Systèmes centralisés : TD3 - Fichiers

Communication par flots de données

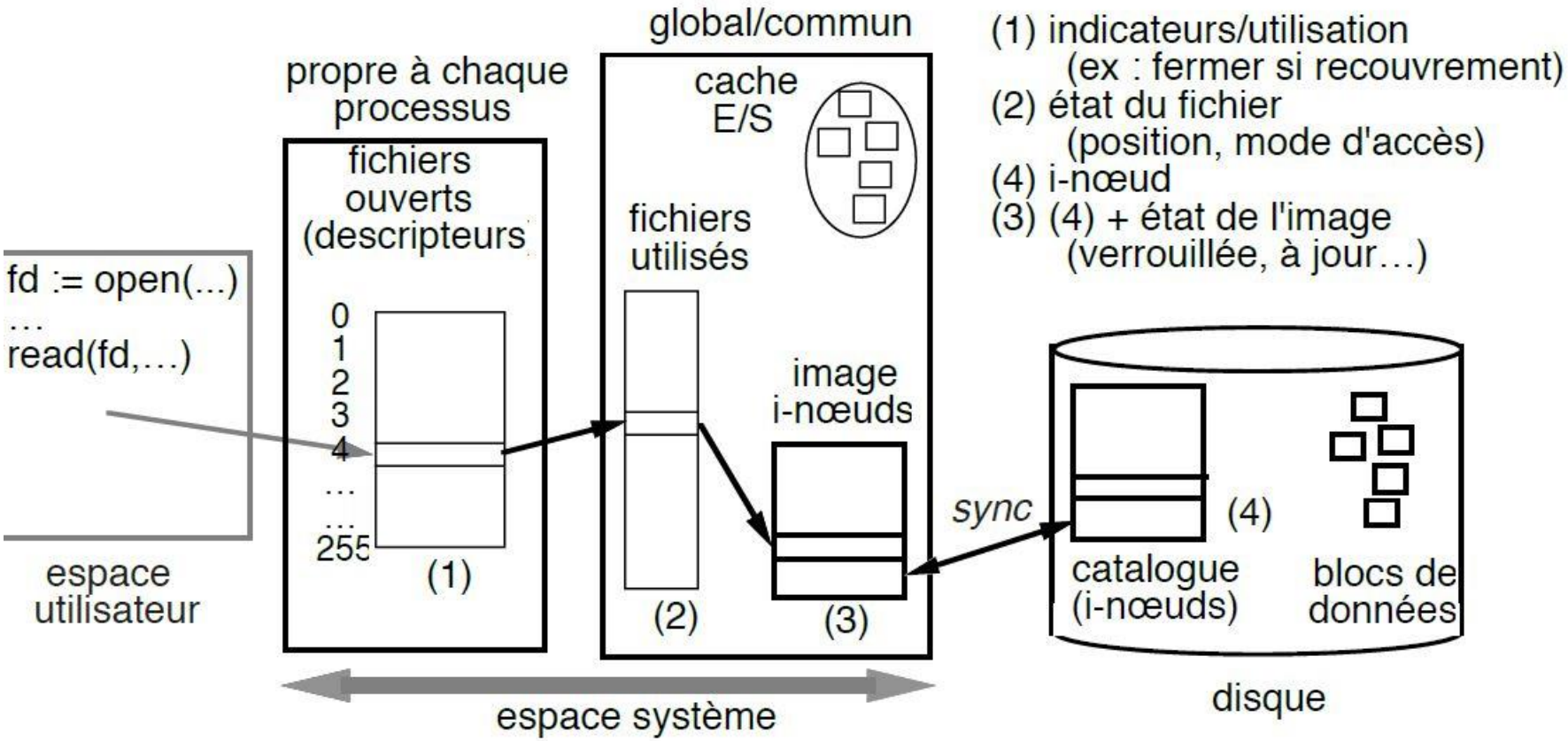
Interface générique pour les échanges avec toutes les ressources

- objet unique : fichier (séquentiel), ou flot
- opérations « génériques » : ouvrir/fermer, lire/écrire, etc.
- Échanges standardisés : caractère, bloc

Catégories de fichiers

- fichiers ordinaires (regular files) : conteneurs de données
- répertoires (directories)
- lien symboliques (soft links)
- tubes (pipes) : canaux de communication FIFO entre processus
- fichiers spéciaux (special files) : permettent de désigner les périphériques comme des fichiers :
  - traditionnellement situés dans /dev
  - deux catégories : bloc et caractère

Systèmes centralisés : TD3 - Fichiers



Systèmes centralisés : TD3 - Fichiers

Stockage d'un fichier sur le disque

- le disque est divisé en blocs
- un fichier = un certain nombre de blocs

Quelle allocation ? Contiguë ? Chaînée ? Indexée ?

