IFT2245

Revision Final

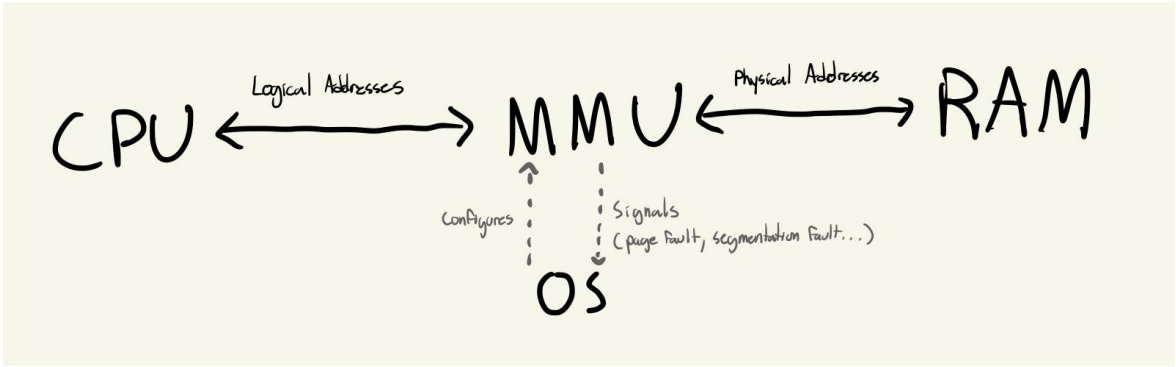
Virtualisation de mémoire

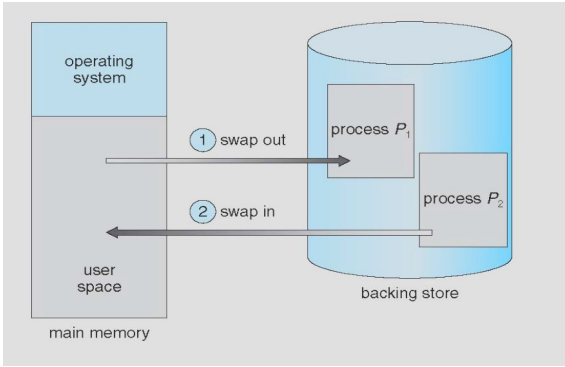
Virtualisation de mémoire

Pourquoi ?

1. **Simplification**: Permet d'écrire du code comme si toute l'espace d'adressage est disponible
2. **Isolation:** Protection de mémoire par le SE
3. **Utilisation efficace:** Addresses logiques > Addresses Physiques

Comment ?





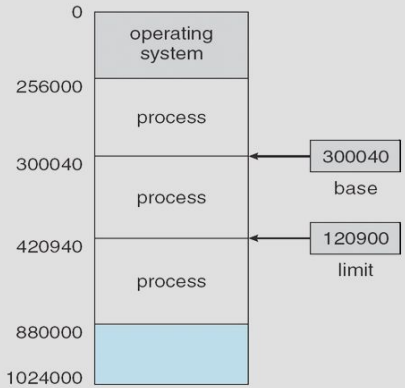
Virtualisation de mémoire

- Adresses logiques (virtuelles) – les adresses utiliser par les processus.
- Adresses physiques – le vrai adresse en mémoire
  - Espace d'adressage logique -> toutes les adresses logiques
  - Espace d'adressage physique -> toutes les addresses physiques

- **Fragmentation externe**: mémoire inutilisable parce que trou trop petit
  - Ex: il y a assez de mémoire libre, mais fragmentée en plusieurs trous tous trop petits
- **Fragmentation interne**: mémoire gaspillée par le système.
  - Ex: le processus a besoin do 600KB mais le SE alloue par morceaux de 1MB, laissant 400KB inutilisées

Virtualisation de mémoire: Segmentation

- Chaque processus reçoit un partie de mémoire
  - Le noyau contrôle les registres base et limit
- CPU vérifie que chaque accès mémoire est entre base et limit
- SE peut accéder à toute la mémoire pour accomplir ses tâches



- First-fit: utilise le premier trou assez grand
- Best-fit: utilise le plus petit trou assez grand
- Worst-fit: utilise le plus gros trou

