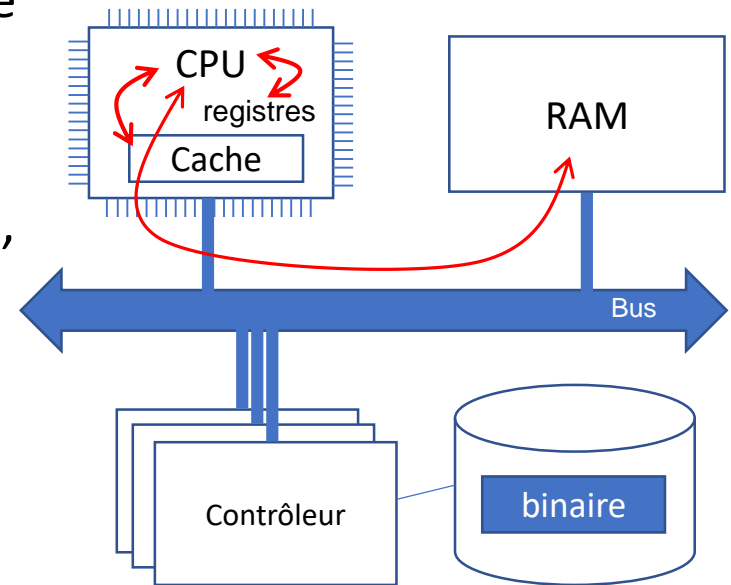


# Cours 6 : Gestion de la mémoire

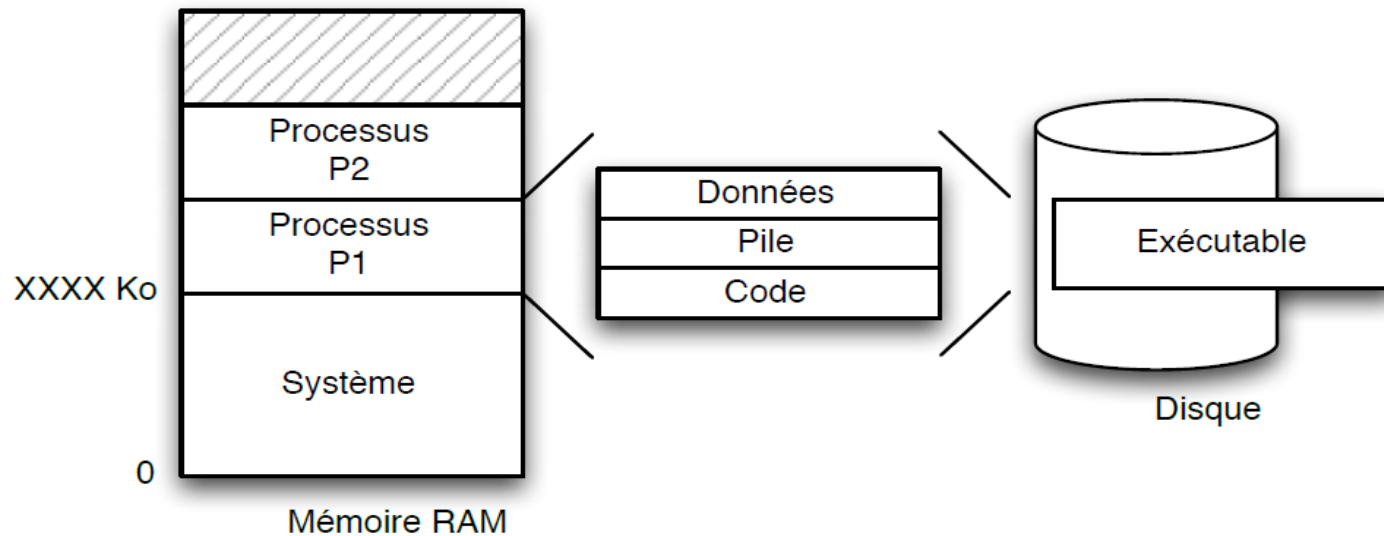
# Gestion de la mémoire

- Un programme doit être chargé en mémoire principale pour pouvoir être exécuté.
- Le CPU n'accède qu'à la mémoire principale, au cache et aux registres.
- Nécessité d'offrir des mécanismes de protection de mémoire pour assurer l'exécution correcte des programmes.
- 2 modèles d'organisation mémoires : linéaire et virtuel



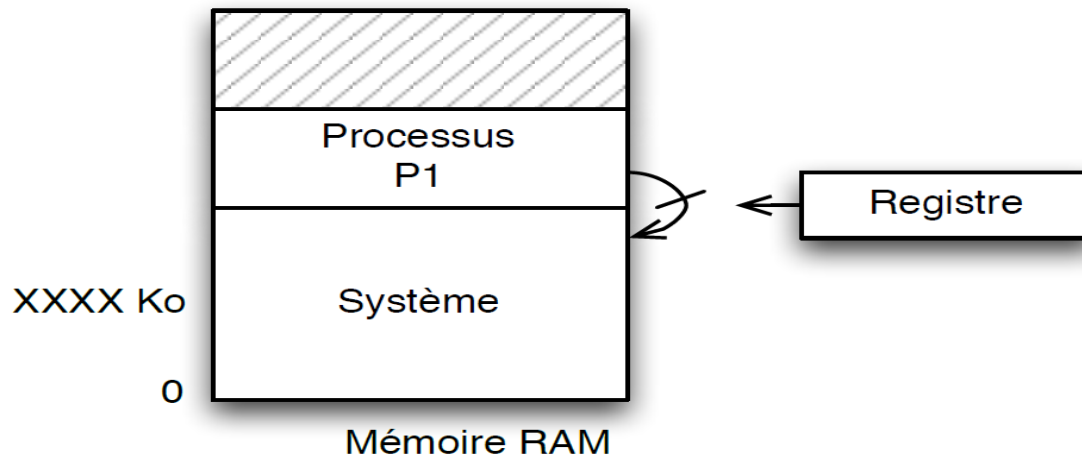
# Mémoire Linéaire

- Les processus occupent des **espaces contigus** au dessus du système
- **Totalité** du code, des données et de la pile chargés en mémoire



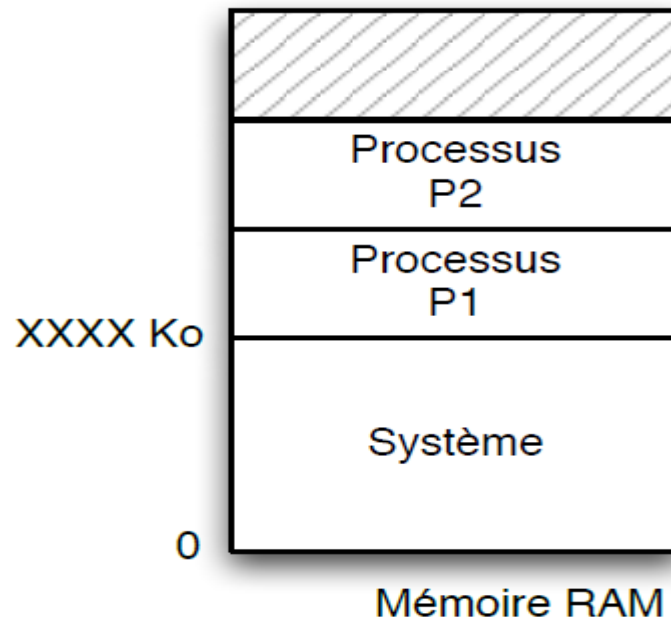
# Mono-programmation

- **Un seul** processus en mémoire au dessus du système.
- MMU (Memory Management Unit)
  - Partie du processeur qui gère la mémoire
- MMU utilise un registre barrière pour (in)valider l'accès
- Coûteux – *Swap* à chaque commutation



# Multi-programmation

- Plusieurs processus partagent la mémoire
  - Ils sont relogeables (leur emplacement peut changer à chaque exécution)
- Conséquence : adresses **relatives** à l'espace de travail

