Module 7 Gestion de la mémoire

Silberschatz: Chapitre 8

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

Gestion de mémoire: objectifs

- **Optimisation de l ’utilisation de la mémoire principale = RAM**
- **Les 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’UCT
  - s’adapter aux besoins de mémoire de l ’usager
    - allocation dynamique au besoin

Gestion de la mémoire: concepts dans ce chapitre

- **Adresse physique et adresse logique**
  - mémoire physique et mémoire logique
- **Allocation contiguë**
  - partitions
- **Segmentation**
- **Pagination**
- **Segmentation et pagination combinées**

Application de ces concepts

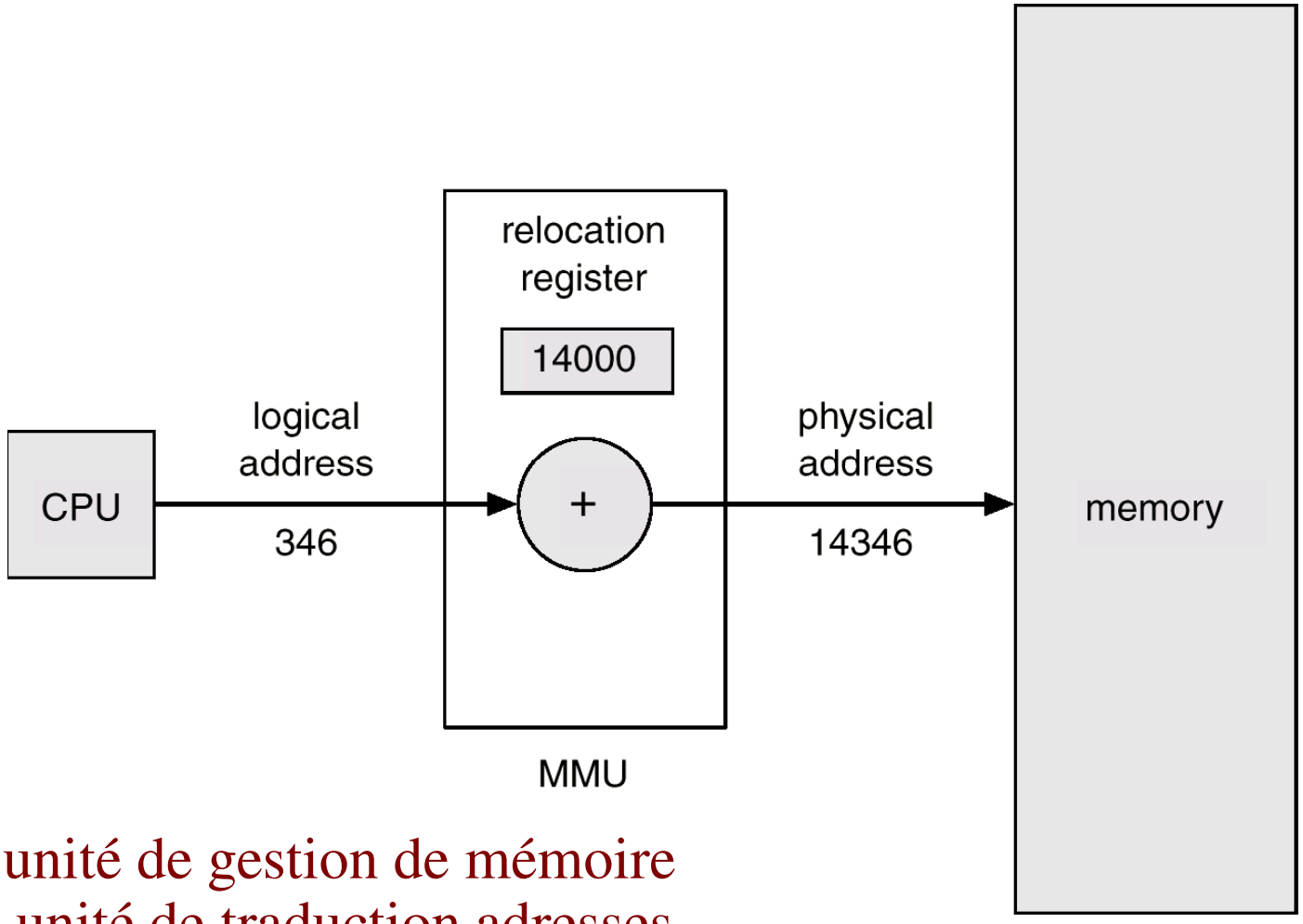
- Pas tous les concepts de ce chapitre sont effectivement utilisés tels quels aujourd'hui dans la gestion de mémoire centrale
- Cependant plusieurs se retrouvent dans le domaine de la gestion de mémoires auxiliaires, surtout disques

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 dans un programme**
- **Adresses logiques: les adresses dans cet espace**

- **Il faut séparer ces concepts car normalement, les programmes sont chargés plusieurs fois durant leur exécution à des positions différentes de mémoire**
  - Donc adresse physique ≠ 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 d'un emplacement dans un programme
  - par rapport au programme lui-même seulement
  - indépendante de la position du programme en mémoire physique

Vue de l'usager

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

- **Étant donné la grande variété de matériaux et logiciels, il est impossible de donner des définitions plus précises.**

Liaison (Binding) d'adresses logiques et physiques (instructions et données)

- **La liaison des adresses logiques aux adresses physiques peut être effectuée à des moments différents:**
  - Compilation: quand l'adresse physique est connue au moment de la compilation (rare)
    - ex. parties du SE
  - Chargement: quand l'adresse physique où le programme est chargé est connue, les adresses logiques peuvent être converties (rare aujourd'hui)
  - Exécution: normalement, les adresses physiques ne sont connues qu'au moment de l'exécution
    - ex. allocation dynamique

Deux concepts de base

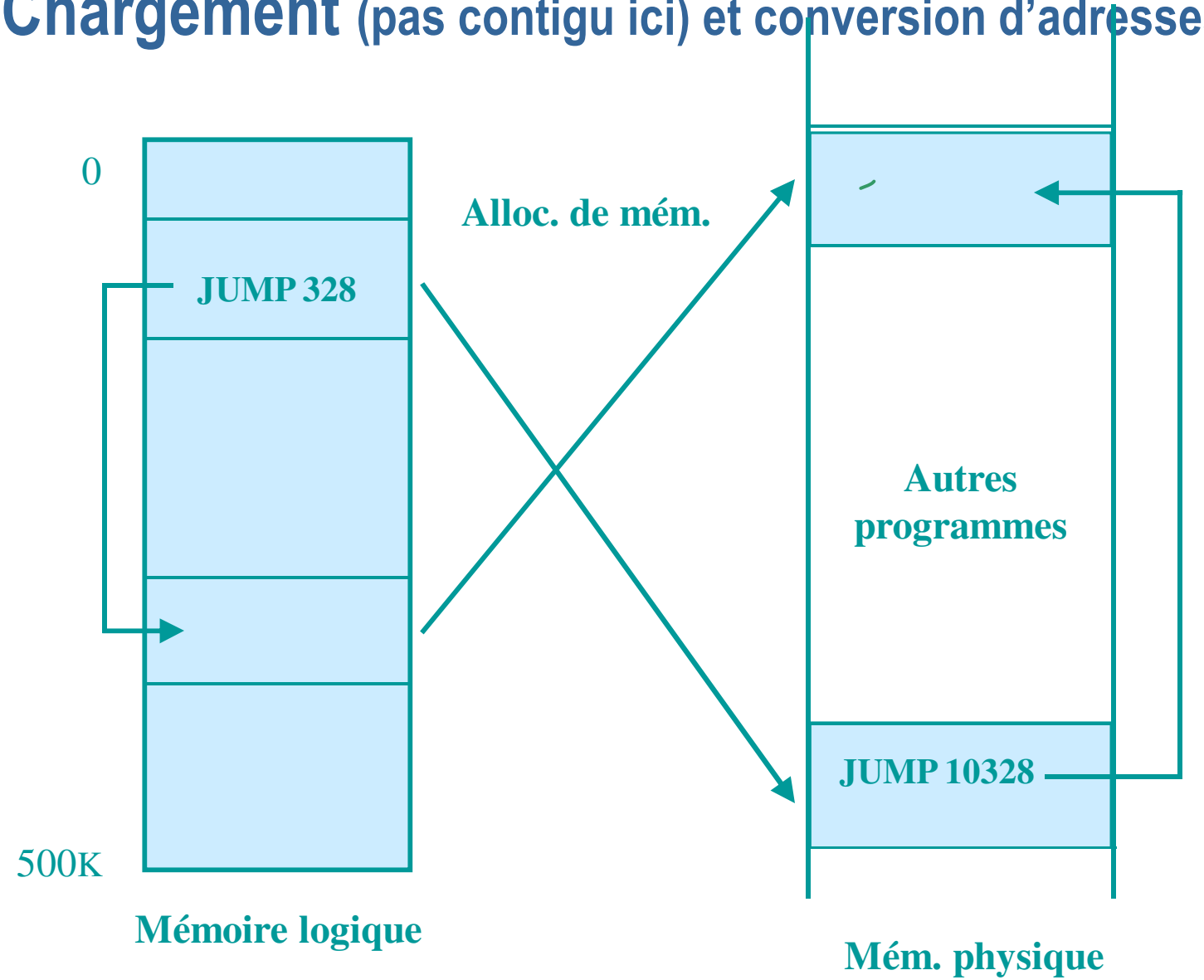
- **Chargement = Loading. Le programme, ou une de ses parties, est chargé en mémoire physique, prêt à exécuter.**
  - statique, dynamique
- **Édition de liens = Liaison (enchaînement) des différentes parties d'un programme pour en faire une entité exécutable.**
  - les références entre modules différents doivent être traduites
  - statique (avant l`exécution)
  - dynamique (sur demande pendant exécution)
    - N.B. parties du programme = modules = segments = sousprogrammes = objets, etc.

Aspects du chargement

- Trouver de la mémoire libre pour un module de chargement: contigu ou non

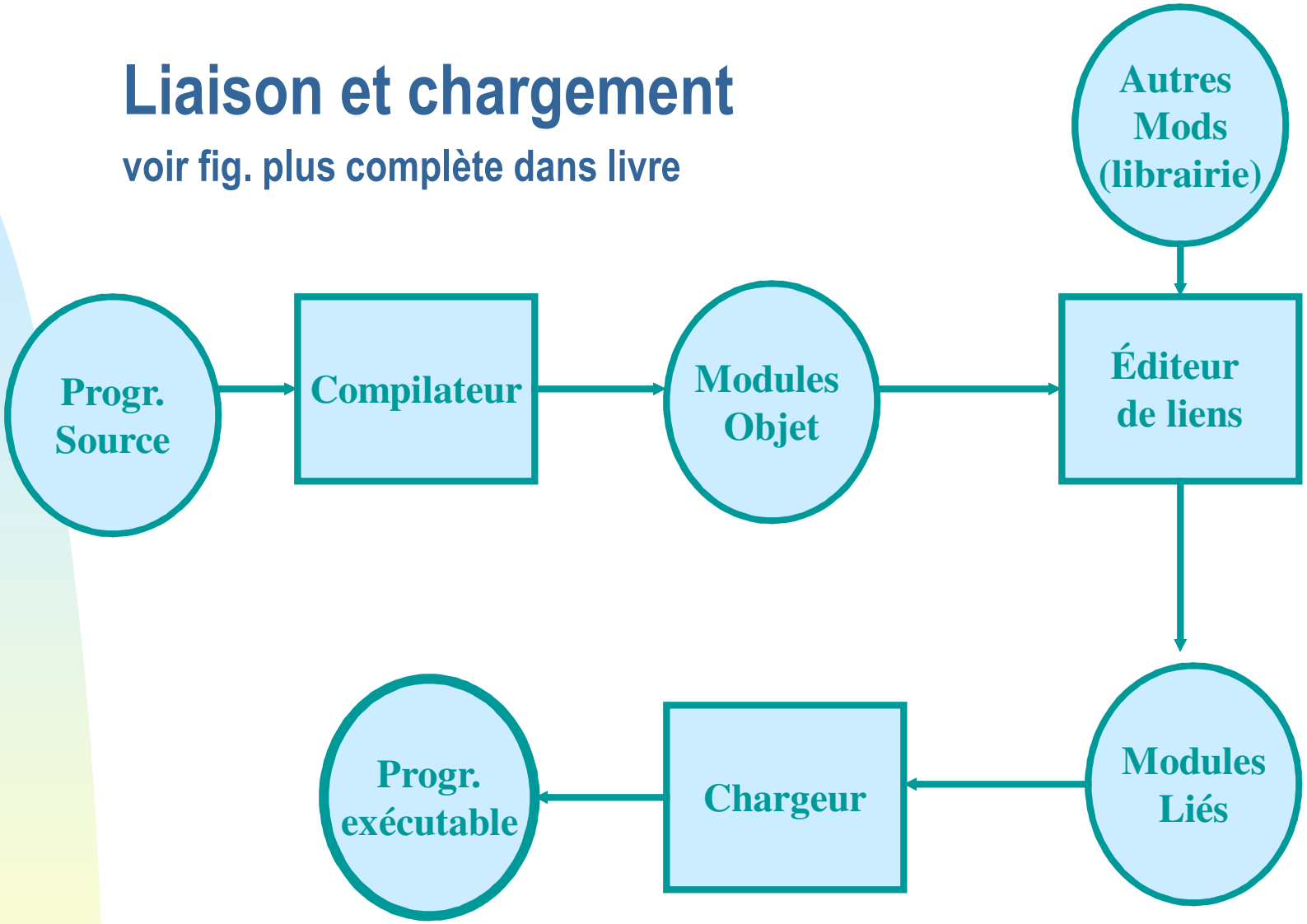
- Convertir les adresses du programme et effectuer les liaisons par rapport aux adresses où le module est chargé

Chargement (pas contigu ici) et conversion d'adresses



Liaison et chargement

voir fig. plus complète dans livre



NB: on fait l`hypothèse que tous les modules sont connus au début
Souvent, ce n'est pas le cas → chargement dynamique

Chargement et liaison dynamique

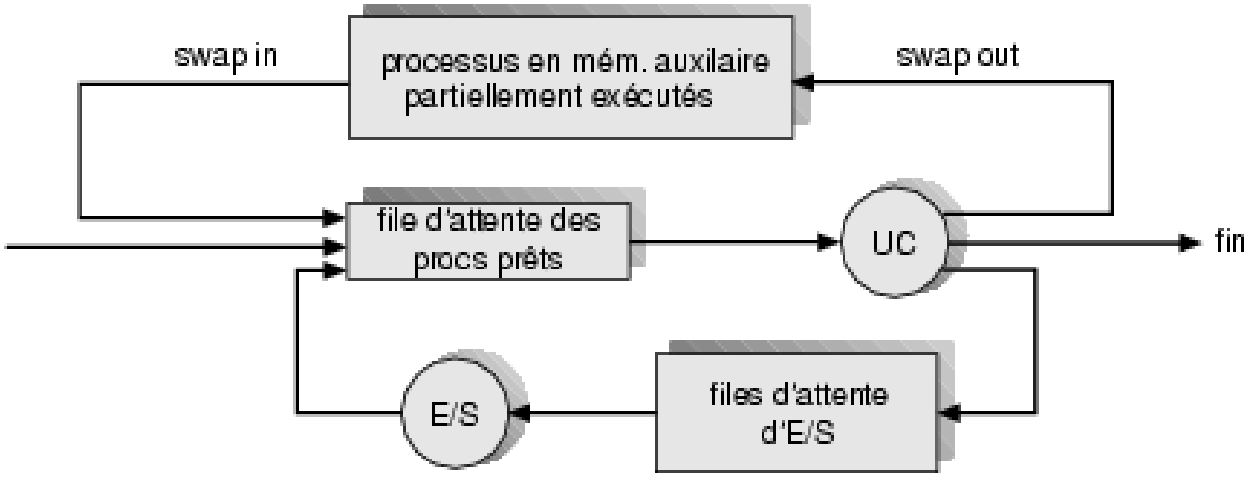
- **Un processus exécutant peut avoir besoin de différents modules du programme à différents moments**
- **Le chargement statique peut donc être inefficace**
- **Il vaut mieux charger les modules sur demande = dynamique**
  - dll, dynamically linked libraries
- **Les modules dynamiquement chargés sont au début représentés par des *stubs* qui indiquent comment arriver à ces derniers (ex. où il se trouve: disque, www, autre...)**
- **À sa 1ère exécution le stub charge le module en mémoire et sa liaison avec le reste du programme**
  - liaison dynamique
- **Les invocations successives du module ne doivent pas passer à travers ça, on saura l'adresse en mémoire**