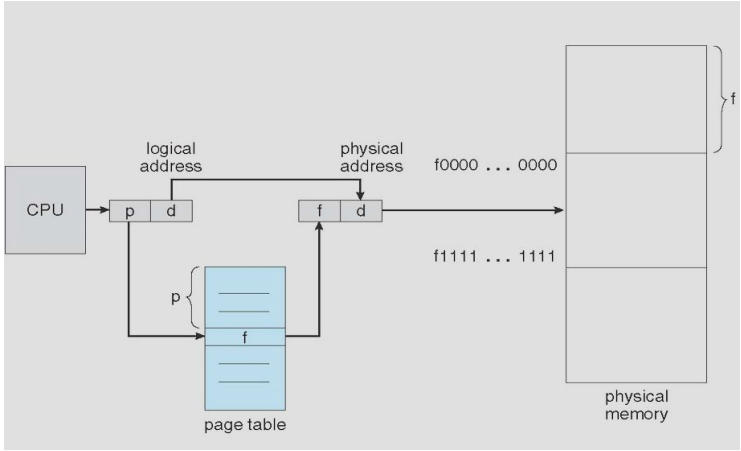
Virtualisation de mémoire: Pagination

- Diviser l'espace logique en **pages**
- Diviser l'espace physique en **frames**
- Chaque page et frame ont la même taille (e.g. 4KB)
- **Table de pages** (page table) effectue la traduction d'adresses logiques en adresses physiques
- Pas de fragmentation externe
- Fragmentation interne à cause de l'allocation par page

- **Numéro de page** (*Page number*) (p) – utilisé comme un index dans une table de page qui contient l'adresse de base de chaque page dans la mémoire physique
- **Décalage** (*Page offset*) (d) – combiné avec l'adresse de base pour définir l'adresse de mémoire physique qui est envoyée à l'unité de mémoire

page number	page offset
p	d
(m – n) bits	n bits

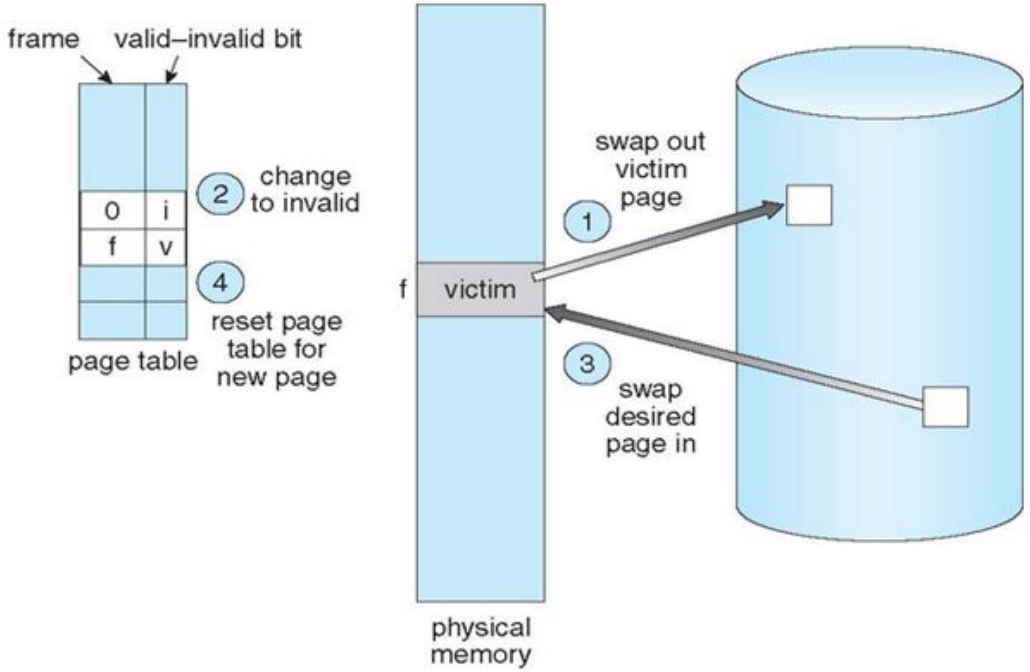
- Pour un espace logique grandeur 2^m et les page de grandeur 2^n



Virtualisation de mémoire: Swapping

• **Valide** (1) : page en mémoire physique, accessible sans défaut de page

• **Invalid** (0) : page hors mémoire, nécessitant un chargement au prochain accès

