Mémoire virtuelle

Pagination sur demande

Copy-on-Write

Remplacement de pages

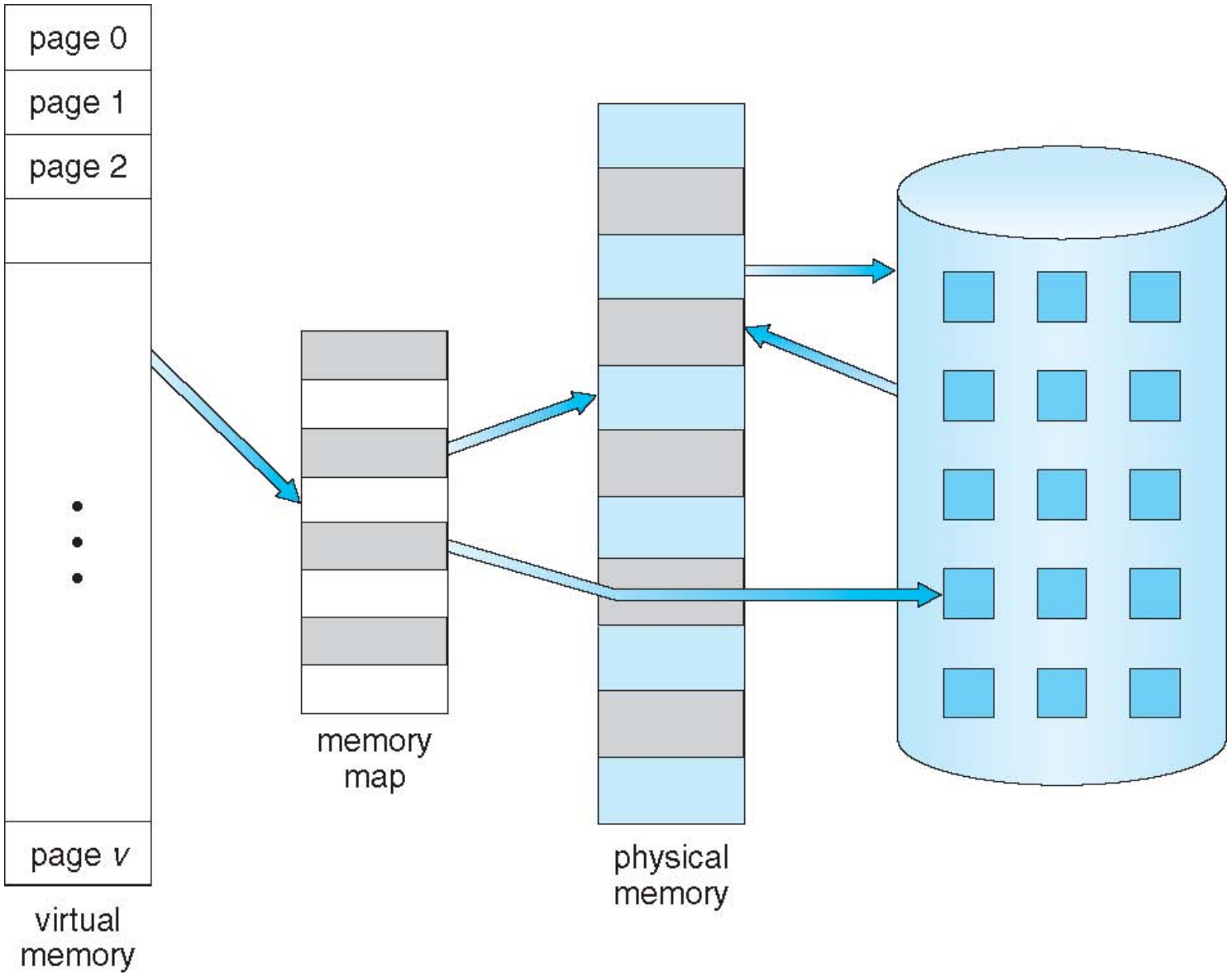
Allocation de *frames*

Thrashing

Fichiers “mappés” en mémoire

Allocation de mémoire du noyau

Schéma de mémoire virtuelle



Pourquoi virtuelle?

Donner l'illusion d'une mémoire infinie

Mémoire centrale est toujours trop petite

Certaines parties du code ne sont jamais exécutées

Certaines données/code momentanément inutiles

Maximiser l'utilité des données en mémoire centrale

Pas besoin de prêter attention à la taille mémoire

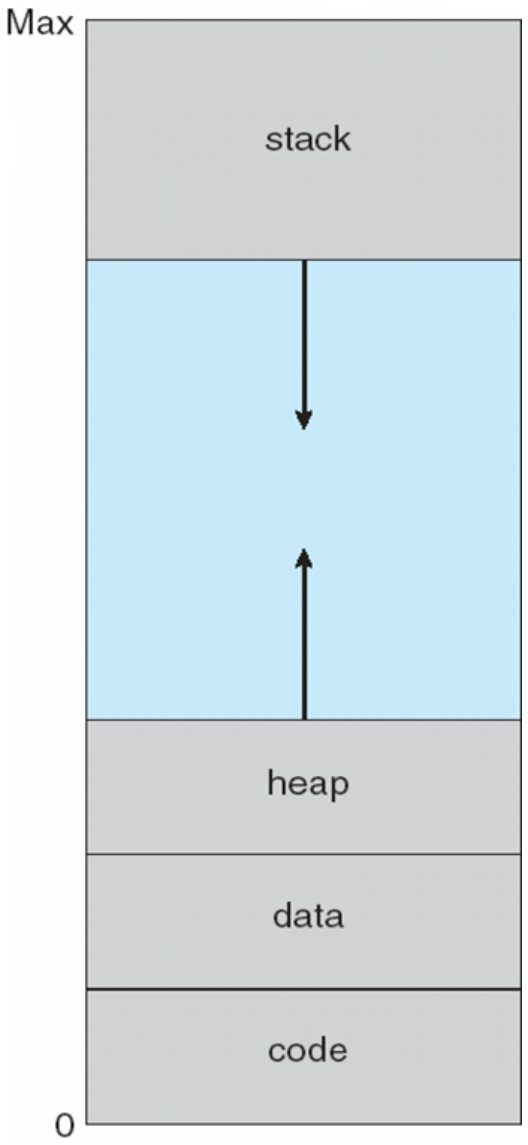
Espace d'adressage virtuel

Espace d'adressage avec des trous

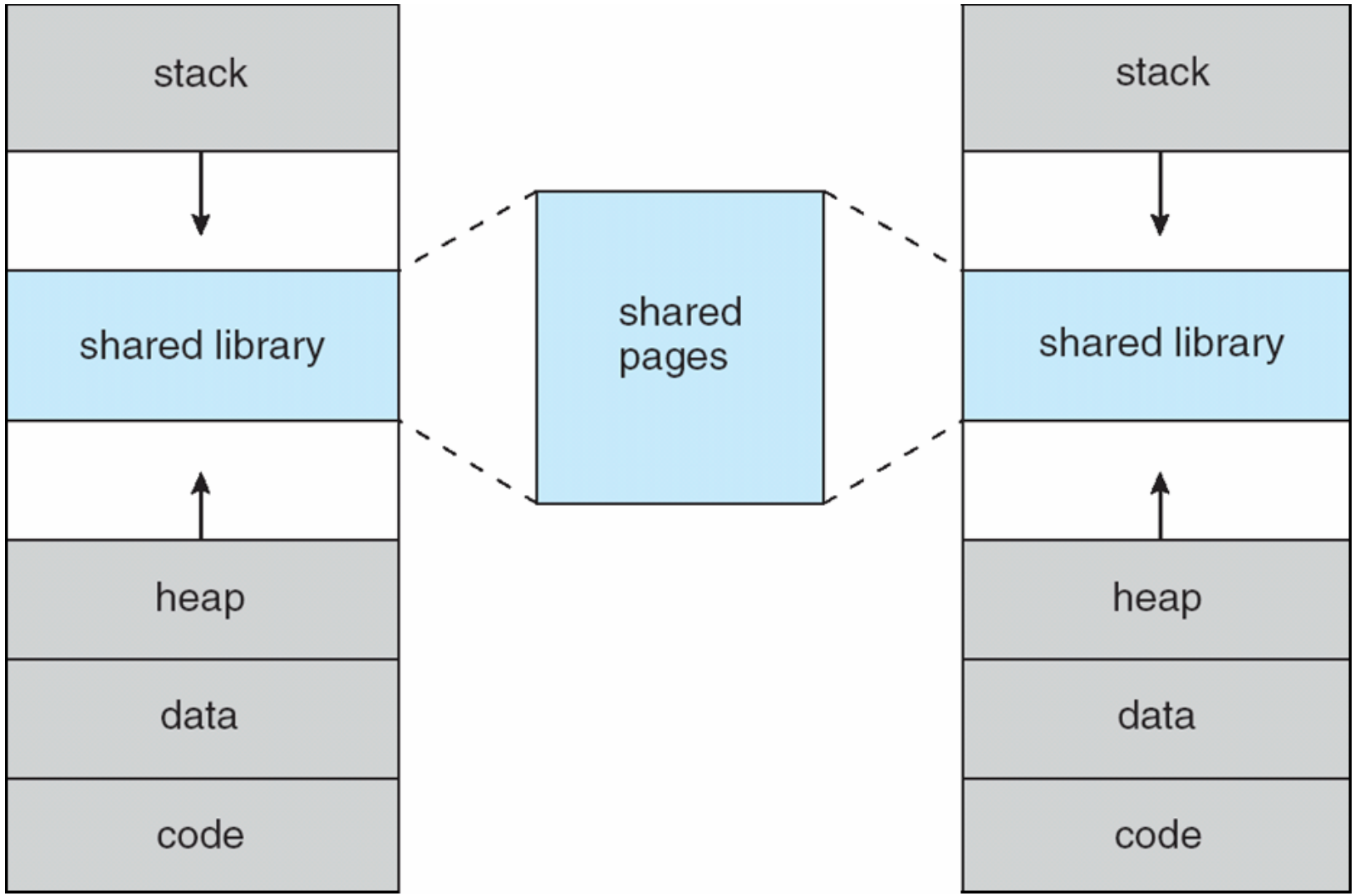
Trous pour faciliter allocations futures:

