Pour réviser...

1. Pourquoi l'appel système `read` retourne-t-il un entier ? Pourquoi ce choix, en quoi est-il utile ?

2. Quand on effectue une lecture ou une écriture dans un fichier, où commence la lecture ou l'écriture dans ce fichier ?

3. Quel est le coût d'un appel système ? Qu'est-ce que cela implique ?

Pour réviser...

1. Pourquoi l'appel système `read` retourne-t-il un entier ? Pourquoi ce choix, en quoi est-il utile ?
   - L'entier indique le nombre d'octets lus, qui peut être inférieur à celui demandé.
   - Hormis le cas 0 (fin de fichier), le système peut donner immédiatement ce qu'il a sous la main, alors que retourner le nombre demandé forcerait une attente.
2. Quand on effectue une lecture ou une écriture dans un fichier, où commence la lecture ou l'écriture dans ce fichier ?

3. Quel est le coût d'un appel système ? Qu'est-ce que cela implique ?

Pour réviser...

1. Pourquoi l'appel système `read` retourne-t-il un entier ? Pourquoi ce choix, en quoi est-il utile ?
   - L'entier indique le nombre d'octets lus, qui peut être inférieur à celui demandé.
   - Hormis le cas 0 (fin de fichier), le système peut donner immédiatement ce qu'il a sous la main, alors que retourner le nombre demandé forcerait une attente.
2. Quand on effectue une lecture ou une écriture dans un fichier, où commence la lecture ou l'écriture dans ce fichier ?
   - à la position courante : find de la denière lecture/écriture, ou du dernier `lseek`.
3. Quel est le coût d'un appel système ? Qu'est-ce que cela implique ?

Pour réviser...

1. Pourquoi l'appel système `read` retourne-t-il un entier ? Pourquoi ce choix, en quoi est-il utile ?
   - L'entier indique le nombre d'octets lus, qui peut être inférieur à celui demandé.
   - Hormis le cas 0 (fin de fichier), le système peut donner immédiatement ce qu'il a sous la main, alors que retourner le nombre demandé forcerait une attente.
2. Quand on effectue une lecture ou une écriture dans un fichier, où commence la lecture ou l'écriture dans ce fichier ?
   - à la position courante : find de la denière lecture/écriture, ou du dernier `lseek`.
3. Quel est le coût d'un appel système ? Qu'est-ce que cela implique ?
   - $>$ 500-1000 cycles, donc ne pas en faire inutilement.

Gestion du système de fichier

- Structures des fichiers sur disques
- Structures des fichiers en mémoire : caches
- Opérations sur les fichiers en mémoire

Problème avec la représentation contigües

...	Fichier A	Libre	Grand Fichier B	Fich. C	...

- On ne peut pas toujours agrandir un fichier existant (*e.g.* B)
- Fragmentation : il peut ne plus y avoir de place pour ranger un fichier de grande taille alors qu'il y a plein de petits trous.
- Problème des défauts matériels (zône défectueuse).

Représentation par blocs

- Le contenu d'un fichier est une suite non contigües de blocs,
- Les blocs sont de taille fixe, pour simplifier :
  - Gros blocs = place perdu pour les petits fichiers
  - Petits blocs = lectures des gros fichiers lente (il faut bouger le bras entre les blocs non contigus)
  - Typiquement, **4KO** $\in$ 1KO–16KO (parfois, 2 tailles possibles)

Un fichier est identifié par un inode

- Le numéro de l'inode sert à désigner le fichier depuis les répertoires qui le contiennent.
- L'inode contient les méta-informations sur le fichier (droits, type, taille, etc.)
- L'inode contient également un tableau des blocs qui composent le fichier.

Un fichier est identifié par un inode

- Le numéro de l'inode sert à désigner le fichier depuis les répertoires qui le contiennent.
- L'inode contient les méta-informations sur le fichier (droits, type, taille, etc.)
- L'inode contient également un tableau des blocs qui composent le fichier.

Question Les blocs pourraient être chainés. Quels sont les avantages à les mettre dans l'inode ?

Un fichier est identifié par un inode

- Le numéro de l'inode sert à désigner le fichier depuis les répertoires qui le contiennent.
- L'inode contient les méta-informations sur le fichier (droits, type, taille, etc.)
- L'inode contient également un tableau des blocs qui composent le fichier.

Question Les blocs pourraient être chainés. Quels sont les avantages à les mettre dans l'inode ?

- Permet un accès séquentiel.
- Sépare la structure de chaînage des données (La taille est un multiple de deux sur le matériel, disque et procésseurs. Un entête sur quelques octets briserait cette régularité d'un coté.)

La structure des fichiers sur le disque

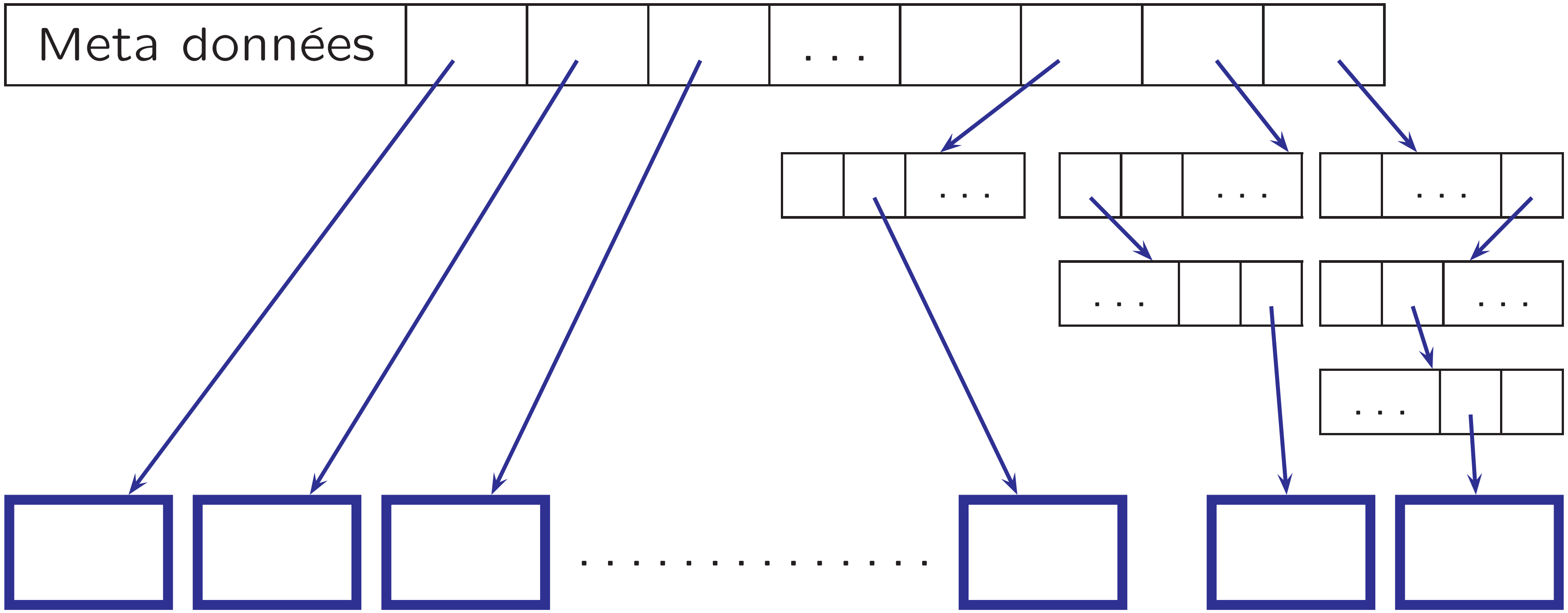
Disque peut contenir plusieurs systèmes de fichier.

- table des partitions
- partitions (en séquence)

Partition a la structure logique suivante :

- Secteur de boot : permet de démarrer sur cette partition, peut être vide.
- Super bloc : contient l'état du disque : nombre de fichiers (inodes), place libre, où trouver de l'espace libre.
- Liste des inodes : leur nombre est déterminé au formattage.
- Blocs de donnée : un bloc appartient au plus à un fichier.
  - Tous les blocs ont la même taille (quelques exceptions).
  - Une plus grande taille permet (dans une certaine mesure) un transfert de données plus économique, mais augmente en moyenne la place perdue sur le disque.

