Système de fichiers

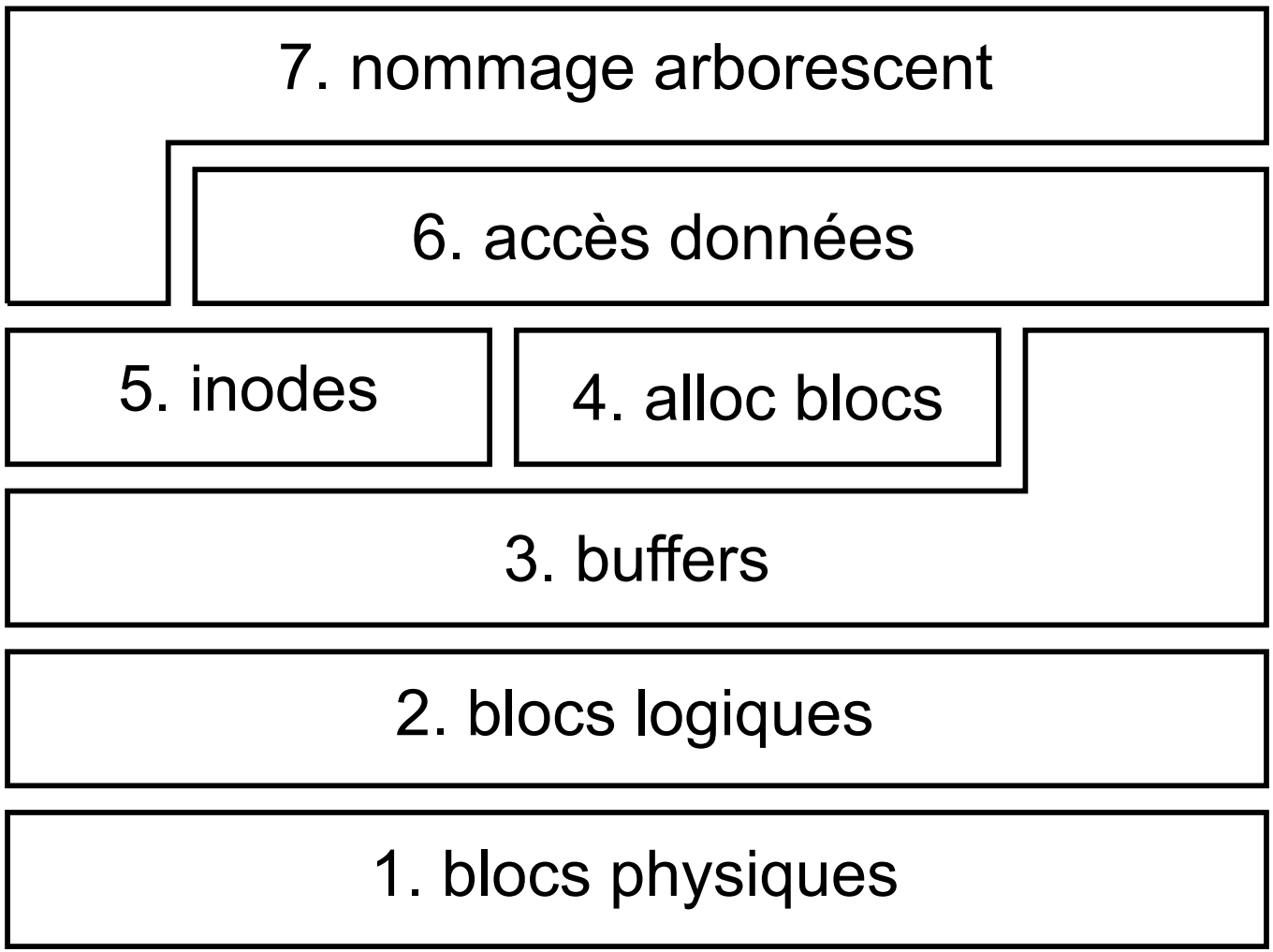
Philippe Quéinnec

18 mai 2020

Plan

1. Structure générale
2. Couches blocs
   - Couche blocs physiques
   - Couche blocs logiques
   - Couche buffer
   - Couche allocation des blocs
3. Couches inodes et accès
   - Couche inode
   - Couche accès données
4. Couche nommage