- Grandir la pile
- Libraires liées dynamiquement

Pages partagées avec d'autres processus



Librairie partagée



Pagination à la demande

Ne pas charger tout le programme au démarrage

Amener les pages en mémoire centrale de manière *paresseuse*

- Moins d'entrées/sorties, moins de mémoire utilisée
- Réponse plus rapide, plus de processus en mémoire

Accès mémoire:

1. En mémoire $\Longrightarrow$ le CPU se charge de tout
2. Sinon, *trap* vers le SE
3. Le SE vérifie si la page existe: si non $\Longrightarrow$ *coredump*
4. Si oui, charger la page du disque et ressayer

Transfert du/vers le disque

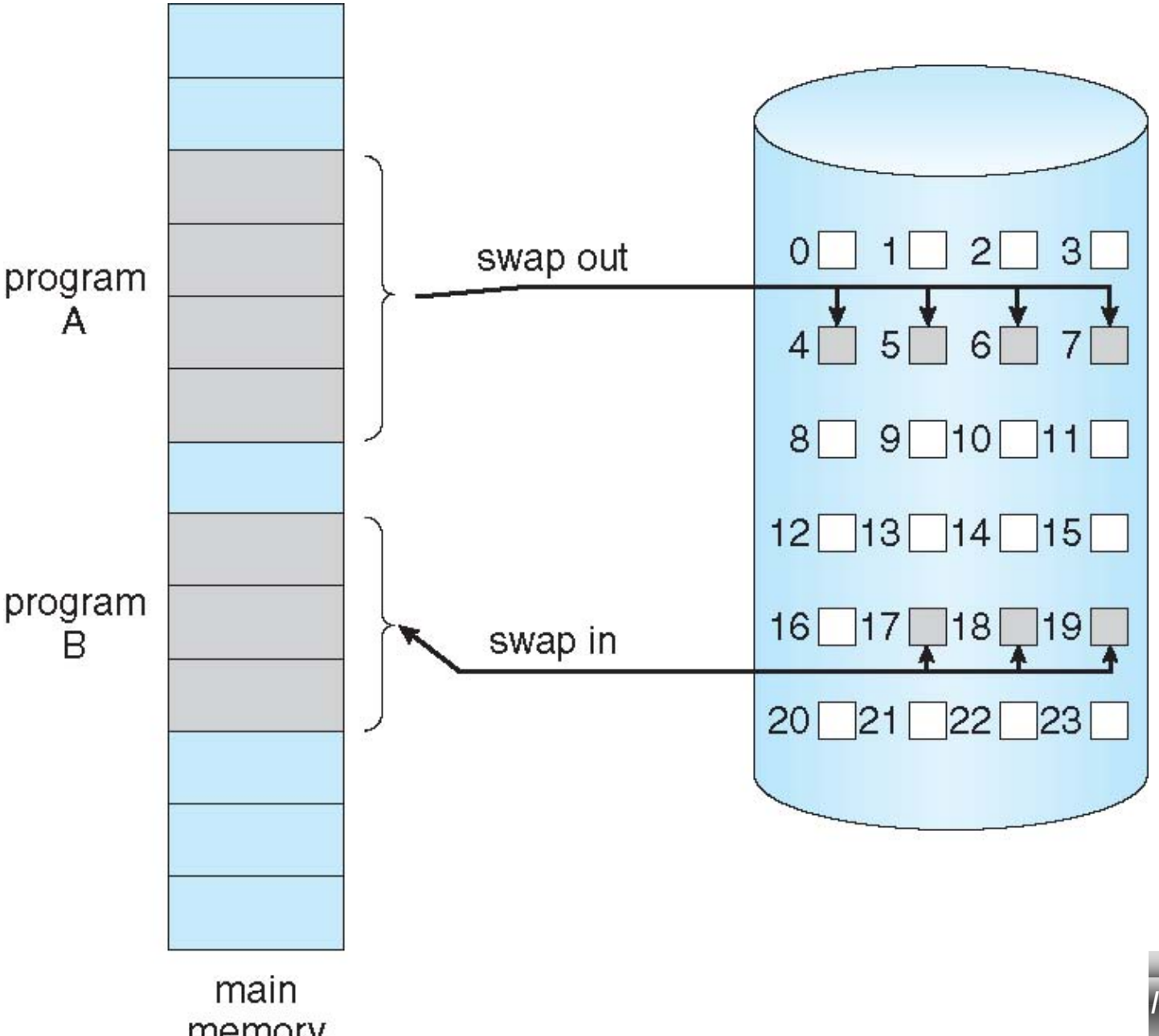
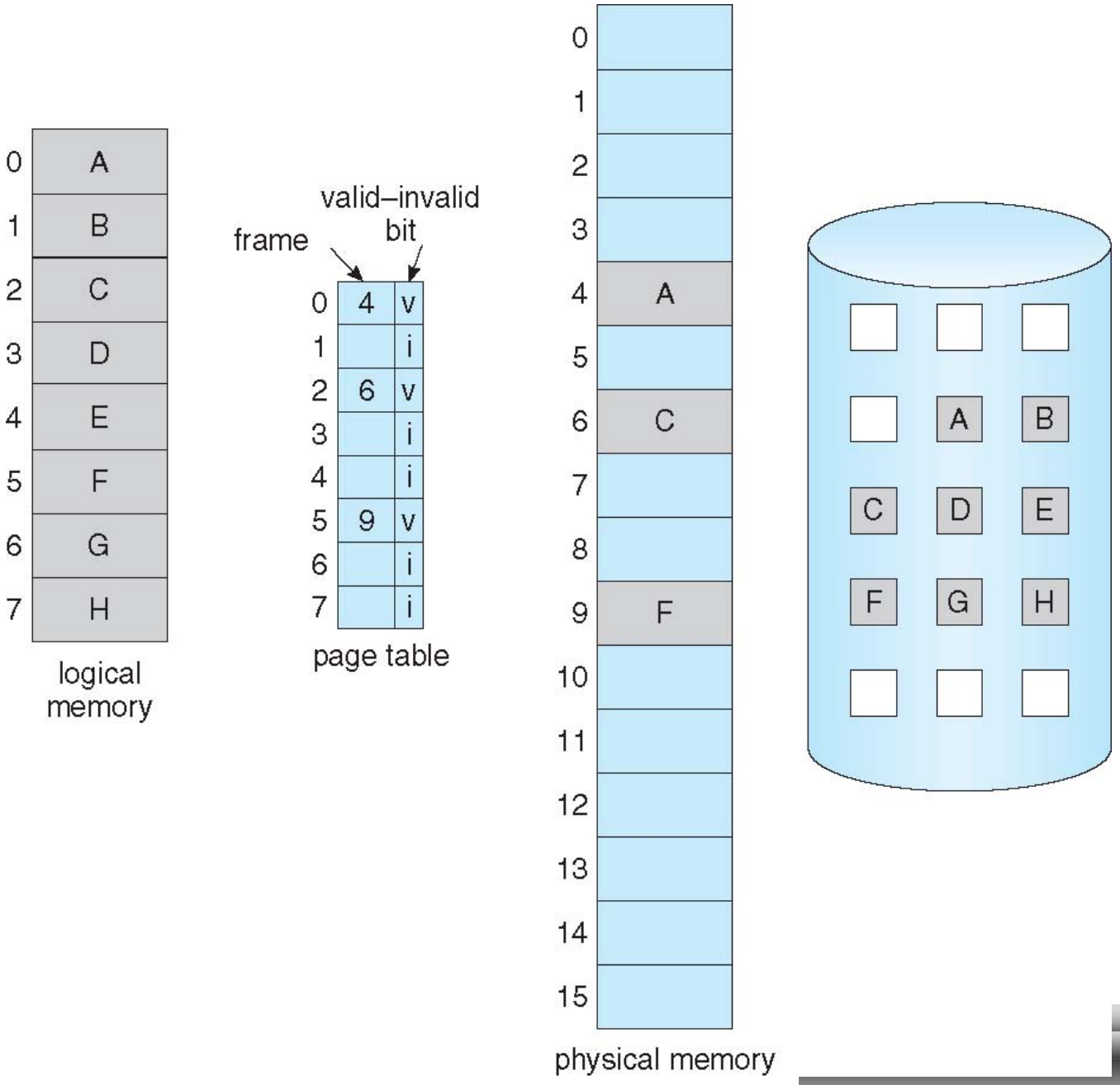


