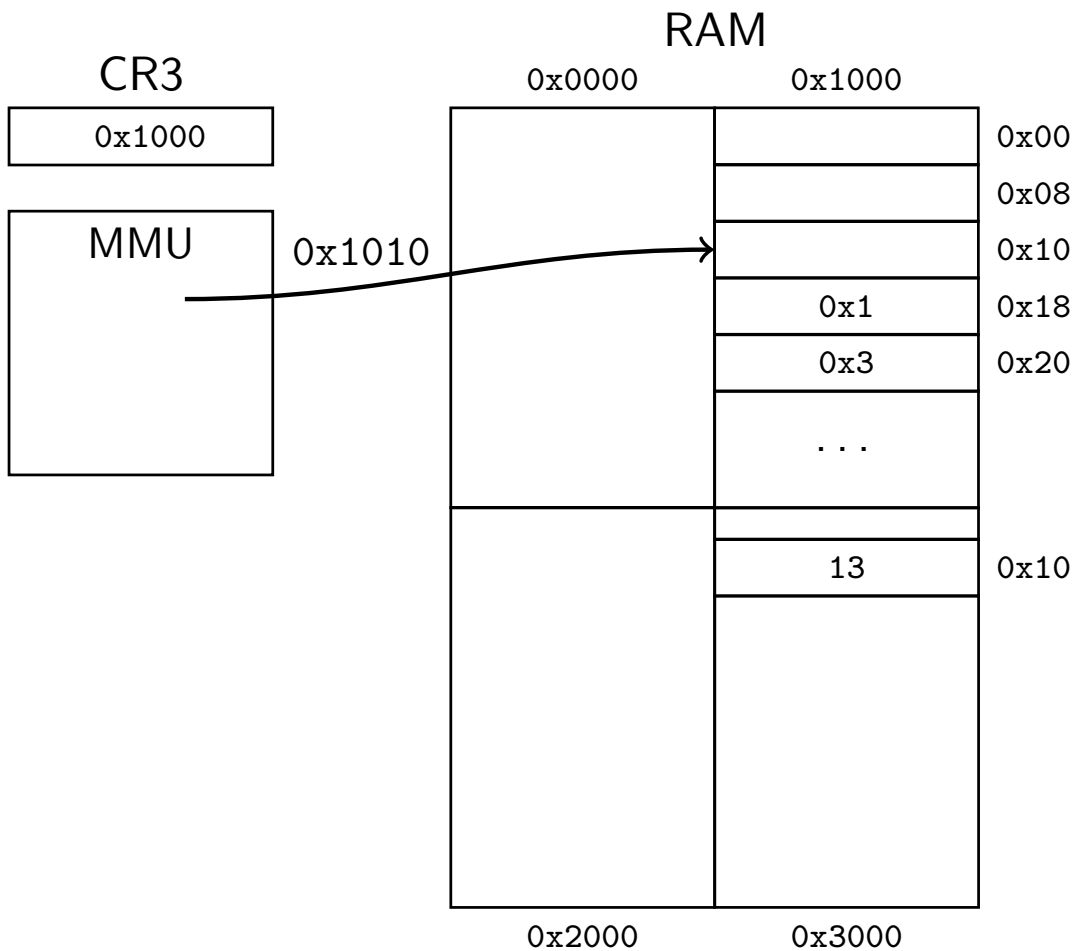
- lire à 0x04010
  - valeur : 13
- associer :
  - l'adresse virtuelle 0x2000
  - l'adresse physique 0



Traduction d'adresse et table des pages : exercice

- L'adresse physique de la table des pages est stockée dans un registre dédié du processeur : le CR3.

- lire à 0x04010
  - ▶ valeur : 13
- associer :
  - l'adresse virtuelle 0x2000
  - l'adresse physique 0
  - ▶ on voudrait écrire 0 à 0x1010 (phys.)



Traduction d'adresse et table des pages : exercice

- L'adresse physique de la table des pages est stockée dans un registre dédié du processeur : le CR3.

- lire à 0x04010
  - valeur : 13
- associer :
  - l'adresse virtuelle 0x2000
  - l'adresse physique 0
  - écrire 0 à 0x1010 (phys.)
  - écrire 0 à 0x3010 (virt.)

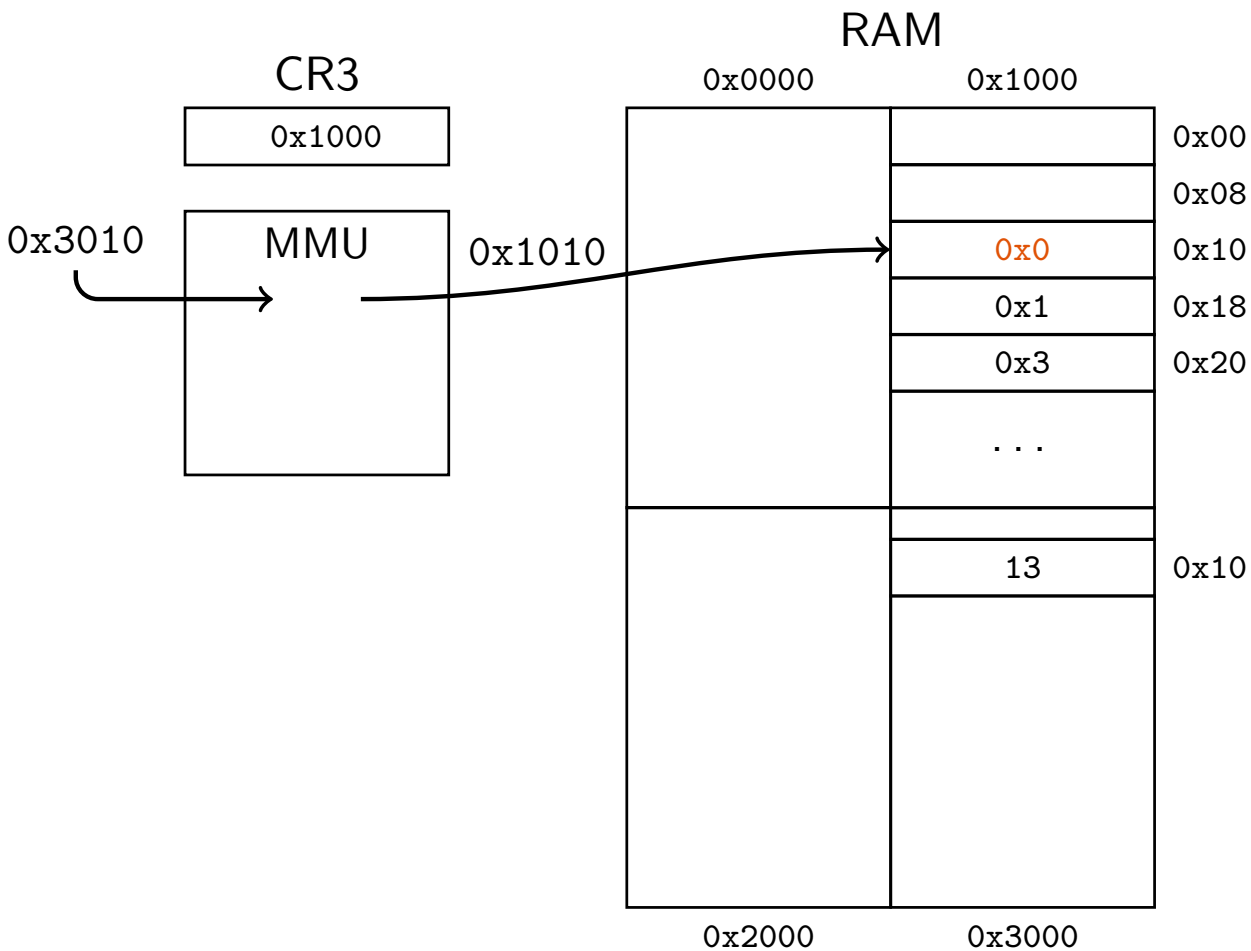


Table des pages : une structure creuse

- Dimensions habituelles sur un processeur x86 64 bits
  - page $\rightarrow 2^{12}$ (4 KiB)
  - adresse physique $\rightarrow 2^{52}$ (4 PiB max) (52/64 bits utilisables)
  - adresse virtuelle $\rightarrow 2^{48}$ (256 TiB) (48/64 bits utilisables)

- Le système peut attribuer n'importe quelle adresse virtuelle, quelque soit la quantité de mémoire physique

- Conséquence : table des pages immense
  - nombre d'entrées $\rightarrow 2^{48-12} = 2^{36}$
  - une seule entrée fait 8 octets
  - taille d'une table $\rightarrow$ 512 GiB
  - la plupart des entrées sont vides

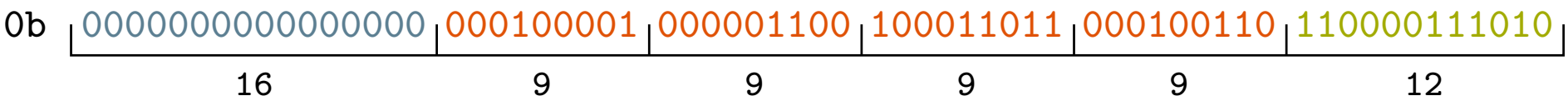
- Solution : utiliser une structure creuse

Table des pages et décomposition d'adresse

```
0b  0000000000000000 000100001 000001100 100011011 000100110 110000111010
```

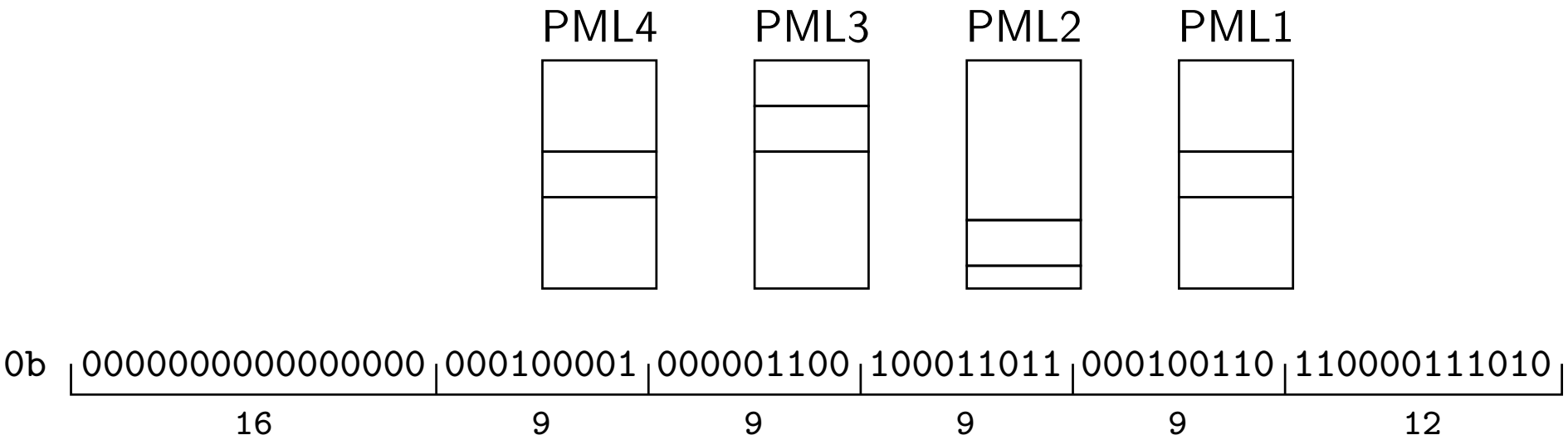
- L'adresse virtuelle est décomposée en 6 parties

Table des pages et décomposition d'adresse



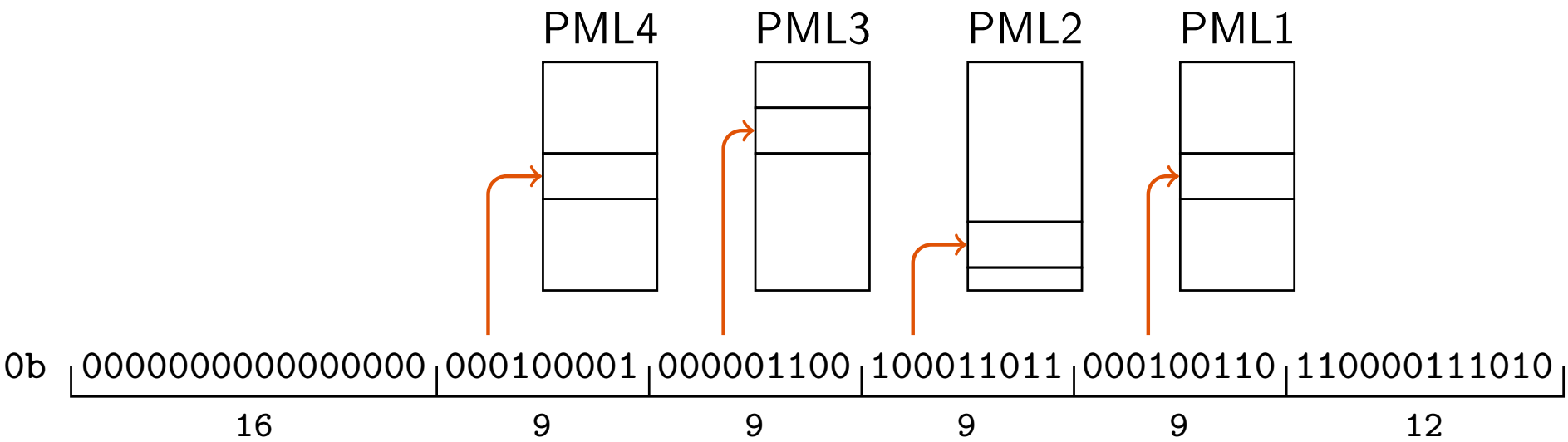
- L'adresse virtuelle est décomposée en 6 parties
  - L'offset
  - Les indices
  - L'extension de signe

Table des pages et décomposition d'adresse



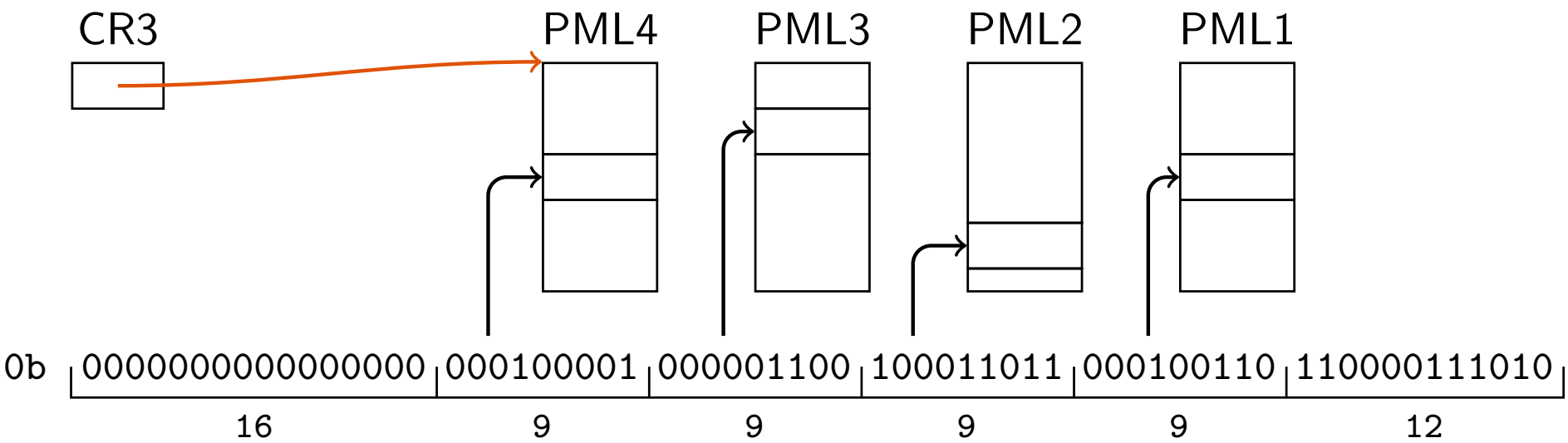
- L'adresse virtuelle est décomposée en 6 parties
  - L'offset
  - Les indices
  - L'extension de signe
- La table des pages est décomposée en 4 niveaux

