

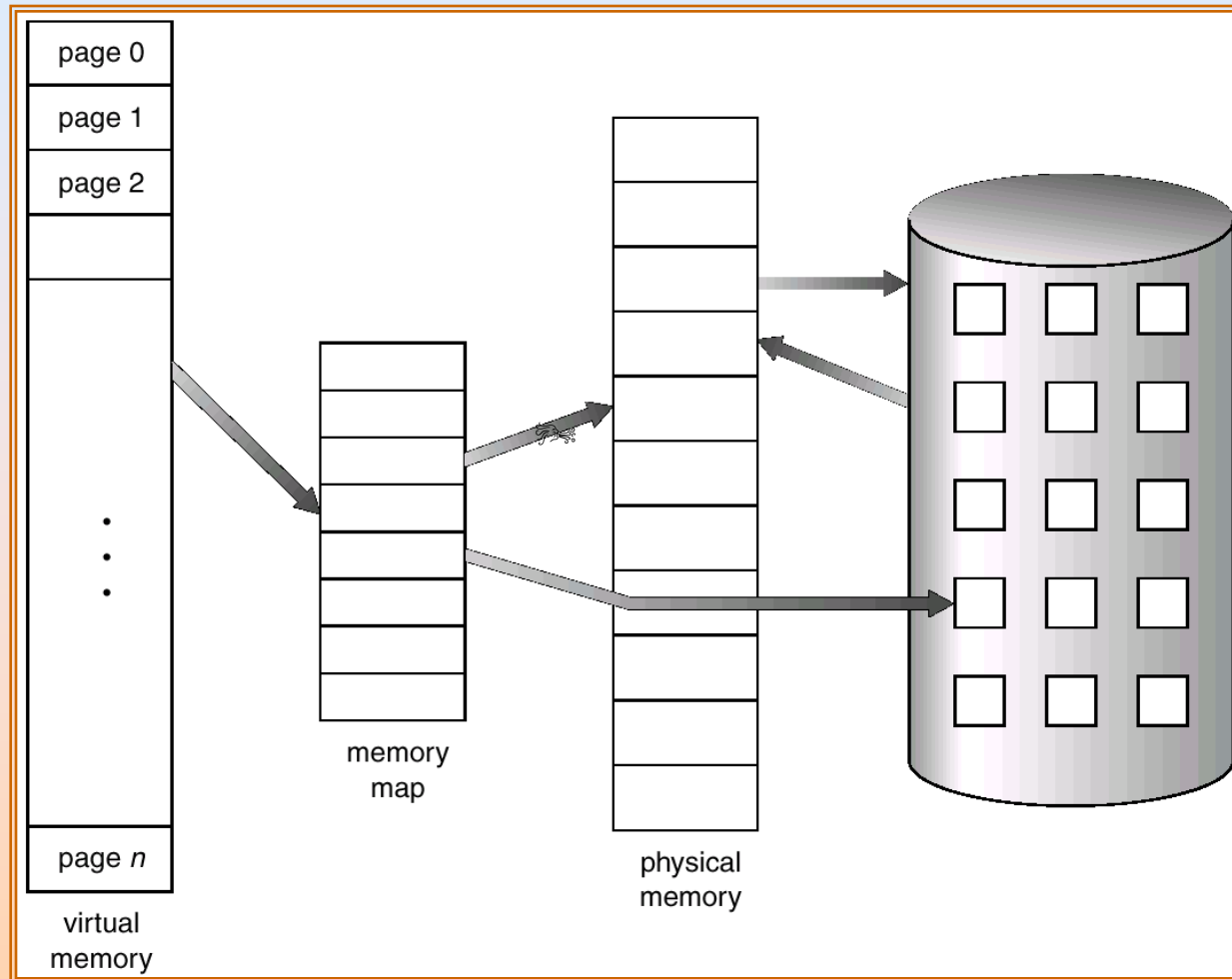
# Mémoire Virtuelle

- Bases
- Pagination à la Demande
- Création de Processus
- Remplacement de Pages
- Allocation de Cadres de page
- Ecroutement
- Segmentation à la Demande
- Exemples OSs

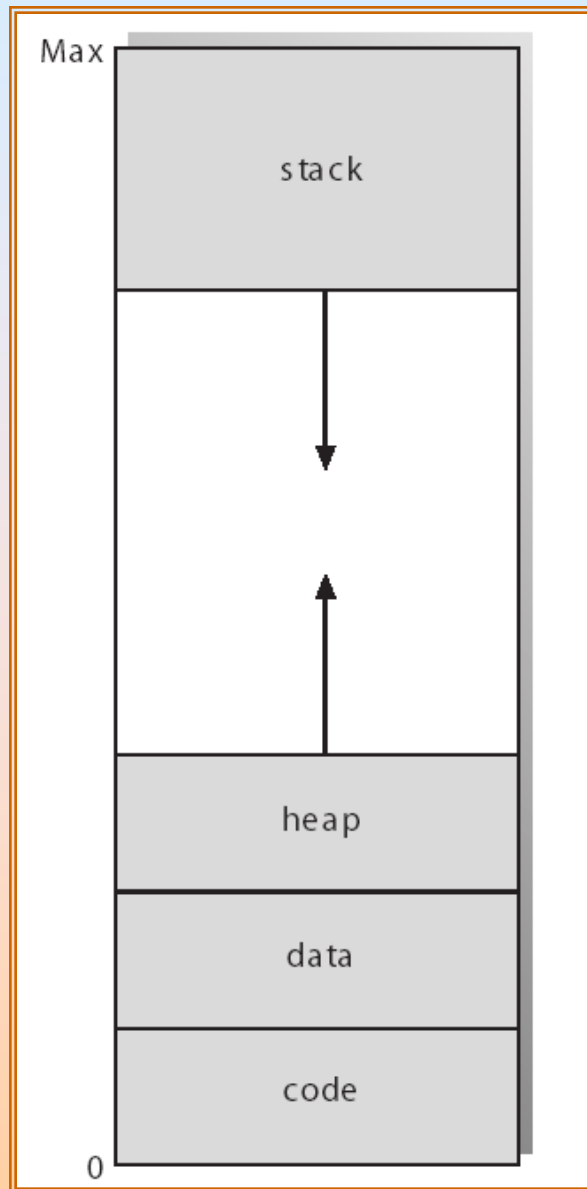
# Bases

- **Mémoire Virtuelle** – séparation de la mémoire utilisateur (logique) et de la mémoire réelle (physique).
  - Seulement une partie du programme a besoin d'être en mémoire pour l'exécution.
  - L'espace d'adressage logique peut ainsi être plus large que l'espace mémoire physique.
  - Permet le partage des espaces d'adressage par plusieurs processus.
  - Permet plus d'efficacité pour la création de processus (due au partage d'espace d'adressage).
- La mémoire virtuelle peut être implémentée via:
  - Demande de pages
  - Demande de segments

