

Séance 7

Mémoire virtuelle et pagination



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

Rappels

- Problème d'**adressage de la mémoire**
 - Protection de la mémoire et liaison des adresses
 - Adresse logique et physique et traduction par MMU
- Techniques d'**allocation de la mémoire** aux processus
 - Allocation contigüe et fragmentation/compactage
 - Segmentation pour coller à la vue programmeur
 - Pagination et table des pages

Objectifs

- Comprendre la **mémoire virtuelle**
 - Mécanisme de pagination à la demande
 - Principe du défaut de page
 - Algorithmes de remplacement de page
- Principes de l'**allocation de cadres**
 - Méthode d'allocation de cadres
 - Écoulement et modèle de la localité

Mémoire virtuelle (1)

- Utilisation de **mémoire virtuelle** pour plusieurs raisons
 - Exécution de processus pas complètement en mémoire
 - Espace d'adresses logiques peut être plus grand que le physique
 - Partage aisément d'espaces d'adresse
 - Création efficace de processus
- Deux formes d'**implémentation**
 - Pagination à la demande
 - Segmentation à la demande

Contraintes

- Instruction à exécuter doit être en mémoire physique

Pas toujours possible d'avoir tout l'espace logique en mémoire

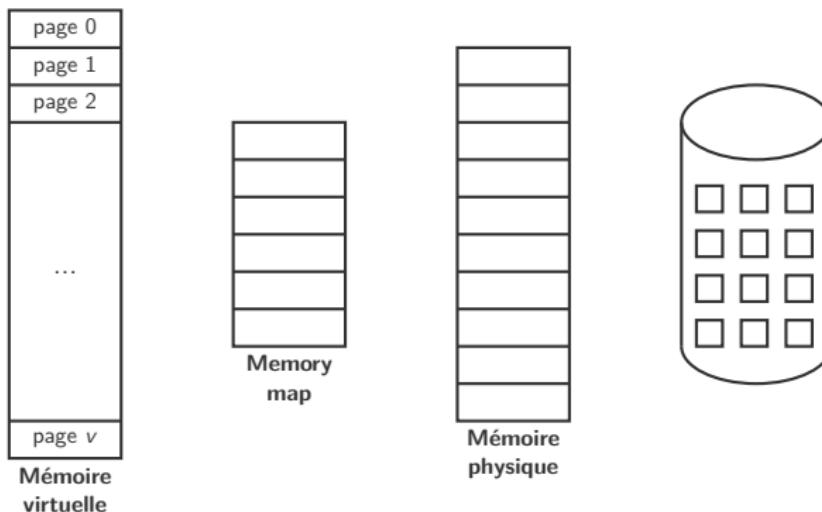
- Le programme complet n'est pas nécessaire...
 - Code de gestion d'erreurs peu fréquemment exécuté
 - Tableaux, listes... demandent plus d'espace que nécessaire
 - Certaines options et fonctionnalités peu utilisées
- ...mais tout pourrait être nécessaire en même temps

Mémoire virtuelle (2)

- Séparation des mémoires **logique/physique**

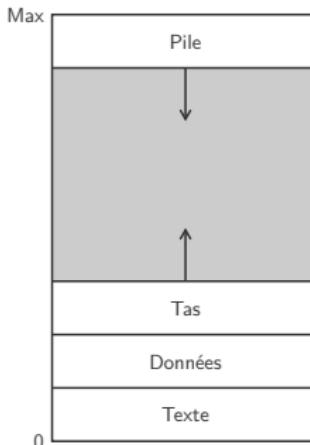
Mémoire logique telle que vue par l'utilisateur