Table des pages et décomposition d'adresse



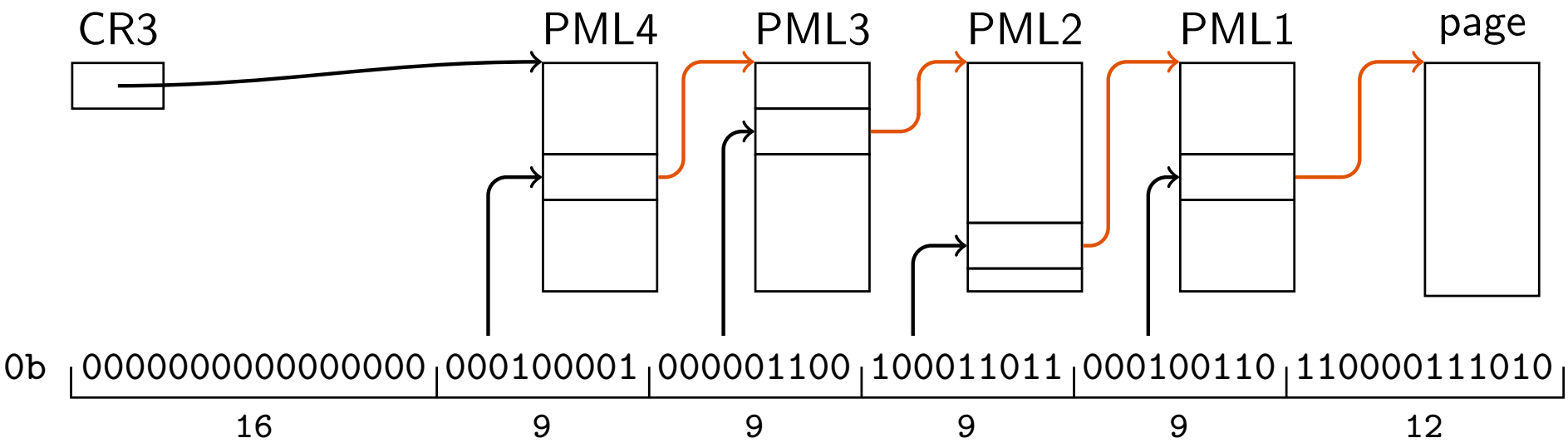
- L'adresse virtuelle est décomposée en 6 parties
  - L'offset
  - Les indices qui indiquent l'entrée dans le niveau de table correspondant
  - L'extension de signe
- La table des pages est décomposée en 4 niveaux
  - Chaque niveau contient des entrées

Table des pages et décomposition d'adresse



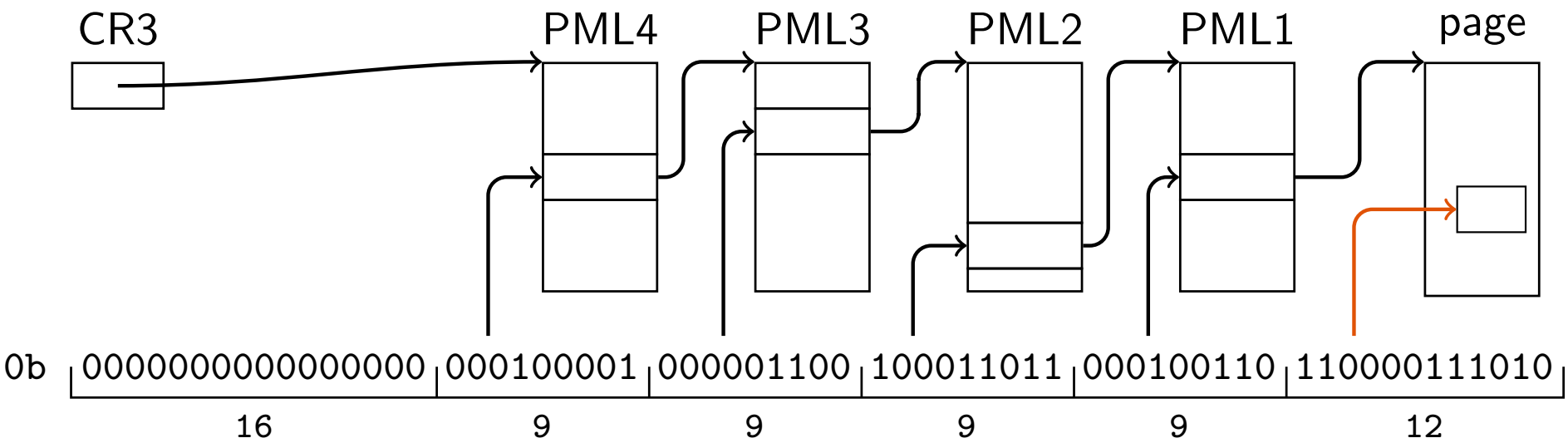
- L'adresse virtuelle est décomposée en 6 parties
  - L'offset
  - Les indices qui indiquent l'entrée dans le niveau de table correspondant
  - L'extension de signe
- La table des pages est décomposée en 4 niveaux
  - Chaque niveau contient des entrées
  - Le premier niveau est pointé (physiquement) par le CR3

Table des pages et décomposition d'adresse



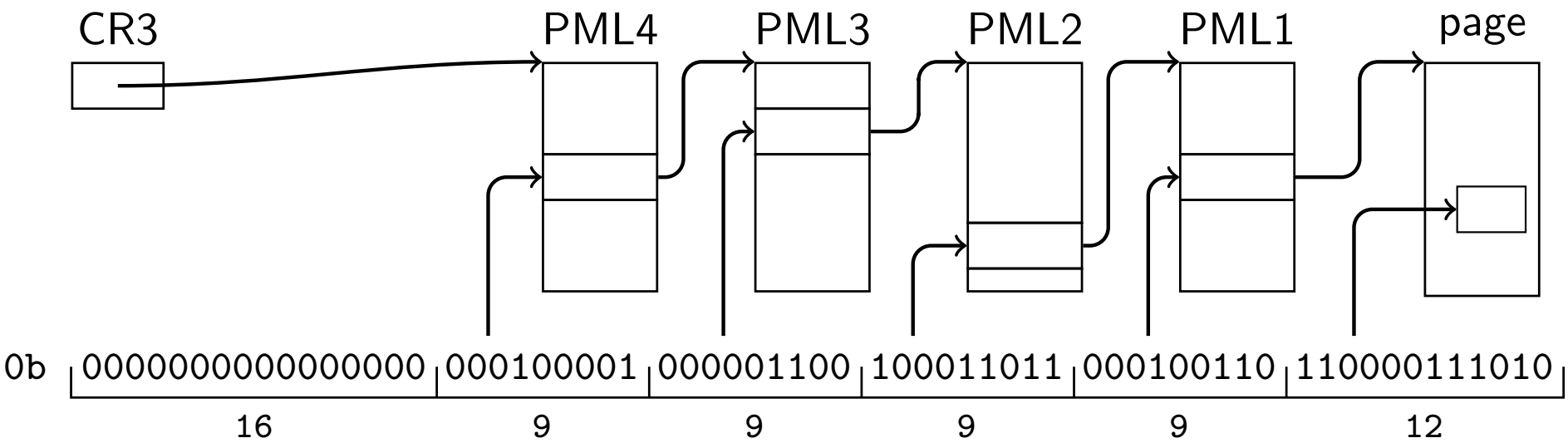
- L'adresse virtuelle est décomposée en 6 parties
  - L'offset
  - Les indices qui indiquent l'entrée dans le niveau de table correspondant
  - L'extension de signe
- La table des pages est décomposée en 4 niveaux
  - Chaque niveau contient des entrées
  - Le premier niveau est pointé (physiquement) par le CR3
  - Chaque entrée non vide pointe (physiquement) un niveau suivant

Table des pages et décomposition d'adresse



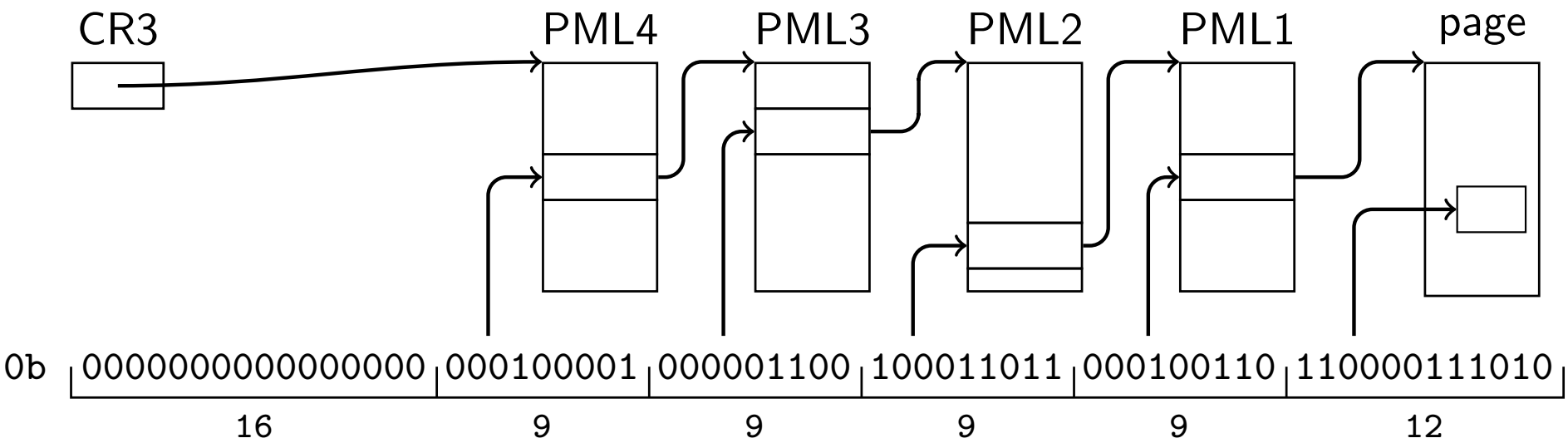
- L'adresse virtuelle est décomposée en 6 parties
  - L'offset qui indique la position de la donnée dans la page
  - Les indices qui indiquent l'entrée dans le niveau de table correspondant
  - L'extension de signe
- La table des pages est décomposée en 4 niveaux
  - Chaque niveau contient des entrées
  - Le premier niveau est pointé (physiquement) par le CR3
  - Chaque entrée non vide pointe (physiquement) un niveau suivant

Table des pages et décomposition d'adresse



- L'adresse virtuelle est décomposée en 6 parties
  - L'offset qui indique la position de la donnée dans la page
  - Les indices qui indiquent l'entrée dans le niveau de table correspondant
  - L'extension de signe qui assure la compatibilité descendante
- La table des pages est décomposée en 4 niveaux
  - Chaque niveau contient des entrées
  - Le premier niveau est pointé (physiquement) par le CR3
  - Chaque entrée non vide pointe (physiquement) un niveau suivant

Table des pages et décomposition d'adresse



- L'adresse virtuelle est décomposée en 6 parties
  - L'offset qui indique la position de la donnée dans la page
  - Les indices qui indiquent l'entrée dans le niveau de table correspondant
  - L'extension de signe qui assure la compatibilité descendante
- La table des pages est décomposée en 4 niveaux
  - Chaque niveau contient des entrées
  - Le premier niveau est pointé (physiquement) par le CR3
  - Chaque entrée non vide pointe (physiquement) un niveau suivant

Traduction d'adresse et table des pages : résumé

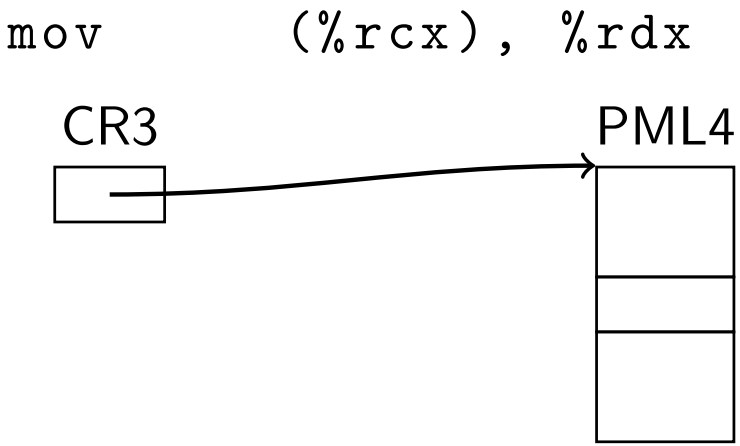
- Une page est une donnée de taille fixe
  - Une page peut être stockée dans un emplacement de page
  - Un emplacement de page en mémoire a une adresse physique
  - Un emplacement de page en mémoire peut avoir une ou plusieurs adresses virtuelles
- La table des pages est une structure de données arborescente
  - Utilisée par la MMU pendant la traduction d'adresse
  - Stockée en mémoire et pointée physiquement par le `CR3`
  - Indiquant uniquement des adresses physiques
  - Indexée uniquement par des adresses virtuelles
- Pendant une traduction d'adresse, l'adresse virtuelle est décomposée
  - L'offset qui indique la position dans la page n'est pas modifié
  - Les indices indiquent quels niveaux successifs utiliser dans la table

Performance et traduction d'adresse

```
mov     (%rcx), %rdx
```

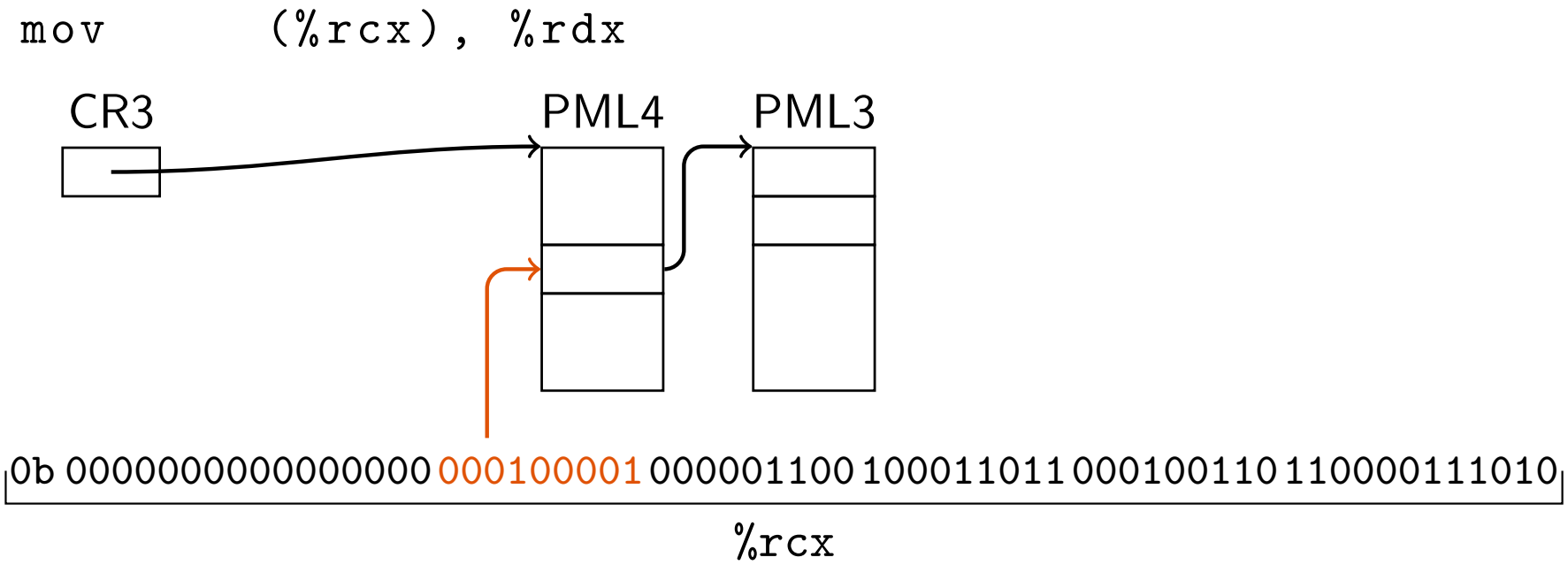
- Combien de lecture en mémoire pour accéder à une donnée ?