Conversion d'adresses logique → physique

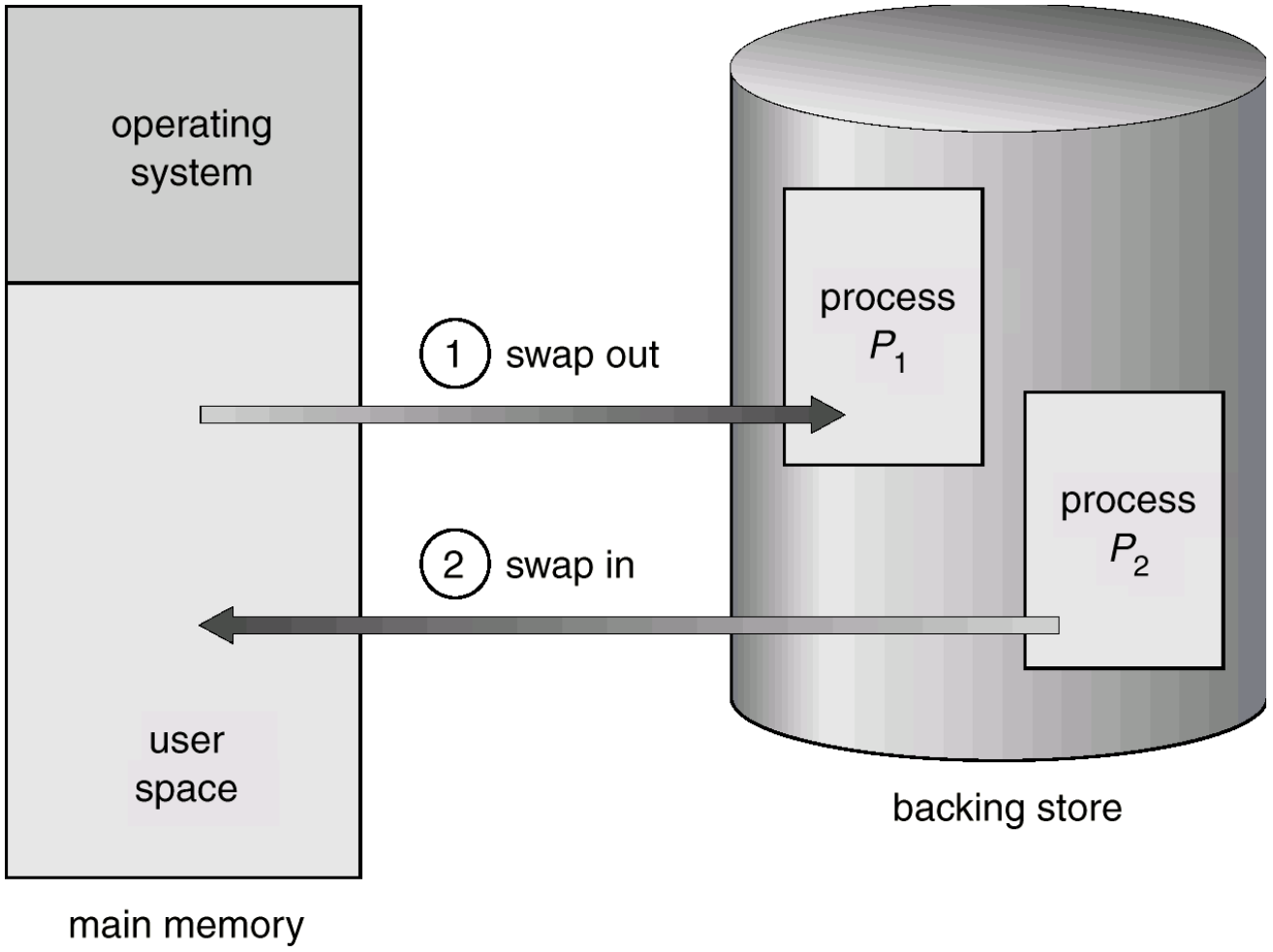
- **Dans les premiers systèmes, un programme était toujours lu aux mêmes adresses de mémoire**
- **Avec la multiprogrammation et l'allocation dynamique, il y a eu nécessité de lire un programme à des positions différentes**
- **Au début, ceci était fait par le chargeur (loader) qui changeait les adresses avant de démarrer l'exécution**
- **Aujourd'hui, ceci est fait par le MMU au fur et à mesure que le programme est exécuté**
- **Cela n'accroit pas le temps d'exécution, car le MMU agit en parallèle avec d'autres fonctions d'UCT**
  - ex. le MMU peut préparer l'adresse d 'une instruction en même temps que I 'UCT exécute l'instruction précédente

Permutation de programmes (swapping)

- **Un programme, ou une partie du programme, peut être temporairement enlevé de la mémoire pour permettre l'exécution d'autres programmes (chap. 4)**
  - il est déplacé dans la mémoire secondaire, normal ou disque



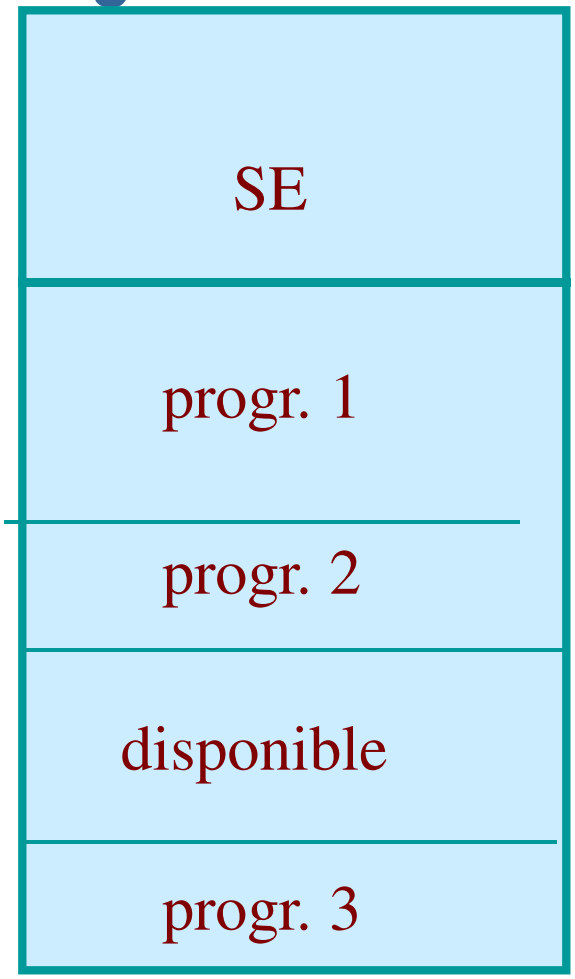
Permutation de programmes (swapping)



Affectation contiguë de mémoire

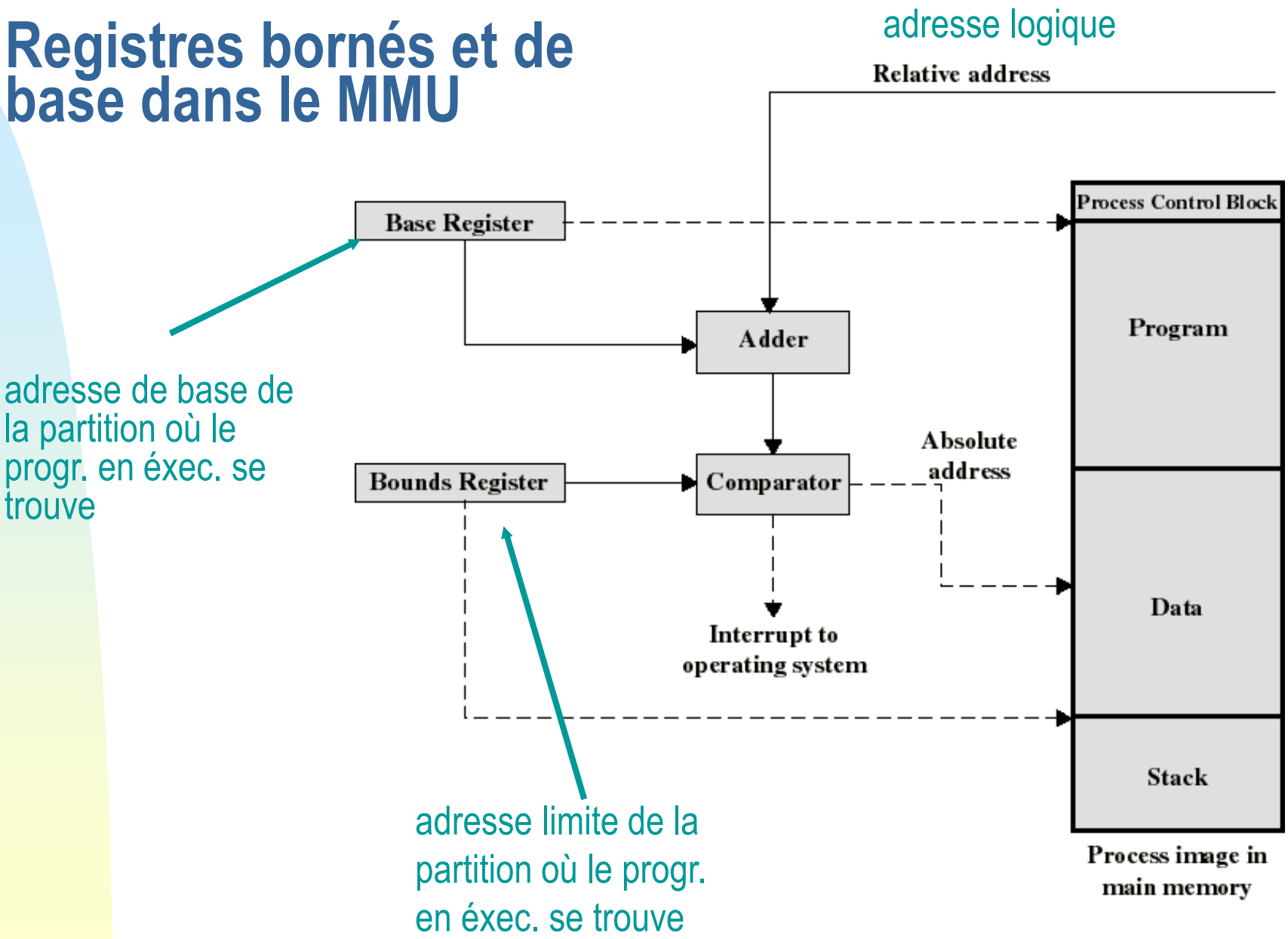
- **Nous avons plusieurs programmes à exécuter**
- **Nous pouvons les charger en mémoire les uns après les autres**
  - le lieu où un programme est lu n'est connu qu'au moment du chargement
- **Besoins de matériel: une registre de base et un registre borné suffisent à décrire l'espace de l'adresse du processus**

Affectation contiguë de mémoire



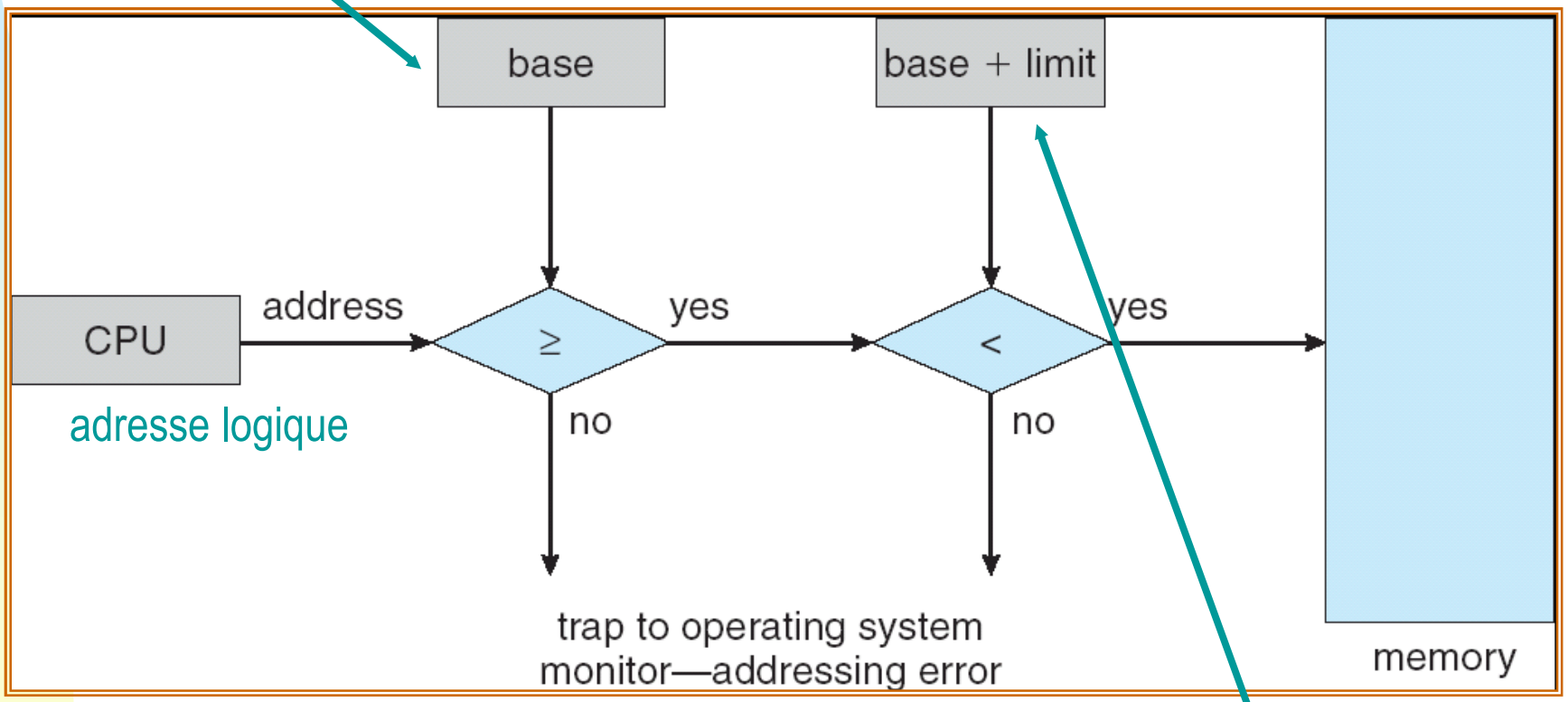
Nous avons ici 4 partitions pour des programmes - chacun est lu dans une seule zone de mémoire

Registres bornés et de base dans le MMU



Registres bornes et de base dans le MMU

adresse de base de la partition où le progr. en éxec. se trouve



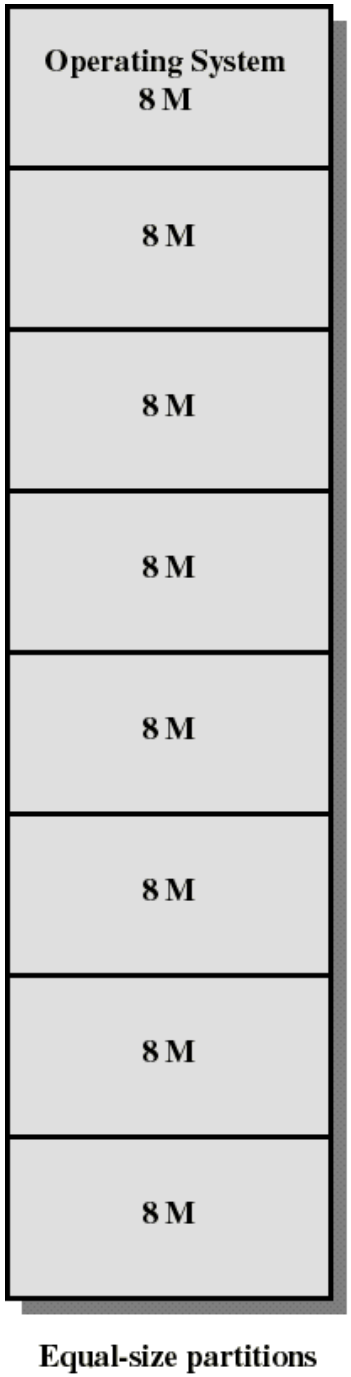
adresse limite de la partition où le progr. en éxec. se trouve