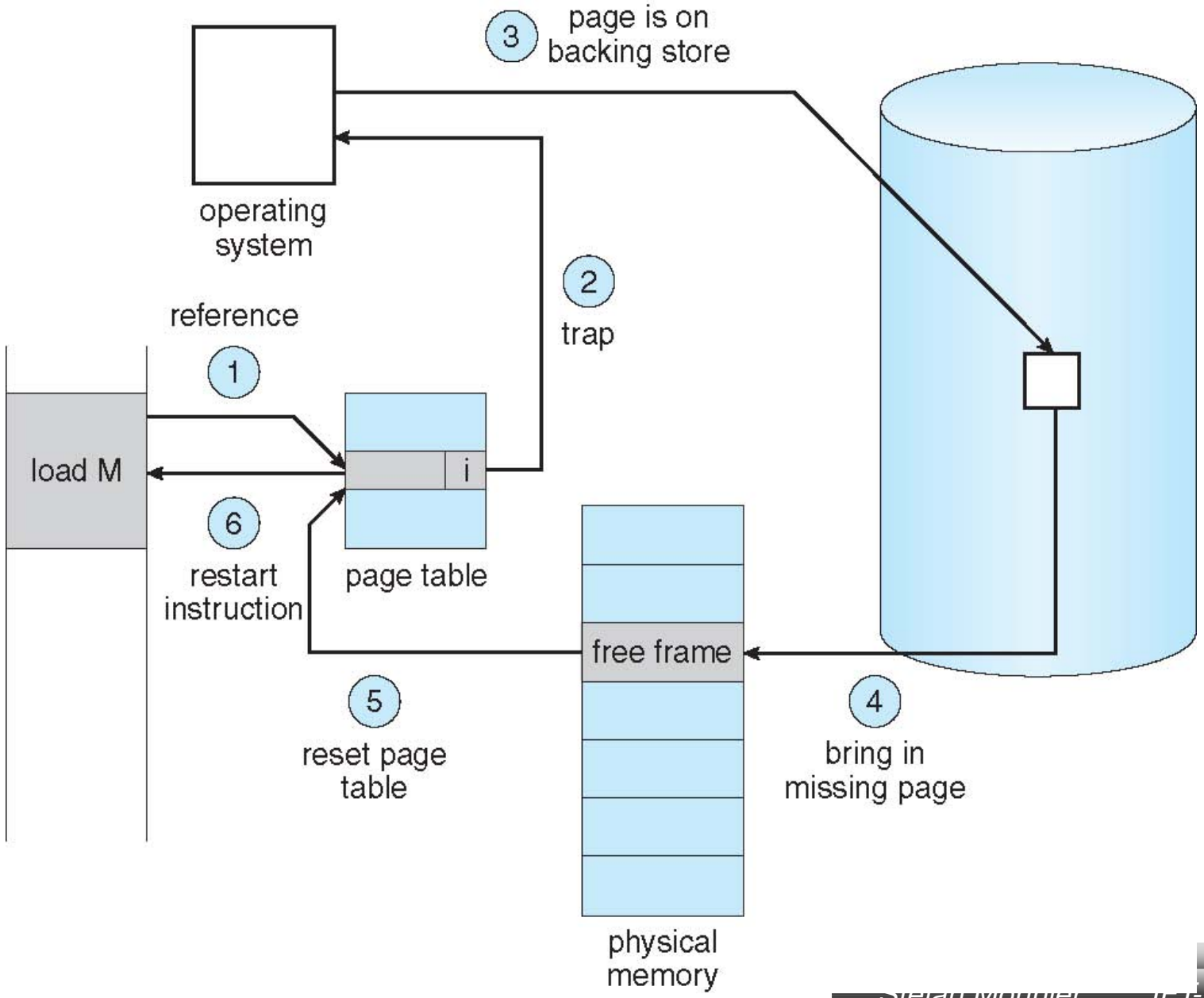
Table de pages avec pages manquantes



Page fault

1. Le CPU utilise table des pages pour trouver l'adresse physique
2. Si la table des pages indique une entrée invalide: *page fault!*
3. Le CPU passe en mode noyau et appelle le SE
4. Le SE cherche dans sa propre table
5. Si l'adresse logique est vraiment invalide: *coredump!*
6. Sinon, trouver une *frame* libre
7. Transférer la page depuis le disque vers cette frame
8. Mettre à jour la table des pages
9. Réexécuter l'instruction du programme

Schéma d'une page fault



Besoins de la pagination sur demande

MMU avec des bits valide/invalide par page

Une mémoire secondaire

La capacité de réexécuter une instruction

- Instructions atomiques
- Garder trace de la partie déjà exécutée

Coût d'un page fault

- *Trap* vers le SE
- Trouver une *frame* libre
- Potentiellement évincer une *frame*
- Envoyer la commande au disque
- *Context switch* à un autre processus en attendant
- *Interruption* vers le SE
- Autre *Context switch* pour revenir au processus
- Corriger la table des pages
- Relancer l'exécution

Temps d'accès effectif

ρ: *miss rate* de la mémoire centrale

M: Temps d'accès à la mémoire centrale

D: Temps de service d'un page fault

$$EAT = (1 - \rho)M + \rho(M + D) = M + \rho D$$

D est principalement déterminé par le temps d'accès au disque

$$D \geq \textit{latence-disque} + \textit{accès-disque}$$

Optimisations

Garder des *frames* libres

Charger plusieurs pages d'un coup

Un accès disque pour 4KB n'est pas plus rapide que pour 64KB

Prefetching

Copy-on-Write (CoW)