# Mémoire Virtuelle > Mémoire Physique



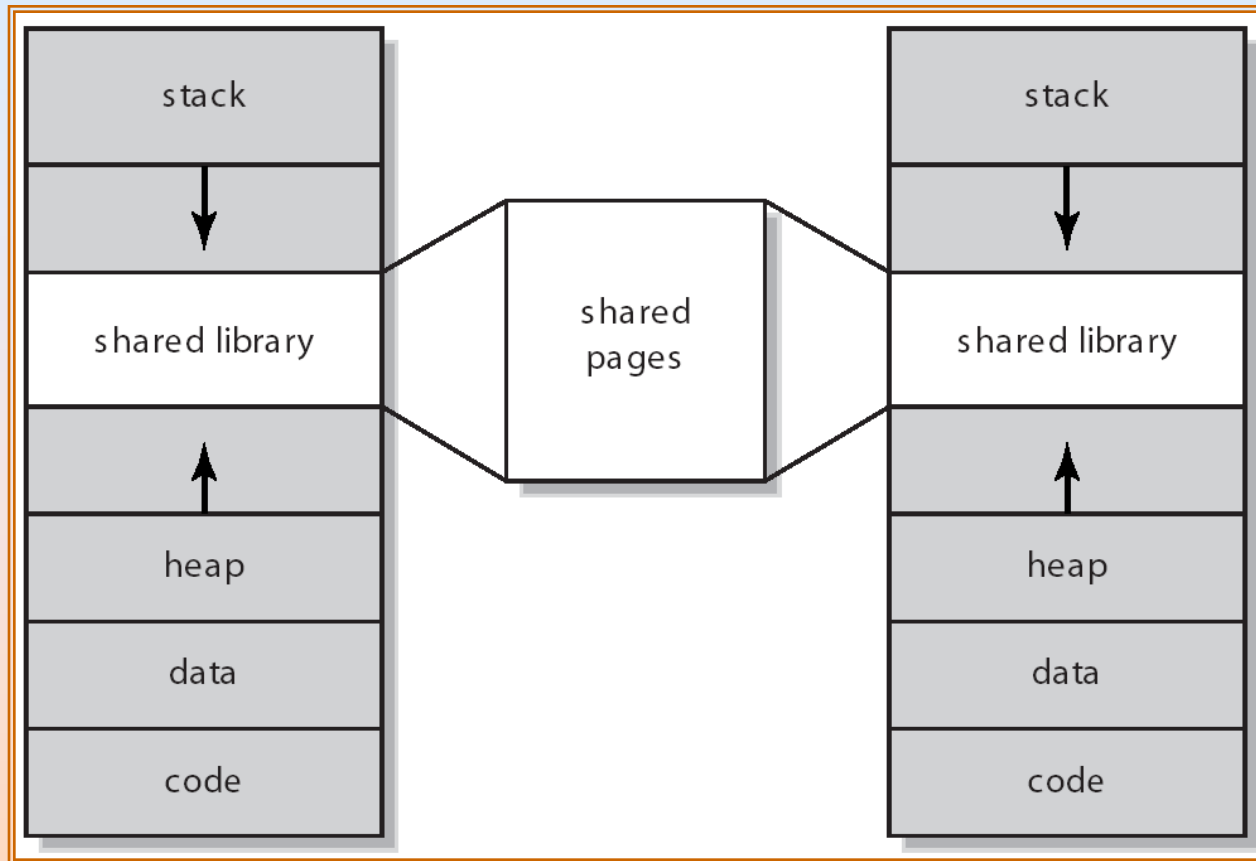
# Espace d'Adressage Virtuel



# Utilité De La Mémoire Virtuelle

- Elle permet facilement le partage de mémoire entre processus

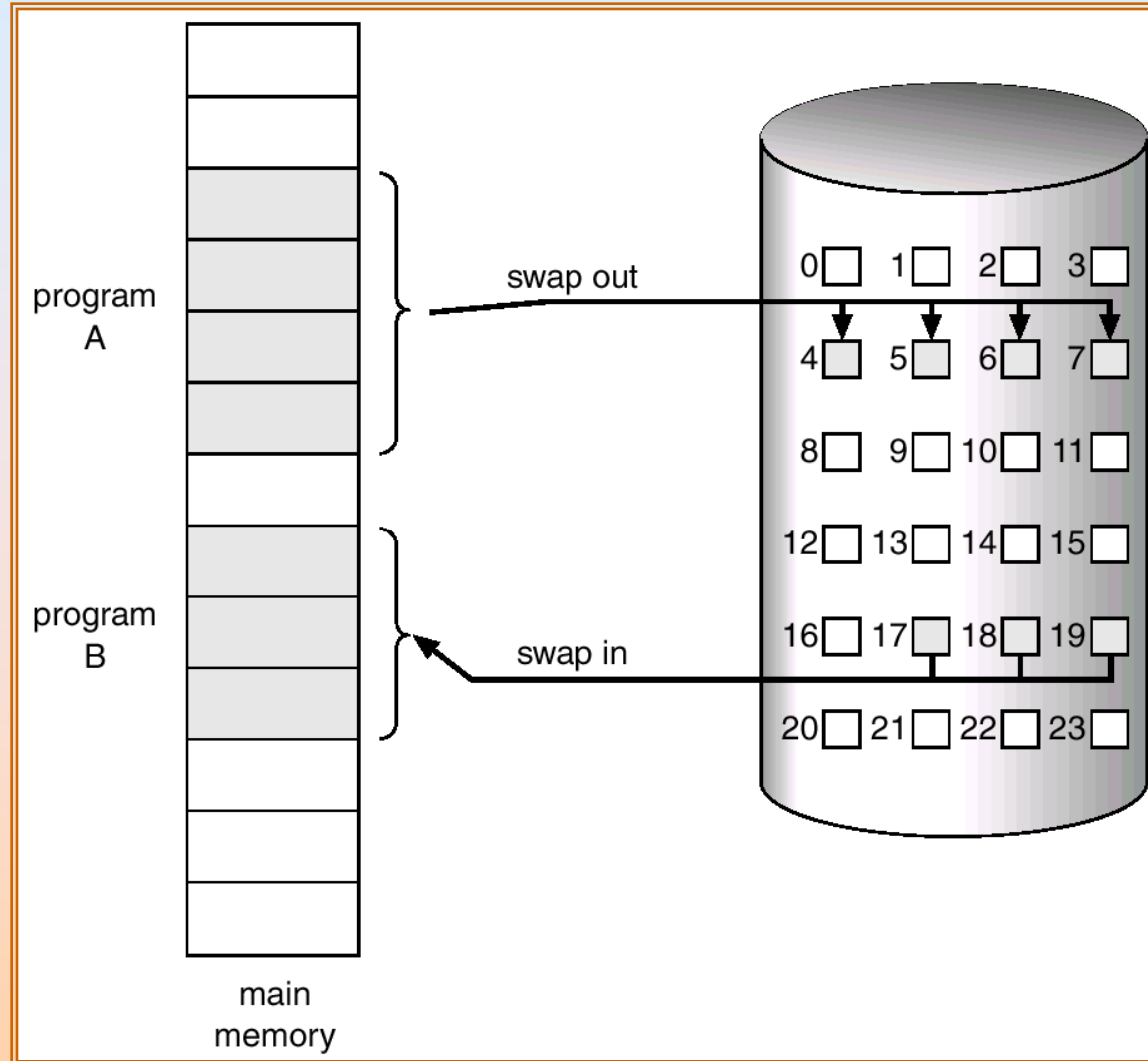
# Bibliothèque Partagée Avec La Mémoire Virtuelle



# Pagination à la Demande

- Importer une page en mémoire SEULEMENT au moment où *elle est demandée*
  - Moins besoin d'I/O
  - Moins besoin mémoire
  - Réponse plus rapide
  - Plus d'utilisateurs
  
- Page demandée ➦ référence à cette page
  - Référence invalide ➦ abort
  - Pas en mémoire ➦ l'importer en mémoire

# Transfert D'une Mémoire Paginée Sur Un Espace Disque Contigu





# Bit Valide-Invalide

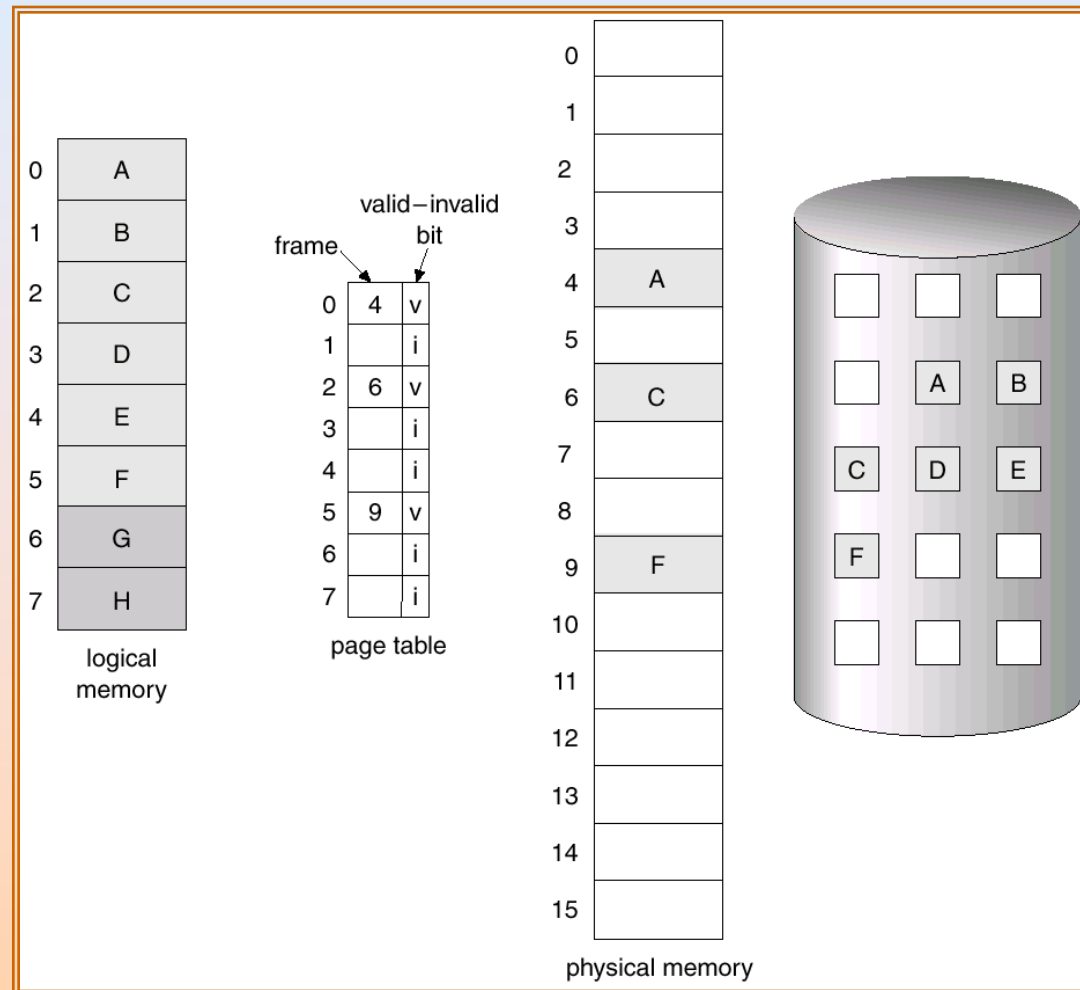
- A chaque entrée dans la table des pages, un bit valide–invalide lui est associé  
(1 🐉 en-mémoire, 0 🐉 pas-en-mémoire)
- Initialement, ce bit est mis à 0 sur toutes les entrées qui ne sont pas en mémoire
- Snapshot d'une table des pages:

Cadre de page #	valide-invalide
	1
	1
	1
	1
	0
⋮	
	0
	0



Table des pages

- Durant la translation d'adresses, si le bit valide–invalide sur une entrée de la table des pages est à 0 🐉 défaut de page