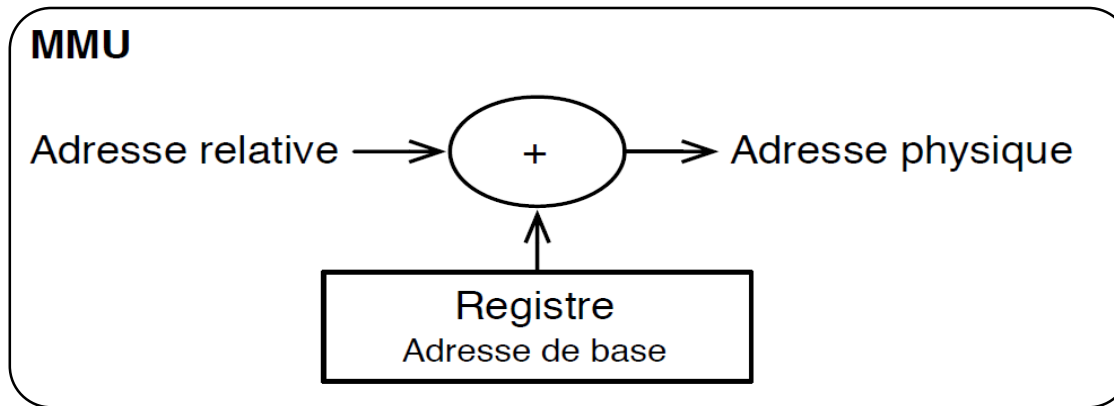


# Multi-programmation

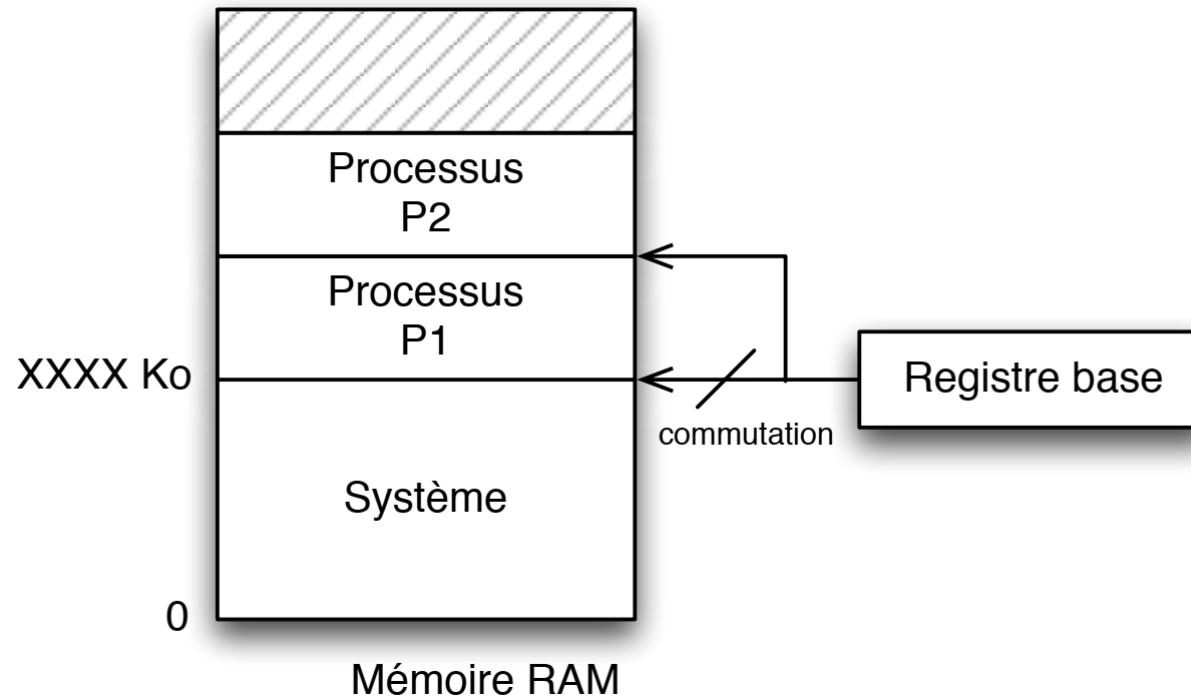
- Le code exécutable possède des adresses logiques

=> Pendant l'exécution, traduction entre les adresses logiques et physiques effectuée **par la MMU**

- La MMU gère la traduction des adresses relatives vers des adresses physiques

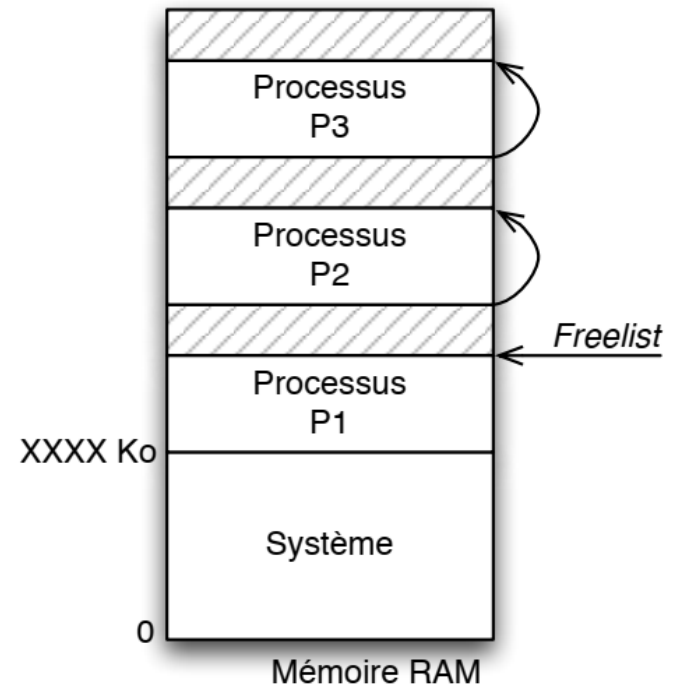


# Commutation



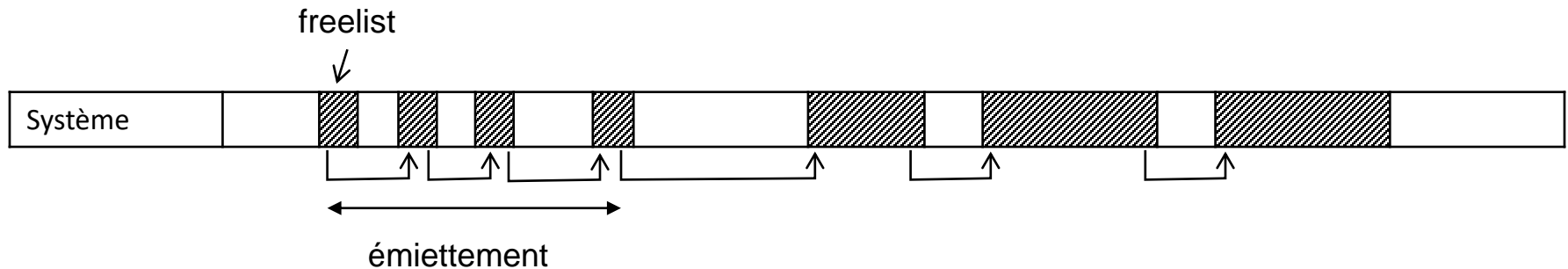
# Allocation mémoire contiguë

- Le système maintient une liste des emplacements libres = freelist
- Stratégies pour sélectionner un emplacement libre :
  - **First Fit**
    - allouer le **premier** emplacement suffisamment gros.
  - **Best Fit**
    - allouer le **plus petit** emplacement.
  - **Worst Fit**
    - allouer le **plus grand** emplacement



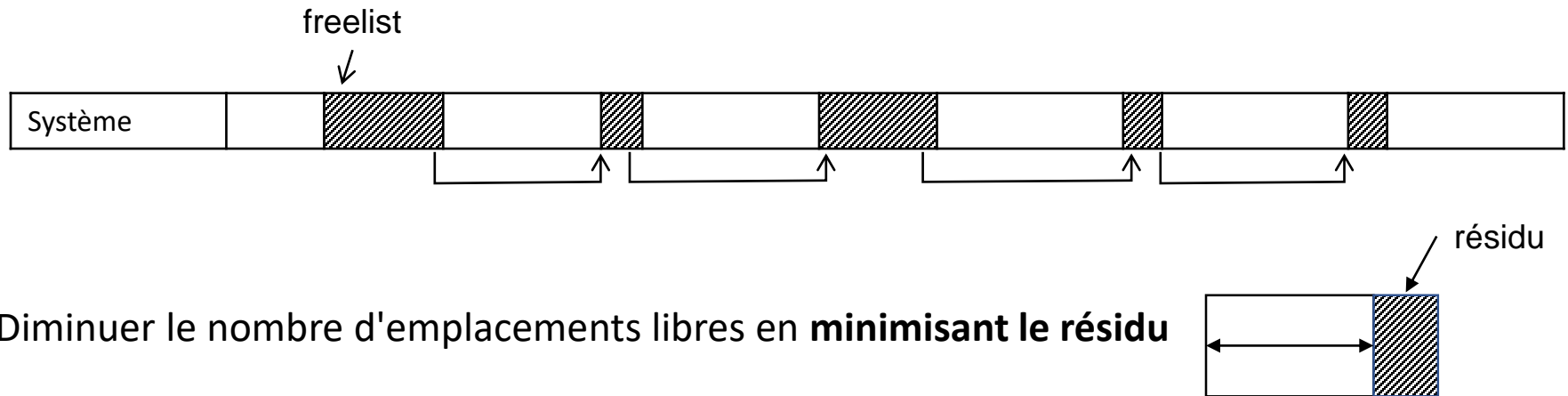


# Allocation first fit



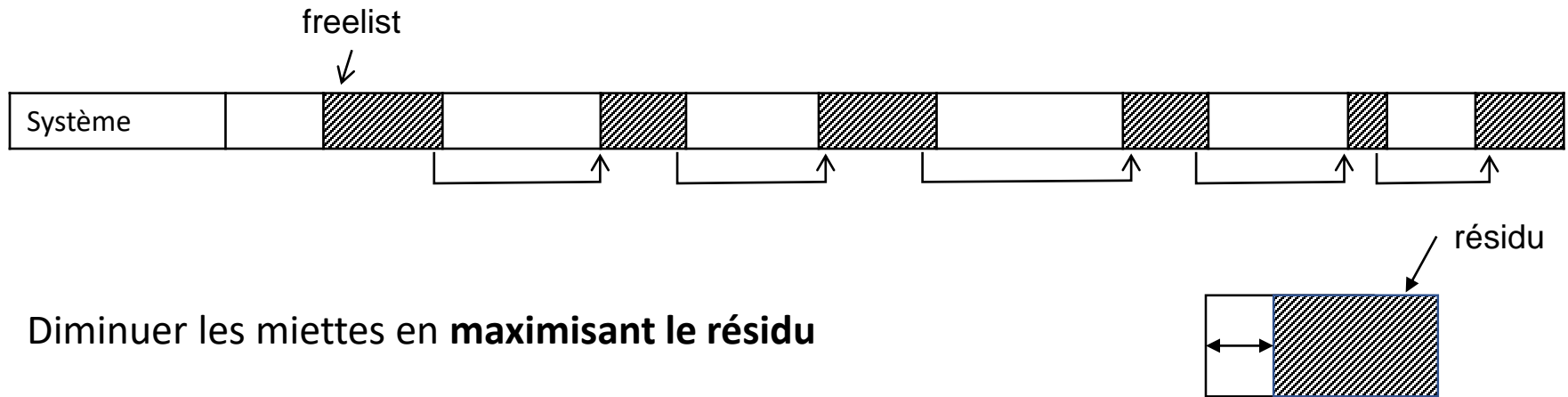
- + Simple
- Tendance à l'émiettement au début de la mémoire (miette : emplacement trop petit)
- Ne minimise pas le nombre de trous (emplacements)