Copie paresseuse via la table des pages

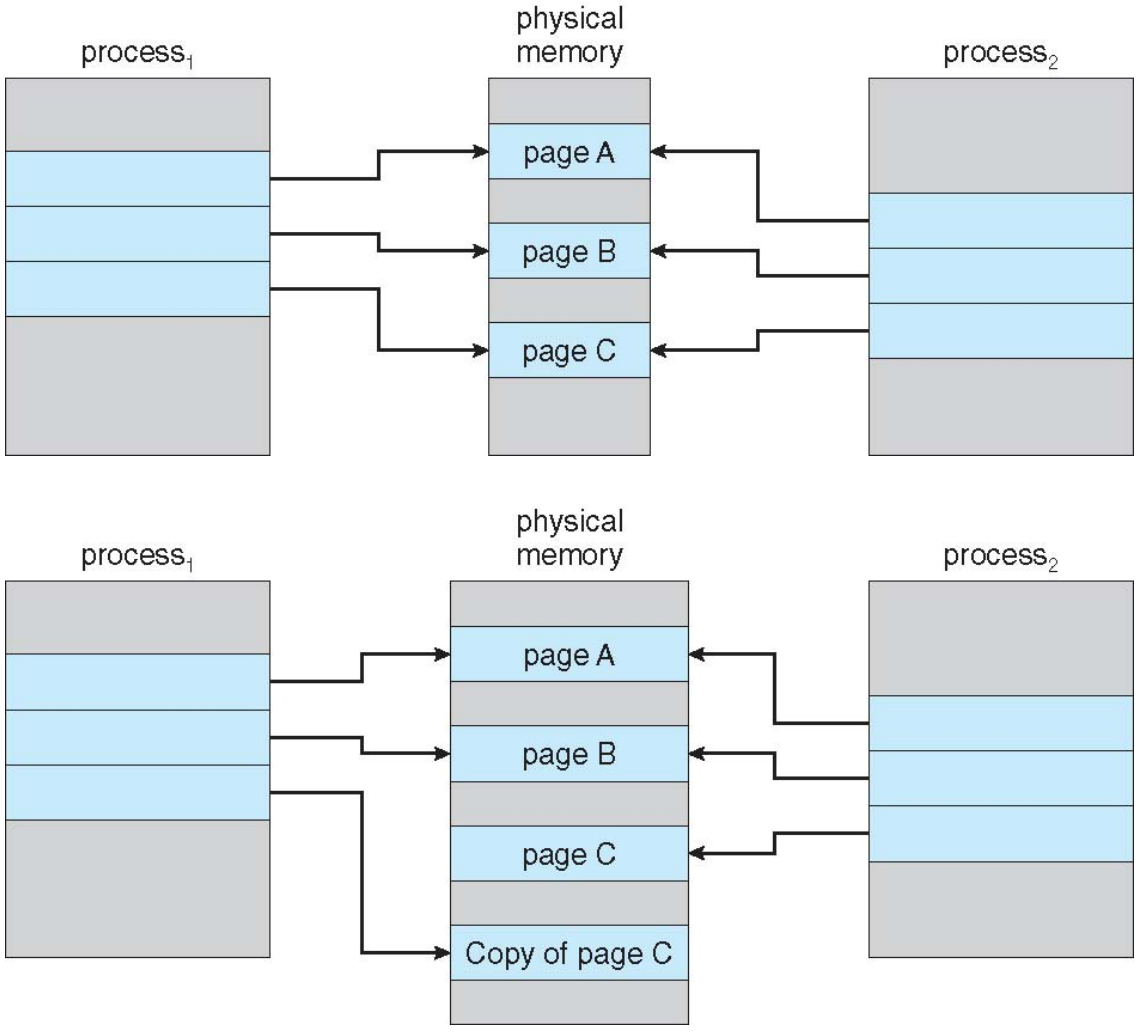
La copie partage la même mémoire physique que l'original

1. Pages marquées *read-only*
2. En cas de *store*, la page affectée est copiée
3. La table des pages est ajustée et marquée *read-write*
4. Et ensuite seulement le *store* est relancé

Indispensable pour `fork`

Utilisé aussi pour l'initialisation à 0

Schéma de Copy-on-Write



Plus de frame *libre*

Que faire s'il n'y a plus de *frame* libre?

- Tuer un ou des processus
- *Swap out* un processus
- Évincer une page

Pour évincer, il est très important de trouver un bon condidat

Remplacement de page

Pour évincer une page il faut la remettre sur le disque

On utilise un *dirty bit* pour savoir si la page a été modifiée

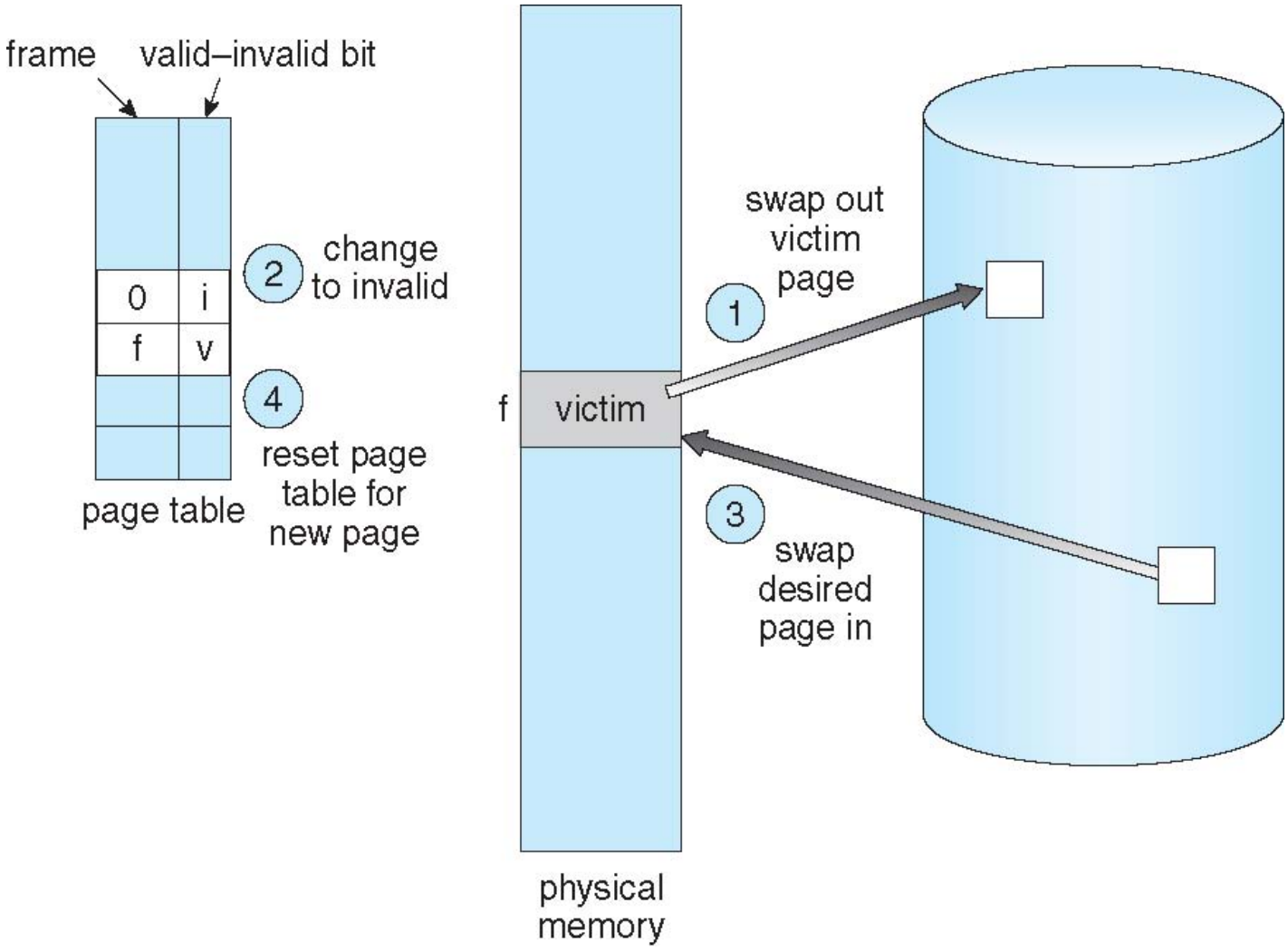
Si elle n'a pas été modifiée, pas besoin de la réécrire sur le disque