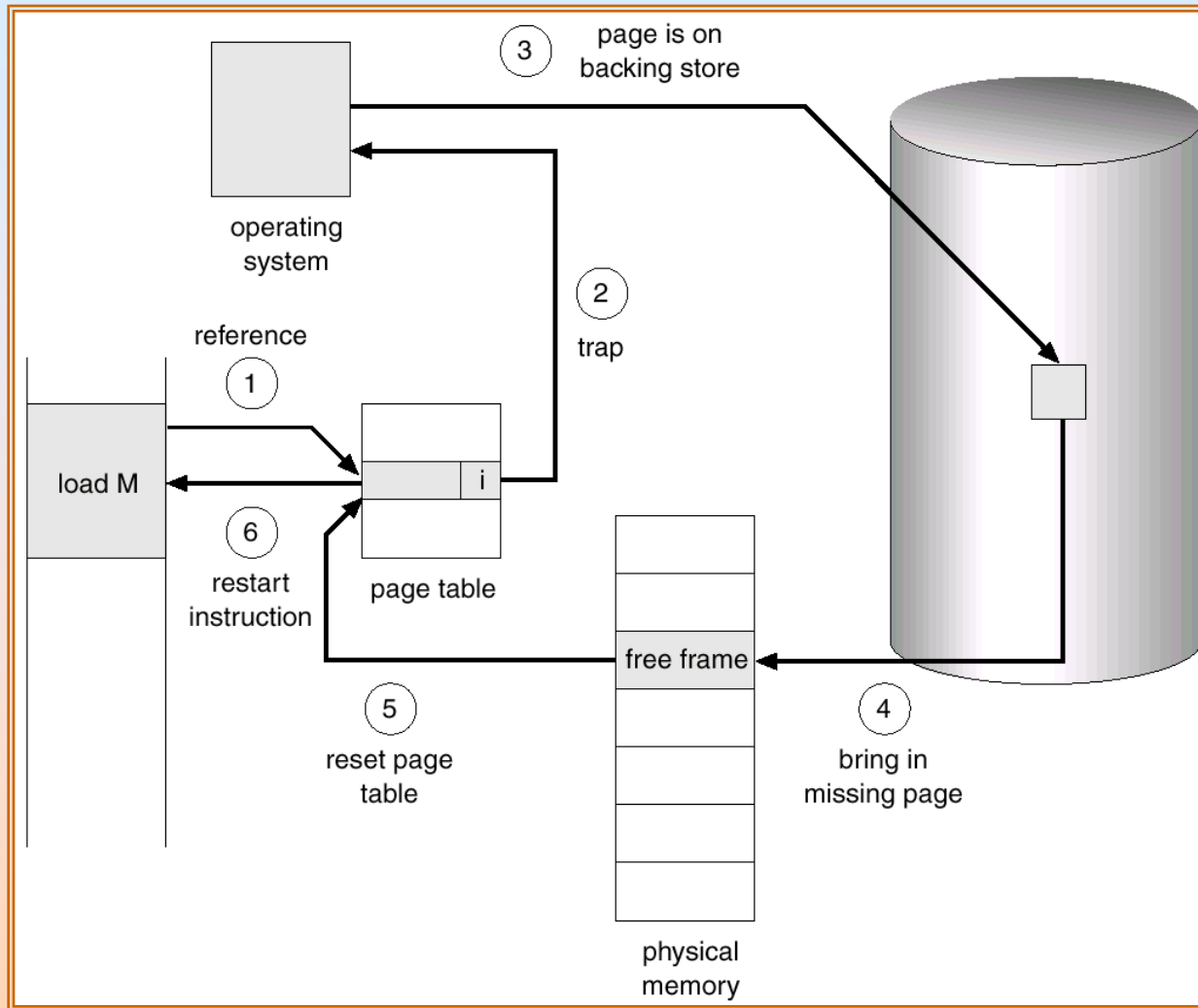
# Table des Pages avec Quelques Pages sur Disque



# Faute De Page

- A la première référence à une page  défaut de page puisque bit valide-invalidé à 0 !
- OS regarde dans une table interne (normalement dans le PCB) pour décider:
  - Référence interdite (pas dans l'espace d'adressage logique du processus)  abort.
  - Sinon, pas en mémoire.
- Trouver une page physique P vide.
- Swapper la page en mémoire dans P.
- Mettre à jour la table, bit de validation = 1.
- Relancer l'instruction

# Traîtement D'une Defaut De Page



# Et S'il N'y A Plus de Cadres de page Vides?

- Remplacement de Pages – trouver une page en mémoire, “pas utilisée”, et la swapper
  - algorithme
  - performance – on voudrait un algorithme qui génère le minimum de défauts de pages
- La même page peut revenir en mémoire plusieurs fois

# Performance : Pagination à la Demande

## ■ Probabilité d'un défaut de page 0 $\leq p \leq 1.0$

- Si  $p = 0$ , pas de défauts de pages
- Si  $p = 1$ , chaque référence est un défaut

## ■ Temps d'Accès Effectif (TAE)

$$\begin{aligned} \text{TAE} = & (1 - p) \times \text{accès mémoire} \\ & + p \times (\text{temps de défaut de page} = \\ & \quad [\text{swap page sur disque}] \\ & \quad + \text{swap page en mémoire} \\ & \quad + \text{relancer instruction}) \end{aligned}$$

# Exemple de Pagination à la Demande

- Temps d'accès mémoire = 1 microseconde
- 50% du temps, la page remplacée a été modifiée => elle sera swappée sur disque
- Temps de Swap = 10 msec = 10,000 microsec

$$\begin{aligned} \text{TEA} &= (1 - p) \times 1 + p (15000) \\ &\sim 1 + 15000p \quad (\text{in msec}) \end{aligned}$$

# Création de Processus

- La mémoire virtuelle permet d'autres bénéfices durant la création de processus:
  - Copy-on-Write
  - Fichiers mappés en mémoire (après)



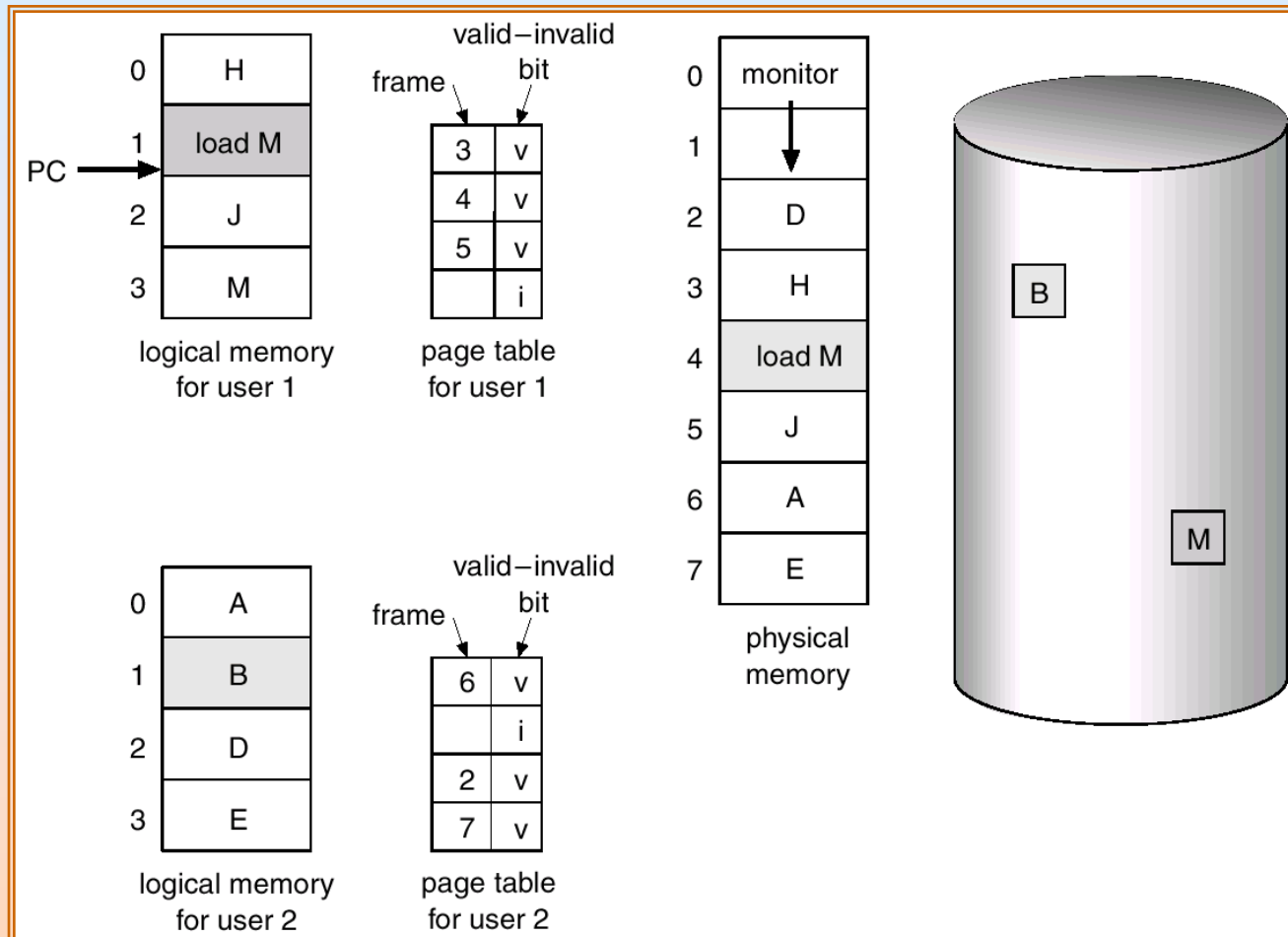
# Copy-on-Write

- Copy-on-Write (COW) permet aux processus père et fil de partager les mêmes pages en mémoire
- Si un des processus modifie une page partagée, seulement cette dernière est copiée
- COW permet une meilleure efficacité à la création de processus puisque seulement les pages modifiées sont copiées
- Les pages libres sont allouées dans un **pool** de pages