Performance et traduction d'adresse



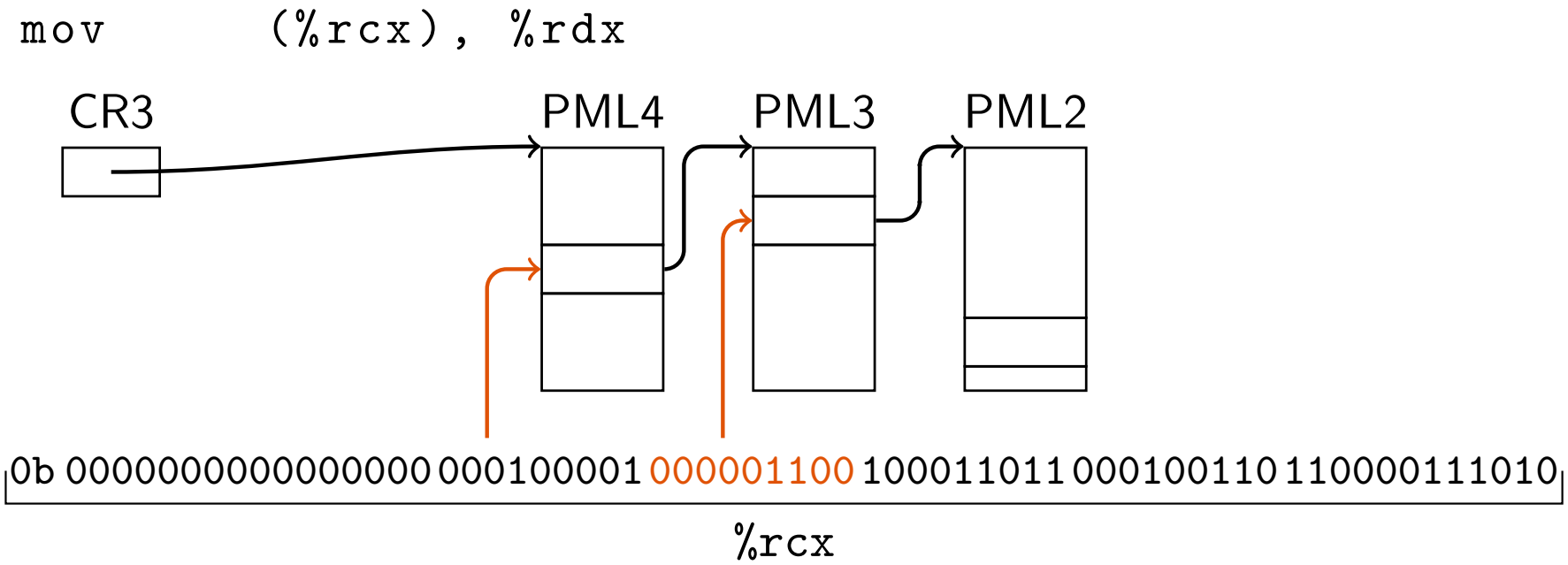
- Combien de lecture en mémoire pour accéder à une donnée ?
  - Lecture du `CR3` (registre processeur) : 0

Performance et traduction d'adresse



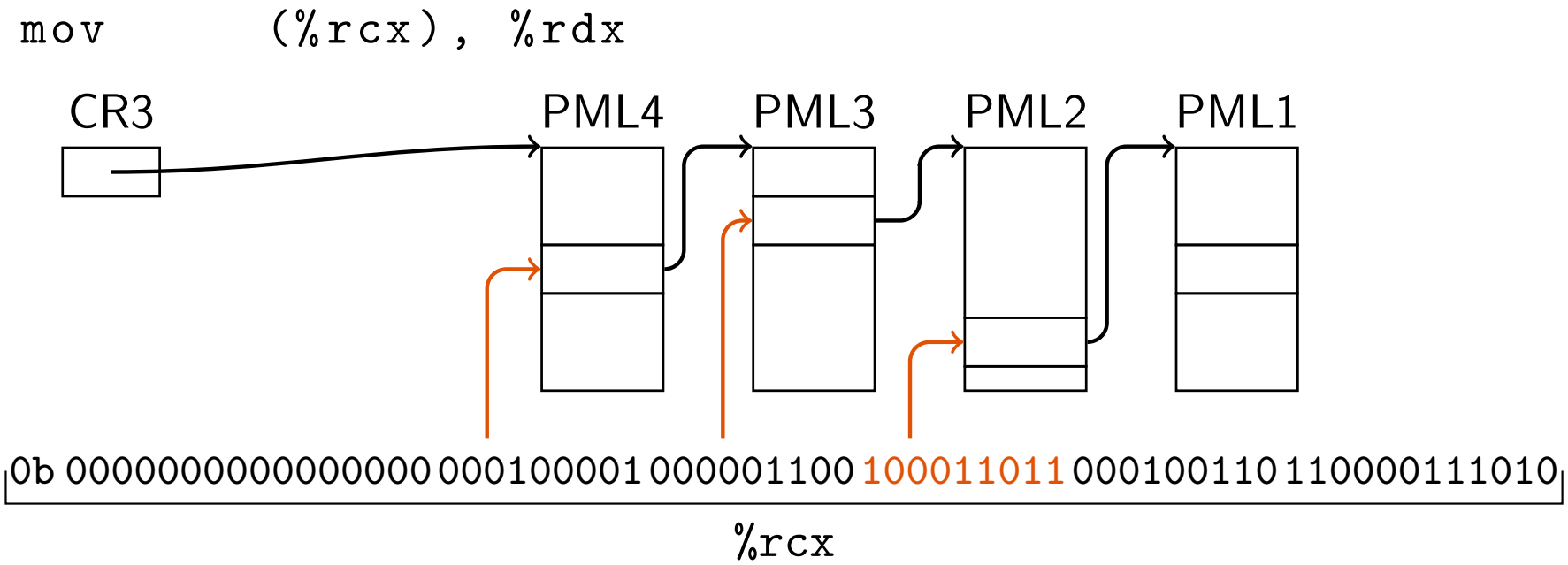
- Combien de lecture en mémoire pour accéder à une donnée ?
  - Lecture du `CR3` (registre processeur) : 0
  - Lecture des entrées des niveaux intermédiaires : 1

Performance et traduction d'adresse



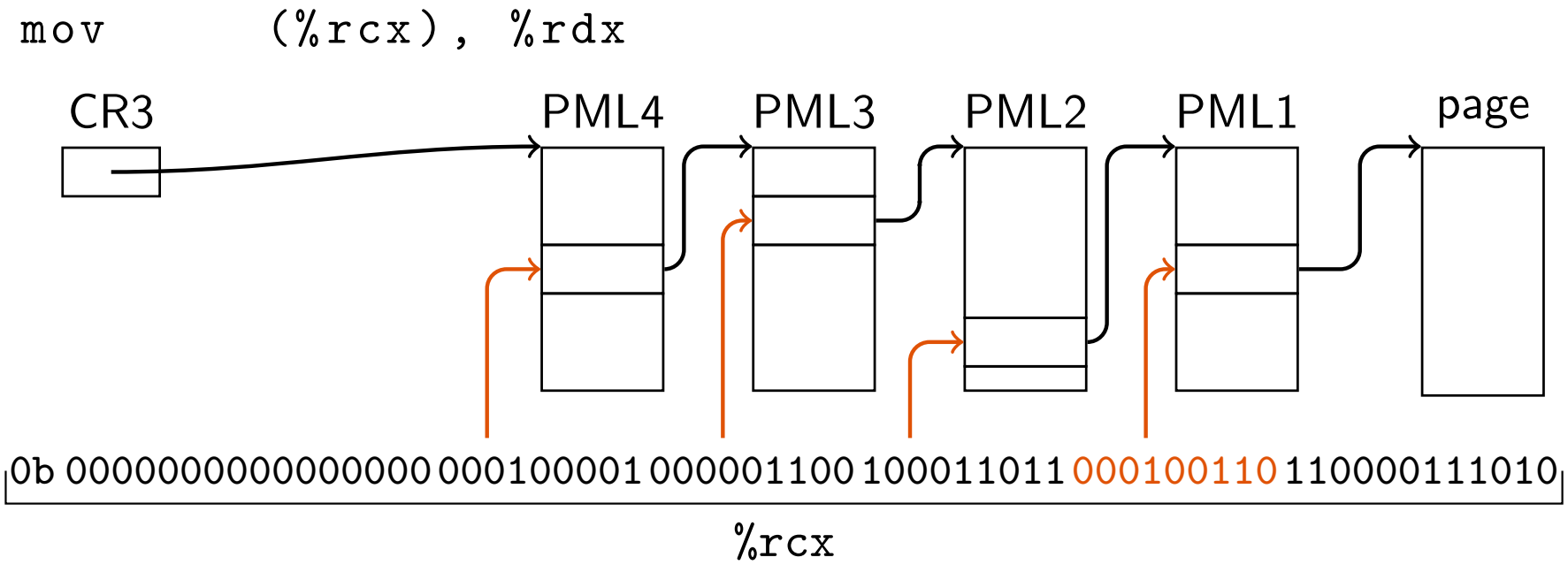
- Combien de lecture en mémoire pour accéder à une donnée ?
  - Lecture du `CR3` (registre processeur) : 0
  - Lecture des entrées des niveaux intermédiaires : 2

Performance et traduction d'adresse



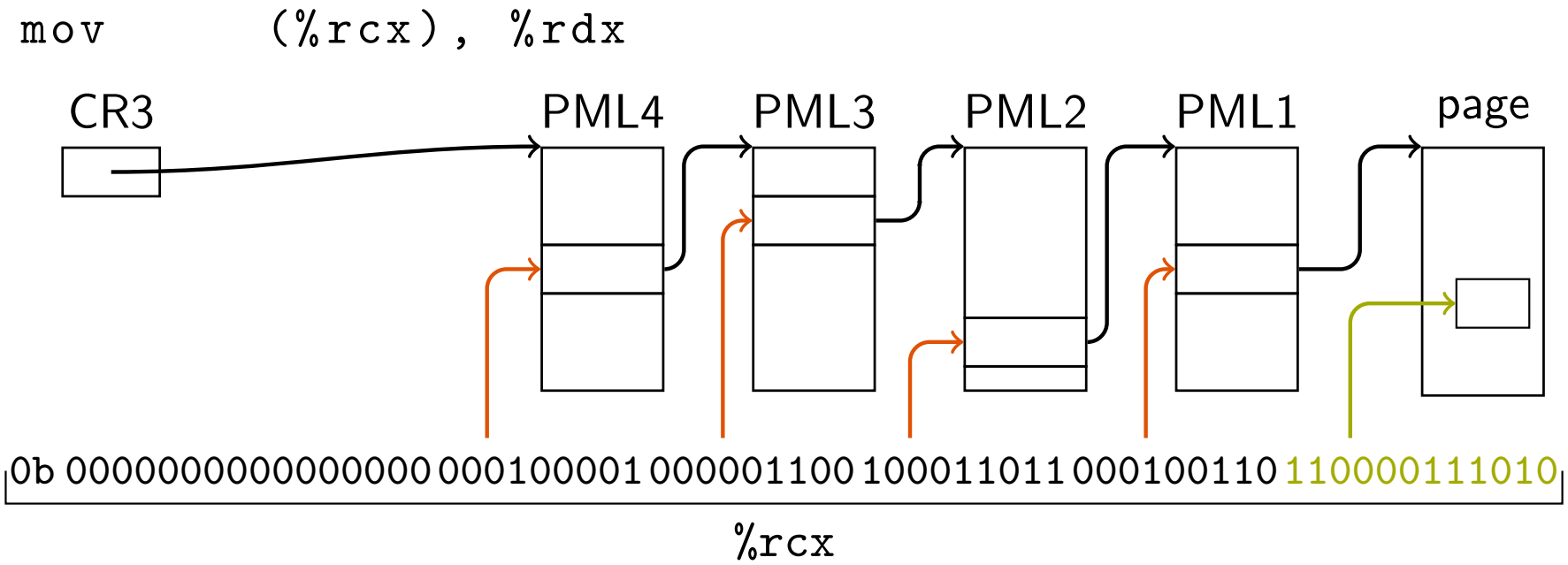
- Combien de lecture en mémoire pour accéder à une donnée ?
  - Lecture du `CR3` (registre processeur) : 0
  - Lecture des entrées des niveaux intermédiaires : 3

Performance et traduction d'adresse



- Combien de lecture en mémoire pour accéder à une donnée ?
  - Lecture du `CR3` (registre processeur) : 0
  - Lecture des entrées des niveaux intermédiaires : 4

Performance et traduction d'adresse



- Combien de lecture en mémoire pour accéder à une donnée ?
  - Lecture du CR3 (registre processeur) : 0
  - Lecture des entrées des niveaux intermédiaires : 4
  - Lecture de la donnée demandée : 1
- Accès mémoire par requête : 5

Performance et *Translation Lookaside Buffer*

- Principe de localité : si un processeur accède à une adresse mémoire, il est probable qu'il accède à une adresse mémoire proche dans un avenir proche

- Les traductions d'adresse faites par la MMU sont mémorisées dans une mémoire cache dédiée
  - Le cache de la MMU s'appelle le Translation Lookaside Buffer
  - Le TLB stocke des paires (adresse virtuelle, adresse physique)

- Le TLB est une mémoire rapide mais de taille limitée

Capacité du *Translation Lookaside Buffer*

```
#define STEP      (1ul << 12)
#define SIZE      6
#define REPEAT    10

void access_sparse(char *mem) {
    size_t i, j;

    for (i = 0; i < REPEAT; i++)
        for (j = 0; j < SIZE; j++)
            access_mem(mem + j * STEP);
}
```

- Avec un TLB pouvant stocker les 4 dernières traductions
- La fonction `access_mem(addr)` fait 2 accès mémoire à `addr`
- Combien d'accès mémoire effectués (sans compter l'accès au code)
  - Pour des pages de 2^{12} octets ?