- Un processus utilise un **espace d'adresses virtuelles**



Adresse creuse

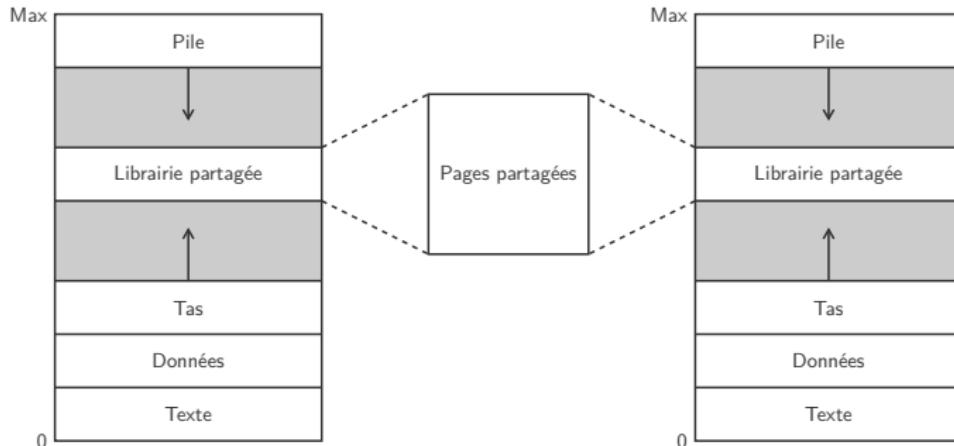
- **Trous** dans l'espace d'adresses virtuel
 - Permet au tas et à la pile de grandir durant l'exécution
 - Permet de partager des librairies (par partage de pages)



Librairie partagée

■ Trous dans l'espace d'adresses virtuel

Liaison dynamique de librairies partagées



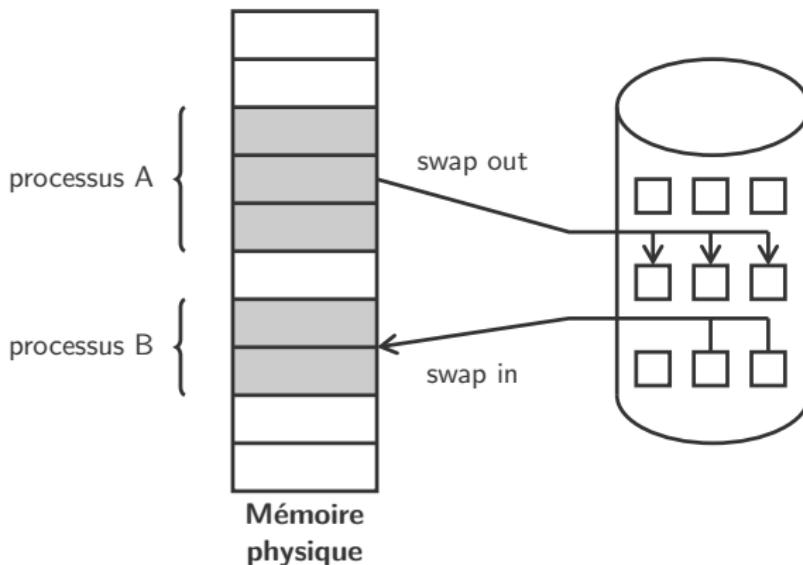
Pagination à la demande



Pagination à la demande (1)

- Pages **chargées à la demande** lorsqu'elles sont nécessaires
 - Moins d'opérations entrée/sortie
 - Moins de mémoire physique nécessaire
 - Réponse plus rapide, plus d'utilisateurs
 - Une page jamais utilisée ne sera jamais chargée en mémoire
- Similaire à un système de pagination avec du **swapping**
Swapper déplace tout un processus, pager uniquement des pages

Pagination à la demande (2)



Défaut de page (1)

- Bit **in/valide** associé à chaque page (dans table des pages)