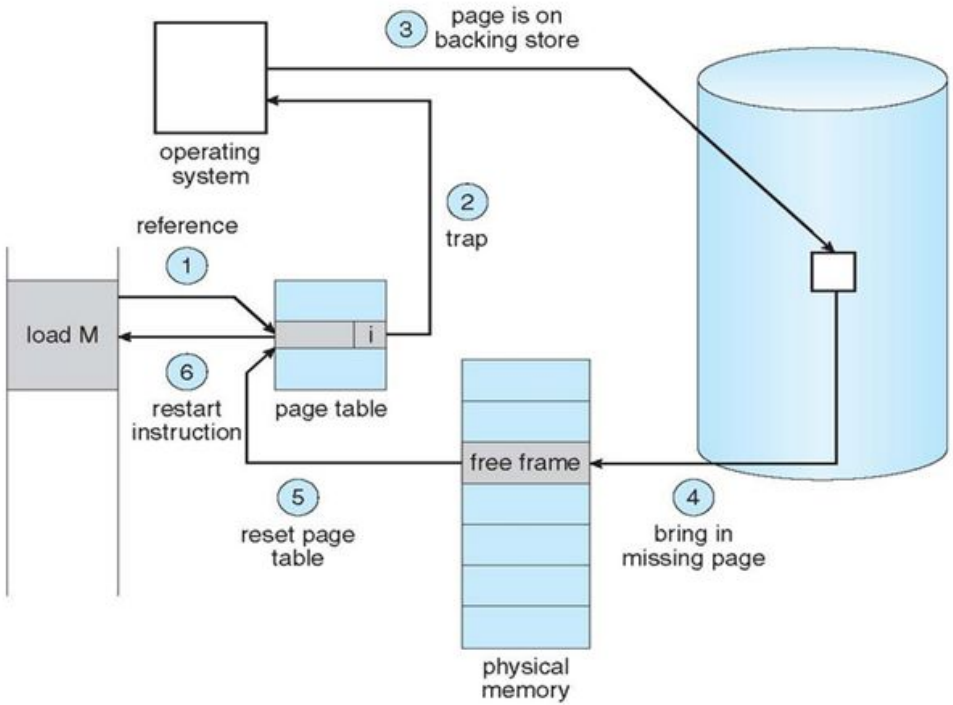


Virtualisation de mémoire: Swapping

Survient lorsqu'une page non chargée en mémoire physique est requise

Processus :

1. Suspension du processus incriminé
2. Localisation de la page sur le disque
3. Utilisation d'un algorithme de remplacement si nécessaire
4. Chargement de la page en mémoire
5. Modification de la page de table
6. Reprise du processus et réexécution de l'instruction



Virtualisation de mémoire: Flux de control

Gestion des accès mémoire virtuelle

```
page = (logical_address & PAGE_MASK) >> SHIFT
tlb_entry = TLB_Lookup(page)
if not tlb_entry: # TLB Hit
  offset = logical_address & OFFSET_MASK
  physical_address = (tlb_entry.frame << SHIFT) | offset
  register = ACCESS_MEMORY(physical_address)
else: # TLB Miss
  pt_entry_address = PT_ComputeAddress(page) # Avec PTBR
  pt_entry = ACCESS_MEMORY(pt_entry_address)
  if not pt_entry: # Page doesn't exist
    raise SEGMENTATION_FAULT
  elif not pt_entry.valid: # Page not in memory
    raise PAGE_FAULT
  else
    TLB_Insert(page, pt_entry.frame) # Replacement algorithm in hardware
    RetryInstruction()
```

Virtualisation de mémoire: Flux de control

Gestion des page faults

```
frame = FindFreeFrame()
if not frame: # No free frames found
  frame = EvictFrame() # Using replacement algorithm of choice
DISK_READ(pt_entry.disk_address, frame)
pt_entry.valid = True
pt_entry.frame = frame
RetryInstruction()
```

Temp d’accès effectif

Temp d'accès mémoire central (Sans page faults)

ε : Temps de cherche dans le TLB

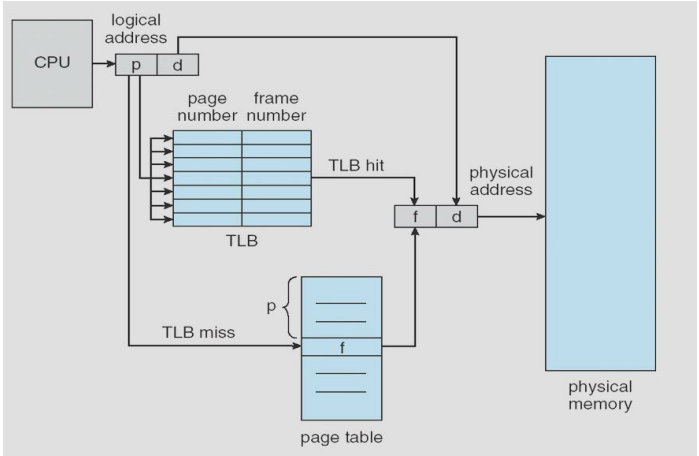
α : TLB hit ratio (% de temps que la frame est trouvée dans le TLB)

m: Temps d’accès à la mémoire

- **Temps d'accès effectif** (*Effective Access Time*) (EAT)

$$EAT = (\varepsilon + 1m)\,\alpha + (\varepsilon + 2m)(1 - \alpha)$$

$$= 2m + \varepsilon - m\,\alpha$$



Temp d’accès mémoire (Avec page faults)