Structure


7. nommage arborescent
6. accès données
5. inodes
4. alloc blocs
3. buffers
2. blocs logiques
1. blocs physiques

Plan

1. Structure générale
2. Couches blocs
   - Couche blocs physiques
   - Couche blocs logiques
   - Couche buffer
   - Couche allocation des blocs
3. Couches inodes et accès
   - Couche inode
   - Couche accès données
4. Couche nommage

Couche blocs physiques

Disque physique = périphérique de stockage permanent

- lecture ou écriture d'un bloc de taille fixée, dépendant du périphérique (généralement petit : 512 octets)
- adressage complexe (cylindre, piste, bloc)
- asynchrone : le noyau initie l'ordre, le contrôleur du périphérique signale la complétion de l'action (interruption)
- possibilité de réordonnancement des ordres par le contrôleur

Très grande diversité des paramètres : taille, adressage, comportement...

Couche blocs logiques

Bloc logique

- Bloc de plus grande taille (8 ko), indépendante du matériel.
- Adressage simple : entier ou ⟨numéro de groupe,sous-numéro⟩
- Rangement : blocs physiques consécutifs, ou sur pistes/plateaux différents mais régulier (fonction triviale n^o bloc logique $\rightarrow$ adresses des blocs physiques le constituant)
- Lecture/écriture synchrone : suspendre l'action en cours jusqu'à sa complétion $\rightarrow$ faire autre chose en attendant $\rightarrow$ réentrance du noyau

Couche buffer (cache des blocs)

Buffer vs bloc

Buffer = zone mémoire avec le contenu d'un bloc logique

Objectifs

- Limiter le plus possible les entrées-sorties effectives (très coûteuses)
- Mécanisme de cache pour conserver en mémoire les blocs lus
- Utilise une petite zone réservée + toute la mémoire disponible de la machine
- Écriture effective la plus retardée possible (accumulation des modifications)
- Risque d'incohérences en cas de plantage (blocs non écrits)

Couche buffer

État d'un buffer

- valide : possède un contenu
- modifié (*dirty*) : devra être écrit avant d'être réutilisé
- verrouillé : non libérable