Valide signifie que la page est légale et en mémoire

Invalide signifie qu'elle est soit invalide, soit pas en mémoire

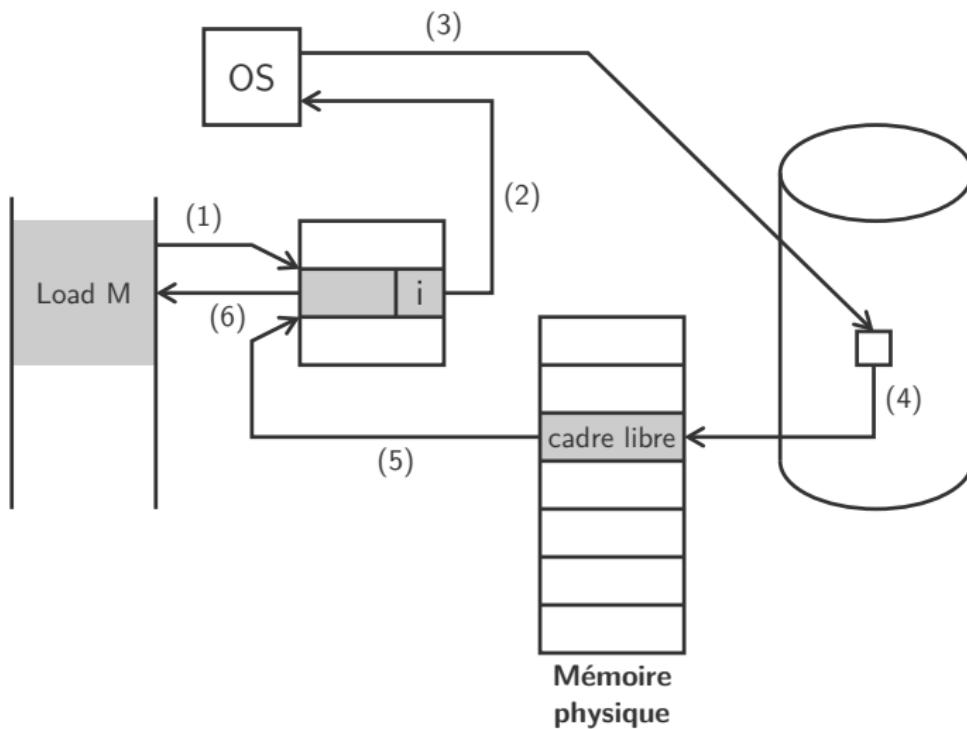
- Accéder à une page invalide provoque un **défaut de page**

Main rendue à l'OS par le gestionnaire de pages hardware

- Exécution d'un **code de gestion** pour rendre la page accessible

Six grosses étapes pour ramener la page en mémoire

Défaut de page (2)



Défaut de page (3)

- 1 Vérification de la **validité de l'adresse** dans une table interne
- 2 Si invalide, fin du processus ; sinon **chargement de la page**
- 3 Choix d'un **cadre libre**
- 4 Demande d'une opération disque dur pour **lire page dans cadre**
- 5 **Modification validité** dans table interne et table des pages
- 6 **Relancement de l'instruction** qui a causé le défaut de page

Localité des références

- Plusieurs pages peuvent être chargées pour une instruction

Une pour l'instruction et plusieurs pour les données

- Se produit rarement par principe de localité des références

Références mémoires toujours proches

Performance (1)

- Si pas de défaut de page

$\text{Temps d'accès effectif} = \text{temps d'accès mémoire (ma)}$

- Si défaut de page, avec probabilité p
 - $(1 - p) \times \text{ma} + p \times \text{temps de gestion du défaut de page}$
- Trois éléments majeurs qui consomment du temps
 - 1 Exécution de l'interruption qui gère le défaut de page
 - 2 Lire la page depuis la mémoire
 - 3 Relancer le processus

Performance (2)

- Valeurs moyennes usuelles des **différents paramètres**
 - ma entre 10 et 200 nanosecondes
 - Temps de gestion du défaut de page environ 8 millisecondes
- **Temps d'accès effectif** usuel
 - $EAT = (1 - p)200 + p(8000000) = 200 + 7999800p$

Avec $p = 1/1000$, on a 8.2 ms (40 fois plus lent)
 - Pour avoir moins de 10% de dégradation de performance
$$220 < 200 + 7999800p \iff p < 0.0000025$$

(1 accès sur 399990 peut faire un défaut de page)

Création de processus

- Premier défaut de page à l'exécution de la première instruction
- Deux solutions en cas de `fork`
 - Copie de toutes les pages du processus
 - Copie-en-écriture
- Copie-en-écriture
 - Pages spéciales marquées comme “*copy-on-write*”
 - Copie de la page lors d'une opération d'écriture
 - Copie vers une page libre et effacée
 - Avantageux pour les processus qui lancent direct un `exec`



Remplacement de page

Sur-allocation de mémoire

- Gain d'opérations E/S par chargement à la demande

Un processus de 10 pages peut ne demander l'accès qu'à 5 pages

- Augmentation du degré de multiprogrammation

Avec 40 cadres, on pourra tourner 8 au lieu de 4 fois le processus

- Possibilité de sur-allocation si pas assez de cadres

- Il n'y a pas que les pages stockées en mémoire principale
- Beaucoup d'espace consommé par les buffers E/S
- Soit terminaison du processus, soit remplacement de page

