Cours de système

Le système de gestion de fichiers

Sébastien Paumier/Sylvain Cherrier

Rôle

- gérer l'organisation des fichiers et de l'arborescence sur les supports physiques
- qualités requises:
  - partage: accès concurrents aux disques et aux fichiers
  - efficacité: vitesse (cache)
  - fiabilité: droit d'accès
  - transparence: accès homogènes

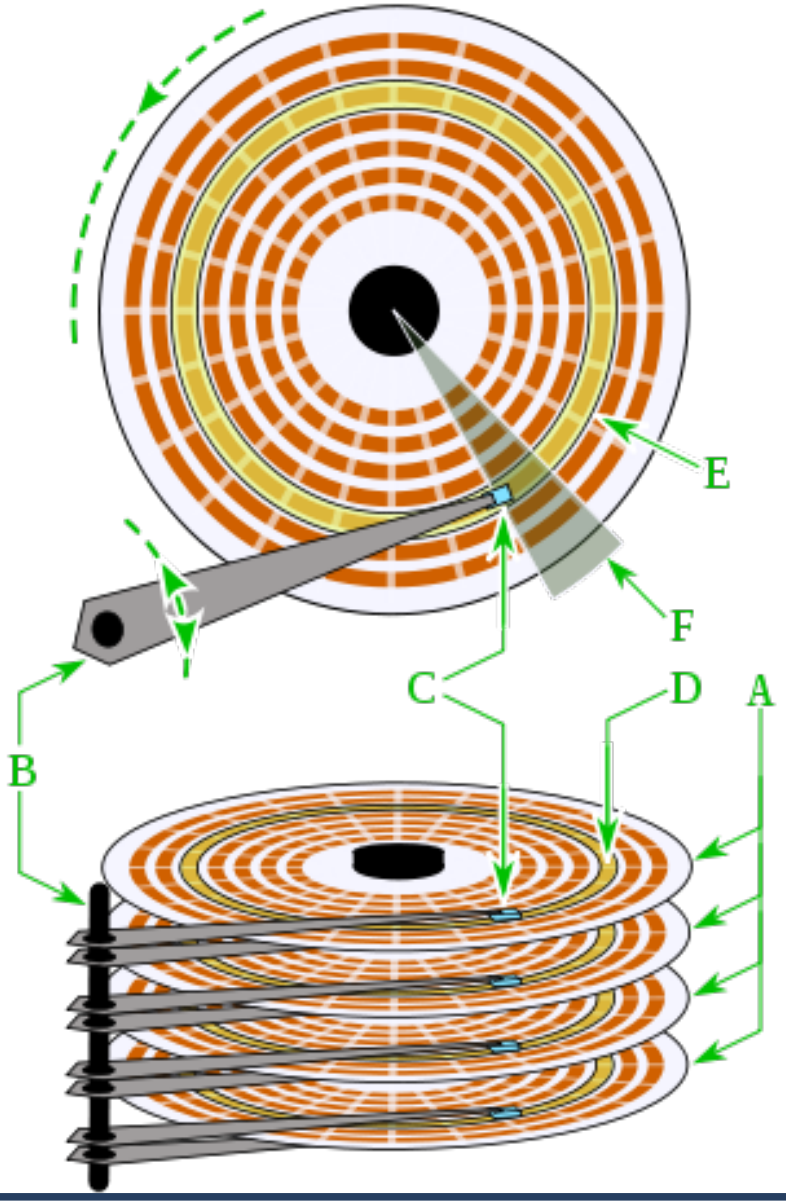
Les fichiers

- concept de fichier=suite finie d'octets accessible en lecture/écriture
- matérialisation physique par des blocs disques et une inode qui contient ses propriétés (mais pas son nom)
- l'inode contient les informations pour localiser le fichier sur le support et répondre à **`stat`**