# Remplacement de Pages

- Prévenir la sur-allocation de la mémoire en modifiant la routine de service des défauts de pages; on lui ajoute une fonction de remplacement de pages
- Utiliser **bit de modification (dirty bit)** pour réduire l'overhead des transferts de pages – seulement les pages modifiées sont copiées sur disque
- Le remplacement de pages complète la séparation entre la mémoire logique et le mémoire physique – une mémoire virtuelle large peut être fournie sur une petite mémoire physique

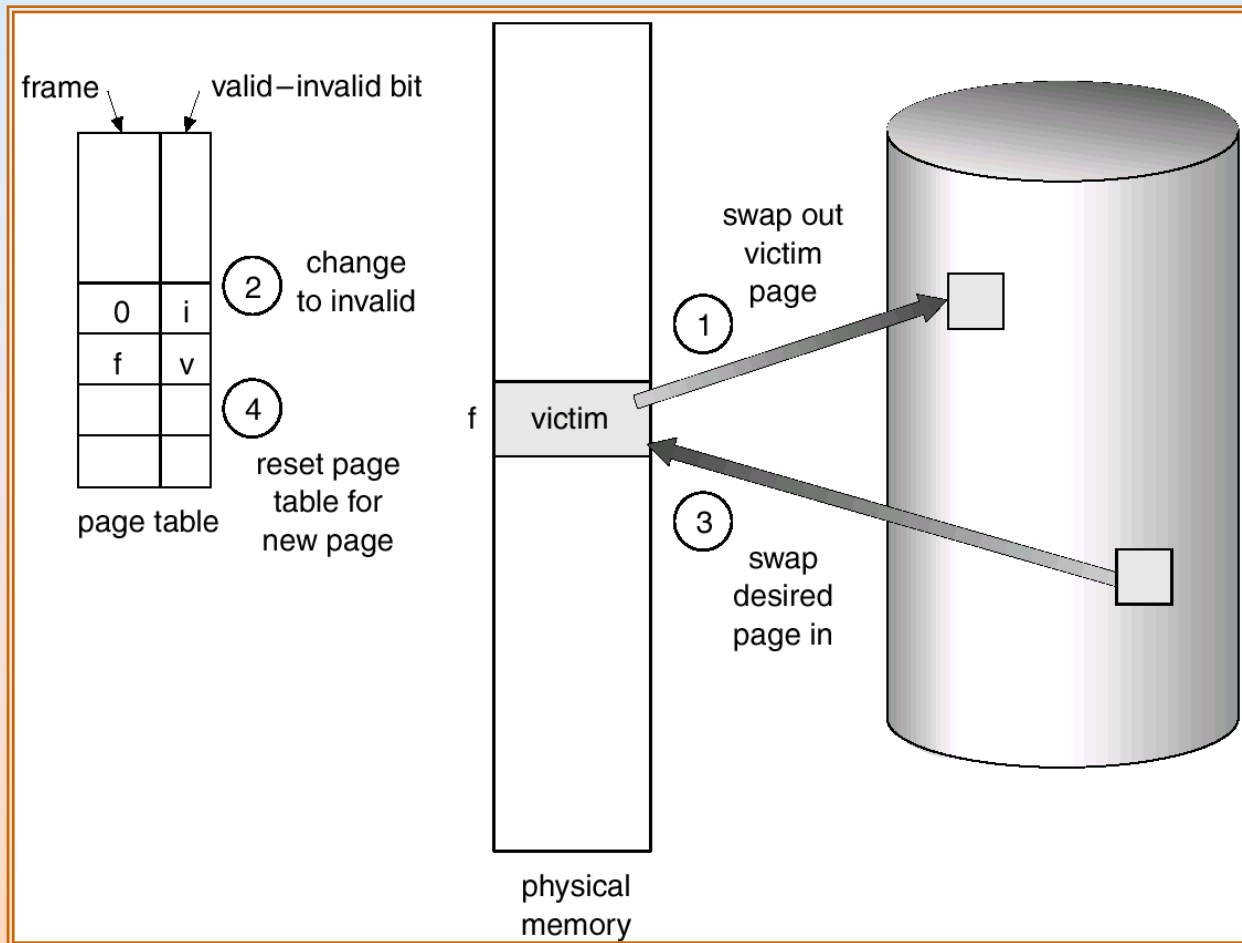
# Besoin de Remplacement de Pages



# Remplacement de Page : Fonctionnement

1. Trouver la page demandée sur disque
2. Trouver un cadre de page libre :
  - Si un cadre de page libre existe, l'utiliser
  - Si un cadre de page libre n'existe pas, utiliser un algorithme de remplacement de page pour choisir un cadre de page **victime**
3. Lire la page demandée dans le cadre de page libre. Mettre à jour les tables des pages et des cadre de page.
4. Relancer le processus

# Remplacement de Pages

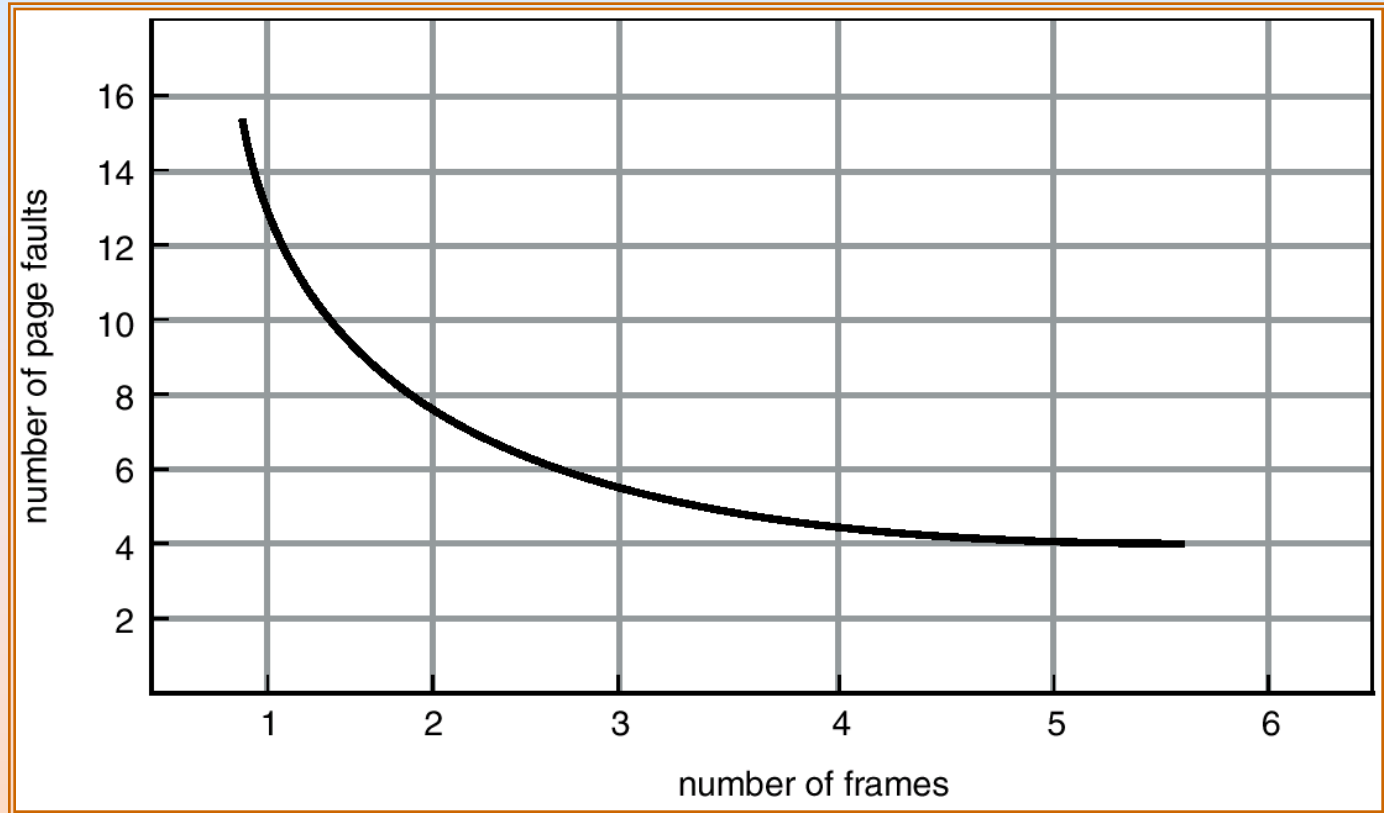


# Algorithmes de Remplacement de Pages

- On voudrait le taux de défauts de pages le plus bas
- Evaluer un algorithme en l'exécutant sur un ensemble de références mémoire et en calculant le nombre de défauts de page en résultant
- Dans tous nos exemples, on utilisera l'ensemble de références suivant

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

# Graphe de Defauts de Pages Vs Nombre de Cadres de page



# Algorithme First-In-First-Out (FIFO)

- Références mémoire : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 cadre de page (3 pages peuvent être en mémoire à un certain moment par processus)

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 cadre de page

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

- Remplacement FIFO – Anomalie de Belady
  - Plus de cadre de page ➡ plus de défauts de pages