# Allocation best fit



- ⊕ Meilleure utilisation de la mémoire (moins de "trous", *freelist* plus petite) ;
- ⊖ Plus complexe : parcourir toute la liste
- ⊖ Tendence à un fort émiettement (si taille du résidu > 0)

# Allocation worst fit



Diminuer les miettes en **maximisant le résidu**



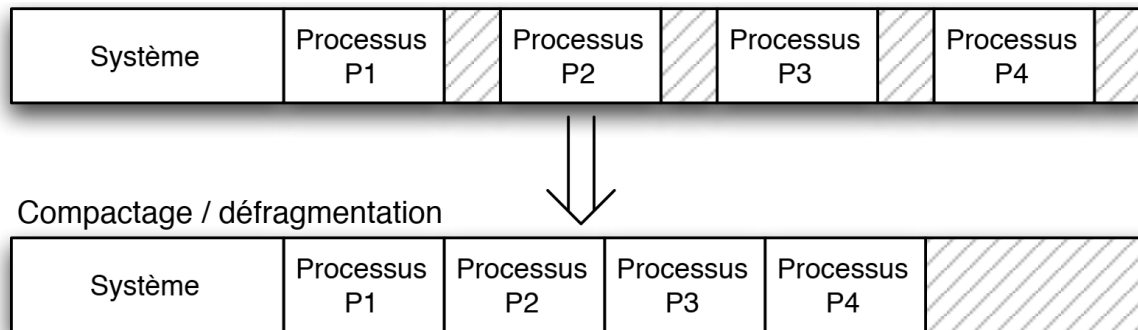
Moins de miettes



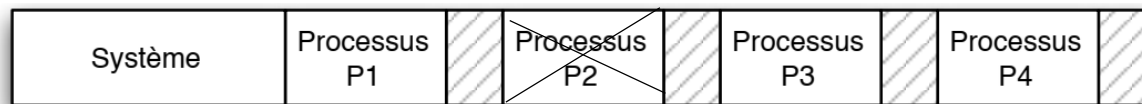
Plus de trous dans la mémoire

# Fragmentation externe

- Suffisamment de mémoire pour satisfaire une demande mais que cet espace n'est pas contiguë.
- Solutions
  1. Compactage : Réorganiser les blocs alloués pour rassembler toute la mémoire libre => freelist = 1 seul emplacement

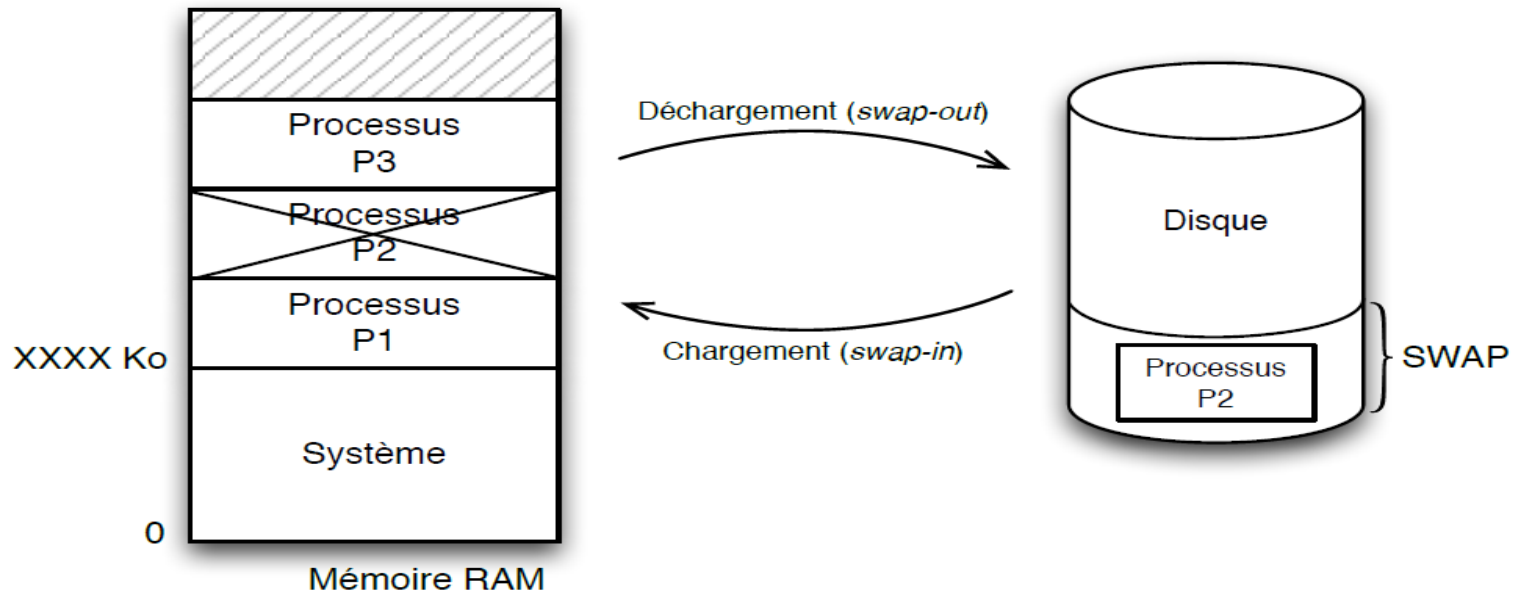


2. Swapping : décharger sur disque des processus pour créer de nouveaux emplacements libres



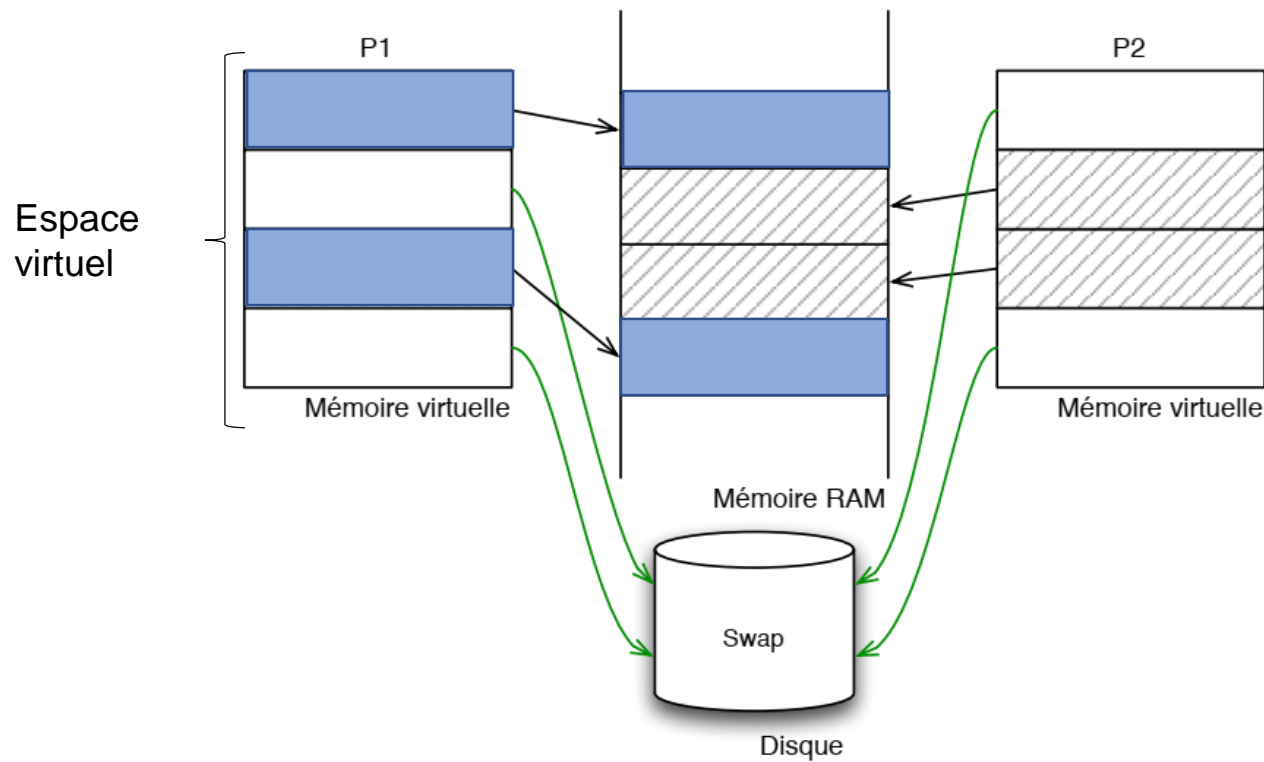
# Swapping en cas de saturation mémoire

- Nombre limité de processus simultanément en mémoire
- Solution : utilisation de mémoire secondaire
  - Zone de disque (*swap*) définie comme extension de la mémoire
  - Solution très coûteuse (chargement/déchargement de processus en entier)



# Mémoire Virtuelle

- Principe : processus non contigu et partiellement en mémoire



# Structuration de la mémoire virtuelle

---

- **Segmentation**

- Mémoire du processus découpée en segments de mots mémoire de **taille variable** et consécutifs

- **Pagination**

- Mémoire du processus découpée en blocs consécutifs de **taille fixe** et unique

