# La mémoire virtuelle

## Traduction d'adresses et pagination à la demande

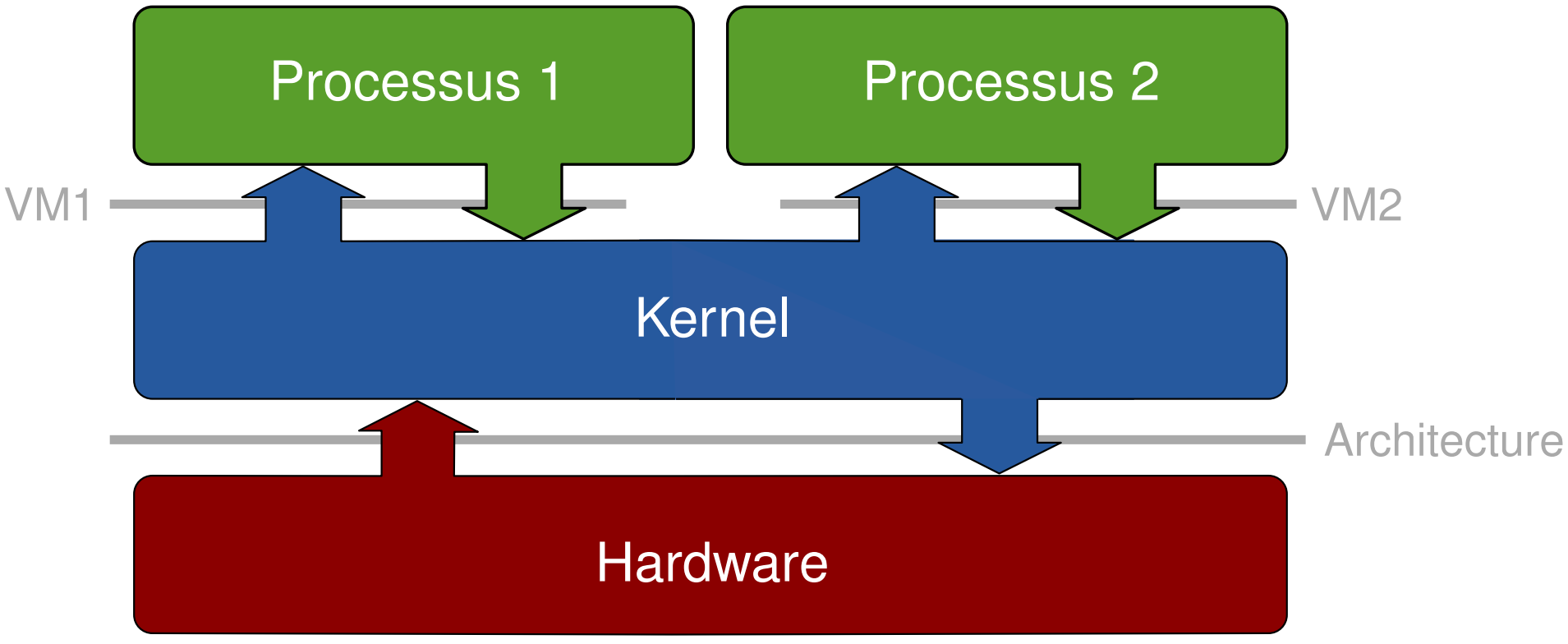
## *Address Translation & Demand Paging*

Guillaume Salagnac

Insa de Lyon – Informatique

2021–2022

# Résumé des épisodes précédents : noyau vs userland




Le processus vu comme une «machine virtuelle»

- un processeur pour moi tout seul : «CPU virtuel»
- une mémoire pour moi tout seul : «mémoire virtuelle»

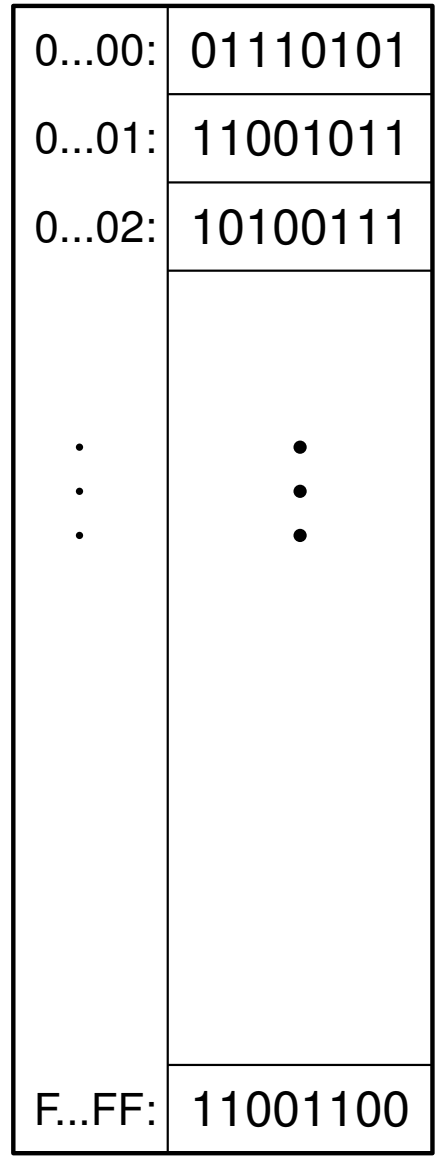
# La mémoire virtuelle : intuition

Principe : chaque processus a sa propre mémoire

- une mémoire = un vaste tableau d'octets
  - indices = adresses
  - contenu : complète liberté du programmeur
- décider où mettre quoi = allocation
  - à l'intérieur du processus (cf chap. 4)
  - ▶ rôle du langage de programmation

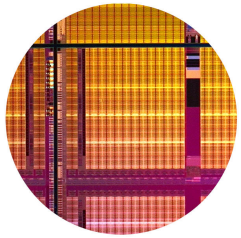
Implémentation :

- combinaison de plusieurs technologies
  - hiérarchie mémoire : cache, RAM, disque
  - en principe, invisible pour le programmeur
- coopération entre noyau et matériel

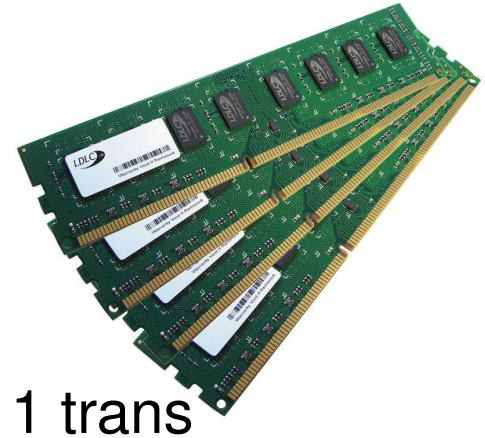


# Problème n° 0 : la mémoire est trop lente

Seulement deux technologies pour la mémoire vive

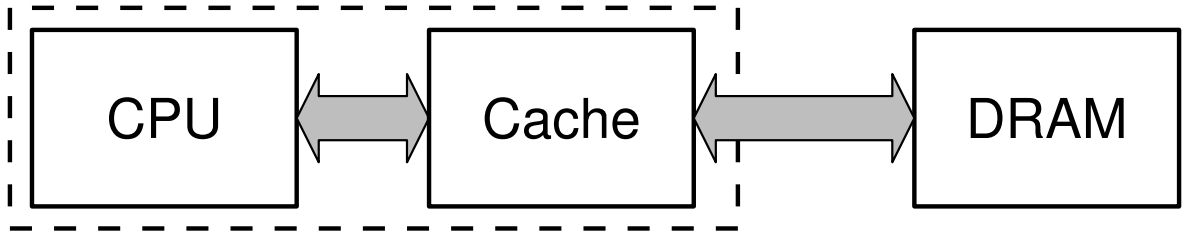


SRAM : 1 bit = 6 transistors
- ▶ rapide mais coûteux



DRAM : 1 bit = 1 capa + 1 trans
- ▶ peu cher mais très lent (100×)

Mémoire cache : ajouter une petite SRAM dans le CPU
- garde une copie des données récemment accédées




- capacité : quelques Mio
- DRAM seulement utile sur défaut de cache (en VO : *cache miss*)
- solution entièrement HW : complètement invisible pour le SW

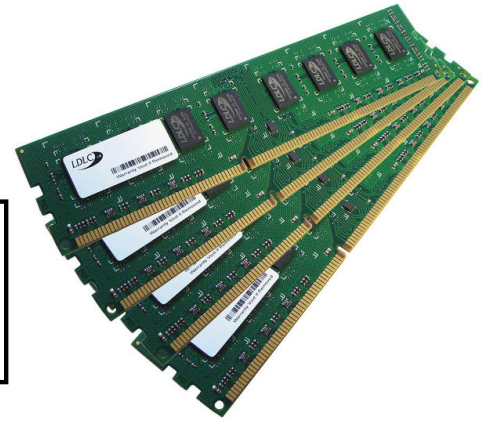
# Problème n° 1 : la mémoire est trop petite

un processus = $2^{64}$ adresses = 18 446 744 073 709 551 616 octets
= 17 179 869 184 Gio