# Remplacement FIFO

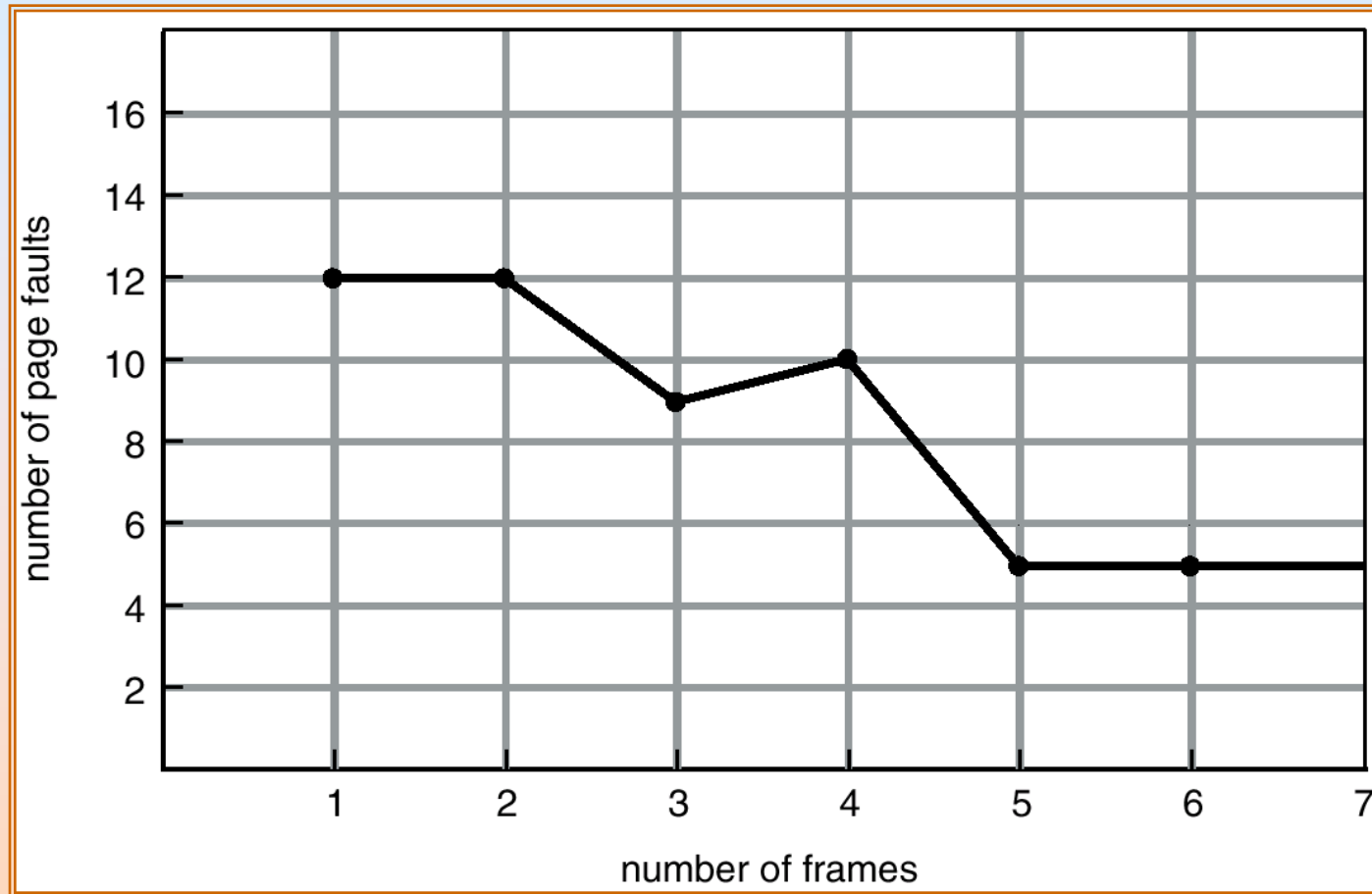
reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2																	
	0	0	0																	
		1	1																	

page frames

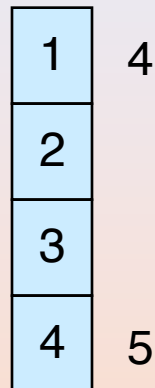
# Anomalie de Belady avec FIFO



# Algorithme Optimal

- Remplacer la page qui ne sera pas utilisée pour la plus longue période
- Exemple avec 4 cadre de page

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



6 page faults

- Difficile à implémenter : Comment avoir l'information sur les références futures ?
- Utilisé pour mesurer les performances des autres algorithmes

# Remplacement Optimal

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2		2		2		2		2				7		
	0	0	0		0		4		0		0		0				0		
		1	1		3		3		3		1						1		

page frames

# Algorithme Least Recently Used (LRU)

- Références mémoire : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	5
2	
3	5 4
4	3

- Implémentation avec compteur
  - Chaque entrée de page a un compteur; à chaque fois qu'une page est référencée, on copie une horloge logique dans ce compteur
  - Quand une page est référencée, incrémenter l'horloge logique et la copier dans le compteur de la page référencée => on a toujours le temps de la dernière utilisation de cette page

# Remplacement LRU

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

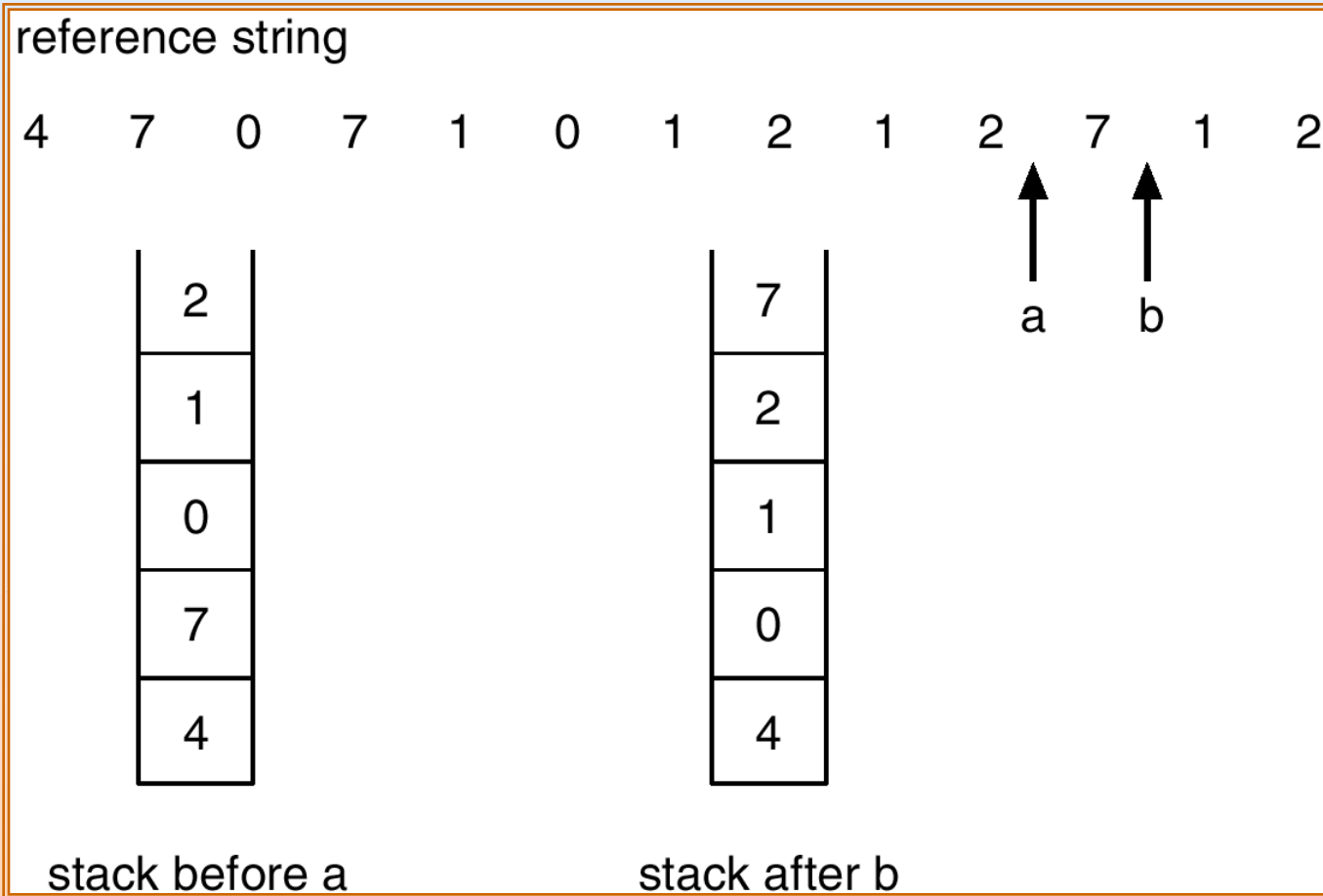
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

page frames

# Algorithme LRU (Cont.)

- Implémentation avec pile – laisser une pile des numéros de page:
  - Page référencée:
    - ▶ Mettre en haut de la pile
    - ▶ Demande le changement de 6 pointeurs au pire !
  - Pas de recherche pour le remplacement