Allocation contiguë :

+ rapidité d'accès

- difficile de prévoir le nombre de blocs qu'il faut réserver au fichier : la taille du fichier évolue : trop ou pas assez de blocs contiguës
- trop : des petits espaces perdus, difficilement exploitables
- pas assez : risque de devoir déplacer le fichier
- fragmentation externe : il peut se créer un grand nombre de petites zones dont la taille ne suffit souvent pas pour allouer un fichier mais dont le total correspond a un espace assez volumineux

Systèmes centralisés : TD3 - Fichiers

Stockage d'un fichier sur le disque

Allocation chaînée : chaque blocs contient la référence du bloc suivant

- accès au fichier est totalement séquentiel, on doit toujours commencer le parcours du fichier à partir du début
- La perte ou l'altération d'un chaînage entraîne la perte ou des erreurs d'accès au reste du fichier

Allocation indexée : les numéros des blocs sont enregistrés dans une table

La majorité des systèmes de fichiers actuels appliquent cette allocation

i_node (i_noeuds) dans unix / linux (i pour index)
= carte d'allocation

Systèmes centralisés : TD3 - Fichiers

int stat(const char *, struct stat *) / int fstat(int, struct stat *); (voir man)

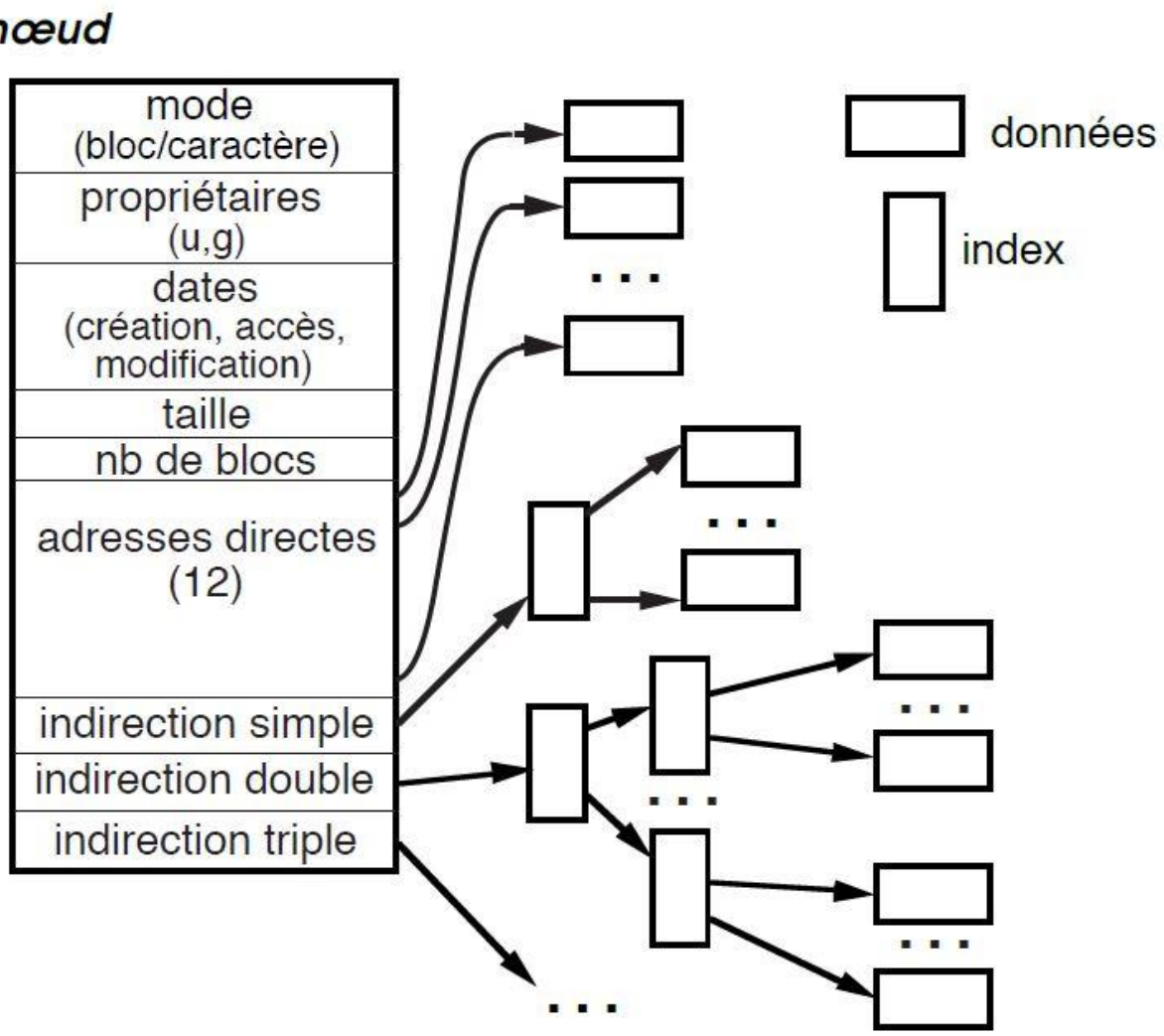
Taille max d'un fichier ?