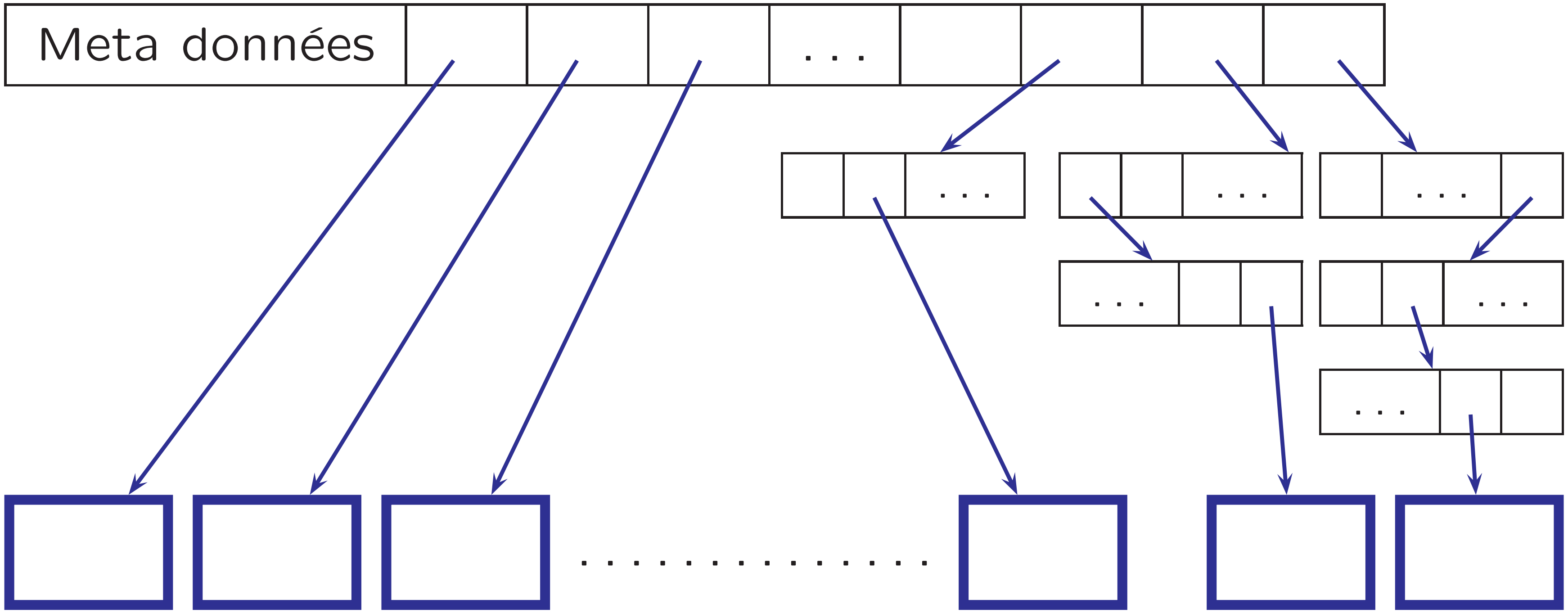
Structure d'un inode



Structure

- Meta-données : taille, dates, *etc.* (information de `stat`)
- Liste des k-premiers blocs (sans indirection)
- Blocs de simple, double et tripple indirection.

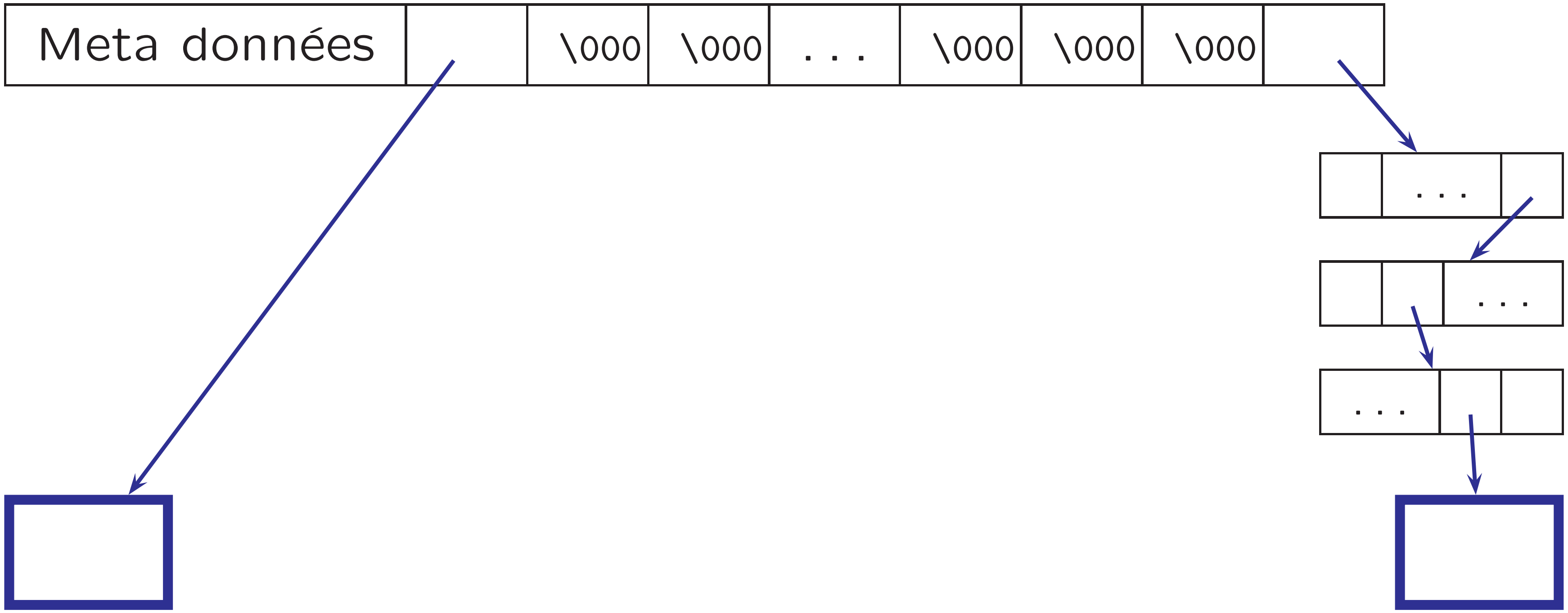
Structure d'un inode



Avantages

- Efficacité : accès directs pour les petits fichiers
- Capacité : 1 TO si blocs de 4 KO (et inodes sur 4 octets).
- Bloc numéro 0 inexistant, réprésente un bloc de '\000', alloué si besoin (écriture d'un octet non nul).

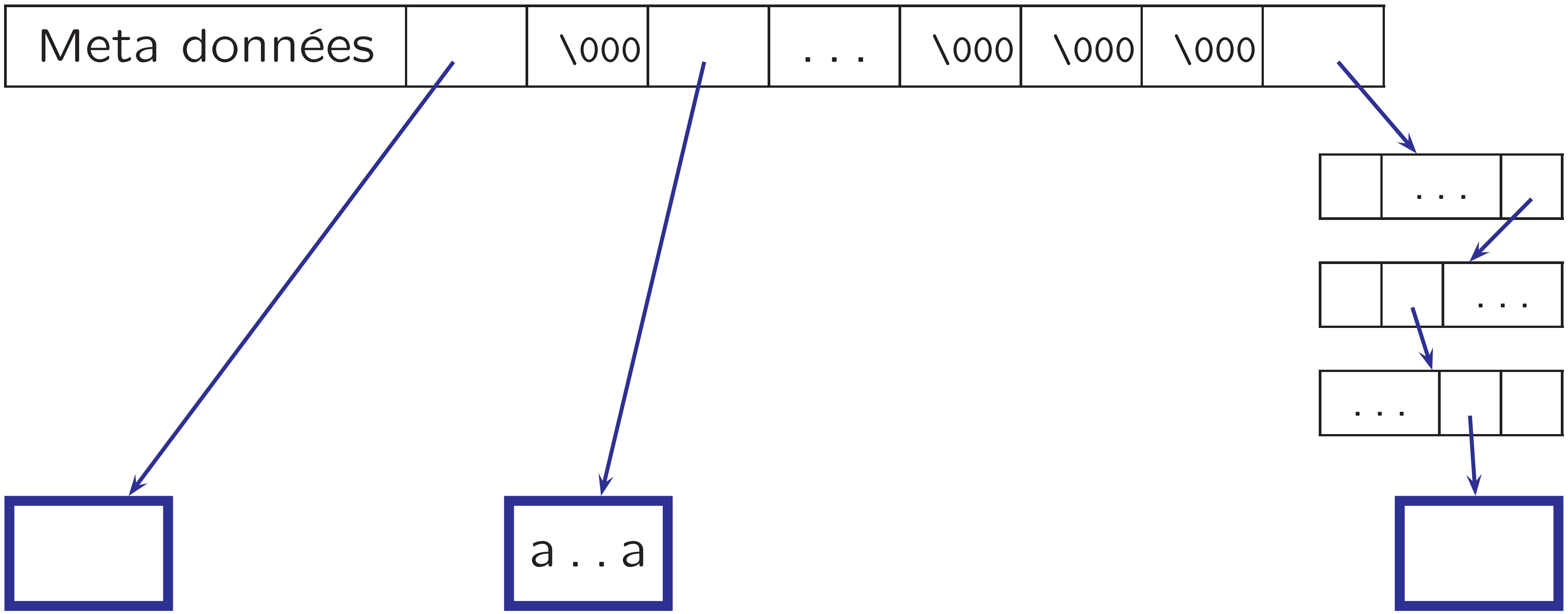
Structure d'un inode



Fichiers presques vides

- Bloc numéro 0 inexistant, réprésente un bloc de '\000',
- ou un bloc indirect de blocs '\000',

Structure d'un inode



Fichiers presques vides

- Bloc numéro 0 inexistant, réprésente un bloc de '\000',
- ou un bloc indirect de blocs '\000',
- alloué si besoin (écriture d'un octet non nul).
- Un fichier vide de grande taille ne contient aucun bloc.

Représentation des répertoires

Même représentation que les fichiers

- Le type de fichier est différent dans les méta-données.
- Le contenu du fichier est une suite de paires formées d'un numéro-d'inode et d'un nom.

Avantages

- Le système utilise le même code pour lire le contenu d'un bloc sur le disque, qu'il s'agisse d'un répertoire ou d'un fichier.
  La différence est son interprétation une fois lu en mémoire.
- Il n'y a pas de limite a priori au nombre d'entrées par répertoire.
- Si les noms de répertoire sont de taille non bornée, les entrées sont chaînées.

Tous les blocs libres sont chaînés dans une liste

- Construite au formattage du disque.
- Lorsque qu'un inode est libéré, ses blocs sont libérés et sont remis dans la liste des blocs libres.

Organisation de la liste libre

- Un champ du super bloc pointe vers la liste des blocs vides.
- On utilise des blocs libres pour former les cellules de la liste des blocs libres (vide quand le disque est plein).
  - Leur contenu est une liste de numéros de blocs libres.
  - Le premier d'entre eux est la cellule suivante de la liste.