DD / FS / Inode

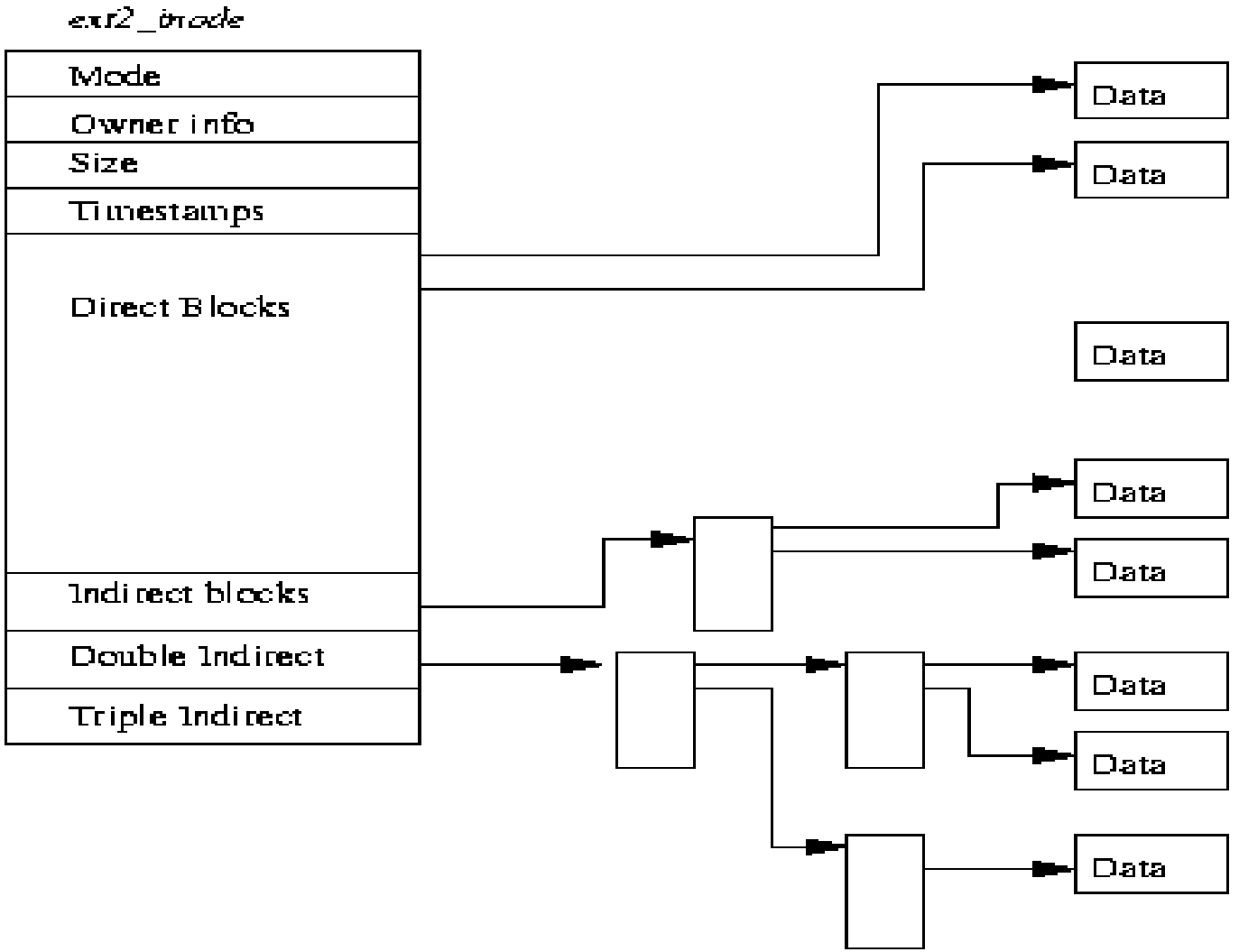
- Un DD contient des blocs adressables de 512 octets
- Un FS organise le contenu du DD
- Les blocs sont regroupés en clusters
- Les clusters :
  - des données,
  - des informations sur l'organisation
- Certains FS intercalent une notion d'inode (nom → inode → données)

Disque Dur



« Basic disk displaying CHS ». Sous licence Domaine public via Wikimedia Commons - https://commons.wikimedia.org/wiki/File:Basic_disk_displaying_CHS.svg#/media/File:Basic_disk_displaying_CHS.svg

Structure de l'Inode



Les fichiers

- ne pas confondre fichier et descripteur!
- exemple: **`lseek`** ne marche que sur les vrais fichiers, comme le dit man

lseek.cpp

ERREURS

ESPIPE fd est associé à un tube (pipe), une socket, ou une file FIFO.