pour un bloc = 1KO

Et bloc d'indirection = 256 pointeurs

12 blocs
+ 256 blocs
+ 256 x 256 blocs
+ 256 x 256 x 256 blocs

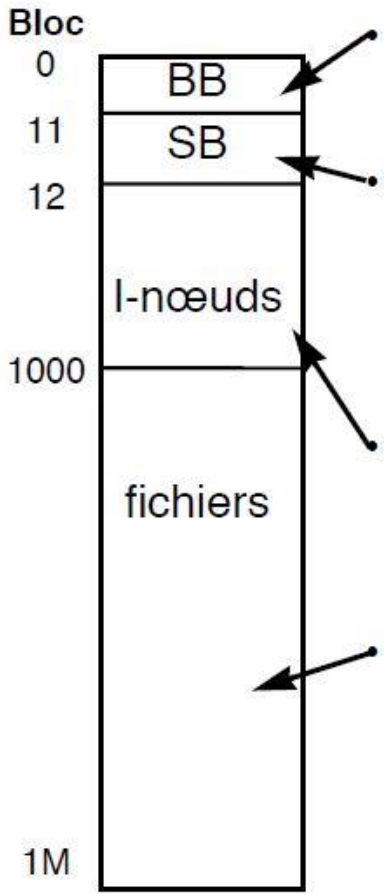
Soit environ 16 GO
(moins en réalité)