Capacité du *Translation Lookaside Buffer*

```
#define STEP      (1ul << 12)
#define SIZE      6
#define REPEAT    10

void access_sparse(char *mem) {
    size_t i, j;

    for (i = 0; i < REPEAT; i++)
        for (j = 0; j < SIZE; j++)
            access_mem(mem + j * STEP);
}
```

- Avec un TLB pouvant stocker les 4 dernières traductions
- La fonction `access_mem(addr)` fait 2 accès mémoire à `addr`
- Combien d'accès mémoire effectués (sans compter l'accès au code)
  - Pour des pages de 2^{12} octets ? $\rightarrow 10 \cdot 6 \cdot (4+2) = 360$

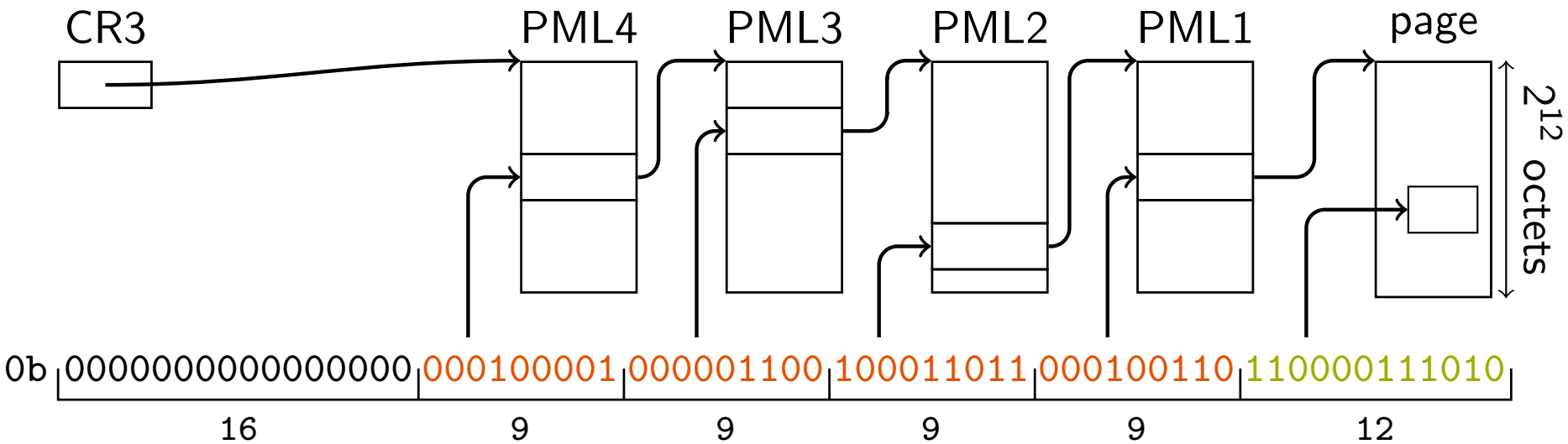
Capacité du *Translation Lookaside Buffer*

```
#define STEP      (1ul << 12)
#define SIZE      6
#define REPEAT    10

void access_sparse(char *mem) {
    size_t i, j;

    for (i = 0; i < REPEAT; i++)
        for (j = 0; j < SIZE; j++)
            access_mem(mem + j * STEP);
}
```

- Avec un TLB pouvant stocker les 4 dernières traductions
- La fonction `access_mem(addr)` fait 2 accès mémoire à `addr`
- Combien d'accès mémoire effectués (sans compter l'accès au code)
  - Pour des pages de 2^{12} octets ? $\rightarrow 10 \cdot 6 \cdot (4+2) = 360$
  - Les deux tiers des accès sont des accès à la table des pages

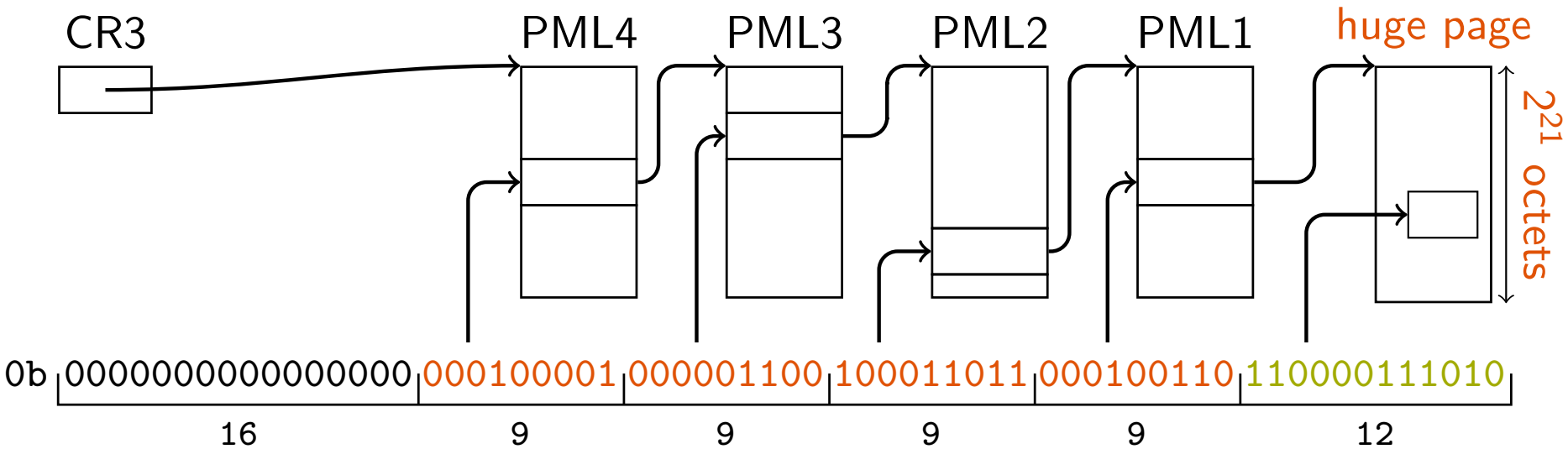
Translation Lookaside Buffer et *Huge Pages*

- Solution : augmenter la taille des pages



Translation Lookaside Buffer et *Huge Pages*

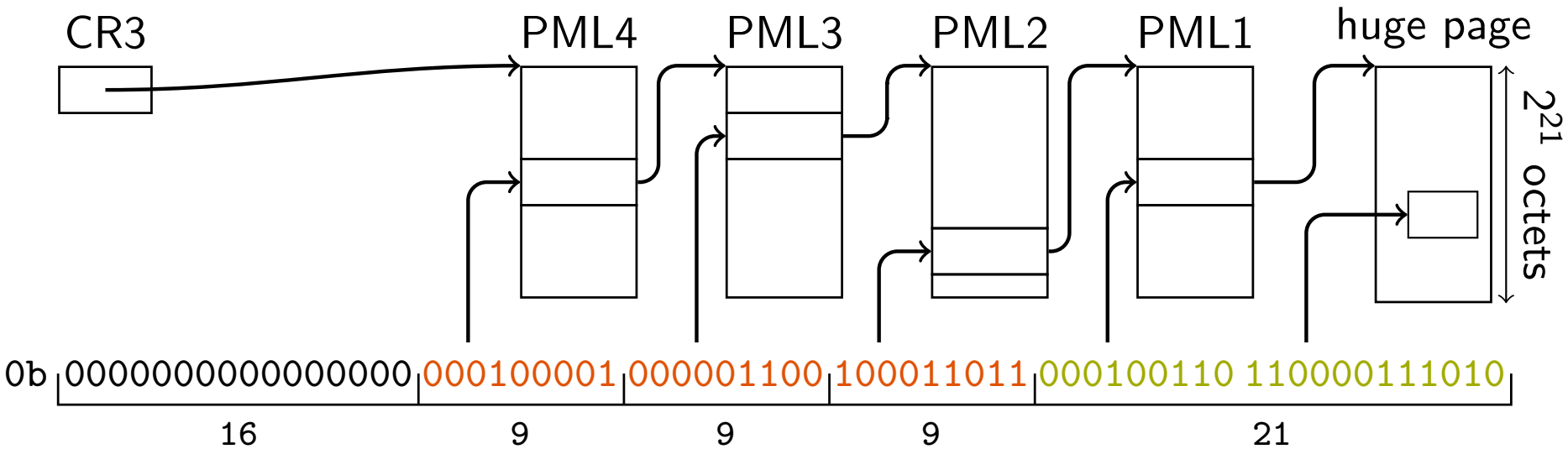
- Solution : augmenter la taille des pages



- La taille d'une page détermine la taille de l'offset

Translation Lookaside Buffer et *Huge Pages*

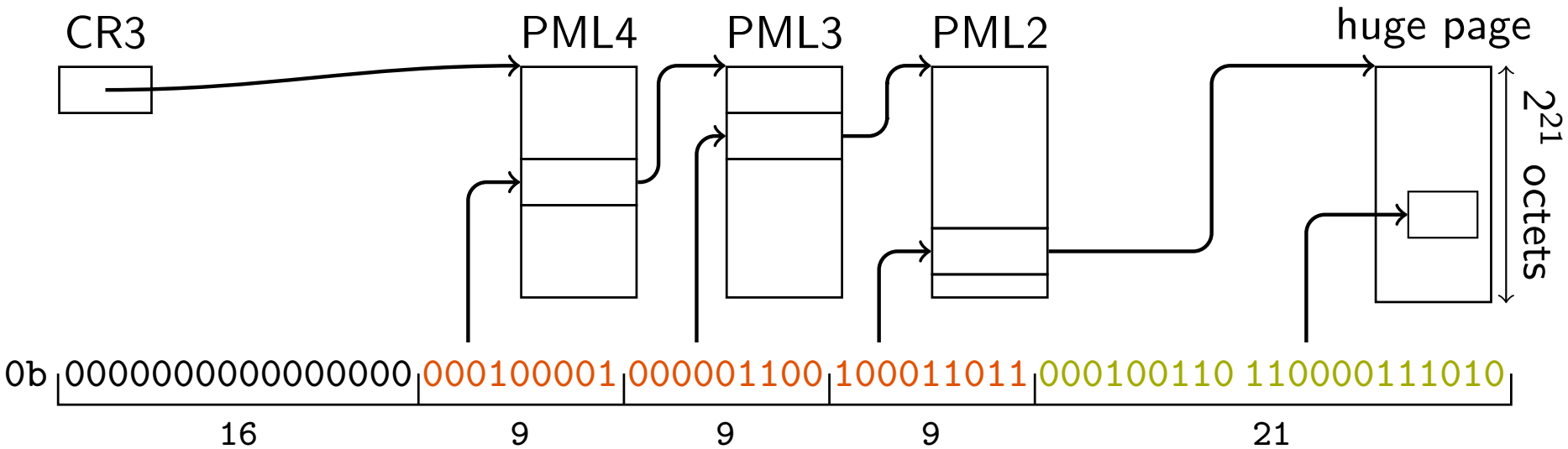
- Solution : augmenter la taille des pages



- La taille d'une page détermine la taille de l'offset
- Ajouter le dernier index à l'offset

Translation Lookaside Buffer et *Huge Pages*

- Solution : augmenter la taille des pages



- La taille d'une page détermine la taille de l'offset
- Ajouter le dernier index à l'offset
- Supprimer le niveau correspondant au dernier index

Capacité du *Translation Lookaside Buffer*

```
#define STEP      (1ul << 12)
#define SIZE      6
#define REPEAT    10

void access_sparse(char *mem) {
    size_t i, j;

    for (i = 0; i < REPEAT; i++)
        for (j = 0; j < SIZE; j++)
            access_mem(mem + j * STEP);
}
```

- Avec un TLB pouvant stocker les 4 dernières traductions
- La fonction `access_mem(addr)` fait 2 accès mémoire à `addr`
- Combien d'accès mémoire effectués (sans compter l'accès au code)
  - Pour des pages de 2^{12} octets ? $\rightarrow 10 \cdot 6 \cdot (4 + 2) = 360$
  - Pour des pages de 2^{21} octets ?

Capacité du *Translation Lookaside Buffer*

```
#define STEP      (1ul << 12)
#define SIZE      6
#define REPEAT    10

void access_sparse(char *mem) {
    size_t i, j;

    for (i = 0; i < REPEAT; i++)
        for (j = 0; j < SIZE; j++)
            access_mem(mem + j * STEP);
}
```

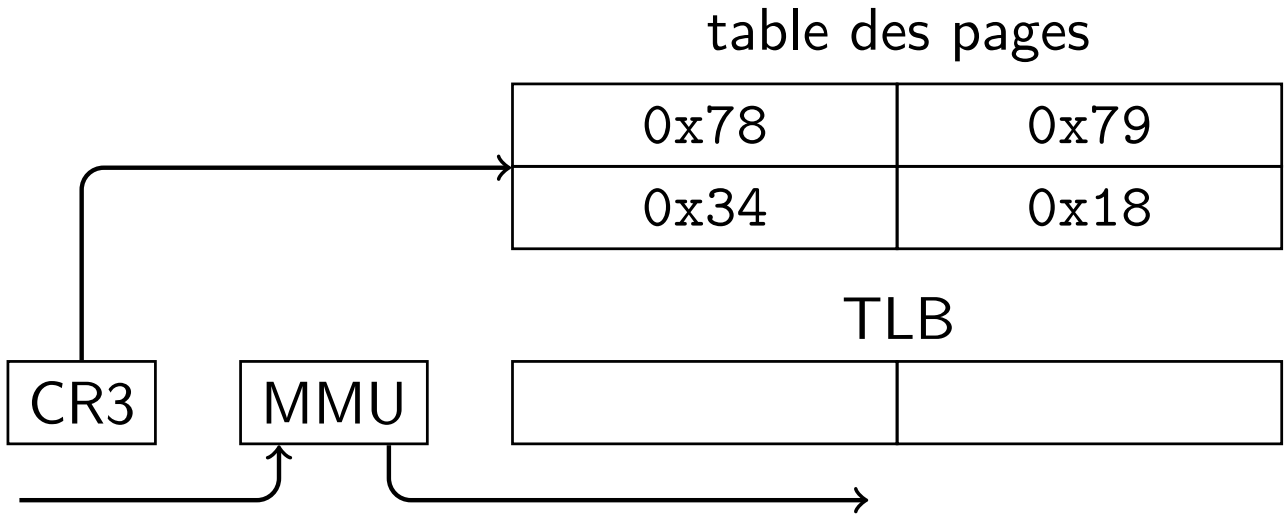
- Avec un TLB pouvant stocker les 4 dernières traductions
- La fonction `access_mem(addr)` fait 2 accès mémoire à `addr`
- Combien d'accès mémoire effectués (sans compter l'accès au code)
  - Pour des pages de 2^{12} octets ? $\rightarrow 10 \cdot 6 \cdot (4 + 2) = 360$
  - Pour des pages de 2^{21} octets ? $\rightarrow 3 + 10 \cdot 6 \cdot 2 = 123$