Besoin de statistiques sur l'usage de chaque page pour un bon choix

Note: une *page fault* peut maintenant faire 2 transferts de page

Le temps d'accès effectif en est augmenté en conséquence

Esquisse de remplacement de page



Algorithmes de remplacement de page

Minimiser le nombre de *page faults* futures

- Soit globalement
- Soit par processus ou utilisateur ou groupe de processus

Évalué sur une séquence de pages

E.g.: $7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$

Remplacement par Random

On choisit une *frame* au hasard

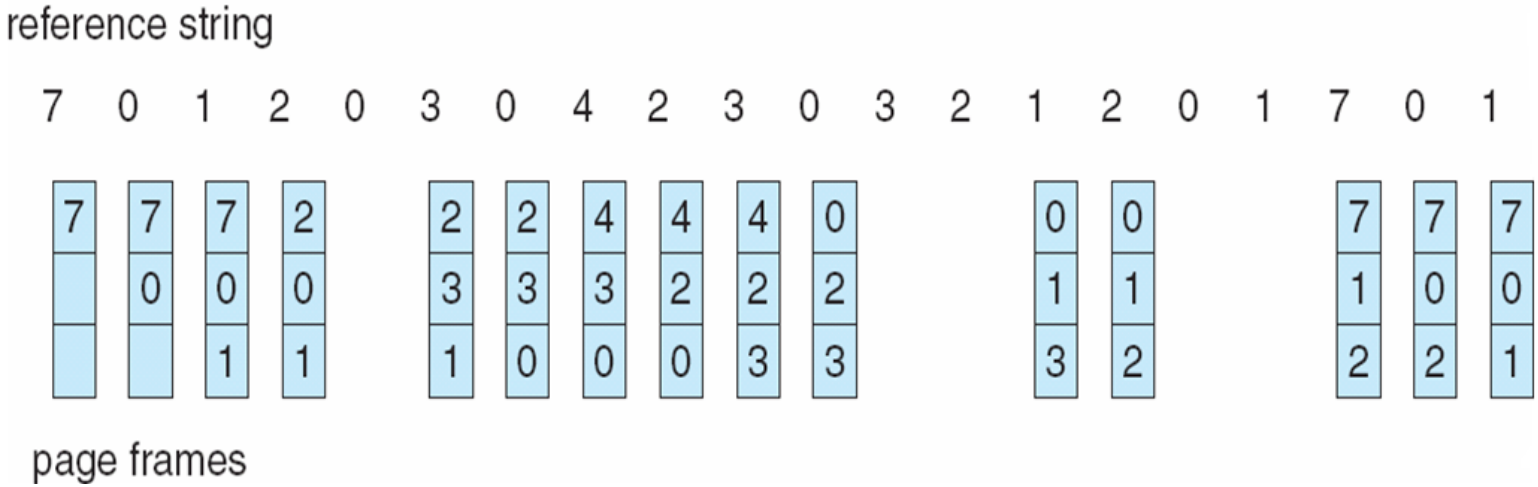
Facile à implémenter

Pas très bon en général

Mais, dans le pire des cas n'est pas pire que les autres

Remplacement par FIFO

On choisit toujours la *frame* suivant la précédente:



Dans cet exemple, on a donc 15 *page faults*

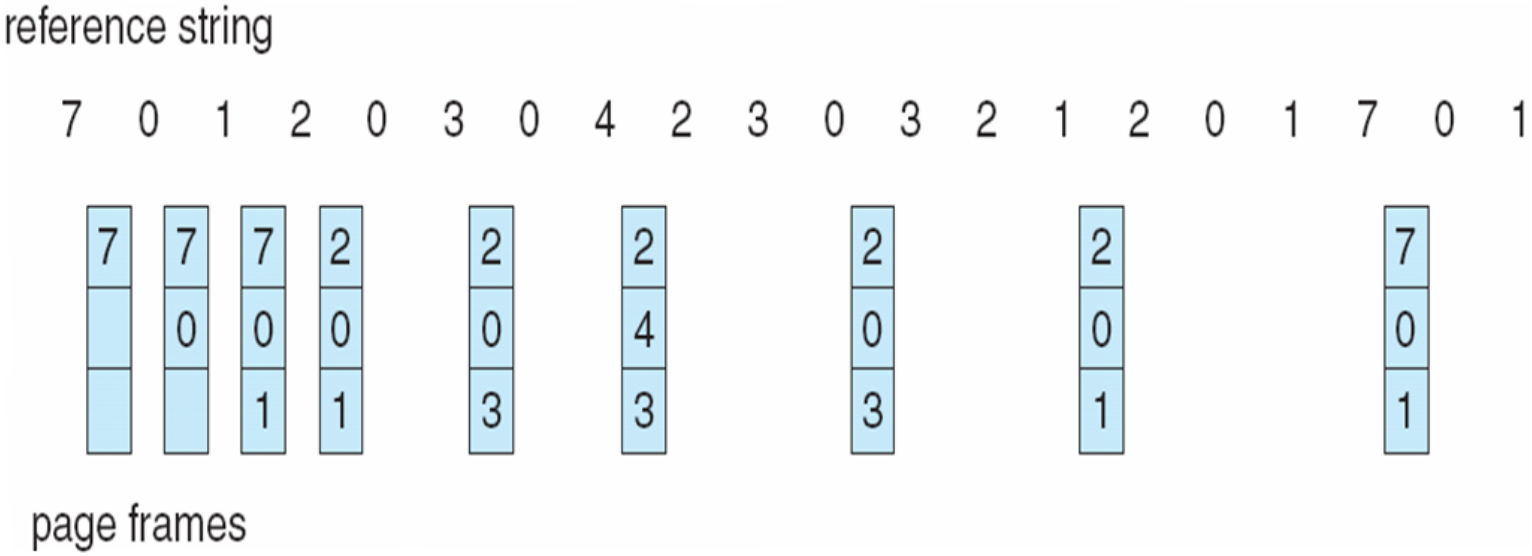
Facile à implémenter

Pas idéal. Mais, dans le pire des cas n'est pas mauvais

Anomalie de Belady: plus de *faults* avec plus de frames!

Remplacement Optimal

Remplacer la page *qui restera inutilisée plus longtemps*



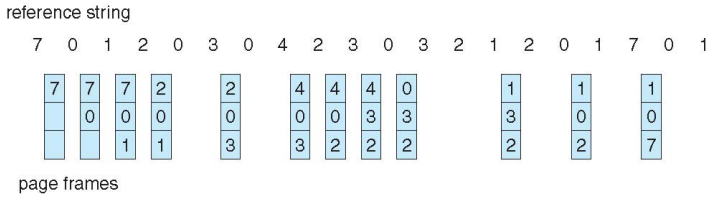
Seulement 9 page faults

Malheureusement, il faut connaître le futur

Bonne référence, cependant: on ne peut pas faire mieux

Remplacement par LRU

Remplace la frame *inutilisée depuis le plus de temps*



12 page faults, dans notre exemple

Généralement un des meilleurs algorithmes

Ne souffre pas de l'anomalie de Belady

LRU est OPT à l'envers $\Rightarrow$ équivalents si le futur reproduit le passé

Talon d'Achille de LRU

Longs accès séquentiels

Sans répétition: le futur ne ressemble pas du tout au passé

Avec répétition: accès à N+1 pages

- Lorsqu'on arrive à N+1, on évince 0
- Lorsqu'on arrive à 0, on évince 1

Random fonctionnerait beaucoup mieux (~1 fault par boucle)

Implémentation de LRU

Horloge:

- Chaque frame garde l'"heure" du dernier accès
- L'horloge est incrémentée à chaque accès
- Lors du remplacement, cherche la frame avec la plus vieille "heure"

Liste (généralement appelée pile):

- Garde une liste ordonnée de toutes les frames
- À chaque accès, déplace la frame en première position
- Lors du remplacement, prend la dernière frame de la liste

Approximation de LRU: referenced *bit*

LRU requiert trop de travail à chaque accès

Certains CPU donnent une approximation

Un bit *referenced* dans la table de page

- Lors de l'accès à une page, le CPU met ce bit à 1
- Périodiquement, le SE collecte tous ces bits et les remet à 0