Fragmentation: mémoire non utilisée

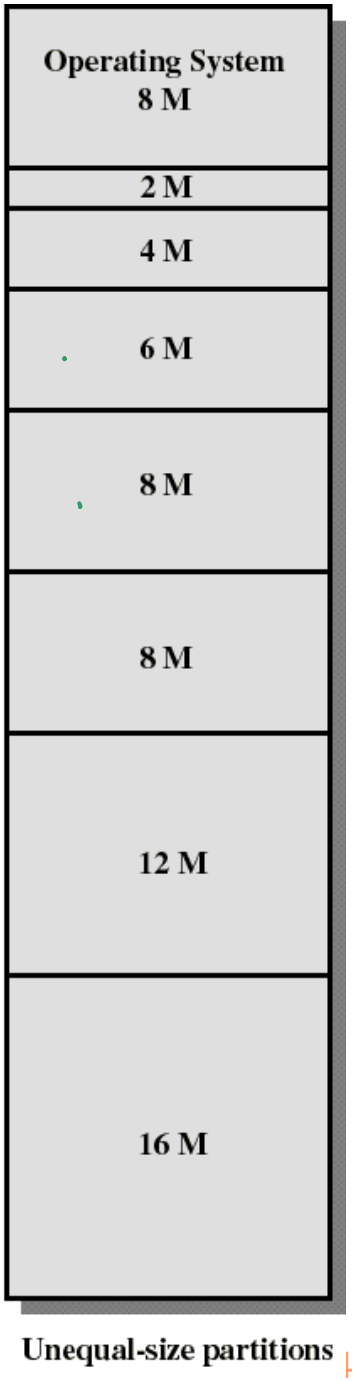
- **Un problème majeur dans l'affectation contiguë:**
  - Il y a assez d'espace pour exécuter un programme, mais il est fragmenté de façon non contiguë
    - externe: l'espace inutilisé est entre partitions
    - interne: l'espace inutilisé est dans les partitions

Partitions fixes

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



Equal-size partitions



Unequal-size partitions

(Stallings)

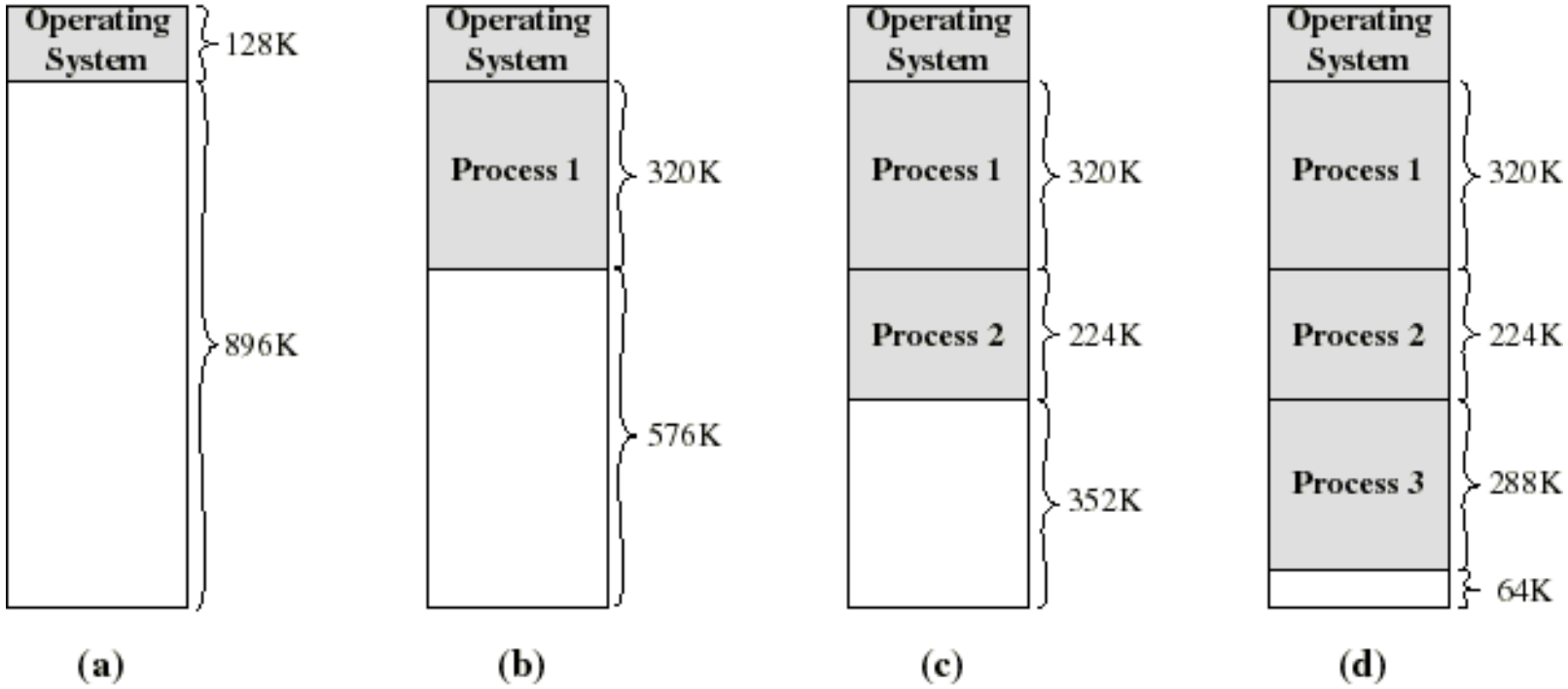
Partitions fixes

- **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 atténuent ces problèmes mais ils y demeurent...**

Partitions dynamiques

- Partitions en nombres 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

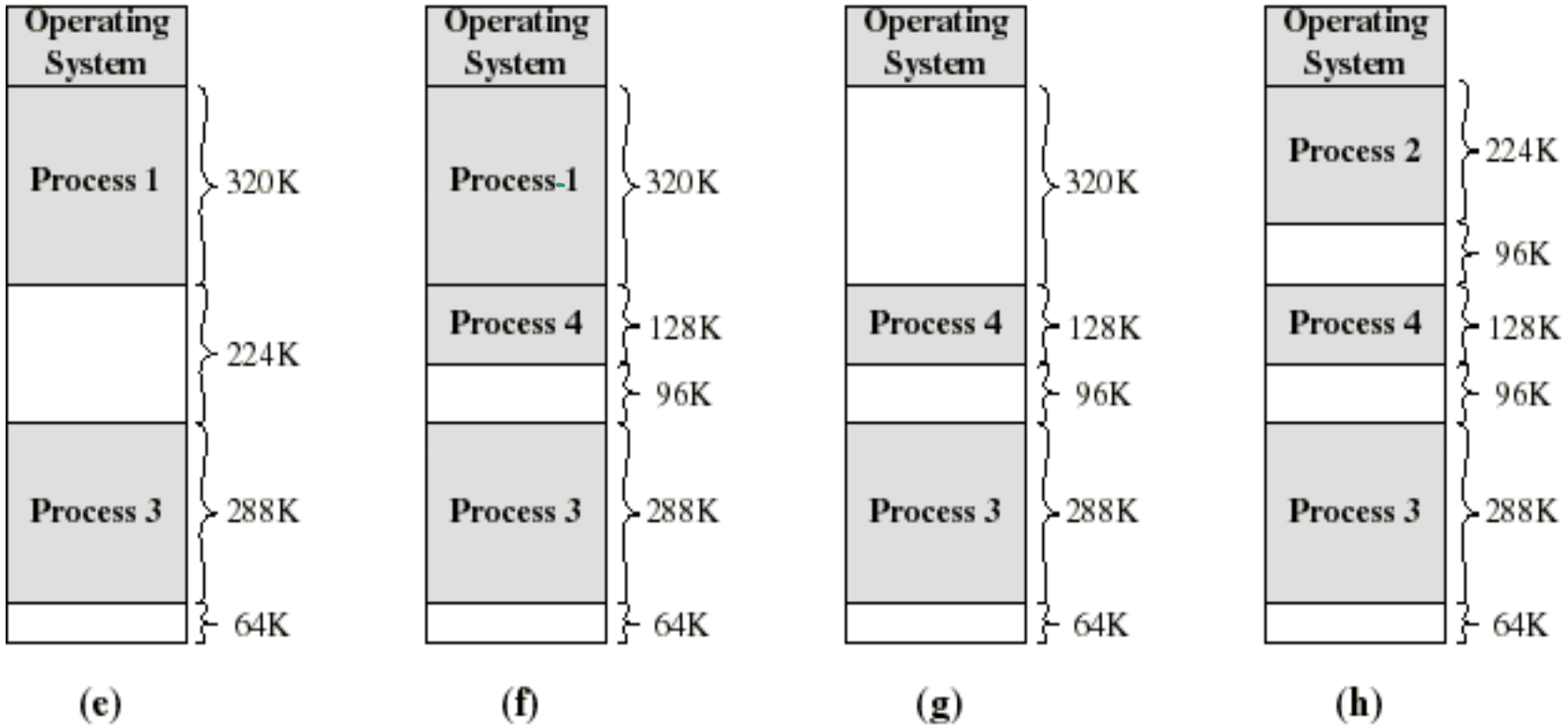
Partitions dynamiques: exemple (Stallings)



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

Swapped out

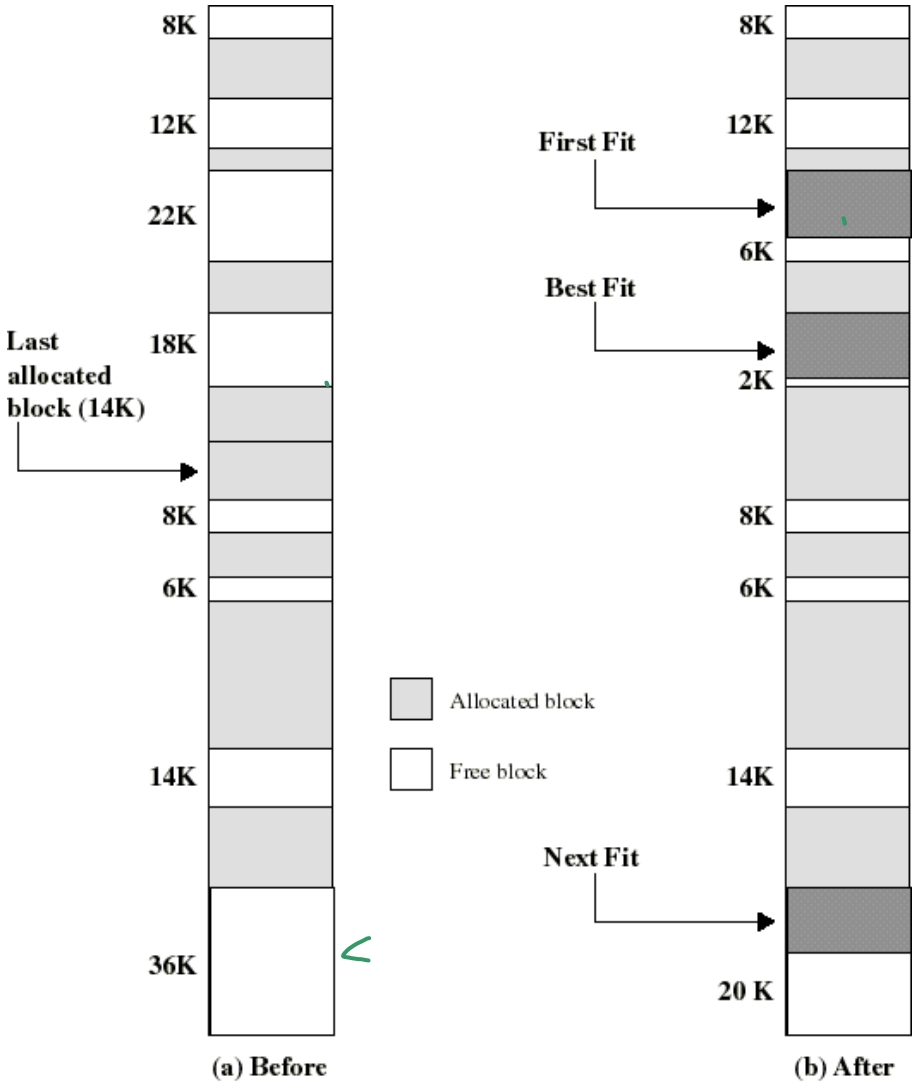
Partitions dynamiques: exemple (Stallings)



- (e-f) Progr. 2 est suspendu, Progr. 4 est chargé. Un trou de 224-128=96K est créé (*fragmentation externe*)
- (g-h) P1 se termine ou il est suspendu, P2 prend 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

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": choisi le plus petit trou
  - "First-fit": choisi le 1er trou à partir du début
  - "Next-fit": choisi le 1er trou à partir du dernier placement



Example Memory Configuration Before and After Allocation of 16 Kbyte Block

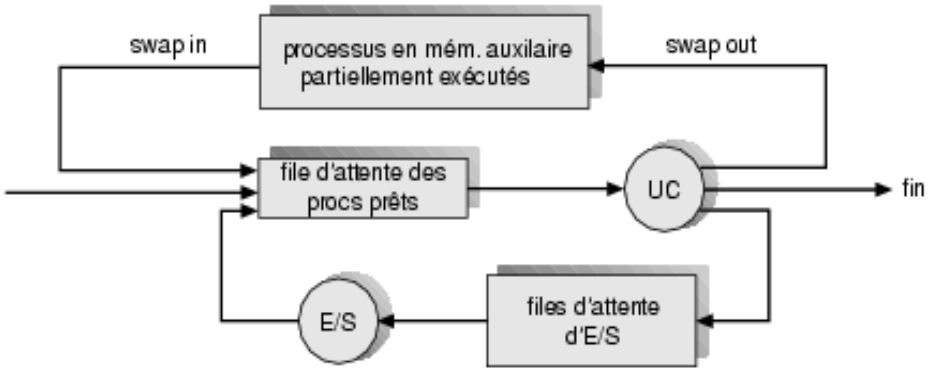
(Stallings)

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 (temps) le plus petit bloc possible: le trou créé est le plus petit possible**
  - la mémoire se remplit de trous trop petits pour contenir un programme
- **Next-fit": les allocations se feront souvent à la fin de la mémoire**

Suspension (v. chap 4)

- **Lorsque tous les programmes en mémoire sont bloqués, le SE peut en suspendre un (swap/suspend)**
  - On transfère au disque un des processus bloqués (en le mettant ainsi en état suspendu) et en le remplace par un processus prêt à être exécuté
    - ce dernier processus exécute une transition d'état New ou Suspended à état Ready



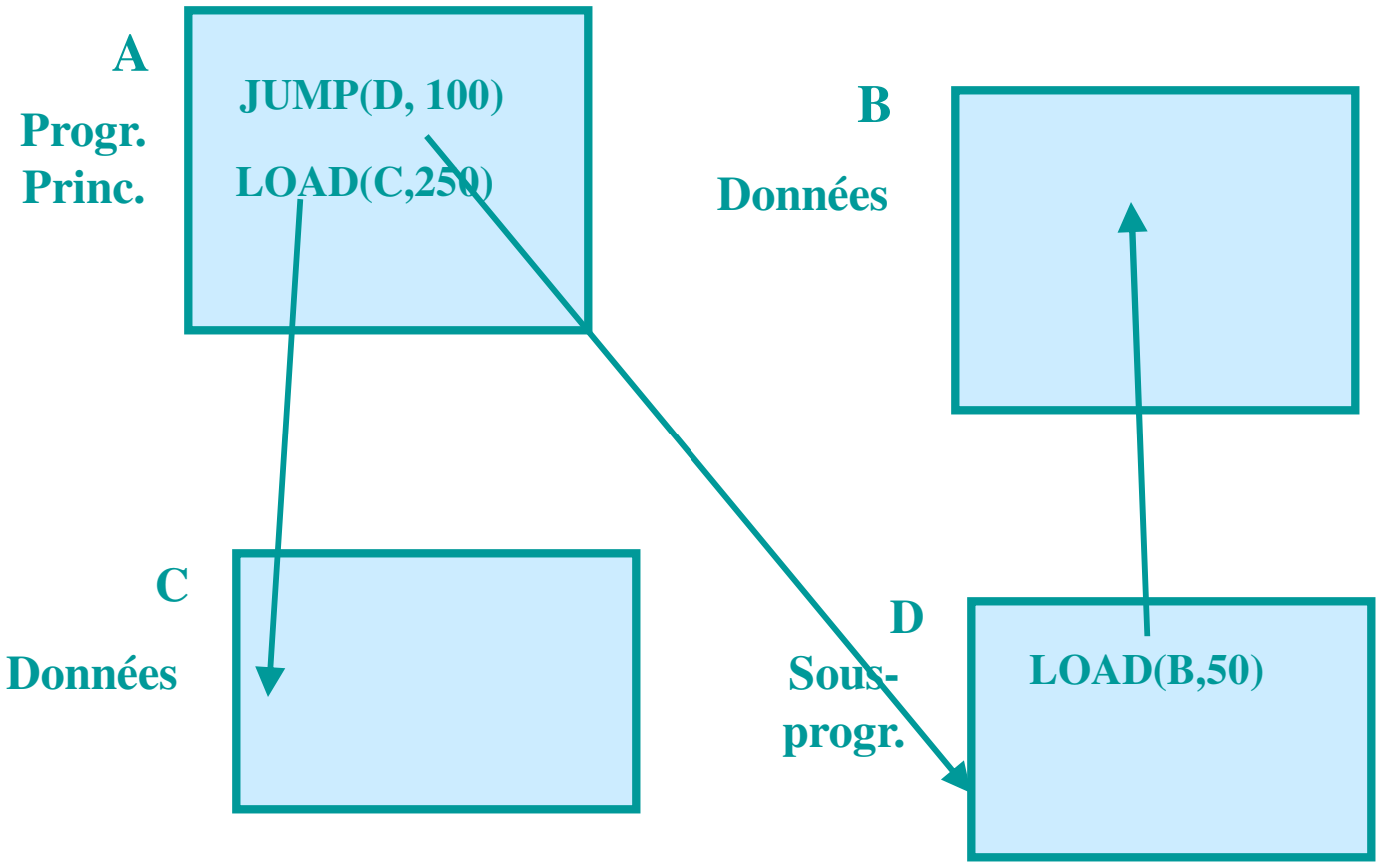
Compression (compaction)

