

# Gestion de mémoire: objectifs

- **Optimisation de l'utilisation de la mémoire principale = RAM**
- **Le plus grand nombre possible de processus actifs doit y être gardé, de façon à optimiser le fonctionnement du système en multiprogrammation**
  - ◆ garder le système le plus occupé possible, surtout l'UC
  - ◆ s'adapter aux besoins de mémoire de l'utilisateur
    - 👉 allocation dynamique au besoin

# Gestion de la mémoire

---

Dans ce chapitre nous verrons que, pour optimiser l'utilisation de la mémoire, les programmes sont éparpillés en mémoire selon des méthodes différentes: Pagination, segmentation

# Gestion de la mémoire: concepts dans ce chapitre

- **Adresse physique et adresse logique**
  - ◆ mémoire physique et mémoire logique
- **Remplacement**
- **Allocation contiguë**
  - ◆ Partitions fixes
  - ◆ Partitions variables
- **Pagination**
- **Segmentation**
- **Segmentation et pagination combinées**

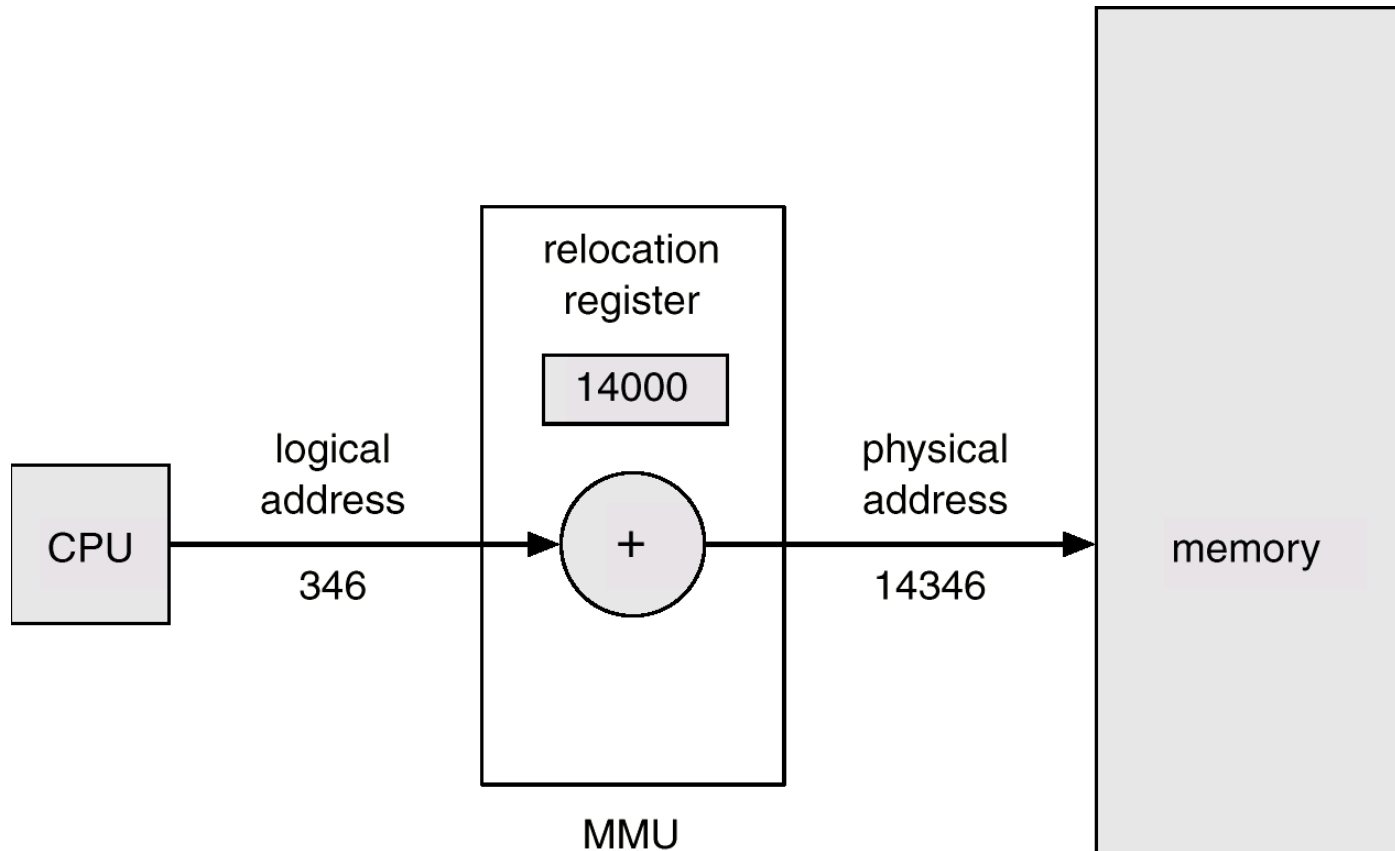
# Application de ces concepts

- **Pas tous les concepts de ce chapitre sont effectivement utilisés tels quels aujourd'hui dans la gestion de mémoire centrale**
- **Cependant plusieurs se retrouvent dans le domaine de la gestion de mémoires auxiliaires, surtout disques**

# Mémoire/Adresses physiques et logiques

- **Mémoire physique:**
  - ◆ la mémoire principale RAM de la machine
- **Adresses physiques: les adresses de cette mémoire**
- **Mémoire logique: l'espace d'adressage d'un programme**
- **Adresses logiques: les adresses dans cet espace**
- **Il faut séparer ces concepts car normalement, les programmes sont à chaque fois chargés à des positions différentes dans la mémoire**
  - ◆ Donc adresse physique  $\neq$  adresse logique

# Traduction adresses logiques → adr. physiques



MMU: unité de gestion de mémoire  
unité de traduction adresses  
(Memory Management Unit)

# Définition des adresses logiques

- ◆ une adresse logique est une adresse à une location de programme
  - ☞ par rapport au programme lui-même seulement
  - ☞ indépendante de la position du programme en mémoire physique

# Vue de l'utilisateur

- **Normalement, nous avons plusieurs types d'adressages p.ex.**
  - ◆ les adresses du programmeur (noms symboliques) sont traduites au moment de la compilation dans des adresses logiques
  - ◆ ces adresses sont traduites en adresses physiques, après chargement du programme en mémoire, par l'unité de traduction adresses (MMU)



# Aspects du chargement

- Pour un module de chargement c'est de trouver de l'espace libre en mémoire soit:
  - 1- contiguë ou
  - 2- non contiguë
- Traduire les adresses du programme et effectuer les liaisons par rapport aux adresses où le module est chargé.

# Traduction d'adresses logique → physique

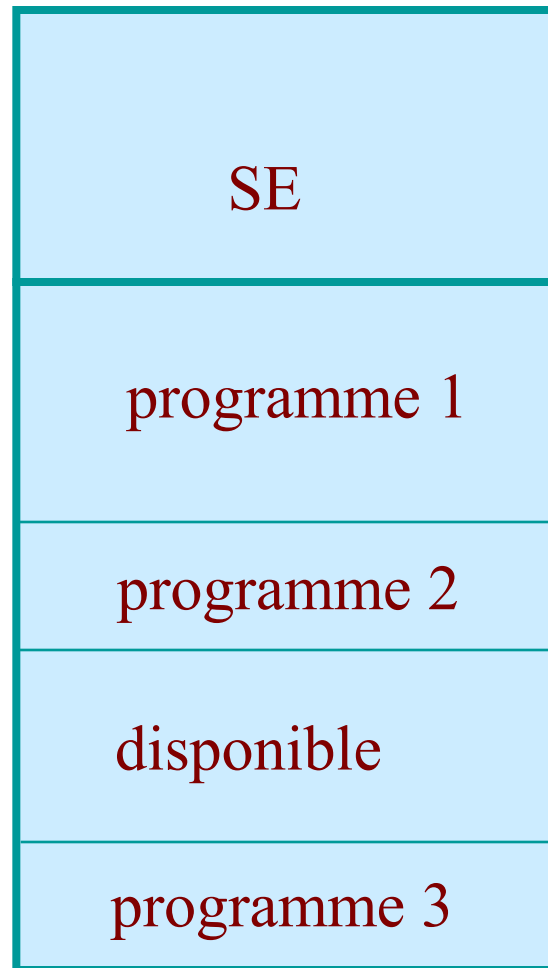
- Dans les premiers systèmes, un programme était toujours chargé aux mêmes adresses de mémoire
- La multiprogrammation et l'allocation dynamique ont engendré le besoin de lire un programme dans de différentes positions
- Au début, ceci était fait par le chargeur (loader) qui changeait les adresses avant de lancer l'exécution
- Aujourd'hui, ceci est fait par le MMU au fur et à mesure que le programme est exécuté
- Ceci ne cause pas d'hausse de temps d'exécution, car le MMU agit en parallèle avec autres fonctions d'UCT
  - ◆ P.ex. l'MMU peut préparer l'adresse d'une instruction en même temps que l'UCT exécute l'instruction précédente

# 1- Affectation contiguë de mémoire

**Affectation de tout le processus en un seul morceau en mémoire**

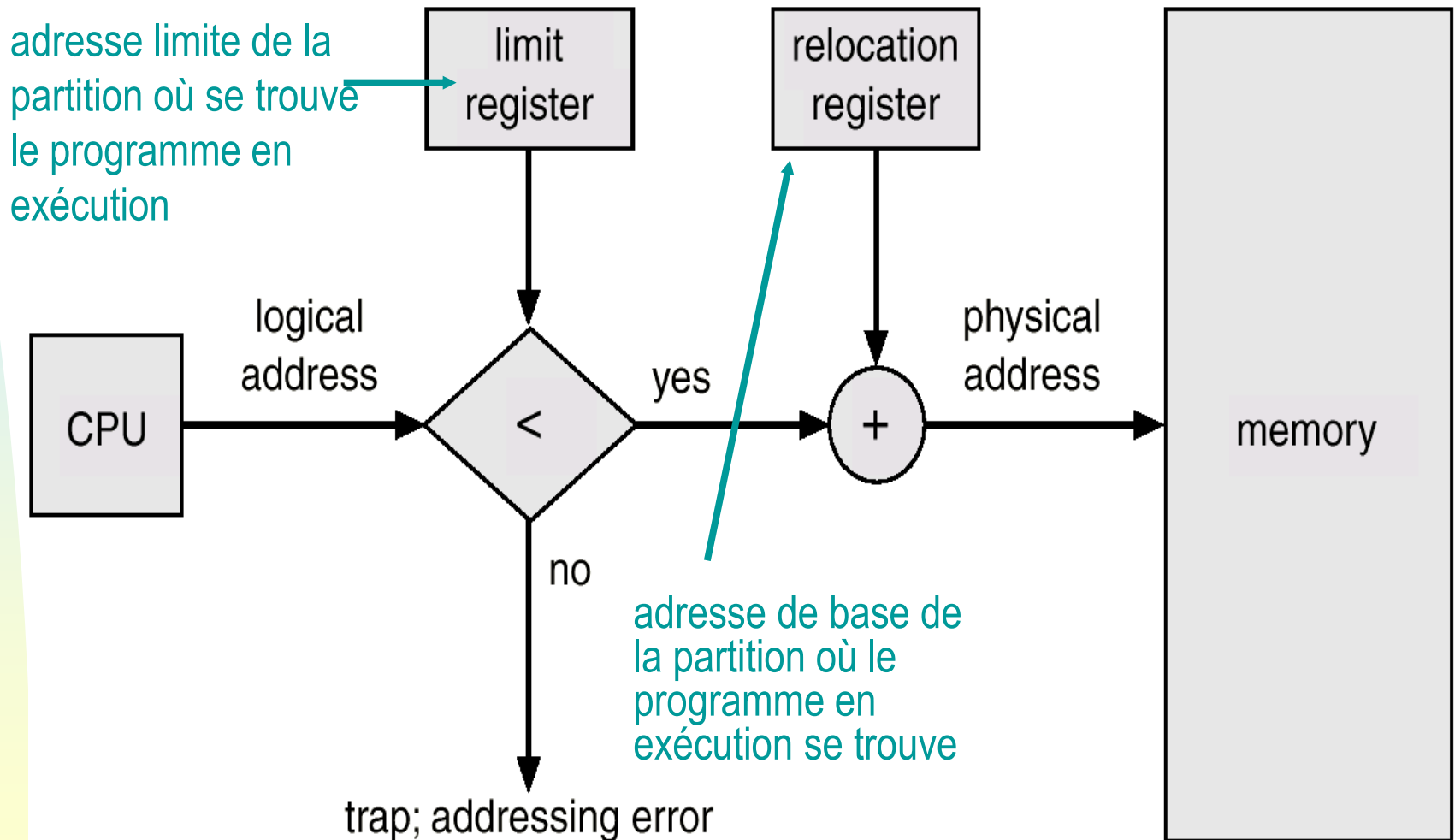
- **Nous avons plusieurs programmes à exécuter**
- **Nous pouvons les charger en mémoire les uns après les autres**
  - ◆ le lieu où un programme est chargé n'est connu que au moment du chargement
- **Besoins de matériel: registres translation et registres limites**

# Affectation contiguë de mémoire



Nous avons ici 4 **partitions** pour des programmes -  
chacun est lu dans une seule zone de mémoire

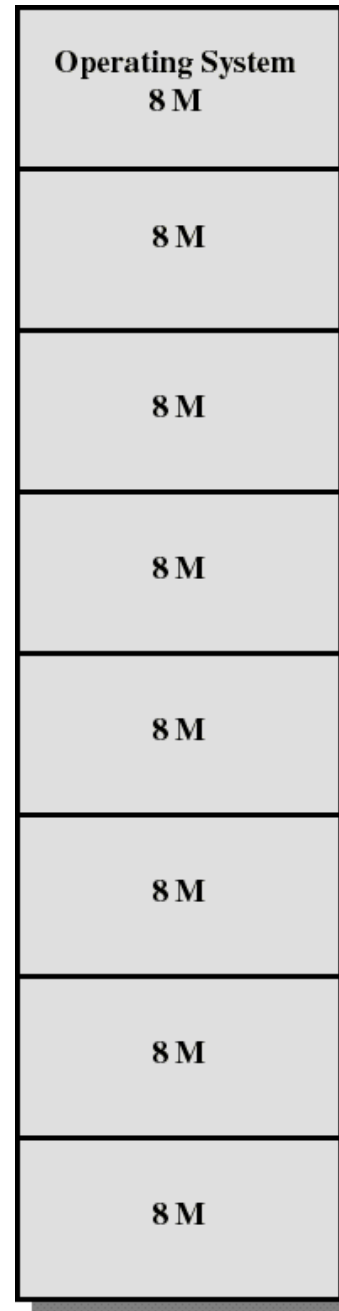
# Registres limites et translation dans MMU