Le SE ne peut plus distinguer l'ordre d'accès dans une même période

Parmi les frames d'une même période on peut utiliser FIFO

Approximation de LRU: Clock

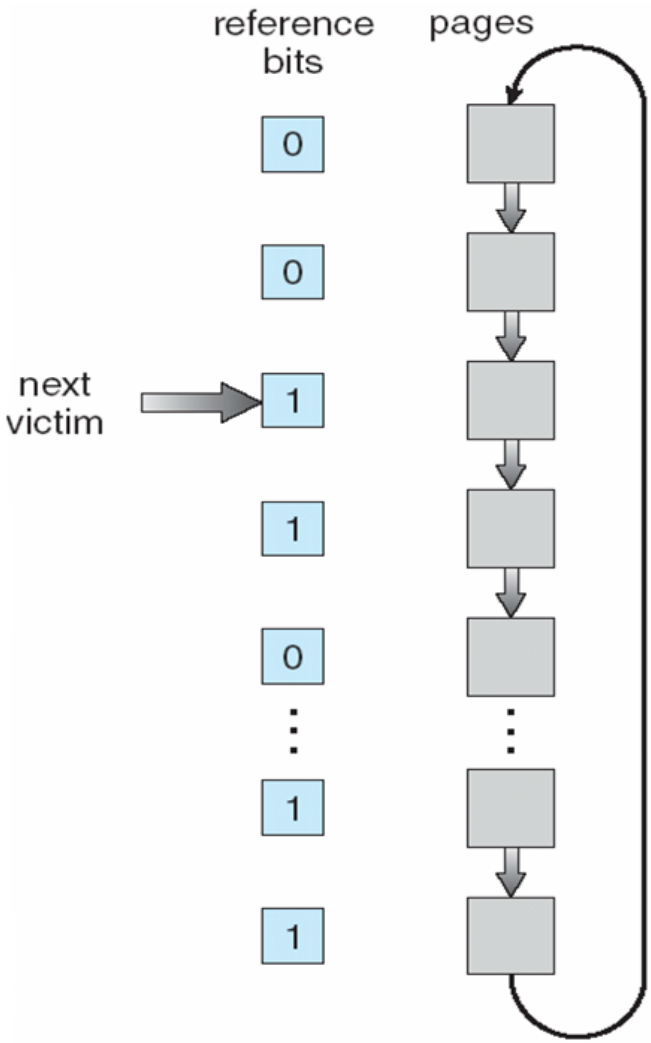
Aussi appelé algorithme de la *deuxième chance*

Comme FIFO, prend la frame suivante, mais vérifie son *referenced* bit:

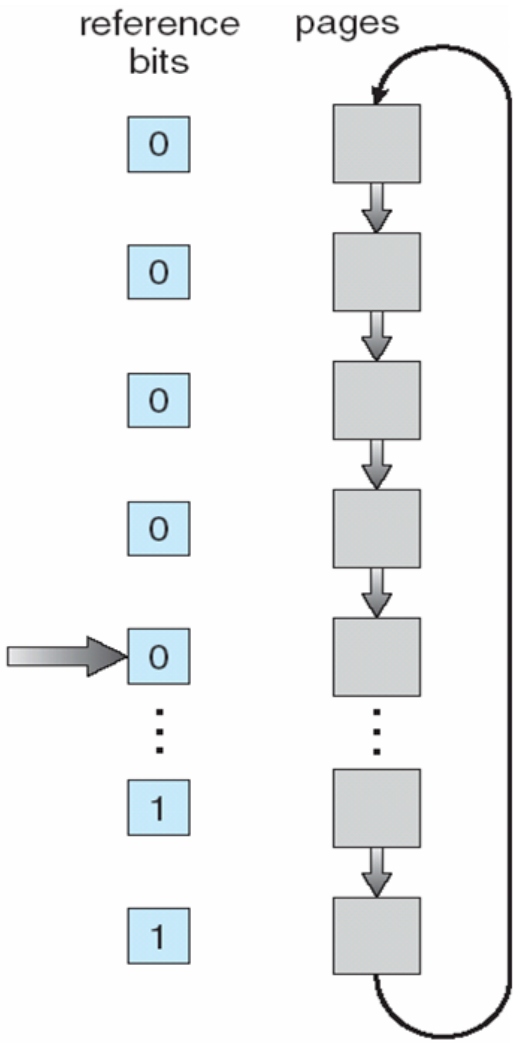
- Si *referenced* est 0, alors remplace cette page
- Si *referenced* est 1, alors remet *referenced* à 0 et passe à la frame suivante

Dégénère à FIFO si les *referenced* sont tous à 0 (ou tous à 1)

Exemple de Clock



(a)



(b)

Implémentation de referenced *bit*

Certains CPU implémentent les bits *referenced* et *dirty*

Sinon, on peut le faire à la main:

- Pour mettre *referenced* à 0, marquer la page comme invalide
- Lors d'un page fault, mettre *referenced* à 1

De même pour *dirty*

- Pour mettre *dirty* à 0, marquer la page comme *read-only*
- Lors d'un page fault en écriture, mettre *dirty* à 1

Parfois plus efficace et pratique de le faire à la main

Algorithmes de page-buffering

Découpler les opérations pour répondre plus vite à un page-fault

Garder des frames libres

- Choisir à l'avance les victimes futures
- Si elle est *dirty*, écrire le contenu dans le disque
- Marquer invalide. Mais si accédée, on peut la ressusciter à bon prix

Ne pas laisser les pages *dirty* trop longtemps

- Quand le disque est inactif, écrire le contenu des pages *dirty*
- Ainsi la page peut être évincée plus rapidement

Garder des frames pré-initialisées à 0

Allocation de frames

Combien de frames allouer à chaque processus

Évincer une page qui appartient à un autre processus?

Allocation *globale*: oui sans équivoque

- Temps d'exécution d'un processus dépend beaucoup des autres
- Populaire: généralement meilleur *throughput*

Allocation *locale*: non, (voire: oui mais)

- Performance d'un processus plus prévisible
- Risque de sous-utiliser la mémoire
- Populaire quand on *partitionne* une machine

NUMA (Non-Uniform Memory Access)

Le temps d'accès à la mémoire peut dépendre de la distance

E.g. plusieurs nœuds (CPU+mémoire) interconnectés

Meilleure performance avec allocation *proche*

Affecte l'ordonnanceur, bien sûr

Lors du remplacement de page, choisir une frame *proche*

Thrashing

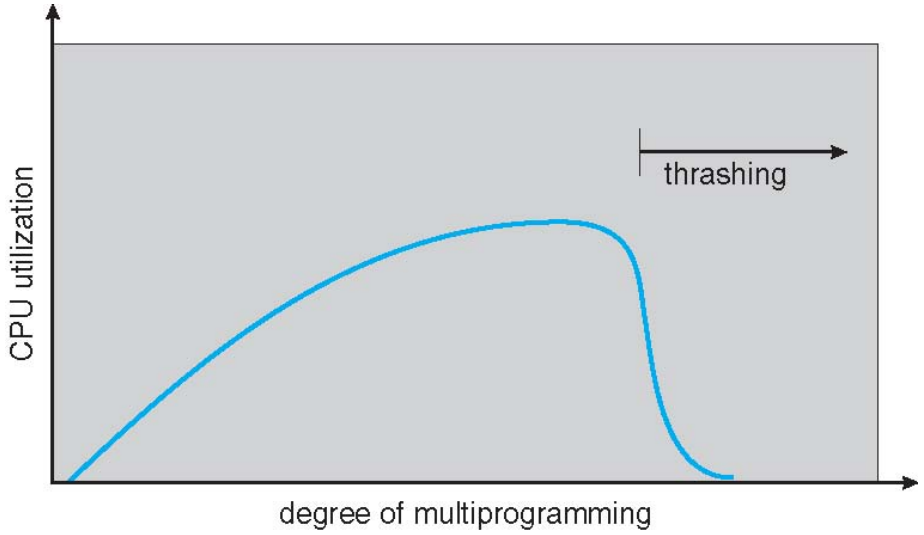
S'il y a trop peu de mémoire physique

Chaque page-fault évince une page dont aura bientôt besoin

CPU sous-utilisé

Sous utilisation du CPU peut inciter à lancer plus de programmes!