Systèmes centralisés : TD3 - Fichiers

Système de fichiers

Organisation et gestion du stockage et de la manipulation des fichiers



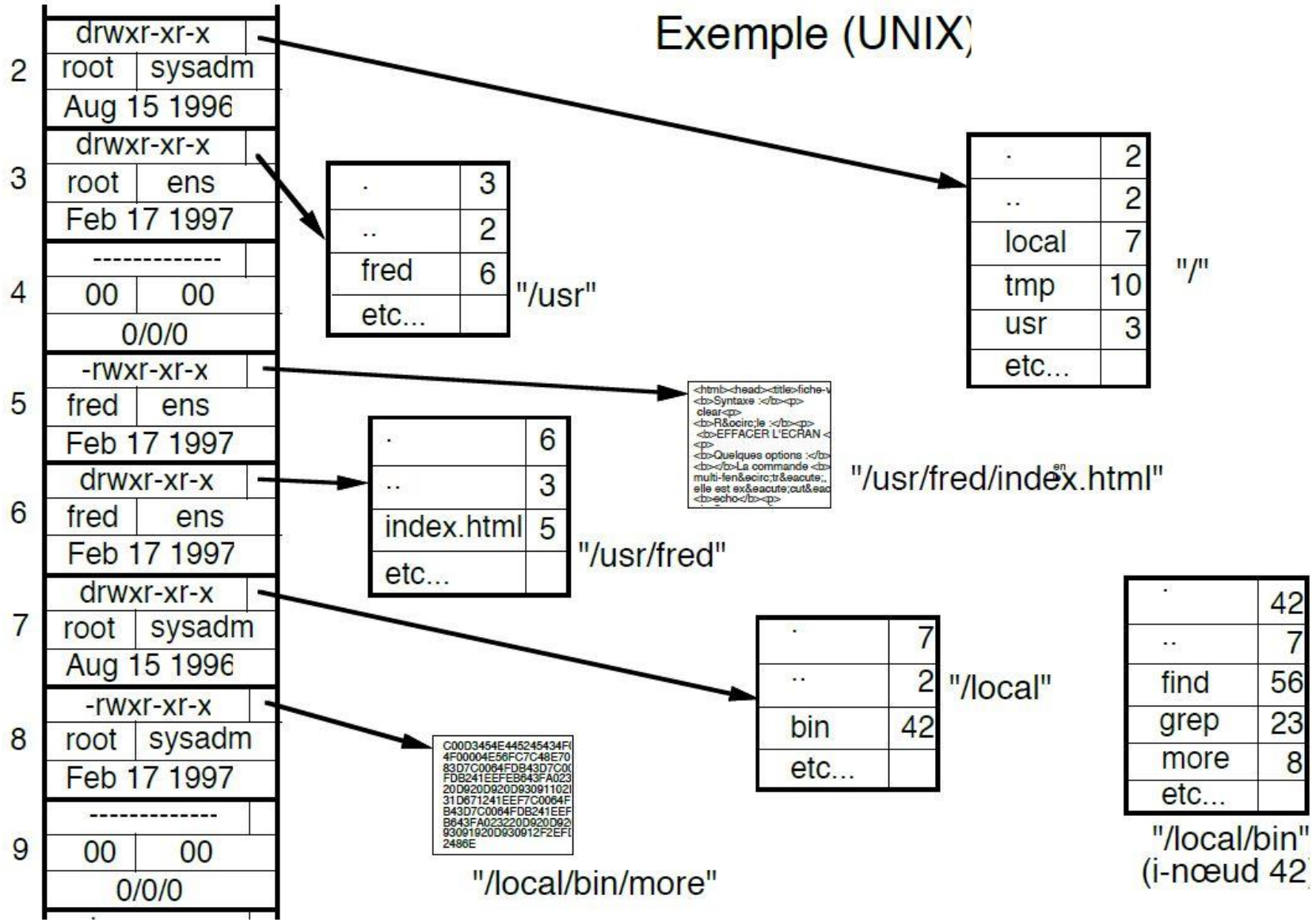
blocs d'amorce
UNIX : `bootblocks (BB)`

informations sur la structure du disque logique : adresse, type des ≠ zones qui le constituent, adresse du premier bloc libre
UNIX : `superbloc (SB)`

catalogue du disque : liste des *descripteurs* des fichiers présents sur le disque logique
UNIX : `table des I-nœuds`

fichiers (ordinaires, répertoires, liens)

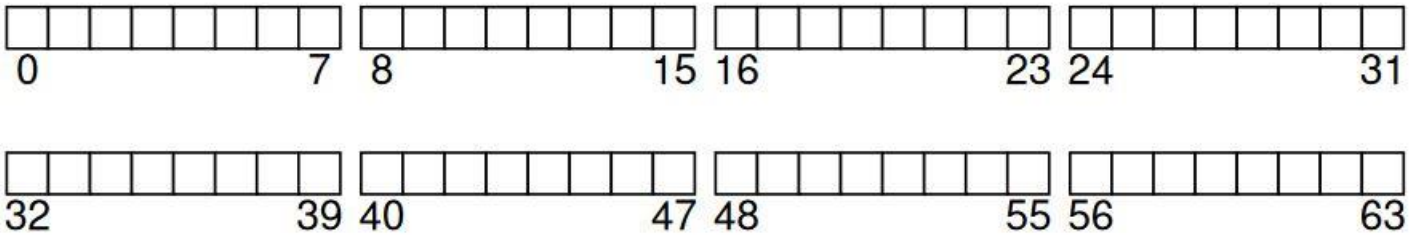
Systèmes centralisés : TD3 - Fichiers



Systèmes centralisés : TD3 - Fichiers

Exemple simple

Un petit disque de 64 blocs de 1KO chacun



Nombre de blocs de données = N < 64

⇒ Nombre max de fichiers = N

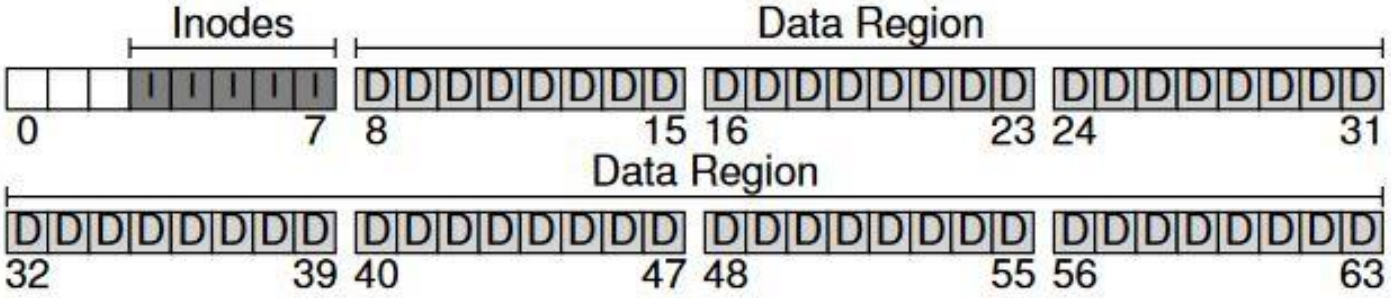
⇒ Nombre max de i_noeuds = N / nombre moyens de blocs par fichier
supposons N / 2