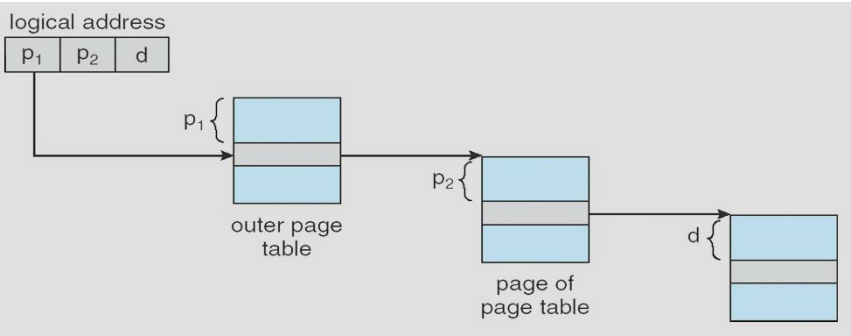
EAT = (1 – p) x memory access time + p x (page fault overhead + swap page in + restart overhead)

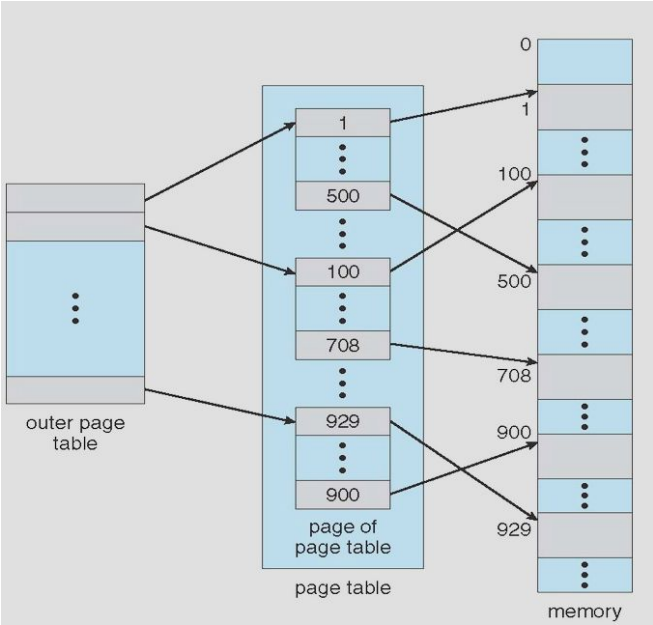
Structures des page de tables

Il existe plusieurs structures possibles:

- Hiérarchique (Multi-Level)
- Inversée
- Hashé

“An extra level of indirection solves every problem!”





Algorithmes de remplacement

Algorithme	Coût	Prédictions Futures	Performance	Anomalie de Belady
OPT	Élevé	Oui	Optimal	Non
Random	Bas	Non	Mauvaise	Oui
FIFO	Médium	Non	Mauvaise	Oui
LRU	Médium	Non	Très bonne	Non
Reference Bit	Bas	Non	Bonne	Oui (Réduit)
Second Chance	Très Bas	Non	Bonne	Oui (Réduit)

Anomalie de Belady: Plus de mémoire != Meilleur performance

Rappel calculs mémoire

2^{10}B = 1024 B = 1KB
2^{20}B = 1 048 576 B = 1MB
2^{30}B = 1GB

$2^{dc} \longrightarrow 2^{c}$ K/M/G… B

Exemple: 64MB = ?? B

$64 = 2^6$ $\quad 2^{20}B = 1MB \quad \Longrightarrow \quad 2^6 \times 1MB = 2^6 \times 2^{20}B = 2^{26}B$

Mais comment faire pour 5GB par exemple???

5GB $\quad 1GB = 2^{30}B \quad \Longrightarrow \quad 5 \times 2^{30}B$

Exercises (Final 2018 Q2)

Les détails suivants s'appliquent aux questions (b) - (d): Étant donné un système avec un espace d'adressage de 32 bits, une taille de page de 4 KB (2^{12} B) et avec la taille d'une entrée dans la table de pages de 4B:

(b) (1 point) Quelle est la taille (en *Bytes*) de la table de page à un niveau ?