Schématiquement

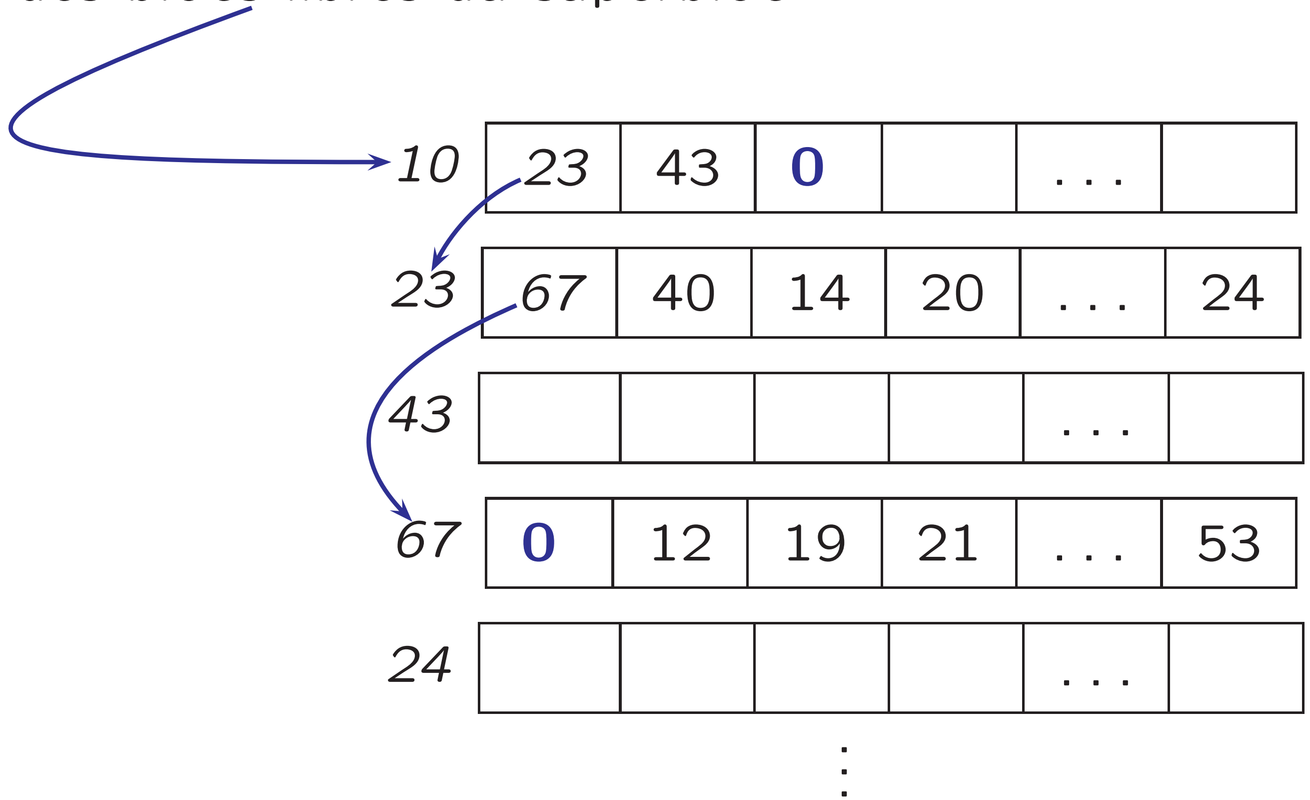
Liste des blocs libres du superbloc

10	*23*	43	**0**		...	
23	*67*	40	14	20	...	24
43					...	
67	**0**	12	19	21	...	53
24					...	