| | |
|---|---|
| 0...00: | 01110101 |
| 0...01: | 11001011 |
| 0...02: | 10100111 |
| ⋮ | ⋮ |
| F...FF: | 11001100 |

VS

DRAM

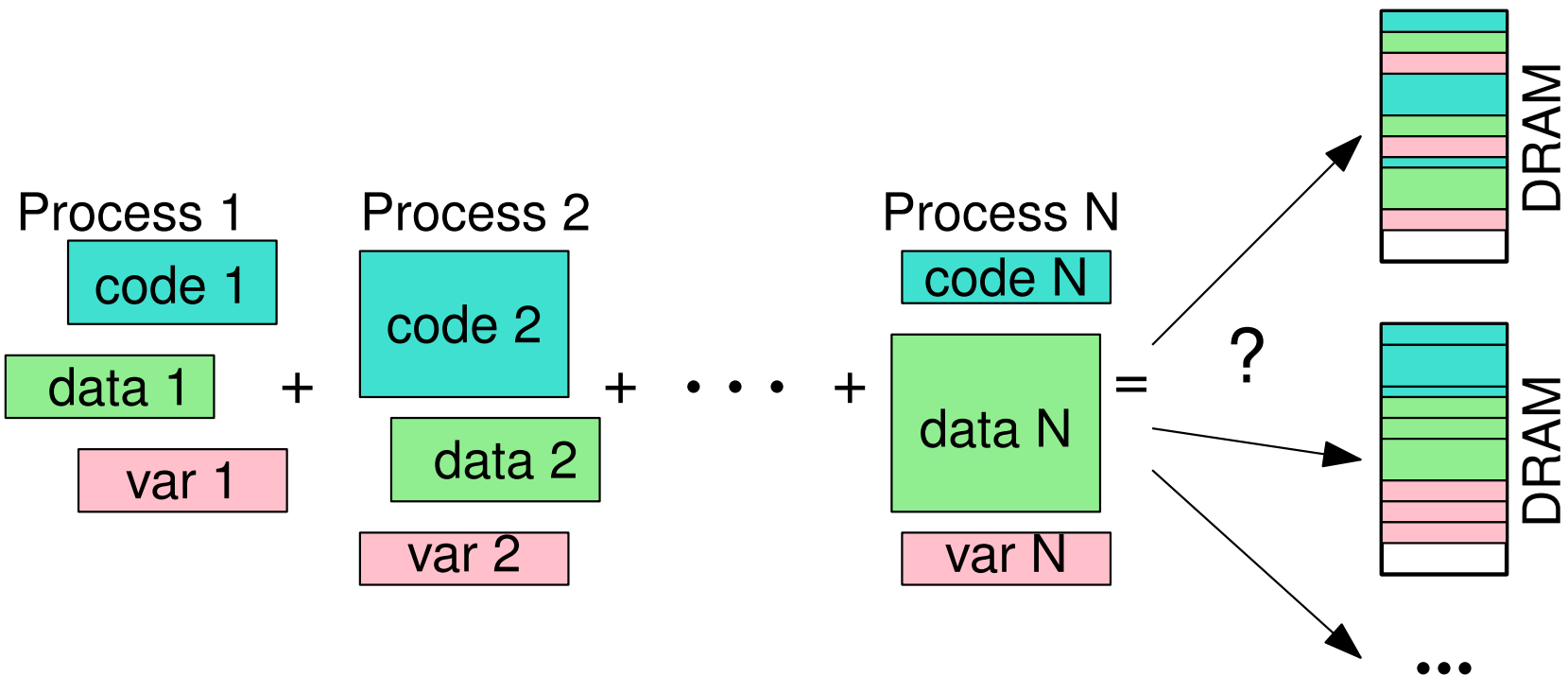


mémoire physique =
quelques Gio

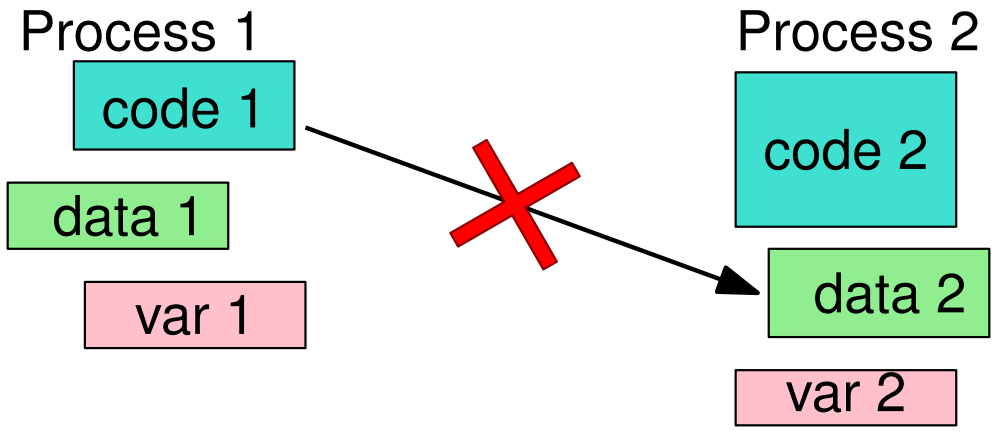
# Problème n° 2 : comment gérer l'espace disponible ?

Même si chaque processus n'a pas besoin de $2^{64}$ octets,

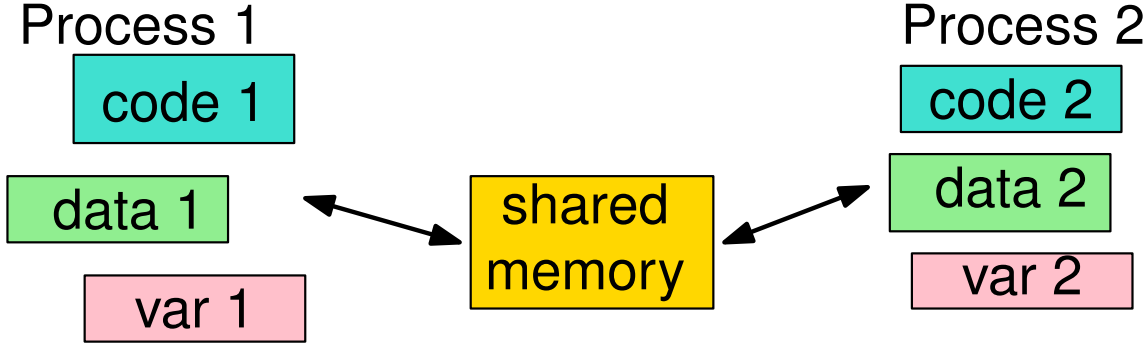
- il y a de nombreux processus simultanément
  - des dizaines ? des centaines ? des milliers ?
  - combien d'espace occupe chacun ? pour combien de temps ?
- qui manipulent des éléments hétérogènes
  - instructions (code) ▶ en lecture seule
  - données (par. ex. fichiers) ▶ chargées depuis le disque
  - variables (locales, globales) ▶ en lecture+écriture



# Problème n° 3 : comment protéger chaque processus ?




# Problème n° 4 : comment permettre la coopération ?



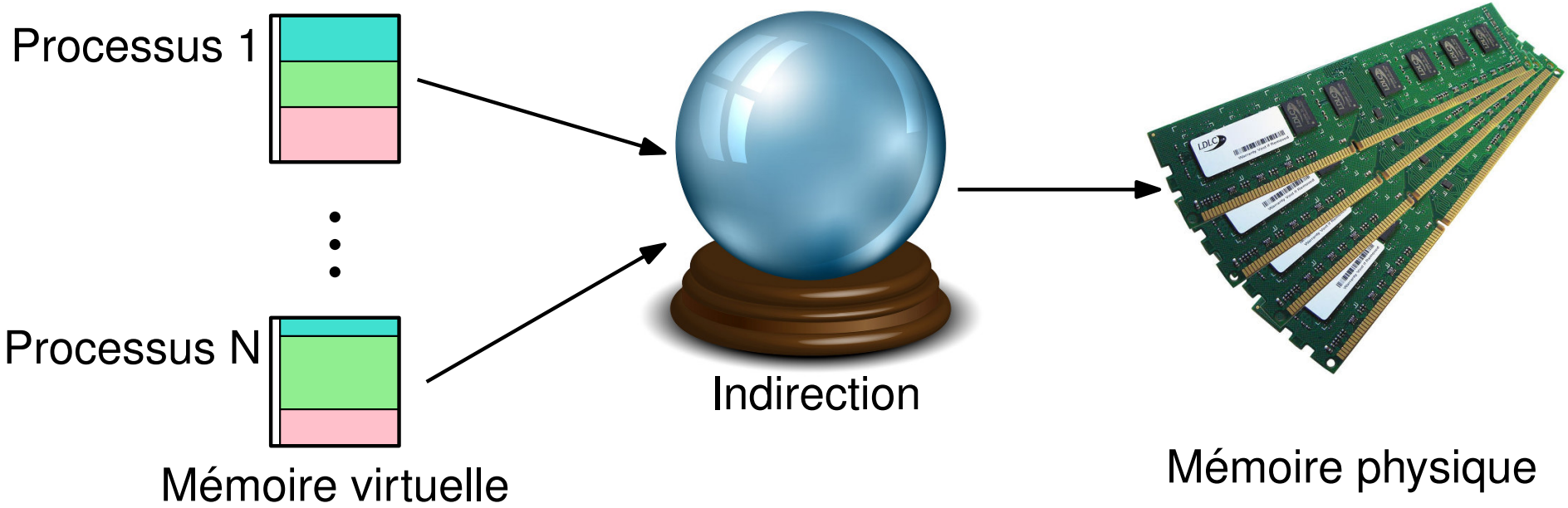
# Solution : ajouter un niveau d'indirection

*All problems in computer science can be solved by another level of indirection*
David Wheeler

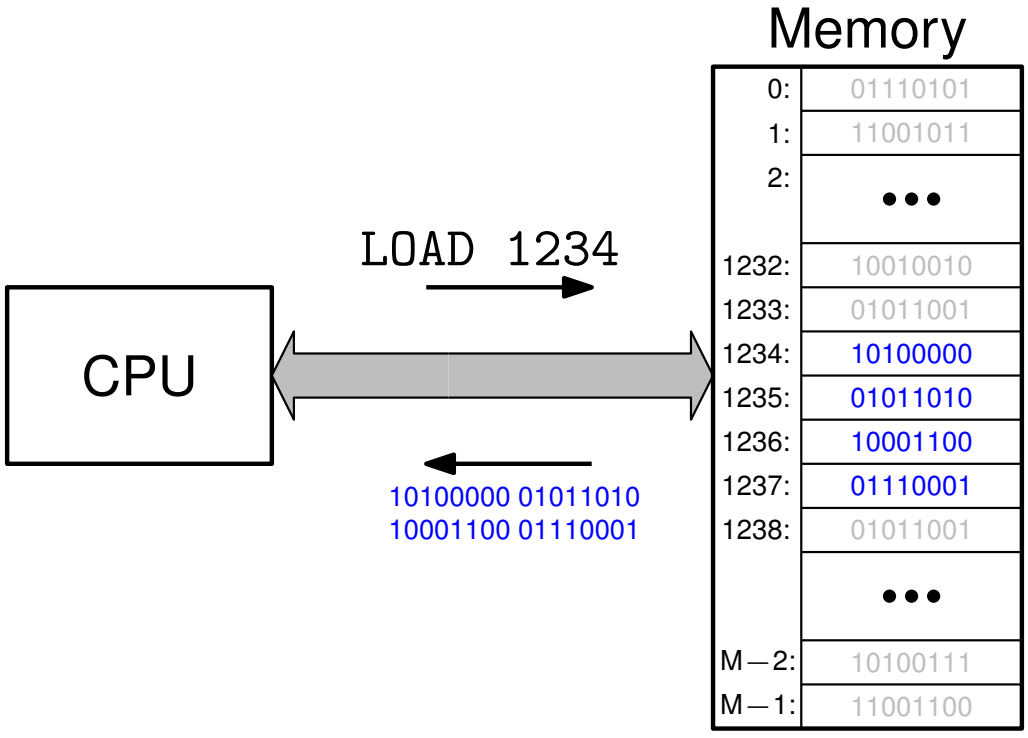
**Définition : Indirection aka déréférencement**

Accès *indirect* à quelque chose, i.e. au travers d'une *référence*

Exemples
- pour joindre quelqu'un au téléphone, devoir passer d'abord par un standard
- invoquer une méthode virtuelle en C++, Java...