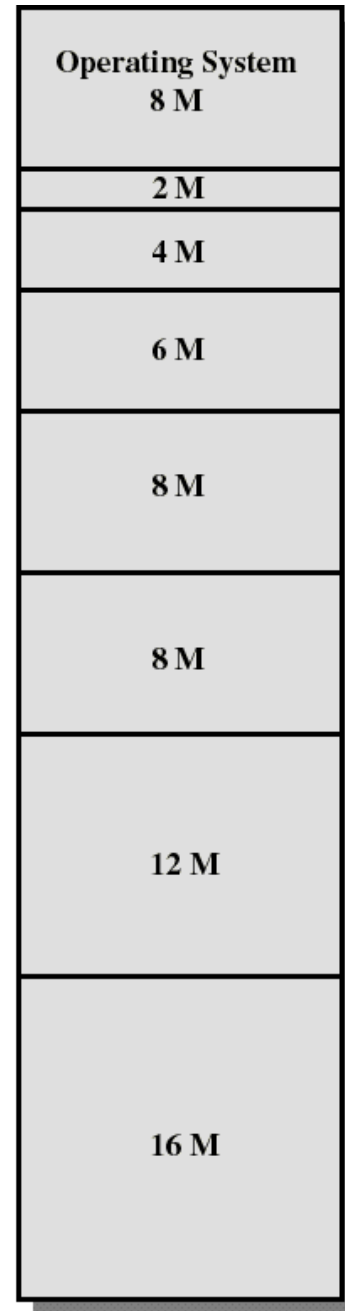
# Partitions fixes

## Première organisation de l'allocation contiguë

- Mémoire principale subdivisée en régions distinctes: **partitions**
- Les partitions sont soit de même taille ou de tailles inégales
- N'importe quel programme peut être affecté à une partition qui soit suffisamment grande



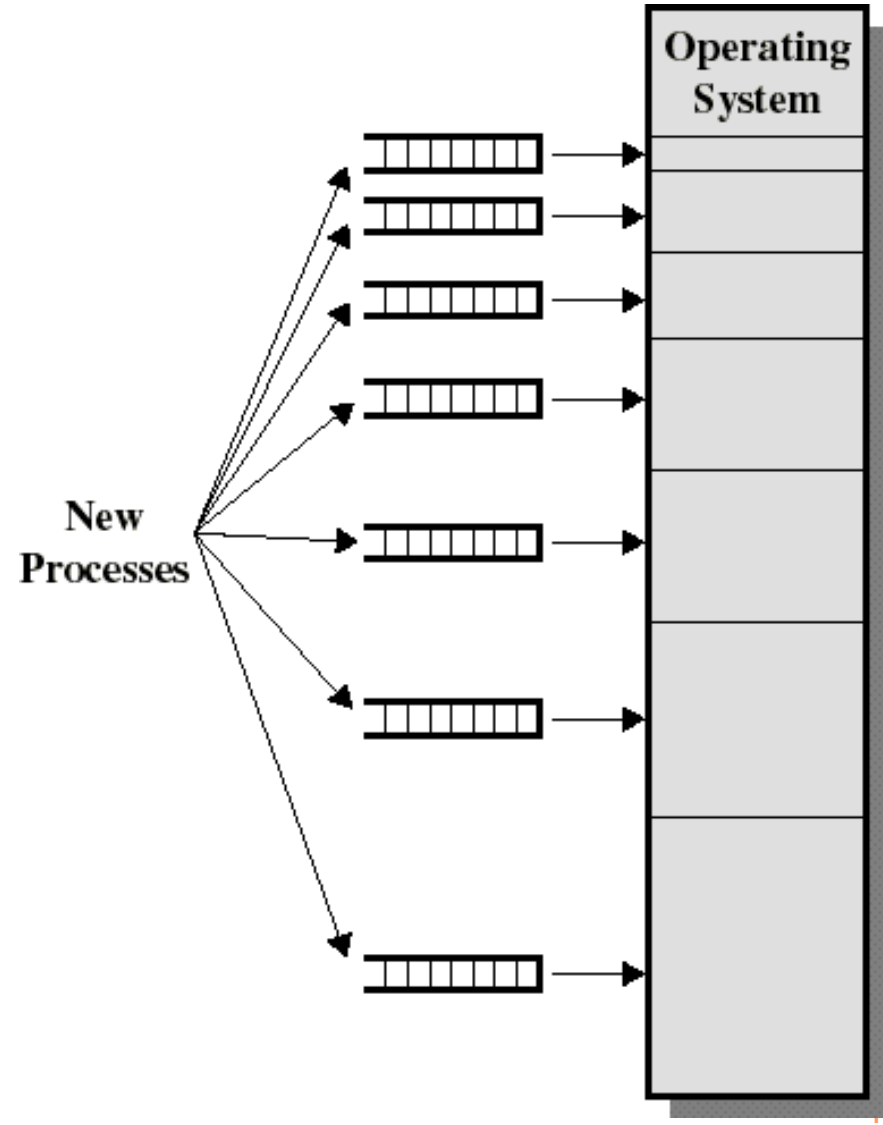
Equal-size partitions



Unequal-size partitions

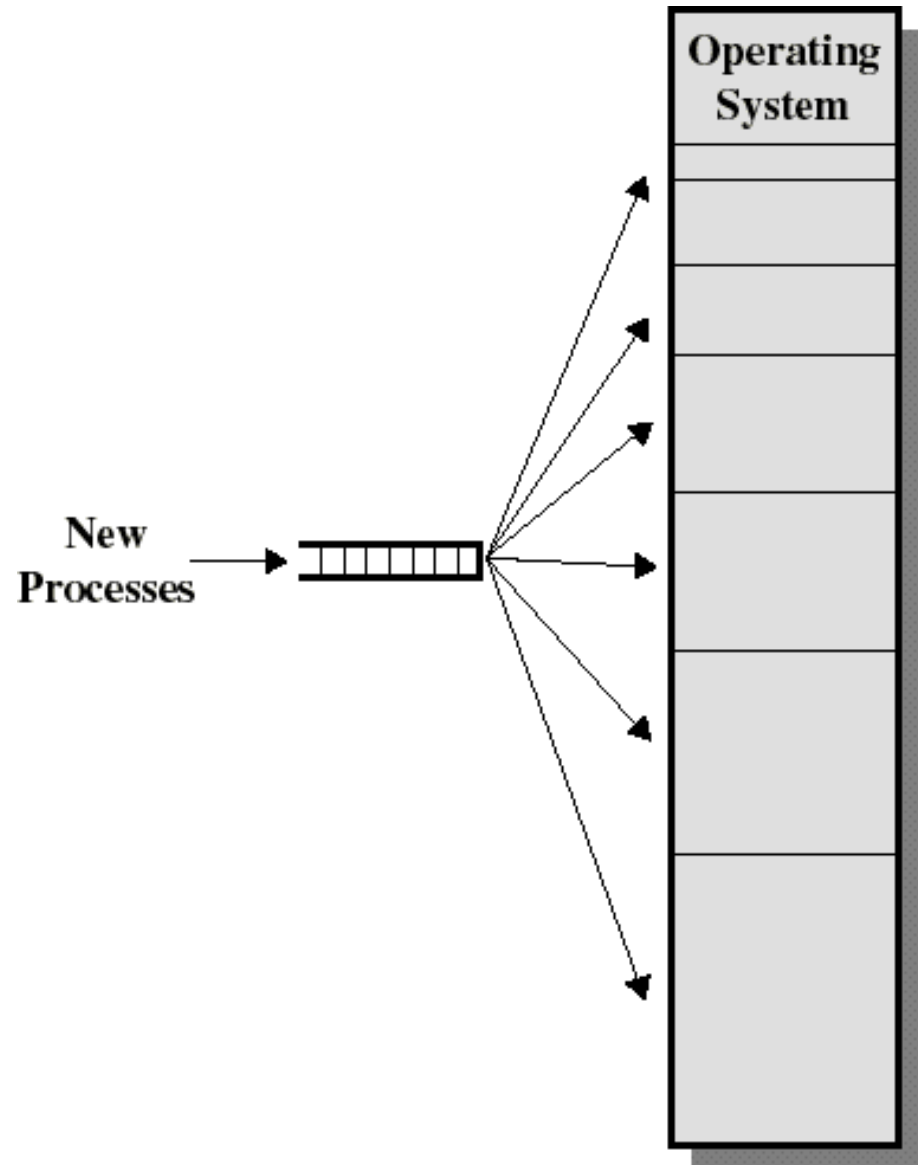
# Algorithme de placement pour partitions fixes

- **Partitions de tailles inégales: utilisation de plusieurs queues**
  - ◆ assigner chaque processus à la partition de la plus petite taille pouvant le contenir
  - ◆ 1 file par taille de partition
  - ◆ tente de minimiser la fragmentation interne
  - ◆ Problème: certaines files seront vides s'il n'y a pas de processus de cette taille (**fragementation externe**)



# Algorithme de placement pour partitions fixes

- **Partitions de tailles inégales: utilisation d'une seule file**
  - ◆ On choisit la plus petite partition libre pouvant contenir le prochain processus
  - ◆ le niveau de multiprogrammation augmente au profit de la fragmentation interne





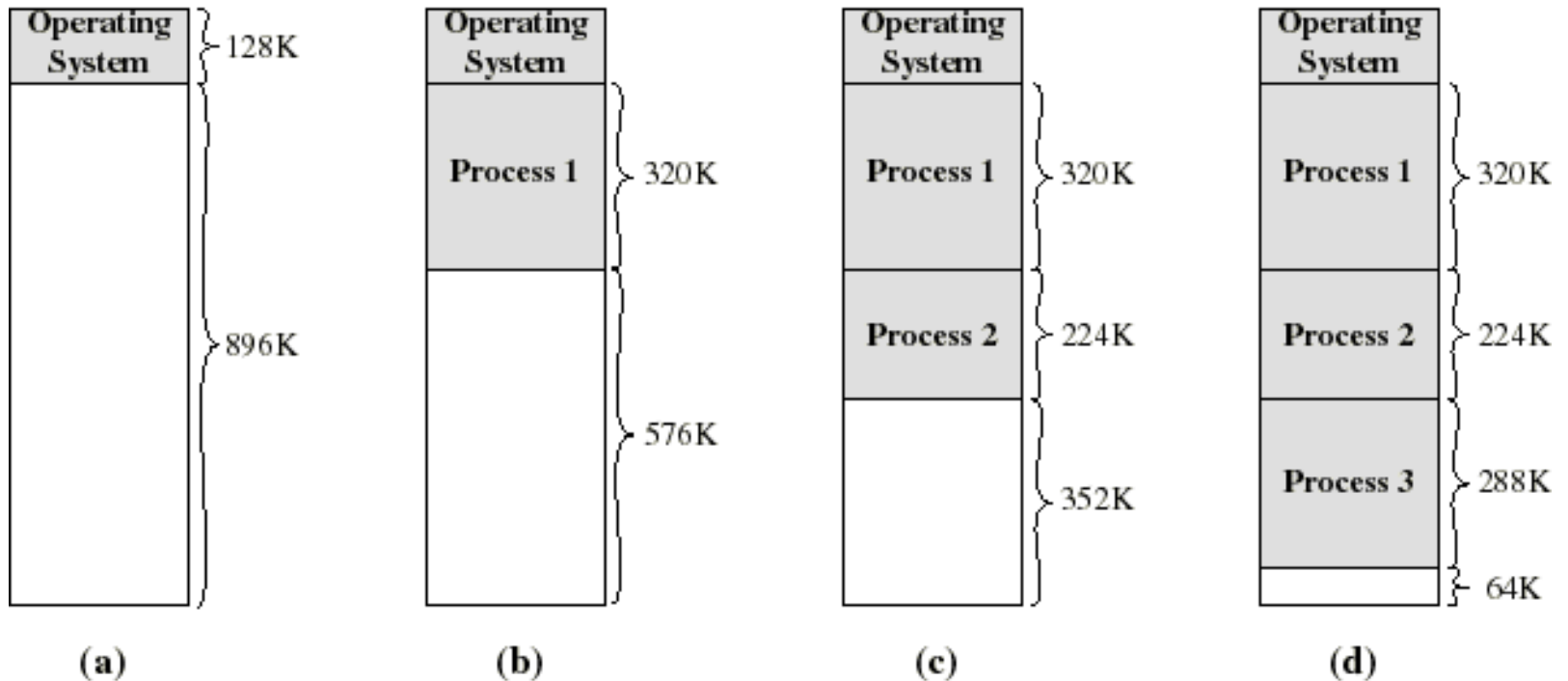
# Partitions fixes

- Simple, mais...
- Inefficacité de l'utilisation de la mémoire: tout programme, si petit soit-il, doit occuper une partition entière. Il y a **fragmentation interne**.
- Les partitions à tailles inégales atténue ces problèmes mais ils y demeurent...

# Partitions dynamiques

- **Partitions en nombre et tailles variables**
- **Chaque processus est alloué exactement la taille de mémoire requise**
- **Probablement des trous inutilisables se formeront dans la mémoire: c'est la fragmentation externe**

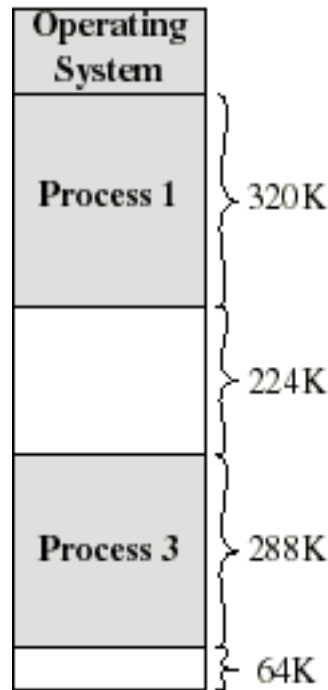
# Partitions dynamiques: exemple



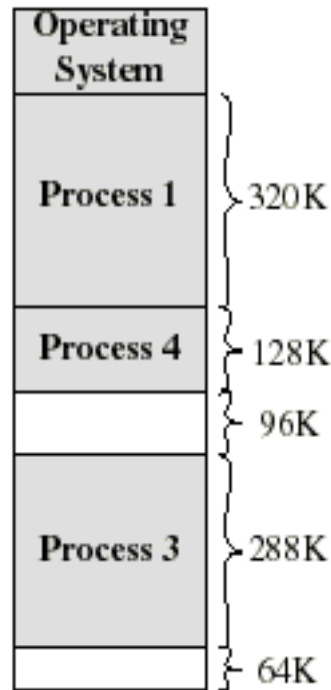
- (d) Il y a un trou de 64K après avoir chargé 3 processus: pas assez d'espace pour autre processus
- Si tous les processus se bloquent (p.ex. attente d'un événement), P2 peut être **permuté** et **P4=128K** peut être chargé.

Swapped out

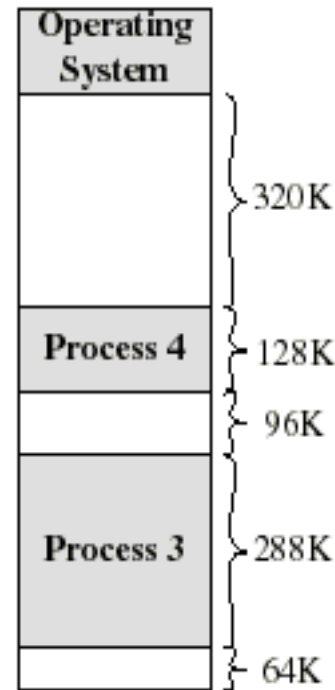
# Partitions dynamiques: exemple



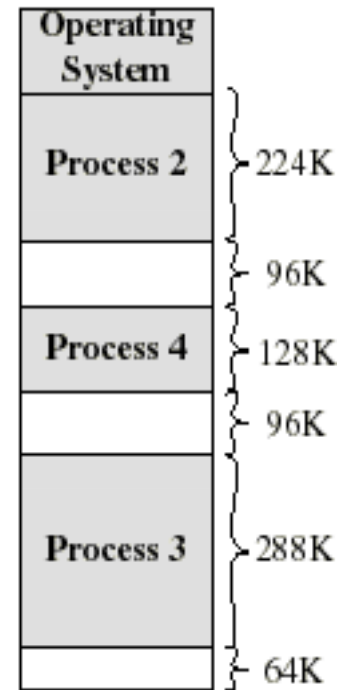
(e)



(f)



(g)



(h)

- (e-f) P2 est suspendu, P4 est chargé. Un trou de  $224-128=96K$  est créé (*fragmentation externe*)
- (g-h) P1 se termine ou il est suspendu, P2 est chargé à sa place: produisant un autre trou de  $320-224=96K$ ...
- Nous avons 3 trous petits et probablement inutiles.  $96+96+64=256K$  de fragmentation externe
- **COMPRESSION** pour en faire un seul trou de 256K

# Technique d'allocation de la mémoire

- Avant d'implanter une technique de gestion de la mémoire centrale par va-et-vient, il est nécessaire de connaître son état : les zones libres et occupées; de disposer d'une stratégie d'allocation et enfin de procédures de libération. Les techniques que nous allons décrire servent de base au va-et-vient; on les met aussi en œuvre dans le cas de la multiprogrammation simple où plusieurs processus sont chargés en mémoire et conservés jusqu'à la fin de leur exécution.