- Structuration Hybride : **Segmentation paginée**

- Combinaison des deux approches

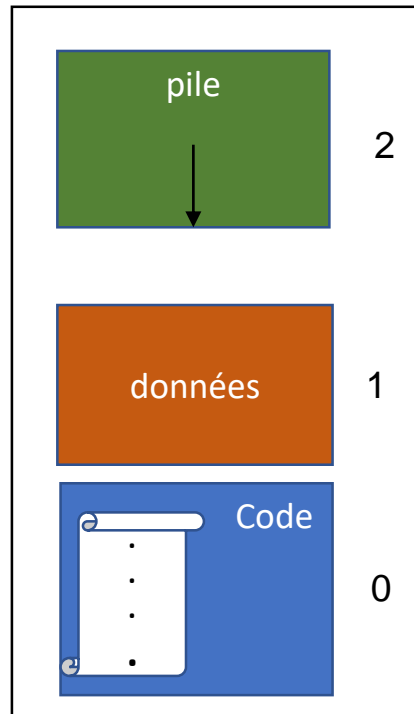
# Segmentation

---

- Processus = ensemble de segments contigus de **taille variable**.
  - Exemple:
    - code
    - données
    - pile
- Les segments sont numérotés
- Problème de fragmentation externe lors de l'allocation contiguë des segments



# Segmentation - Principes



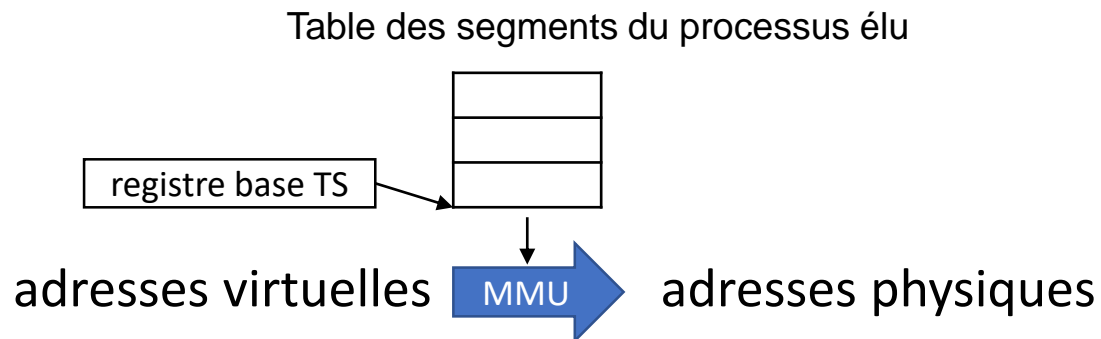
Processus P (espace virtuel)

Mémoire physique (RAM)

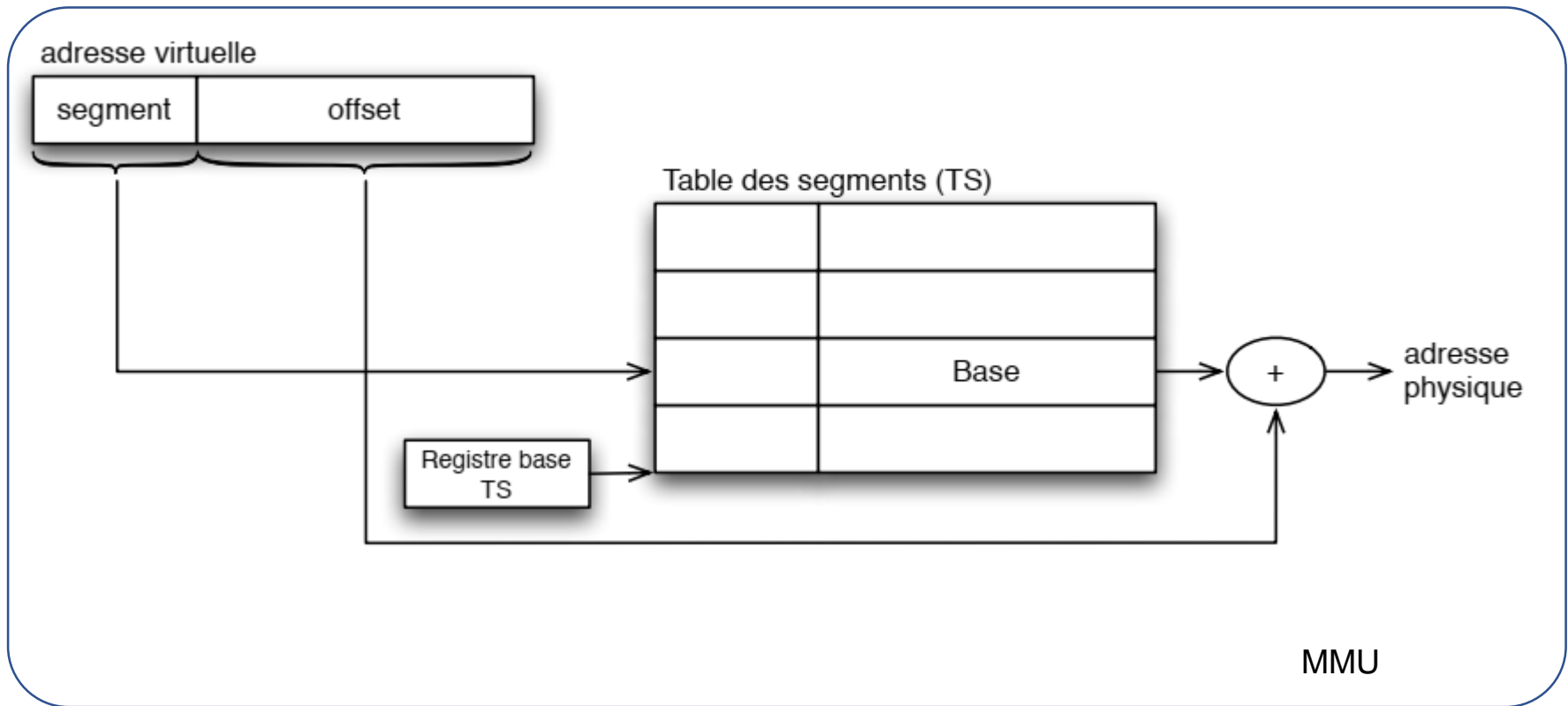


# Segmentation - traduction d'adresses

- adresse virtuelle :  
<numéro segment, déplacement dans segment (offset) >
- Prise en charge par le matériel (MMU)
- 1 table par processus = la **table des segments** désigné par un registre

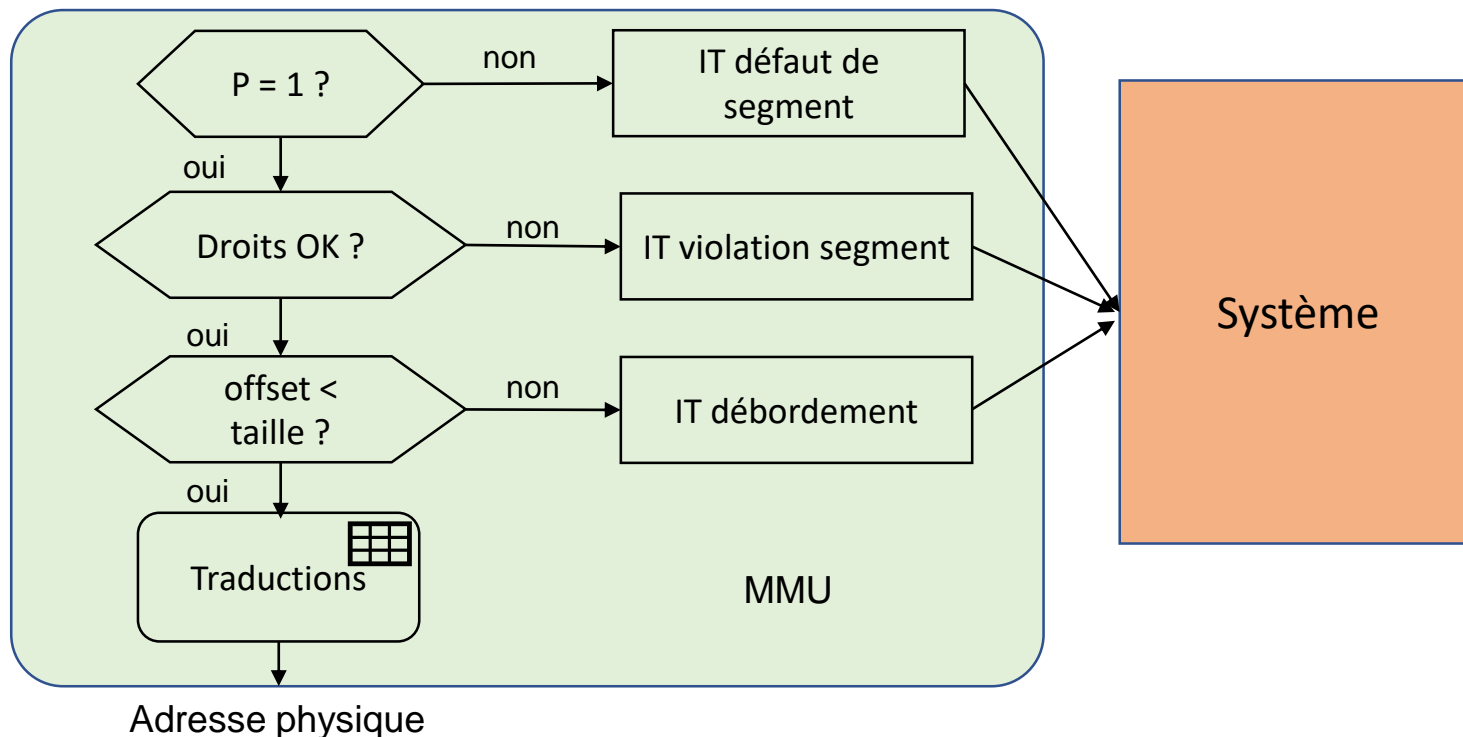


# Segmentation - traduction d'adresses

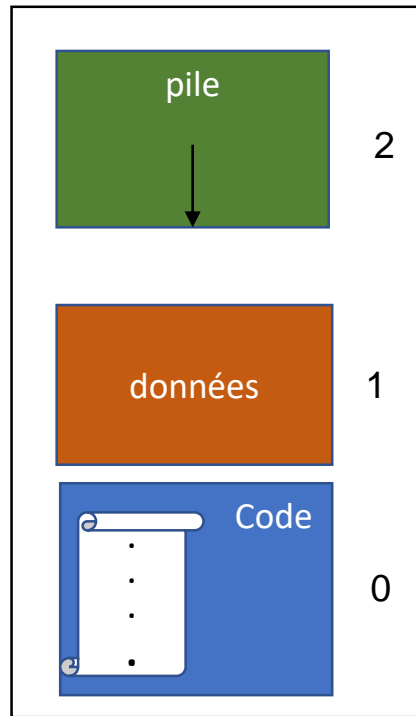


# Segmentation - MMU

- Entrée de la table des segments <P, Droits, Taille, Base>
  - P : présence (0 : non présent, 1 : présent) ;
  - Droits : r (Read ), w (Write), x (eXecutable).
- La MMU (processeur) vérifie pour une adresse <segment, offset>