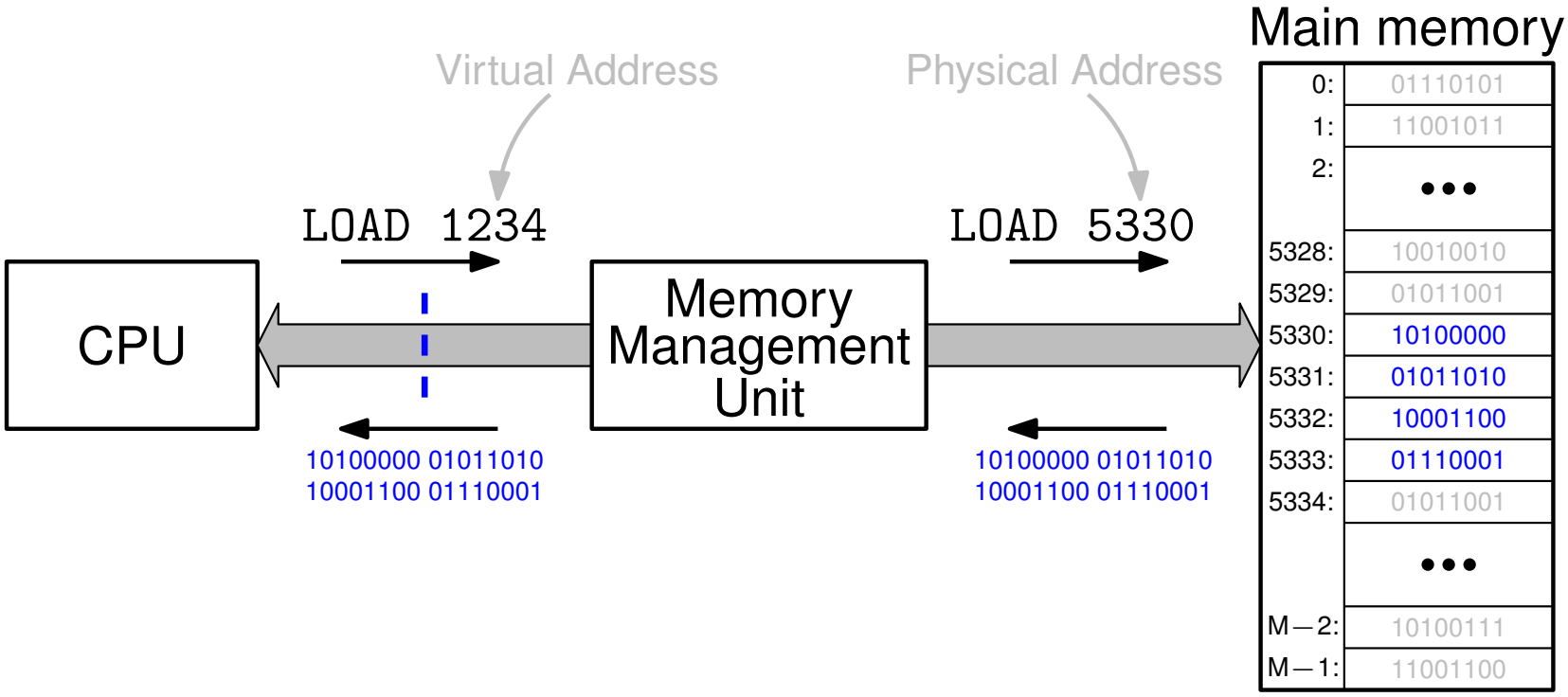


# Abstraction : adressage direct




- la mémoire = tableau de $M$ cases numérotées de 0 à $M-1$
- une case = un octet (1B = 1 byte) = 8 bits
- un accès (i.e. lecture ou écriture) = un mot (word) = un paquet de plusieurs octets consécutifs
  - en général 4 octets (= 32 bits) ou 8 octets (= 64 bits)

# Réalité : adressage virtuel




Sur une architecture moderne :

- CPU ne manipule que des «adresses virtuelles» (VA)
- composant supplémentaire : Memory Management Unit (MMU)
  - traduit à la volée chaque VA en une «adresse physique» (PA)
- mem centrale = mem principale = mem physique = DRAM

# Virtualisation des adresses : remarques

Espace d'adressage virtuel = VAS = $\{0, 1, 2, ..., N-1\}$

- le CPU exprime les adresses sur n bits ▶ N=$2^n$
- chaque processus dispose d'un VAS privé

Espace d'adressage physique = PAS = $\{0, 1, 2, ..., M-1\}$

- une adresse pour chaque octet de (DRAM + périphériques)
- adresses exprimées sur m bits ▶ M=$2^m$

**La Memory Management Unit**

- traduit les adresses virtuelles en adresses physiques
  - ▶ se comporte comme un dictionnaire : VAS $\mapsto$ PAS
- sera reconfigurée par l'OS à chaque changement de contexte

# À retenir : pourquoi la mémoire virtuelle

Problème 1 : pas assez de mémoire vive

- va-et-vient (en VO *swapping*) = ne pas tout stocker en DRAM
  = utiliser la DRAM comme un cache pour les VAS

Problème 2 : gestion des zones libres/occupées (en DRAM)

- pagination (en VO *paging*) = une unique taille de bloc

Problème 3 : isolation entre processus, protection du noyau

- garantir que les VAS ne se «chevauchent» pas (dans le PAS)

Problème 4 : partage de données entre processus

- une même page *peut* apparaître dans deux VAS

# Plan

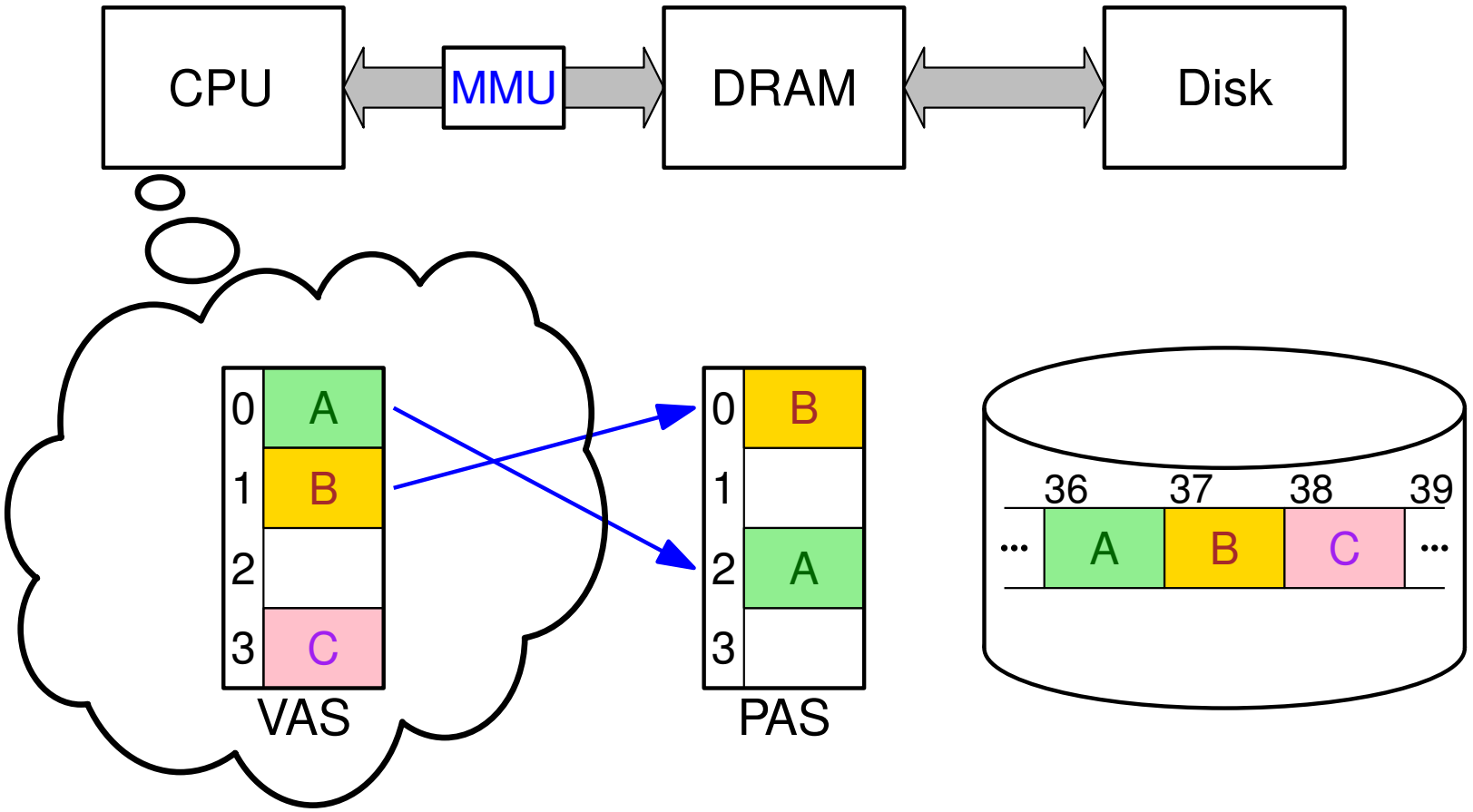
1. La mémoire virtuelle : intuitions et motivation
2. Pagination et traduction d'adresses
3. Memory swapping : pagination à la demande
4. Gestion de la mémoire physique
5. Protection, Isolation et partage

# Pagination à la demande : principe

utiliser le disque pour «agrandir la mémoire vive»