# État de la mémoire

- **Le système garde la trace des emplacements occupés de la mémoire par l'intermédiaire :**
  - ◆ D'une table de bits ou bien
  - ◆ D'une liste chaînée.

**La mémoire étant découpée en unités, en blocs, d'allocation**

# Tables de bits

On peut conserver l'état des blocs de mémoire grâce à une table de bits. Les unités libres étant notées par 0 et ceux occupées par un 1. (ou l'inverse).

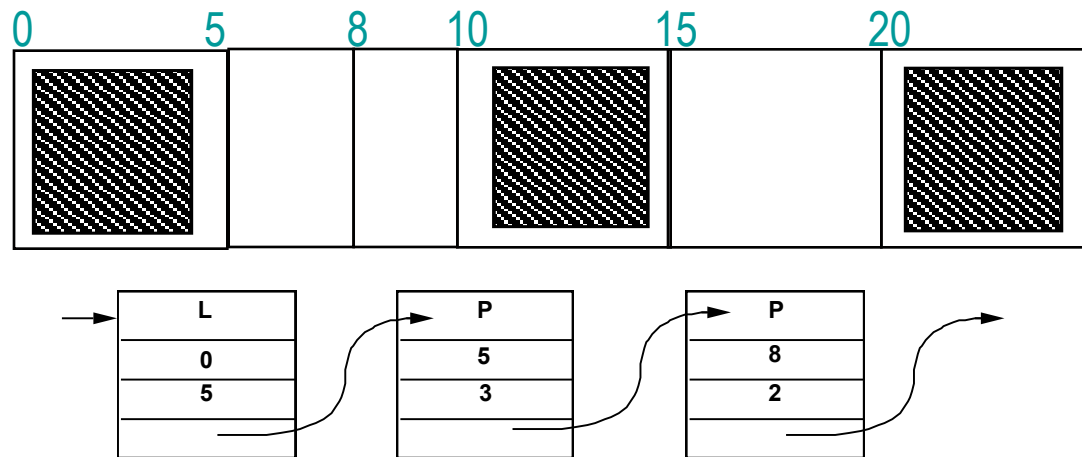
0	0	1	1	0	0									
---	---	---	---	---	---	--	--	--	--	--	--	--	--	--

La technique des tables de bits est simple à implanter, mais elle est peu utilisée. On peut faire la remarque suivante : plus l'unité d'allocation est petite, moins on a de pertes lors des allocations, mais en revanche, plus cette table occupe de place en mémoire.

# Listes chaînées

On peut représenter la mémoire par une liste chaînée de structures dont les membres sont :

- 👉 le type (libre ou occupé),
- 👉 l'adresse de début,
- 👉 la longueur, et
- 👉 un pointeur sur l'élément suivant.

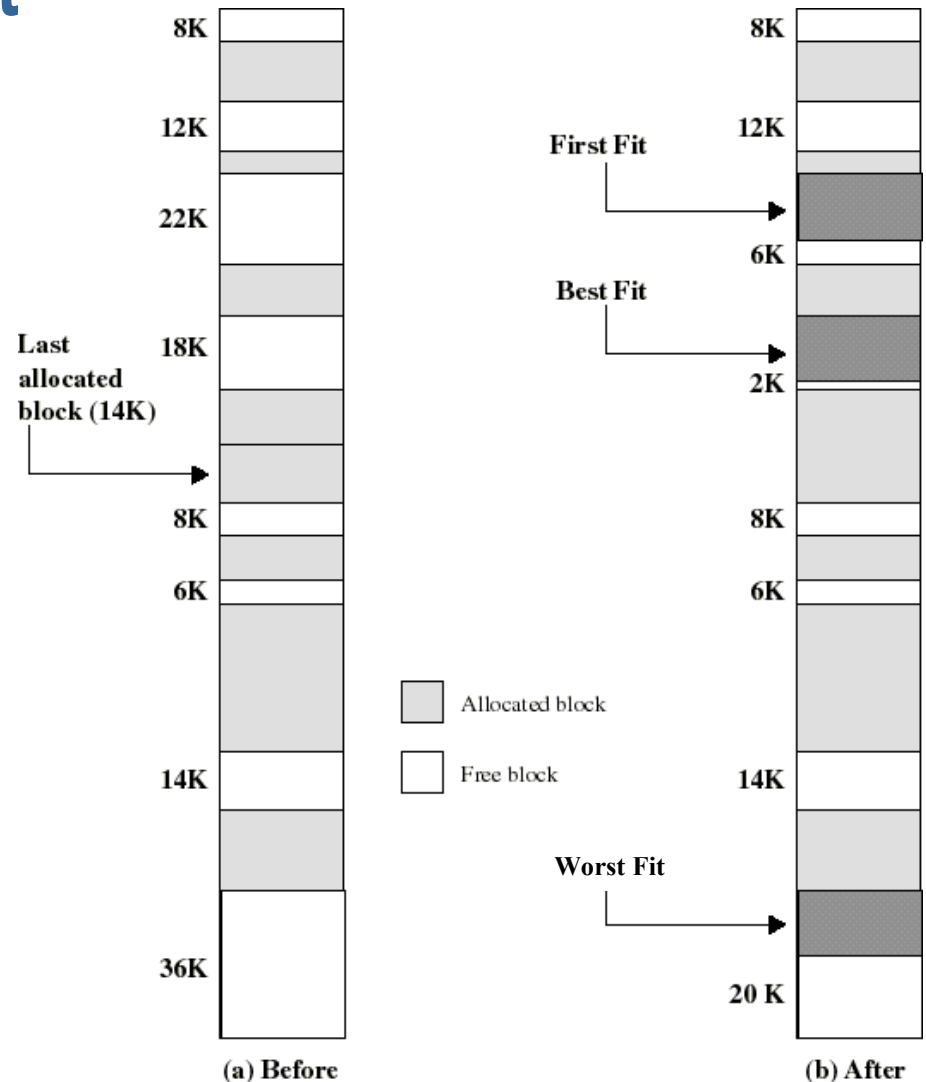


On peut légèrement modifier ce schéma en prenant deux listes : l'une pour les processus et l'autre pour les zones libres.



# Algorithmes de Placement

- pour décider de l'emplacement du prochain processus
- **But: réduire l'utilisation de la compression** (prend du temps...)
- **Choix possibles:**
  - ◆ **“Best-fit”**: choisir l'emplacement dont la taille est la plus proche
  - ◆ **“First-fit”**: choisir le 1er emplacement à partir du début
  - ◆ **“Worst-fit”**: choisir l'emplacement dont la taille est la plus loin



Example Memory Configuration Before and After Allocation of 16 Kbyte Block

# Algorithmes de placement: commentaires

- **Quel est le meilleur?**
  - ◆ critère principal: diminuer la probabilité de situations où un processus ne peut pas être servi, même s'il y a assez de mémoire...
- **La simulation montre qu'il ne vaut pas la peine d'utiliser les algorithmes les plus complexes... donc **first fit****
- **“Best-fit”: cherche le plus petit bloc possible: l'espace restant est le plus petit possible**
  - ◆ la mémoire se remplit de trous trop petits pour contenir un programme
- **“Worst-fit”: les allocations se feront souvent à la fin de la mémoire**

# Fragmentation: mémoire non utilisée

- **Un problème majeur dans l'affectation contiguë:**
  - ◆ Il y a assez d'espace pour exécuter un programme, mais il est fragmenté de façon non contiguë
    - ➡ **externe**: l'espace inutilisé est **entre** partitions
    - ➡ **interne**: l'espace inutilisé est **dans** les partitions

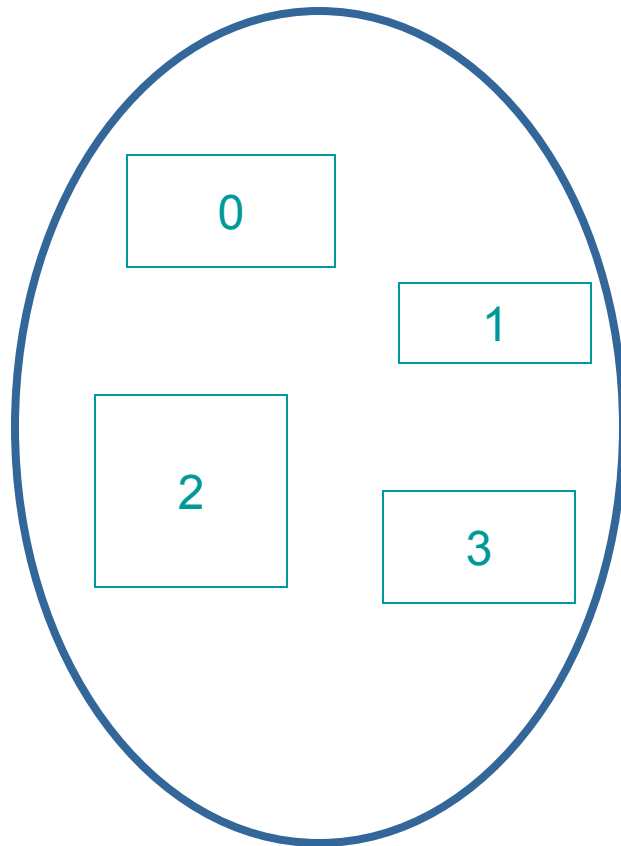
# Compaction

- **Une solution pour la fragmentation externe**
- **Les programmes sont déplacés en mémoire de façon à réduire à 1 seul grand trou plusieurs petits trous disponibles**
- **Effectuée quand un programme qui demande d'être exécuté ne trouve pas une partition assez grande, mais sa taille est plus petite que la somme des fragmentations externes existantes**
- **Désavantages:**
  - ◆ temps de transfert programmes
  - ◆ besoin de rétablir tous les liens entre adresses de différents programmes

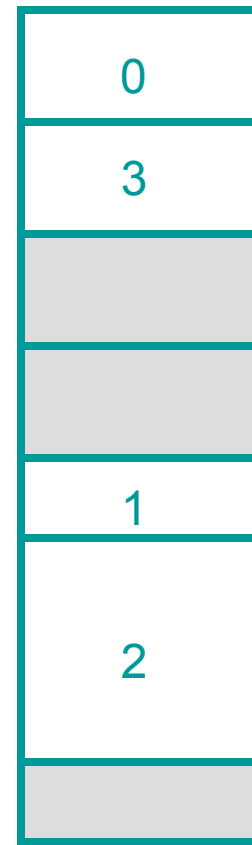
# Allocation non contiguë

- **A fin réduire le besoin de compression, le prochain pas est d'utiliser l'allocation non contiguë**
  - ◆ diviser un programme en morceaux et permettre l'allocation séparée de chaque morceau
  - ◆ les morceaux sont beaucoup plus petits que le programme entier et donc permettent une utilisation plus efficace de la mémoire
    - ☞ les petits trous peuvent être utilisés plus facilement
- **Il y a deux techniques de base pour faire ceci: la pagination et la segmentation**
  - ◆ la segmentation utilise des parties de programme qui ont une valeur logique (des modules)
  - ◆ la pagination utilise des parties de programme arbitraires (division du programmes en pages de longueur fixe).
  - ◆ elles peuvent être combinées

# Les segments comme unités d'alloc mémoire



espace usager

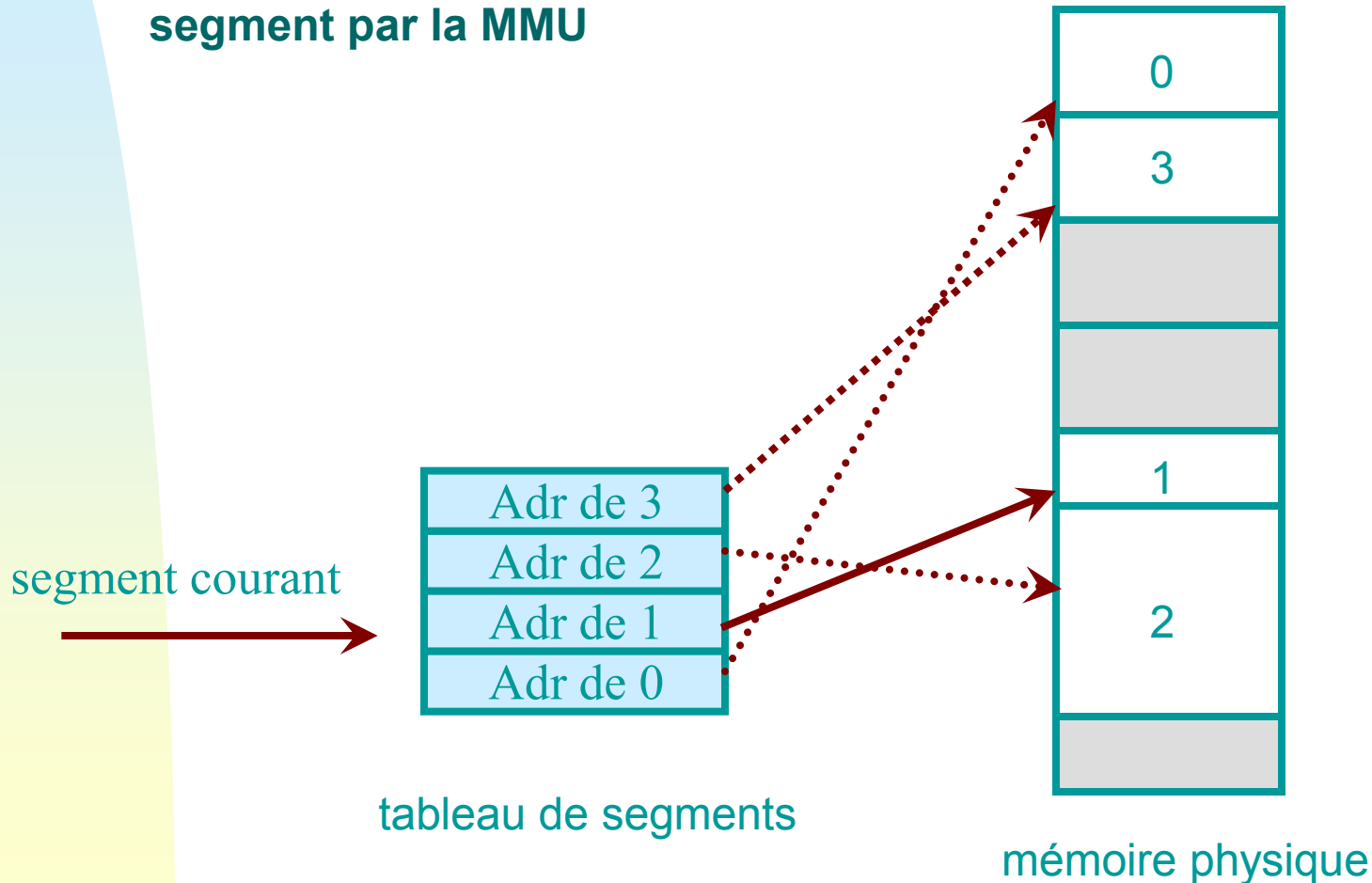


mémoire physique

Étant donné que les segments sont plus petits que les programmes entiers, cette technique implique moins de fragmentation (qui est externe dans ce cas)