Remplacement de pages

- Défaut de page, mais **aucun cadre libre**
- Choisir un cadre **victime** non utilisé, et le libérer
 - 1 Écriture du contenu du cadre en swap
 - 2 Mise à jour de la table des pages (invalide)
 - 3 Placer la page demandée dans le cadre libéré
- **Dirty bit** sur chaque page/page en lecture seule

Seule une page modifiée sera écrite sur le disque lors du swap

Algorithme et évaluation (1)

- **Objectif** : Défaut de page le plus faible possible
- Deux **problèmes** à résoudre
 - Allocation des cadres
 - Remplacement des pages
- **Évaluation** d'un algorithme
 - Exécution de l'algorithme sur une séquence de référence
 - Comptage du nombre de défauts de page

Algorithme et évaluation (2)

- Séquence d'adresses enregistrées

0100 0432 0101 0612 0102 0103 0104 0101 0611 0102 0103
0104 0101 0610 0102 0103 0104 0101 0609 0102 0105

- Séquence de référence pour page de 100 octets

1 4 1 6 1 6 1 6 1 6 1

- #Cadre ↑ implique #Défaut de page ↓

Pour l'exemple, avec trois cadres, trois défauts de page

Algorithme FIFO

- La victime est la **page la plus ancienne**

La page qui a été placée dans un cadre le moins récemment

- Une page **activement utilisée** peut être remplacée...

Un défaut de page suivra immédiatement

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

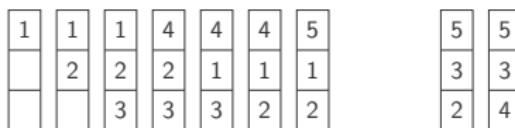
⇒ 15 défauts de page

Anomalie de Belady (1)

- **Augmentation** des défauts de page avec plus de cadres

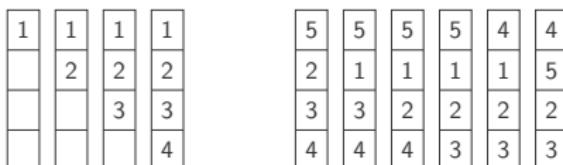
Alors que ce nombre devrait intuitivement diminuer

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



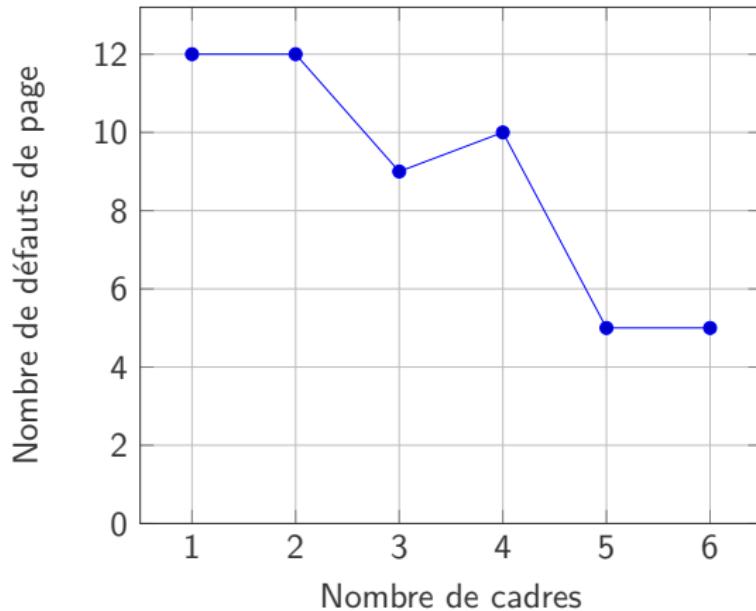
⇒ 9 défauts de page

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



⇒ 10 défauts de page

Anomalie de Belady (2)



Algorithme optimal

- Page victime celle **pas utilisée avant le temps le plus long**

Comment prévoir ce temps ?