Opérations

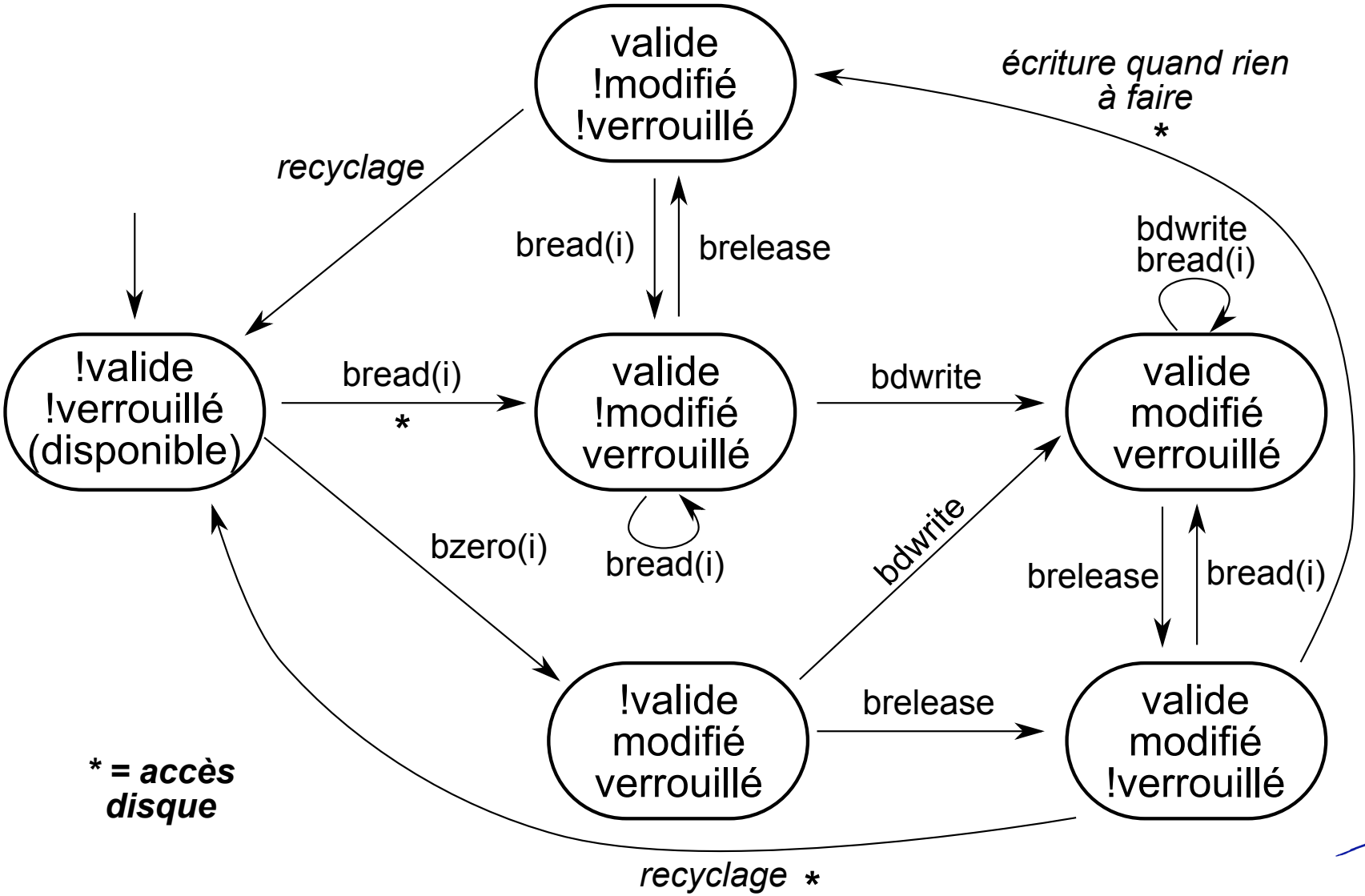
bread(i)→buf lecture d'un bloc dans un buffer

bzero(i)→buf initialisation à zéro d'un bloc dans un buffer

brelease(buf) relâche un buffer (le rend libérable)

bdwrite(buf) (delayed write) indique la modification du contenu d'un buffer : devra être écrit avant réutilisation

États d'un buffer



Couche allocation des blocs

Objectif : marquer qu'un bloc est utilisé

- Obtenir ou rendre disponible un bloc logique
- Information pérenne $\rightarrow$ rangée sur le disque
- Minimiser les accès disque nécessaires pour allouer / libérer un bloc
- Minimiser la mémoire et le temps nécessaire

FAT – File Allocation Table

- Solution historique
- Un nombre N fixe de blocs réservés qui contiennent au i-ième bit l'indication si le bloc i est libre ou pas.
- Un bloc de taille 1 Ko contient l'information pour 8192 blocs (= 8 Mo de contenu)
  Bien suffisant pour des disques de 10 ou 20 Mo (2 ou 3 blocs nécessaires, probablement cachés en mémoire)
- Disque de 1 To / bloc de taille 8 Ko
  $\Rightarrow$ nécessiterait 16384 blocs...

FAT

Allouer un bloc : balloc

```
for int numbloc = 0 to NbBlocsFAT-1 loop
   buf ← bread(numbloc);
   for int mot = 0 to taillebloc-1 loop
       for int bit = 0 to taillemot-1 loop
          if buf[mot] & (1 << bit) == 0 then
              buf[mot] ← buf[mot] | (1 << bit)
              bdwrite(buf)
              brelease(buf)
              return numbloc*mot*taillemot + bit
          endif
       endfor
   endfor
   brelease(buf)
endfor
return -1 // plus de bloc libre
```

Optimisation : se souvenir d'où commencer la recherche.