(c) (1 point) Considérant que nous voulons que chaque table de pages dans un système de pagination hiérarchique s'inscrive dans un *frame* de mémoire (c'est-à-dire ne soit pas plus grande qu'une page), combien de niveaux de pagination sont nécessaires et comment l'espace d'adressage est-il divisé ?

(d) (2 points) Considérant qu'un programme a besoin de 64 MB (2^{26} B) d'espace mémoire virtuel, combien de tables de pages totales sont nécessaires pour la configuration hiérarchique décrite en (c).

Exercises

Les détails suivants s'appliquent aux questions (b) - (d): Étant donné un système avec un espace d'adressage de 64 bits, une taille de page de 1 MB (2^{20} B) et avec la taille d'une entrée dans la table de pages de 8B (=64b):

(b) (1 point) Quelle est la taille (en *Bytes*) de la table de page à un niveau ?

(c) (1 point) Considérant que nous voulons que chaque table de pages dans un système de pagination hiérarchique s'inscrive dans un *frame* de mémoire (c'est-à-dire ne soit pas plus grande qu'une page), combien de niveaux de pagination sont nécessaires et comment l'espace d'adressage est-il divisé ?

(d) (1 points) Considérant qu'un programme a besoin de 1GB (2^{30} B) d'espace mémoire virtuel, combien de tables de pages totales sont nécessaires pour la configuration hiérarchique décrite en (c).

Bonus (1 point) Dans le schéma d'adressage ci-dessus, pour un de les niveaux dans la hiérarchie, le nombre maximal de bits (pour que la table de pages puisse tenir en mémoire) n'est pas utilisé. Pourquoi est-il préférable que c'est le *premier niveau* de la hiérarchie qui utilise moins de bits ?

Exercises

Les détails suivants s'appliquent aux questions (c) - (f): Étant donné un système avec un espace d'adressage de 64 bits, une taille de page de 2MB et avec la taille d'une entrée dans la table de pages de 64b

(c) (1 point) Quelle est la taille (en *Bytes*) de la table de page à un niveau ?

(d) (1 point) Considérant que nous voulons que chaque table de pages dans un système de pagination hiérarchique s'inscrive dans un *frame* de mémoire (c'est-à-dire ne soit pas plus grande qu'une page),

i. (0.5 points) combien de niveaux de pagination sont nécessaires ?
ii. (0.5 points) combien de bits sont utilisés pour la table de pages la plus externe (*outer*) ?

(e) (1 points) Considérant qu'un processus avec 128MB stocké dans la memoire centrale, combien de rangée dans la table de page d'un niveau décrite en (c) vont être marquées "valide" ?

(f) (1 points) Considérant qu'un programme a besoin de 64GB d'espace mémoire virtuel, combien de tables de pages totales sont nécessaires pour la configuration hiérarchique décrite en (d).

Exercises

Question 3. *Mémoire Virtuelle (4 points au total)*

(a) (3 points) Un programme s'exécute sur une machine et a été alloué **3 frames**. Il accède aux pages suivantes dans l'ordre indiqué:

1 2 3 4 1 5 4 2 3 6 2 5

En présumant que toutes les *frames* sont vides au départ, indique le contenu des *frames* et le nombre total de *page faults* pour chacun des algorithmes de remplacement de page suivants:

i. FIFO (*first in first out*)

ii. Deuxième Chance (*Second Chance* ou "horloge")

iii. Optimal

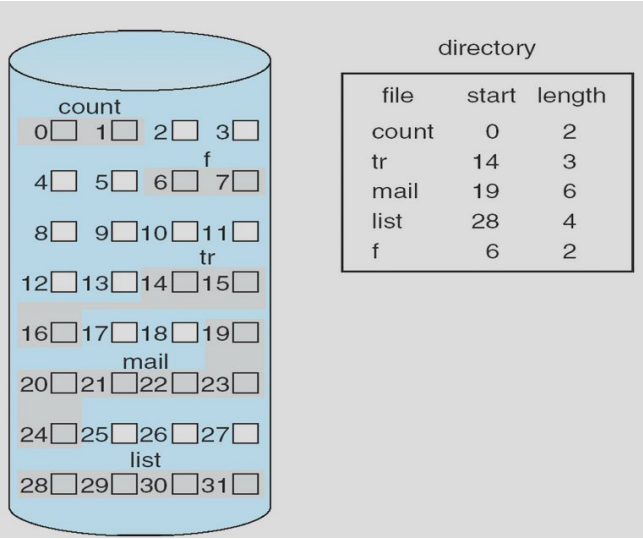
iv) LRU (*least recently used*)