# Pile pour l'Enregistrement des Pages Référencées Récemment





# Approximation Algorithme LRU

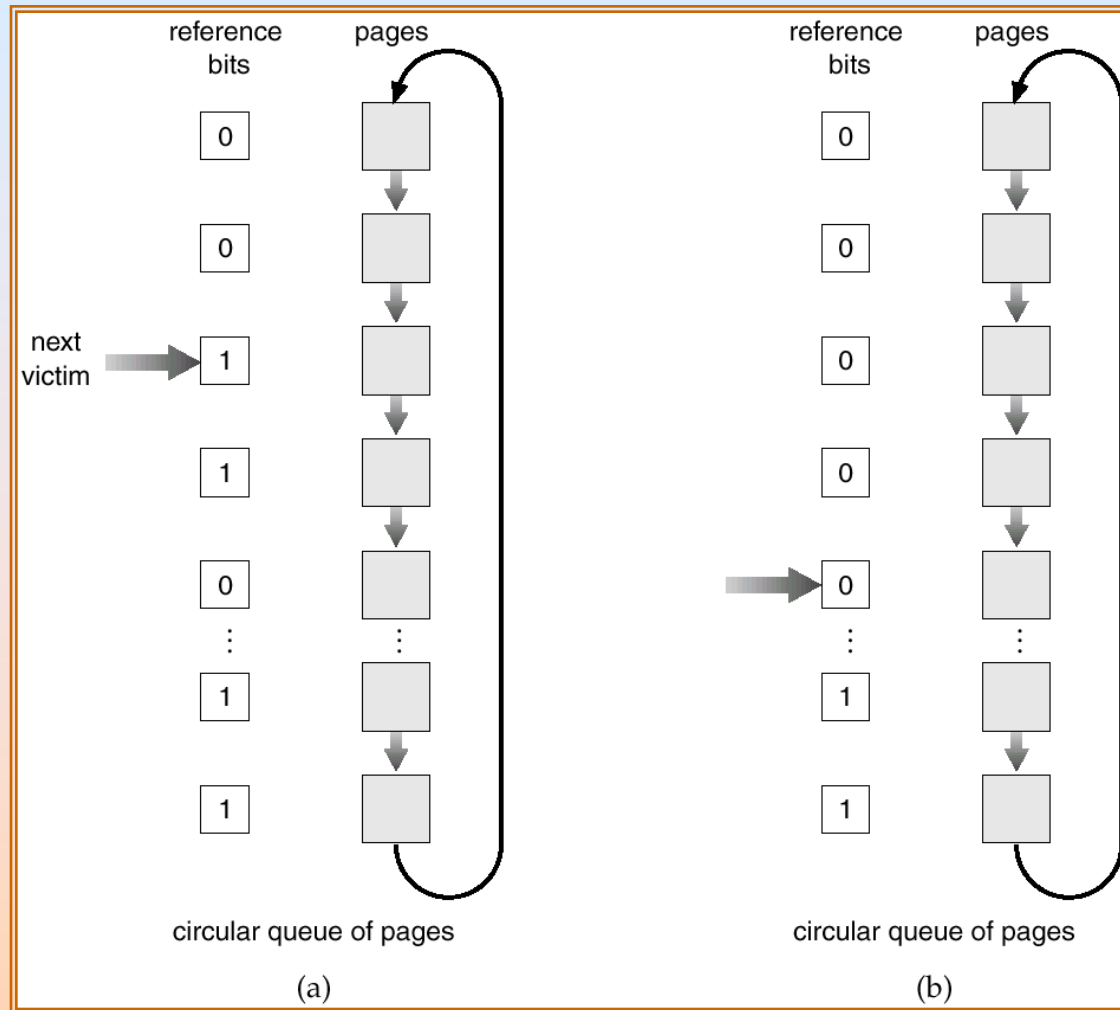
## ■ Bit de Référence

- Avec chaque page, associer un bit, initialement à 0
- Quand une page est référencée, bit mis à 1
- Remplacer une page dont le bit est à 0 (s'il en existe). On ne connaît pas l'ordre !

## ■ Seconde chance

- Bit de référence
- Remplacement type FIFO
- Si la page a remplacer a le bit de reference = 1 alors:
  - ▶ Mettre le bit de reference a 0
  - ▶ Laisser la page en memoire
  - ▶ Remplacer la prochaine page (en suivant l'ordre FIFO), avec les memes regles emises ci-dessus

# Algo. de Remplacement Seconde Chance



# Algorithmes avec Compteur

- Garder un compteur du nombre de références faites sur chaque page
- **Algorithme LFU**: remplace la page avec le plus petit compteur
- **Algorithme MFU**: basé sur l'argument que la page avec le compteur le plus petit a été importée en mémoire et n'a pas encore été utilisée !

# Allocation de Cadres de page

- Chaque processus a besoin d'un nombre *minimum* de pages
- Exemple: IBM 370 – 6 pages pour l'instruction SS MOVE:
  - instruction sur 6 octets, peut être sur 2 pages
  - 2 pages pour *from*
  - 2 pages pour *to*
- Deux schémas majeurs d'allocation
  - Allocation fixe
  - Allocation prioritaire

# Allocation Fixe

- Allocation égale – e.g., si 100 cadres de page et 5 processus, donner à chacun 20 pages
- Allocation proportionnelle – Allouer en fonction de la taille du processus

–  $s_i$  = size of process  $p_i$

–  $S = \sum s_i$

–  $m$  = total number of frames

–  $a_i$  = allocation for  $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$


# Allocation Prioritaire

- Utiliser une allocation proportionnelle avec la priorité comme critère plutôt que la taille
- Si le processus  $P_i$  génère un défaut de page,
  - Choisir pour le remplacement un des ses cadre de page
  - Choisir pour le remplacement un cadre de page d'un processus moins prioritaire

# Allocation Globale vs. Locale

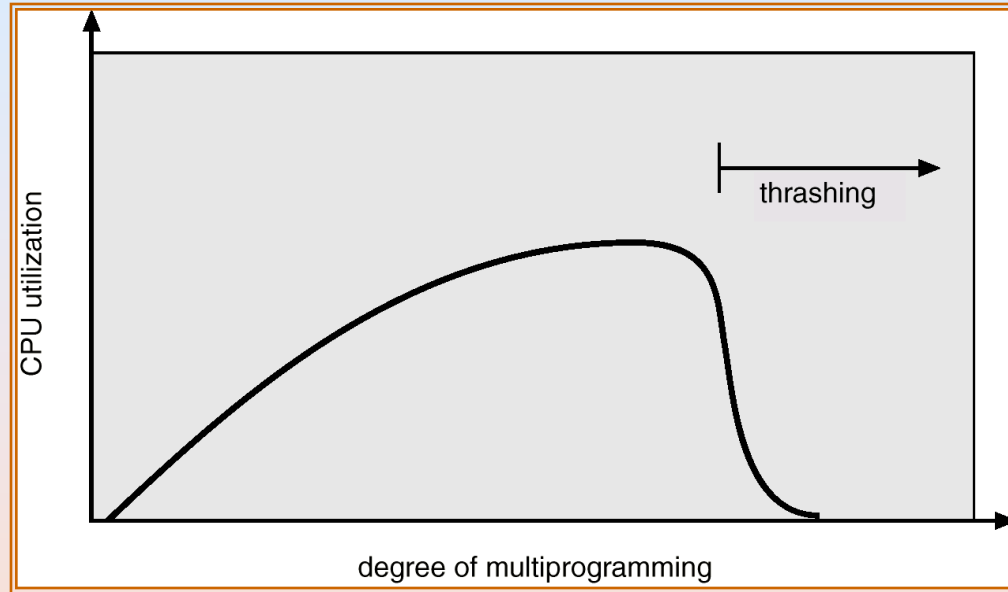
- **Remplacement global** – le processus choisit un cadre de page de remplacement de l'ensemble de tous les cadres de page; un processus peut récupérer un cadre de page d'un autre processus
- **Remplacement local** – chaque processus choisit parmi les cadres de page qui lui sont déjà allouées

# Ecrroulement

- Si un processus n'a pas "assez" de pages, le taux de défauts de pages est assez haut =>
  - Utilisation de la CPU basse
  - L'OS pense qu'il a besoin d'accroître son degré de multiprogrammation
  - Un autre processus est ajouté au système
- **Ecrroulement**  un processus est occupé à *swapper* des pages (in et out)



# Ecrroulement



## ■ Pourquoi la pagination fonctionne ?

Modèle de localité

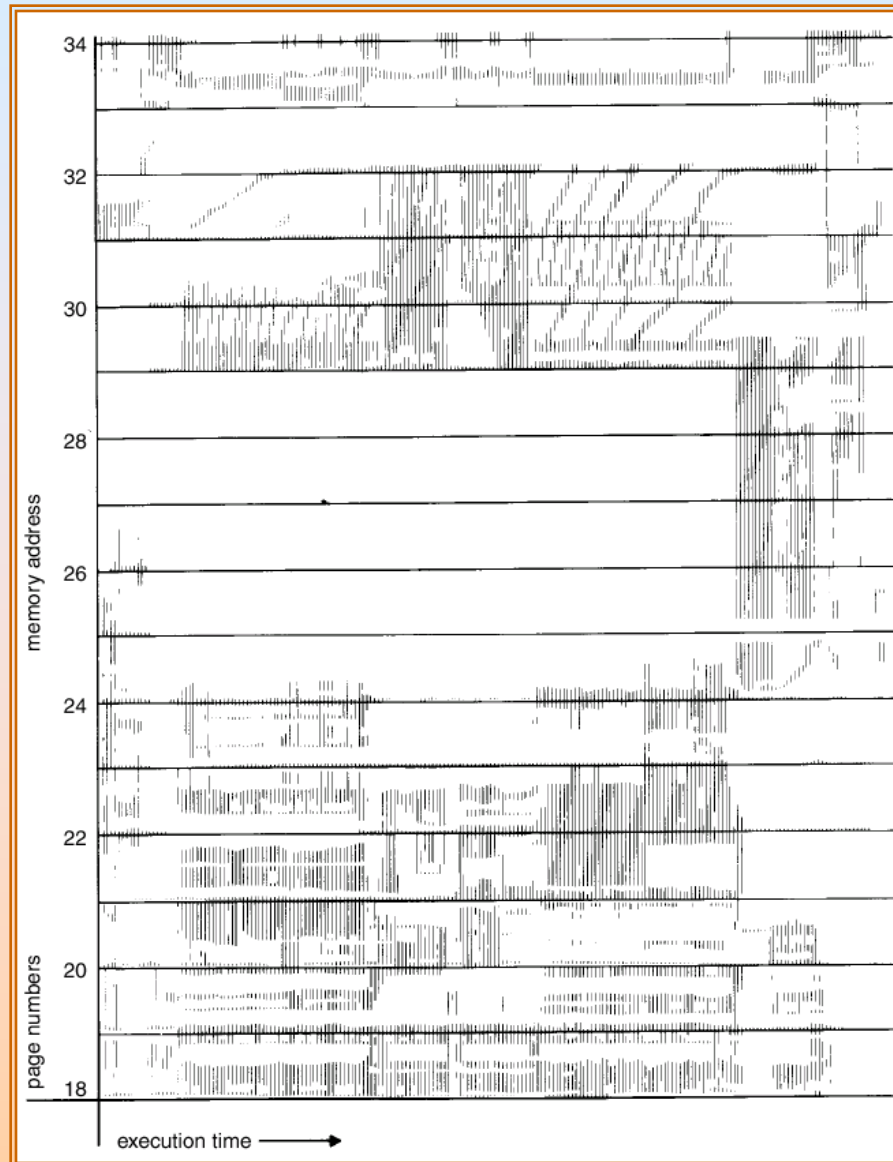
- Les processus migrent d'une localité à une autre
- Les localités peuvent s'entrelacer