- **Garantie prouvée** du plus petit nombre de défauts de page

Comme avec l'algorithme d'ordonnancement SJF

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

⇒ **9 défauts de page**

Least Recently Used Algorithm (LRU) (1)

- La victime est la page **pas utilisée depuis le plus longtemps**
 - Approximation de l'algorithme optimal
 - Moment de dernière utilisation associé à chaque page
- Ne souffre pas de l'**anomalie de Belady**

Comme c'est le cas avec l'algorithme optimal

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

⇒ **12 défauts de page**

Least Recently Used Algorithm (LRU) (2)

■ Compteur

- Compteur ajouté au CPU
- Timestamp d'utilisation associé à chaque page
- Recherche dans toutes les pages du plus petit timestamp

■ Pile

- Maintien d'une pile de numéro de page
- Page référencée retirée et remise au-dessus de la pile
- La moins récemment utilisée est toujours en bas de pile

Approximations de l'algorithme LRU

- LRU nécessite du **support hardware**

Opérations à faire à chaque accès mémoire (horloge ou pile)

- Un **bit de référence** passé à 1 lorsqu'une page est accédée

Initialisation à zéro de tous les bits de référence

- Information associée avec les entrées de la **table des pages**

Information sur l'accès, mais pas l'ordre d'accès

Algorithme bits de références additionnels

- Historique des valeurs des bits de référence

Enregistrement régulier des bits de référence

- Par exemple, un octet de 8 bits pour chaque page

- Mis à jour par une interruption, toutes les 100ms par exemple
 - Ajout du bit de référence à droite et décalage

- Choisir la plus grande valeur

11000100 vient d'être utilisé deux fois (196)

01110111 n'a pas été utilisée au dernier tour (119)

Algorithme seconde chance

- Variante de l'algorithme FIFO

Taille de l'historique réduite à un seul bit

- Utilisation du **bit de référence**

- Si 0, on remplace la page

- Si 1, on change en zéro, mise à jour temps d'arrivée de la page, et on passe à la suite pour trouver la victime (seconde chance)

- Une page avec une seconde chance **gagne tout un tour**

Elle ne sera pas remplacée avant que tout le monde soit passé

Algorithme seconde chance amélioré

- Utilisation de la **paire** (*reference bit, dirty bit*)
- **Quatre valeurs** possibles pour la paire de bits
 - 1 (0, 0) ni utilisée, ni modifiée récemment
Meilleur candidat au remplacement
 - 2 (0, 1) pas récemment utilisée, mais modifiée
Devra être écrite sur le disque en cas de remplacement
 - 3 (1, 0) récemment utilisée, mais inchangée
Sera sûrement utilisée prochainement
 - 4 (1, 1) récemment utilisée et modifiée
Sûrement utilisée prochainement et devra être écrite sur disque

Algorithmes basé sur des compteurs

- Stockage du **nombre de références** faites pour chaque page
- **Least Frequently Used (LFU)**
 - Remplacement du plus petit compteur
 - Page seulement utilisée fréquemment au début lors de l'initialisation restera trop longtemps pour rien en mémoire
- **Most Frequently Used (MFU)**
 - Remplacement du plus grand compteur
 - Page avec petit nombre de références sera sûrement encore bientôt utilisée

Allocation des cadres



Allocation des cadres

- Comment **allouer tous les cadres** entre les processus ?

Comment répartir 93 cadres entre 2 processus ?

- Cas facile du système avec **un seul utilisateur**

- 128 Ko de mémoire, pages de 1 Ko, donc 128 cadres
- Si l'OS prend 35 Ko, il reste 93 cadres libres
- Les 93 premiers défauts de page seront alloués à ces cadres
- À partir du 94^e on lance l'algorithme de remplacement de pages

Allocation fixe

- Allocation **égale** des cadres

Pour m cadres et n processus, m/n cadres par processus

- Allocation **proportionnelle** des cadres

Avec s_i la mémoire virtuelle du processus p_i , et $S = \sum s_i$

Pour m cadres, p_i reçoit $a_i = s_i/S \times m$ cadres

- Les cadres restants forment un **pool de cadres libres**

Stock de sécurité pour des exécutions « urgentes »