$\vdots$

Schématiquement

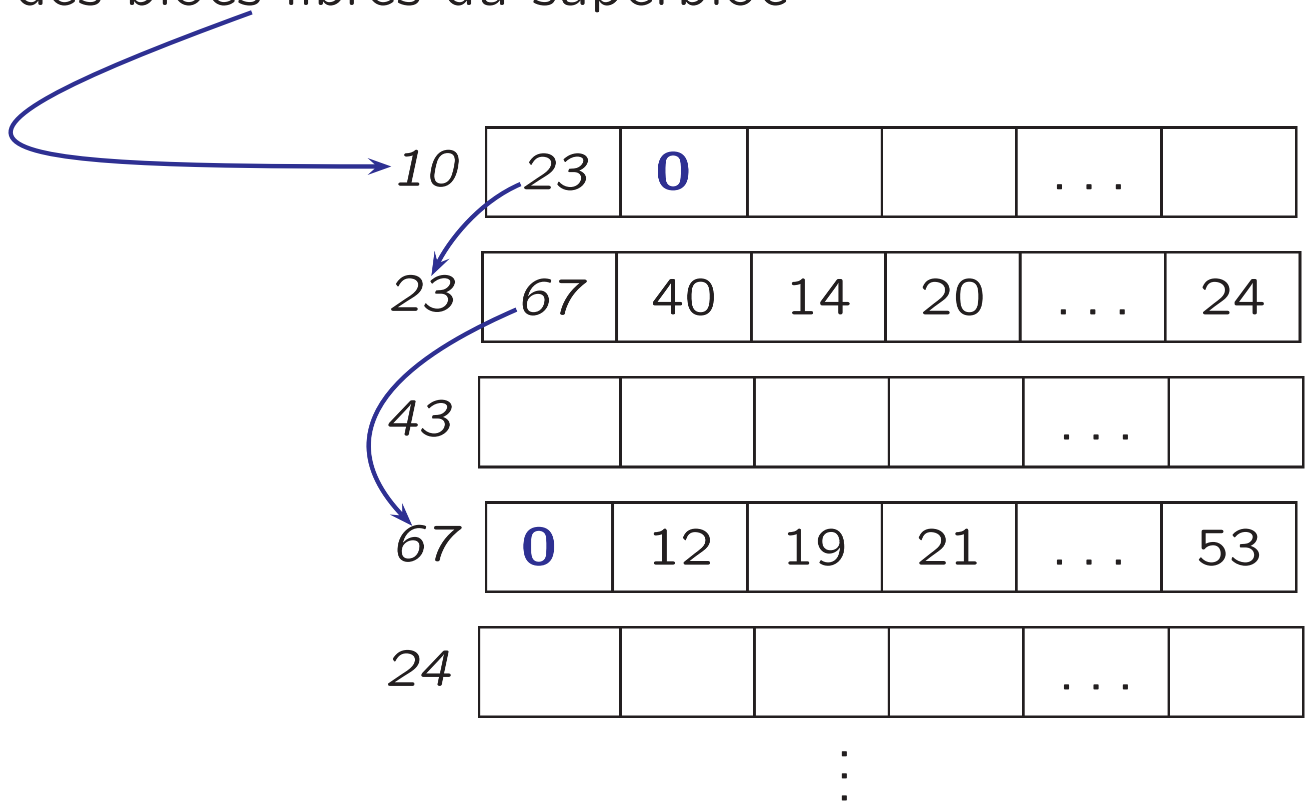
Liste des blocs libres du superbloc



- Allocation d'un bloc

Schématiquement

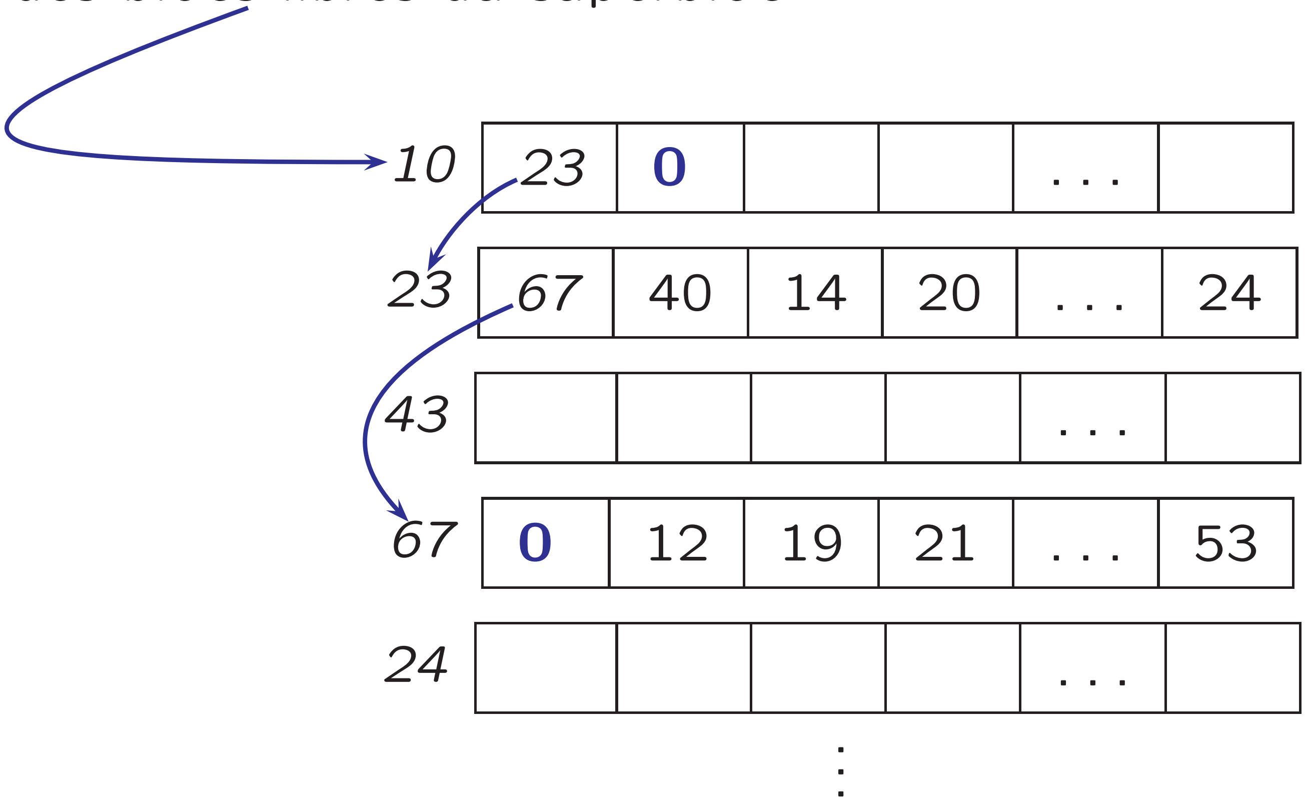
Liste des blocs libres du superbloc



- Allocation d'un bloc $\longrightarrow$ 43

Schématiquement

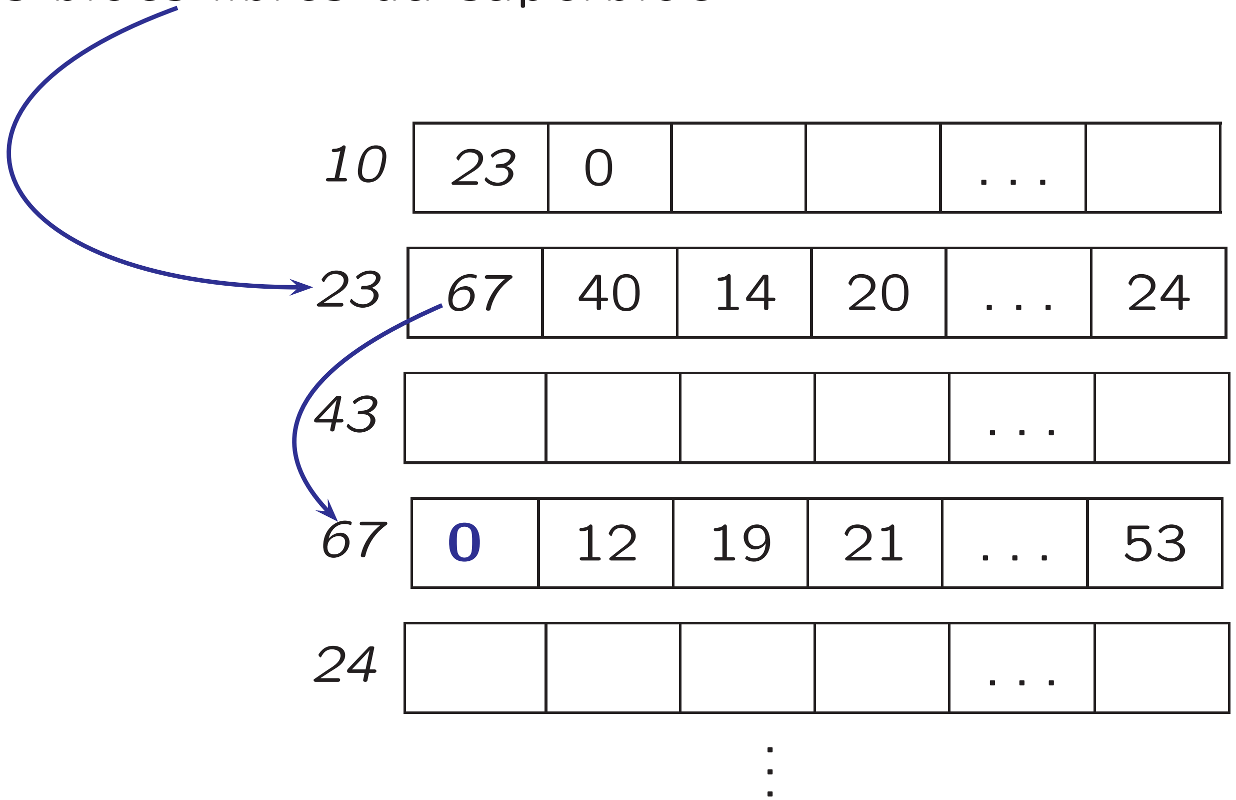
Liste des blocs libres du superbloc



- Allocation d'un bloc $\longrightarrow$ 43
- Allocation d'un bloc

Schématiquement

Liste des blocs libres du superbloc



- Allocation d'un bloc $\longrightarrow$ 43
- Allocation d'un bloc $\longrightarrow$ 10

Gestion des inodes libres

Une solution mixte possible

- Liste partielle des inodes libres (comme les blocs libes)
- Numéro du plus grand inode dans la liste libre
- Nombre total d'inodes libres.

Recherche d'un inode libre

- On les prend dans la liste des inodes libres.
- Quand il n'y en a plus on reconstruit une liste partielle d'inodes libres en parcourant la liste des inodes à partir de l'ancien plus grand inode libre dans la liste libre (maintenant vide).

Explication des différences

- On peut tester si un inode est libre par son nombre de liens.
- Ce n'est pas possible pour les blocs sans un parcours du disque complet : un bloc n'a pas de structure.
- Les blocs sont beaucoup plus sollicités que les inodes.

Autres solutions

Utilisation de tables de bits

- Prend une place constante.
- Recherche plus chère quand le disque est presque plein.

Exercice

Combien faut-il d'accès disque pour lire le contenu d'un petit fichier désigné par son nom dans le répertoire courant ?

Autres solutions

Utilisation de tables de bits

- Prend une place constante.
- Recherche plus chère quand le disque est presque plein.

Exercice

Combien faut-il d'accès disque pour lire le contenu d'un petit fichier désigné par son nom dans le répertoire courant ?

- Lire l'inode du répertoire,
- Lire le contenu du répertoire,
- Lire l'inode du fichier,
- Lire le fichier

Petites variations

- La taille des blocs peut varier d'un système de fichier à un autre.
- Un système de fichiers peut utiliser plusieurs tailles de blocs ou fragmenter les blocs élémentaires en 4 ou 8 et donc pouvoir utiliser un seul bloc pour plusieurs très petits fichiers.
- Pour les petits fichiers, les données peuvent aussi être stockées directement dans les méta-données à la place de la table des blocs.
- On peut partitionner les blocs en groupes et mettre dans chaque groupe les informations sur la gestion des blocs et des inodes, afin d'augmenter la localité et de limiter les dommages en cas de corruption (logiciel ou matériel).
- Utilisation de B-trees plutôt que de listes d'association pour représenter les répertoires.

Journalisation

- Augmenter la robustesse en cas de panne : on peut reconstruire le dernier état cohérent précédent la panne (mais certaines écritures, les dernières, seront ignorées).
- Augmenter la localité donc la rapidité d'accès.

Cela se fait en journalisant les méta-données et éventuellement les données. Exemples : `UFS`, `ReiserFS`, `ext3` (NB : `ext3` est implémenté au dessus de la structure de `ext2`).

- Toutes les écritures sont contigües dans une zône `journal`.
- Elle maintient une table de redirections des anciens blocs.
- Les anciens blocs sont libérés.
- Les blocs non libérés sont recopiés pour former une nouvelle zône journal. (Sorte de GC à recopie du disque).

Systèmes de fichiers de type base-de-donnée

(c'est assez tendance)

- On abandonne la struture de répertoires.
- On la remplace par une struture de base de donnée : un ou plusieurs fichiers sont le résultat d'une requête.

Systèmes de fichiers distribués

Très complexe à implémenter : problèmes de synchronisation et de cohérence.

Difficiles à utiliser : quelles sont les garanties ?

Système de fichiers pour autres médias

CD-ROM, DVD

- Ils ne sont pas ré-écrivables (incrémentalement).
- Les blocs peuvent êtres alloués de façon contigües.
- Permet une lecture séquentielle très rapide.
- Seek possible mais moins rapide.

Les données sur le disque
ne sont pas manipulées directement

- Elles sont copiées en mémoire ;
- Puis lues et écrites en mémoire, par blocs ;
- Et sauvegardées sur le disque, paresseusement.

Les données sur le disque ne sont pas manipulées directement

Simplicité d'implémentation

- Le protocole de transfert entre la mémoire et le disque ignore la structure.
- La structure est analysée en mémoire.
- La représentation en mémoire ne dépend pas de l'architecture sur le disque. La même structure interne est utilisée pour des systèmes sur disque différents (sauf le transfert et la localisation des blocs).

Les données sur le disque ne sont pas manipulées directement

Simplicité d'implémentation

Économies de transfert (coûteux par défaut)

- Une donnée transférée en mémoire peut être lue et relue.
- Mêmes les écritures sont retardées : des données peuvent être modifiées plusieurs fois en mémoire avant d'être écrites sur le disque.

Les données sur le disque
ne sont pas manipulées directement

Simplicité d'implémentation

Économies de transfert (coûteux par défaut)

Écritures sur disques

- En principe, la sauvegarde sur disque n'a lieu que sur demande explicite (sync) ou lorsqu'il n'y a plus de place en mémoire.
- Pour éviter de perdre des données en cas de panne (*e.g.* courant), un démon exécute `sync` à intervalles réguliers.
  N.B. on peut arrêter le sync... et le disque devient silencieux !

Les caches

**Les données sur le disque
ne sont pas manipulées directement**

Simplicité d'implémentation

Économies de transfert (coûteux par défaut)

Écritures sur disques

Ne pas confondre (n'ont rien à voir entre eux)

- Caches (inodes, blocs),
- Mémoire cache et
- Mémoire virtuelle

Parenthèse (cours suivants)

- Même sur une machine mono-processeur, plusieurs processus tournent en parallèle (de façon entrelacée).
- Le système travaille toujours pour/sous le compte d'un processus.
- Le code système peut donc lui-même être entrelacé entre plusieurs processus :
  - Une ressource système peut être utilisée par un processus qui est mis en attente (temporairement)
  - Un processus qui a besoin de cette resource doit se mettre en attente.

  Il faut utiliser des verrous et faire attention aux situtations d'inter-blocage.