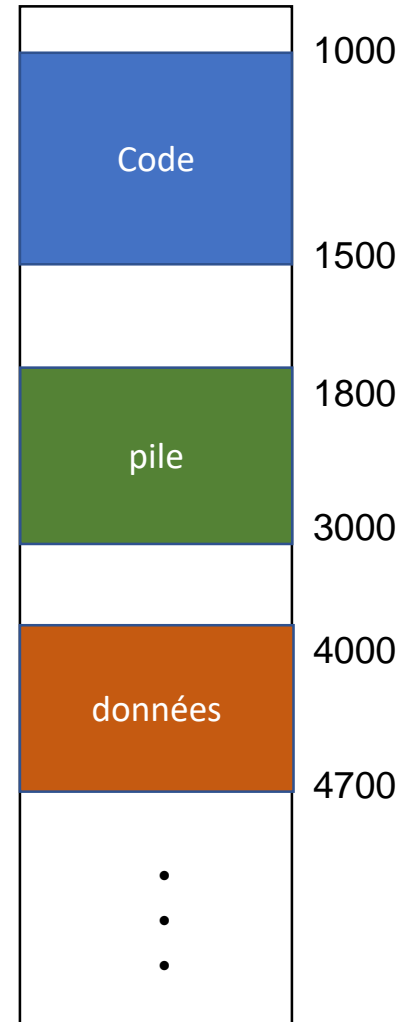
# Segmentation - table des segments



Processus P (espace virtuel)

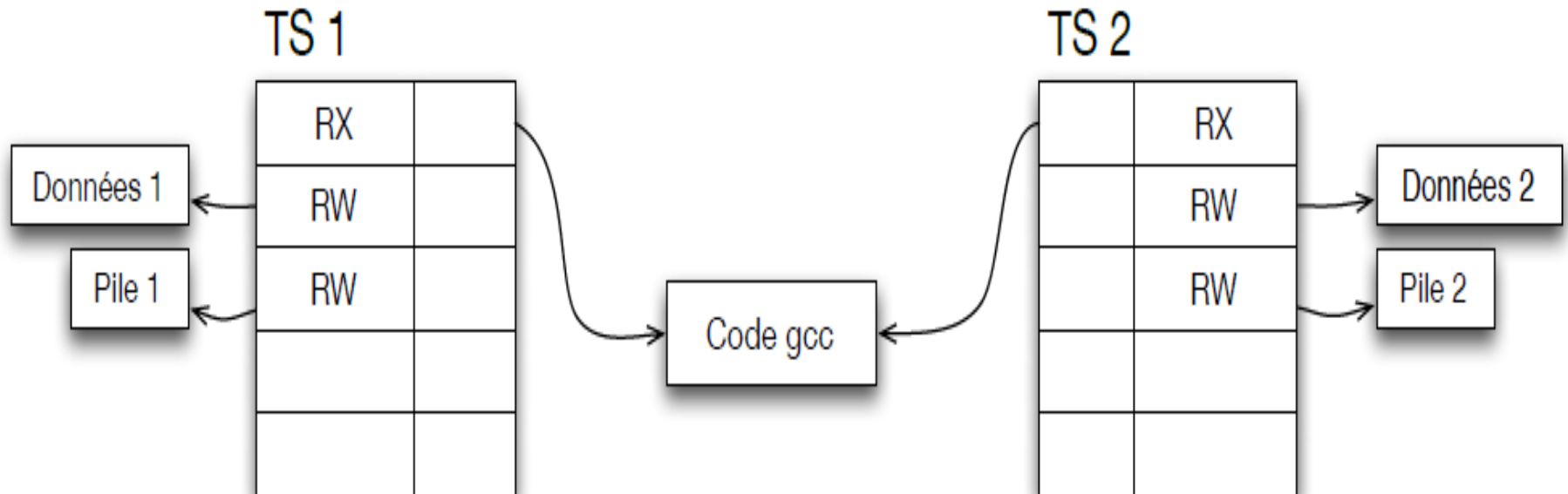
	P	Droits	Base	Taille
0	1	RX	1000	500
1	1	RW	4000	700
2	1	RW	1800	1200

Mémoire physique (RAM)



# Segmentation - partage

## Partage de segment



# Segmentation - Bilan

---

- ⊕ Permet la protection et la partage de zone (eg. code)
- ⊖ Limitation : allocation contiguë des segments en mémoire  
⇒ fragmentation externe

# Pagination

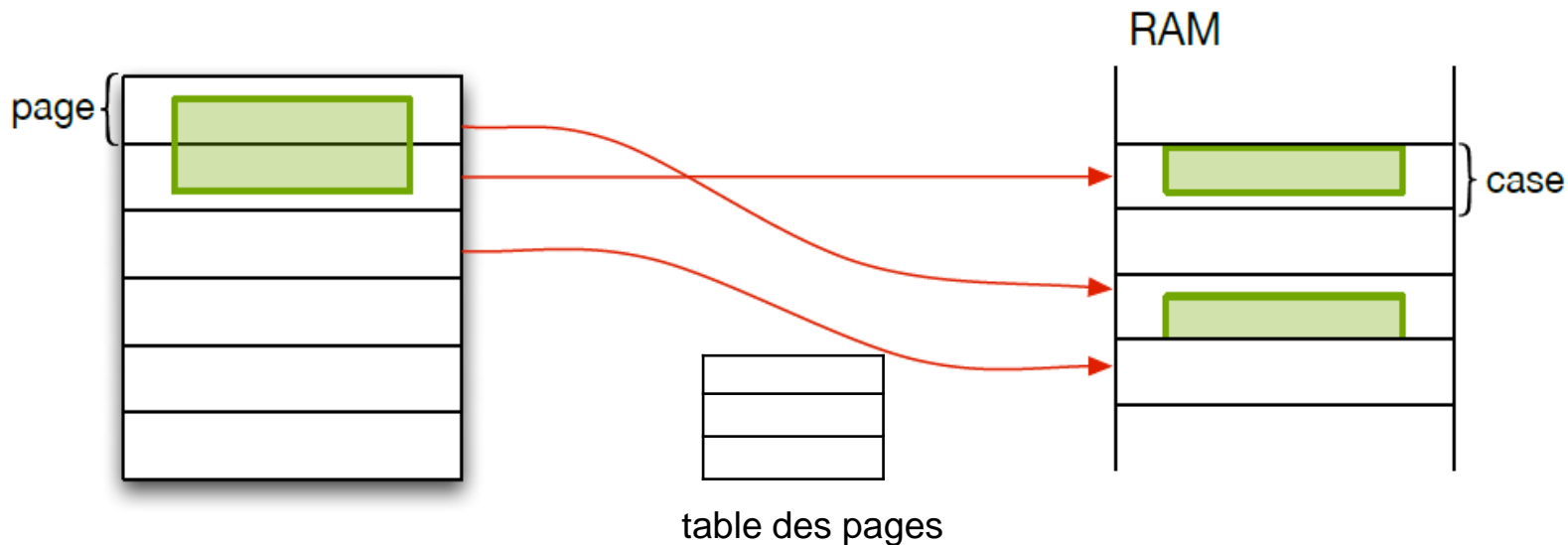
---

- Espace virtuel d'un processus *non contigu*.
- L'espace du processus est divisé en blocs de taille fixe appelés **pages**.
- La mémoire physique est divisée en blocs de taille fixe appelés **cases** ou cadres de page (frame).
- Taille case = Taille page
- Lorsqu'un processus s'exécute ses pages sont chargées dans des cases disponibles en mémoire.