FAT (2)

Libérer un bloc : bfree(n)

```
// n : le bloc de données qui est libéré
// numbloc : le bloc contenant l'info d'allocation pour n
// nummot : l'octet contenant l'info d'allocation pour n
// but : le bit contenant l'info d'allocation pour n
numbloc ← n / (taillebloc * taillemot)
buf ← bread(numbloc)
nummot ← (n % taillebloc) / taillemot
bit ← (n % taillebloc) % taillemot
buf[nummot] ← buf[nummot] & (0 << bit)
bdwrite(buf)
brelease(buf)
```

UFS - Unix File System

- Les numéros des blocs libres sont rangés dans. . . des blocs libres !
- Les blocs de blocs libres sont chaînés : liste des blocs de blocs libres
- Un bloc de blocs libres contient :
  - suivant : le numéro du bloc suivant dans la liste des blocs de blocs libres
  - un ensemble de numéros de blocs libres
- Le superbloc (bloc en position 0, contenant les informations générales sur le système de fichier) contient le numéro du premier bloc de la liste (bloc de tête)
- Le premier bloc de blocs libres est verrouillé en mémoire (`bread` au montage du système de fichier)
- Le noyau conserve en mémoire une liste partielle de blocs libres (le contenu du bloc de tête)

Unix (2)

Allouer un bloc : balloc

```
si la liste partielle de blocs libres n'est pas vide alors
    résultat ← extraire un élément de la liste
sinon
    -- le bloc de blocs libres en tête est épuisé
    résultat ← numéro de l'actuelle tête
    tête ← bloc suivant
    mettre à jour superbloc (bread+bdwrite+brelease)
    initialiser la liste partielle de blocs libres
        à partir du contenu du nouveau bloc de tête (bread)
finsi
```

Unix (3)

Libérer un bloc : bfree(b)

```
si la liste partielle de blocs libres n'est pas pleine alors
    ajouter b à la liste partielle courante
sinon
    copier la liste dans le buffer de l'actuel bloc de tête
    marquer ce buffer comme modifié (bdwrite)
    relâcher ce buffer (brelease)
    b devient le bloc de tête :
        tête ← b
        b.suivant ← l'ancienne tête
        b.ensemble de blocs libres ← vide
    mettre à jour superbloc (bread+bdwrite+brelease)
finsi
```

Plan

1. Structure générale
2. Couches blocs
   - Couche blocs physiques
   - Couche blocs logiques
   - Couche buffer
   - Couche allocation des blocs
3. Couches inodes et accès
   - Couche inode
   - Couche accès données
4. Couche nommage

Fichiers à plat

inode

Une inode (= un inœud) contient les attributs d'un fichier :

- propriétaire
- droit d'accès
- dates de création, modification...
- compteur de référence (pour la libération)
- la longueur du fichier (en octets)
- les numéros des blocs de données
- mais pas le nommage

Pas de structure : système de fichiers = ensemble à plat d'inodes

Rangement des données : allocation contiguë

- Rangement contigu des blocs de données : l'inode contient uniquement le numéro du premier bloc + la longueur du fichier
- Les blocs sont donc de N à N + (longueur / taille bloc)
- Problèmes :
  - Fragmentation : la destruction d'un fichier laisse des trous
  - Agrandissement d'un fichier : un fichier qui croît peut ne pas tenir dans la zone actuelle ⇒ recopie complète dans une autre zone plus grande
  - Performance : lecture anticipée possible selon le périphérique, mais pas de possibilité d'éparpiller les blocs sur le disque
- ⇒ systèmes de fichiers en lecture seule (CD-ROM, DVD-ROM)

Rangement des données : allocation chaînée

- Chaque bloc de données contient le numéro du bloc suivant. L'inode contient uniquement le numéro du premier bloc.
- Problèmes
  - Réservation de place dans le bloc $\Rightarrow$ taille utile $\neq 2^n$
  - Coûteux accès aléatoire au contenu (nécessité de lire tous les blocs précédents)
  - La destruction d'un fichier nécessite de le lire intégralement
- $\ll$ Optimisation $\gg$: conserver en mémoire la table d'allocation et de chaînage (FAT : File Allocation Table)
  Mais taille de la table pour les gros disques...