Structure d'un cache

- Il contient l'information sur le disque,
- Plus des informations système, non recopiées sur le disque :
  - Identification : numéros de device et de bloc ou d'inode
  - État du cache : `Free`, `Busy` ou `Delayed_write` + un booléen `requested`.
  - État des données : des booléens `valid`, `modified`, `old`.
  - Des informations pour l'accès aux caches.
  - Des informations pour leur recyclage.
- Les caches sont alloués en fonction de la mémoire libre à partager avec la mémoire des processus (réglage délicat).

Inode Information sur le disque ($\approx$ type `stat`) enrichie.

Blocs Le bloc tel qu'il est sur le disque (suite d'octets).

Recherche d'un bloc en mémoire

Problème

À un bloc identifié par `bid` (ses numéros de device et de bloc), trouver ce bloc dans le cache ou en allouer un nouveau sinon.

Structure de recherche

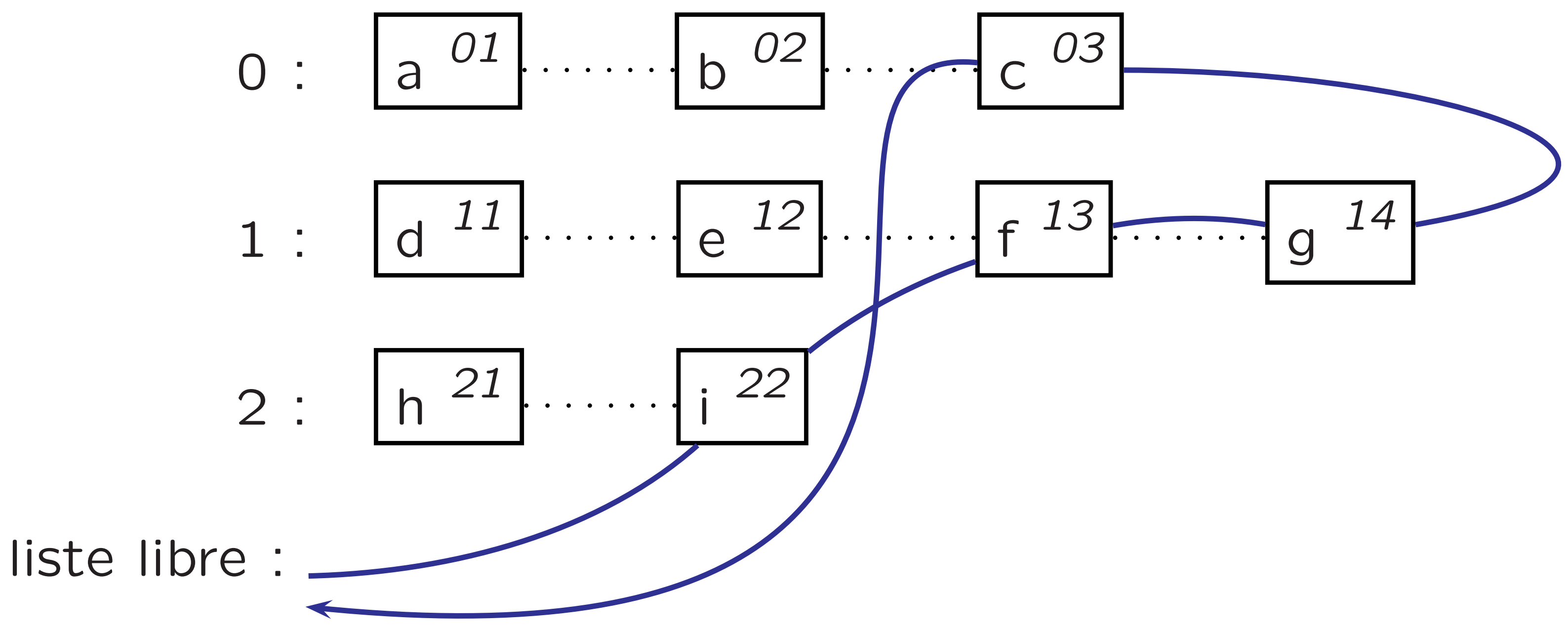
- Clé de hash sur `bid`.
- Liste doublement chaînée des blocs dans chaque baquet.

Astuces

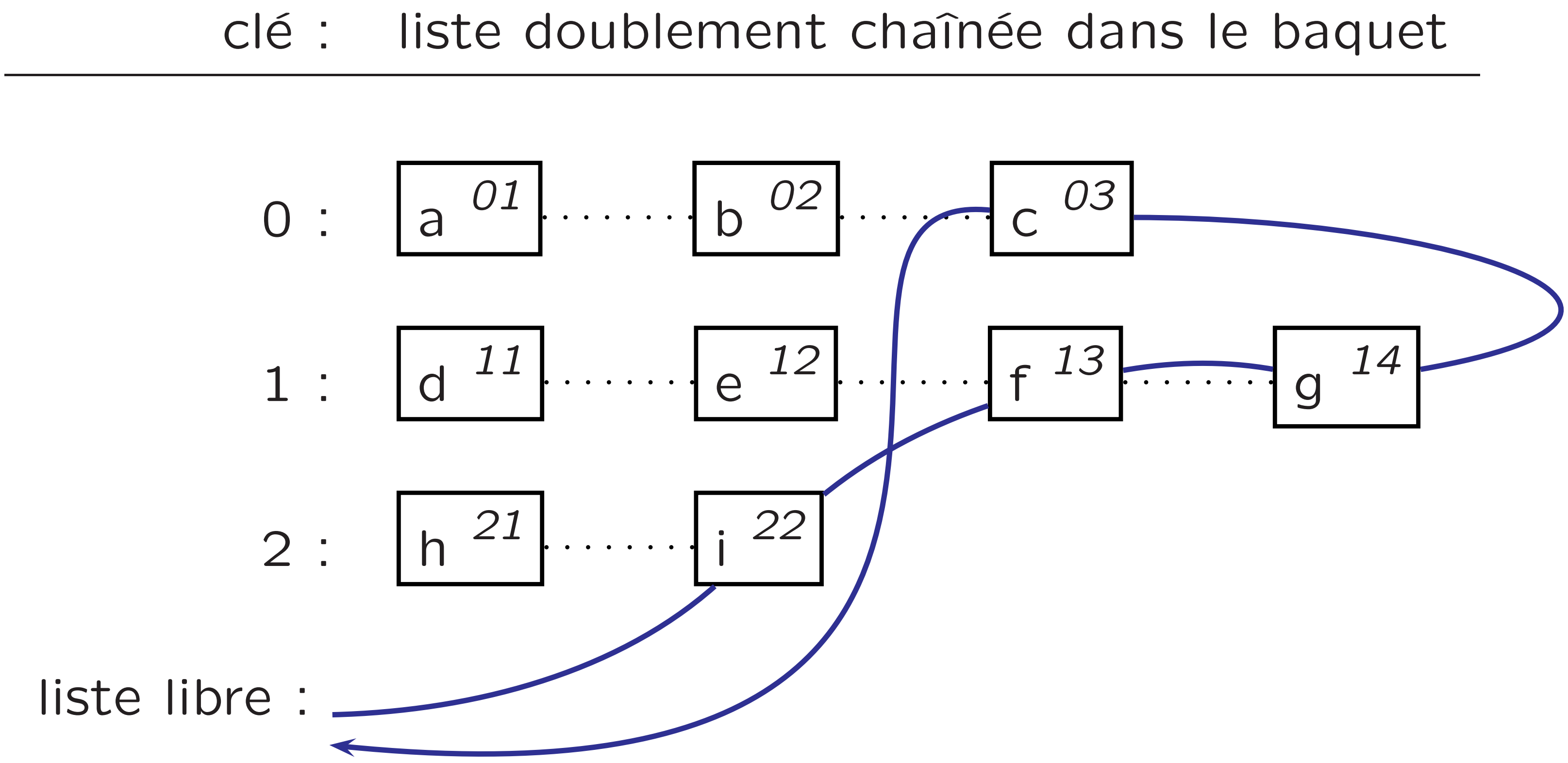
- Les caches libres sont dans une liste doublement chaînée, mais laissés dans leur baquet :
  - $\in$ leur baquet : on peut les réutiliser (pour le même bloc).
  - $\in$ liste libre : on peut les recycler (pour un autre bloc).
- Écritures paresseuses : effectuées lorsqu'un bloc est réutilisé. (état delayed-write). Rangés par ordre d'ancienneté.