$\Longleftrightarrow$

utiliser la DRAM comme un cache du disque



# Pagination à la demande : remarques

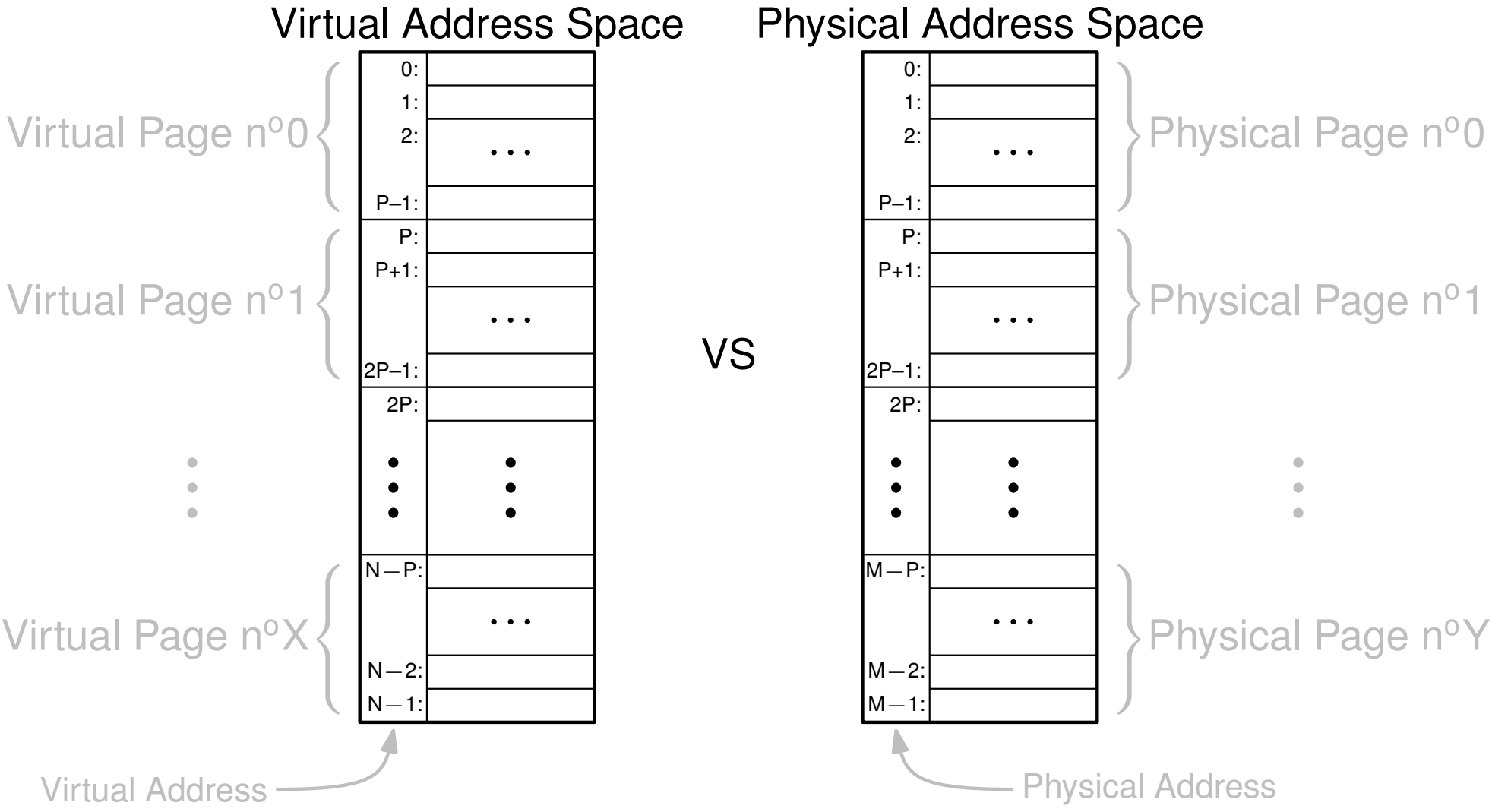
- contenu des processus «alloué sur le disque»
  - dans un fichier (ou plusieurs) appelé le *swap file*
  - données chargées en RAM seulement lorsque accédées
- latence disque trop importante pour faire attendre le CPU
  - DRAM = 10x à 100x plus lent que SRAM/CPU
  - disque = 1000x à 100 000x plus lent que DRAM
- trop complexe pour être géré purement en matériel

▶ mécanisme implémenté par le noyau
  - avec la coopération du matériel
  - invisible pour le programmeur d'application

Une unique taille de bloc : P=$2^p$ octets (en général P=4Kio)

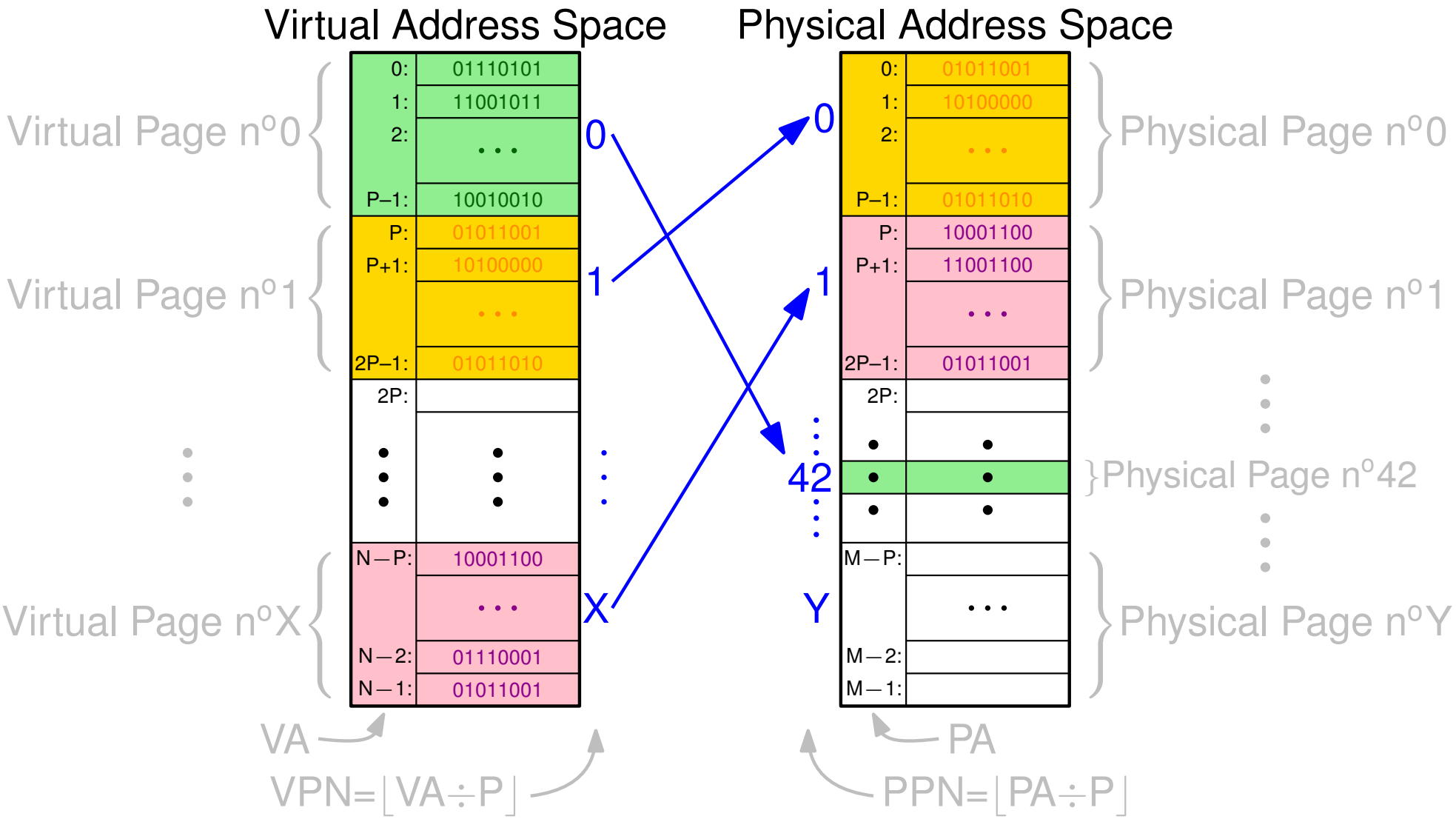
- ▶ disque dur géré par blocs de taille P appelés secteurs
- ▶ DRAM gérée par blocs de taille P appelés pages physiques
  - parfois : «frames» en VO, soit «cadres» en VF
- ▶ VAS géré par blocs de taille P appelés pages virtuelles

# Numérotation des pages




- Numéro de page virtuelle : VPN $\in \{0, 1, ..., X\}$ avec $X = N \div P$
- Numéro de page physique : PPN $\in \{0, 1, ..., Y\}$ avec $Y = M \div P$

# Traduire des adresses = traduire des nº de pages




## Rôle de la Memory Management Unit

À chaque accès mémoire, «traduire» le VPN en son PPN

# Traduction d'adresses : principe

un Virtual Address Space = une «vue» sur la mémoire physique

- en vérité chaque donnée est toujours bien stockée en DRAM
- mais visible depuis le CPU seulement à une adresse virtuelle