- **Une solution pour la fragmentation externe**
- **Les programmes sont déplacés dans la 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 fragmentation externe existante**
- **Désavantages:**
  - temps de transfert programmes
  - besoin de rétablir tous les liens entre adresses de différents programmes

Allocation non contiguë

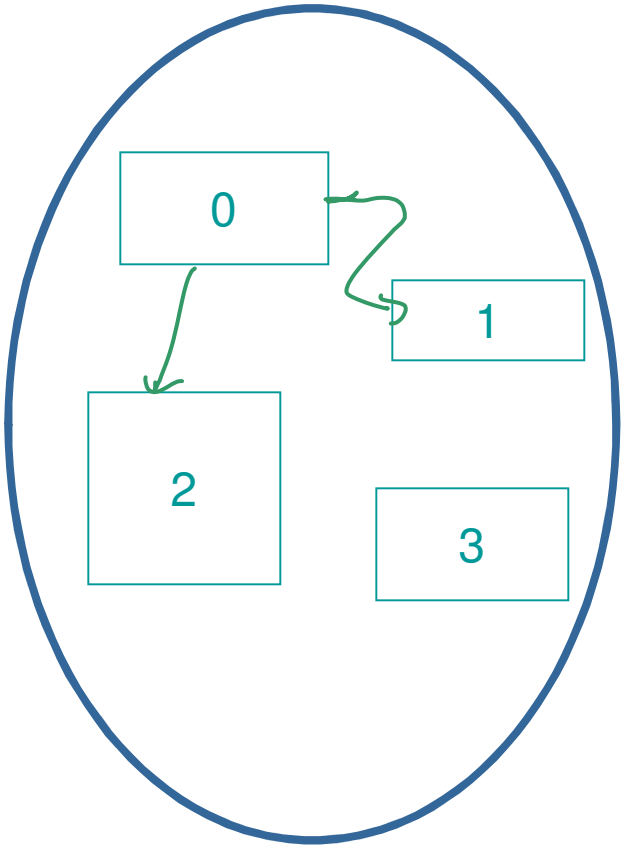
- **A fin réduire le besoin de compression, la prochaine option est d'utiliser l'allocation non contiguë**
  - diviser un programme en modules et permettre l`allocation séparée de chaque module
  - les modules sont beaucoup plus petits que le programme entier et donc permettent une utilisation plus efficace 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 segments: main, méthodes, variables globales, objets, matrices, etc.)
  - la pagination utilise des parties de programme arbitraires (morcellement du programmes en pages de longueur fixe).
  - elles peuvent être combinées

Les segments sont des parties logiques du programme

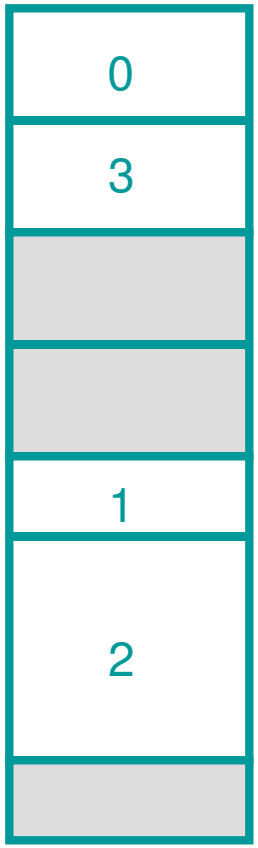


4 segments: A, B, C, D

Les segments comme unités d'alloc mémoire



espace usager

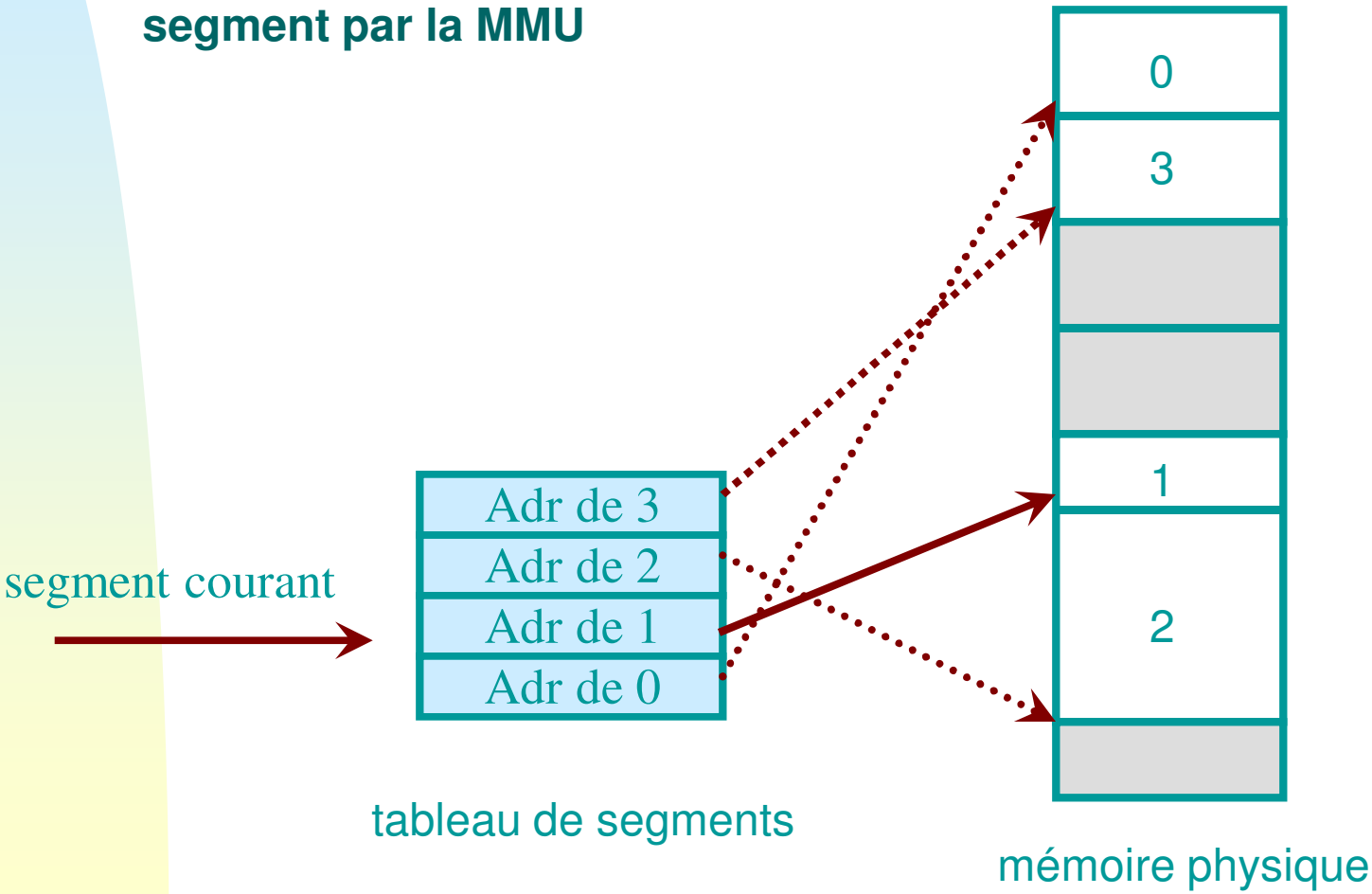


mémoire physique

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**



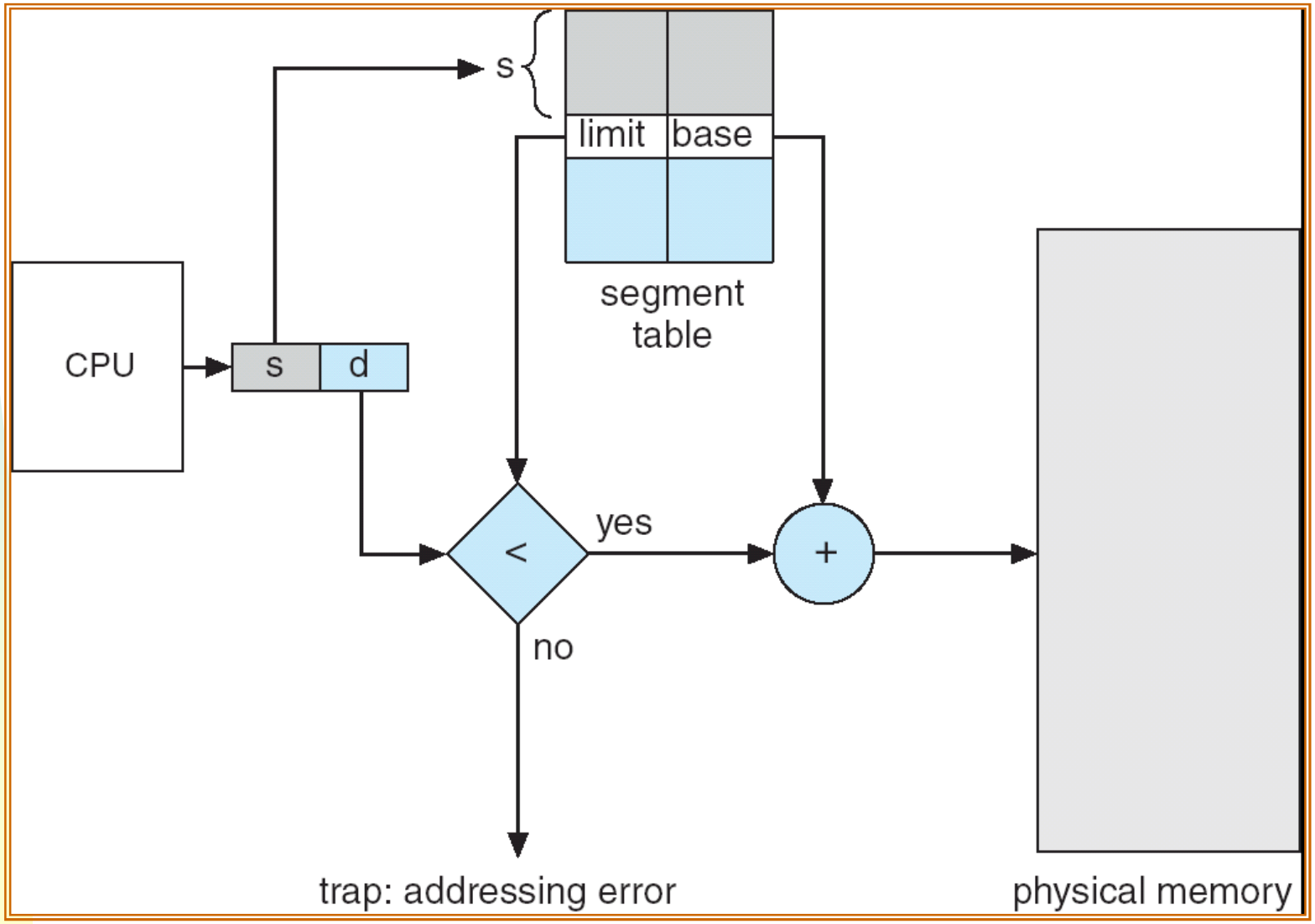
Détails

- **L'adresse logique consiste d 'une paire:**

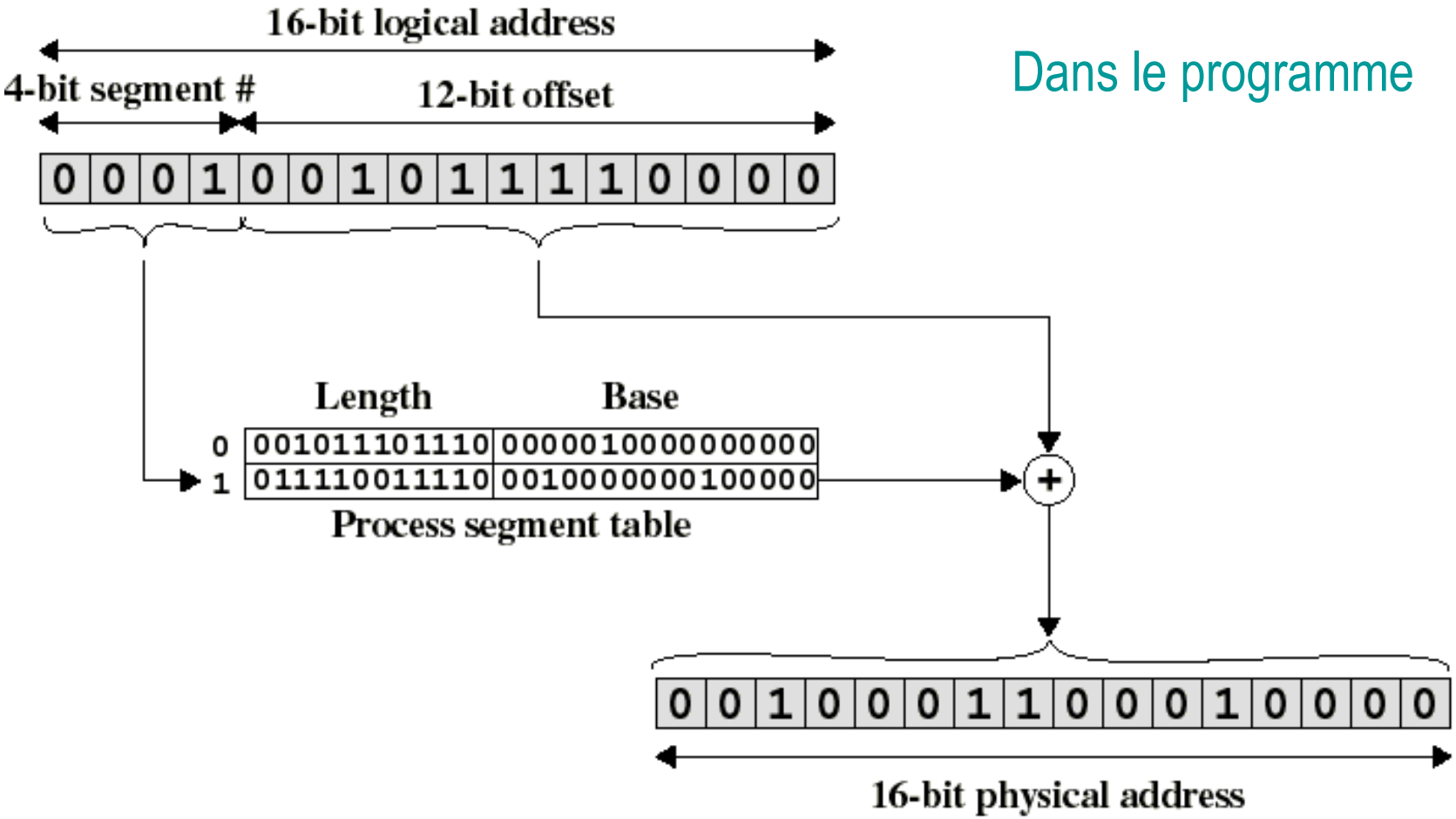
  <No de segm, décalage>

  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, on verra…
- **Dans le PCB du processus il y aura un pointeur à l 'adresse en mémoire du tableau des segments**
- **Il y aura aussi le nombre de segments dans le processus**
- **Au moment de la commutation de contexte, ces infos seront chargées dans les registres appropriés d'UCT**

Conversion d'adresses dans la segmentation



Le mécanisme en détail (implanté dans le matériel)