## ■ Pourquoi le ecrroulement a lieu ?












taille des localités > taille totale de la mémoire

# Localité des Références Mémoire



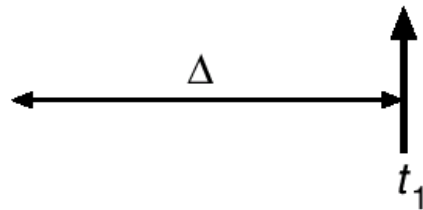
# Working-Set Model

- fenêtre working-set un nombre fixe de références de pages  
Exemple: 10,000 instructions
- $WSP_i$  (working set du Process  $P_i$ ) =  
nombre total de pages référencées durant le temps passé le  
plus récent (varie en fonction du temps)
  - si très petit, la localité ne sera pas prise en compte
  - si trop large, on englobera plusieurs localités
  - si =  on englobera la totalité du programme
- $D = \text{stack icon}$   $WSP_i$  demande totale de cadre de page
- si  $D > m$  (# total de cadre de page)  Ecrroulement
- Politique: si  $D > m$ , alors suspendre un des processus

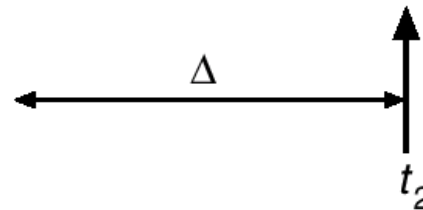
# Modèle du Working-set

page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .





$WS(t_1) = \{1, 2, 5, 6, 7\}$

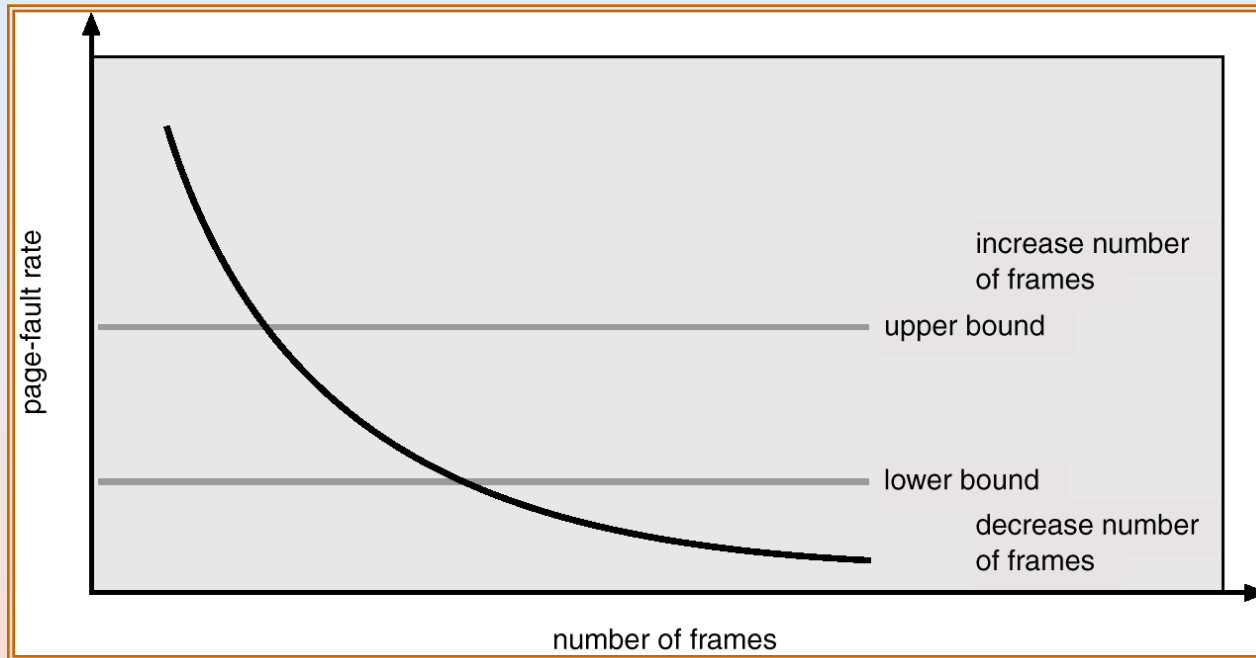


$WS(t_2) = \{3, 4\}$

# Implémentation du Working Set

- Approximation avec une horloge + un bit référence
- Exemple:  = 10,000
  - Interruption d'horloge chaque 5000 unités de temps
  - Garder en mémoire 2 bits pour chaque page
  - A chaque interruption, on copie les bits de référence en mémoire et on les remet à 0
  - Si un des bits en mémoire = 1  page dans le working set
- Pourquoi ce n'est pas totalement précis ?
- Amélioration = 10 bits et interruption chaque 1000 unités de temps

# Fréquence des Defauts de Page

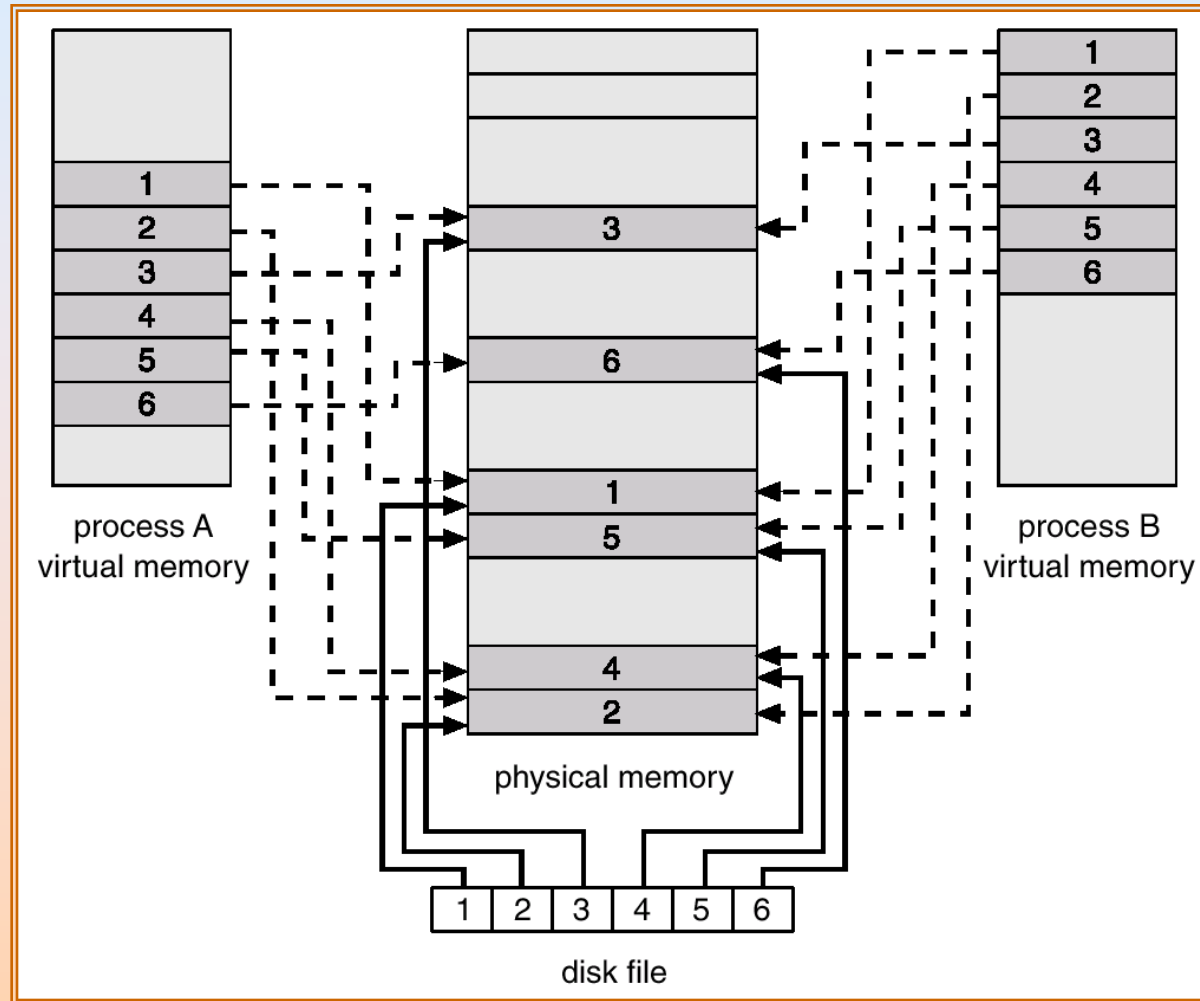


- Etablir un taux “acceptable” de défauts de page
  - Si taux trop bas, le processus perd des cadres de page
  - Si taux trop haut, le processus reçoit des cadres de page en plus