**Algorithme de traduction VA $\mapsto$ PA (implémenté par la MMU)**

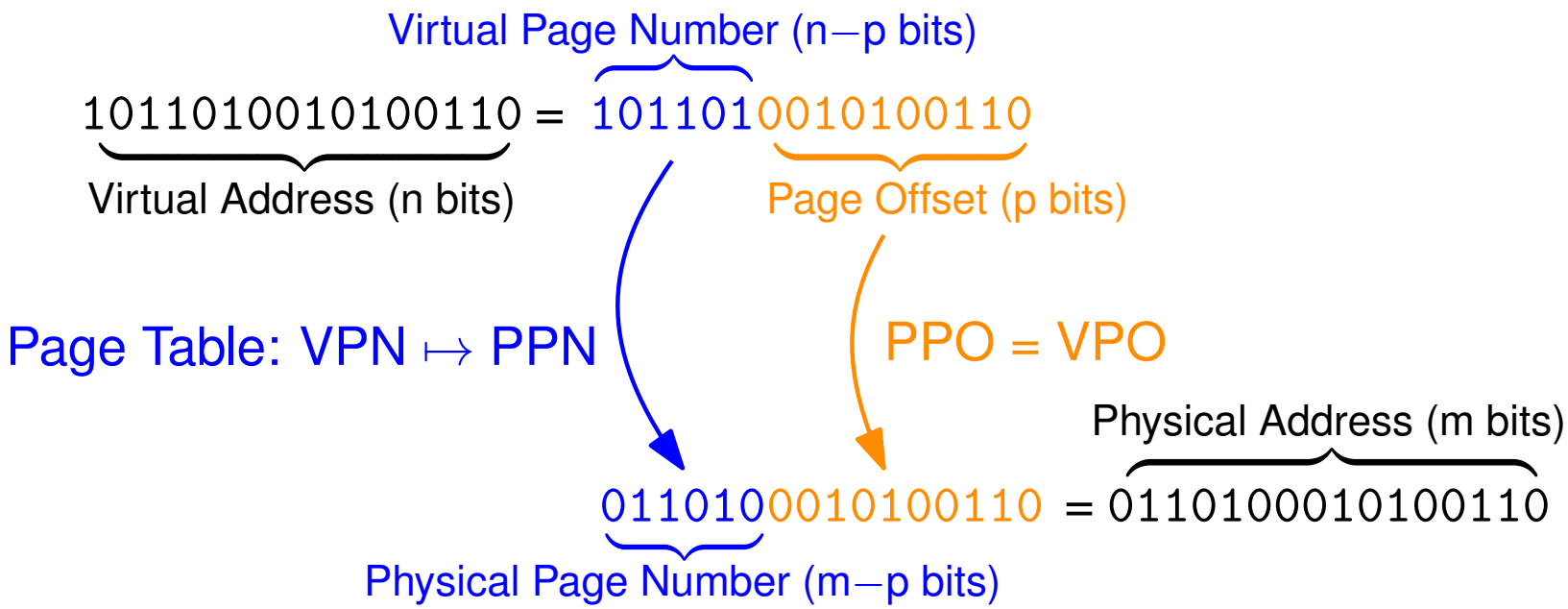
1) calculer VPN = $\lfloor \text{VA} \div \text{P} \rfloor$ et PO = VA mod P
   - autrement dit, VA = VPN $\times$ P + PO
   - PO = Page Offset = position à l'intérieur de la page
2) trouver le PPN correspondant au VPN
   - dans un annuaire appelé la Table de Pagination = *Page Table* PT
3) ) calculer PA = PPN $\times$ P + PO
   - offset identique dans la page virtuelle et la page physique

# Implémentation des calculs d'adresses

Taille de page = P = $2^p$ donc :

- division entière par P = gratuite
- division modulo P = gratuite
- multiplication par P = gratuite
- addition de l'offset = gratuite

Exemple avec m=n=16 et p=10 :



# Implémentation de la table de pagination

PT = dictionnaire = «table de correspondances» = *lookup table*

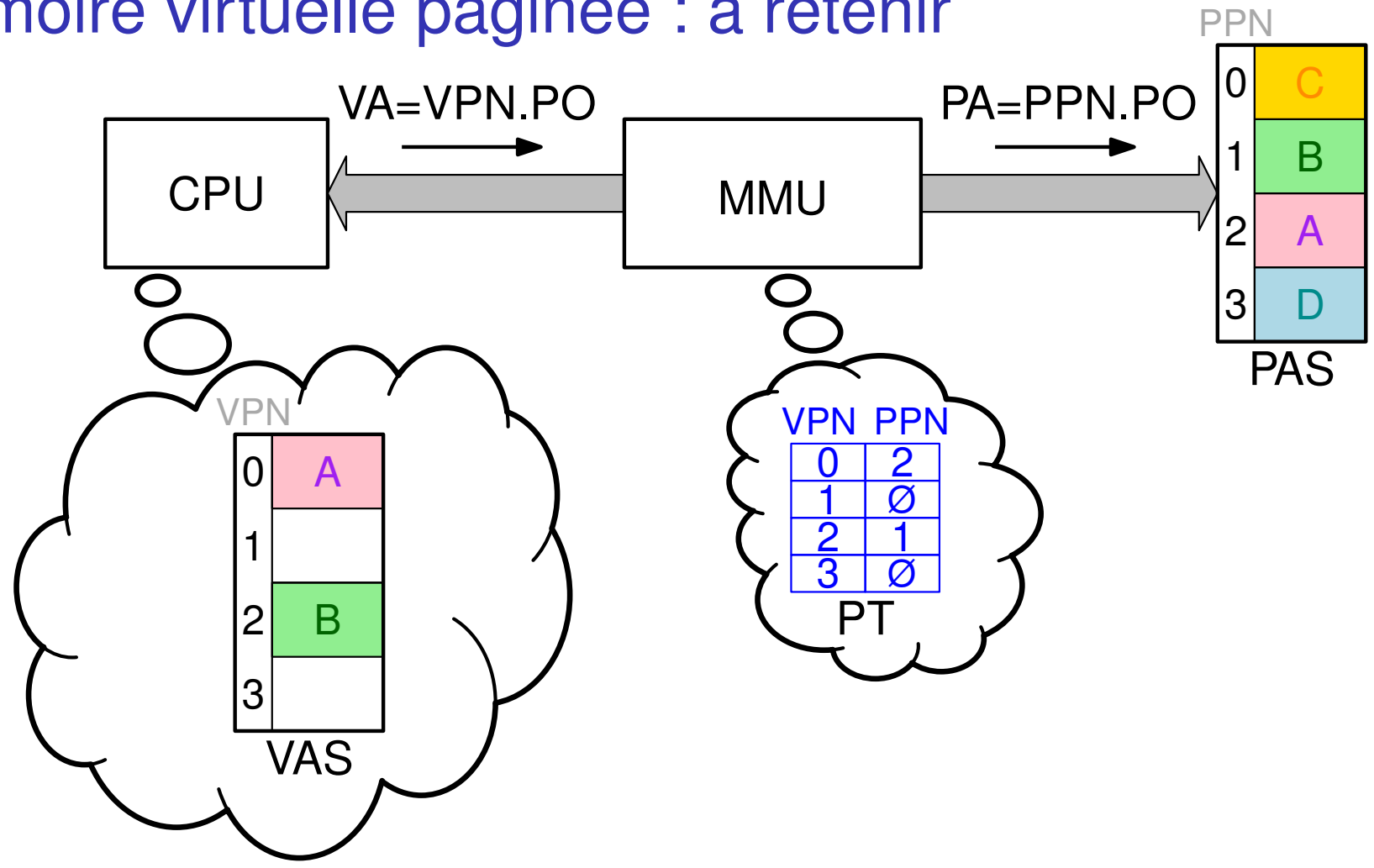
- une clé = un nº de pages virtuelle VPN
- une valeur = un nº de page physique PPN + des métadonnées
- une paire clé+valeur = une Page Table Entry PTE

PT elle-même stockée en mémoire principale

- beaucoup trop de PTE pour tenir en entier dans la MMU
- autrefois : tableau exhaustif. aujourd'hui : arbre de recherche
- consultation : implémentée en matériel par la MMU

Rappel : 1 processus = 1 VAS = 1 PT

# Mémoire virtuelle paginée : à retenir




1) CPU demande à accéder à une adresse virtuelle VA
2) MMU consulte le bon PTE (dans la PT) pour connaître le PPN
3) MMU transmet la requête traduite PA sur le bus système
4) réponse transmise au CPU sans intervention de la MMU

# Plan

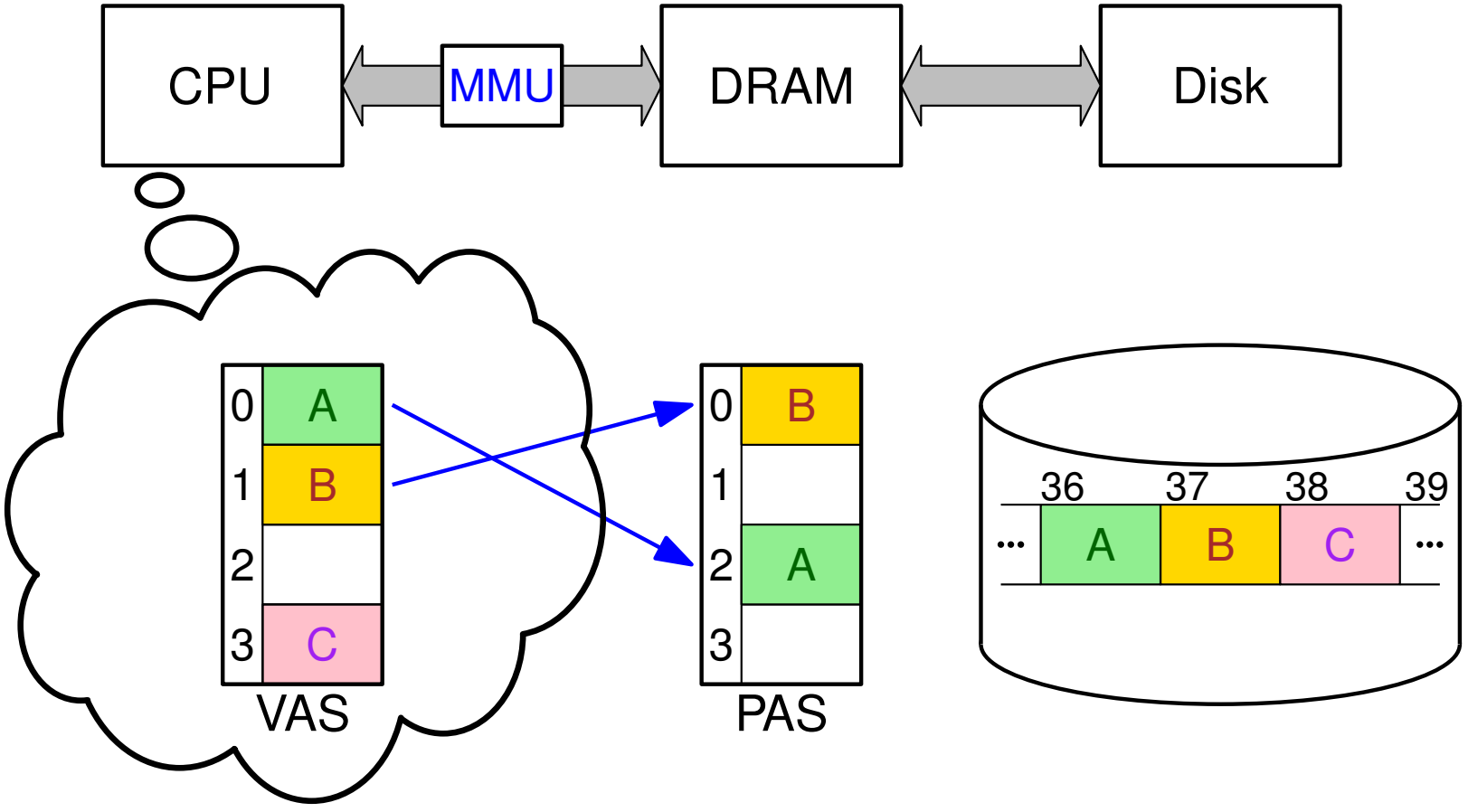
1. La mémoire virtuelle : intuitions et motivation
2. Pagination et traduction d'adresses
3. Memory swapping : pagination à la demande
4. Gestion de la mémoire physique
5. Protection, Isolation et partage

# Pagination à la demande : principe

utiliser le disque pour «agrandir la mémoire vive»

$\Longleftrightarrow$

utiliser la DRAM comme un cache du disque




Problème : et si le programme accède à sa page n° 2 ? et n° 3 ?