# Pagination - Principe

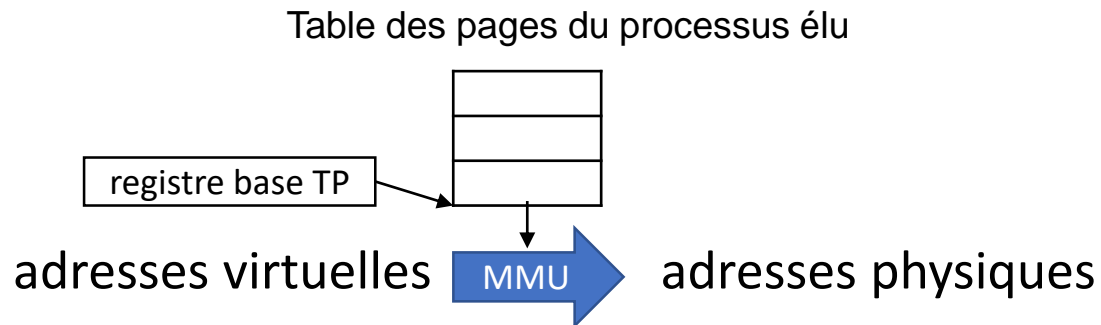
- Espace de travail découpé en pages de taille fixe
- Mémoire physique découpée en cases (*frames*)
- Cases et pages sont de taille identique



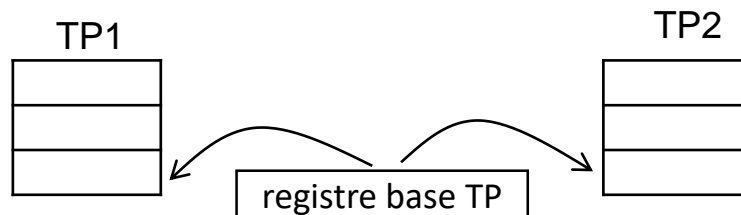
- Evite le problème de fragmentation externe

# Pagination – Traduction d'adresse

- Adresse virtuelle : <page, déplacement >
- Prise en charge par le matériel (MMU)
- 1 **table des pages** par processus = la table désignée par un registre



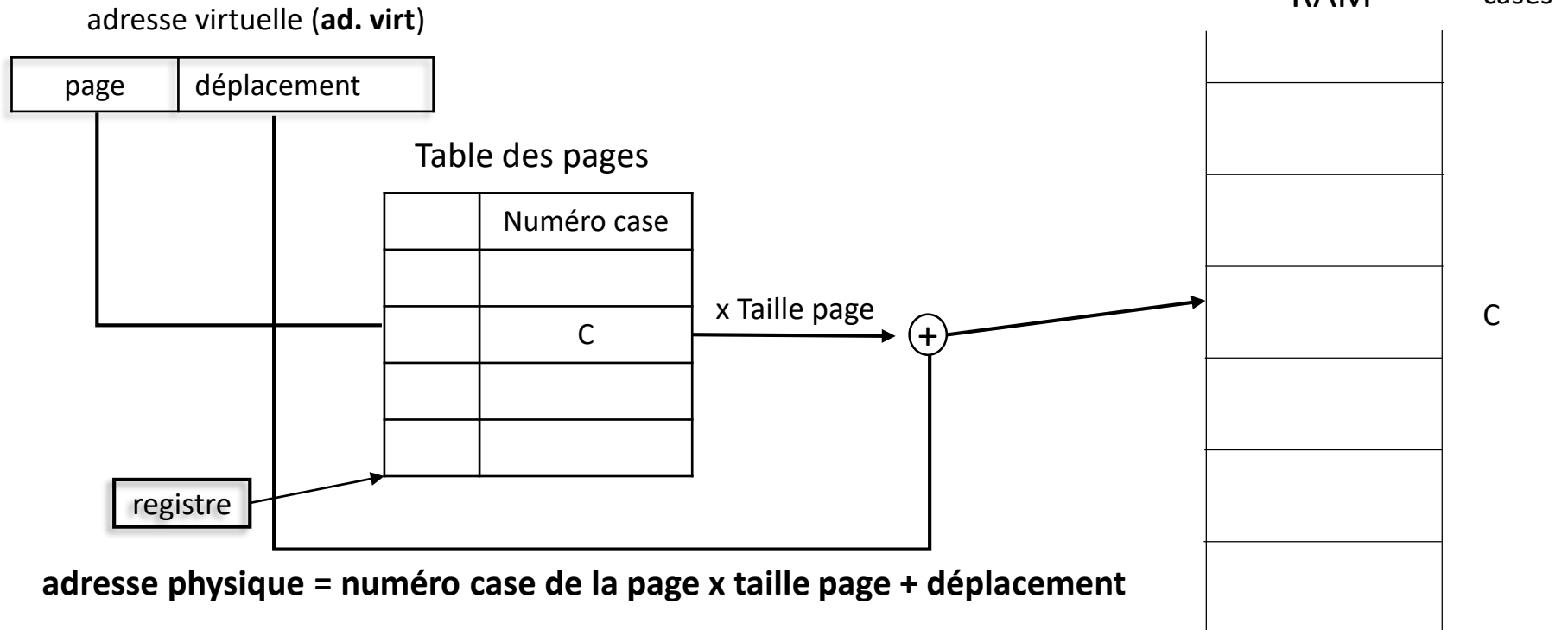
- Commutation : le système change la valeur du registre



# Pagination – traduction d'adresse

$page = ad. \text{ virt. } \mathbf{div} \text{ taille page}$

$déplacement = ad. \text{ virt. } \mathbf{mod} \text{ taille page}$



ex : page de taille 1024 octets, page 10 dans case 20

accès **ad. virt.** 10 300

page =  $10\ 300 \div 1024 = 10$

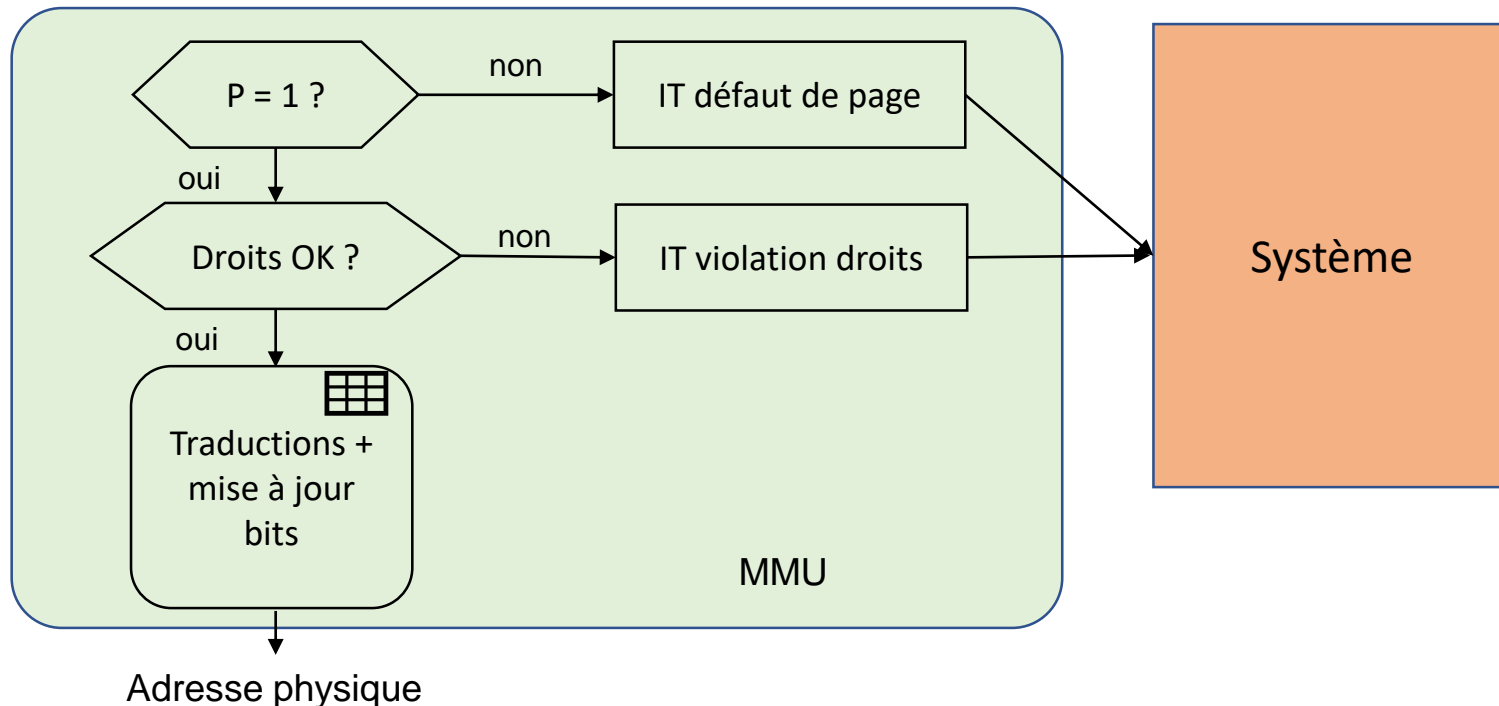
déplacement =  $10\ 300 \bmod 1024 = 60$