Allocation par priorité

- Prendre en compte la **priorité** des processus

Petit plus prioritaire aura moins que gros moins prioritaire

- Un **processus prioritaire** pourrait recevoir plus de mémoire

Pour possiblement accélérer sa terminaison

- Allocation proportionnelle en combinant **taille et priorité**

Toujours mieux de diminuer le degré de multiprogrammation

Allocation globale ou locale

■ Remplacement **global**

- Choix d'un cadre parmi tous les cadres
- Un processus peut voler le cadre d'un autre processus
- Un plus prioritaire peut voler cadre d'un moins prioritaire
- Processus ne contrôle plus son propre taux de défaut de page

■ Remplacement **local**

- Choix parmi les cadres du processus
- Un processus sera coincé dans son ensemble de cadres alloués
- Impact négatif sur le débit de processus exécuté



Écroulement

Écoulement (1)

- Tout processus a un **nombre minimum de cadres** nécessaires

En-deca duquel son exécution est mise en péril

- Nombre de cadres pour un processus **sous son minimum**

- Processus interrompu
- Sortir ses pages de la mémoire, libérer les cadres alloués

- **Écoulement** lorsque le processus swappe des pages

- Pas suffisamment de cadres
- Plus de pages actives que de cadres disponibles

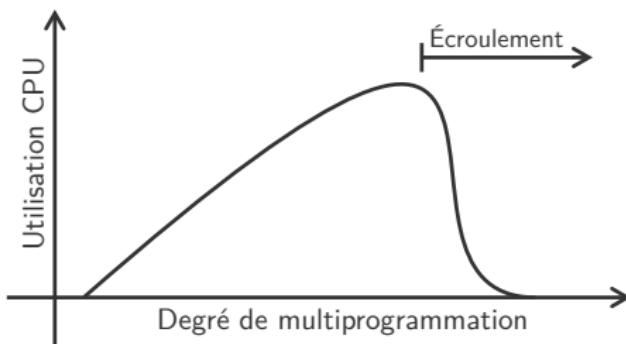
Écoulement (2)

- Au début, l'OS monitore l'**utilisation du CPU**

Ajout de processus augmente le degré de multiprogrammation

- Processus en **attente d'échange de page**

L'utilisation du CPU chute, lancement de nouveau process



Modèle de la localité

- Ensemble de **pages activement utilisées** ensemble (localité)

Exemple : une fonction, avec ses variables locales et globales

- Localités définies par la **structure du programme** et données

Un processus se déplace de localité en localité

- **Assez de cadres** pour la localité courante d'un processus

- Défauts de page pour toutes les pages de la localité
 - Si pas assez de cadres, écroulement

Modèle Working-Set (1)

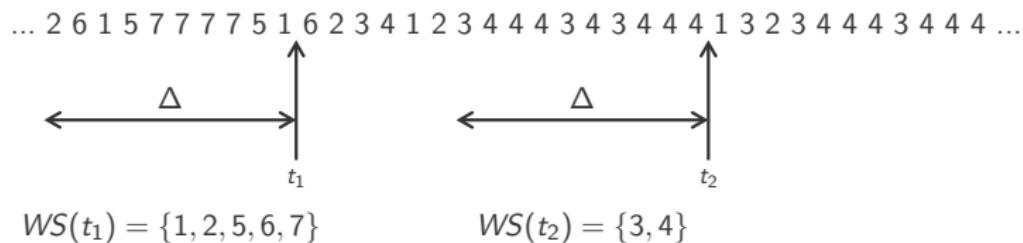
- Fenêtre Working-Set Δ

Examen des Δ références de page récentes

- Valeur de Δ importante pour mesurer au mieux les localités

Mais une trop grande valeur pourrait recouvrir des localités

Référence des pages avec $\Delta = 10$



Modèle Working-Set (2)

- Taille du working-set WSS_i ;
- Demande totale de cadres $D = \sum WSS_i$
 - Si $D > m$, écroulement
 - Il faut alors suspendre un processus pour libérer des cadres
- Implémentation du modèle Working-Set
 - Timer à intervalle fixe avec interruption
 - Bit de référence pour tester l'appartenance au working set

Crédits

- <https://www.flickr.com/photos/slipstreamjc/1336400109>
- <https://www.flickr.com/photos/dolfindans/3725424636>
- <https://www.flickr.com/photos/ansik/2634130269>
- <https://www.flickr.com/photos/mohamedn/2161432715>