# Mécanisme pour la segmentation

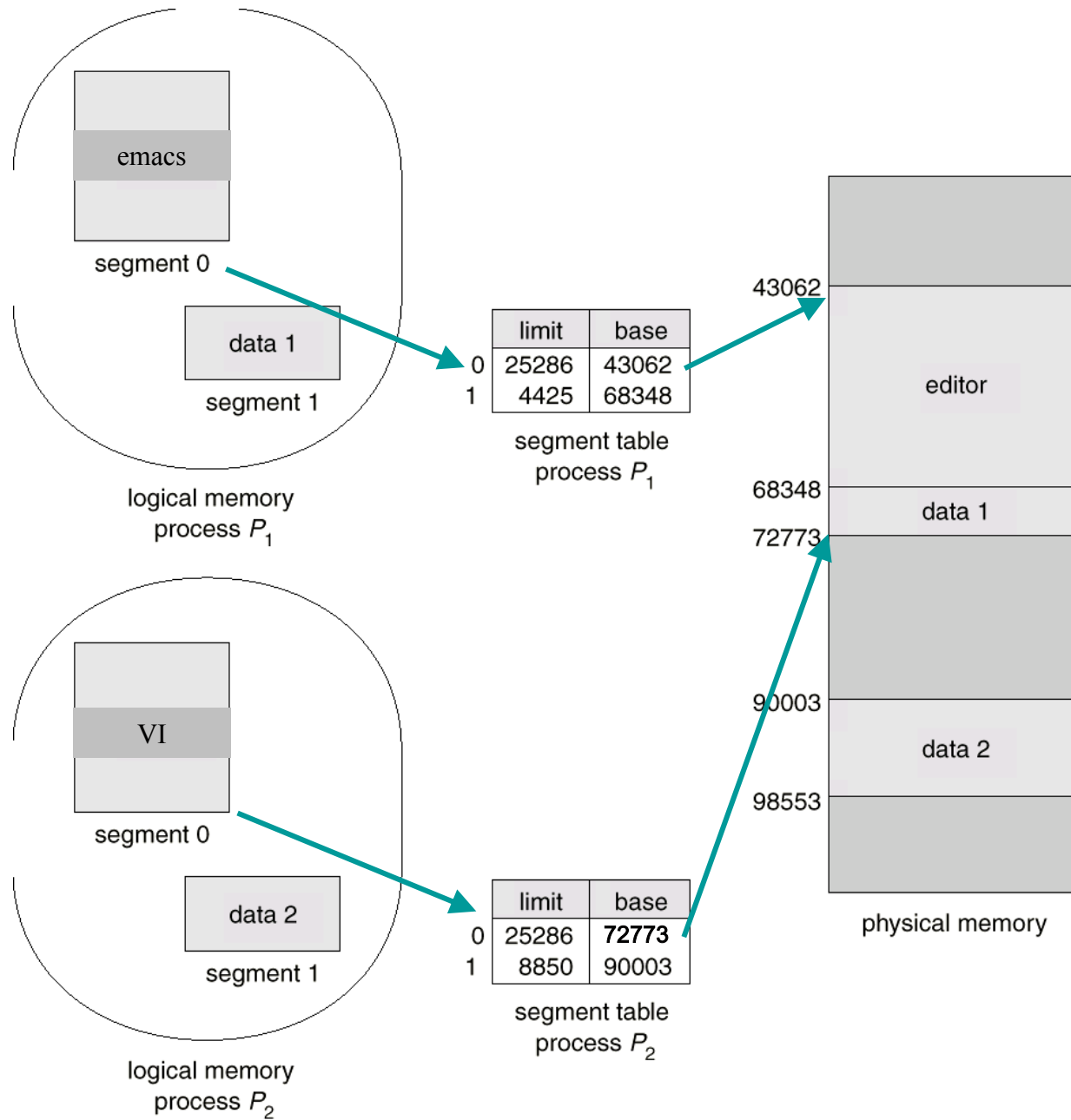
- Un tableau contient l'adresse de début de tous les segments dans un processus
- Chaque adresse dans un segment est ajoutée à l'adresse de début du segment par la MMU



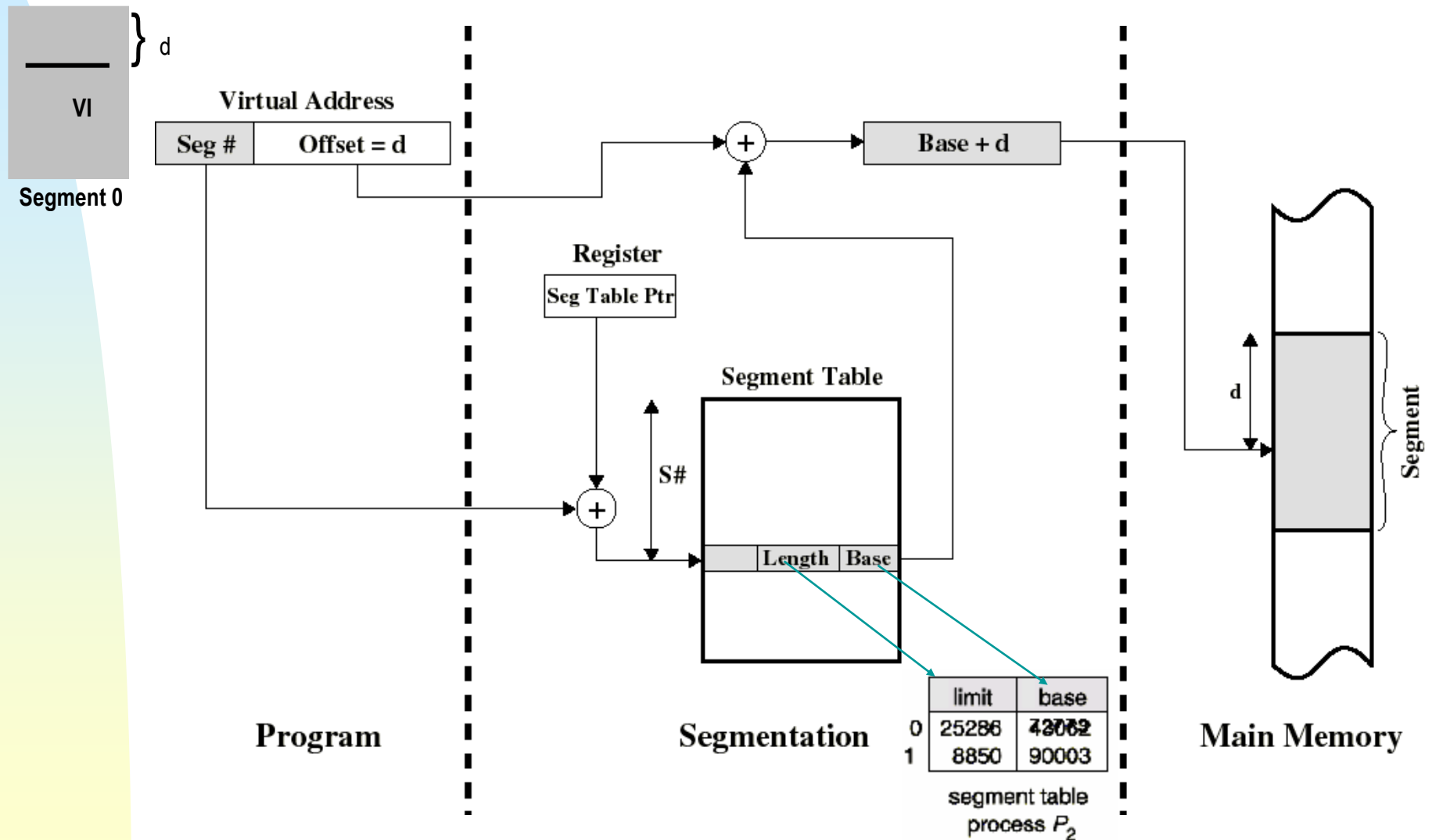
# Détails

- L'adresse logique consiste d'une paire:  
    <No de segm, décalage>  
    où décalage est l'adresse *dans* le segment
- le tableau des segments contient: **descripteurs de segments**
  - ◆ adresse de base
  - ◆ longueur du segment
  - ◆ Infos de protection
- Dans le PCB du processus il y aura un pointeur à l'adresse en mémoire du tableau des segments
- Il y aura aussi là dedans le nombre de segments dans le processus
- Au moment de la commutation de contexte, ces infos seront chargées dans les registres appropriés d'UC



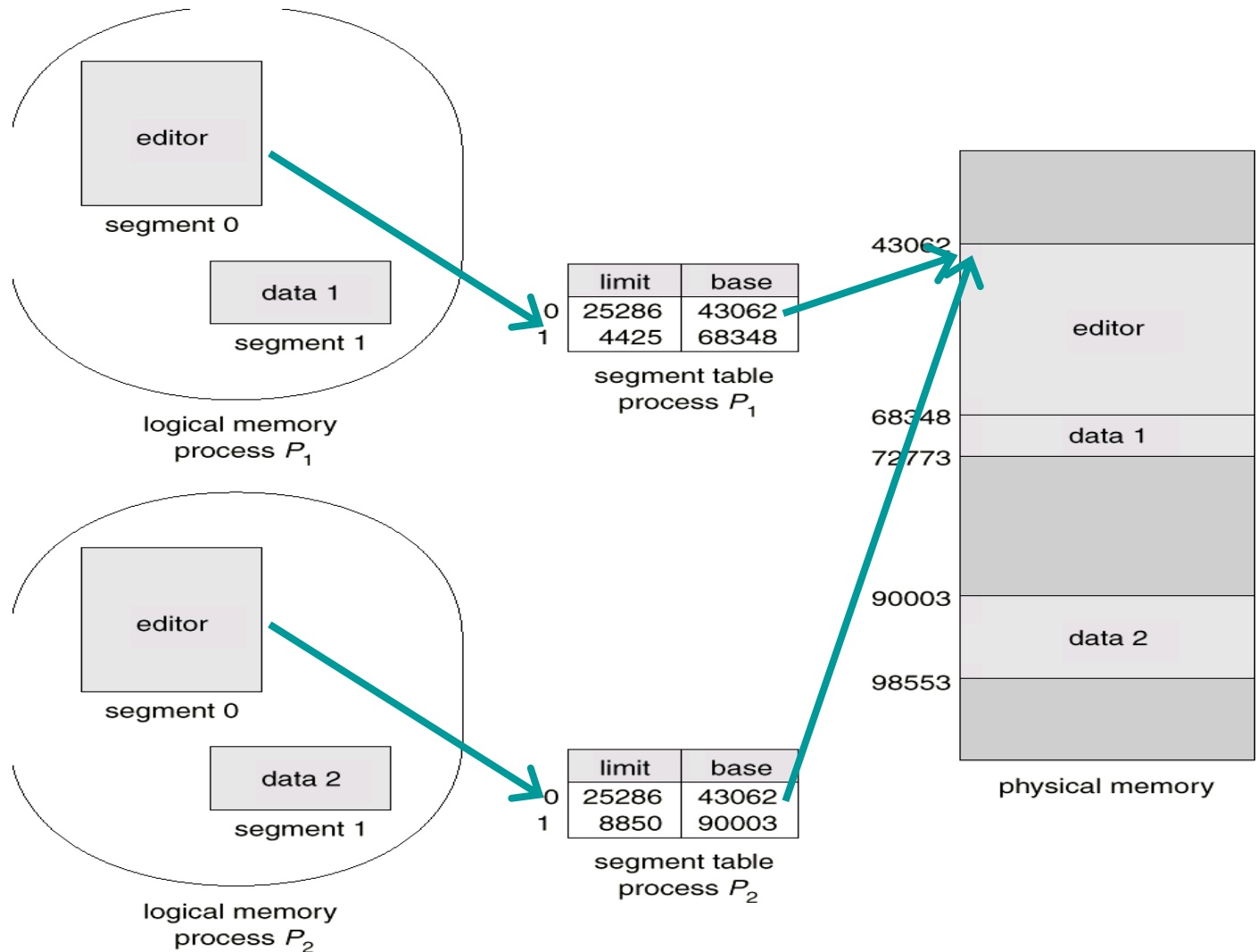


# Traduction d'adresses dans la segmentation



Aussi, si  $d > \text{longueur}$ : erreur!

# Partage de segments: le segment 0 est partagé



# Segmentation et protection

- **Chaque entrée dans la table des segments peut contenir des infos de protection:**
  - ◆ longueur du segment
  - ◆ privilèges de l'utilisateur sur le segment: lecture, écriture, exécution
    - ☞ Si au moment du calcul de l'adresse on trouve que l'utilisateur n'a pas droit d'accès → interruption
    - ☞ ces infos peuvent donc varier d'un utilisateur à autre, par rapport au même segment!

limite	base	read, write, execute?
--------	------	-----------------------

# Évaluation de la segmentation simple

- **Avantages: l'unité d'allocation de mémoire (segment) est**
  - ◆ plus petite que le programme entier
  - ◆ une entité logique connue par le programmeur
  - ◆ les segments peuvent changer de place en mémoire
  - ◆ la protection et le partage de segments sont faciles (en principe)
- **Désavantage: le problème des partitions dynamiques:**
  - ◆ La fragmentation externe n'est pas éliminée:
    - ☞ trous en mémoire, compression?
- **Une autre solution est d'essayer à simplifier le mécanisme en utilisant unités d'allocation mémoire de tailles égales**

☞ **PAGINATION**

# Segmentation contre pagination

- **Le problème avec la segmentation est que l'unité d'allocation de mémoire (le segment) est de longueur variable**
- **La pagination utilise des unités d'allocation de mémoire fixe, éliminant donc ce problème**

# Pagination simple

- La mémoire est partitionnée en petits morceaux de même taille: les **pages physiques** ou 'cadres' ou 'frames'
- Chaque processus est aussi partitionné en petits morceaux de même taille appelés **pages (logiques)**
- Les pages logiques d'un processus peuvent donc être assignés aux cadres disponibles n'importe où en mémoire principale
- **Conséquences:**
  - ◆ un processus peut être éparpillé n'importe où dans la mémoire physique.
  - ◆ la fragmentation **externe** est éliminée

# Exemple de chargement de processus

Frame number	Main memory
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

(a) Fifteen Available Pages

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

(b) Load Process A

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	
8	
9	
10	
11	
12	
13	
14	

(b) Load Process B

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

(d) Load Process C

- Supposons que le processus B se termine ou est suspendu



# Exemple de chargement de processus

- Nous pouvons maintenant transférer en mémoire un processus D, qui demande 5 cadres
  - ◆ bien qu'il n'y ait pas 5 cadres contigus disponibles
- La fragmentation externe est limitée au cas que le nombre de pages disponibles n'est pas suffisant pour exécuter un programme en attente
- Seule la dernière page d'un processus peut souffrir de **fragmentation interne**