10 300 correspond à <10,60>

**adresse physique** =  $20 * 1024 + 60 = 20\ 540$

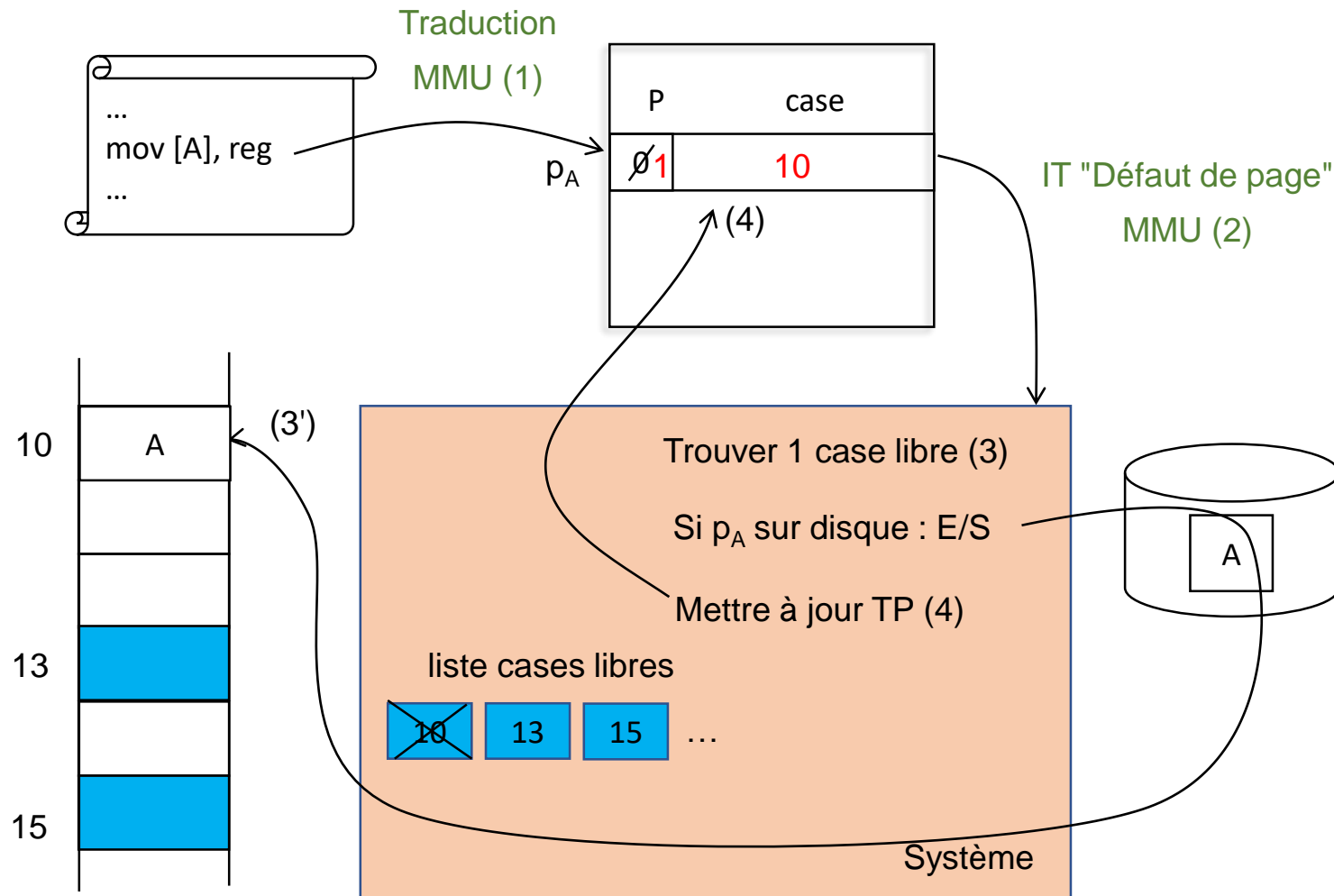
# Pagination - MMU

- Entrée de la table des pages
  - <Présence, Droits, Modification, Verrouillé, ... Numéro cases>
  - Modification : 0 non modifiée, 1 modifiée
  - Verrouillé : 0 non verrouillée, 1 verrouillée
- La MMU (processeur) vérifie pour une adresse <page, offset>



# Pagination – chargement à la demande

Les pages sont chargées "à la demande" en fonction des accès

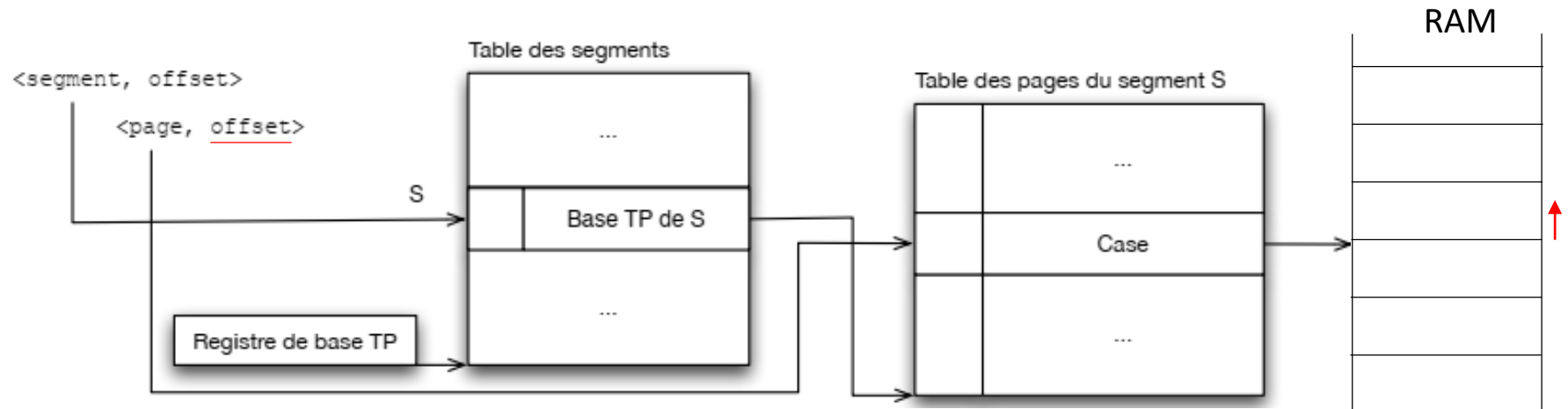


# Mémoire Segmentée Paginée

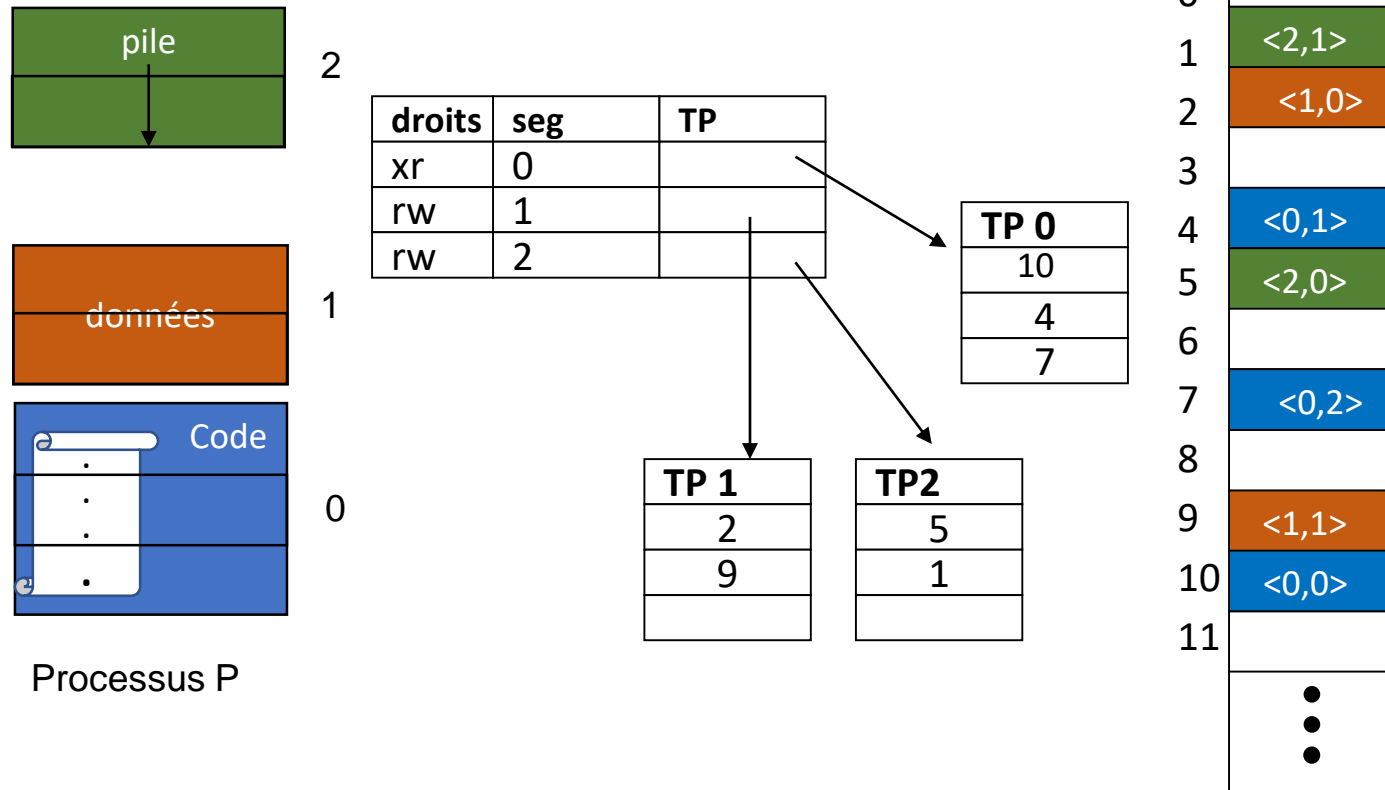
---

- Principe : 1 mémoire segmentée dont les segments sont découpés en pages
- Adresse virtuelle  
    <Segment,     déplacement    >  
        <page,déplacement>
- Schéma "classique " : 1 table des segments par processus + 1 table des pages par segment

# Mémoire Segmentée Paginée - traduction



# Segmentation avec pagination





# Améliorer les performances de la traduction

---

- La traduction faite par MMU critique (fait à chaque référence mémoire)
- Lecture des tables en mémoire (segments / pages) => augmente le nombre d'accès

ex : paginée simple, lecture d'une variable

⇒ lecture TP + lecture de la variable

⇒ x 2 nombre d'accès mémoire !