# Table de pagination : PTE valide vs PTE invalide

Page Virt. n° 3 : existe sur le disque mais pas chargée en DRAM

- en VO *swapped-out* ou *uncached*

Page Virt. n° 2 : n'existe pas du tout : adresses virtuelles inutilisées

- en VO *unallocated* ou *unmapped*

Du point de vue de la MMU :

- page présente en DRAM = accès possible = PTE valide
  - si CPU essaye d'accéder ▶ MMU traduit vers adresse phy
- page absente de DRAM = accès impossible = PTE invalide
  - si CPU essaye d'accéder ▶ MMU lève une interruption

implem : un drapeau (booléen) dans le PTE : le *«valid bit»*

# Accès à une page virtuelle au PTE invalide

Quand un processus accède à une page invalide :

- MMU lève une interruption logicielle (trappe)
- CPU saute dans le noyau et exécute l'ISR associée
  - si page non-allouée ▶ erreur irrécupérable
    - ▶ le noyau tue le processus fautif (VO : *segmentation fault*)
  - si page déchargée ▶ défaut de page (VO : page fault)
    - ▶ le noyau doit charger la page en DRAM

Idée : noter le n° de secteur dans la PTE, à la place du PPN !

Exemple :

| | PTE valide ? | n° de page physique ou n° de secteur |
|---|---|---|
| PTE n° 0 | 1 | PP n° 2 |
| PTE n° 1 | 1 | PP n° 0 |
| PTE n° 2 | 0 | Ø |
| PTE n° 3 | 0 | secteur n° 38 |

(en vrai c'est un peu plus compliqué, cf `vm_area_struct` dans Linux mais l'idée est la même)

# Faute de page : déroulement temporel

1. CPU demande une certaine adresse virtuelle
2. MMU trouve un PTE invalide dans la PT
3. MMU envoie une requête d'interruption
4. OS vérifie que la page virtuelle demandée existe bien
5. OS trouve une page physique libre
   - il faut parfois décharger (swap out) une autre page
6. OS charge (swap in) la page demandée depuis le disque
   - latence disque = I/O burst ▶ changement de contexte
   - exécution d'un autre processus pour rentabiliser le CPU

7. lorsque page chargée : OS met à jour le PTE dans la PT
8. OS rend la main au processus d'origine
   - accès mémoire ré-exécuté, cette fois avec succès
   - toujours «invisible» pour le programmeur

# Pagination à la demande VS performances

Temps d'accès moyen : Average Memory Access Time

- AMAT = page hit time + (page fault rate $\times$ page fault penalty)
- page hit time $\approx$ latence DRAM $\approx$ 50 ns
- page fault penalty $\approx$ latence disque $\approx$ 5 ms

▶ les fautes de pages doivent rester rares

**Loi empirique : le principe de localité des accès**

Des adresses proches sont accédées à des instants proches

▶ Working Set d'un processus = ensemble des pages virtuelles accédées récemment par ce processus

Attention : si $\Big($somme( Working Set Sizes ) > taille(DRAM)$\Big)$ alors

▶ Écroulement soudain des performances (en VO *thrashing*) causé par un excès de fautes de pages

remède : réduire le degré de multiprogrammation

# Plan

# Pourquoi la mémoire virtuelle ?

Problème 1 : pas assez de mémoire vive

- va-et-vient (en VO *swapping*) = ne pas tout stocker en DRAM
  = utiliser la DRAM comme un cache pour les VAS

Problème 2 : gestion des zones libres/occupées (en DRAM)

- pagination (en VO *paging*) = une unique taille de bloc

Problème 3 : isolation entre processus, protection du noyau

- garantir que les VAS ne se «chevauchent» pas (dans le PAS)

Problème 4 : partage de données entre processus

- une même page *peut* apparaître dans deux VAS

# Allocation dynamique de pages

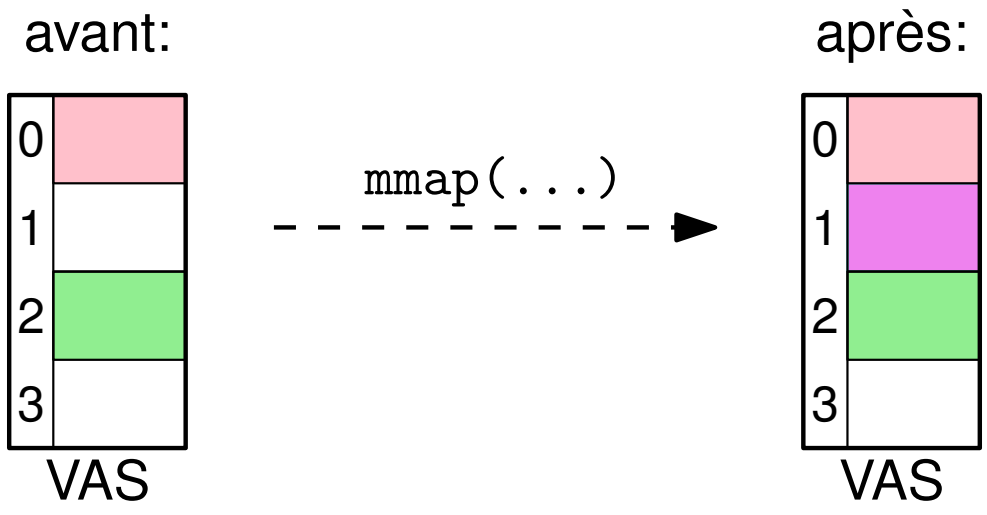
Grâce à la pagination

- chaque VP peut être placée dans n'importe quelle PP
- typiquement, dans différentes PP au cours du temps

▶ facilite la gestion de la mémoire physique

Allocation de nouvelle(s) page(s) vierge(s) pour un processus

▶ avec l'appel système `mmap()`



# L'appel système `mmap()`

## Pour demander dynamiquement l'allocation de nouvelles pages

```
#include <sys/mman.h>

void * mmap(NULL,
            size_t len, // length of region
            int prot,   // protections
            int flags,  // type of mapping
            0,0);
```

Valeur renvoyée ▶ adresse virtuelle de la nouvelle région

`len` ▶ noyau allouera toujours un nombre entier de pages

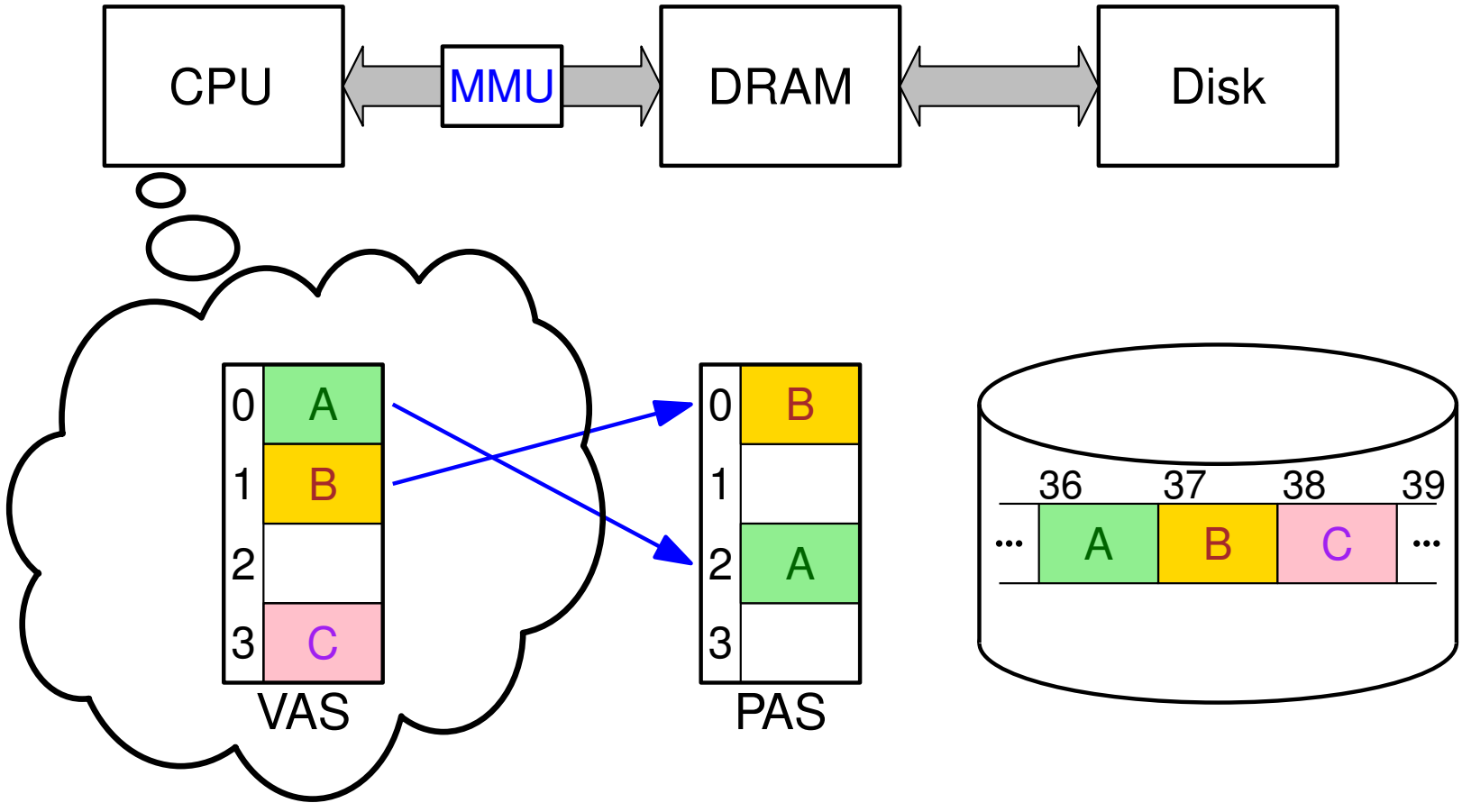
`prot` ▶ `PROT_READ` et/ou `PROT_WRITE` (et/ou `PROT_EXEC`)

`flags` ▶ `MAP_ANONYMOUS` | `MAP_PRIVATE`

(des pages vierges) (juste pour moi)

# Notion de «Swap File»

Rappel : chaque page virtuelle existe d'abord sur le disque




memory swapping = va-et-vient entre DRAM et un swap file :

- swap in lors d'une faute de page
- swap out pour libérer des pages physiques

# Différents types de mapping mémoire

`MAP_ANONYMOUS` ▶ swap depuis/vers un fichier anonyme

- page(s) vierges créée(s) de façon paresseuse
  - le premier accès cause une faute de page
  - noyau alloue alors une page physique quelconque
  - dont il aura effacé le contenu
- si `MAP_SHARED` ▶ plusieurs processus peuvent accéder
- si `MAP_PRIVATE` ▶ isolation via Copy-On-Write

`MAP_FILE` ▶ swap depuis un fichier ordinaire !