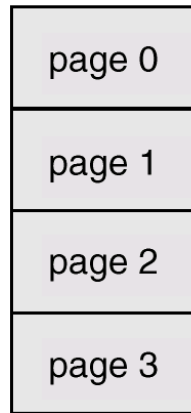
Main memory	
0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

(e) Swap out B

Main memory	
0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

(f) Load Process D

# Tableaux de pages

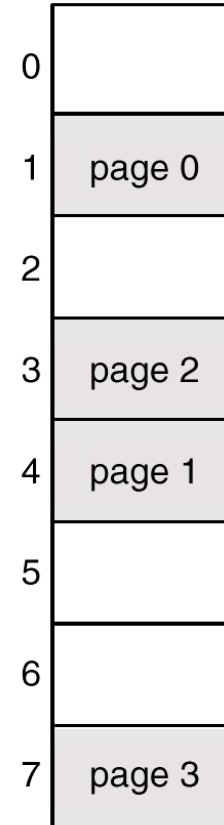


logical  
memory

0	1
1	4
2	3
3	7

page table

frame  
number



physical  
memory

# Tableaux de pages

0	0
1	1
2	2
3	3

Process A  
page table

0	—
1	—
2	—

Process B  
page table

0	7
1	8
2	9
3	10

Process C  
page table

0	4
1	5
2	6
3	11
4	12

Process D  
page table

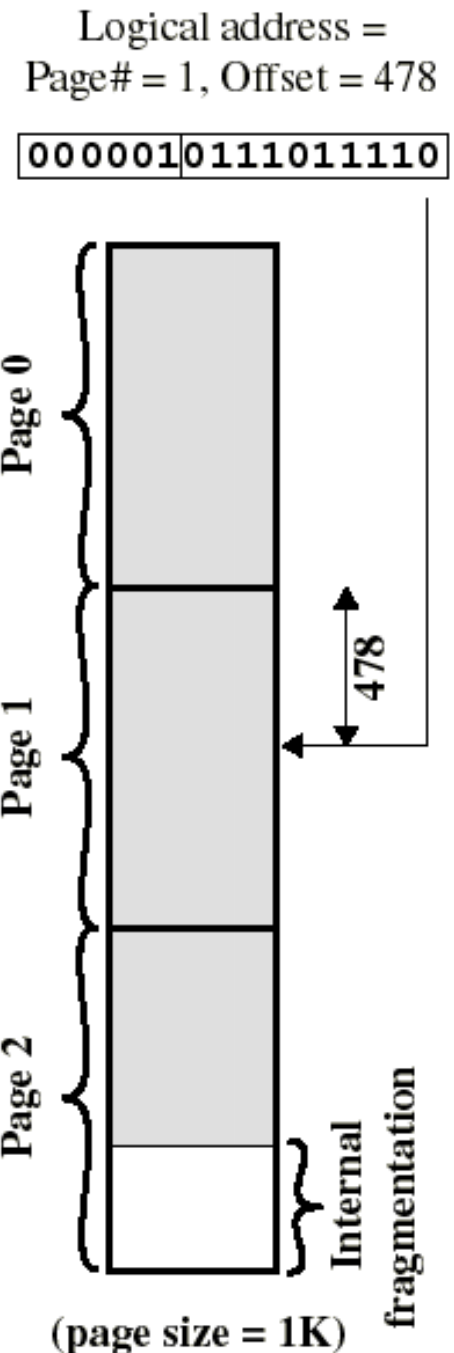
13
14

Free frame  
list

- Le SE doit maintenir une **table de pages** pour chaque processus
- Chaque entrée d'une table de pages contient le numéro de cadre où la page correspondante est physiquement localisée
- Une table de pages est indexée par le numéro de la page afin d'obtenir le numéro du cadre
- Une liste de cadres disponibles est également maintenue (free frame list)

# Adresse logique (pagination)

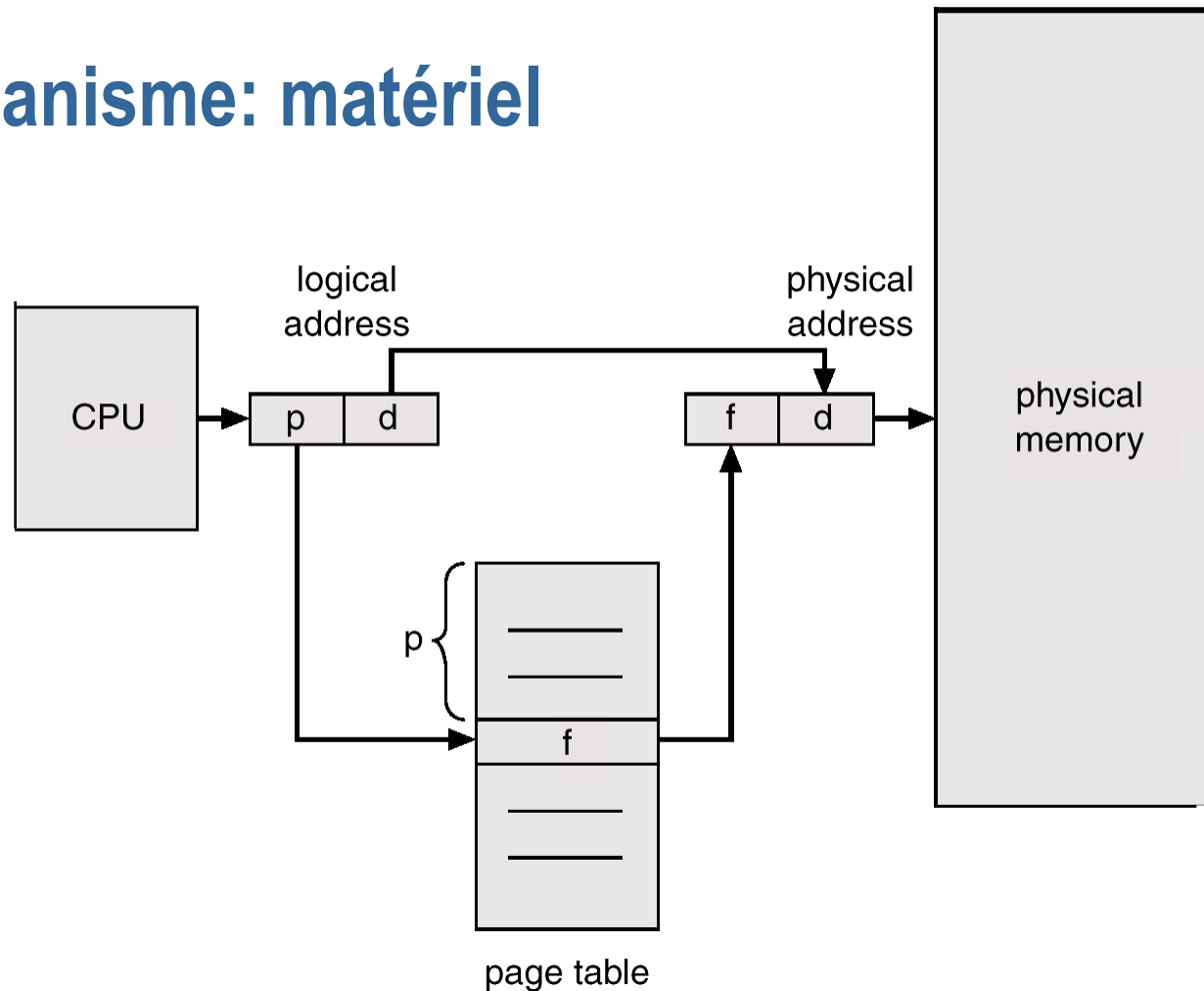
- L'adresse logique est facilement traduite en adresse physique car la taille des pages est une puissance de 2
- L'adresse logique (n,d) est traduite à l'adresse physique (k,d) en utilisant n comme index sur la table des pages et en le remplaçant par l'adresse k trouvée
  - ◆ d ne change pas



# Adresse logique (pagination)

- **Donc les pages sont invisibles au programmeur, compilateur ou assembleur (seule les adresses relatives sont employées)**
- **La traduction d'adresses au moment d'exécution est facilement réalisable par le matériel:**
  - ◆ l'adresse logique  $(n,d)$  est traduite en une adresse physique  $(k,d)$  en indexant la table de pages et en annexant le même décalage  $d$  au numéro du cadre  $k$
- **Un programme peut être exécuté sur différents matériels employant dimensions de pages différentes**

# Mécanisme: matériel



**Traduction d'adresses: segmentation et pagination**

Tant dans le cas de la segmentation, que dans le cas de la pagination, nous *ajoutons* le décalage à l'adresse du segment ou page.

# Deux petits problèmes

A) Considérez un système de 4 cadres ou pages physiques, chacune de 4 bytes. Les adresses sont de 4 bits, deux pour le numéro de page, et 2 pour le décalage. Le tableau de pages du processus en exécution est:

Numéro de page	Numéro de cadre
00	11
01	10
10	01
11	00

Considérez l'adresse logique 1010. Quelle sera l'adresse physique correspondante?

B) Considérez maintenant un système de segmentation, pas de pagination. Le tableau des segments du processus en exécution est comme suit:

Segment number	Base
00	110
01	100
10	000

Considérez l'adresse logique (no de seg, décalage)= (01, 01) , quelle est l'adresse physique?

# Segmentation simple vs Pagination simple

- La pagination se préoccupe seulement du problème du chargement, tandis que
- La segmentation est visible au programmeur mais la pagination ne l'est pas
- Le segment est une unité logique de protection et partage, tandis que la page ne l'est pas
  - ◆ Donc la protection et le partage sont plus aisés dans la segmentation
- La segmentation requiert un matériel plus complexe pour la traduction d'adresses (addition au lieu d'enchaînement)
- La segmentation souffre de fragmentation *externe* (partitions dynamiques)
- La pagination produit de fragmentation *interne*, mais pas beaucoup (1/2 cadre par programme)
- Heureusement, la segmentation et la pagination peuvent être combinées



# Récapitulation sur la fragmentation

- **Partition fixes:** fragmentation interne car les partitions ne peuvent pas être complètement utilisées + fragm. externe s'il y a des partitions non utilisées
- **Partitions dynamiques:** fragmentation externe qui conduit au besoin de compression.
- **Segmentation sans pagination:** pas de fragmentation interne, mais fragmentation externe à cause de segments de longueur différentes, stockés de façon contiguë (comme dans les partitions dynamiques)
- **Pagination:** en moyenne, 1/2 cadre de fragmentation interne par processus

# Mémoire Virtuelle

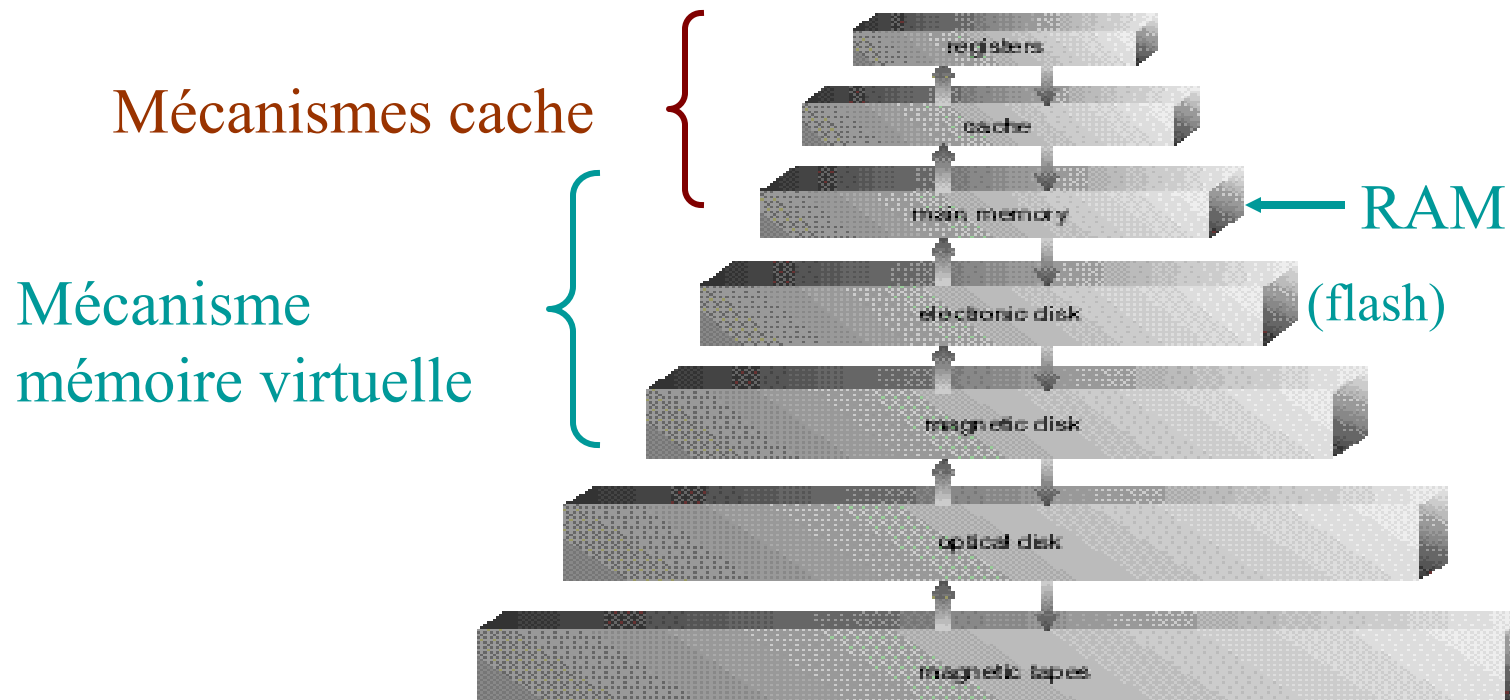
- **Pagination sur demande**
- **Problèmes de performance**
- **Algorithmes de remplacement de pages**
- **Allocation de cadres de mémoire**

# Concepts importants

- **Localité des références**
- **Mémoire virtuelle implémentée par va-et-vient des pages, mécanismes, défauts de pages**
- **Adresses physiques et adresses logiques**
- **Temps moyen d'accès à la mémoire**
  - ◆ Réécriture ou non de pages sur mém secondaire
- **Algorithmes de remplacement pages:**
  - ◆ OPT, LRU, FIFO, Horloge
  - ◆ Fonctionnement, comparaison
- **Écroulement, causes**
- **Relation entre la dimension de pages et le nombre d'interruptions**
- **Prépagination, post-nettoyage**
- **Effets de l'organisation d'un programme sur l'efficacité de la pagination**

# La mémoire virtuelle est une application du concept de hiérarchie de mémoire

- **C'est intéressant de savoir que des concepts très semblables s'appliquent aux mécanismes de la mémoire cache**
  - ◆ Cependant dans ce cas les mécanismes sont surtout de matériel