-Taille d'un i_noeud : typiquement 128 octets

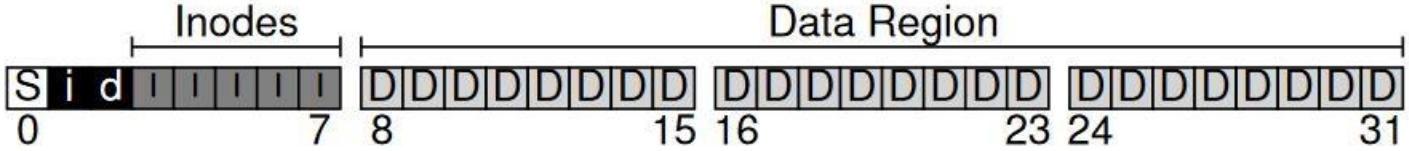
- 1 bloc de 1KO = 8 i_noeuds

- 4 blocs = 32 i_noeuds / **5 blocs** = 40 i_noeuds

Systèmes centralisés : TD3 - Fichiers



- comment savoir si un bloc est libre ou occupé ?
- **une BitMap** : un bit par bloc (= 0 si bloc libre, 1 si occupé)
- 1 bloc est largement suffisant pour la bitmap des blocs de données
- de même 1 bloc pour la bitmap des i_noeuds



Le superbloc (SB) : contient un ensemble d'informations : nombre de i_noeuds, nombre de blocs de données, premier bloc i_noeuds, référence du sf, etc.

Systèmes centralisés : TD3 - Fichiers

Séquencement d'un opération de création et d'écriture :

.	2
..	2
local	7
tmp	10
usr	3
etc...	

/usr/fred/ff.txt

- lecture i_noeud (2) de la racine, lecture bloc données de racine /

.	3
..	2
fred	6
etc...	

"/usr"

- lecture i_noeud du dossier usr (3)
- lecture bloc données du dossier usr

.	6
..	3
index.html	5
etc...	

- lecture i_noeud de fred (6), lecture bloc données de fred

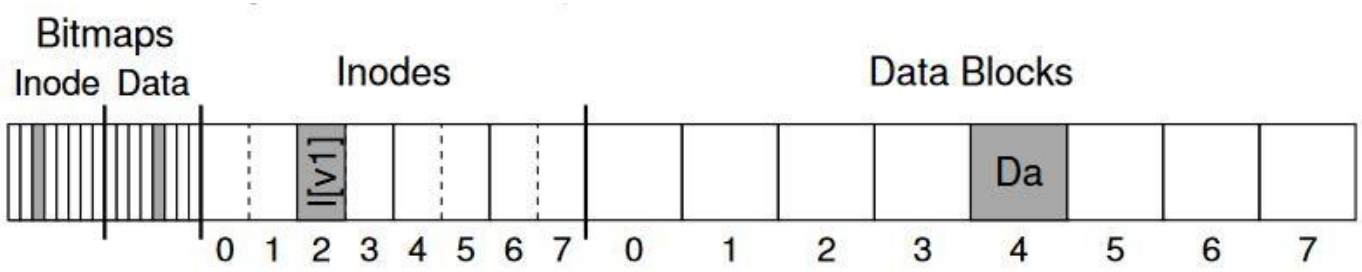
- lecture de la bitmap i_noeuds (i_noeud libre ?), et écriture (alloué)
- écriture dans i_noeud de ff.txt (propriétaire, droits, date, etc)
- écriture dans le bloc de donnée du dossier fred (ff.txt, i_noeud)

- à chaque écriture dans un nouveau bloc :

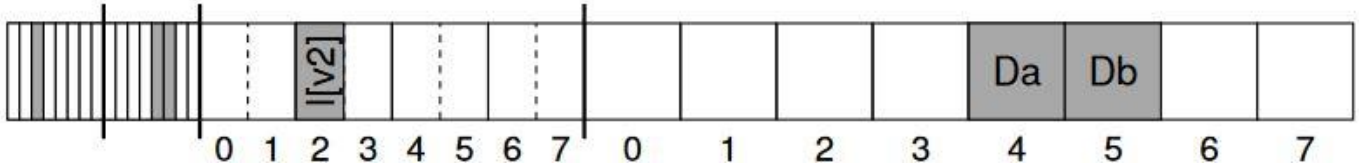
- lecture de la bitmap données (bloc libre), et écriture (alloué)
- écriture dans le bloc de données de ff.txt
- écriture dans le i_noeud de ff.txt

Systèmes centralisés : TD3 - Fichiers

Opération d'ajout – un exemple simplifié :



- fichier occupant le bloc de données Da4
- Ajout de données nécessitant un bloc supplémentaire ⇒ 3 écritures
- bitmap de données (nouveau bloc alloué)
- i_noeud du fichier (nombre de blocs, pointeur sur nouveau bloc)
- les données dans le nouveau bloc



⇒ 3 écritures séparées dans 3 blocs différents

? qu'arrive-t-il en cas de crash entre 2 écritures ?