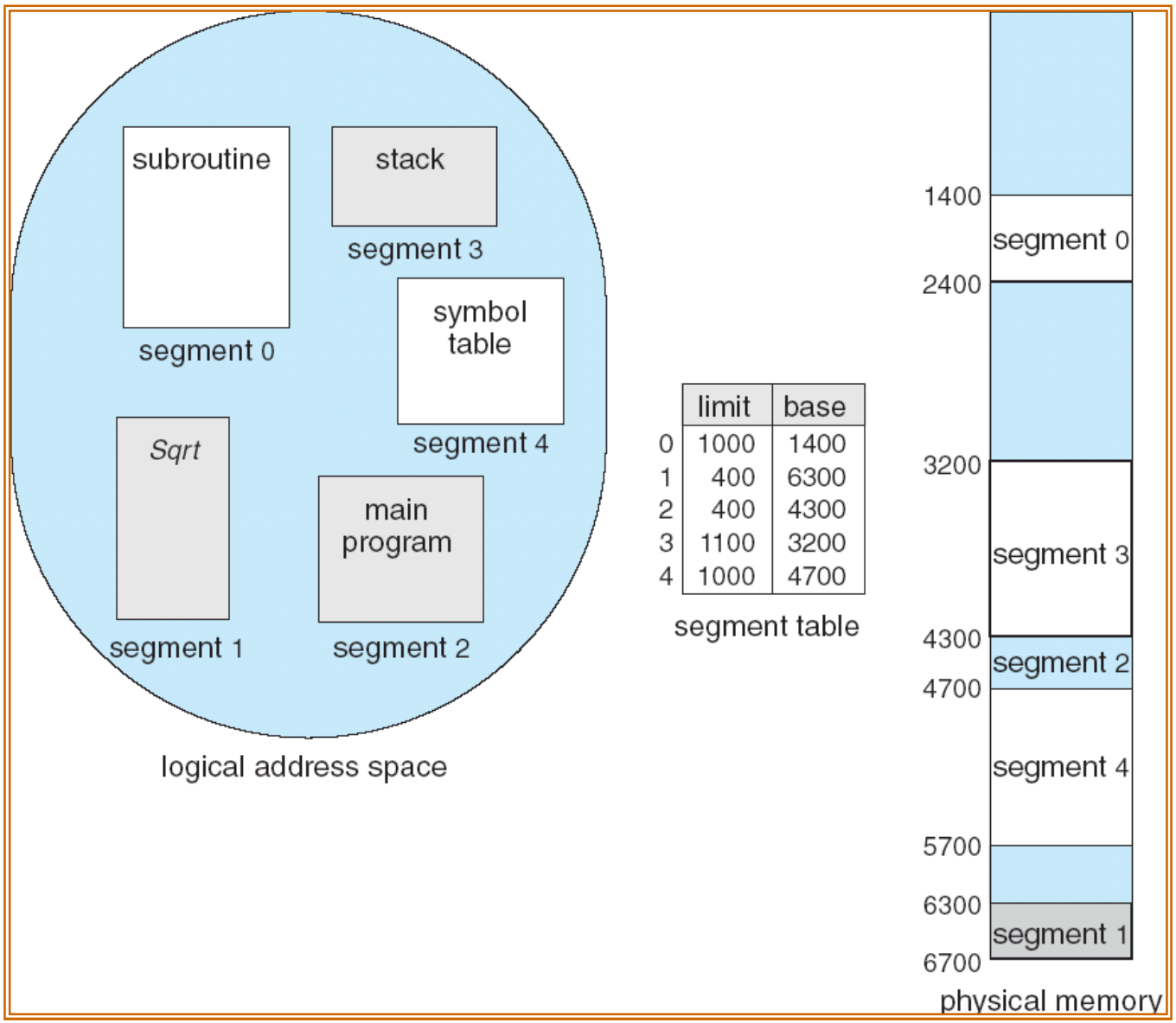


Dans le programme

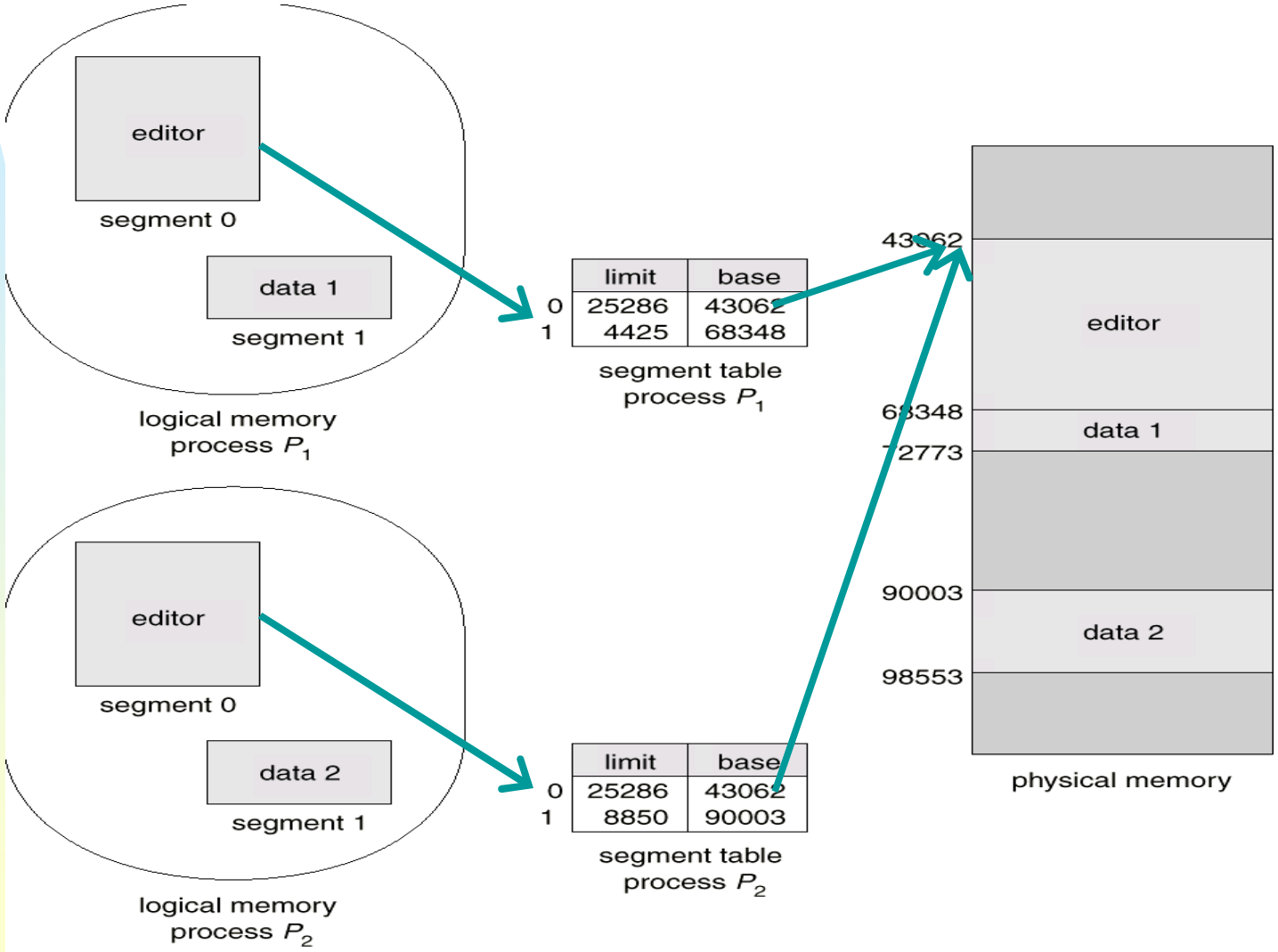
Stallings

Adresse finale

Exemple de la segmentation simple

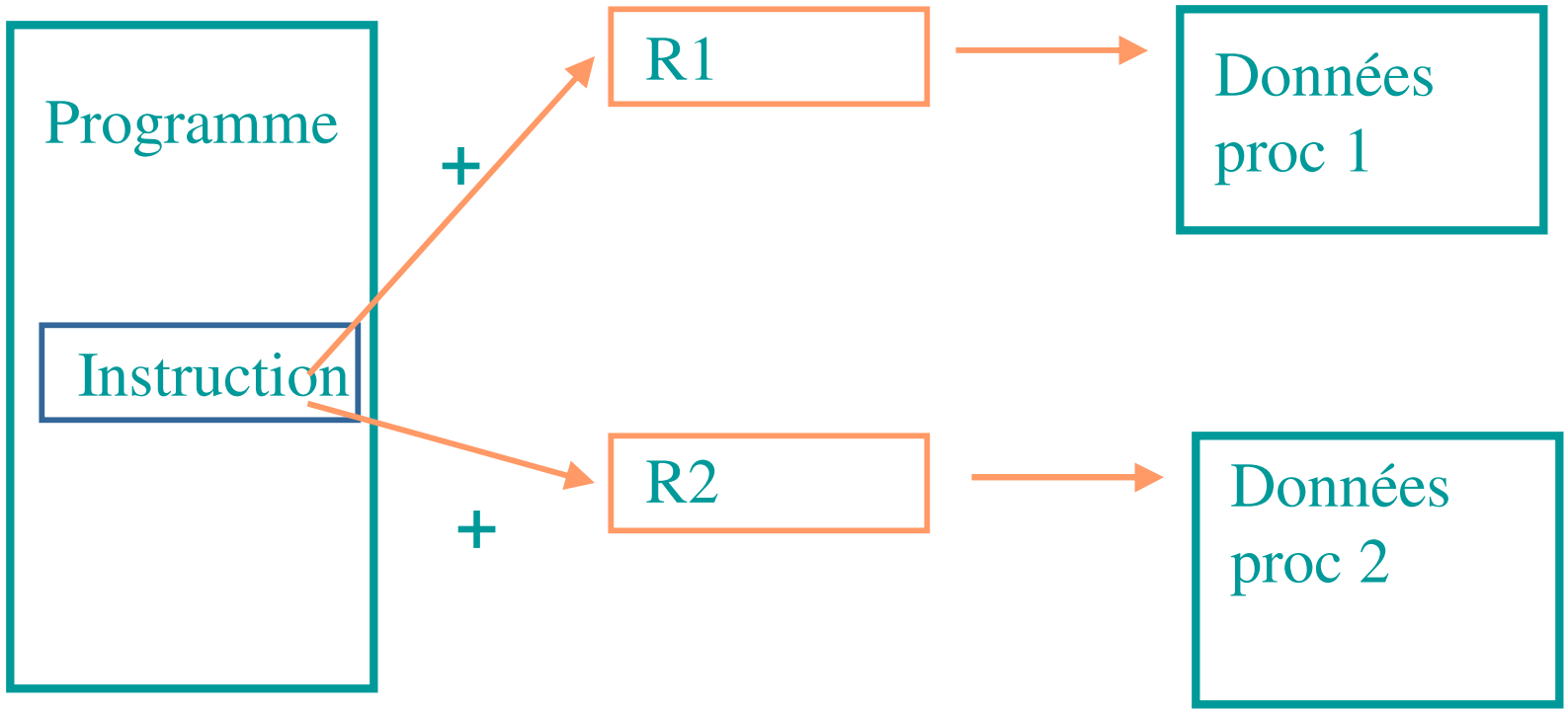


Partage de segments: le segment 0 est partagé



ex: DLL utilisé par plusieurs usagers

Mécanisme pour 2 processus qui exécutent un seul programme sur données différentes



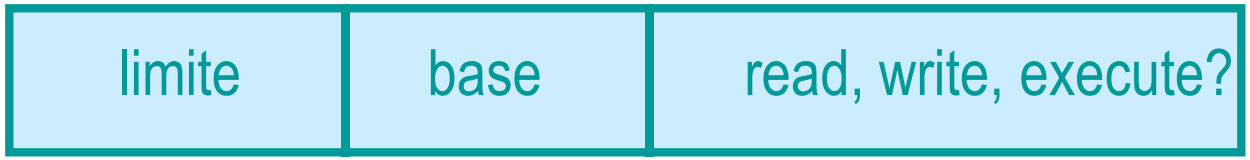
La même instruction, si elle est exécutée

- par le proc 1, son adresse est modifiée par le contenu du reg de base 1
- par le proc 2, son adresse est modifiée par le contenu du reg de base 2

Ceci fonctionne même si l'instruction est exécutée par plusieurs UCT au même instant, si les registres se trouvent dans des UCT différentes

Segmentation et protection

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



Évaluation de la segmentation simple

- **Avantages: l'unité d'allocation de mémoire 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 aisés (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 de simplifier le mécanisme en utilisant des unités d`allocation de mémoire de tailles égales**

    - **PAGINATION**

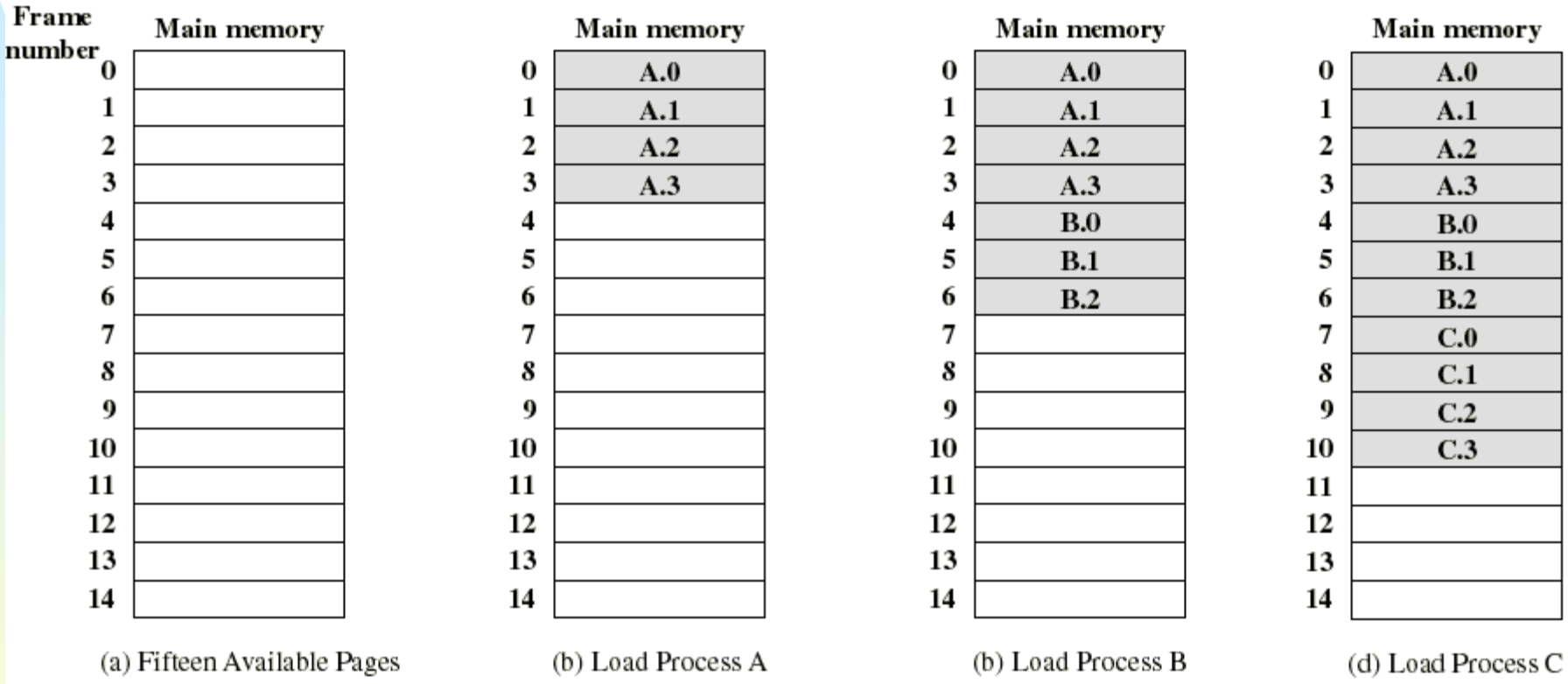
Segmentation versus pagination

- 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



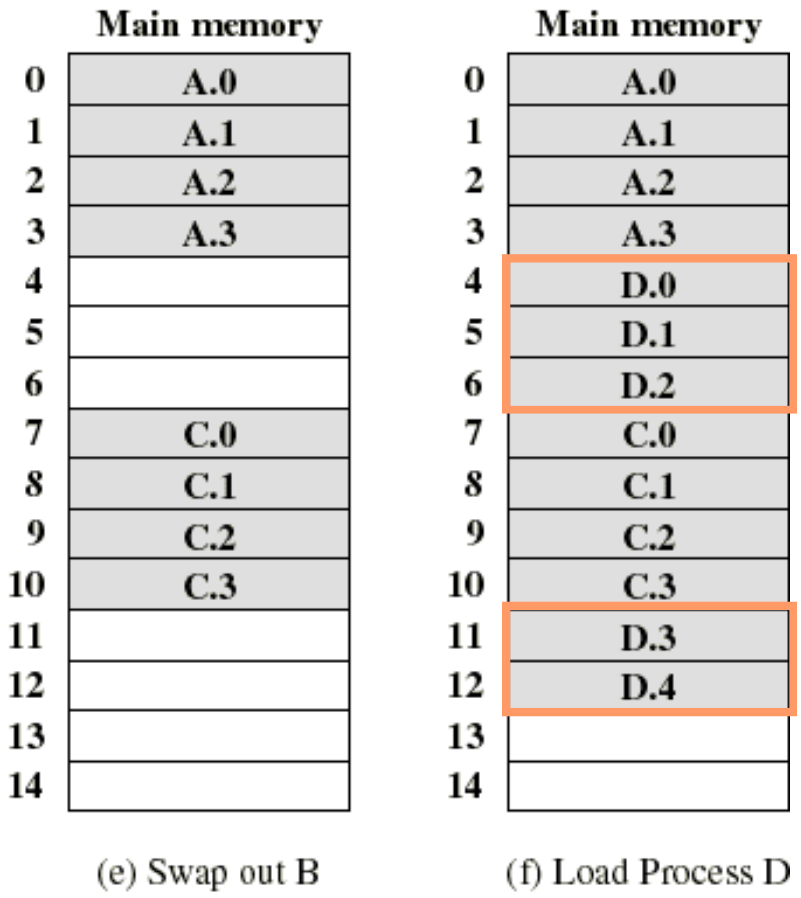
(a) Fifteen Available Pages (b) Load Process A (b) Load Process B (d) Load Process C