Le système peut paraître figé pendant plusieurs minutes



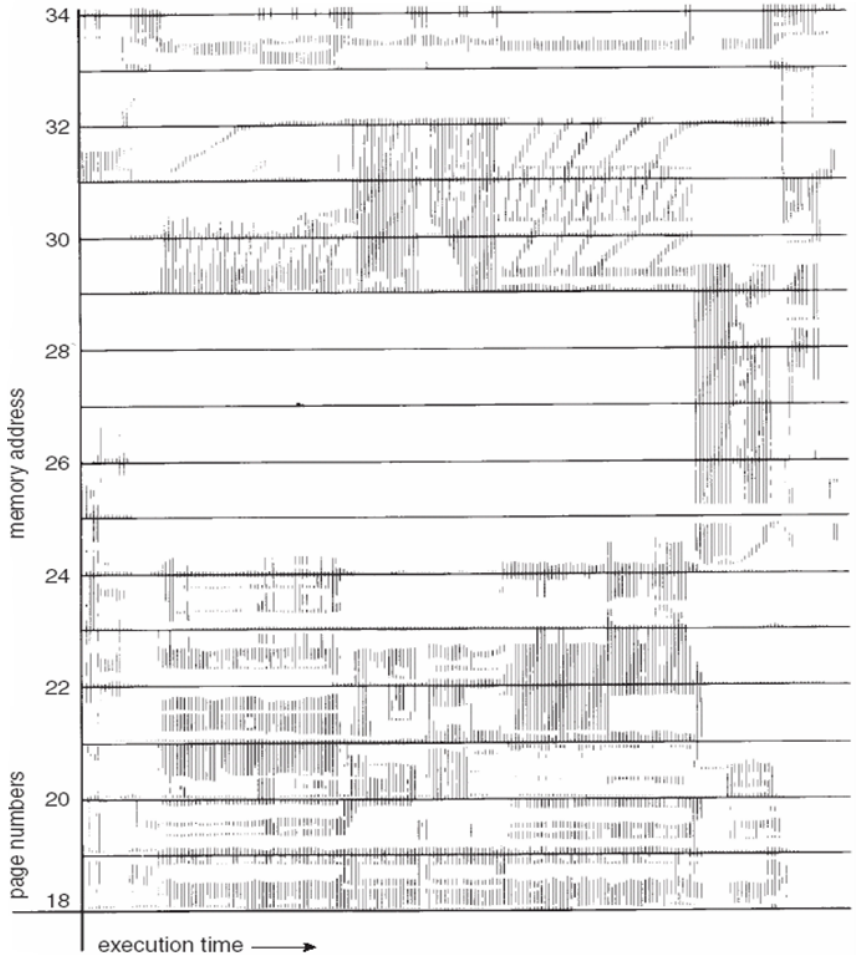
Pagination et thrashing

Pagination marche parce que:

- Exécution migre de “localité” à “localité”
- Localités peuvent se chevaucher
- On peut revenir à une localité

Thrashing

- Taille de localité > mémoire
- Peut se limiter aux processus coupables



Modèle du working-set

Δ = fenêtre du *working-set*: un certain intervalle de temps

- Un Δ trop petit ne couvre pas toute la localité
- Un Δ trop grand sera pessimiste
- Δ dépend typiquement du temps d'accès au disque

Peut se mesurer en instructions ou en accès mémoire

WSS_i = Nombre de pages référencées par P_i pendant Δ

$D = \Sigma WSS_i$: Taille totale des localités courantes

$D >$ *Mémoire* $\Rightarrow$ thrashing: il faut suspendre des processus

Fréquence de page-fault

Autre approche, plus directe, basée sur un *remplacement local*

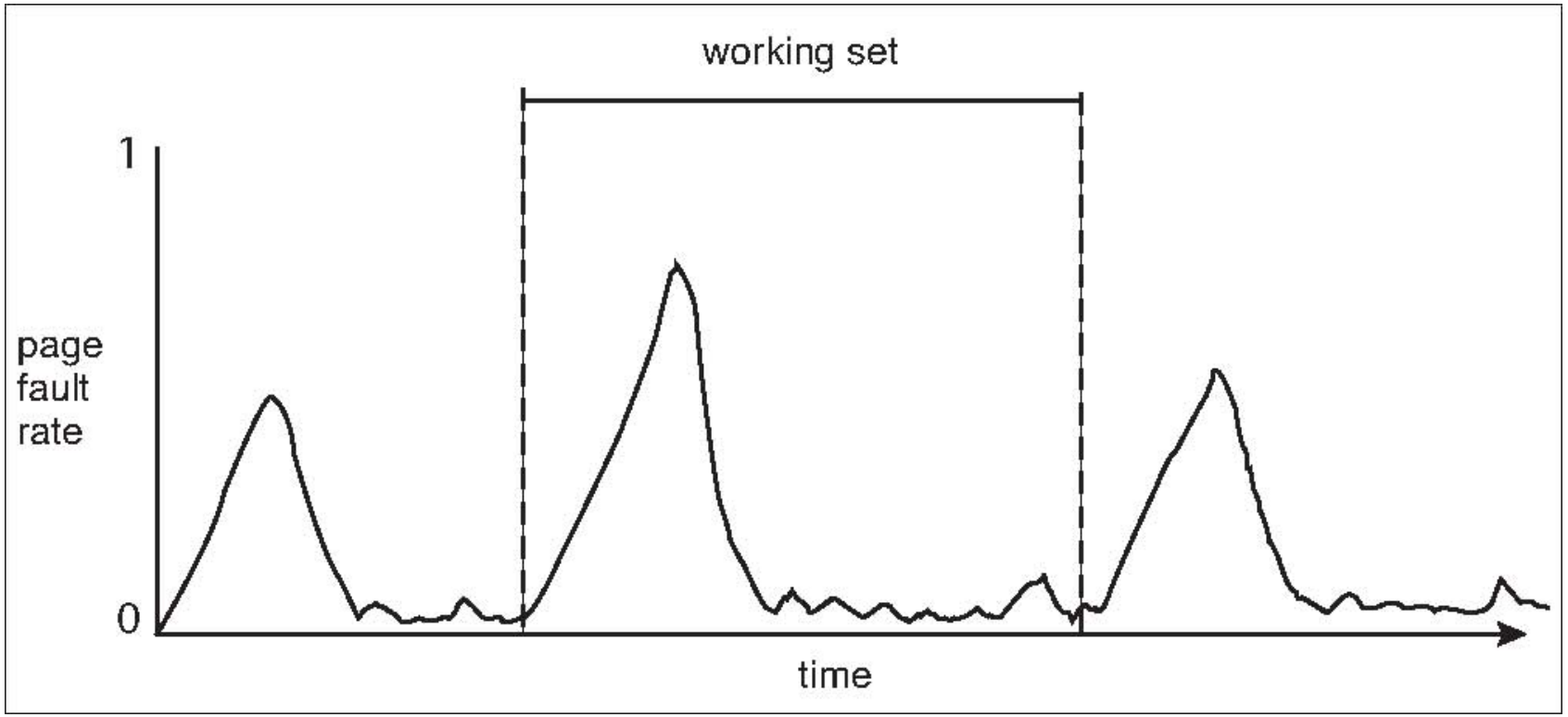
Décide d'une fréquence acceptable de page-faults

Mesure fréquence de page faults pour chaque processus

- Fréquence trop basse: diminuer la mémoire allouée au processus
- Fréquence trop élevée: augmenter la mémoire allouée