# La mémoire virtuelle

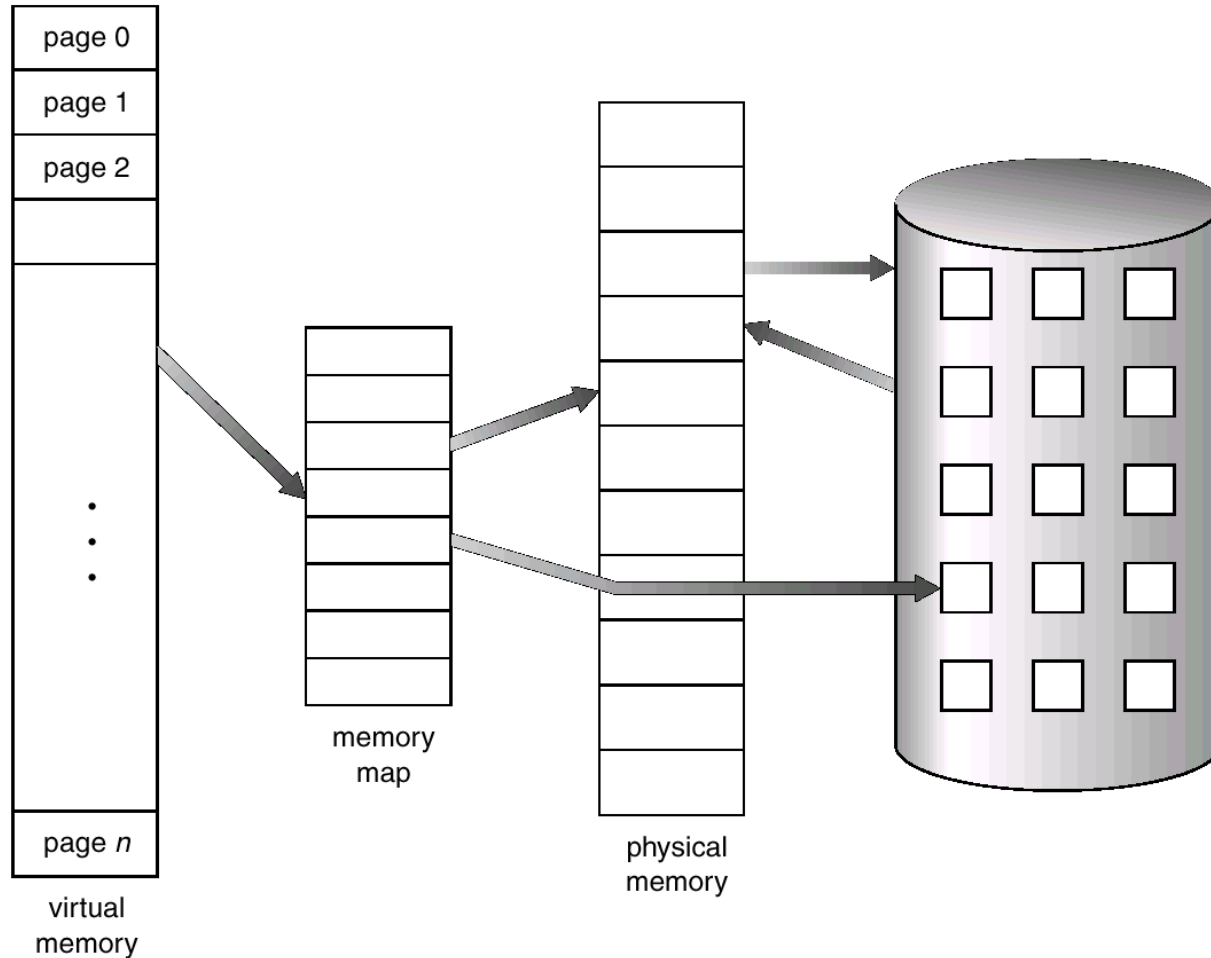
- À fin qu'un programme soit exécuté, il ne doit pas nécessairement être tout en mémoire centrale!
- Seulement **les parties qui sont en exécution** ont besoin d'être en mémoire centrale
- Les autres parties peuvent être sur mémoire secondaire (p.ex. disque), prêtes à être amenées en mémoire centrale sur demande
  - ◆ Mécanisme de va-et-vient ou swapping
- Ceci rend possible l'exécution de programmes beaucoup plus grands que la mémoire physique
  - ◆ Réalisant une **mémoire virtuelle** qui est plus grande que la mémoire physique

# De la pagination et segmentation à la mémoire virtuelle

- Un processus est constitué de **morceaux** (pages ou segments) ne nécessitant pas d'occuper une région contiguë de la mémoire principale
- Références à la mémoire sont traduites en adresses physiques au moment d'exécution
  - ◆ Un processus peut être déplacé à différentes régions de la mémoire, aussi mémoire secondaire!
- **Donc: tous les morceaux d'un processus ne nécessitent pas d'être en mémoire principale durant l'exécution**
  - ◆ L'exécution peut continuer à condition que la prochaine instruction (ou donnée) est dans un morceau se trouvant en mémoire principale
- **La somme des mémoires logiques des processus en exécution peut donc excéder la mémoire physique disponible**
  - ◆ Le concept de base de la mémoire virtuelle
- Une image de tout l'espace d'adressage du processus est gardée en mémoire secondaire (normal. disque) d'où les pages manquantes pourront être prises au besoin
  - ◆ Mécanisme de va-et-vient ou swapping

# Mémoire virtuelle:

résultat d'un mécanisme qui combine  
la mémoire principale et les mémoires secondaires



# Localité et mémoire virtuelle

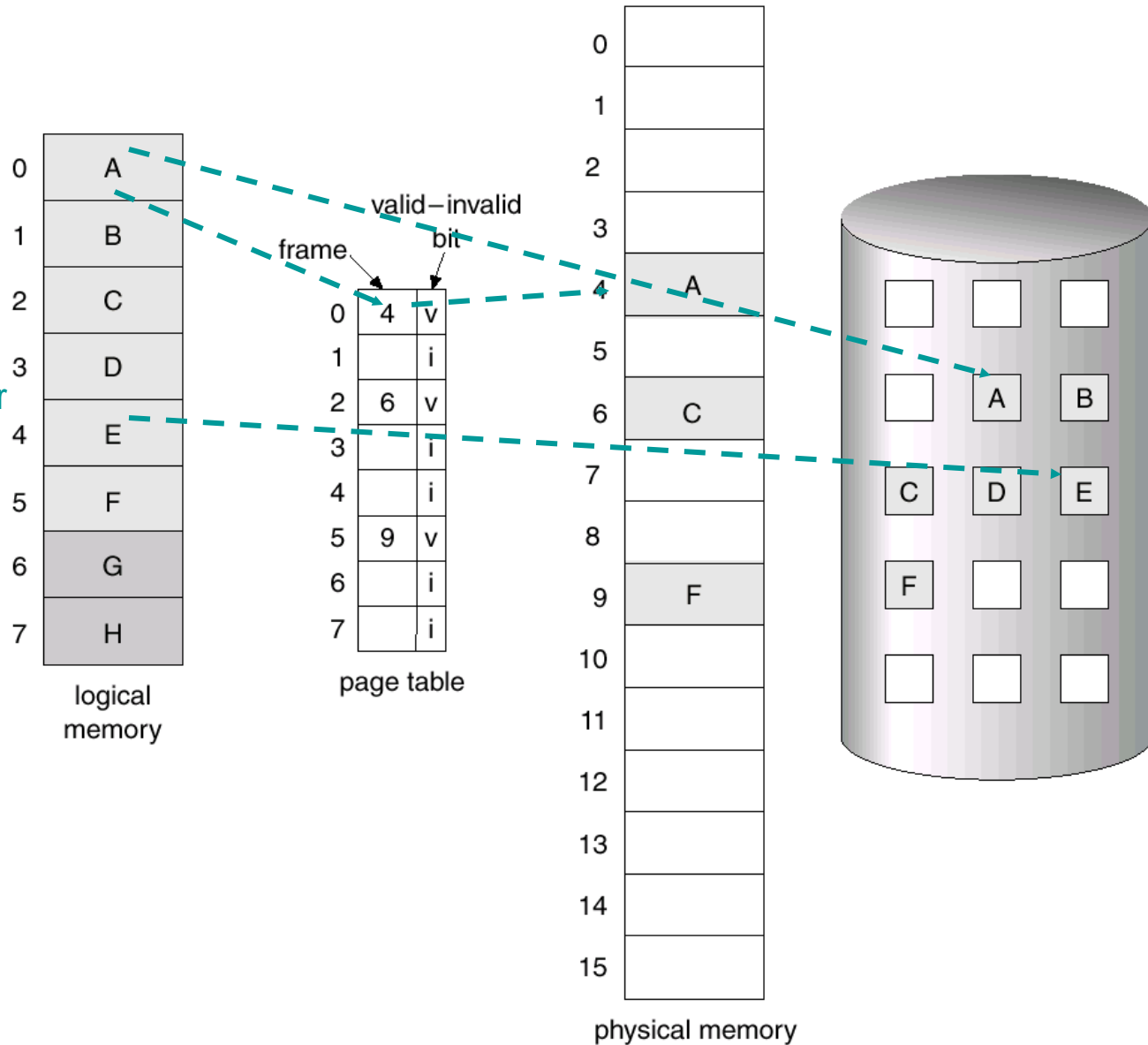
- Principe de **localité des références**: les références à la mémoire dans un processus tendent à se regrouper
- Donc: seule quelques pièces d'un processus seront utilisées durant une petite période de temps (pièces: pages ou segments)
- Il y a une bonne chance de “deviner” quelles seront les pièces demandées dans un avenir rapproché



# Pages en RAM ou sur disque

Page A en RAM et sur disque

Page E seulement sur disque



# Nouveau format du tableau des pages (la même idée peut être appliquée aux tableaux de segments)

Si la page est en RAM, ceci  
est une adr. de  
mém. principale  
sinon elle est une adresse de  
mémoire secondaire

Adresse de la page	Bit présent

bit *présent*  
1 si en RAM.,  
0 si sur Disque.

Au début, bit présent = 0 pour toutes les pages

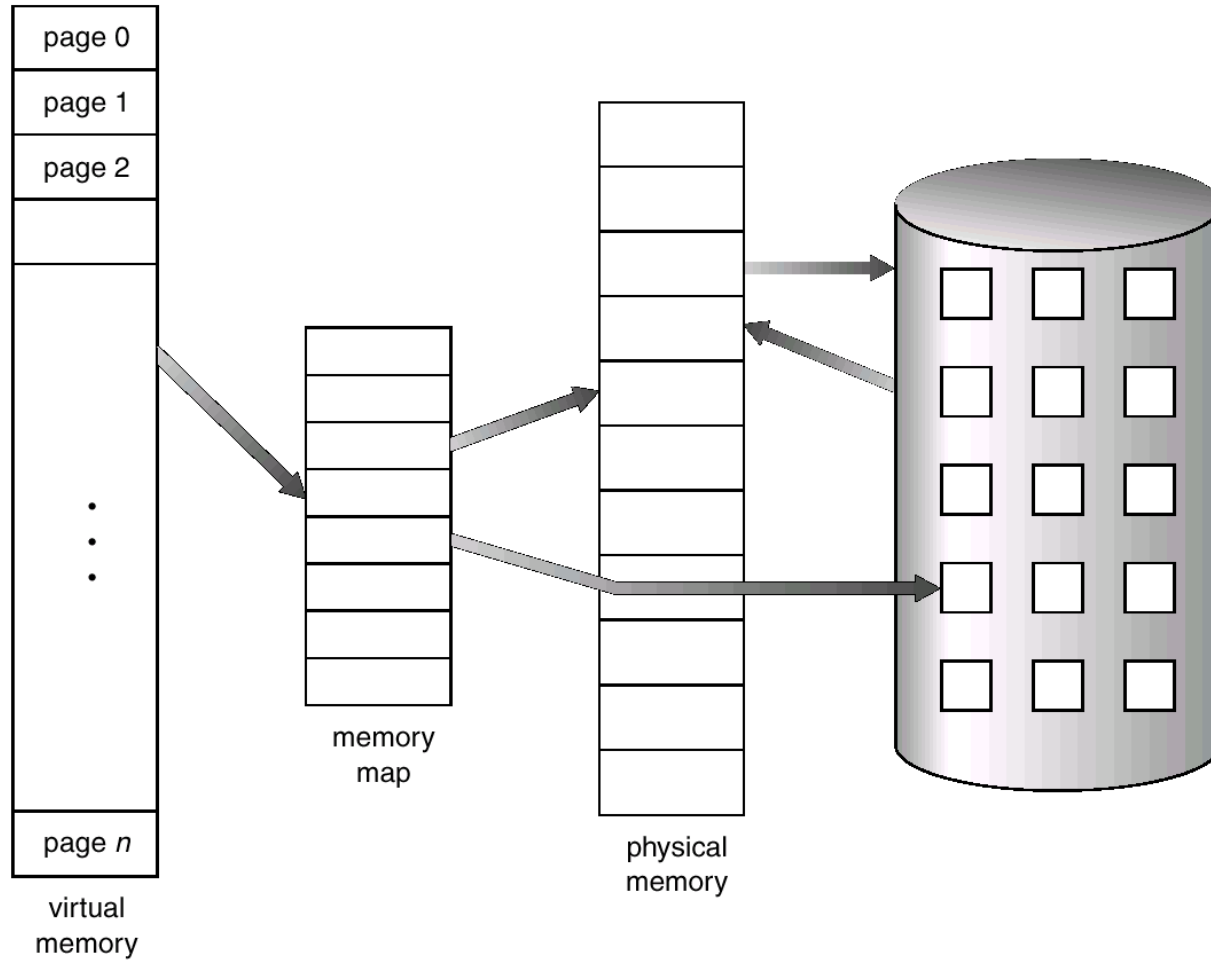
# Avantages du chargement partiel

- **Plus de processus peuvent être maintenus en exécution en mémoire**
  - ◆ Car seules quelques pièces sont chargées pour chaque processus
  - ◆ L'utilisateur est content, car il peut exécuter plusieurs processus et faire référence à des gros données sans avoir peur de remplir la mémoire centrale
  - ◆ Avec plus de processus en mémoire principale, il est plus probable d'avoir un processus dans l'état prêt, meilleure utilisation d'UCT
- **Plusieurs pages ou segments rarement utilisés n'auront peut être pas besoin d'être chargés du tout**
- **Il est maintenant possible d'exécuter un ensemble de processus lorsque leur taille excède celle de la mémoire principale**
  - ◆ Il est possible d'utiliser plus de bits pour l'adresse logique que le nombre de bits requis pour adresser la mémoire principale
  - ◆ Espace d'adressage logique >> esp. d'adressage physique

# Mémoire Virtuelle

- **La mémoire logique est donc appelée *mémoire virtuelle***
  - ◆ Est maintenue en mémoire secondaire
  - ◆ Les pièces sont amenées en mémoire principale seulement quand nécessaire, sur demande
- **Pour une meilleure performance, la mémoire virtuelle se trouve souvent dans une région du disque qui n'est pas gérée par le système de fichiers**
  - ◆ Mémoire va-et-vient, swap memory
- **La mémoire physique est celle qui est référencée par une adresse physique**
  - ◆ Se trouve dans le RAM et cache