Huge Page : avantages et inconvénients

- Avantages :
  - Prend moins de place dans le TLB (économie de temps)
  - Permet des tables de page avec moins de niveau (économie d'espace)
- Inconvénients :
  - Moins de finesse dans le contrôle d'accès
  - Moins de finesse dans le grain d'allocation
- Le matériel donne le choix
  - *Huge Page* activée par un bit dédié dans chaque entrée de la table des pages au niveau PML2 (et aux niveaux PML3 et PML4 si supporté par le processeur)
  - Possibilité de mélanger pages normales et *huge pages*
  - Attention : un emplacement de page (normale ou huge) est aligné en mémoire sur la taille de sa page

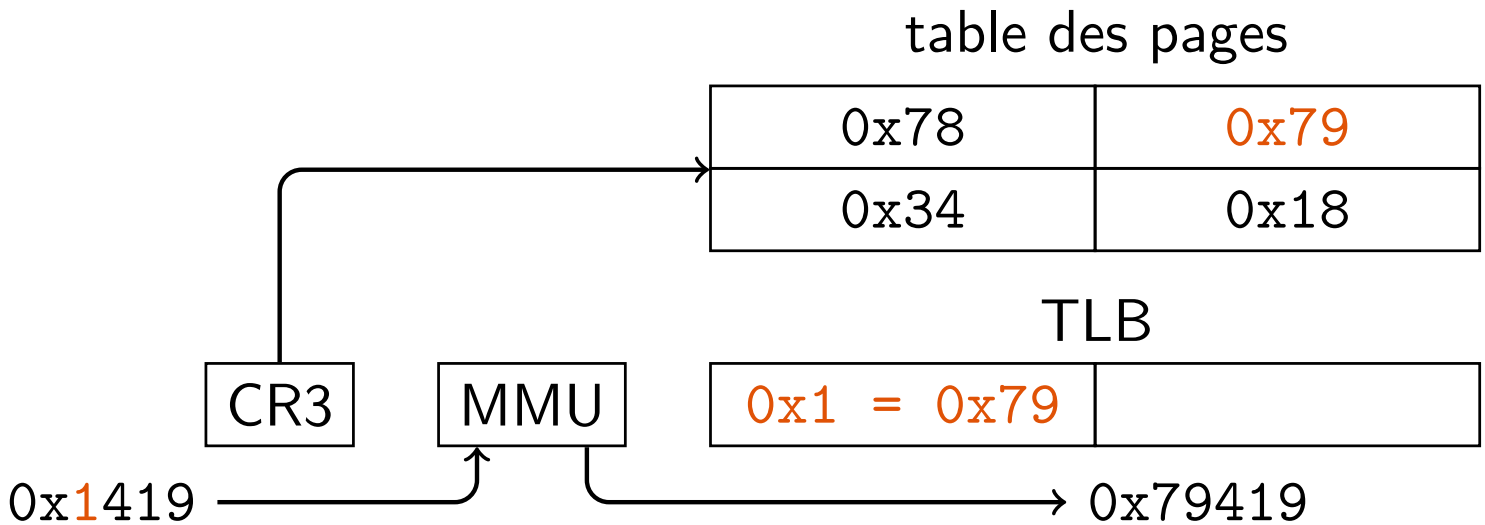
Gestion du *Translation Lookaside Buffer*

- Le TLB est automatiquement rempli à chaque traduction réussie



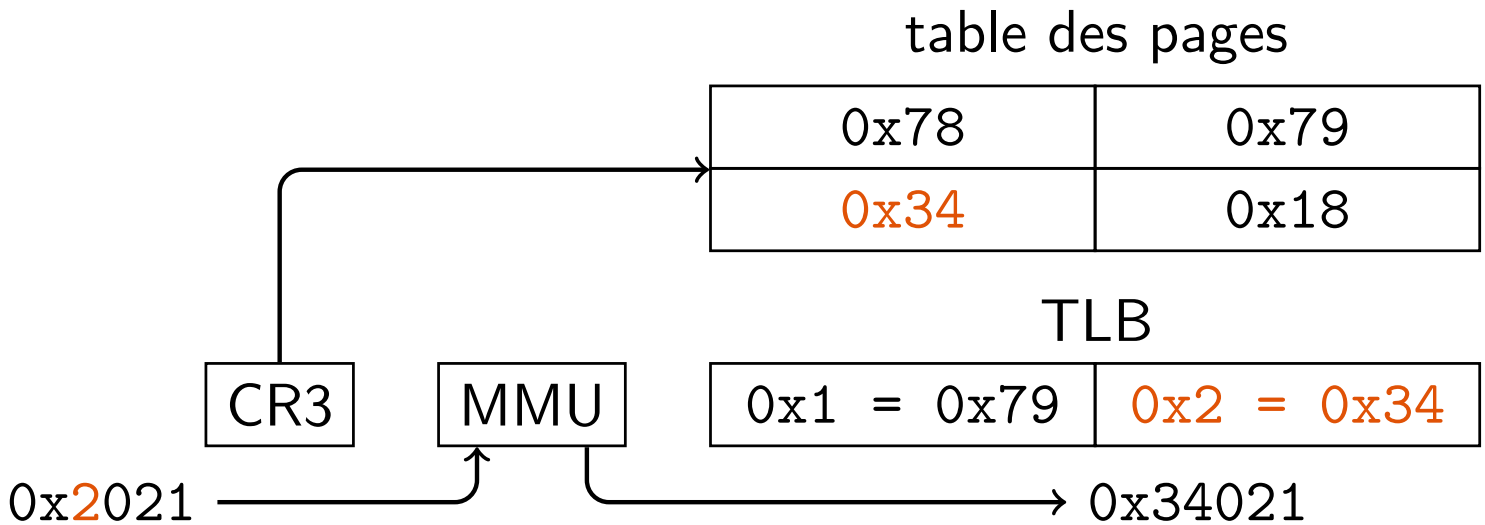
Gestion du *Translation Lookaside Buffer*

- Le TLB est automatiquement rempli à chaque traduction réussie



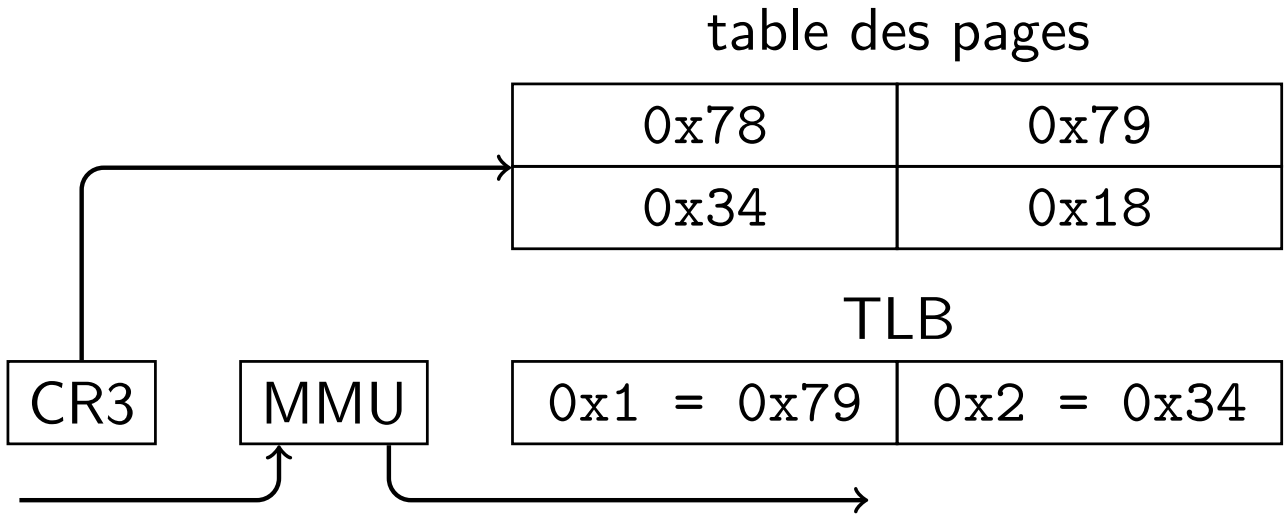
Gestion du *Translation Lookaside Buffer*

- Le TLB est automatiquement rempli à chaque traduction réussie



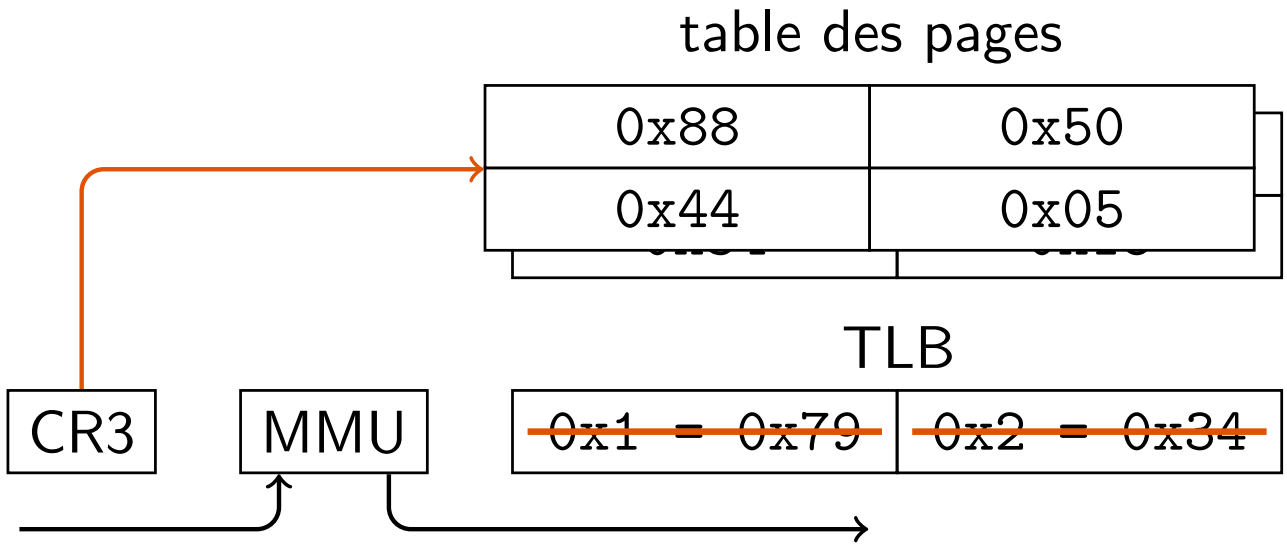
Gestion du *Translation Lookaside Buffer*

- Le TLB est automatiquement rempli à chaque traduction réussie
- Le TLB est automatiquement vidé quand



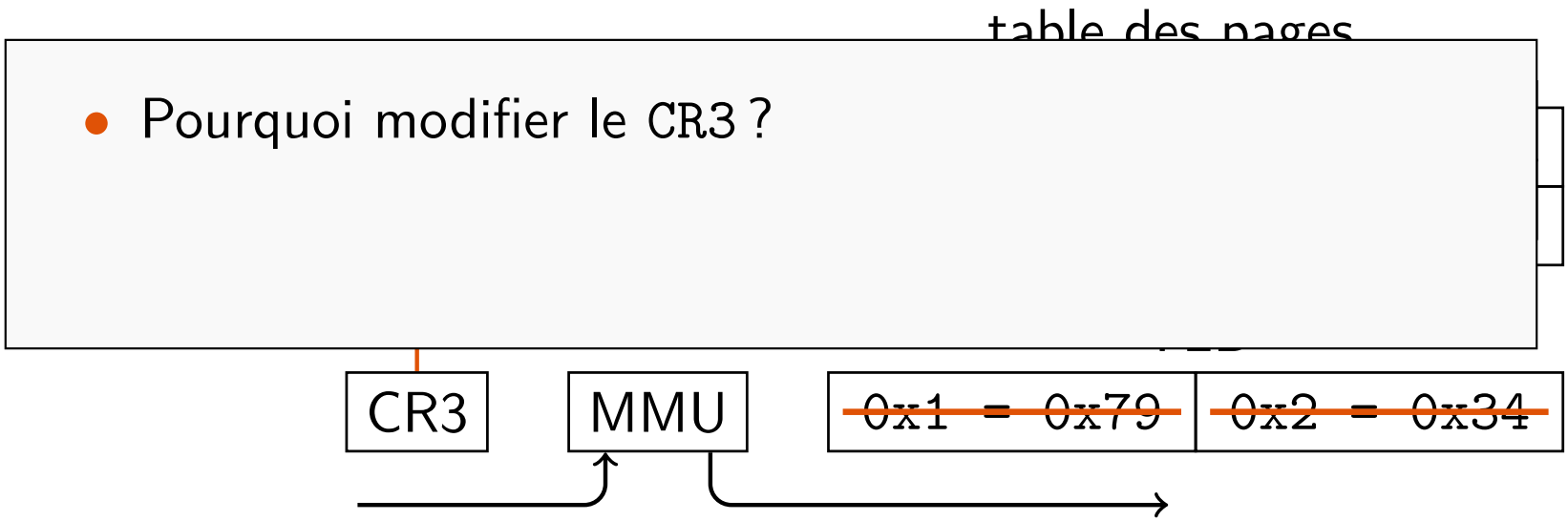
Gestion du *Translation Lookaside Buffer*

- Le TLB est automatiquement rempli à chaque traduction réussie
- Le TLB est automatiquement vidé quand le `CR3` est modifié
  - Modifier le `CR3` = changer la manière dont la MMU traduit les adresses
  - Les traductions faites précédemment sont donc caduques



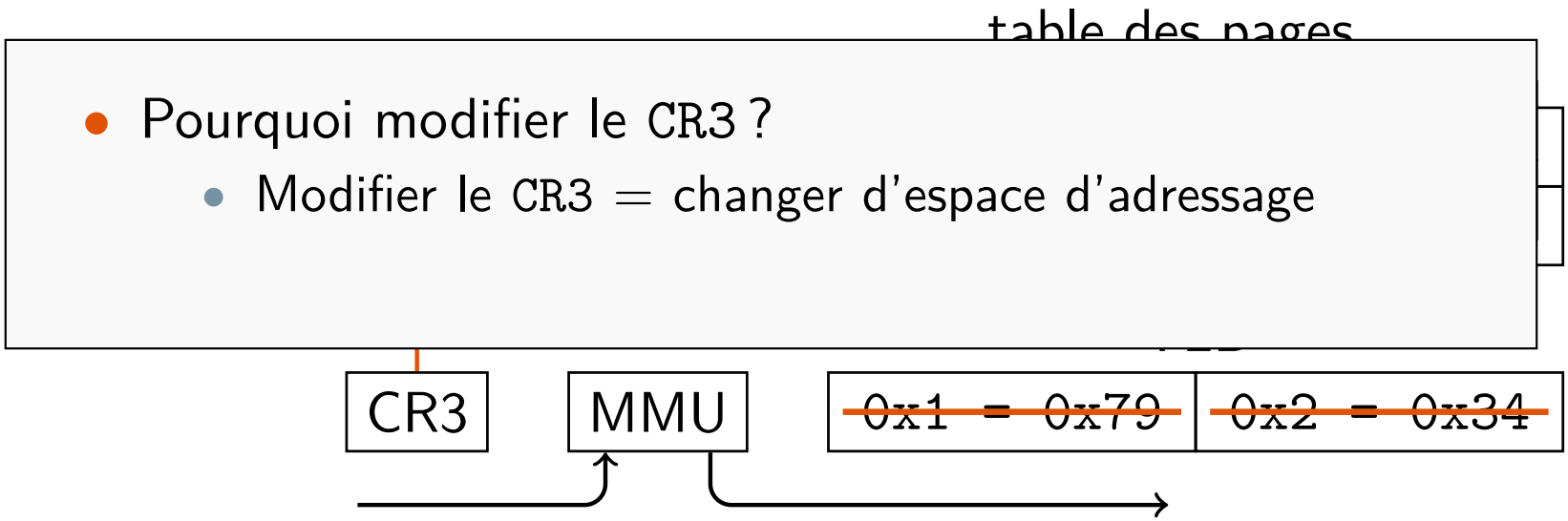
Gestion du *Translation Lookaside Buffer*