Systèmes centralisés : TD3 - Fichiers

Problème de consistance

3 écritures séparées dans 3 blocs différents
crash entre 2 écritures ⇒ incohérence / inconsistance des données

? peut-on détecter / corriger ?

1- détection + correction : fsck (file system checker)

S'appuie sur la comparaison entre les bitmap, les i_noeuds (blocs alloués dans les i_noeuds mais libres dans la bitmap, cohérence entre les données des i_neouds, etc. (voir plus de détails dans le sujet du TD, et sur la source référencée)

⇒ beaucoup de vérification (lenteur) sans assurance d'une correction efficace dans de nombreux cas.

Systèmes centralisés : TD3 - Fichiers

Problème de consistance

2- journalisation (similaire aux bases de données)

- avant les 3 écritures, enregistrer une note dans un **espace stable** indiquant ce qu'on va faire (journa

- en cas de crash avant la fin des écritures effectives dans le fichier, on sait ce qu'on doit refaire (voir plus de détails dans le sujet du TD)

- 5 champs à écrire : individuellement avec synchronisation ? Très lent
- une seule écriture ? On ne maîtrise pas l'ordre interne des écritures

⇒ écriture en 2 temps : les 4 premiers champs d'abord, puis le champ TxE

Le coût reste élevé. Pour améliorer les performances :
- écriture des blocs de données directement dans l'espace du fichier
- écriture des métadonnées (i_noeuds, et bitmap) dans le journal

Systèmes centralisés : TD3 - Fichiers

Primitives de fichiers

Ouverture

Avant d'utiliser un fichier, il faut l'ouvrir, ce qui lui alloue un descripteur.

```
int open (const char *chemin, int mode, mode_t droits);
```

descripteur

nom du fichier

(combiné avec le masque de création)

O_RDONLY	ouverture en lecture
O_WRONLY	ouverture en écriture
O_RDWR	ouverture en lecture et écriture
O_APPEND	ouverture en écriture en fin de fichier
O_CREAT	création du fichier avec droits d'accès définis par `droits`
O_EXCL	avec `O_CREAT` provoque un échec si le fichier existe déjà.
O_TRUNC	ramène la taille du fichier à zéro si le fichier existe déjà.

…

Note : les modes peuvent être combinés (quand cela a un sens) avec le ou (|)

permissions = droit & ~umask (~ : complément binaire)

-rwx rwx rwx (en octal 0777)

Systèmes centralisés : TD3 - Fichiers

Exemples :

desc = open("toto.txt", O_WRONLY | O_CREAT | O_TRUNC, 0640);

ouvre le fichier "toto.txt" en écriture s'il existe, en le vidant de son contenu (O_TRUNC)

Ou, le crée avec des droits rw- r-- --- et l'ouvre en écriture (O_CREAT)

desc = open("toto.txt", O_WRONLY | O_CREAT | O_APPEND, 0640);

Si le fichier existe, les nouvelles écritures viennent s'ajouter aux données anciennes

desc = open("toto.txt", O_WRONLY | O_CREAT , 0640);

- Si le fichier existe, les nouvelles écritures commencent en début du fichier
- et viennent remplacer progressivement les données précédente,
- mais ne les écrasent complètement que si le volume des nouvelles données est supérieur ou égal à celui des anciennes

desc = open("toto.txt", O_WRONLY | O_CREAT | O_EXCL, 0640);

Ouvre le fichier en Provoque une erreur (desc < 0) si le fichier existe

Systèmes centralisés : API en langage C

```
desc = open(argv[1], O_RDONLY);
printf("Resultat ouverture RDONLY = %d \n", desc);
if (desc>0) close(desc);

desc = open(argv[1], O_WRONLY);
printf("Resultat ouverture WRONLY = %d \n", desc);
if (desc>0) close(desc);

desc = open(argv[1], O_RDWR);
printf("Resultat ouverture RDWR = %d \n", desc);
if (desc>0) close(desc);

desc = open(argv[1], O_WRONLY | O_CREAT | O_EXCL,0640);
printf("Resultat ouverture WRONLY | CREATE | EXCL =
%d            \n", desc);
if (desc>0) close(desc);

desc = open(argv[1], O_WRONLY | O_CREAT, 0640);
printf("Resultat ouverture WRONLY= %d \n", desc);
if (desc>0) close(desc);
```

exécution avec un fichier inexistant puis 2nde exécution ?