Les fichiers

- les fichiers ordinaires sont non typés (contre-exemple, caractère de fin de fichier sous DOS)
- Fichier(données) = inode POSIX:
  - localisation sur le disque
  - type: `- d l s p c b`
  - propriétaire et groupe du propriétaire, droits d'accès, taille, dates (création, modification, accès), nombre de liens physiques

Fichiers spéciaux

- contenu structuré protégé par le système et accessible seulement via des appels dédiés pour préserver leur intégrité
- répertoires
- périphériques physiques/virtuels (**`/dev`**):
  - mode caractère: terminaux, clavier, etc
  - mode bloc: disques
- fichiers logiques:
  - liens, sockets, tubes, **`/dev/null`**, etc

L'arborescence

- depuis MULTICS, système unifié représenté par un arbre:
  - racine unique
  - extensibilité: un nouveau périphérique est vu comme un répertoire monté dans l'arborescence (montage)
- en réalité, à cause des liens physiques autorisés sur les fichiers, c'est un graphe, mais acyclique, donc, plein de bonnes propriétés algorithmiques

Graphe acyclique

- tous les algos de parcours sont beaucoup plus simples sur les graphes acyliques
  - ramasse-miettes pour récupérer des inodes ou des blocs perdus après un crash
- comme exprimer le répertoire parent s'il n'est pas unique ?

Liens symboliques

- fichier spécial décrivant un pointeur vers un fichier ou un répertoire
- c'est aux applications de vérifier qu'elles ne suivent pas les liens symboliques sur les répertoires pouvant induire des cycles:

```
$>mkdir toto
$>mkdir toto/titi
$>cd toto/titi
$>ln -s ../.. ./toto2
$>ls -RL ../..
...
ls: ../../toto/titi/toto2: ne peut lister un répertoire déjà listé
```

Les inodes

- quand un fichier est utilisé par un processus, son inode est chargée en mémoire et on lui associe des informations en plus:
  - fichier ou inode à écrire
  - le fichier est un point de montage
  - informations sur les verrous
  - informations sur l'éventuel montage à distance
  - etc

Inodes et noms de fichiers

- découplage entre les deux qui permet une gestion très souple:

```
$>mkdir toto
$>cd toto
$>touch titi
$>rm -Rf ../toto
$>ls -al
total 0
$>mkdir ../toto
$>touch ../toto/tata
$>ls -al ../toto
total 8
drwxr-xr-x 2 paumier paumier 4096 2010-09-19 11:19 .
drwxr-xr-x 3 paumier paumier 4096 2010-09-19 11:19 ..
-rw-r--r-- 1 paumier paumier    0 2010-09-19 11:19 tata
```

même pas `.` et `..`

Inodes et noms de fichiers

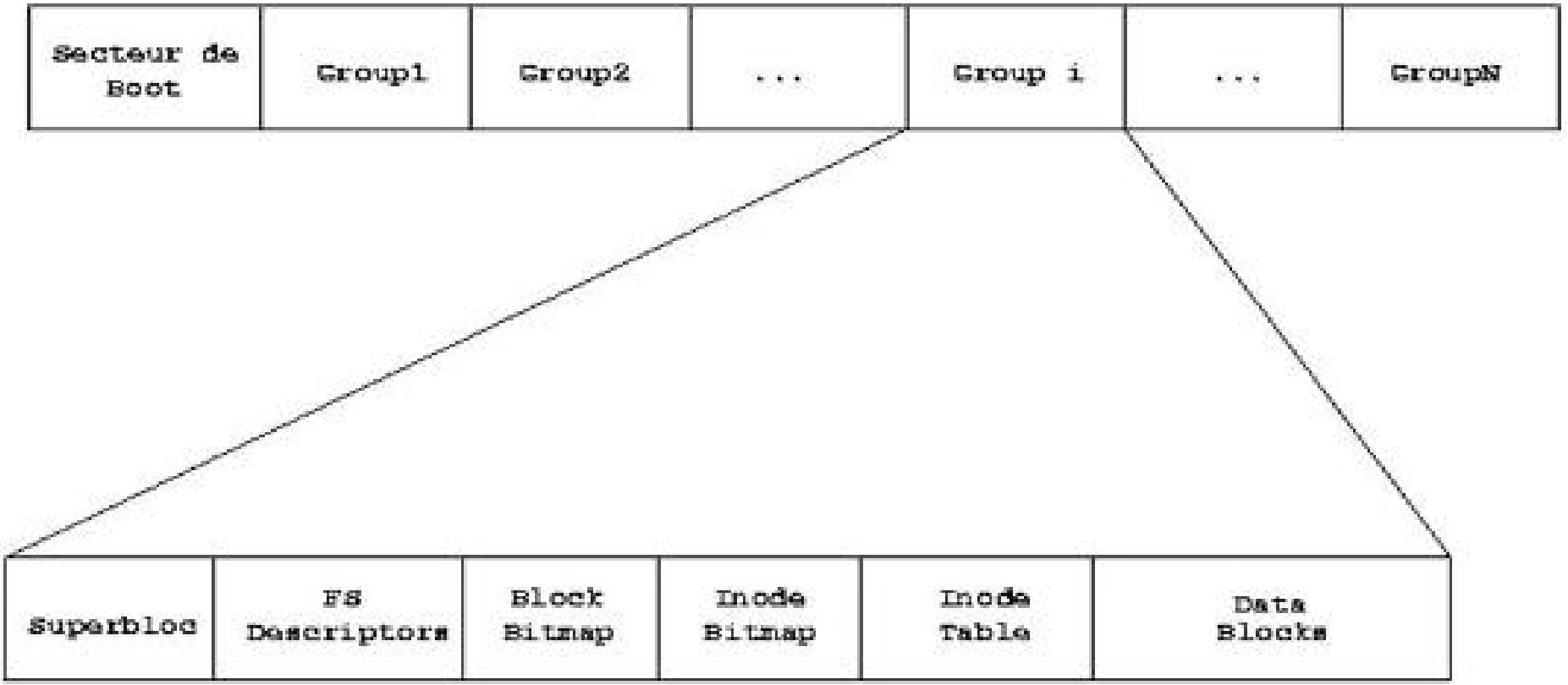
- en détruisant **toto**, on laisse le processus sans répertoire courant
- c'est le shell qui permet de remonter vers la racine avec **..**

```
$>touch tutu
touch: ne peut faire un touch sur `tutu': Aucun fichier
ou dossier de ce type
$>pwd
/home/paumier/toto
$>/bin/pwd
/bin/pwd: ne peut repérer l'entrée du répertoire dans
`..' concordant avec le inode
```