- Le TLB est automatiquement rempli à chaque traduction réussie
- Le TLB est automatiquement vidé quand le `CR3` est modifié
  - Modifier le `CR3` = changer la manière dont la MMU traduit les adresses
  - Les traductions faites précédemment sont donc caduques



Gestion du *Translation Lookaside Buffer*

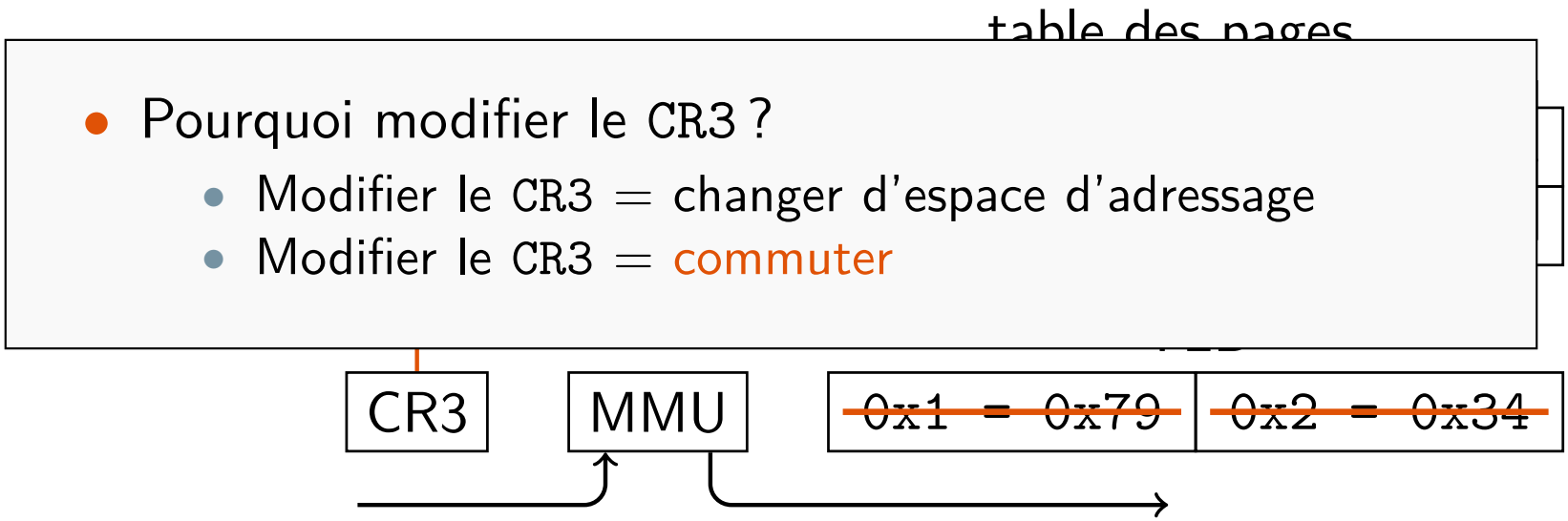
- Le TLB est automatiquement rempli à chaque traduction réussie
- Le TLB est automatiquement vidé quand le `CR3` est modifié
  - Modifier le `CR3` = changer la manière dont la MMU traduit les adresses
  - Les traductions faites précédemment sont donc caduques



Gestion du *Translation Lookaside Buffer*

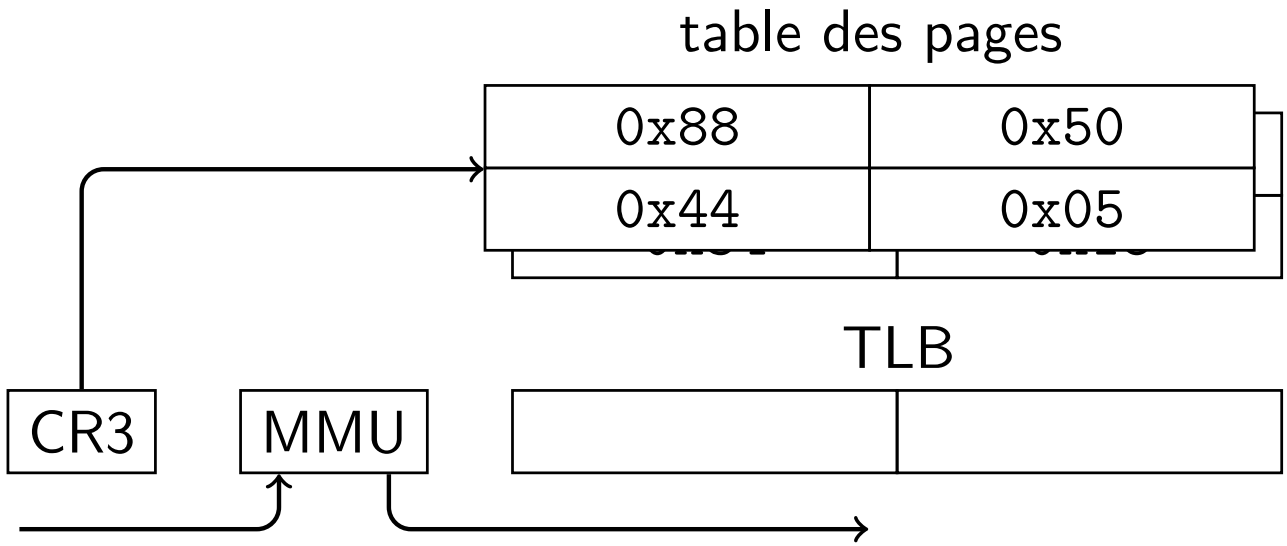
- Le TLB est automatiquement rempli à chaque traduction réussie
- Le TLB est automatiquement vidé quand le `CR3` est modifié
  - Modifier le `CR3` = changer la manière dont la MMU traduit les adresses
  - Les traductions faites précédemment sont donc caduques

- Pourquoi modifier le `CR3` ?
  - Modifier le `CR3` = changer d'espace d'adressage
  - Modifier le `CR3` = commuter



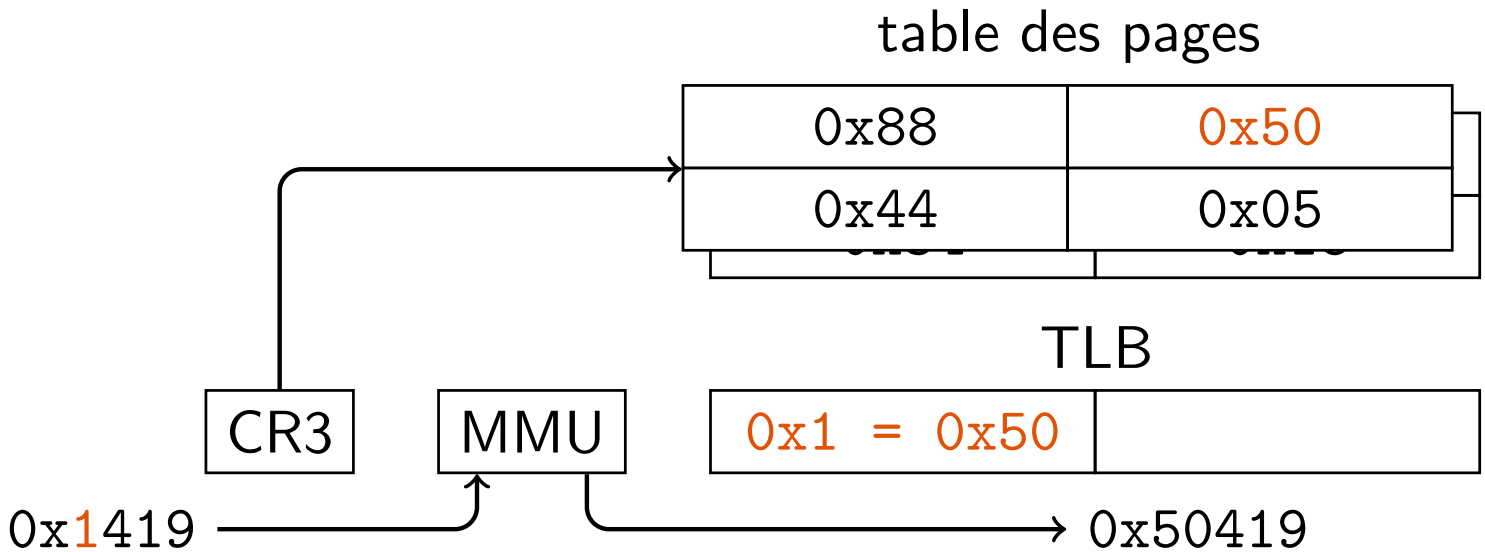
Gestion du *Translation Lookaside Buffer*

- Le TLB est automatiquement rempli à chaque traduction réussie
- Le TLB est automatiquement vidé quand le `CR3` est modifié
  - Modifier le `CR3` = changer la manière dont la MMU traduit les adresses
  - Les traductions faites précédemment sont donc caduques



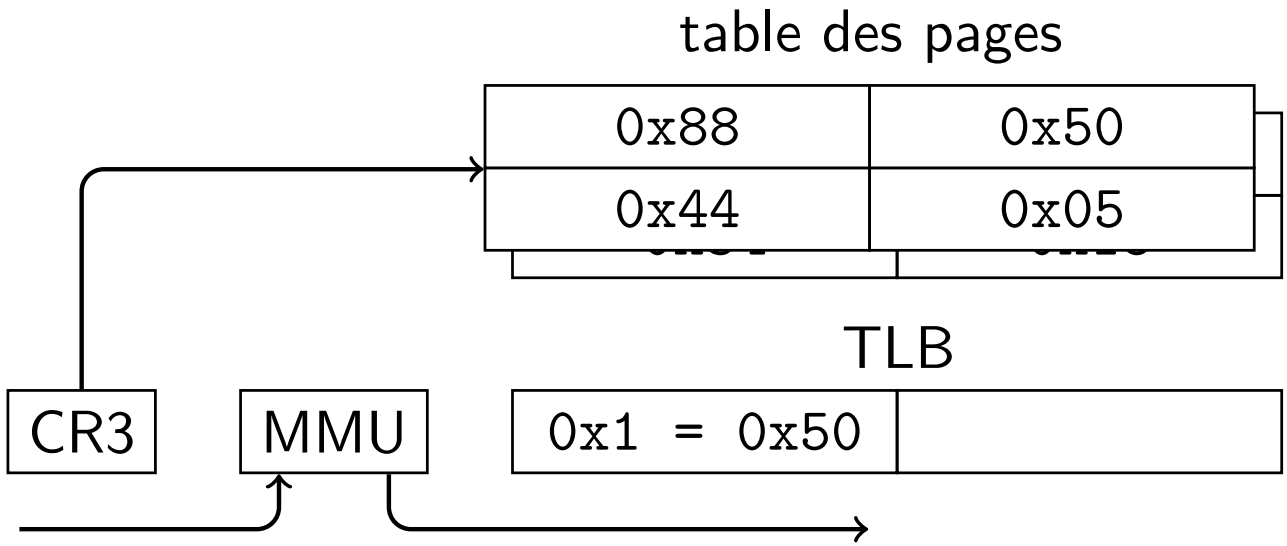
Gestion du *Translation Lookaside Buffer*

- Le TLB est automatiquement rempli à chaque traduction réussie
- Le TLB est automatiquement vidé quand le `CR3` est modifié
  - Modifier le `CR3` = changer la manière dont la MMU traduit les adresses
  - Les traductions faites précédemment sont donc caduques



Gestion du *Translation Lookaside Buffer*

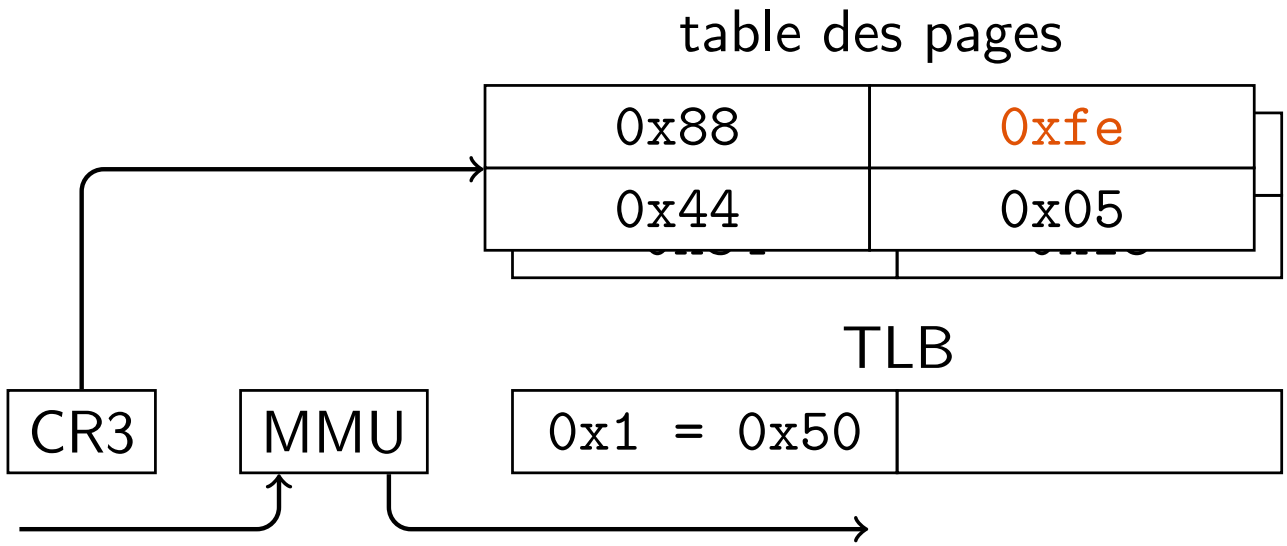
- Le TLB est automatiquement rempli à chaque traduction réussie
- Le TLB est automatiquement vidé quand le `CR3` est modifié
  - Modifier le `CR3` = changer la manière dont la MMU traduit les adresses
  - Les traductions faites précédemment sont donc caduques