Allocation des caches

clé : liste doublement chaînée dans le baquet

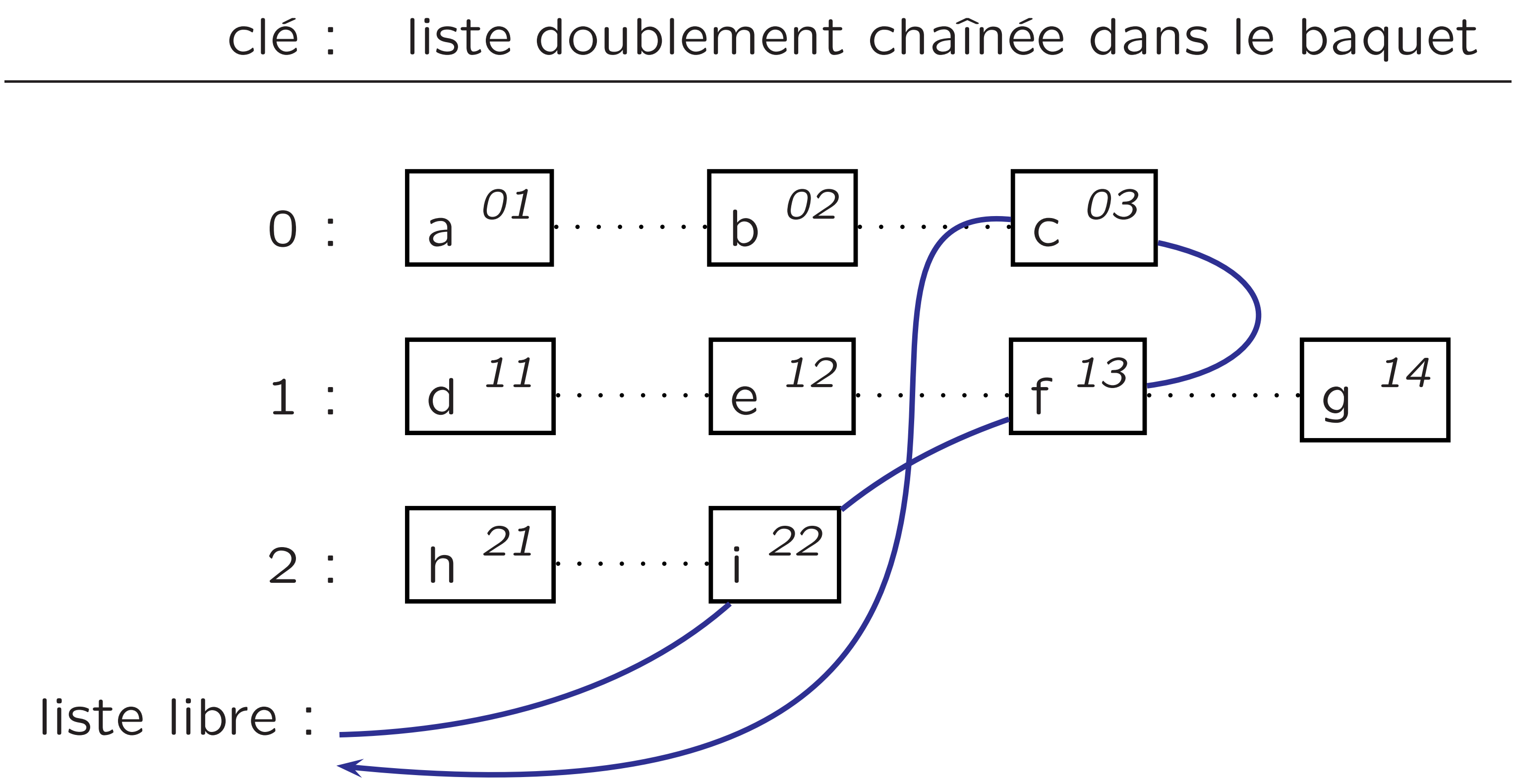


Allocation des caches



Allocation du bloc *14* :

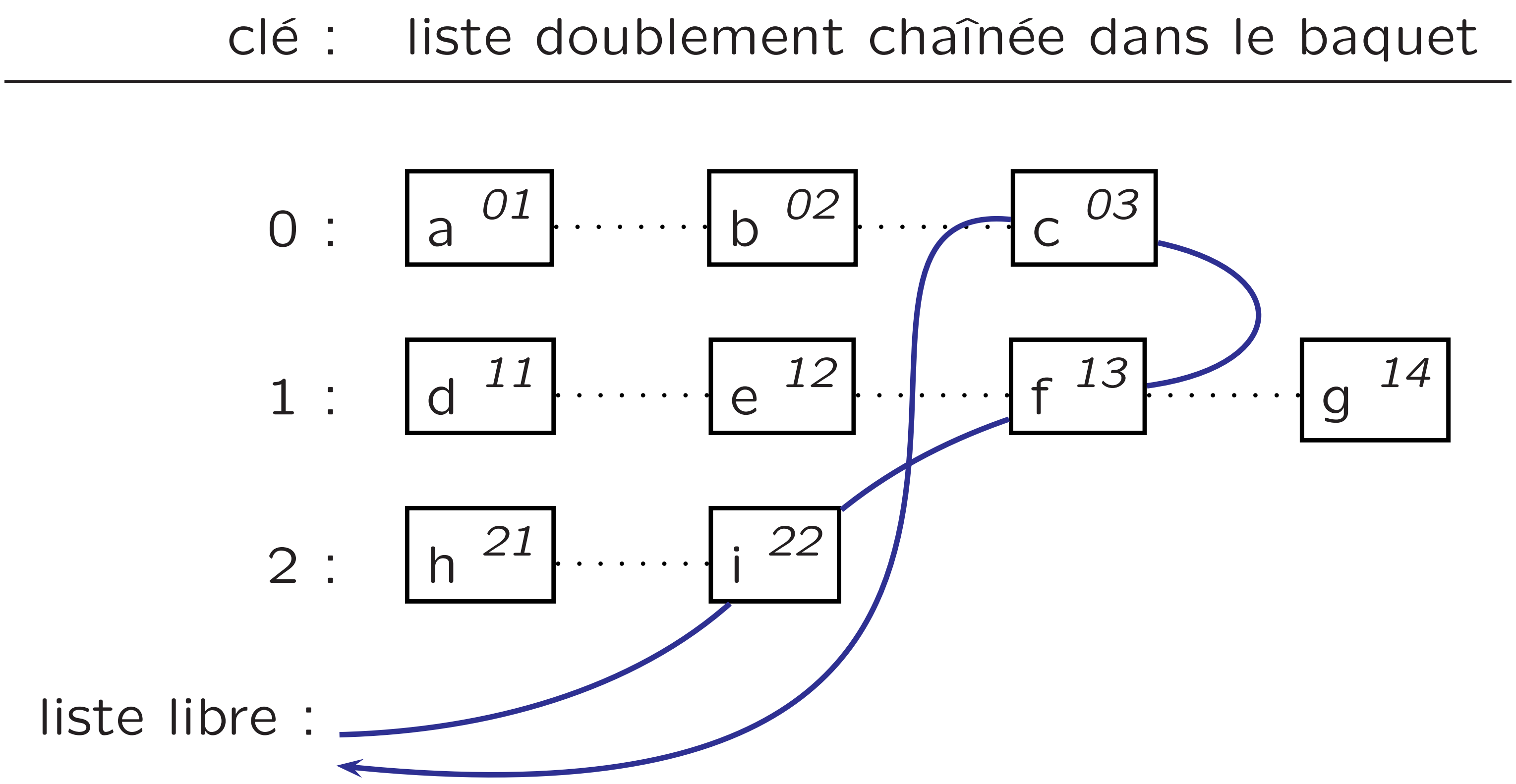
Allocation des caches



Allocation du bloc *14* : $\longrightarrow$ g rétrouvé

Il est libre mais encore dans son baquet et ses données sont valides.

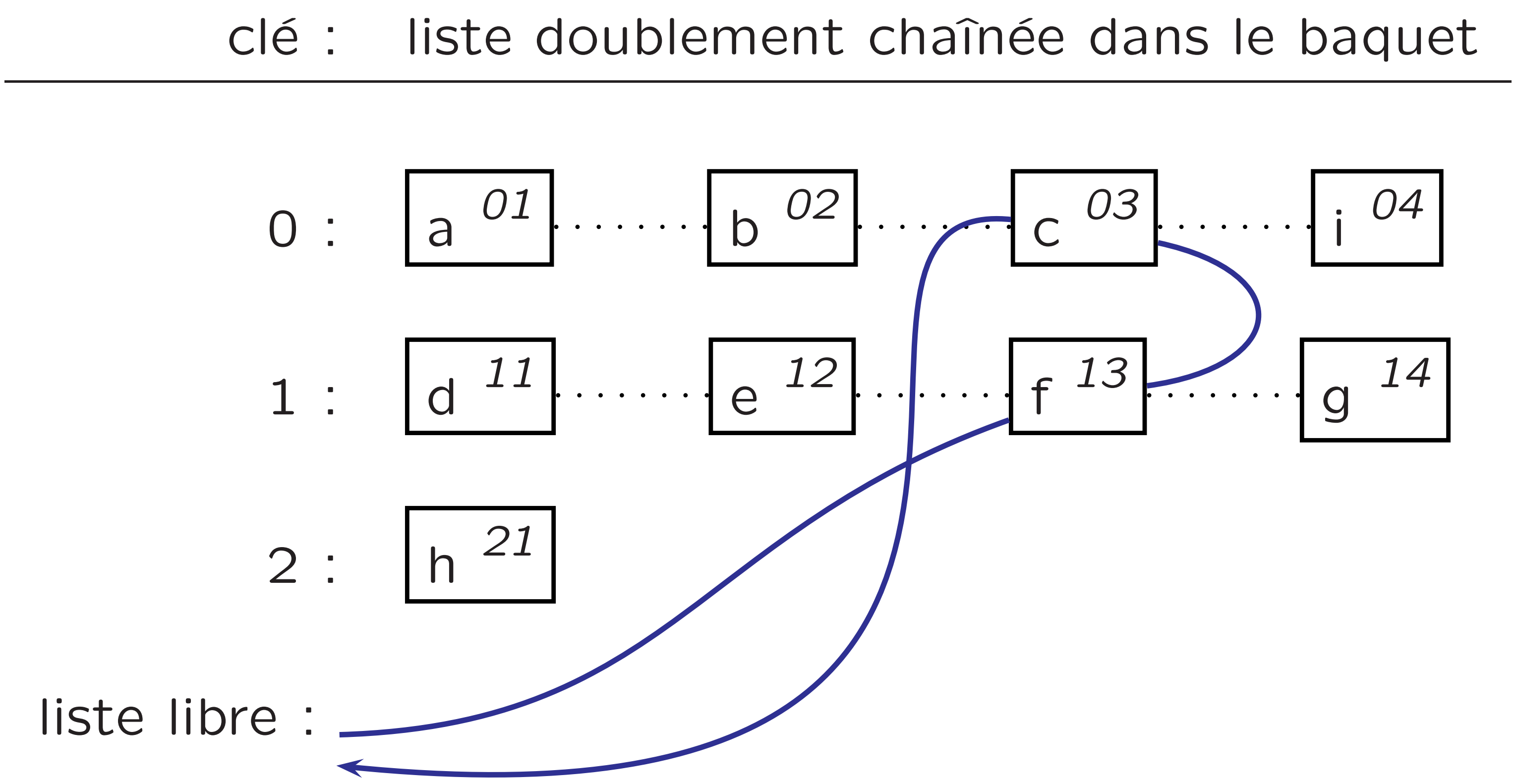
Allocation des caches



Allocation du bloc *14* : $\longrightarrow$ g rétrouvé

Allocation du bloc *04* :

Allocation des caches



Allocation du bloc *14* : ⟶ g rétrouvé

Allocation du bloc *04* : ⟶ i recyclé

Il n'est pas trouvé dans son baquet, on alloue un nouveau bloc pris dans la liste libre qui n'est pas `Delayed_write`.