# Fichiers Mappés en Mémoire

- Fichiers mappés en mémoire permet que les E/S soient traitées comme des accès en mémoire en mappant un bloc disque à une page en mémoire
- Un fichier est initialement lu avec une demande de pagination. Une portion du fichier de la taille d'une page mémoire est copiée à partir du système de fichiers en mémoire. Les accès suivants reads/writes suivants du/au fichier sont traités comme des accès mémoire ordinaires
- Simplifie les accès fichiers en traitant les E/S fichiers via la mémoire qu'à travers des appels système **read()** **write()**
- Permet aussi à plusieurs processus de mapper le même fichier en mémoire, et ainsi de partager les pages en mémoire

# Fichiers Mappés en Mémoire





# Fichiers Mappés en Mémoire en Java

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
public class MemoryMapReadOnly
{
    // Assume the page size is 4 KB
    public static final int PAGE_SIZE = 4096;
    public static void main(String args[]) throws IOException {
        RandomAccessFile inFile = new RandomAccessFile(args[0], "r");
        FileChannel in = inFile.getChannel();
        MappedByteBuffer mappedBuffer =
            in.map(FileChannel.MapMode.READ_ONLY, 0, in.size());
        long numPages = in.size() / (long)PAGE_SIZE;
        if (in.size() % PAGE_SIZE > 0)
            ++numPages;
    }
}
```

# Fichiers Mappés en Mémoire en Java (cont)



```
// we will "touch" the first byte of every page
int position = 0;
for (long i = 0; i < numPages; i++) {
    byte item = mappedBuffer.get(position);
    position += PAGE_SIZE;
}
in.close();
inFile.close();
}
```

■ L'API pour la methode map() est la suivante:

map(mode, position, size)

# Autres Considérations

## ■ Pagination à l'avance

- Pour réduire le nombre de défauts de page qui intervient au démarrage du processus
- Paginer à l'avance tout ou une partie des pages dont un processus a besoin, avant qu'elles ne soient référencées
- Mais, si les pages paginées à l'avance ne sont pas utilisées, les E/S et la mémoire sont gaspillées
- Soient  $s$  pages paginées à l'avance et  $\alpha$  les pages utilisées parmi ces pages
  - ▶ Est-ce que le coût des  $s * \alpha$  défauts de page économisés  $>$  ou  $<$  que le coût de la pagination à l'avance des  $s * (1 - \alpha)$  pages non nécessaires ?
  - ▶ Si  $\alpha$  proche de 0  pagination à l'avance mauvaise
  - ▶ Si  $\alpha$  proche de 1  pagination à l'avance gagne

## ■ Le choix de la taille de page doit prendre en considération:

- fragmentation
- taille de table
- Overhead de l'E/S
- localité

# Autres Considérations (Cont.)

- **Couverture TLB** – La taille mémoire accessible à partir du TLB
- $\text{Couverture TLB} = (\text{Taille TLB}) \times (\text{Taille Page})$
- Idéalement, le working set de chaque processus se trouve dans le TLB. Sinon, il y a un degré élevé de défauts de page.

# Autres Considérations (Cont.)

- **Incrémenter la taille de page.** Ceci peut amener à une augmentation de la fragmentation puisque toutes les applications n'en n'ont pas besoin.
- **Fournir des tailles de page différentes.** Ceci permet aux applications qui demandent des tailles de page plus grandes de les utiliser sans augmenter la fragmentation.

# Autres Considérations (Cont.)

## ■ Structure du programme

- `int A[][] = new int[1024][1024];`

- Chaque ligne est sauvegardée sur une page

- Program 1
  - `for (j = 0; j < A.length; j++)`
    - `for (i = 0; i < A.length; i++)`
      - `A[i,j] = 0;`

1024 x 1024 défauts de page

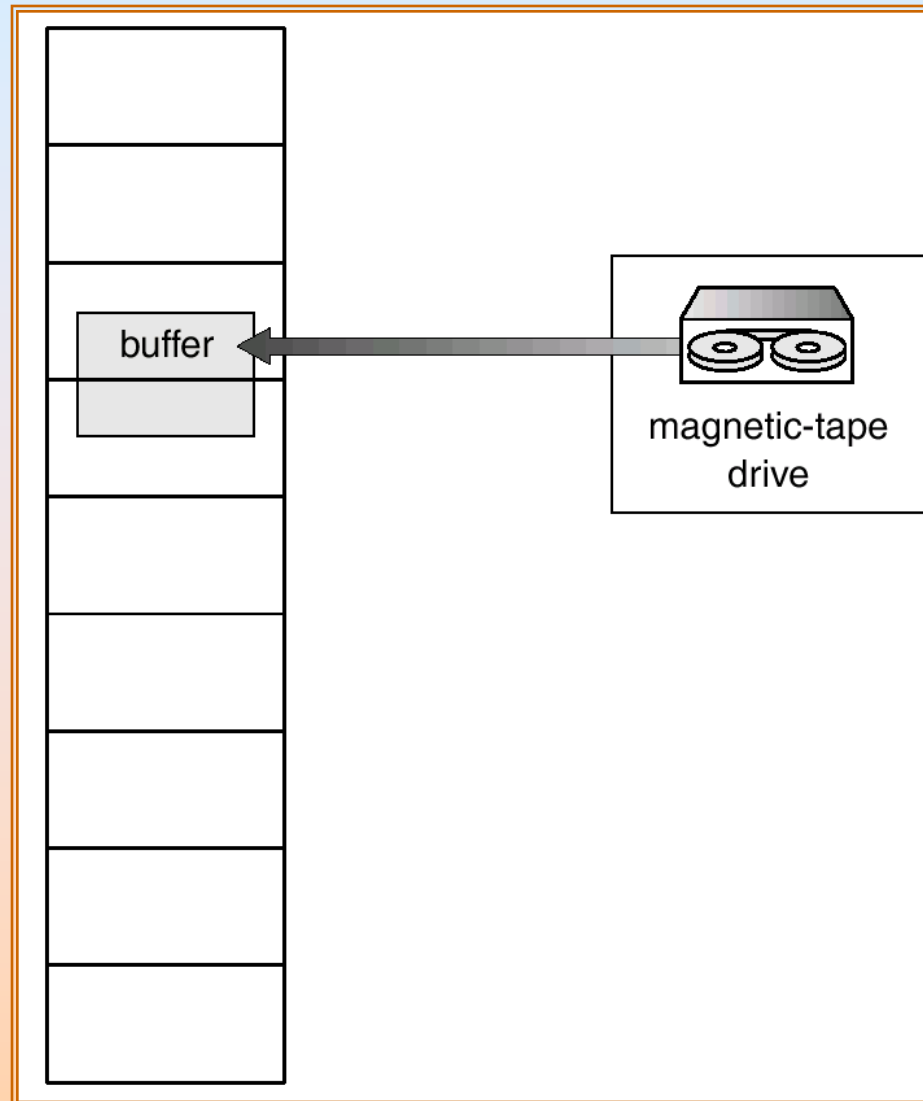
- Programme 2
  - `for (i = 0; i < A.length; i++)`
    - `for (j = 0; j < A.length; j++)`
      - `A[i,j] = 0;`

1024 défauts de page

# Autres Considerations (Cont.)

- **Verrouillage pour les E/S** – Les pages doivent parfois être verrouillées en mémoire
- Considérer les E/S. Les pages utilisées pour copier un fichier d'un périphérique doivent être verrouillées pour ne pas être éligibles à l'éviction par un algorithme de remplacement de pages.

# Pourquoi les Cadres de page d'E/S Restent en Mémoire ?





# Segmentation à la Demande

- Utilisée quand il n'y a pas un support matériel suffisant pour la pagination.
- OS/2 alloue la mémoire en segments; l'information sur ces derniers est sauvegardée dans des descripteurs de segments
- Un segment de descripteur contient un bit de validité pour indiquer si le segment est actuellement en mémoire.
  - Si le segment est en mémoire principale, l'accès continue,
  - S'il n'est pas en mémoire, défaut de segment !

# Exemples d'OS

- Windows XP
- Solaris 2

# Windows XP

- Utilise la pagination à la demande avec du **clustering**. Clustering récupère en mémoire les pages entourant la page “fautive”.
- Les processus sont assignés un **working set minimal** et un **working set maximal**
- Le Working set minimal est le nombre minimum de pages que le processus a la garantie d’avoir en mémoire
- Un processus peut avoir autant de pages en mémoire jusqu’à son working set maximal
- Quand le montant de la mémoire libre dans le système devient plus petit qu’un certain seuil, de l’**élagage automatique du working set** est effectué pour restorer le montant de mémoire libre
- L’élagage du working set supprime les pages des processus qui ont des pages en excès de leur working set minimal

# Solaris

- Maintient une liste de cadres de page libres à accorder aux processus “fautifs”
- *Lotsfree* – seuil (montant de mémoire libre) pour commencer la pagination
- *Desfree* – seuil pour augmenter la pagination
- *Minfree* – seuil pour commencer la pagination
- La pagination est effectuée par le processus *pageout*
- Le processus *pageout* scanne les pages utilisant un algorithme d’horloge modifié
- *Scanrate* est le taux de scan des pages. Ceci va de *slowscan* à *fastscan*
- Le processus *pageout* est appelé plus fréquemment si jamais le montant de la mémoire libre diminue

# Solaris 2 : Scanner de Page