Rangement des données : Unix File System

Rangement non contigu d'un fichier

Fichiers de grande taille

Accès arbitraire au contenu d'un fichier

Quelques blocs directs, 1 bloc de simple indirection, 1 bloc de double indirection...

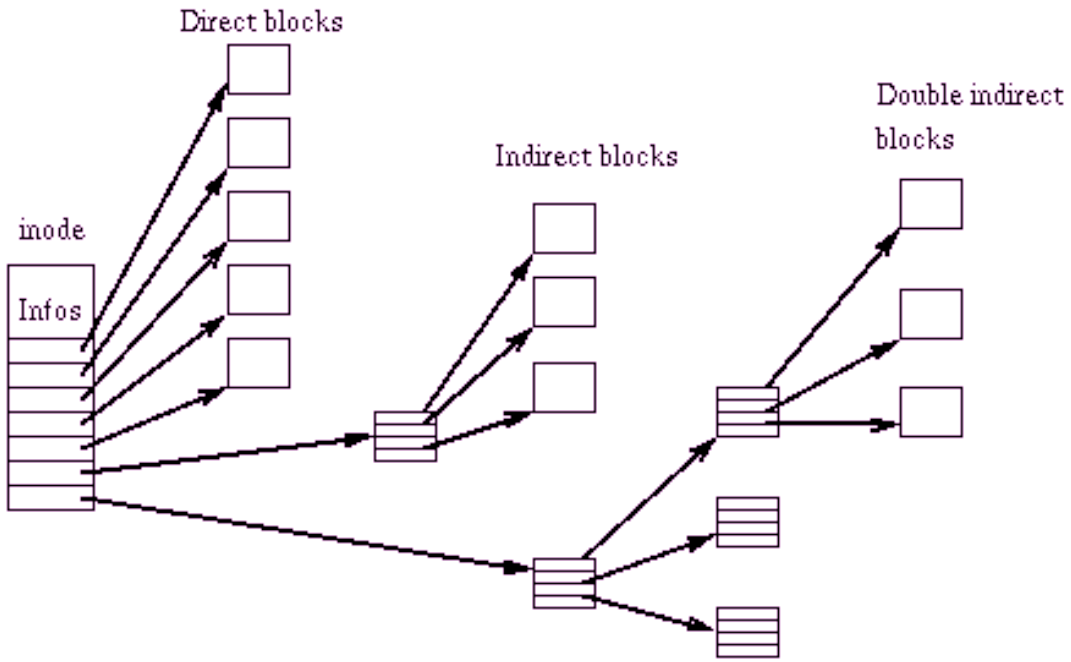
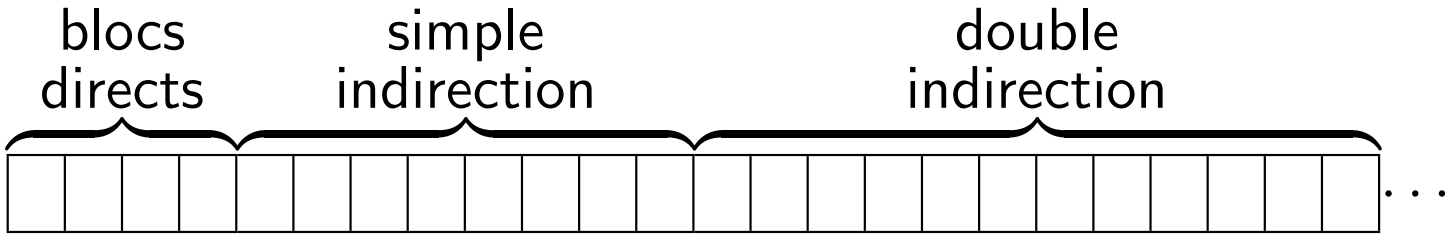


figure d'après wikipedia

Obtention d'un bloc de données d'une inode



bmap

- Entrée : inode, offset
- Sortie
  - numéro du bloc de données
  - offset dans ce bloc
  - numéro du bloc de données suivant (read ahead)
- nécessite de 0 à 3 lectures de blocs selon l'offset (d'où l'importance de la gestion des buffers)
- s'il n'existe pas encore de bloc pour contenir une donnée à cet offset, l'allouer (balloc) et allouer ou mettre à jour le(s) bloc(s) d'indirection