Concurrence

- Un bloc peut être utilisé par plusieurs processus.
- Une lecture ou écriture est faite de façon asynchrone, si possible (on passe alors à un autre processus)
- Un block en service est dans l'état `Busy` ; par exemple le système est en train de le lire sur le disque ou de faire le transfert vers la mémoire utilisateur.
- Il ne doit pas le rester arbitrairement longtemps, en particulier être relâché avant le retour de l'appel système.
- Si un bloc est «Busy», on se met en attente sur une condition «Block-Not-Busy».
- À un moment le bloc sera libéré et le processus en attente on sera réveillé.

Trouver un bloc (approximation)


```
let rec getblk bid (* find and locks a buffer for bid *) =
  try match Hashtbl.find bid with
  | { status = Busy; } as b ->
      wait_for ((* b.status != Busy *)); getblk bid
  | b -> b.status <- Busy; remove_from free_blocks b; b
  with Not_found ->
    match !free_blocks with
    | [] ->
        wait_for ((* !free_blocs != [] *)); getblk bid
    | b::_ ->
        remove_from free_blocks b;
        match b.status with
        | Delayed_write -> async_bwrite b; getblk bid
        | _ -> Hashtbl.remove b.id b; Hashtbl.add bid b;
               b.valid <- false; b.status <- Busy; b;;
```

Libérer un bloc (block release)


```
let brelse b (* b is locked, frees it *) =
  wakeup_waiting_for ((* !free_blocks != [] *));
  wakeup_waiting_for ((* b.status != busy *));
  raise_execution_level block_interrupts;
  if b.valid && not b.old
  then enqueue_end free_blocks b
  else enqueue_beginning free_blocks b;
  lower_execution_level allow_interrupts;
  b.status <- if b.modified then Delayed_write else Free;;
```

Libérer un bloc (block release)


```
let brelse b (* b is locked, frees it *) =
  wakeup_waiting_for ((* !free_blocks != [] *));
  wakeup_waiting_for ((* b.status != busy *));
  raise_execution_level block_interrupts;
  if b.valid && not b.old
  then enqueue_end free_blocks b
  else enqueue_beginning free_blocks b;
  lower_execution_level allow_interrupts;
  b.status <- if b.modified then Delayed_write else Free;;
```

Attention ! contrôle des interruptions pour `getblk` aussi.

Bloquer les interruptions avant la manipulation des listes libres

- le traitement d'une interruption pourrait appeler `brelse` et corrompre `free_blocks`.
- modification similaire à faire dans `remove_from` in `getblk`.

Race conditions : exemple

Le processus B cherche un bloc et le trouve dans l'état `Busy`.

Scénario possible

- A alloue un bloc, le lit sur le disque et se met en attente.
- B cherche ce bloc et se met obligatoirement en attente (ne peut pas réallouer un bloc existant) et met le bloc `resquested`.
- A libère le bloc : comme il est `requested`, il réveille les processus en attente sur ce bloc ou sur un bloc libre.
- C peut prendre ce bloc avant B.
- B à son réveil doit vérifier
  - que le bloc est libre (*c.f.* cours sur `Mutex` et `Condition`)
  - que le bloc est le bon : C a pu le recycler pour un autre bloc.
  - B doit donc rechercher le bloc à nouveau.

Liveness

Un cache ne peut pas rester bloqué indéfiniment

- Un processus ne peut bloquer qu'un seul cache.
  Il y a beaucoup plus de caches que de processus.
- Le mode utilisateur n'a pas directement accès aux caches.
- Lorsqu'un appel système retourne, le processus ne peut pas laisser de cache à l'état `Busy`.
- Un appel système peut être suspendu, avec un bloc `Busy` seulement en attente d'une lecture-écriture sur disque.
- Le temps est forcément borné. (Un pilote de disque doit se prémunir contre une erreur matérielle qui le laisserait bloqué.)

Un processus pourrait être affamé

Par un autre qui prend toujours priorité sur lui.
Possible, mais improbable, problème d'ordonancement.

Lire un bloc


```
let bread bid =
  let b = getblk (* retourne le bloc dans l'état Busy *) in
  if not b.valid_data then
    initiate_disk_read b;
    wait_for ((* disk_read_completed b *));
  end;
  b;;
```

Variante `bread_ahead`

- Lecture du bloc demandé si nécessaire
- Lecture asynchrone du bloc suivant s'il n'est pas en mémoire. Attention ! à la complétion de la lecture (asynchrone), il faudra libérer le bloc (passer sont statut à `Free`).
- Heuristique (`read`) : si deux blocs d'un fichier sont lus séquentiellement, le noyau suppose que le suivant sera lu.

Écriture d'un bloc

Synchrone

```
let bwrite b =
  initiate_disk_drive;
  wait_for ((* disk_write_completed *));
  brelse b;;
```

Asynchrone

```
let async_bwrite b =
  initiate_disk_drive
  if b.status = Delayed_write then
  mark_put_ahead_of_free_list b;;
```

Le bloque sera relaché par `brelse` lorsque le disque signalera la complétion du write asynchrone.

Allouer et libérer un inode en mémoire

Mécanisme similaire

Quelques différences

- Un bloc est libre s'il n'est pas occupé.
- Un inode a un compteur de références `refcount`
  - indique le nombre de fois qu'il est référencé par un processus
  - ne pas confondre avec `nlink` (nombre de liens durs)
- Un inode est récupérable si `refcount` $= 0$.
  - Il peut être réutilisé pour cacher un autre inode du disque.
  - Sinon, s'il n'est pas `Busy` et si `refcont` > 0, un autre processus peut l'utiliser pour le même fichier, mais pas le désallouer/réallouer. Opérations `lock` et `unlock`.
  - *e.g.* Si un processus ouvre un descripteur sur un fichier, puis fait unlink sur le fichier, alors (*e.g.*) `nlink` $= 0$ mais `refcount` > 0 tant que le descripteur reste ouvert.

namei

- Trouver l'inode associé à un chemin dans le système de fichiers
- C'est l'opération la plus courante sur le système de fichiers.
- Nécessite des fonctions auxiliaires...
  - `find_dir_entry`
    Trouver une entrée de répertoire dans un répertoire...
    - `bmap`
      Trouver le numéro du bloc de donnée correspondant à une position dans un fichier.

bmap

- Calcule l'identité du bloc à une position dans un fichier.
- Réponse immédiate pour les blocs directement référencés
- Il faut lire les blocs intermédiaires pour les blocs indirectement référencés.

```
let bmap inode offset =
  if offset > inode.st_size then raise Not_found else
  let k = offset / block_size in
  if k < number_direct_blocks then inode.direct.(k)
  else if k < number_of_simple_indirect_blocks then
    let b = bread i.simple_indirect in
    let k' = k - number_direct_blocks in
    let bid = read_nth_inode_field k' b.data in
    brelse b; bid
  else (* indirection double et triple *)
```

find_dir_entry

- On parcourt les blocs jusqu'à trouver l'entrée recherchée...
- Pour chaque bloc, on parcourt les entrées jusqu'à trouver celle recherchée.
- C'est une erreur si on arrive à la fin sans l'avoir trouvée.
- Facile... mais il faut allouer/lire les blocs (imaginer le code déroulé par `bread`...) et bien relâcher les blocs ouverts !

Une entrée de répertoire est un triplet :

- inode : un entier ;
- len : un entier, la taille de l'entrée ;
- name : le nom de l'entrée, éventuellement suivie de \000.

(Lorsqu'on retire une entrée, on la remplace pas des \000 et on ajuste la longueur de l'entrée précédente.)

(Les anciens systèmes de fichiers utilisaient des tailles fixes pour les noms de fichiers.)

find_dir_entry


```
let rec find_dir_entry dir_inode name =
  let rec find offset b pos =
    if pos >= dir_inode.size then raise Not_found else
    if pos < block_size then
      let k, len, n = dir_entry b pos in
      if n = name then (brelse b; k) else find b (pos + len)
    else begin
      brelse b;
      let offset = offset + block_size in
      if offset > dir_inode.st_size then raise Not_found;
      let b = bread (bmap dir_inode offset) in
      find offset b 0
    end in
  find 0 (bread dir_inode.direct.(0)) 0;;
```

name_inode (esquisse...)


```
let rec namei p =
  let rec find w = function
    | [] -> w
    | "." :: tail -> find w tail
    | ".." :: tail when && w = root_inode -> find w tail
    | name :: tail  ->
        (* check access rights, etc. *)
        if w.stat.st_kind != S_DIR then raise Error;
        try
          let wid' = find_dir_entry w name in
          iput w; find (iget wid') tail
        with Not_found -> iput w; raise Error in
  let wid = if is_relative p then cwd_id else root_id in
  find (get_inode wid) (split "/" p);;
```

Résumé des opérations de bas niveau

	blocs	inodes
Caches	getblk brelse bread bwrite	iget iput bread bwrite
Adressage	bmap	namei

Autres opérations élémentaires

Manipulation de noeuds et blocs dans la structure de fichiers.