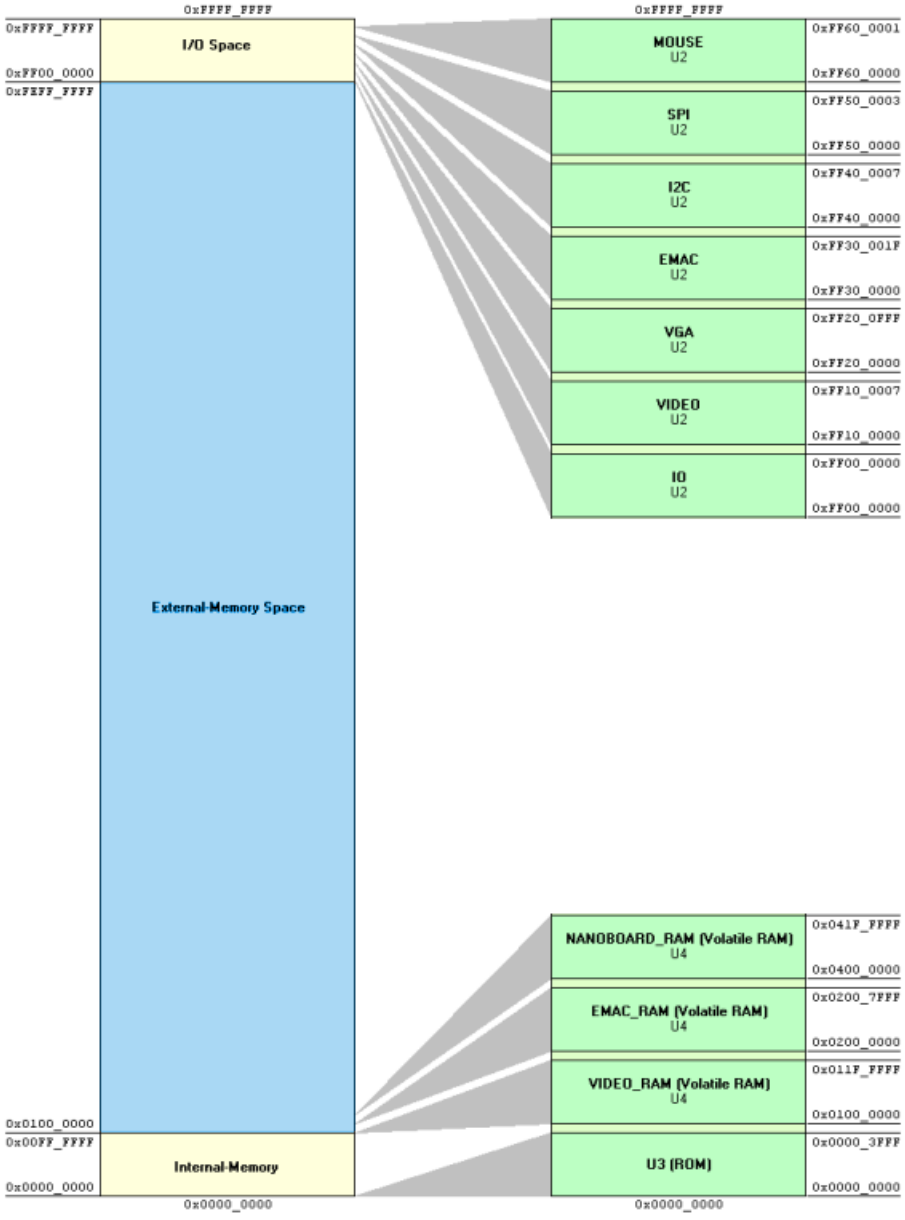
- contenu mémoire initial = lu depuis le fichier
  - signature `void* mmap(...,..., int fd, int offset);`
  - lecture paresseuse, seulement lorsque faute de page
- si `MAP_SHARED` ▶ swap vers le fichier d'origine
- si `MAP_PRIVATE` ▶ Copy-On-Write + swap vers fich. anonyme

# Différents types de mapping mémoire

| | `MAP_ANONYNOUS` | | `MAP_FILE` sur le fichier `fd` | |
|---|---|---|---|---|
| `MAP_SHARED` | contenu initial | pages vierges | contenu initial | lu depuis le fichier `fd` |
| | écritures | visibles depuis tous les processus qui ont ce mapping | écritures, swap out | vers le fichier `fd` |
| | swap out | vers fichier anonyme | | |
| `MAP_PRIVATE` | contenu initial | pages vierges | contenu initial | lu depuis le fichier `fd` |
| | écritures | privées (copy-on-write) | écritures | privées (copy-on-write) |
| | swap out | vers fichier anonyme | swap out | vers fichier anonyme |

# Plan

# Protection des périphériques




**Rappel : Memory-Mapped I/O**

communication avec les périphs via leurs adresses physiques

En général

- adresses DRAM : allouées aux processus userland
- adresses MMIO : réservées au noyau + drivers

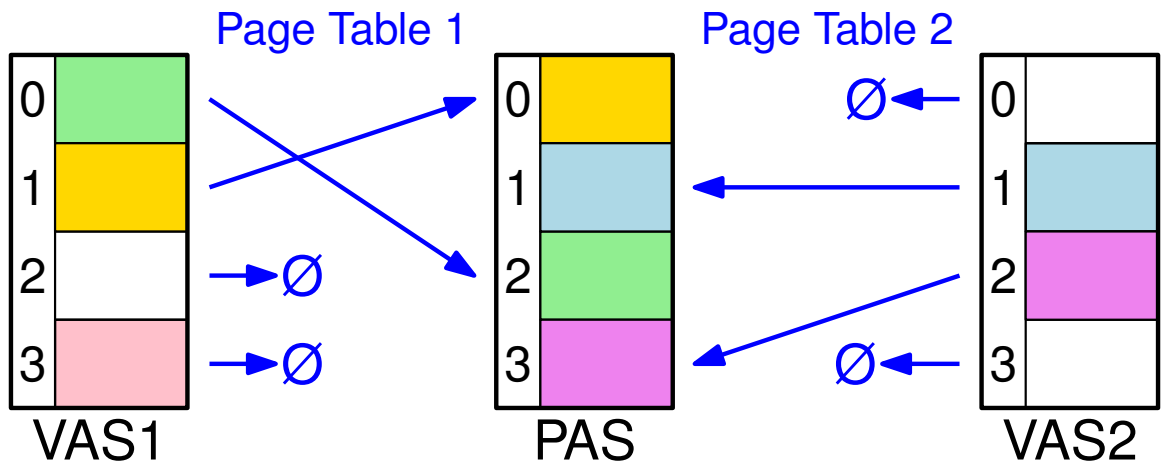
▶ facile à garantir via pagination

Note : la MMU elle-même est un périphérique

# Isolation entre processus

Principe : chaque processus dispose d'un VAS individuel

- l'OS maintient une table de pagination pour chaque processus



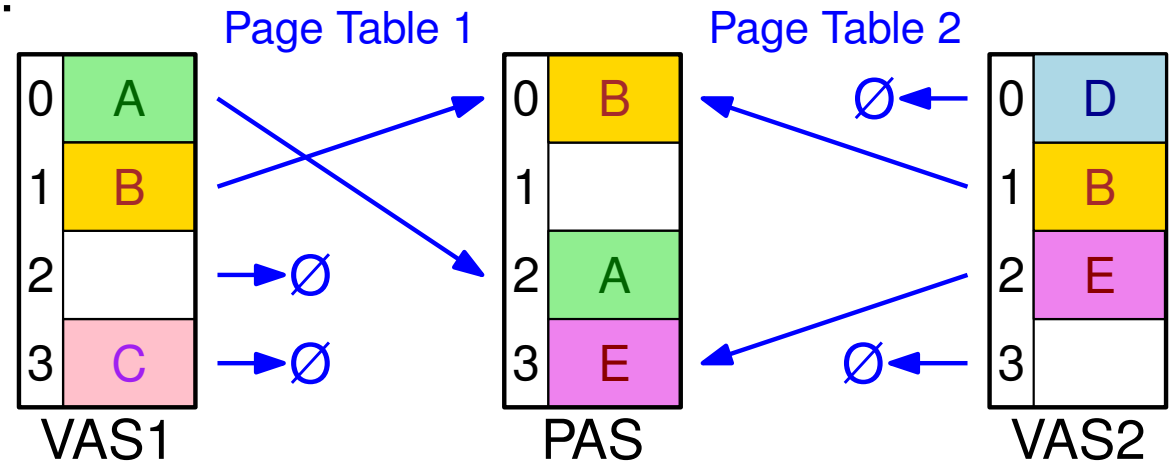

Remarques :

- RAM allouée aux pages virtuelles «les plus utiles»
- MMU reconfigurée à chaque changement de contexte
- permet aussi d'isoler le noyau vs processus userland

# Isolation vs partage : quelques cas particuliers

Rappel : chaque processus dispose d'un VAS distinct...
...mais une même page peut être présente dans plusieurs VAS !