- Le système peut modifier la table des pages à tout moment

Gestion du *Translation Lookaside Buffer*

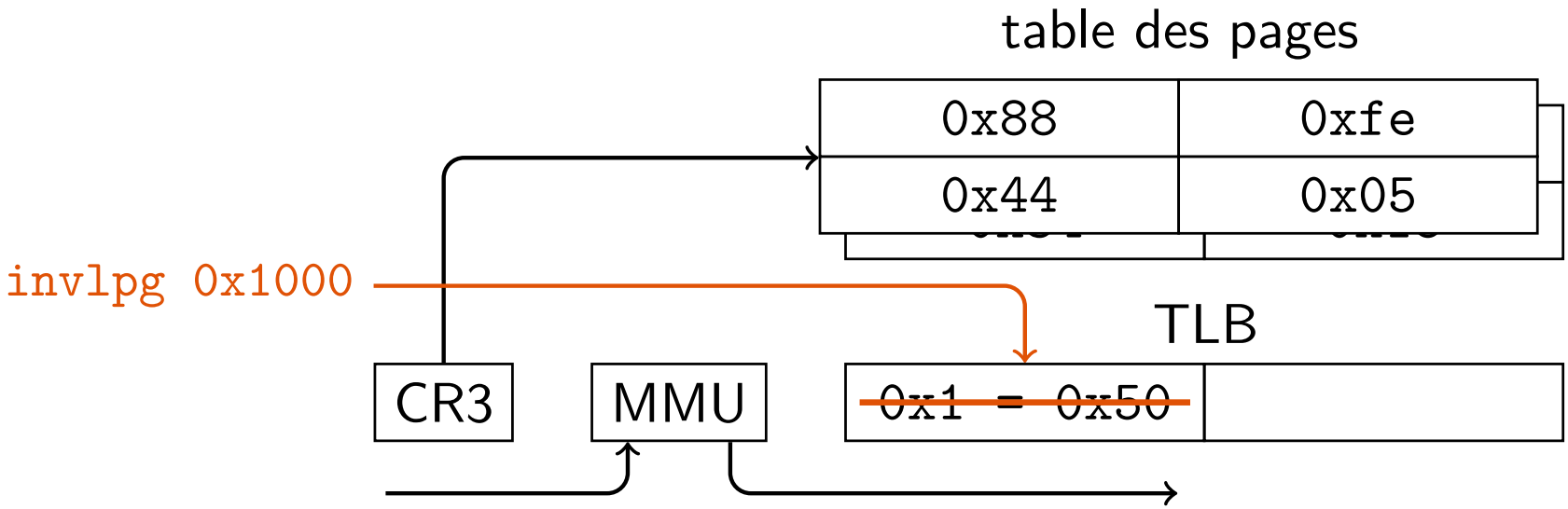
- Le TLB est automatiquement rempli à chaque traduction réussie
- Le TLB est automatiquement vidé quand le `CR3` est modifié
  - Modifier le `CR3` = changer la manière dont la MMU traduit les adresses
  - Les traductions faites précédemment sont donc caduques



- Le système peut modifier la table des pages à tout moment

Gestion du *Translation Lookaside Buffer*

- Le TLB est automatiquement rempli à chaque traduction réussie
- Le TLB est automatiquement vidé quand le `CR3` est modifié
  - Modifier le `CR3` = changer la manière dont la MMU traduit les adresses
  - Les traductions faites précédemment sont donc caduques



- Le système peut modifier la table des pages à tout moment
  - Le système doit alors invalider le TLB pour l'entrée modifiée
  - Le processeur fournit une instruction dédiée (`invlpg`)

Translation Lookaside Buffer modernes et *Process Context*

- Invalider tout le *Translation Lookaside Buffer* a un coût
  - Il faut le remplir à nouveau à partir de la nouvelle table
  - Provoque beaucoup d'accès mémoire

- Commuter $\Leftrightarrow$ changer le `CR3` $\Leftrightarrow$ invalider le TLB
  - Invalider le TLB représente une grosse part du coût d'une commutation
  - Beaucoup de temps de calcul de perdu

- Les processeurs modernes implémentent les *Process Context IDentifier*
  - Le système associe un PCID (12 bits) à chaque table des pages
  - Le PCID courant est stocké dans le `CR3` avec l'adresse de la table
  - Chaque nouvelle entrée du TLB est taguée avec le PCID courant
  - Seule les entrées du TLB avec le PCID courant sont utilisées
- Plus besoin d'invalider le TLB à la commutation

Performance et *Translation Lookaside Buffer* : résumé

- La traduction d'adresse est une opération coûteuse
  - Pour les implémentations actuelles, 4 accès mémoire sont nécessaires
- Pour économiser le coût des accès, la MMU est dotée d'un TLB
  - Le TLB est une mémoire cache, interne à la MMU
  - Quand une traduction réussie, les indices physique et virtuel sont stockés dans le TLB
  - Avant une traduction, la MMU vérifie si le TLB contient une entrée pour l'index virtuel demandé
- Le TLB a une capacité limitée
  - La MMU permet d'utiliser plusieurs tailles de page
  - Les huge pages sont plus grandes et saturent moins le TLB
- La gestion du TLB est partiellement manuelle
  - Le TLB est automatiquement vidé quand le `CR3` est modifié
  - Les entrées du TLB doivent être invalidées explicitement quand la table des pages est modifiée

Plan du cours

Mémoire virtuelle dans Linux et segmentation

- Les processeurs x86 utilisent la pagination et la segmentation
  - La segmentation est un autre mécanisme de gestion mémoire
  - Un segment est une plage d'adresse de taille arbitraire
  - À chaque segment est associé un offset
  - Le processeur transforme une adresse virtuelle en adresse linéaire en ajoutant l'offset du segment correspondant
  - Ce mécanisme est plus ancien et plus simple que la pagination
  - Ce mécanisme est aussi moins puissant (fragmentation)
- En 32 bits, la segmentation est obligatoire et la pagination optionelle
  - Linux utilise des segments : `0x0` → `0xffffff...` avec offset = 0
  - Dans Linux, on a toujours adresse virtuelle = adresse linéaire
- En 64 bits, la pagination est obligatoire et la segmentation optionelle
  - Linux n'utilise pas la segmentation

Traduction d'adresse et faute de page

- En plus de stocker l'adresse physique du niveau suivant, une entrée de table des pages stocke des bits d'information
  - Le bit **P**age **S**ize indique si le niveau pointé est une huge page
  - Le bit **N**o e**X**ecute indique si la page peut être accédée en exécution
  - Le bit **U**ser indique si la page peut être accédée depuis le userland
  - Le bit **W**ritable indique si la page peut être accédée en écriture
  - Le bit **P**resence indique si l'entrée courante est valide

- Si le processeur tente d'utiliser une entrée invalide ou avec des permissions insuffisantes
  - La traduction s'arrête et le TLB n'est pas rempli
  - La MMU signale une faute de page au processeur qui fait l'accès

Exceptions et interruptions : rappels

- En temps normal, le processeur exécute le flot d'instructions principal
  - Exécuter l'instruction courante puis aller à l'instruction suivante
  - Si l'instruction est un *jump*, aller à l'instruction indiquée

- Il existe des évènements exceptionnels
  - Évènements synchrones : division par zéro, breakpoint, ...
  - Évènements asynchrones : réception ethernet, alerte thermique, ...

- Ces évènements requièrent une intervention du processeur
  - Cette intervention est généralement indépendante du flot principal

- Le processeur peut être dérouté vers un flot d'instruction secondaire
  - Évènement synchrone $\rightarrow$ exception
    - Instruction impossible à exécuter $\rightarrow$ faute
    - Appel explicite au système $\rightarrow$ trappe
  - Évènement asynchrone $\rightarrow$ interruption
  - Au *boot*, le système associe un traitement à chaque type d'évènement

Gestion d'exception : exemple

```c
/* userland */

int main(void)
{
        long a, b, c;

        b = 14;
        c = 0;
        a = b / c;

        return 0;
}
```

```c
/* kernel */

unsigned long retry = 0;

void handle_div_by_zero(void)
{
        printk("Error\n");

        if (++retry == 2) {
                kill_current_task();
                retry = 0;
        }
}
```

Gestion d'exception : exemple

```c
/* userland */

int main(void)
{
        long a, b, c;

        b = 14;
        c = 0;
        a = b / c;

        return 0;
}
```

```c
/* kernel */

unsigned long retry = 0;

void handle_div_by_zero(void)
{
        printk("Error\n");

        if (++retry == 2) {
                kill_current_task();
                retry = 0;
        }
}
```

Gestion d'exception : exemple

```c
/* userland */

int main(void)
{
        long a, b, c;

        b = 14;
        c = 0;
        a = b / c;

        return 0;
}
```

```c
/* kernel */

unsigned long retry = 0;

void handle_div_by_zero(void)
{
        printk("Error\n");

        if (++retry == 2) {
                kill_current_task();
                retry = 0;
        }
}
```

- La division par zéro est une faute → déroutage

Gestion d'exception : exemple

```c
/* userland */

int main(void)
{
        long a, b, c;

        b = 14;
        c = 0;
        a = b / c;

        return 0;
}
```

```c
/* kernel */

unsigned long retry = 0;

void handle_div_by_zero(void)
{
        printk("Error\n");

        if (++retry == 2) {
                kill_current_task();
                retry = 0;
        }
}
```

- La division par zéro est une faute → déroutage
- Le traitement est exécuté, puis le processeur reprend le flot principal

Gestion d'exception : exemple

```c
/* userland */

int main(void)
{
        long a, b, c;

        b = 14;
        c = 0;
        a = b / c;

        return 0;
}
```

```c
/* kernel */

unsigned long retry = 0;

void handle_div_by_zero(void)
{
        printk("Error\n");

        if (++retry == 2) {
                kill_current_task();
                retry = 0;
        }
}
```

- La division par zéro est une faute → déroutage
- Le traitement est exécuté, puis le processeur reprend le flot principal
- L'exécution reprend avant l'instruction fautive

Gestion d'exception : exemple

```c
/* userland */

int main(void)
{
        long a, b, c;

        b = 14;
        c = 0;
        a = b / c;

        return 0;
}
```

```c
/* kernel */

unsigned long retry = 0;

void handle_div_by_zero(void)
{
        printk("Error\n");

        if (++retry == 2) {
                kill_current_task();
                retry = 0;
        }
}
```

- La division par zéro est une faute → déroutage
- Le traitement est exécuté, puis le processeur reprend le flot principal
- L'exécution reprend avant l'instruction fautive

Mémoire virtuelle et faute de page : résumé

- Le processeur peut tenter d'accéder à une adresse invalide
  - L'adresse virtuelle ne correspond à aucune adresse physique
  - L'adresse virtuelle n'est pas accessible en écriture / exécution
  - L'adresse virtuelle n'est accessible qu'en mode système
- Dans ce cas, la MMU génère une faute de page pour ce processeur
  - Le processeur est dérouté vers le traitement associé
  - Ce traitement est définit au *boot* par le système
- Le système peut alors tenter de résoudre la cause de la faute
  - Associer une adresse physique à l'adresse virtuelle demandée
  - Dérouter le programme fautif (via un signal : `man sigaction`)
  - Tuer le programme fautif

Allocation paresseuse

- Un traitement est dit paresseux s'il n'est effectué qu'au moment où on en a besoin.
- En particulier, un traitement paresseux n'est jamais exécuté si son résultat n'est pas utilisé.

- Exemple : `if (func_a() && func_b()) { ... }`
  - L'opérateur && est "à évaluation paresseuse"
  - `func_b()` n'est appellée que quand on connait le résultat de `func_a()`
  - Si `func_a()` retourne `0`, alors `func_b()` n'est pas appellée

- Sous Linux, l'allocation mémoire est paresseuse
  - Conséquence : `malloc(1 << 40)` retourne une adresse utilisable
  - La mémoire ne sera allouée quand l'adresse sera utilisée

Allocation d'adresses virtuelles

- Le noyau Linux gère les adresses virtuelles et physiques de manière indépendante
- À chaque processus est lié un arbre des adresses virtuelles utilisables

- Lors d'un appel à `mmap()` (appel système derrière `malloc()`)
  - Le système trouve une plage d'adresses libre de la taille demandée entre 0 et 2^{48}
  - Une nouvelle feuille correspondant à cette plage d'adresses est insérée dans l'arbre
  - La première adresse de cette plage d'adresses est retournée

- Aucune mémoire physique n'est allouée dans l'opération
- La table des pages du processus n'est pas modifiée
- Plusieurs processus peuvent utiliser les mêmes adresses virtuelles

Allocation d'adresses physiques

```
/* userland */
char *a, *b;

a = malloc(0x3000);
a[0] = 0;

b = NULL;
b[0] = 0;
```

```
/* kernel */

void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

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

        paddr = allocate_page();
        map_page(vaddr, paddr);
}
```

Allocation d'adresses physiques

```c
/* userland */
char *a, *b;

a = malloc(0x3000);
a[0] = 0;

b = NULL;
b[0] = 0;
```

```c
/* kernel */

void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

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

        paddr = allocate_page();
        map_page(vaddr, paddr);
}
```

- Pour la MMU, les adresses retournées par `mmap()` sont invalides

Allocation d'adresses physiques

```c
/* userland */
char *a, *b;

a = malloc(0x3000);
a[0] = 0;

b = NULL;
b[0] = 0;
```

```c
/* kernel */

void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

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

        paddr = allocate_page();
        map_page(vaddr, paddr);
}
```

- Pour la MMU, les adresses retournées par `mmap()` sont invalides
  - Un premier accès à ces adresses provoque une faute de page

Allocation d'adresses physiques

```c
/* userland */
char *a, *b;

a = malloc(0x3000);
a[0] = 0;

b = NULL;
b[0] = 0;
```

```c
/* kernel */

void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

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

        paddr = allocate_page();
        map_page(vaddr, paddr);
}
```

- Pour la MMU, les adresses retournées par `mmap()` sont invalides
  - Un premier accès à ces adresses provoque une faute de page
- Pour le système ces adresses sont légitimes
  - La faute de page est gérée par une allocation d'adresses physiques
  - L'adresse virtuelle fautive est mappée

Allocation d'adresses physiques

```c
/* userland */
char *a, *b;

a = malloc(0x3000);
a[0] = 0;

b = NULL;
b[0] = 0;
```

```c
/* kernel */

void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

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

        paddr = allocate_page();
        map_page(vaddr, paddr);
}
```

- Pour la MMU, les adresses retournées par `mmap()` sont invalides
  - Un premier accès à ces adresses provoque une faute de page
- Pour le système ces adresses sont légitimes
  - La faute de page est gérée par une allocation d'adresses physiques
  - L'adresse virtuelle fautive est mappée
  - Le contrôle est ensuite redonné au processus

Allocation d'adresses physiques

```c
/* userland */
char *a, *b;

a = malloc(0x3000);
a[0] = 0;

b = NULL;
b[0] = 0;
```

```c
/* kernel */

void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

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

        paddr = allocate_page();
        map_page(vaddr, paddr);
}
```

- Pour la MMU, les adresses illégitimes sont également invalides

Allocation d'adresses physiques

```
/* userland */
char *a, *b;

a = malloc(0x3000);
a[0] = 0;

b = NULL;
b[0] = 0;
```

```
/* kernel */

void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

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

        paddr = allocate_page();
        map_page(vaddr, paddr);
}
```

- Pour la MMU, les adresses illégitimes sont également invalides
  - Un premier accès à ces adresses provoque une faute de page

Allocation d'adresses physiques

```c
/* userland */
char *a, *b;

a = malloc(0x3000);
a[0] = 0;

b = NULL;
b[0] = 0;
```

```c
/* kernel */

void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

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

        paddr = allocate_page();
        map_page(vaddr, paddr);
}
```

- Pour la MMU, les adresses illégitimes sont également invalides
  - Un premier accès à ces adresses provoque une faute de page
- Pour le système ces adresses sont également invalides
  - La faute de page est gérée par un envoi de signal `SIGSEGV`

Allocation paresseuse et *First Touch* : résumé

- Dans Linux, l'allocation mémoire est paresseuse
- La mémoire virtuelle est allouée pendant l'appel à `mmap()`
  - Le système trouve une plage d'adresse virtuelle inutilisée pour le processus courant
  - Aucune mémoire physique n'est allouée pendant l'appel à `mmap()`
- La mémoire physique est allouée à la première touche d'une page
  - La mémoire physique est allouée page par page
  - Les pages sont allouées depuis le *pool* de mémoire commun à tous les processus

Appel système et adressage du noyau

- Lors d'un appel système, le processeur bascule en mode noyau
  - Soit via une trappe (exception)
  - Soit via un mécanisme matériel dédié (`syscall` et `sysret`)

- Le processeur exécute alors une fonction du noyau
  - Quelle est l'adresse physique de cette fonction ?
  - Quelle est l'adresse virtuelle de cette fonction ?