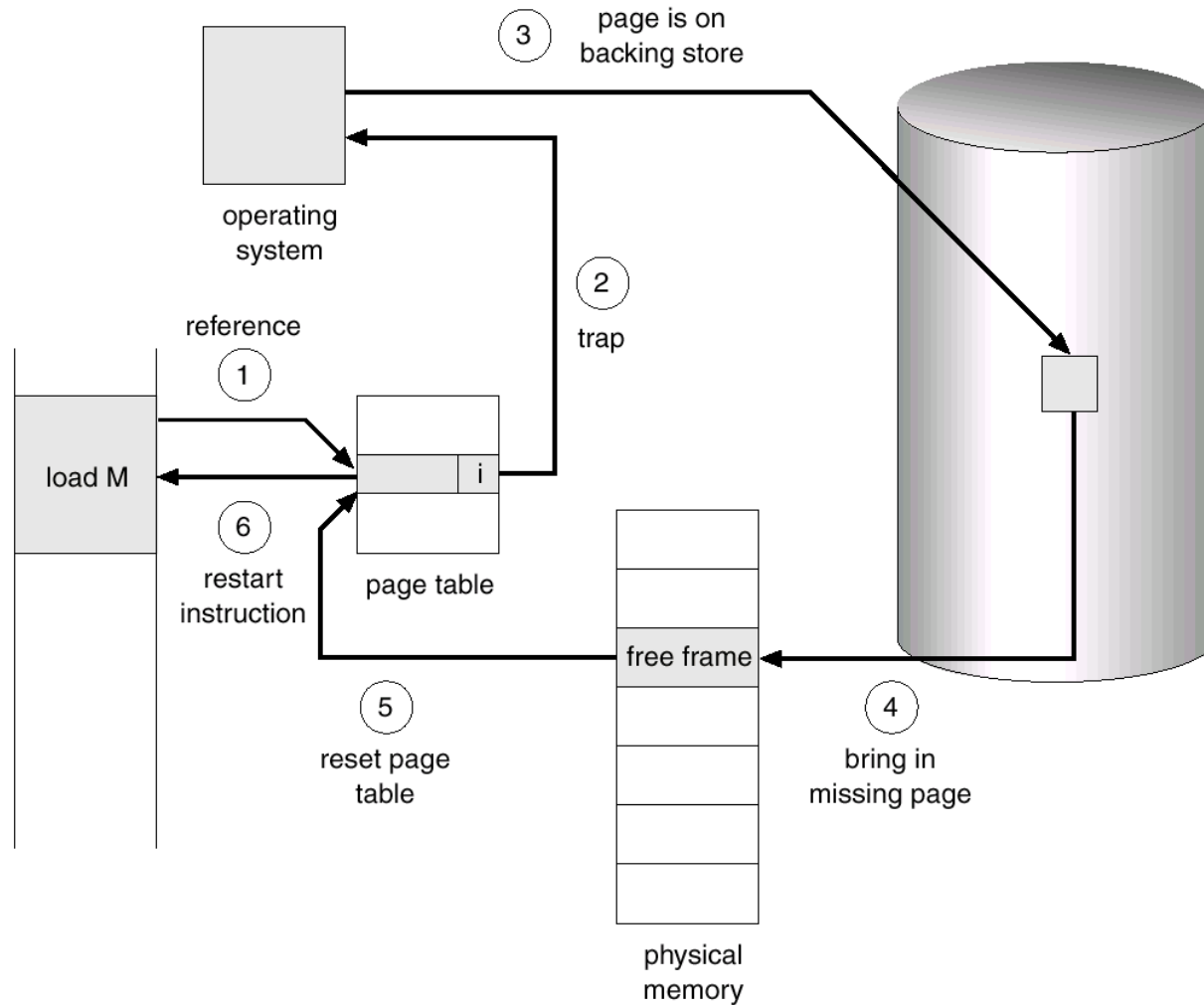
# Mémoire virtuelle: le mécanisme de va-et-vient



# Exécution d'un Processus

- Le SE charge la mémoire principale de quelques pièces (seulement) du programme (incluant le point de départ)
- Chaque entrée de la table de pages (ou segments) possède un **bit présent** qui indique si la page ou segment se trouve en mémoire principale
- **L'ensemble résident (résident set)** est la portion du processus se trouvant en mémoire principale
- Une interruption est générée lorsque l'adresse logique réfère à une pièce qui n'est pas dans l'ensemble résident
  - ◆ défaut de pagination (page fault)

# Exécution d'un défaut de page: va-et-vient plus en détail

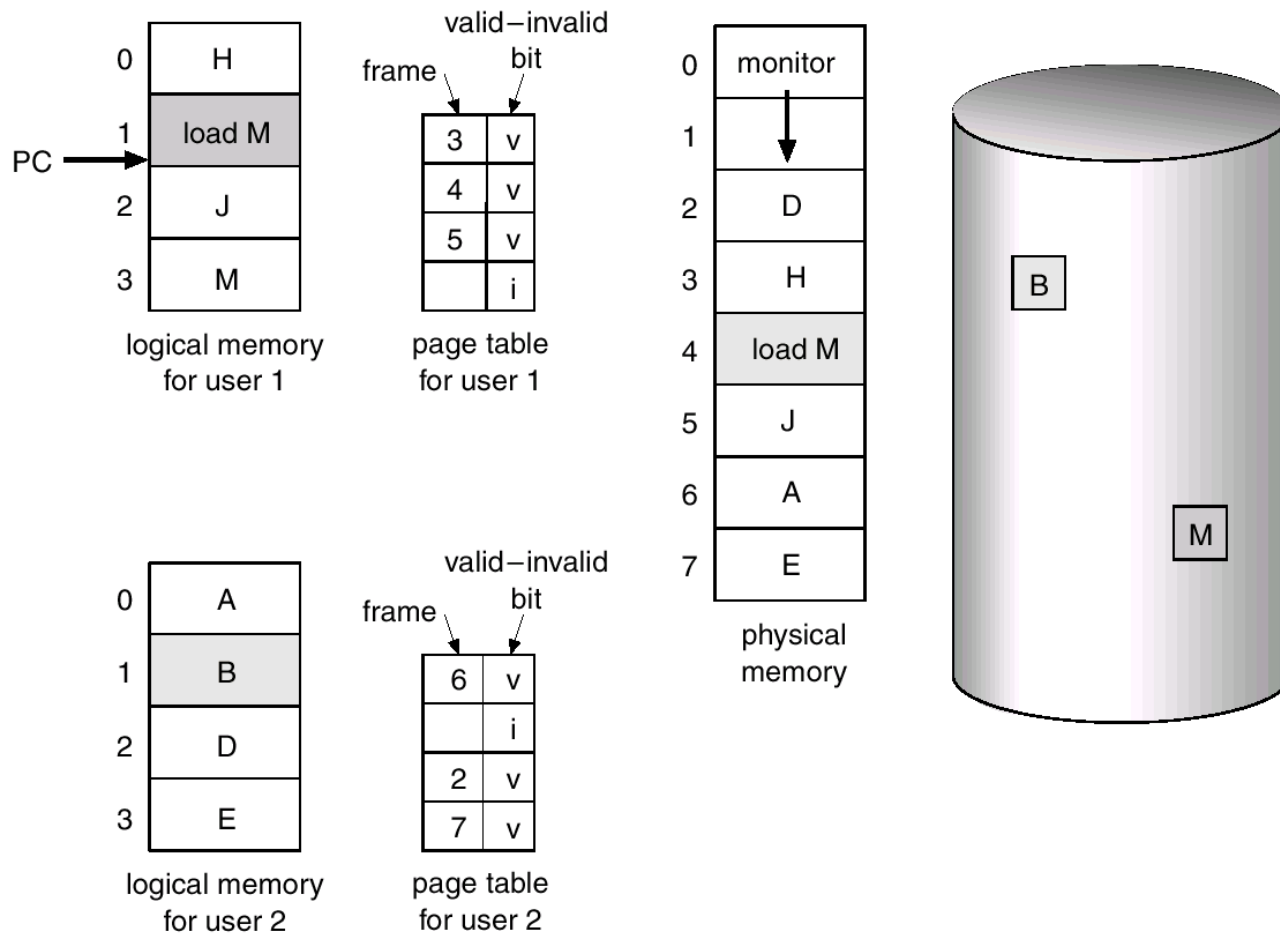


# Séquence d'événements pour défaut de page

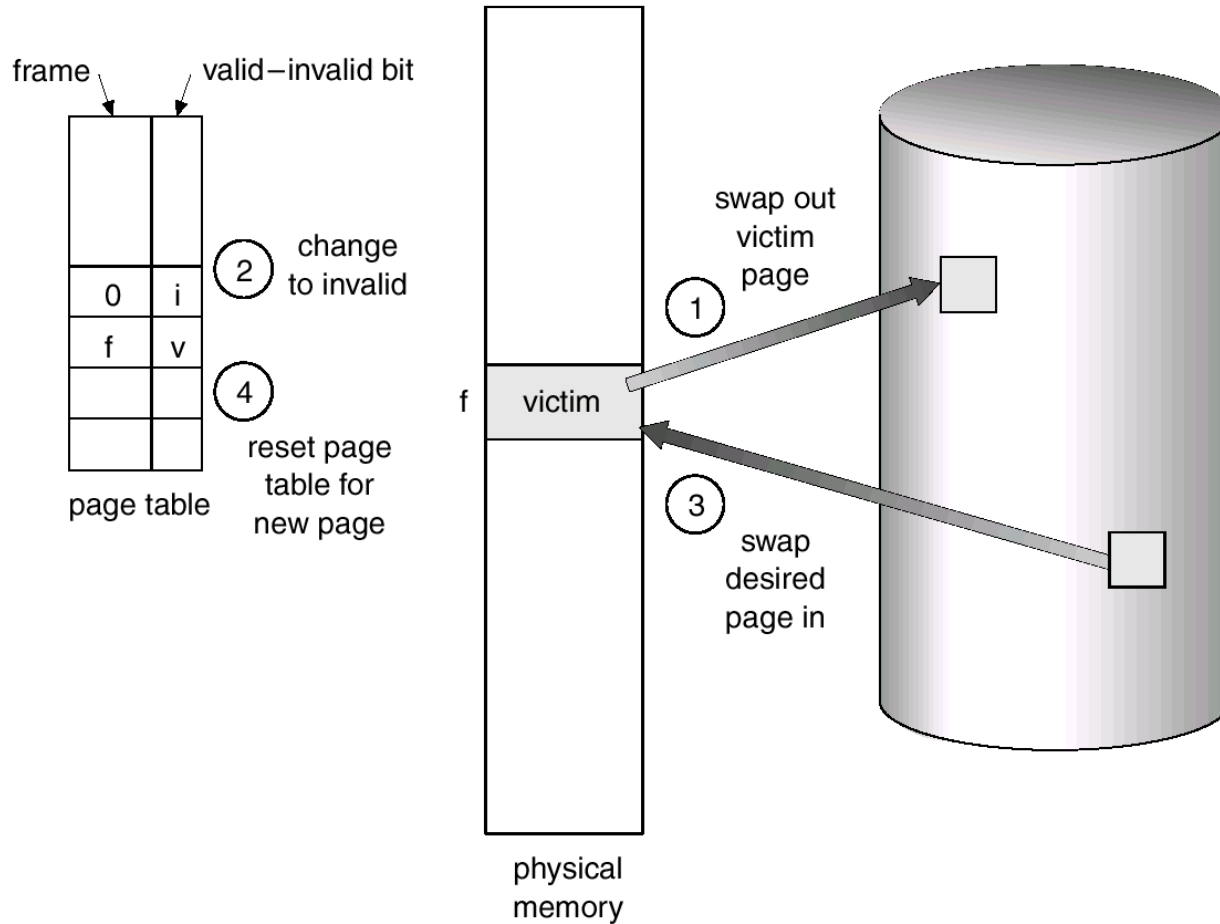
- **Trappe au SE: page demandée pas en RAM**
- **Sauvegarder le PCB**
- **Un autre processus peut maintenant avoir l'UCT**
- **SE trouve la page sur disque**
- **lit la page du disque dans un cadre de mémoire libre (supposons qu'il y en un!)**
  - ◆ exécuter les opérations disque nécessaires pour lire la page
- **L'unité disque a complété le transfert et interrompt l'UCT**
  - ◆ sauvegarder le PCB du processus s'exécutant
- **SE met à jour le contenu du tableau des pages du processus qui a causé le défaut de page**
- **Ce processus devient prêt=ready**
- **la page désirée étant en mémoire, il pourra maintenant continuer**



# Quand la RAM est pleine mais nous avons besoin d'une page pas en RAM



# La page victime...



# Remplacement de pages

- Quoi faire si un processus demande une nouvelle page et il n'y a pas de cadres libres en RAM?
- Il faudra choisir une page déjà en mémoire principale, appartenant au même ou à un autre processus, qu'il est possible d'enlever de la RAM
  - ◆ la **victime**!
- Un cadre de mémoire sera donc rendu disponible
- Évidemment, plusieurs cadres de mémoire ne peuvent pas être `victimisés`:
  - ◆ p.ex. cadres contenant le noyau du SE, tampons d'E/S...

## Bit de modification , *dirty bit*

- La 'victime' doit-elle être réécrite en mémoire secondaire?
- Seulement si elle a été changée depuis qu'elle a été amenée en mémoire principale
  - ◆ sinon, sa copie sur disque est encore fidèle
- Bit de modif sur chaque descripteur de page indique si la page a été changée
- Donc pour calculer le coût en temps d'une référence à la mémoire il faut aussi considérer la probabilité qu'une page soit modifiée et le temps de réécriture dans ce cas

# Algorithmes de remplacement pages

- Choisir la victime de façon à minimiser le taux de défaut de pages
  - ◆ pas évident!!!
- Page dont nous n`aurons pas besoin dans le futur? impossible à savoir!
- Page pas souvent utilisée?
- Page qui a déjà séjournée longtemps en mémoire??
- etc.

# Critères d'évaluation des algorithmes

- Les algorithmes de choix de pages à remplacer doivent être conçus de façon à **minimiser le taux de défaut de pages à long terme**
- Mais il ne peuvent pas impliquer des temps de système excessifs, p.ex. mise à jour de tableaux en mémoire pour chaque accès de mémoire

# Explication et évaluation des algorithmes

- Nous allons expliquer et évaluer les algorithmes en utilisant la **chaîne de référence** pages suivante :

2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

- Attention: les séquences d'utilisation pages ne sont pas aléatoires...
- L'évaluation sera faite sur la base de cet exemple, évidemment pas suffisant pour en tirer des conclusions générales

# Algorithmes pour la politique de remplacement

- **L'algorithme optimal (OPT) choisit pour page à remplacer celle qui sera référencée le plus tardivement**
  - ◆ produit le + petit nombre de défauts de page
  - ◆ impossible à réaliser (car il faut connaître le futur) mais sert de norme de comparaison pour les autres algorithmes:
    - ➡ Ordre chronologique d'utilisation : la moins récemment utilisé Least recently used (LRU)
    - ➡ Ordre chronologique de chargement (FIFO)
    - ➡ Deuxième chance ou Horloge (Clock)



# Algorithmes pour la politique de remplacement

- **Ordre chronologique d'utilisation (LRU)**
- **Remplace la page dont la dernière référence remonte au temps le plus lointain (le passé utilisé pour prédire le futur)**
  - ◆ Il s'agit de la page qui a le moins de chance d'être référencée
  - ◆ performance presque aussi bonne que l'algorithme OPT

# Comparaison OPT-LRU

- **Exemple: Un processus de 5 pages s'il n'y a que 3 pages physiques disponibles.**
- **Dans cet exemple, OPT occasionne 3+3 défauts, LRU 3+4.**

Page address  
stream

2 3 2 1 5 2 4 5 3 2 5 2

OPT

2	2	2	2	2	2	4	4	4	2	2	2
	3	3	3	3	3	3	3	3	3	3	3
			1	5	5	5	5	5	5	5	5
				F		F			F		

LRU

2	2	2	2	2	2	2	2	3	3	3	3
	3	3	3	5	5	5	5	5	5	5	5
			1	1	1	4	4	4	2	2	2
				F		F		F	F		

## Note sur le comptage des défauts de page

- Lorsque la mémoire principale est vide, chaque nouvelle page que nous ajoutons est le résultat d'un défaut de page
- Mais pour mieux comparer les algorithmes, il est utile de garder séparés ces défauts initiaux
  - ◆ car leur nombre est le même pour tous les algorithmes