Persistence

Systèmes de fichiers

Méthodes d'allocation

Contiguë



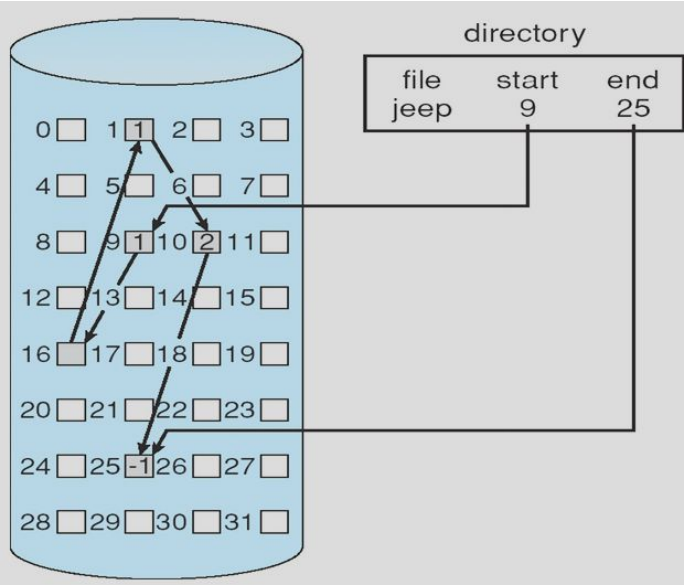
Extent

- ensemble de blocs contigus
- peut être taille fixe (pas de fragmentation externe) ou taille variable (pas de fragmentation interne)

Systèmes de fichiers

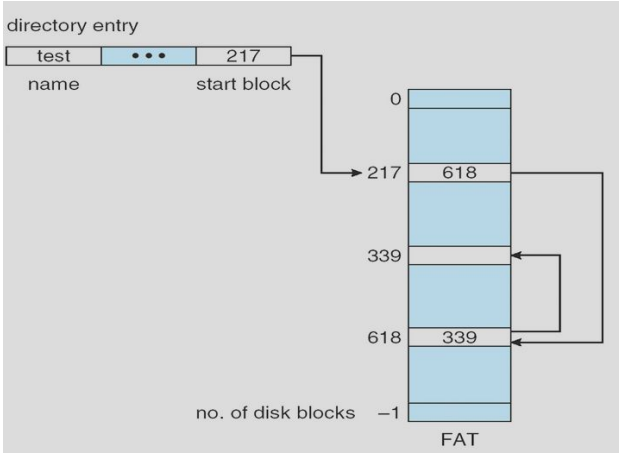
Méthodes d'allocation

Chaîné



- Pas de fragmentation externe, pas de compactage nécessaire
- Améliorer l'efficacité en regroupant les blocs en groupes, mais cela augmente la fragmentation interne
- L'accès direct peut être inefficace
- La fiabilité peut être un problème -> pointeur endommagé perd le reste du fichier