- **Supposons que le processus B se termine ou est suspendu**

Stallings

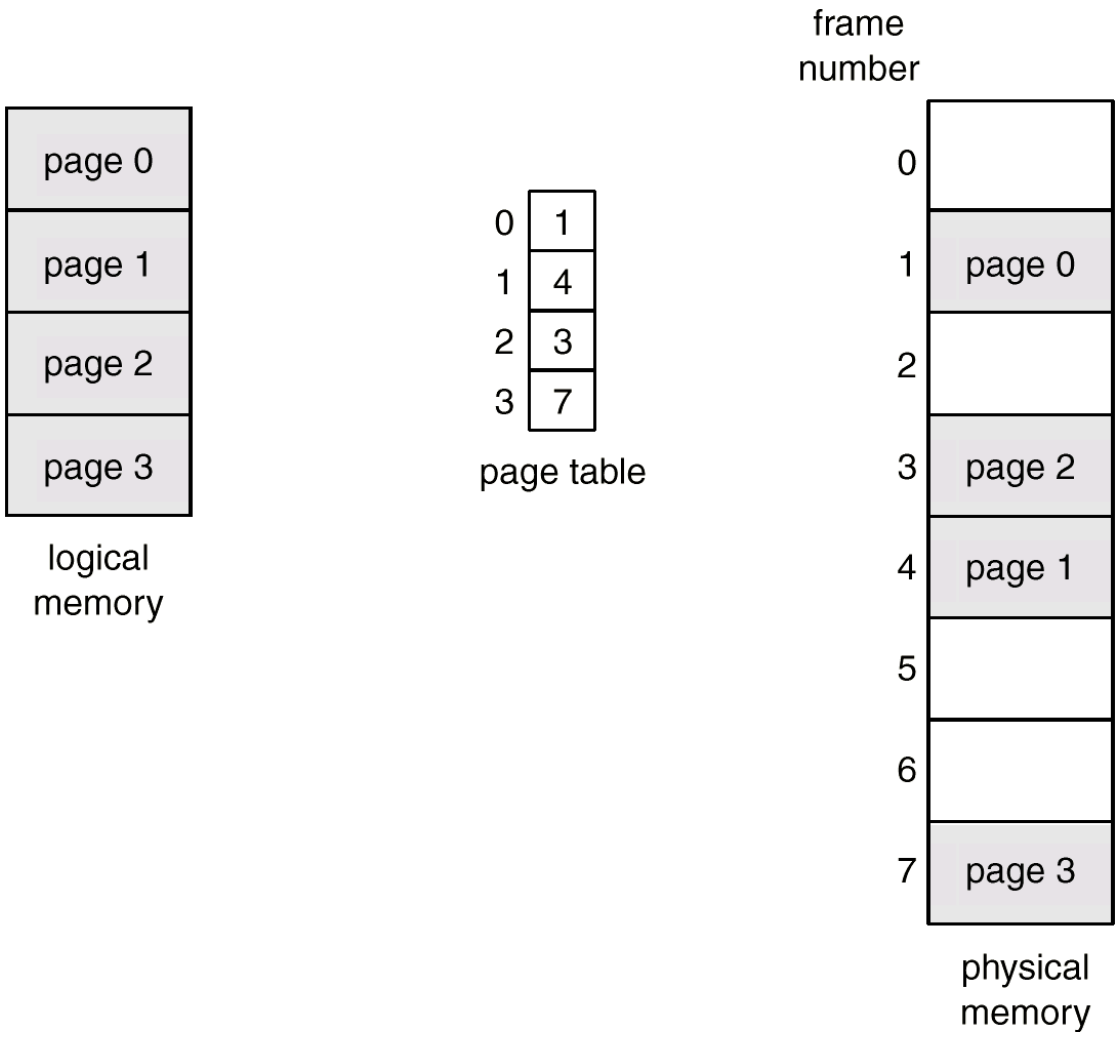
Exemple de chargement de processus (Stallings)

- **Nous pouvons maintenant transférer en mémoire un programme D, qui demande 5 pages**
  - bien qu`il n'y ait pas 5 pages contigus disponibles
- **La fragmentation externe est limitée lorsque 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 (moy. 1/2 cadre par proc)**



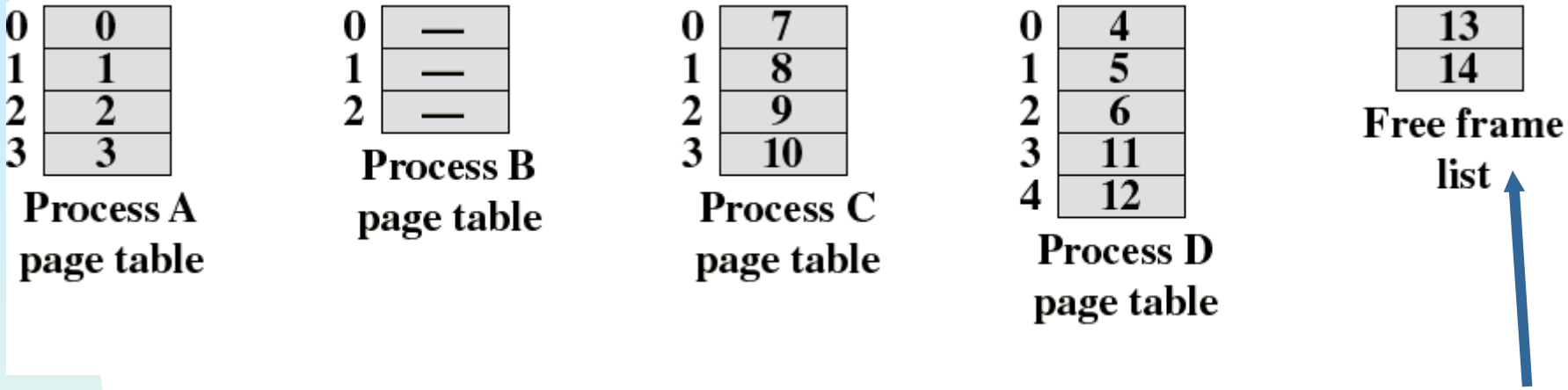
(e) Swap out B (f) Load Process D

Tableaux de pages



Tableaux de pages

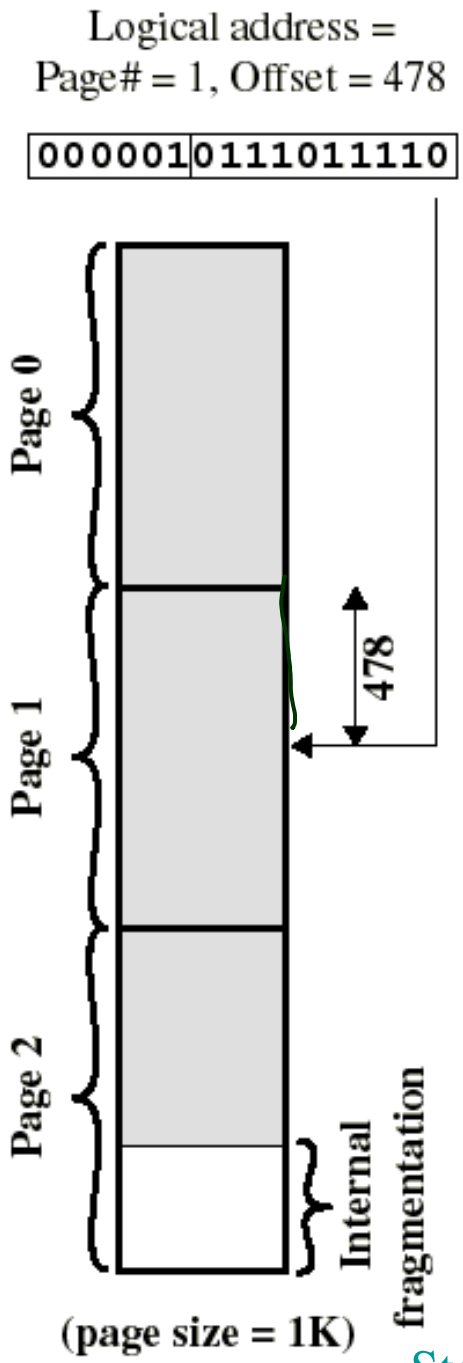
Stallings



- 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
- Un tableau 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)

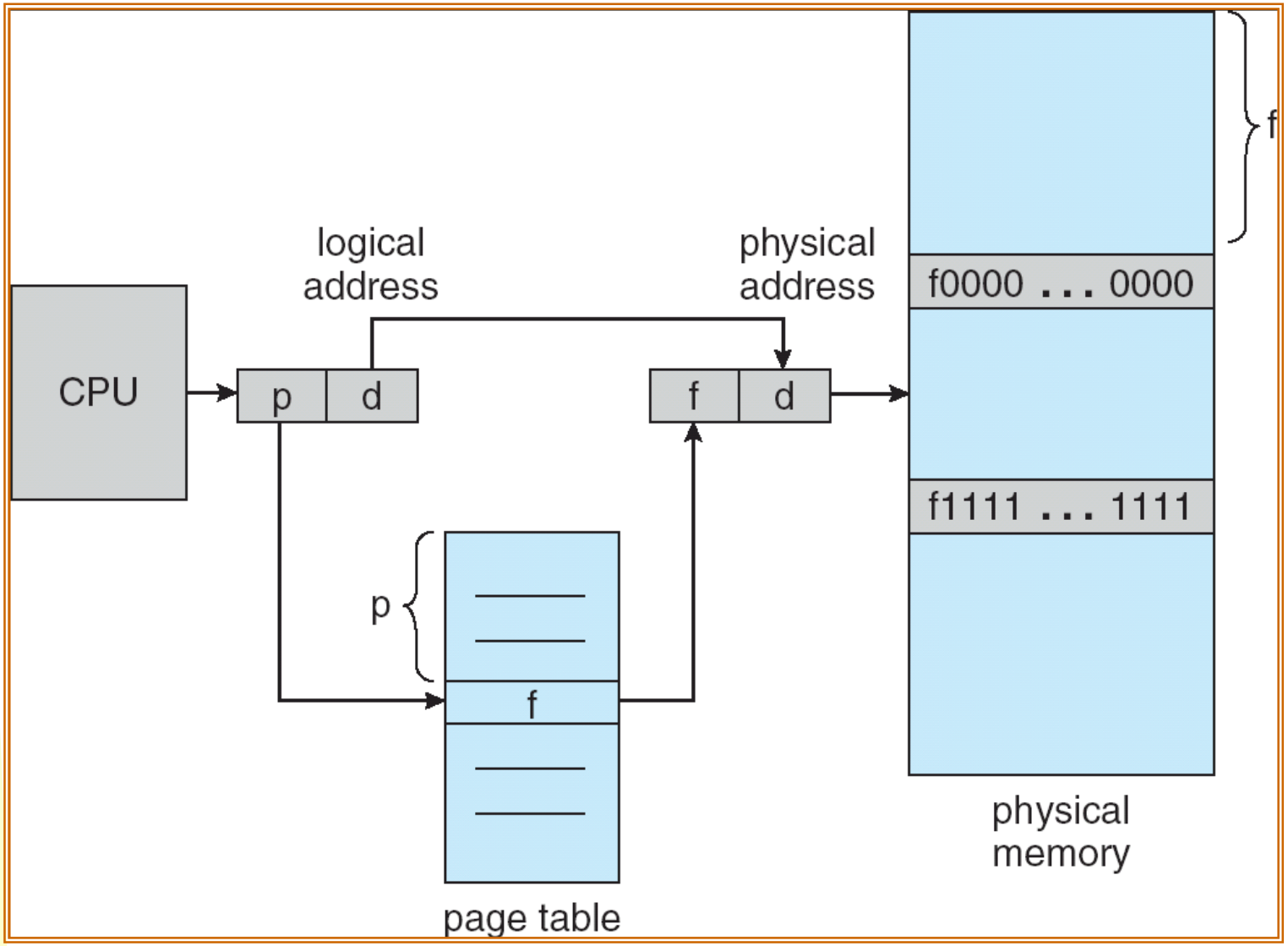
- L'adresse logique est facilement traduite en adresse physique car la taille des pages est une puissance de 2
- Ex: si 16 bits sont utilisés pour les adresses et que la taille d'une page = 1K: on a besoins de 10 bits pour le décalage, laissant ainsi 6 bits pour le numéro de page
- L'adresse logique (n,m) est convertie en une adresse physique (k,m) en utilisant n comme index sur la table des pages et en le remplaçant par l 'adresse k trouvée
  - m ne change pas



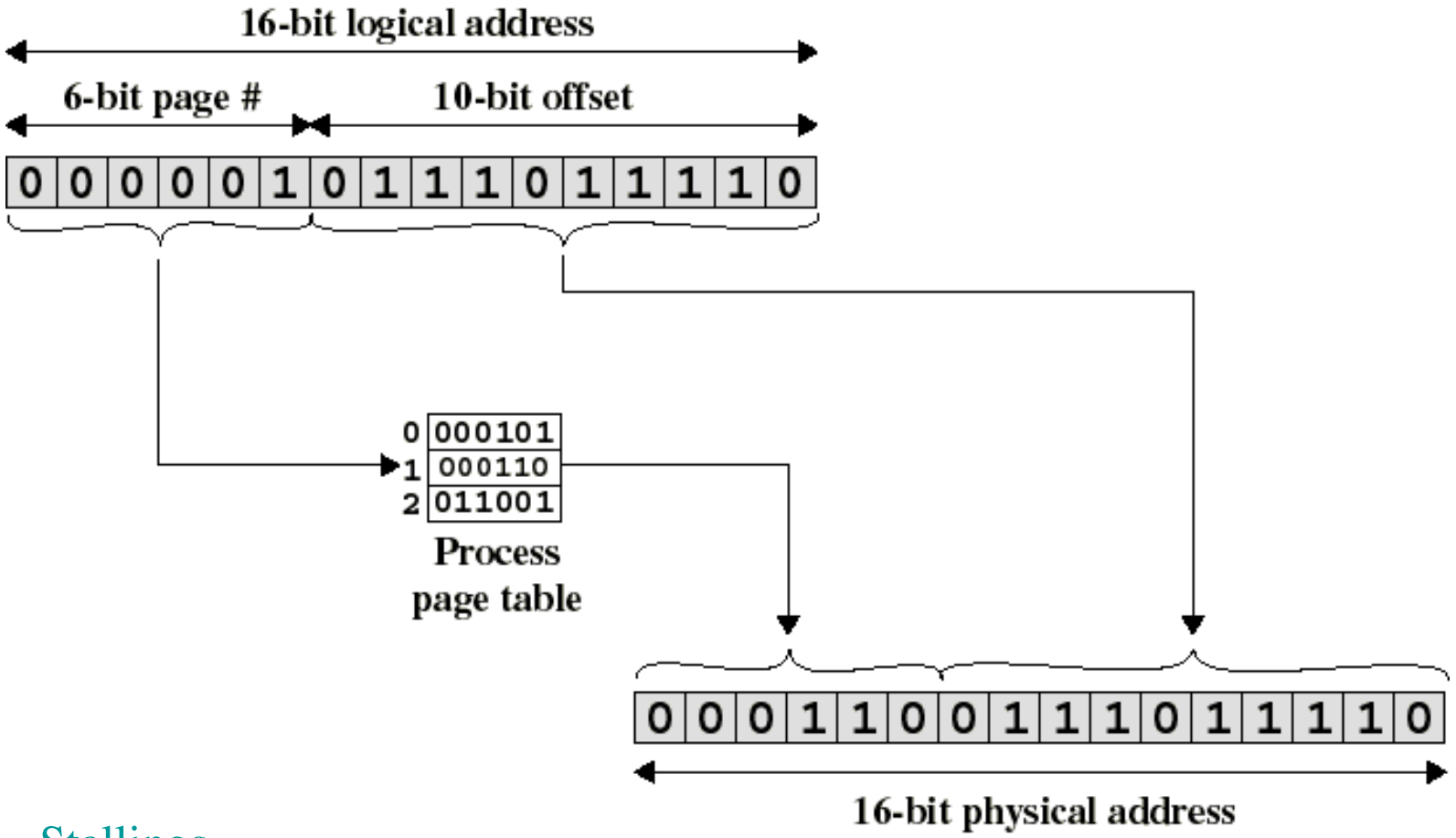
Adresse logique (pagination)