Allocation des inodes

- Hypothèse : peu d'allocations / libérations (à l'échelle des accès données)
- Réservation d'un ensemble de blocs consécutifs pour les inodes
- Pour chaque inode : 1 bit = libre / utilisée
- En mémoire (et dans le superbloc) une liste partielle d'inodes libres (liste de taille max N)

Allocation des inodes (2)

allocation d'une inode (ialloc)

```
si la liste d'inodes libres est vide alors
    parcourir les blocs d'inodes pour remplir la liste
        avec N inodes libres (bread, brelease)
finsi
extraire un élément de la liste
```

libération d'une inode (ifree)

```
// hypothèse : inode en mémoire
Libérer les blocs de données et d'indirections (bfree)
Mettre le bit d'utilisation de l'inode à 0
Marquer le buffer contenant cette inode comme modifié
si la liste d'inodes libres n'est pas pleine alors
      ajouter l'inode à la liste
finsi
```

Libération effective des inodes

Le choix de libérer une inode se fait par un ramasse-miette élémentaire basé sur un compteur de référence :

- L'inode contient un compteur de référence
- Quand l'inode est « ouverte » (obtention d'un descripteur de fichier pour l'accès aux données) : $+1$
- Quand le descripteur est fermé : -1
- Quand un nom est associé à l'inode (cf nommage) : $+1$
- Quand un nom est supprimé : -1
- Si le compteur passe à 0 : libération (ifree)

Lecture d'une inode

iget(inum)

```
numbloc ← inum / (nbre d'inodes par bloc)
buf ← bread(premier bloc d'inodes + numbloc)
dépl ← (inum % (nbre d'inodes par bloc))
                * (taille inode)
return &(buf[dépl])
```

Note : l'inode (et donc le buffer) est verrouillée en mémoire

iput(inum) (= irelease)

Déverrouille l'inode et éventuellement le buffer (compteur de référence)

Descripteurs de fichiers ouverts

Descripteur de fichier

- l'inode
- la position courante dans le fichier (offset)

Une table globale des descripteurs + une table locale par processus (qui pointe vers la table globale)

fdalloc(i)

Allocation dans la table globale d'un nouveau descripteur de fichier pour l'inode spécifiée.

Lecture

read(fd, adr, len) → nblu

```
fd → inode, offset courant
obtenir l'inode (iget)
à_lire ← min(longueur - offset, len)
tant que pas tout lu faire
    déterminer le numéro du bloc de données (bmap)
    lire le bloc (bread)
    copier dans *adr les données demandées
    relâcher le buffer (brelease)
    mettre à jour offset dans le descripteur
fin tq
déverrouiller l'inode (iput)
```

Et inversement pour `write`, en utilisant `bdwrite`.

Rq : on peut aussi garder l'inode verrouillée du `open` au `close`.

Plan

1. Structure générale
2. Couches blocs
   - Couche blocs physiques
   - Couche blocs logiques
   - Couche buffer
   - Couche allocation des blocs
3. Couches inodes et accès
   - Couche inode
   - Couche accès données
4. Couche nommage

Nommage : système de fichiers arborescent

Principe

À un nom (chaîne de caractères), faire correspondre une inode

Répertoire

- Un répertoire est un ensemble d'associations nom $\rightarrow$ inode
- Un répertoire est un fichier (quasiment) comme un autre
- Organisation arborescente avec ".." = répertoire parent

Chemin d'accès

Chemin d'accès = suite de noms séparés par /

Note : différents noms peuvent correspondre à la même inode (liens durs par exemple)