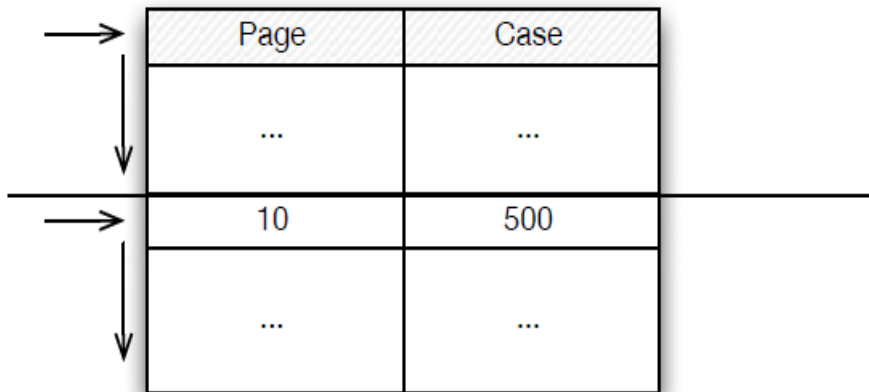
Utilisation d'une mémoire interne (pas en RAM) : la TLB

# TLB (Translation Lookaside Buffer)

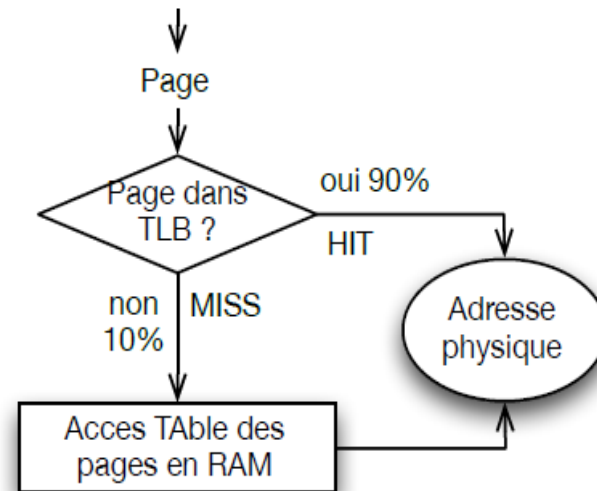
- Mémoire associative interne à la CPU
- Enregistre les traductions de page des accès les plus récents

## TLB : Translation Look-aside Buffer

n-way associative : possibilité de scanner n entrees en parallele



## MMU : Adresse virtuelle



- ex : TLB hit = 98%, accès TLB = 10ns, accès RAM 100ns  
temps d'accès moyen =

$$\begin{aligned} & 0,98 \times (10 \text{ (TLB)} + 100 \text{ (données)}) + \\ & 0,02 \times (10 \text{ (TLB)} + 100 \text{ (Table)} + 100 \text{ (données)}) \\ & = 112 \text{ ns} \end{aligned}$$

# Remplacement de pages

---

- Régulation de l'activité mémoire
- En cas de pénurie mémoire (plus assez de cases libres), maintenir uniquement les pages les plus utiles

## 1) Choisir une page (victime) à évincer

Si la page est modifié ( $M=1$ ), la victime est copiée sur le disque dans le swap

## 2) Utiliser la case libérée par la victime

**Objectif** : choisir une "bonne" victime (i.e., une page peu utile) pour minimiser le nombre de défaut de pages

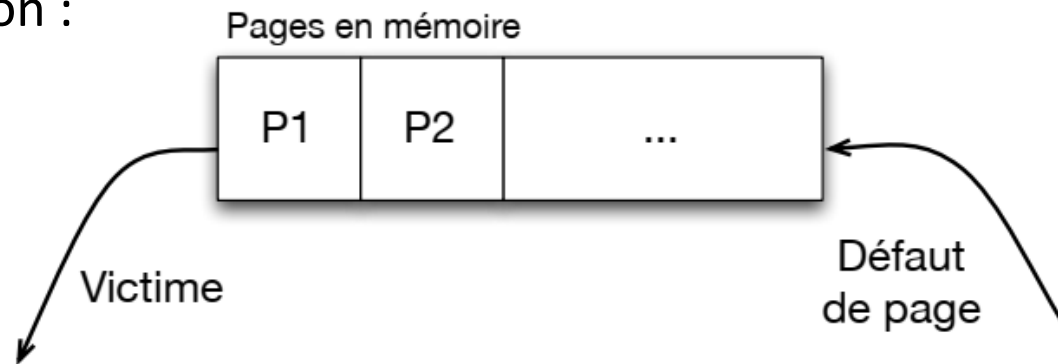
# Algorithme de remplacement - Optimal

---

- Choisir la page victime utilisée le plus tardivement  
⇒ Minimise le taux de défaut (la victime est celle qui engendre le plus tard un défaut de page)
- Nécessite de connaître le futur
- Algorithme théorique, sert de base de comparaison

# Algorithme de remplacement - FIFO

- Choisir la page victime la plus anciennement chargée en mémoire
- Implémentation :



⊕ Simple à implémenter

⊖ Peu performant car ne prend pas en compte l'utilisation des pages

# Algorithme de remplacement - LRU

---

- Choisir la page la moins récemment utilisée (LRU : Least Recently Used)

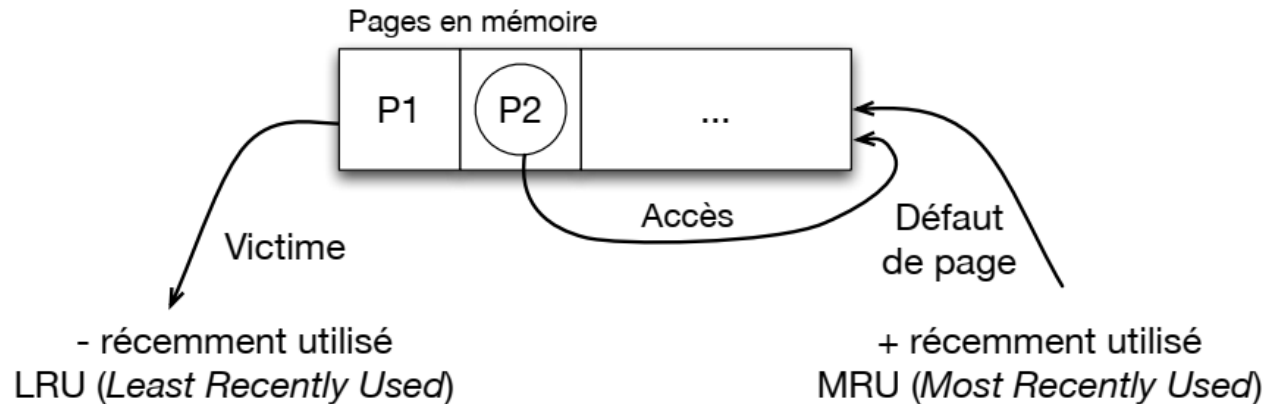
- *Principe* : Exploiter la localité temporelle des programmes

"une page (variable) utilisée il y a longtemps à peu de chance d'être utilisée dans un futur proche"

- Chaque programme a un *working set* = ensemble des pages les plus utilisées

# Algorithme de remplacement - LRU

- Implémentation



- Nécessite de mettre à jour la liste à chaque accès mémoire

Trop coûteux !

=> Approximation de LRU

# Algorithme de remplacement - NRU

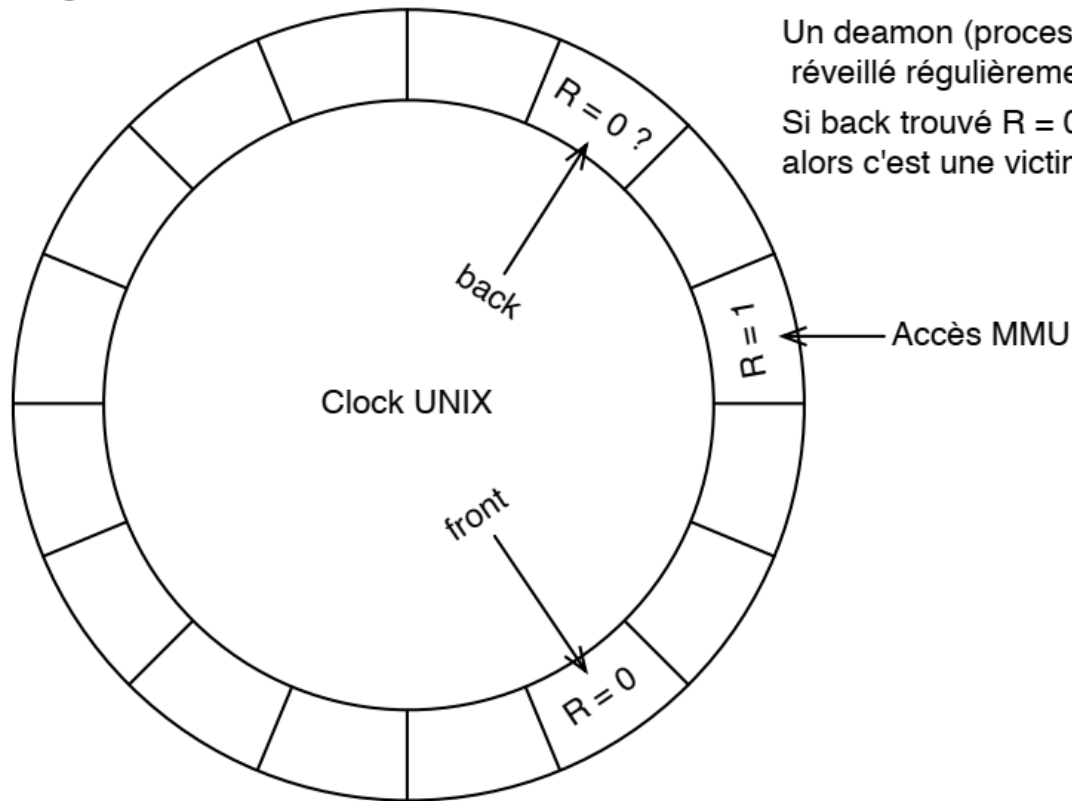
---

- Approximer LRU en choisissant une page non récemment utilisée (NRU : Not Recently Used)
- Implémentation
  - Aide du matériel : 1 bit **R**eference (**R**) est positionné à 1 par MMU à chaque accès à une page (Access bit - Intel)
  - Le système re-positionne régulièrement les bits R à 0 ;  
Bit R testés plus tard par le système, si R toujours à 0  
⇒ page n'a pas été récemment utilisée



# NRU : algorithme clock

Pages en mémoire



Un daemon (processus en mode superviseur)  
réveillé régulièrement  
Si back trouvé  $R = 0$ ,  
alors c'est une victime potentielle