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

Mécanisme: matériel



Conversion d'adresse (logique-physique) de la pagination



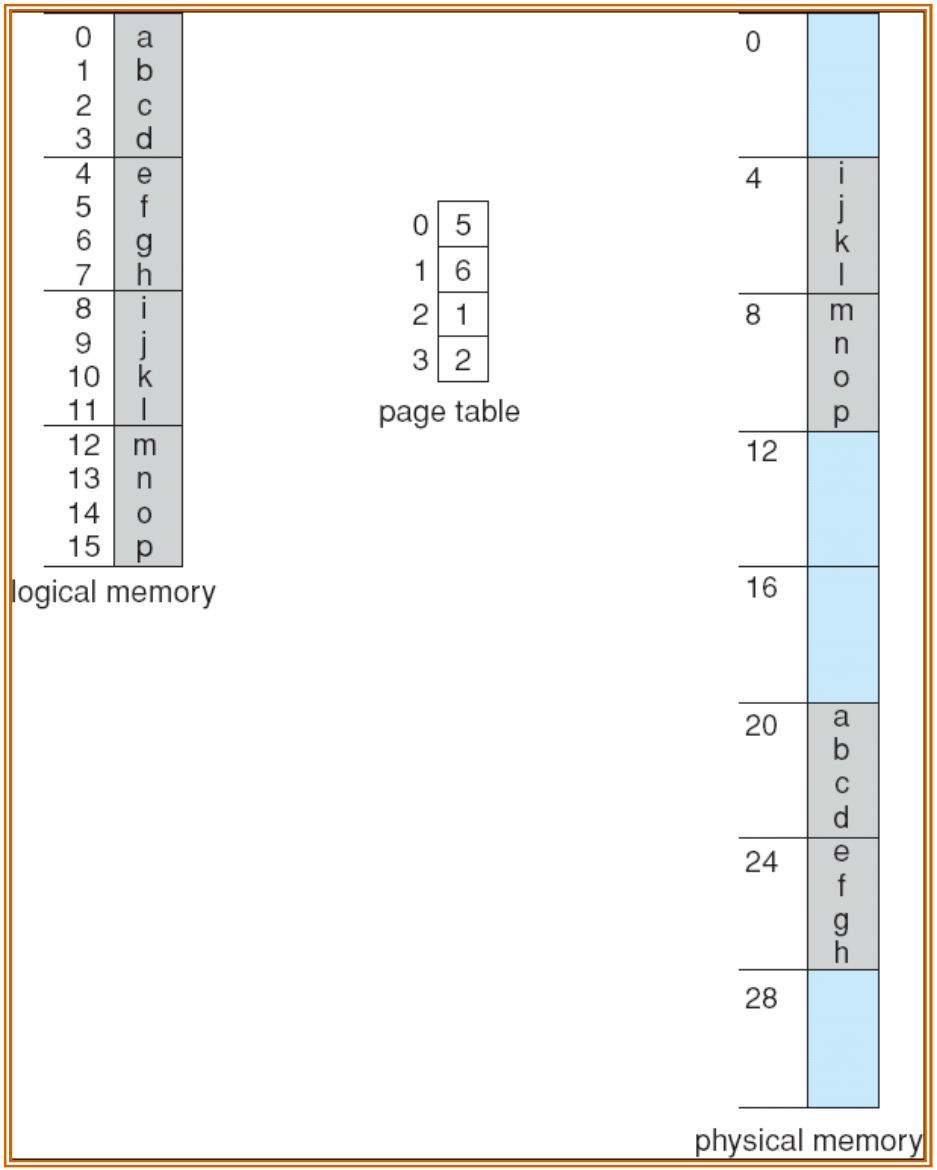
Stallings

Conversion d'adresses: segmentation et pagination

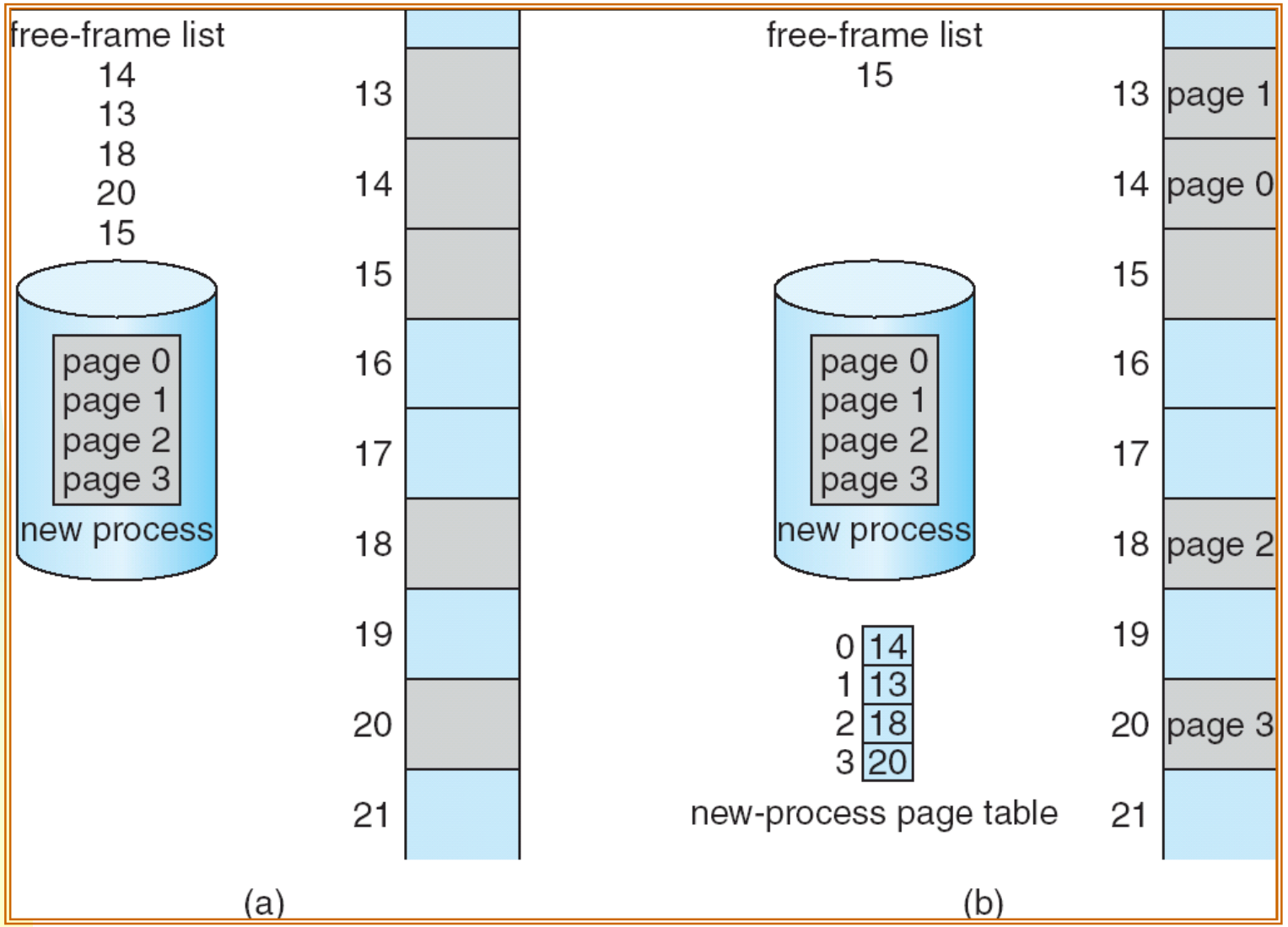
- **Que ce soit dans la segmentation ou dans la pagination, nous *additionnons* le décalage à l'adresse du segment ou page.**
- **Cependant, dans la pagination, les adresses de pages sont toujours des multiples de 2, et il y a autant de 0 à droite qu'il y a de bits dans le décalage... donc l'ajout peut être faite par simple concaténation:**

11010000 +1010
=
1101 1010

Exemple de pagination



Liste de trames libres



Problèmes d'efficacité

- **La conversion d`adresses, y compris la recherche des adresses des pages et de segments, est exécutée par des mécanismes de matériel**
- **Cependant, si la table des pages est en mémoire principale, chaque adresse logique occasionne au moins 2 références de la mémoire**
  - Une pour lire l'entrée de la table de pages
  - L'autre pour lire le mot référencé
- **Le temps d`accès de mémoire est doublé...**

Pour améliorer l`efficacité

- **Où mettre les tables des pages (les mêmes idées s'appliquent aussi aux tables de segments)**
- **Solution 1: dans des registres de UCT.**
  - avantage: vitesse
  - désavantage: nombre limité de pages par processus, la taille de la mémoire logique est limitée
- **Solution 2: en mémoire principale**
  - avantage: taille de la mémoire logique illimitée
  - désavantage: mentionné
- **Solution 3 (mixte): les tableaux de pages sont en mémoire principale, mais les adresses les plus utilisées sont aussi dans des registres de l`UCT.**

Régistres associatifs

TLB: Translation Lookaside Buffers, ou *caches* d'adressage

- **Recherche parallèle d'une adresse:**
  - l'adresse recherchée est cherchée dans la partie gauche de la table en parallèle (matériel spécial)
- **Conversion page → frame**
  - Si la page recherchée a été utilisée récemment elle se trouvera dans les registres associatifs
    - recherche rapide

No Page	No Frame
3	15
7	19
0	17
2	23

Recherche *associative* dans TLB

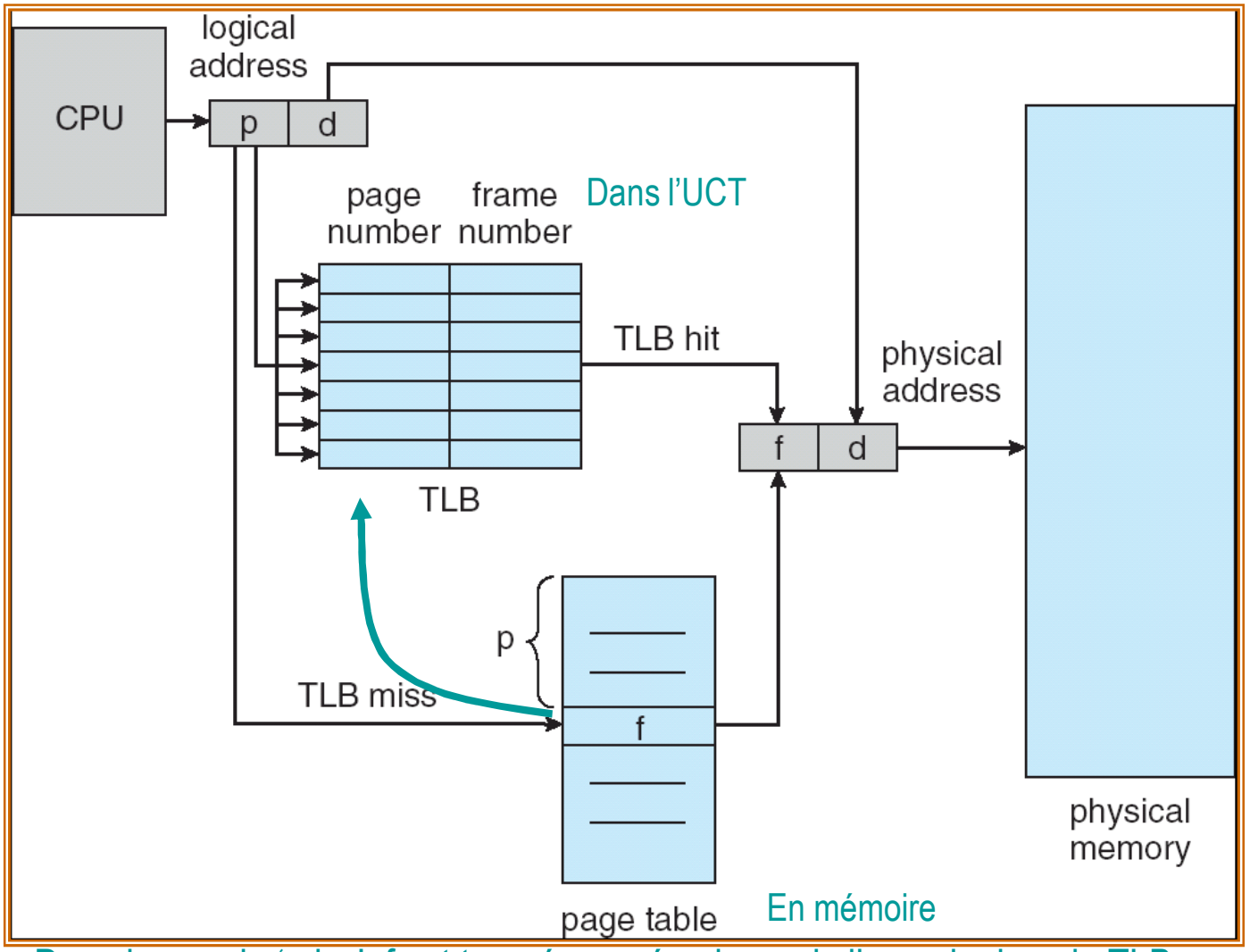
- **Le TLB est un petit tableau de registres de matériel où chaque ligne contient une paire:**
  - Numéro de page logique, Numéro de cadre
- **Le TLB utilise du matériel de *mémoire associative*: interrogation simultanée de tous les numéros logiques pour trouver le numéro physique recherché**
- **Chaque paire dans le TLB est fournie d'un indice de référence pour savoir si cette paire a été utilisée récemment. Sinon, elle est remplacée par la dernière paire dont on a besoin**

Translation Lookaside Buffer (TLB)

- **Sur réception d'une adresse logique, le processeur examine la cache TLB**
- **Si cette entrée de page y est , le numéro de frame en est extrait**
- **Sinon, le numéro de page indexe la table de page du processus (en mémoire)**
  - Cette nouvelle entrée de page est mise dans le TLB
  - Elle remplace une autre pas récemment utilisée
- **Le TLB est vidé quand l'UCT change de processus**

- **Les premières trois opérations sont faites par matériel**

Schéma d'utilisation TLB



Dans le cas de 'miss', f est trouvé en mémoire, puis il es mis dans le TLB

Temps d'accès réel

- **Recherche associative = ε unités de temps (normalement petit)**
- **Supposons que le cycle de mémoire soit 1 microseconde**
- **α = pourcentage de touches (hit ratio) = pourcentage de fois qu'un numéro de page est trouvé dans les registres associatifs**
  - ceci est en relation avec le nombre de registres associatifs disponibles

- **Temps effectif d'accès t_{ea}:**

$$t_{ea} = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

si α est près de 1 et ε est petit, ce temps sera près de 1.