# Premier arrivé, premier sorti (FIFO)

- **Logique:** une page qui a été longtemps en mémoire a eu sa chance pour s'exécuter
- **Les cadres forment conceptuellement un tampon circulaire, débutant à la plus vieille page**
  - ◆ Lorsque la mémoire est pleine, **la plus vieille** page est remplacée. Donc: "first-in, first-out"
- **Simple à mettre en application**
  - ◆ tampon consulté et mis à jour seulement aux défauts de pages...
- **Mais: Une page fréquemment utilisée est souvent la plus vieille, elle sera remplacée par FIFO!**

# Comparaison de FIFO avec LRU

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2																																				
LRU	<table><tr><td>2</td></tr><tr><td></td></tr><tr><td></td></tr></table>	2			<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td></td></tr></table>	2	3		<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td></td></tr></table>	2	3		<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td>1</td></tr></table>	2	3	1	<table><tr><td>2</td></tr><tr><td>5</td></tr><tr><td>1</td></tr></table>	2	5	1	<table><tr><td>2</td></tr><tr><td>5</td></tr><tr><td>1</td></tr></table>	2	5	1	<table><tr><td>2</td></tr><tr><td>5</td></tr><tr><td>4</td></tr></table>	2	5	4	<table><tr><td>2</td></tr><tr><td>5</td></tr><tr><td>4</td></tr></table>	2	5	4	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>4</td></tr></table>	3	5	4	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>2</td></tr></table>	3	5	2	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>2</td></tr></table>	3	5	2	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>2</td></tr></table>	3	5	2
	2																																															
2																																																
3																																																
2																																																
3																																																
2																																																
3																																																
1																																																
2																																																
5																																																
1																																																
2																																																
5																																																
1																																																
2																																																
5																																																
4																																																
2																																																
5																																																
4																																																
3																																																
5																																																
4																																																
3																																																
5																																																
2																																																
3																																																
5																																																
2																																																
3																																																
5																																																
2																																																
				F		F			F	F																																						
FIFO	<table><tr><td>2</td></tr><tr><td></td></tr><tr><td></td></tr></table>	2			<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td></td></tr></table>	2	3		<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td></td></tr></table>	2	3		<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td>1</td></tr></table>	2	3	1	<table><tr><td>5</td></tr><tr><td>3</td></tr><tr><td>1</td></tr></table>	5	3	1	<table><tr><td>5</td></tr><tr><td>2</td></tr><tr><td>1</td></tr></table>	5	2	1	<table><tr><td>5</td></tr><tr><td>2</td></tr><tr><td>4</td></tr></table>	5	2	4	<table><tr><td>5</td></tr><tr><td>2</td></tr><tr><td>4</td></tr></table>	5	2	4	<table><tr><td>3</td></tr><tr><td>2</td></tr><tr><td>4</td></tr></table>	3	2	4	<table><tr><td>3</td></tr><tr><td>2</td></tr><tr><td>4</td></tr></table>	3	2	4	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>4</td></tr></table>	3	5	4	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>2</td></tr></table>	3	5	2
	2																																															
2																																																
3																																																
2																																																
3																																																
2																																																
3																																																
1																																																
5																																																
3																																																
1																																																
5																																																
2																																																
1																																																
5																																																
2																																																
4																																																
5																																																
2																																																
4																																																
3																																																
2																																																
4																																																
3																																																
2																																																
4																																																
3																																																
5																																																
4																																																
3																																																
5																																																
2																																																
				F	F	F			F		F	F																																				

- Contrairement à FIFO, LRU reconnaît que les pages 2 et 5 sont utilisées fréquemment
- La performance de FIFO est moins bonne:
  - ◆ dans ce cas, LRU = 3+4, FIFO = 3+6

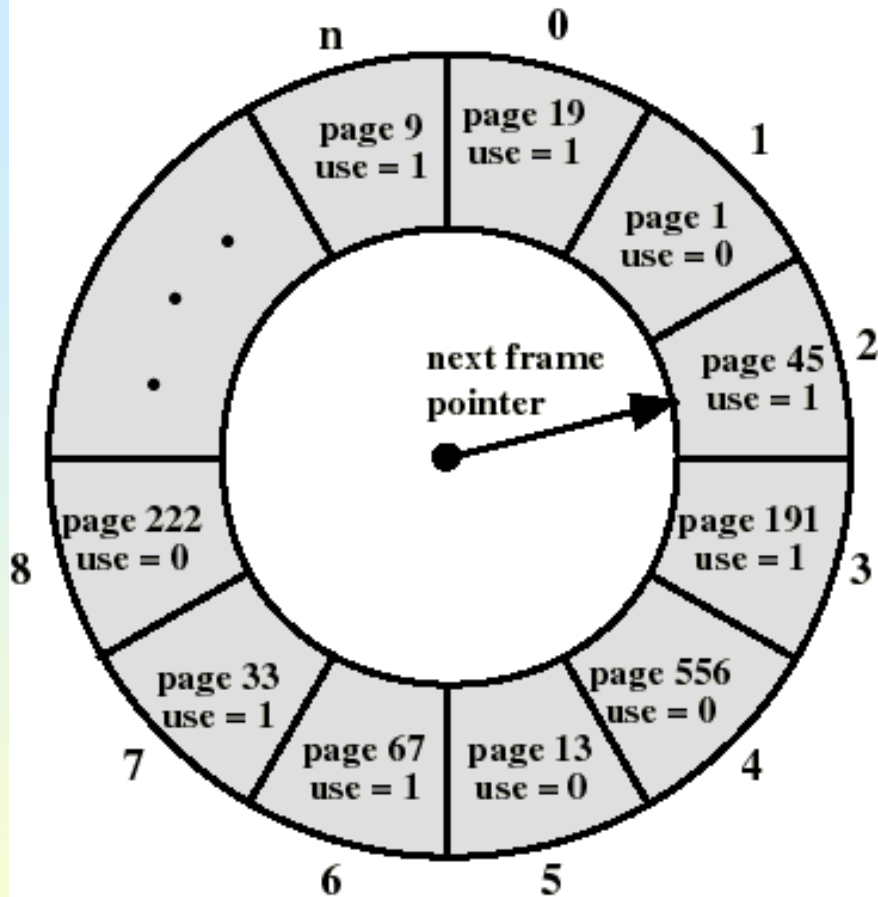
# Problème conceptuel avec FIFO

- **Les premières pages amenées en mémoire sont souvent utiles pendant toute l'exécution d'un processus!**
  - ◆ variables globales, programme principal, etc.
- **Ce qui montre un problème avec notre façon de comparer les méthodes sur la base d'une séquence aléatoire:**
  - ◆ les références aux pages dans un programme réel ne seront pas vraiment aléatoires

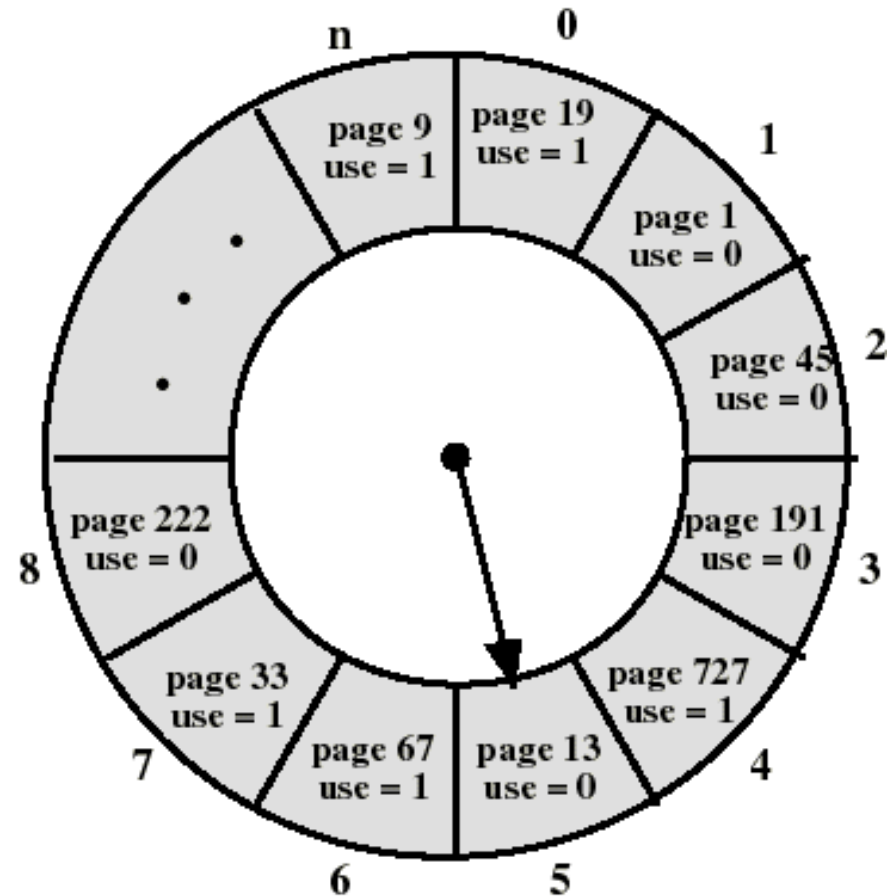
# L'algorithme de l'horloge (deuxième chance)

- **Semblable à FIFO, mais les cadres qui viennent d'être utilisés (bit=1) ne sont pas remplacés (deuxième chance)**
  - ◆ Les cadres forment conceptuellement un tampon circulaire
  - ◆ Lorsqu'une page est chargée dans un cadre, un pointeur pointe sur le prochain cadre du tampon
- **Pour chaque cadre du tampon, un bit "utilisé" est mis à 1 (par le matériel) lorsque:**
  - ◆ une page y est nouvellement chargée
  - ◆ sa page est utilisée
- **Le prochain cadre du tampon à être remplacé sera le premier rencontré qui aura son bit "utilisé" = 0.**
  - ◆ Durant cette recherche, tout bit "utilisé" = 1 rencontré sera mis à 0

# Algorithme de l'horloge: un exemple



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

La page 727 est chargée dans le cadre 4.  
La prochaine victime est 5, puis 8.



# Comparaison: Horloge, FIFO et LRU

Page address  
stream

2	3	2	1	5	2	4	5	3	2	5	2
2	2	2	2	2	2	2	2	3	3	3	3
	3	3	3	5	5	5	5	5	5	5	5
			1	1	1	4	4	4	2	2	2
				F		F		F	F		

<b>FIFO</b>	<table><tr><td>2</td></tr><tr><td></td></tr><tr><td></td></tr></table>	2			<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td></td></tr></table>	2	3		<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td></td></tr></table>	2	3		<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td>1</td></tr></table>	2	3	1	<table><tr><td>5</td></tr><tr><td>3</td></tr><tr><td>1</td></tr></table> <b>F</b>	5	3	1	<table><tr><td>5</td></tr><tr><td>2</td></tr><tr><td>1</td></tr></table> <b>F</b>	5	2	1	<table><tr><td>5</td></tr><tr><td>2</td></tr><tr><td>4</td></tr></table> <b>F</b>	5	2	4	<table><tr><td>5</td></tr><tr><td>2</td></tr><tr><td>4</td></tr></table>	5	2	4	<table><tr><td>3</td></tr><tr><td>2</td></tr><tr><td>4</td></tr></table> <b>F</b>	3	2	4	<table><tr><td>3</td></tr><tr><td>2</td></tr><tr><td>4</td></tr></table>	3	2	4	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>4</td></tr></table> <b>F</b>	3	5	4	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>2</td></tr></table> <b>F</b>	3	5	2
	2																																															
2																																																
3																																																
2																																																
3																																																
2																																																
3																																																
1																																																
5																																																
3																																																
1																																																
5																																																
2																																																
1																																																
5																																																
2																																																
4																																																
5																																																
2																																																
4																																																
3																																																
2																																																
4																																																
3																																																
2																																																
4																																																
3																																																
5																																																
4																																																
3																																																
5																																																
2																																																

CLOCK

2*

2*
3*

2*
3*

2*
3*
1*

5*
3
1

5*
2*
1

5*
2*
4*

5*
2*
4*

3*
2
4

3*
2*
4

3*
2
5*

3*
2*
5*

F

F

F

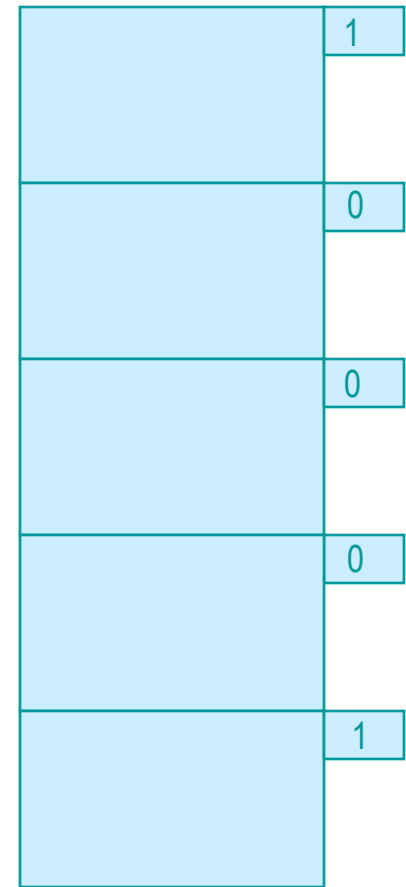
F

F

- Astérisque indique que le bit utilisé est 1
- L'horloge protège du remplacement les pages fréquemment utilisées en mettant à 1 le bit "utilisé" à chaque référence
- LRU = 3+4, FIFO = 3+6, Horloge = 3+5

# Matériel additionnel pour l'algorithme CLOCK

- Chaque bloc de mémoire a un bit 'touché' (use)
- Quand le contenu du bloc est utilisé, le bit est mis à 1 par le matériel
- Le SE regarde le bit
  - ◆ S'il est 0, la page peut être remplacée
  - ◆ S'il est 1, il le met à 0



Mémoire

# Comparaison: Horloge, FIFO et LRU

- **Les simulations montrent que l'horloge est presque aussi performant que LRU**
  - ◆ variantes de l'horloge ont été implantées dans des systèmes réels
- **Lorsque les pages candidates au remplacement sont locales au processus souffrant du défaut de page et que le nombre de cadres alloué est fixe, les expériences montrent que:**
  - ◆ Si peu (6 à 8) de cadres sont alloués, le nombre de défaut de pages produit par FIFO est presque double de celui produit par LRU, et celui de CLOCK est entre les deux
  - ◆ Ce facteur s'approche de 1 lorsque plusieurs (plus de 12) cadres sont alloués.
- **Cependant le cas réel est de milliers et millions de pages et cadres, donc la différence n'est pas trop importante en pratique...**
  - ◆ On peut tranquillement utiliser LRU

# Algorithmes *compteurs*

- Garder un compteur pour les références à chaque page
- LFU: Least Frequently Used: remplacer la pages avec le plus petit compteur
- MFU: Most Frequently Used: remplacer les pages bien utilisées pour donner une chance aux nouvelles
- Ces algorithmes sont d'implantation couteuse et ne sont pas très utilisés

# Anomalie de Belady

- **Pour quelques algorithmes, dans quelques cas il pourrait avoir plus de défauts avec plus de mémoire!**
  - ◆ p. ex. FIFO, mais pas LRU, OPT, CLOCK