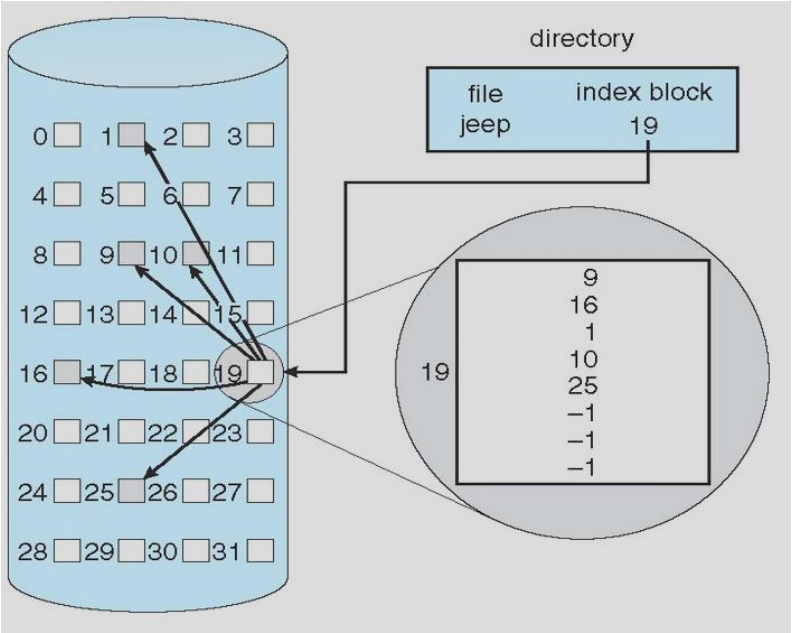
Exemple: FAT



Systèmes de fichiers

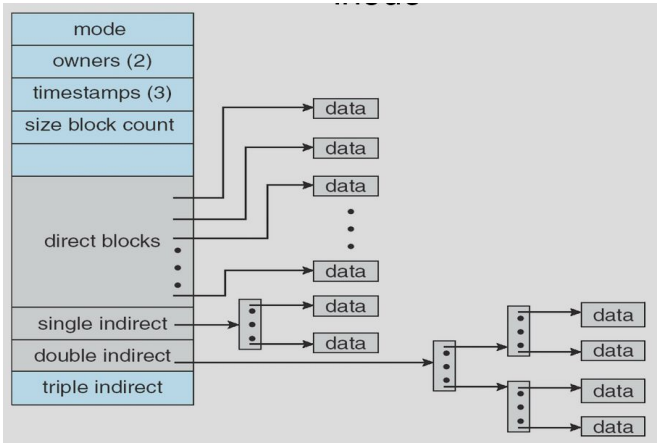
Méthodes d'allocation

Index



- chaque fichier a son propre bloc d'index de pointeurs vers ses blocs de données
- permet accès aléatoire efficace
- pas de fragmentations (sauf le bloc index)
- pas besoin de compresser
- la taille maximale d'un fichier est limitée par la taille du bloc d'index

Exemple: ext



Systèmes de fichiers

Question 4. *Système de fichiers (4 points au total)*

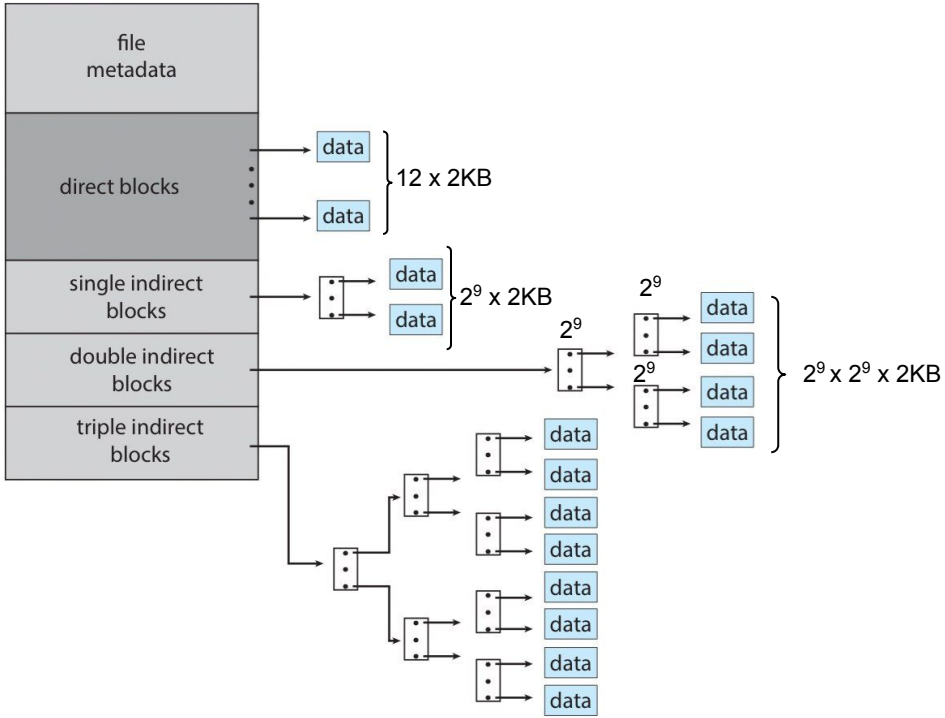
(a) (3 points) Un système de fichiers `ext2` utilise un adressage **32 -bit** et une taille de bloc de **2 KB**.

i. (1 point) Quelle est la taille maximale qu'un fichier peut avoir pour qu'il n'ait pas besoin d'utiliser le pointeur simple (*single*) indirection dans l'inode (en B, peut être une expression) ?