Se fréquence trop élevée, mais plus de mémoire disponible: suspendre

Working-set et fréquence de page-faults



Fichiers memory-mapped

Rendre un fichier accessible comme une partie de la mémoire

Réutilise la pagination sur demande

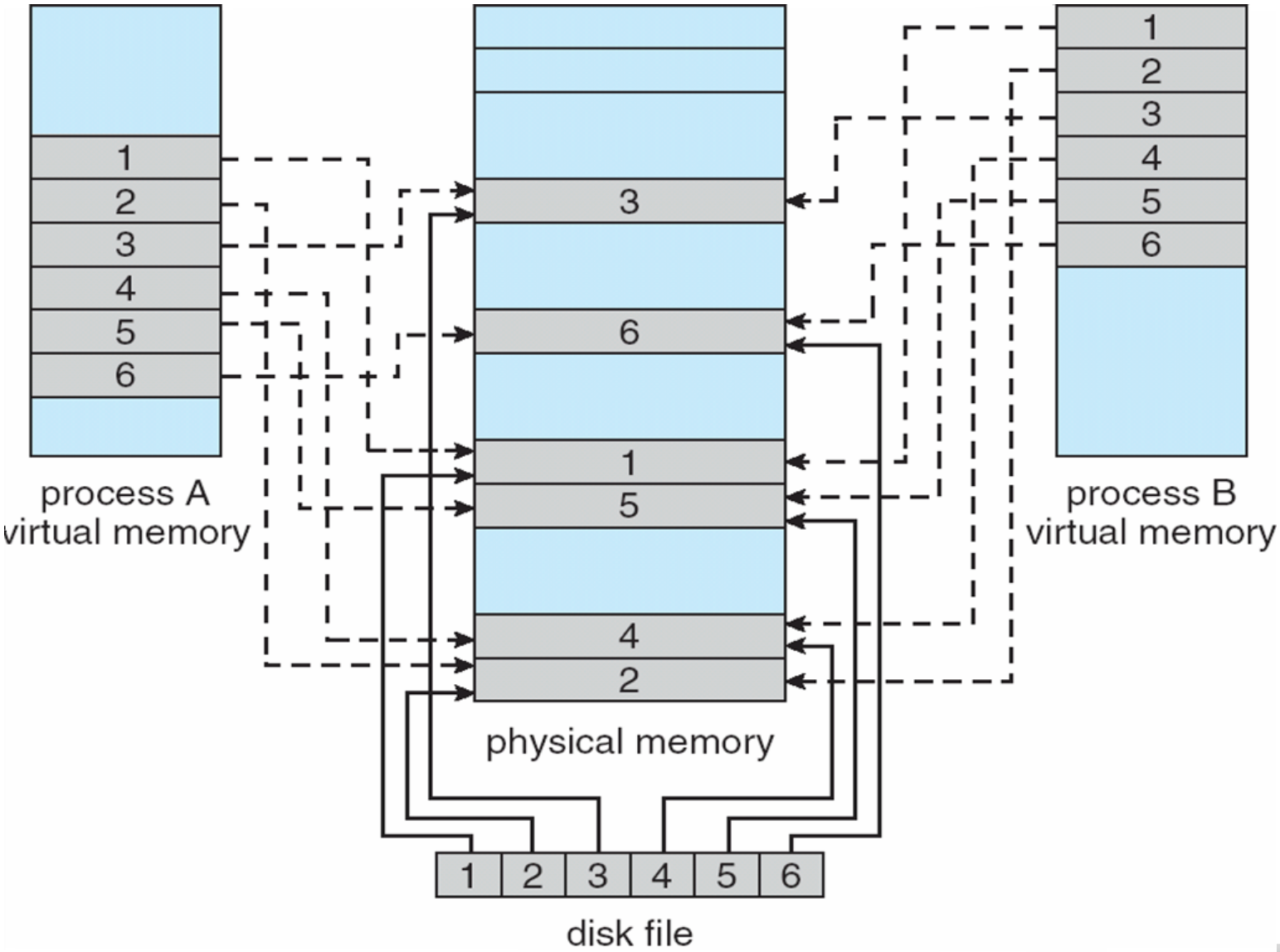
Modifications écrites sur le disque par le code de pagination

Partage efficace d'un fichier via `mmap` entre processus

Peut accélérer exécution en évitant appels systèmes `read` et `write`

Moins de contrôle sur les accès disque; pas d'accès asynchrone

Exemple de memory mapping



Allocation de mémoire noyau

La mémoire du noyau est traitée différemment

Mémoire des processus gérée par page

Mémoire du noyau gérée par objet, comme avec `malloc`

Certaines adresses sont spéciales

Certaines parties doivent être contiguës

Allocation style buddy

Utilise une zone mémoire contiguë de taille fixe

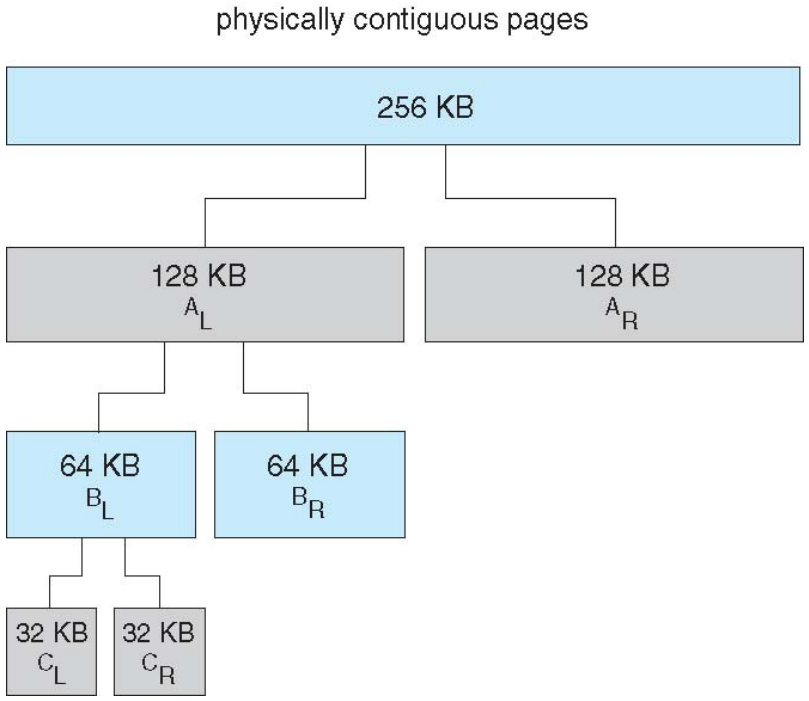
Cette zone est divisée successivement par 2

Chaque *chunk* occupe un espace de taille 2^n

objet $< {}^1/_2$ *chunk* $\Rightarrow$ divise

Avantage: facile de re-réunir des petits blocs (*coalesce*)

Inconvénient: fragmentation, interne et externe



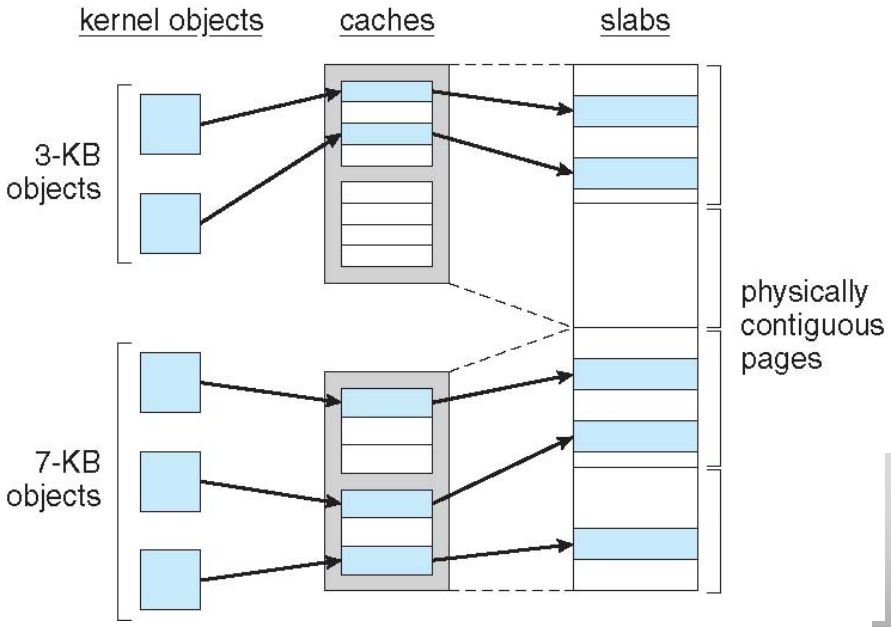
Allocation style Slab

Slab: une ou plusieurs pages, physiquement contiguës

Cache: ensemble de *slab*s

Chaque *cache* dédié à un type (ou une taille) d'objets

Cache grandi par slab, divisée en objets (initialement libres)



Utilise le fait que les tailles d'objets ne sont pas aléatoires

Peu de fragmentation, en pratique

Efficace, pas besoin de *coalesce*

Taille des pages

La taille des pages est parfois imposée, mais pas toujours

Facteurs de choix:

- Fragmentation: mieux vaut des petites pages
- Taille de la table: mieux vaut de grosses pages
- Efficacité du TLB: mieux vaut de grosses pages
- Localité: mieux vaut des petites pages
- Résolution: mieux vaut des petites pages
- Coût I/O et page-faults: mieux vaut de grosses pages

En général, les tailles devraient augmenter peu à peu

Prépagination

Charger certaines pages d'un processus à l'avance

- Pour éviter page-faults au démarrage
- Gaspillage si ces pages ne sont pas utilisées
- Gaspillage d'I/O peut être mineur (accès séquentiel)
- Minimiser le gaspillage de mémoire: candidats à l'éviction

Clustering: lors d'un page fault, amener tout un *cluster*

- Forme de prefetching
- Simule des pages plus grandes, sans les inconvénients

TLB reach

TLB reach: quantité de mémoire accessible depuis le TLB

TLB reach = taille TLB $\times$ *taille de page*

Si trop petit, beaucoup de TLB-misses

Utiliser des *superpages* là où c'est possibles

Gérer la localité

```
int data[128][128];
```

Quel accès choisir:

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i, j] = 0;
```

ou:

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i, j] = 0;
```

Beaucoup de manières d'influencer la performance d'un programme