Résolution d'un chemin

namei(chemin) → inode

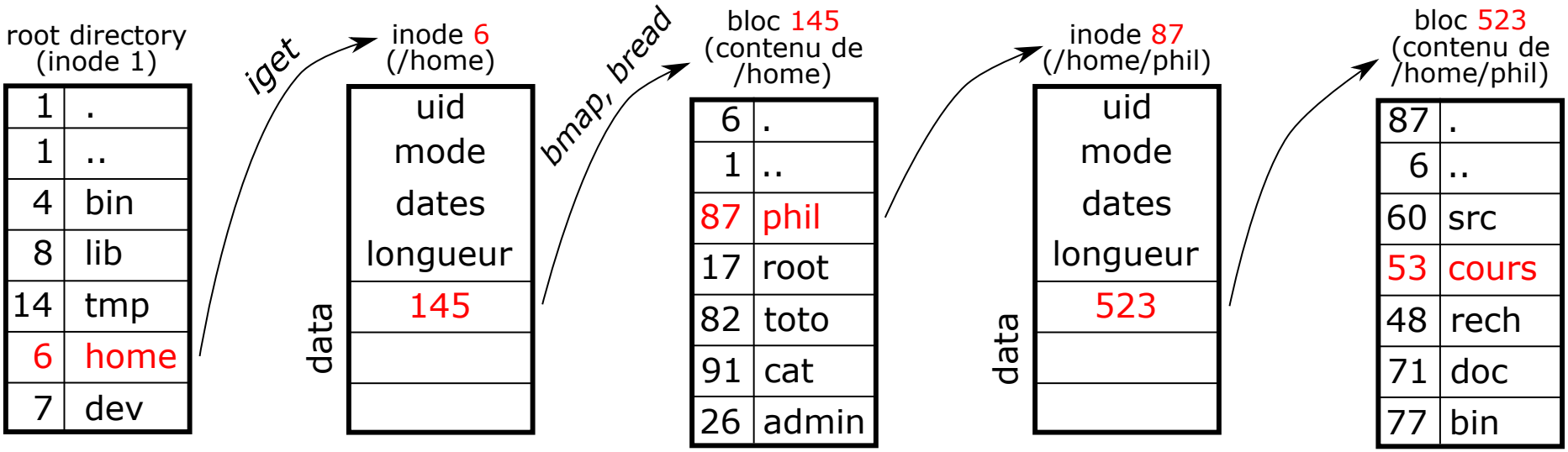
```
si le chemin commence à la racine / alors
  in ← inode racine (d'après superbloc) (iget)
sinon
  in ← inode du répertoire courant du proc. demandeur (iget)
finsi
tant qu'il reste des composants dans le chemin faire
    obtenir le composant suivant
    vérifier que in est un répertoire
    lire les données de in (bmap, bread, brelease) jusqu'à
            trouver une entrée égale au composant recherché
    si trouvé alors
        libérer in (iput)
        in ← inode trouvée (iget)
    sinon
        retourner échec
fintq
retourner in
```