Exemple :

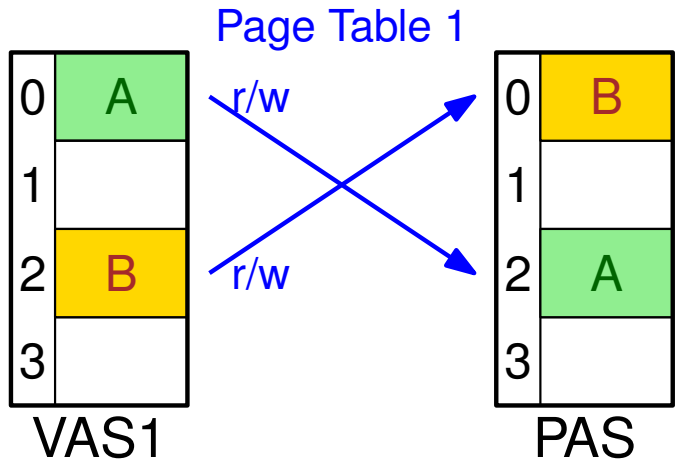



Projection des pages

- en lecture seule : par ex, le fichier exécutable
  - ne consomme aucun espace supplémentaire sur le disque
- en lecture/écriture : mémoire partagée
  - permet la communication entre plusieurs processus
- en copy-on-write : duplication paresseuse
  - évite de stocker deux fois des pages identiques

# Copy-On-Write vs `fork()`

Idée : ne pas dupliquer les données immédiatement, mais attendre qu'une des deux copies soit effectivement modifiée

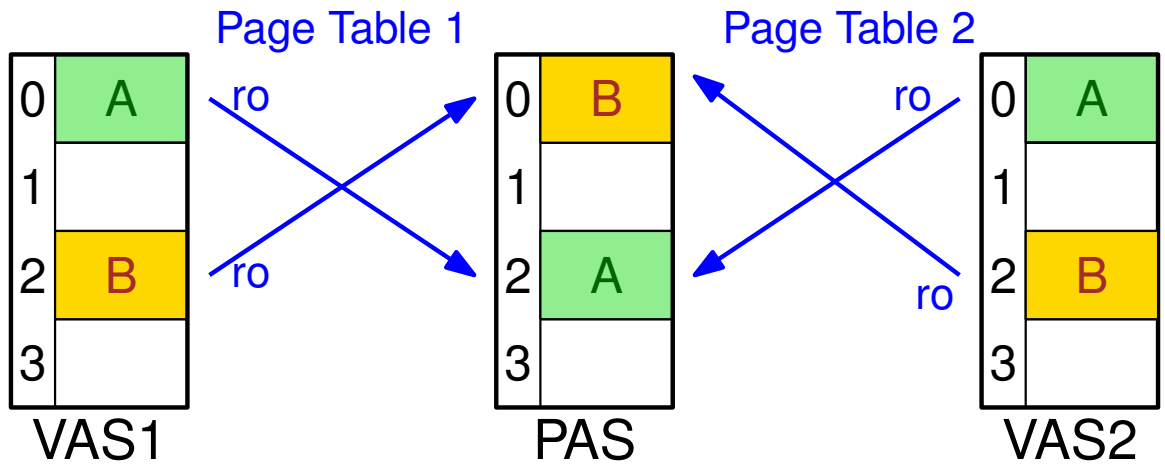



Imaginons un certain processus P1 avec deux pages A et B :
on note PCB1 son *Process Control Block*,
et on note PT1 sa table de pagination

Initialement, les deux pages sont accessibles en lecture-écriture

# Copy-On-Write vs `fork()`

Idée : ne pas dupliquer les données immédiatement, mais attendre qu'une des deux copies soit effectivement modifiée

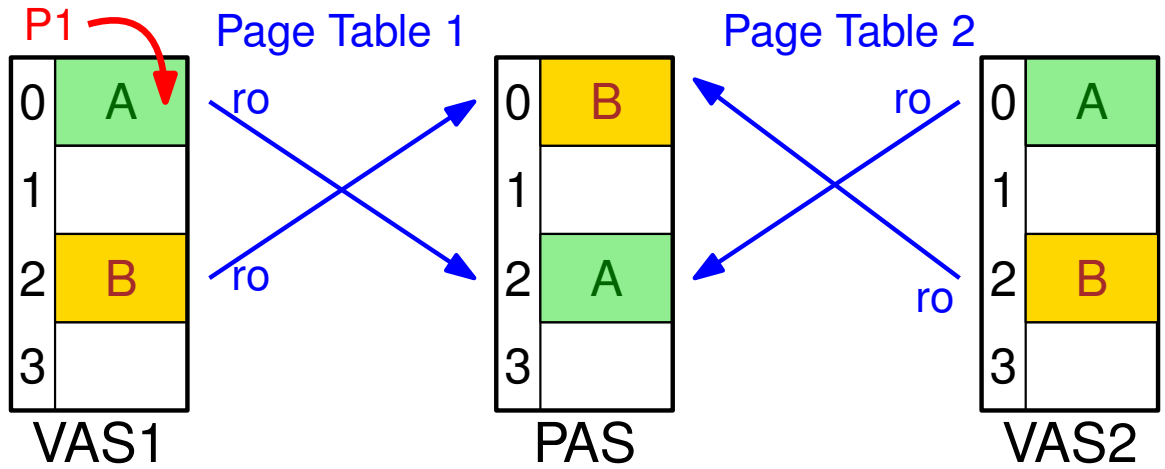



P1 fait un `fork()`, donnant naissance à P2. Le noyau :

- duplique PCB et table de pagination
- marque *tous* les PTE (de PT1 et de PT2) en lecture seule
- marque dans les deux PCB ces pages comme *copy-on-write*

# Copy-On-Write vs `fork()`

Idée : ne pas dupliquer les données immédiatement, mais attendre qu'une des deux copies soit effectivement modifiée

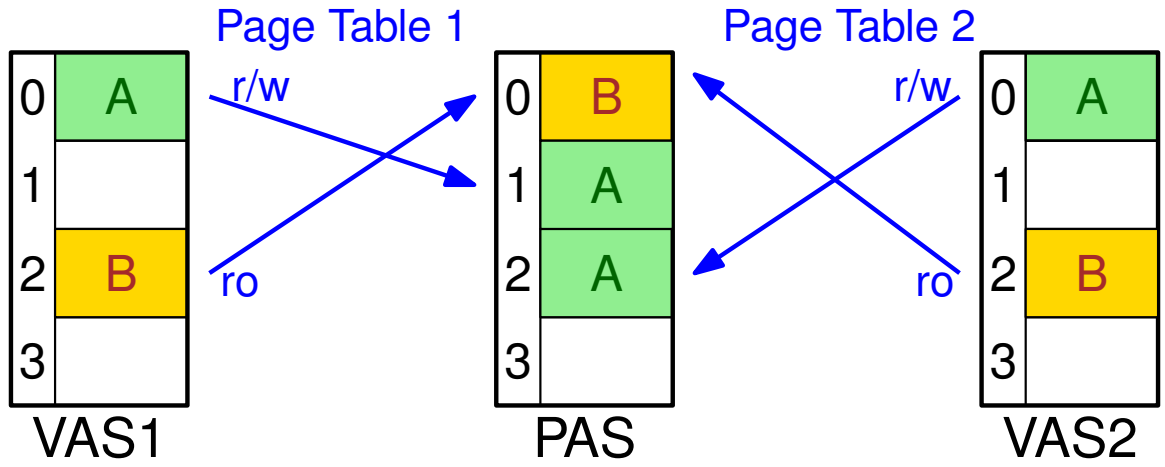



Lorsque P1 essaye d'écrire dans sa page n° 0 :

- MMU trouve un PTE en lecture seule
- lève une interruption («faute de protection»)

# Copy-On-Write vs `fork()`

Idée : ne pas dupliquer les données immédiatement, mais attendre qu'une des deux copies soit effectivement modifiée

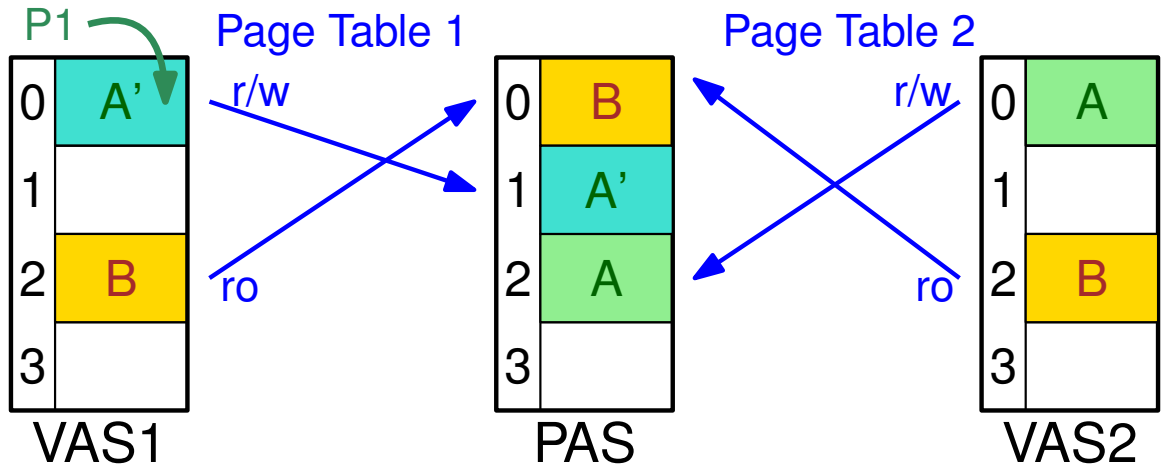



Le noyau constate dans PCB1 que la page est en copy-on-write

- il duplique finalement le contenu de la page
- et modifie les deux PT+PCB pour autoriser lecture/écriture
- puis, rend la main à P1

# Copy-On-Write vs `fork()`

Idée : ne pas dupliquer les données immédiatement, mais attendre qu'une des deux copies soit effectivement modifiée




P1 essaye à nouveau d'écrire dans sa page n° 0

- nouvelle donnée A' visible seulement depuis P1

En résumé : `fork()` «duplique» bien le VAS, mais ne copie pas les pages avant que ce soit vraiment utile

# Plan

# À retenir : la mémore virtuelle

La hiérarchie mémoire

- plusieurs technologies aux latences très différentes (ns vs ms)
- pr. de localité des accès : adresses proches $\sim$ instants proches
- cache du processeur : géré en matériel

Traduction d'adresses

- découplage addresses CPU et adresses physiques
- VA/PA, VAS/PAS, MMU, PP/VP, PPN/VPN, PO, PT/PTE...

Pagination à la demande

- performance : faute de page (swap), copy-on-write
- multiprogrammation : isolation des processus, mem partagée
- protection : user space vs kernel space, memory-mapped I/O