Allocation	alloc_block free_block	alloc_inode free_inode

Ouverture d'un fichier

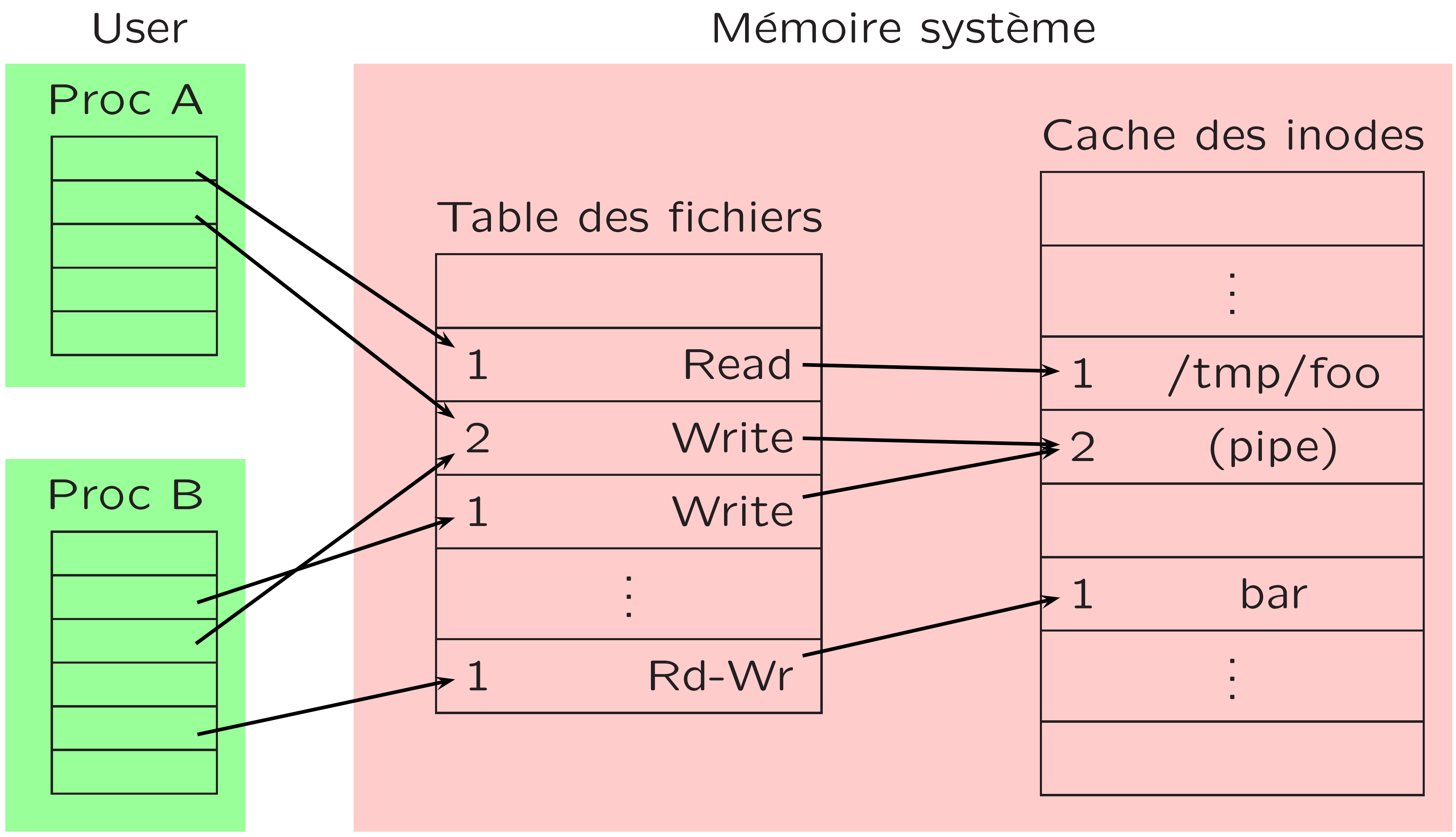
Les descripteurs (coté utilisateur)

- Seulement un tableau qui pointe vers des tables systèmes

Les tables de fichiers (coté système)

- Pour chaque ouverture de fichier, il y a une nouvelle entrée.
- Une entrée mémorise :
  - le noeud du fichier (pointe vers un inode en mémoire),
  - le pointeur courant de lecture,
  - des informations sur le mode d'ouverture (Read/Write).
- Cette table est partagée :
  - Ouverture, puis fork : les deux processus ont la même entrée dans la table.
  - Ouverture d'un fichier, puis ouverture à nouveau : un même processus à plusieurs entrées sur le même inode.

Fichiers : descripteurs, tables et inodes



Exemple 1 : open


```
let open filename =
  let inode = namei filename in
  (* namei lève Error si filename est inacessible *)
  let f = new_file_entry inode;
    f.count <- 1;
    let user_descr = new_user_desc file_entry in
    if truncate_option then free_all_blocks inode;
    unlock inode;
    user_descr
```

Exemple 2 : read


```
let read descr buf pos len =
  let file_entry = user_descr_table.(desc) in
  copy buf, pos, len file_entry.cur into u_area ; u_area.res <- 0;
  let inode = file_entry.inode in
  lock inode;
  begin try while u_area.len > 0 do
    let b, pos, left = bmap inode u_area.cur in
    let n = max u_area.len left in
    bread b; memcpy b.data pos n buf pos;
    u_area.(res <- res + n; pos <- pos + n; len <- len - n);
    brelse b
  done with Not_found (* bmap *) -> ()  end;
  unlock inode;
  file_entry.cur <- file_entry.cur + u_area.res;
  u_area.res;;
```

Exemple 2 : correction

Modifier le code pour ne pas bloquer s'il y a déjà des données disponibles.

(Retourner moins de données que celles demandées).

link : attention aux deadlock

- Manipule deux chemins indépendant...
  Exemple de scénario («[» = lock and «]» = unlock) :

process	source	target
A	a/b/c/d	e/f/bar
verrous	a[a] b[b] c[c] d[	e[e] f
blocage	possède d	attend f
B	e/f	a/b/c/d/gnu
verrous	e[e] f[f	a[a] b[b] c[c] d !
blocage	possède f	attend d

- Relacher le verrou sur la source après avoir incrémenté son compteur de référence et avant de rechercher la destination.

Les tuyaux

- Ils ont un inode en mémoire qui est inaccessible dans la hiérarchie, (sauf pour les tuyaux nommés).
- les données sont stockés dans des blocs comme pour les fichiers (seulement les blocs directs)
- Le système gère les blocs de données comme un tableau circulaire.

Risque de corruption du disque

- Le disque est intègre au formattage (on peut le vérifier après le formattage, ou après une longue période).
- Il peut être corrompu après un arrêt brutal de la machine si certaines données ont été écrites et d'autres non.
  Exemple de scénario :

  1. Un block est retiré d'un `inode`
  2. CRASH
  3. Le block n'a pas été mis dans la liste libre.

Risque de corruption du disque

Invariants

- Tous les blocks doivent être atteignables : par un inode ou dans la liste libre.
  - un bloc non atteignable : ses données sont perdues, car on ne sait pas où le rattacher (Fichier `lost+found` créé par `fsck`)
- `nlink` indique le nombre de fois qu'un inode est atteignable.
  - si le compteur est trop petit... les données ne sont pas encore perdues, mais elles risquent de l'être prochainement.
  - si le compteur est trop grand, l'inode ne sera plus recyclé.
- Fichier corrompu : le système de fichier n'est pas endommagé, mais un bloc à été recyclé (puis réutilisé) à tord.

Risque de corruption du disque

Invariants

Comment se prémunir

- On ne peut pas (les écritures des invariants ne peuvent pas être simultanées), sans utiliser des transactions (journal).
- On peut chercher à ordonner les écritures pour que les situations corrompues soient récupérables.
- *e.g.* Le bloc b du fichier f est désalloué puis réutilisé pour le bloc b du fichier g.
  - retirer b du fichier f
  - écrire le bloc b
  - mettre le bloc dans le fichier g.
- Écrire les inodes sur le disque régulièrement.

Un inode spécial qui fait une «redirection»

- contient un pointeur vers un autre système de fichier.
- il faut modifier les algorithmes (surtout `namei`) pour traiter le cas de mount.
- Cela suppose que les systèmes de fichiers sont du même type.

Abstaction du système de fichiers

- Des systèmes de fichiers de types différents ont des implémentations différentes.
- EXT2, EXT3 : (petites différences de représentation sur disque, des fonctions de bas niveaux)
- FAT : groses différences... certains opérations n'existent pas.
- NFS : très grosses différences.
- LUFS (Linux Userland FileSystem)...

Système de fichiers virtuel

- On abstrait l'implémentation des opérations.
- On ajoute une couche intermédiaire : les inodes génériques.
  Ce sont des paires cohérentes comportant

  1. opérations pour le type de fichier considéré.

ext2	ext3	nsf
`ext2_open`	`ext3_open`	`nfs_open`
`ext2_close`	`ext3_close`	`nfs_close`
`ext2_read`	`ext3_read`	`nfs_read`
`ext2_write`	`ext3_write`	`nfs_write`
⋮	⋮	⋮

  2. inode réel dans un système de fichier concret.

  Comme des objets ou fermetures (cf. cours de compilation)

```
oper ginode args  →  ginode.oper ginode.inode args
```

Le sous-système de fichiers : une abstraction puissante

Vision de haut niveau simple et uniforme

- Un fichier est une séquence linéaire
- Presque tout passe par un descripteur

Des structures intermédiaires efficaces

- minimiser les lectures écritures sur disque (haut niveau)
- opérations asynchrones qui optimisent le média (bas niveau)

Des couches empilées

- les drivers de disque (lectures/écritures de bloc)
- les appels systèmes pour un type de système de fichiers.
- le système virtuel de fichiers.
- les opérations sur les caches.