- Certains appels système prennent des paramètres par adresse (`write()`)
  - Ces adresses sont-elles valides en mode noyau ?

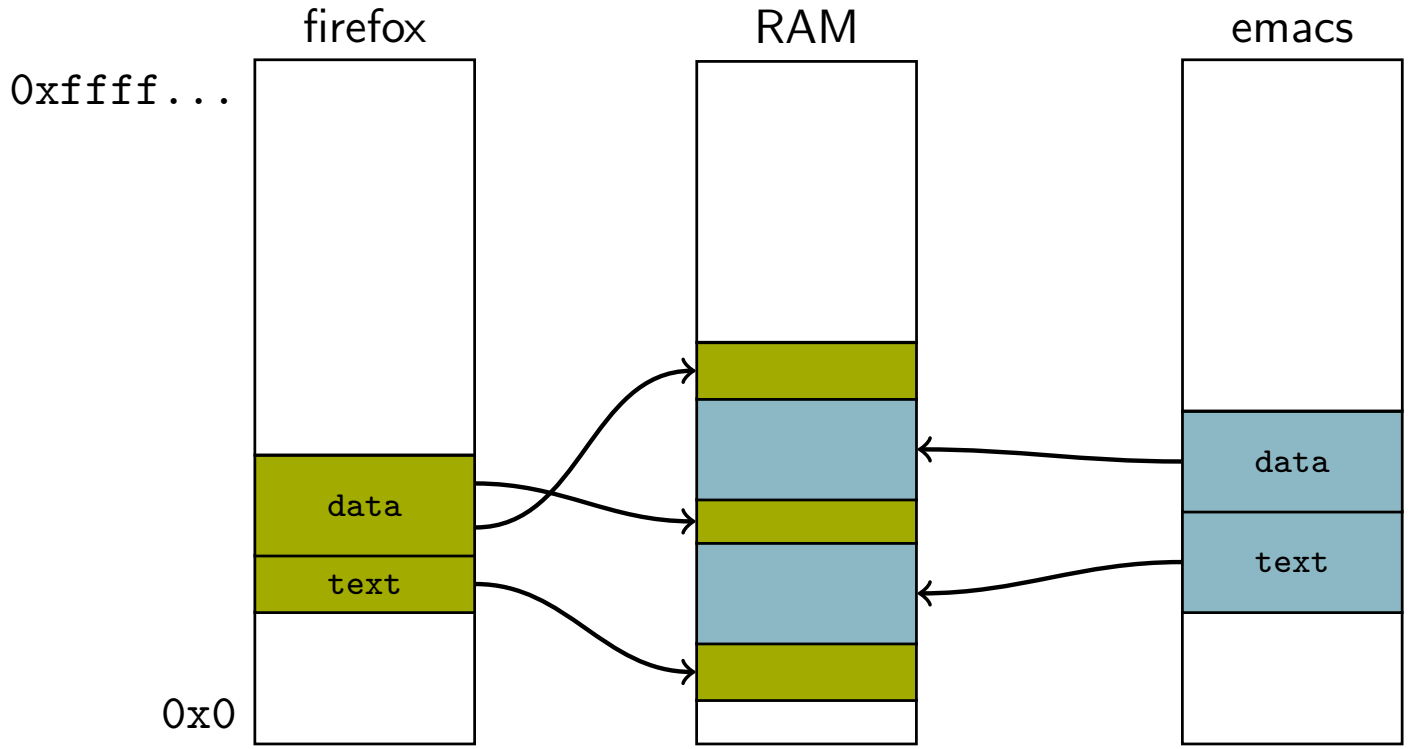
Appel système et adressage du noyau

- Lors d'un appel système, le processeur bascule en mode noyau
  - Soit via une trappe (exception)
  - Soit via un mécanisme matériel dédié (`syscall` et `sysret`)

- Le processeur exécute alors une fonction du noyau
  - Quelle est l'adresse physique de cette fonction ?
  - Quelle est l'adresse virtuelle de cette fonction ?

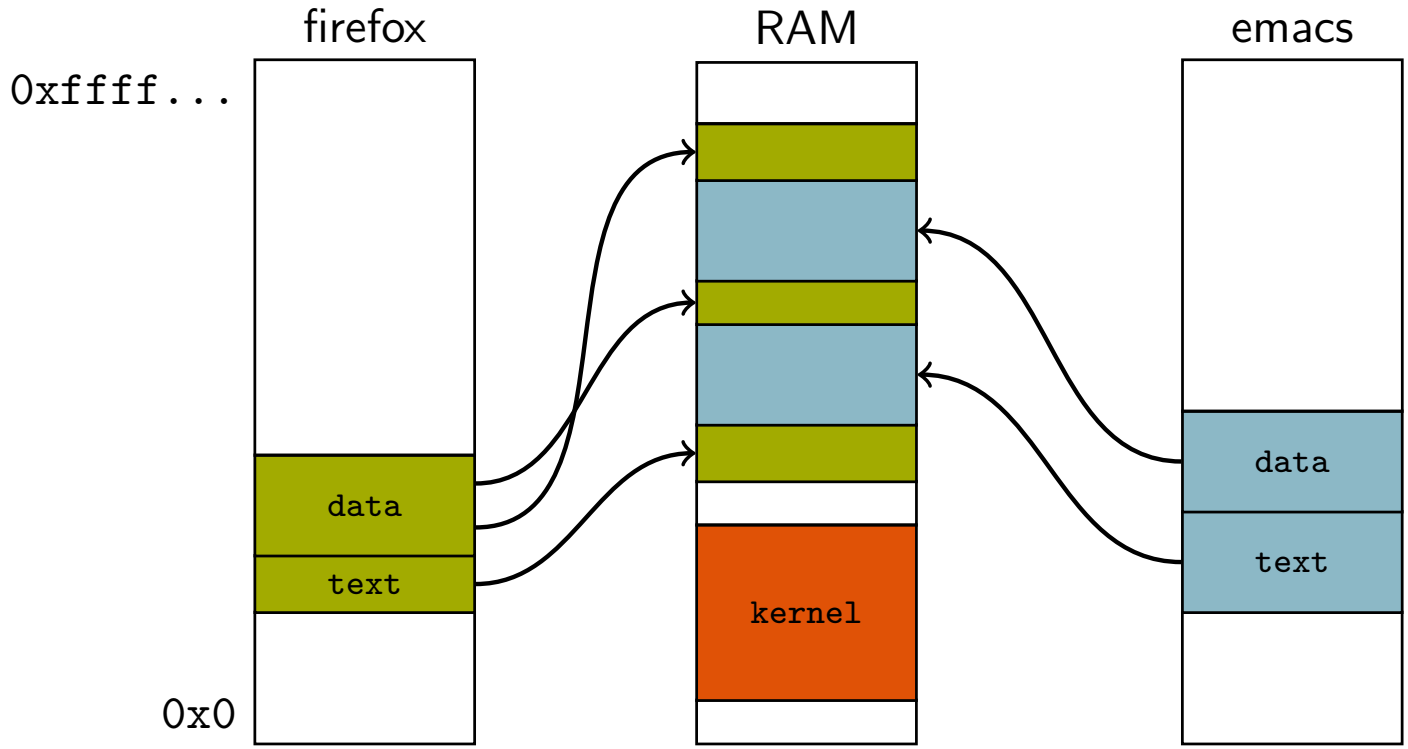
- Certains appels système prennent des paramètres par adresse (`write()`)
  - Ces adresses sont-elles valides en mode noyau ?

Adressage virtuel du noyau



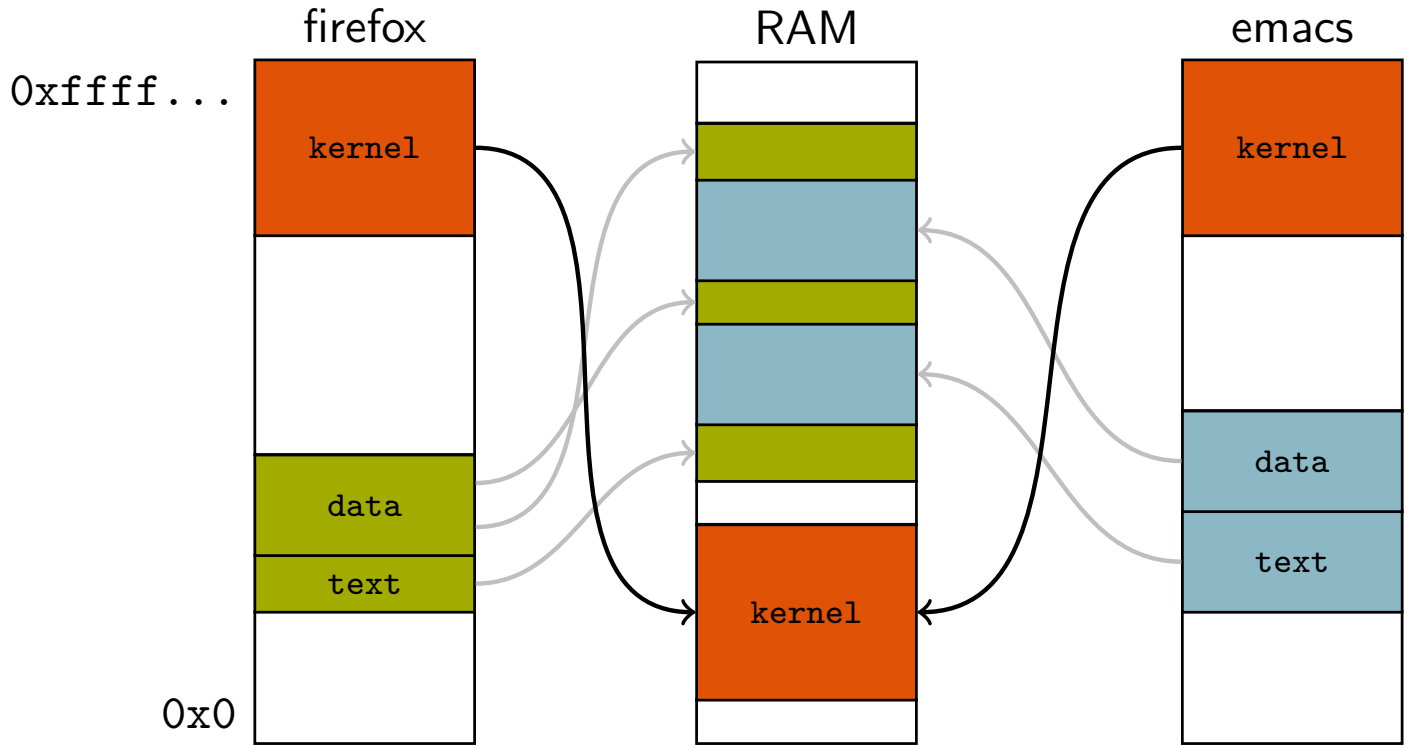
- Les sections de code, données, etc . . . sont mappées sur les adresses physiques différentes.

Adressage virtuel du noyau



- Les sections de code, données, etc . . . sont mappées sur les adresses physiques différentes.
- Le noyau Linux (code + données) est chargé au *boot* dans les adresses physiques basses

Adressage virtuel du noyau



- Les sections de code, données, etc . . . sont mappées sur les adresses physiques différentes.
- Le noyau Linux (code + données) est chargé au *boot* dans les adresses physiques basses
- Le noyau Linux est mappé dans chaque processus dans des adresses virtuelles hautes

Adressage du noyau : résumé

- Le noyau (code + données) est mappé dans chaque processus
  - Toujours mappé aux mêmes adresses virtuelles
  - Ses pages sont protégées, inaccessible en mode utilisateur
- Pendant un appel système, la table des pages n'est pas changée
  - Évite de vider le TLB $\rightarrow$ meilleurs performances
  - Le matériel n'est pas prévu pour ça
- Linux peut accéder simplement à la mémoire du processus

Conclusion : adresses virtuelles et physiques

- Il existe deux types d'adresse : virtuelle et physique
- Les adresses physiques sont les adresses vues par la mémoire
  - Une adresse physique correspond à une position en mémoire et *vice versa*
  - Le nombre d'adresses physiques est déterminé par la capacité en RAM de l'ordinateur
- Les adresses virtuelles sont les adresses vues par le processeur
  - À un instant donnée et pour un cœur donnée, une adresse virtuelle correspond à zéro ou une adresse physique
  - À un instant donnée et pour un cœur donnée, une adresse physique correspond à zéro, une ou plusieurs adresses virtuelles
  - Le nombre d'adresse virtuelles est déterminé par l'architecture du processeur
- La correspondance entre adresses virtuelles et physiques est définie par le système d'exploitation
  - Cette correspondance peut être modifiée à tout moment
  - Généralement, il y a une correspondance par processus

Conclusion : *Memory Management Unit*

- Un circuit spécialisé, la *Memory Management Unit* traduit les adresses virtuelles en adresses physiques à chaque accès par le processeur
  - La MMU traduit en fonction de la table des pages indiquée par le système
  - Les traductions réussie sont stockées dans le *Translation Lookaside Buffer*

- La table des pages est une structure arborescente
  - Stockée en mémoire
  - Renseignée par le système d'exploitation
  - Indexée par adresse virtuelle
  - Pointe vers des adresses physiques

- Le *Translation Lookaside Buffer* est une mémoire cache spécialisée pour la traduction d'adresse
  - Évite à la MMU d'accéder à la mémoire à chaque traduction
  - Capacité très limitée
  - Entièrement invalidée par un changement de table des pages
  - Invalidée partiellement par le système par instruction explicite

Conclusion : adressage dans Linux

- Linux alloue sa mémoire de manière paresseuse
  - Un appel à `mmap` retourne une plage d'adresses virtuelles non mappées
  - Au premier accès à une zone virtuelle, la MMU avertit le noyau Linux
  - Le noyau alloue et associe une adresse physique à l'adresse virtuelle fautive

- Linux est mappé dans l'espace d'adresses virtuelles de tous les processus
  - Le noyau est toujours mappé aux mêmes adresses
  - Les adresses virtuelles du noyau sont protégées des accès utilisateur