Systèmes centralisés : TD3 - Fichiers

Positionnement

- Les opérations d'accès au fichier sont effectuées à partir d'une position courante (*offset*).
- Initialement (à l'ouverture) la position courante est 0.
- Cette position est modifiée
  - ◊ indirectement, par les opérations d'accès : lecture (`read`) et écriture (`write`).
  - ◊ directement, par l'opération `lseek`

```
long lseek (int descripteur, long offset, int origine);
```

nouvelle position

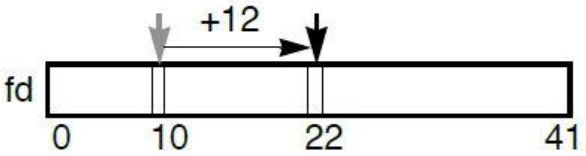
déplacement demandé

SEEK_SET	à partir du début du fichier
SEEK_CUR	à partir de la position actuelle
SEEK_END	à partir de la fin du fichier

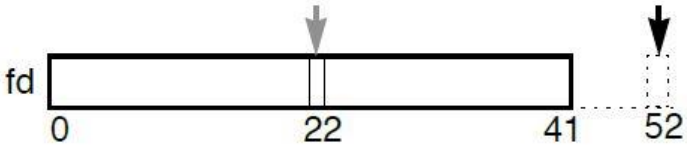
- La position courante peut être fixée après la fin actuelle du fichier.

Exemples

lseek(fd, 12, SEEK_CUR)
la position courante progresse de12 octets depuis sa valeur actuelle



lseek(fd, 52, SEEK_SET)
la position courante est fixée à 52



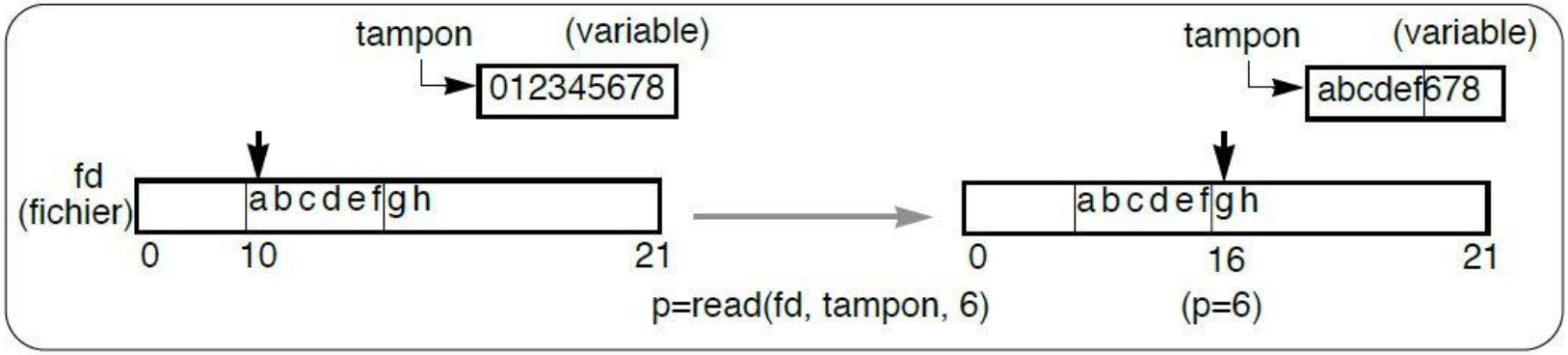
Systèmes centralisés : TD3 - Fichiers

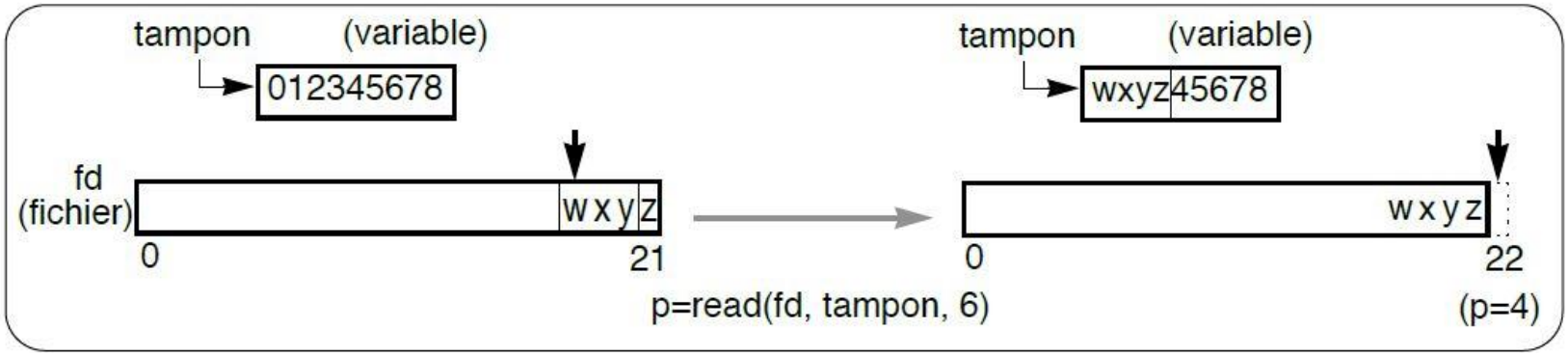
Lecture

```
ssize_t read (int descripteur, void *tampon, size_t taille);
```