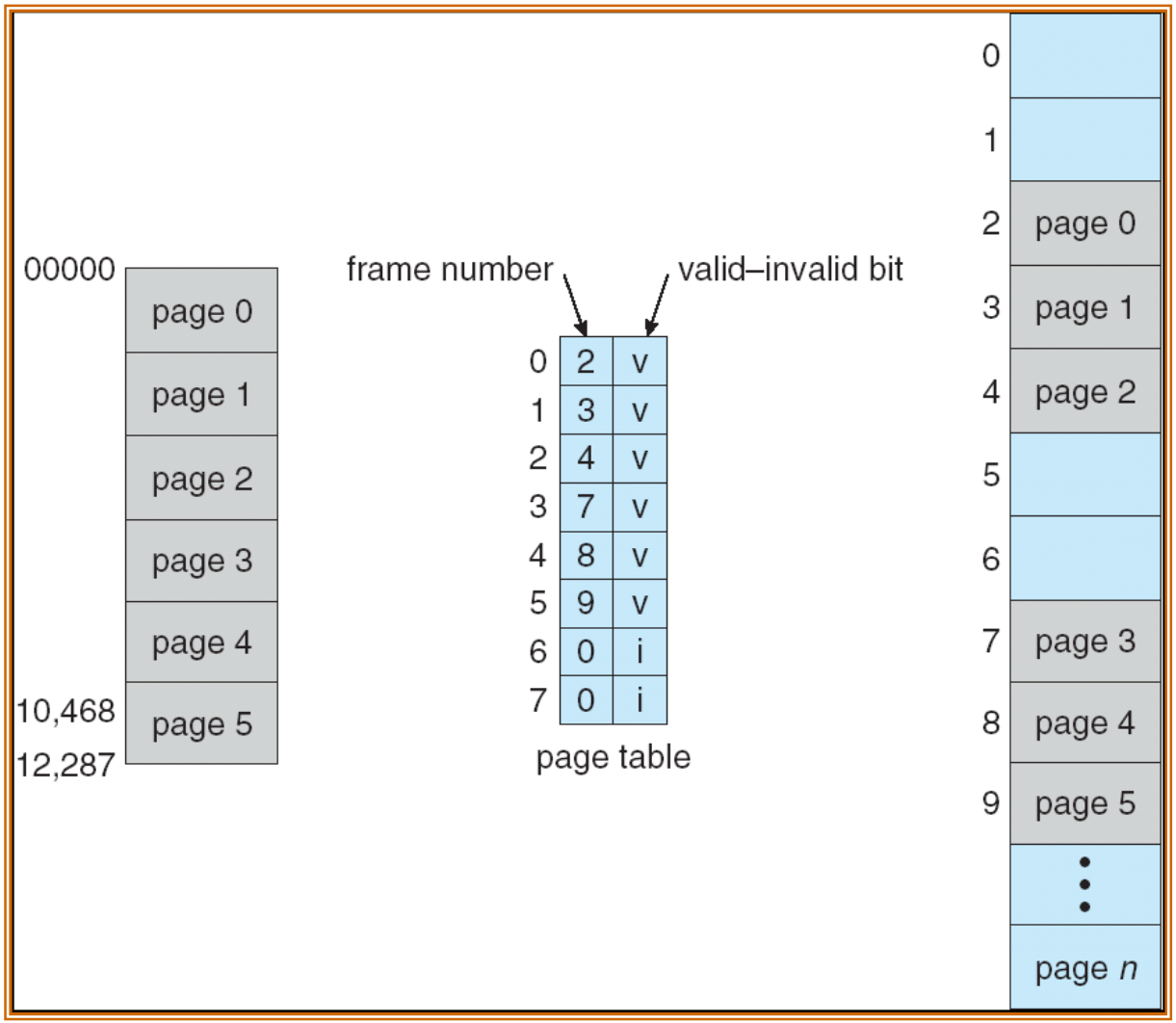
- **Généralisation de la formule prenant m comme temps d'accès à la mémoire centrale:**

$$t_{ea} = (m+ \varepsilon)\,\alpha + (2m+ \varepsilon)(1- \alpha) = m\,\alpha + \varepsilon\alpha + 2m - 2m\,\alpha + \varepsilon - \varepsilon\alpha =$$
$$= 2m + \varepsilon - m\,\alpha$$

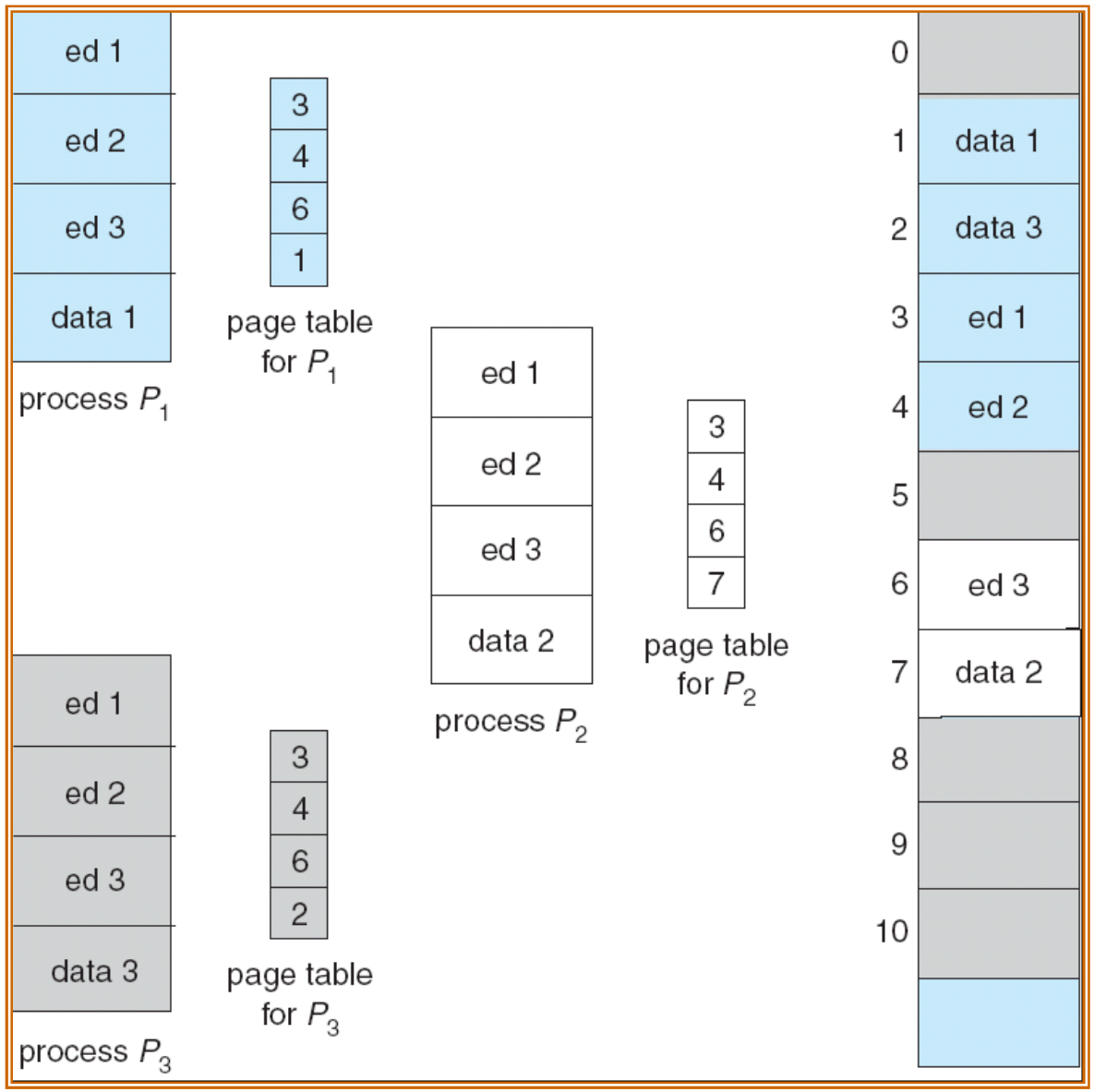
Protection de la mémoire

- **Associe plusieurs bits à chaque page dans le tableau de pages aussi bien que dans le tableau de segments**
  - Ex. bit valide-invalide indique les pages valides du processus
- **Pour vérifier si une adresse est valide**
  - Avec un tableau de pages ayant une grandeur fixe: bits valide-invalide
  - Avec un tableau de pages ayant une grandeur variable (seulement les pages valides du processus): Compare le # de page avec le registre PTLR (page-table length registre)

Bit Valide-Invalide (v-i) dans le tableau de pages



Partage de pages:
3 proc. partageant un éditeur, sur des données privées de chaque proc



Structure des tableaux de pages

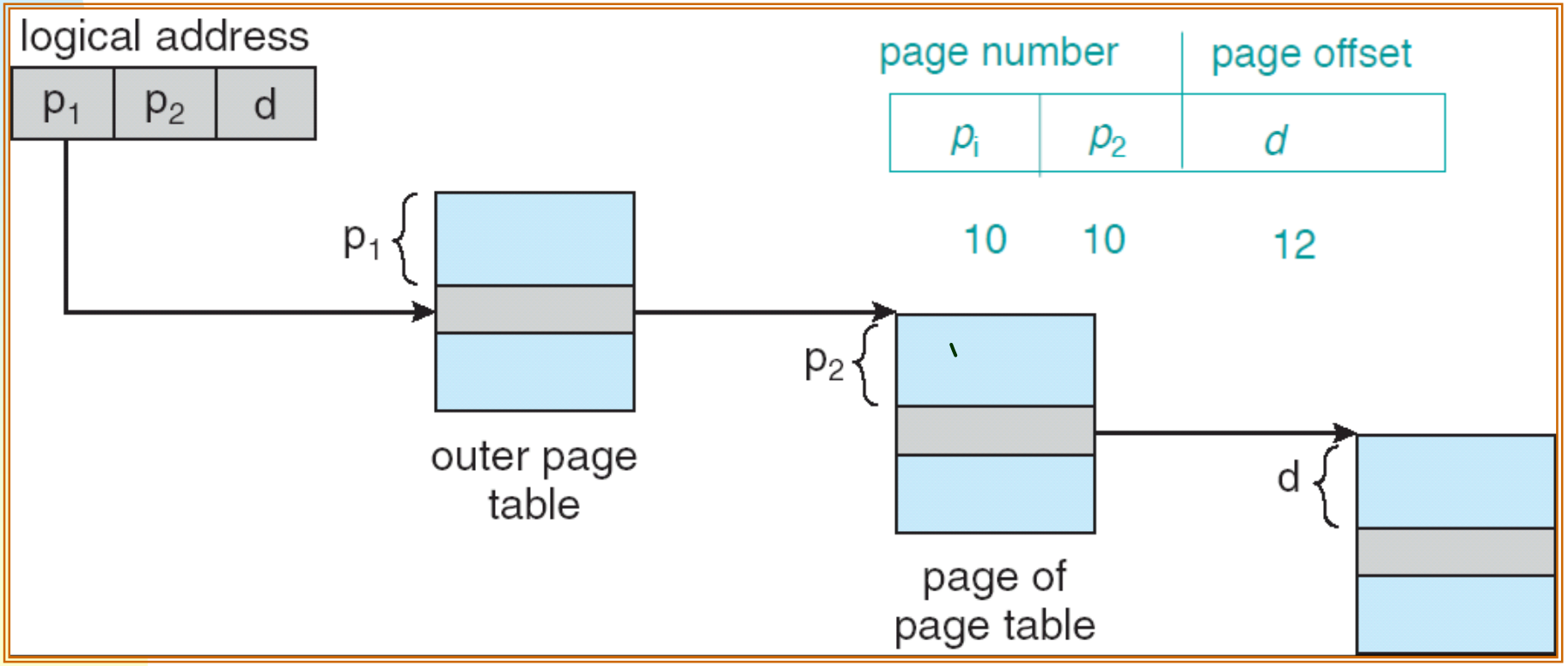
Quelle longueur peut-être un tableau de pages?

- Adresses de 32-bit, page de 4k octets → 2^{20} pages = 1M entrées de page!

Tableau de pages hiérarchique

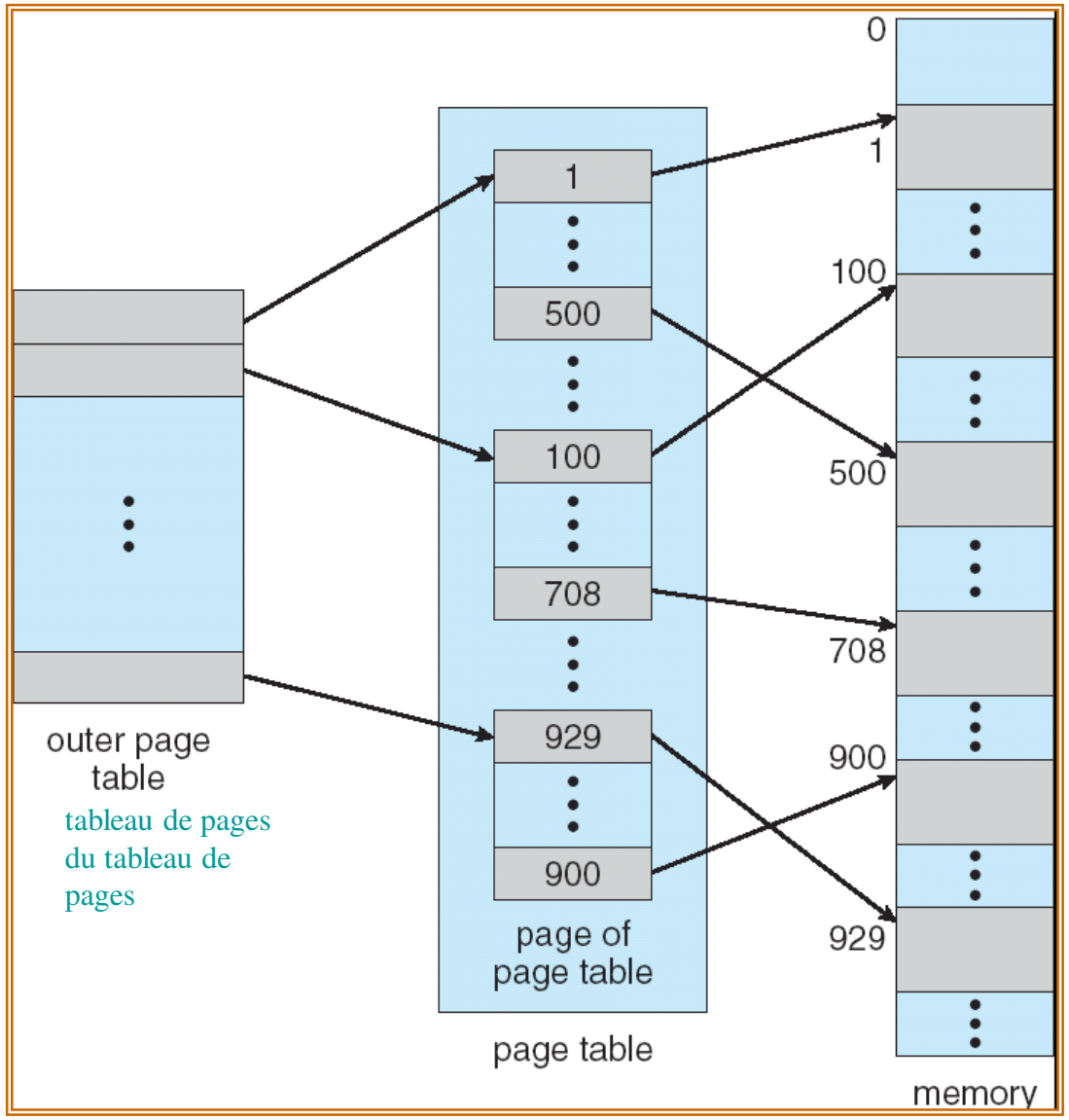
Tableaux de pages hiérarchique à deux niveaux

- **La partie de l'adresse qui appartient au numéro de page est elle-même divisée en 2**



Tableaux de pages hiérarchique à deux niveaux

(quand les tableaux de pages sont très grands, ils peuvent être eux mêmes paginés)



Utilisation de Translation Lookaside Buffer

- Dans le cas de systèmes de pagination à plusieurs niveaux, l'utilisation de TLB devient encore plus importante pour éviter les accès multiples de la mémoire pour calculer une adresse physique
- Les adresses les plus récemment utilisées sont trouvées directement dans la TLB.

Segmentation simple vs Pagination simple

- **La pagination se préoccupe seulement du problème du chargement, tandis que**
- **La segmentation vise aussi le problème de la liaison**
- **La segmentation est visible au programmeur mais la pagination ne l'est pas**
- **Le segment est une unité logique de protection et de 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 conversion d'adresses (addition au lieu d`enchaînement)**
- **La segmentation souffre de fragmentation *externe* (partitions dynamiques)**
- **La pagination produit de la fragmentation *interne*, mais pas beaucoup (1/2 frame par programme)**
- **Heureusement, la segmentation et la pagination peuvent être combinées**

Conclusions sur la Gestion Mémoire

- **Problèmes de:**
  - fragmentation (interne et externe)
  - complexité et efficacité des algorithmes
- **Deux méthodes très utilisées**
  - Pagination
  - Segmentation
- **Problèmes en pagination et segmentation:**
  - taille des tableaux de segments et de pages
  - pagination de ces tableaux
  - efficacité fournie par Translation Lookaside Buffer
- **Les méthodes sont souvent utilisées conjointement, donnant lieu a des systèmes complexes**

Récapitulation sur la fragmentation

- **Partitions fixes: fragmentation interne car les partitions ne peuvent pas être complètement utilisées + fragmentation 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 frame de fragmentation interne par processus**