Accès à un fichier

open(chemin) → fd

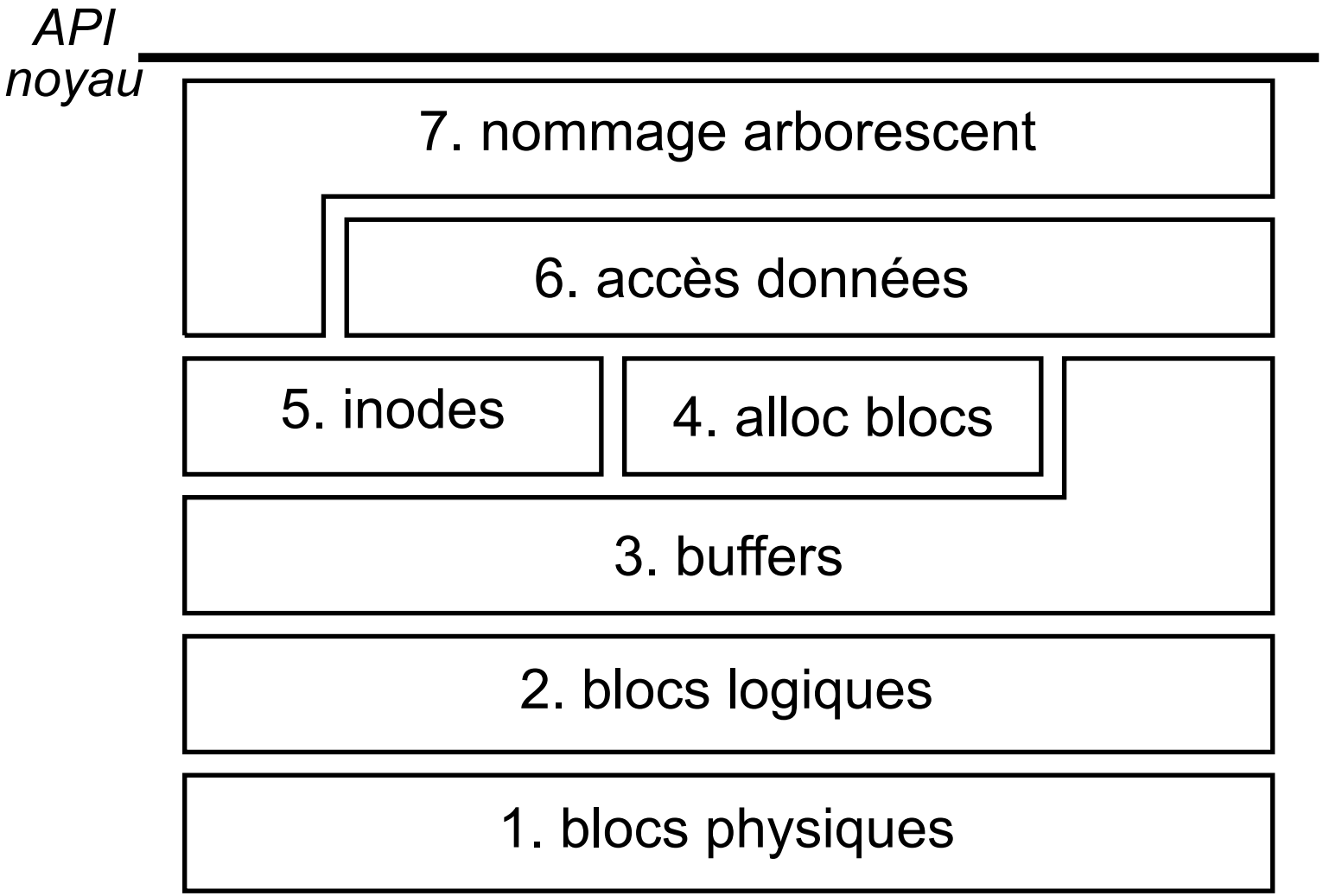
```
si ouverture fichier déjà existant alors
    trouver l'inode correspondant au chemin (namei)
    s'il faut tronquer le fichier alors
        parcourir les blocs de données et blocs indirects
            et les libérer (bfree)
    finsi
sinon
    trouver l'inode du répertoire parent (namei)
    allouer une nouvelle inode (ialloc)
    ajouter l'entrée (nv nom, nv inode)
                (bmap, bread, bdwrite, brelease)
    libérer l'inode du répertoire (iput)
finsi
obtenir un nouveau descripteur de fichier (fdalloc)
déverrouiller l'inode (iput)
retourner le descripteur de fichier
```

Synthèse



Résolution de /home/phil/cours -> inode 53

Conclusion


API
noyau
7. nommage arborescent
6. accès données
5. inodes
4. alloc blocs
3. buffers
2. blocs logiques
1. blocs physiques

Partage des rôles

Que veut l'utilisateur :

- Persistance des données (arrêt, crash)
- Vitesse d'accès
- Grande capacité de stockage
- Partage et protection
- Facilité d'utilisation

Matériel

- Persistance : disque non volatile
- Vitesse : accès aléatoire
- Capacité : ↑
- Partage : disque amovible

Système d'exploitation

- Persistance : redondance
- Vitesse : cache
- Partage/protection : droits utilisateur
- Facilité : IHM