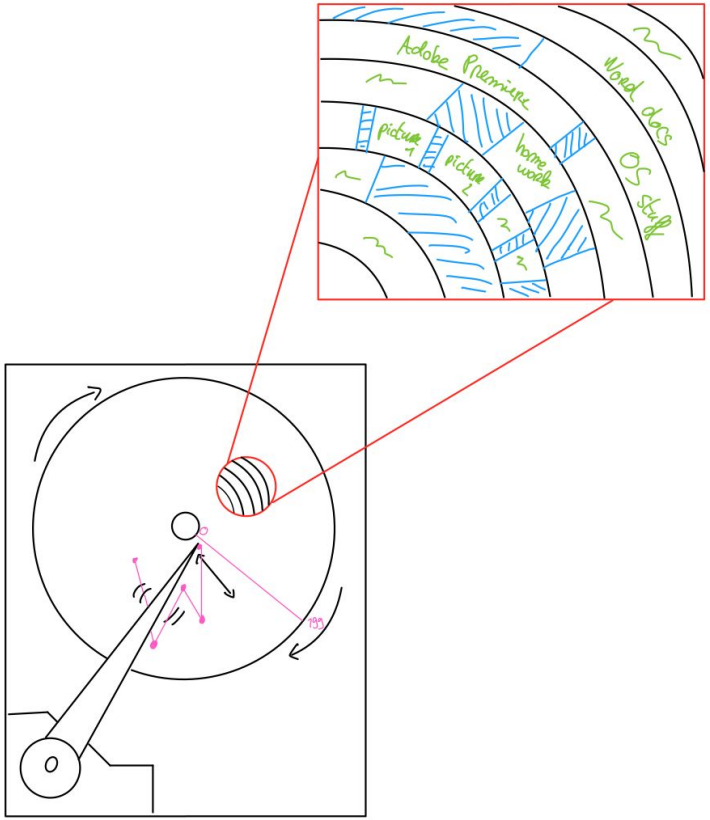
ii. (1 point) Quelle est la taille maximale qu'un fichier peut avoir pour qu'il n'ait pas besoin d'utiliser le pointeur double indirection dans l'inode (en B peut être une expression) ?

iii. (1 point) Quelle est la taille maximale qu'un fichier peut avoir pour qu'il n'ait pas besoin d'utiliser le pointeur triple indirection dans l'inode (en B, peut être une expression) ?

Systèmes de fichiers



Mémoire en masse (HDDs)



- **Transfer time (T_t):** temps pour transférer **un bloc**

$$IOPS = throughput / block\_size$$
$$T_{io} = T_a + T_l + T_t = 1/IOPS$$
$$T_t = block\_size / R_s$$
$$throughput = R_s * T_t / T_{io}$$
$$\text{cas optimal: } T_r = T_{io}$$

Algos:

- FCFS
- SSTF (probleme de famine)
- SCAN (touche le bord)
- LOOK (s'arrête sur celui avant le bord)
- C-SCAN (idem SCAN mais 1 seul sens)
- C-LOOK (idem LOOK mais 1 seul sens)

Mémoire en masse (HDDs)

Parity code

Implique l'ajout d'un seul bit de parité à une chaîne de données pour s'assurer que le nombre total de 1 dans le mot (y compris le bit de parité) est pair (parité paire) ou impair (parité impaire).

Détection d'erreurs: peuvent détecter un nombre impair d'erreurs de bits dans une chaîne de données. Si le nombre d'erreurs rend le total de 1 incohérent avec la parité attendue, une erreur est détectée.

Correction d'erreurs: ne peuvent pas corriger les erreurs ; ils peuvent seulement indiquer qu'une erreur s'est produite.

Plus simple que hamming code.

Hamming code

Vérifie systématiquement les erreurs non seulement dans le mot de données dans son ensemble mais aussi dans des segments du mot de données.

Détection d'erreurs: peuvent détecter et corriger une erreur de bit unique dans une chaîne de données. Ils peuvent également détecter (mais pas corriger) les erreurs de deux bits.

Correction d'erreurs: peuvent corriger une erreur de bit unique dans n'importe quelle chaîne de données, les rendant plus robustes pour la correction d'erreurs que les codes de parité.

Hamming Code

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	✓		✓		✓		✓		✓		✓		✓		✓		✓		✓		…
	p2		✓	✓			✓	✓			✓	✓			✓	✓			✓	✓		
	p4				✓	✓	✓	✓					✓	✓	✓	✓					✓	
	p8								✓	✓	✓	✓	✓	✓	✓	✓						
	p16																✓	✓	✓	✓	✓	

Mémoire en masse (HDDs)

(c) (2 points) Avec un système avec codage de Hamming à parité paire calculée de gauche à droite, vous recevez le code de Hamming 12 bits suivant:

0 0 1 1 1 0 0 1 0 1 1 1

Ce code a une erreur de 1 bit, quel bit ?