- nombre d'octets effectivement lus (-1 si erreur)
- adresse du résultat de la lecture
- nombre d'octets à lire

Exemple





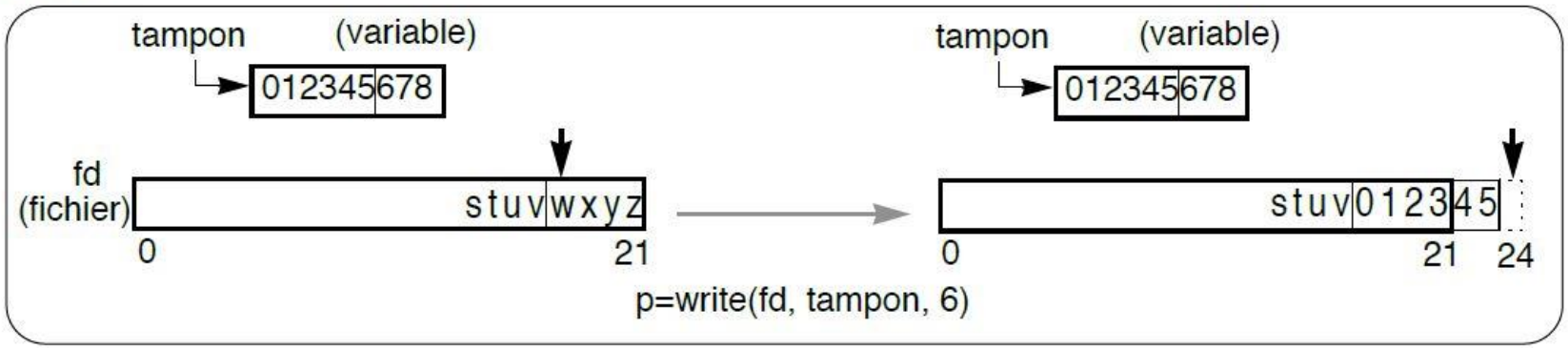
Systèmes centralisés : TD3 - Fichiers

Ecriture

```
ssize_t write (int descripteur, void *tampon, size_t taille);
```

- nombre d'octets effectivement écrits (-1 si erreur)
- adresse de la zone contenant les données à écrire
- nombre d'octets à écrire

Exemple



Remarque

Il existe des primitives permettant de lire ou d'écrire des vecteurs d'octets (**`readv/writev`**), ou à partir d'une position donnée (**`pread/pwrite`**)

Fermeture

Un fichier qui n'est plus utilisé peut être fermé, en fournissant son descripteur en paramètre

```
int close (int descripteur);
```

Systèmes centralisés : API en langage C

Fichier partagé :

Une seule ouverture / lectures concurrentes

- Père ouvre un ficher en lecture et crée un fils
- Fichier : ‘’abcdefghijklmnopqrstuvwxyz’’

père lit	fils lit
4 caractères	
	6 caractères
4 caractères	
	6 caractères

Ouvertures séparées / lectures concurrentes

Chaque processus effectue sa propre ouverture

Mêmes lectures que dessus

Systèmes centralisés : API en langage C

Une seule ouverture / écritures concurrentes

- Père ouvre un ficher en écriture (O_CREAT | O_TRUNC) et crée un fils

père écrit	fils écrit
abcd	
	klmnop
efgh	
	qrstuv

Ouvertures séparées / écritures concurrentes

Chaque processus effectue sa propre ouverture

Mêmes écritures que dessus

Projet minishell

Minishell (père)

- Crée un fils
- attend qu'il termine (wait)

fils : exécute la commande externe

Question 5 : comment faire si commande en arrière plan (&) ?

Question 6 : liste des commandes : tient à jour l'état de toutes les commandes en cours : wait suffit-il ?

- La commande fg relance ou ramène une commande en avant plan

⇒ doit bloquer le terminal, comment ? où ?

- une commande peut être suspendu, relancée, tuée depuis un autre terminal. Comment le savoir ? Comment tenir la liste à jour ?

Question 7 : ctrl-Z [resp. ctrl-C] provoque l'envoi du signal SIGTSTP [resp SIGINT] au processus en avant plan et ses fils :

- Le minishell et ses fils doivent être protégés par rapport à ces signaux
- exec conserve SIG_IGN, mais remplace tout autre traitant pas SIG_DFL
- une commande peut passer de suspendue à avant plan et inversement