Organisation du disque

- disque physique partionné en disques logiques
- architecture d'un disque logique sous System V (il y a d'autres architectures possibles):
  - boot bloc: amorce pour charger l'OS
  - super bloc: informations sur le disque logique
  - table des inodes
  - blocs de données

Organisation du Disque



Structure Physique du système de fichiers Ext2fs

Organisation du disque

- ext2: les inodes sont groupés avec des blocs par secteurs pour éviter des allers-retours coûteux à la tête de lecture
- mais, le modèle System V retrouve de l'intérêt sur les supports en lecture seule (CD, DVD): tous les blocs peuvent/doivent être contigus
  - ne pas "sauter" pendant la lecture d'un gros fichier multimédia

Journalisation

- en cas de crash, le disque peut être dans un état incohérent (exemple: inode qui devait être écrite)
- pour faciliter la récupération d'erreurs sans avoir à explorer tout le disque, on utilise la journalisation:
  - ext3, ext4, NTFS, etc

RAID

- Redundant Array of Inexpensive Disks
- opposé au SLED: Single Large Expensive Disk
- buts:
  - augmenter la tolérance aux pannes (il faut plusieurs pannes simultanées pour compromettre les données)
  - faciliter la maintenance (pas nécessairement besoin d'arrêter le système pour changer un disque)

RAID

- différentes sortes:
  - RAID 0: répartition sur plusieurs disques pour accélérer les accès
  - RAID 1: redondance pour améliorer la tolérance aux pannes
  - RAID 5: les deux à la fois, avec contrôle de parité
- cf. exposés des anciens IR3

RAID matériel

- totalement invisible pour le système
- ne consomme pas de CPU
- permet parfois le remplacement de disque à chaud
- ne fonctionne que sur certains types de périphérique (pas possible d'avoir une combinaison SCSI/USB)
- possède ses propres risques de panne et de maintenance (matériel+firmware)

RAID logiciel

- couche d'abstraction matérielle gérée par le système
- pas besoin de matériel supplémentaire
- possibilité de combiner tous types de périphériques
- consommation de CPU
- pas toujours possible de stocker le système sur un RAID, puisque c'est le système qui permet d'y accéder

Les descripteurs

- les appels systèmes manipulent des descripteurs entiers pour garantir un polymorphisme maximum
- ce ne sont pas forcément des fichiers !
- un processus a 3 descripteurs par défaut: 0 (stdin), 1 (stdout) et 2 (stderr)

Les appels système

- grâce aux descripteurs, les primitives d'E/S sur fichiers sont indépendants des supports physiques
- les appels système sont exécutés par le noyau
- renvoient -1 en cas d'erreur
  - toujours les tester!

```
if (-1==sys_biniou(...)) {
    perror("sys_biniou");
    /* maybe exit if needed */
}
```

open

- `int open(const char *pathname, int flags);`
- `int open(const char *pathname, int flags, mode_t mode);`
- ouvre un fichier, qui restera ouvert jusqu'à la fin du processus ou **`close`**
- par défaut, **`execve`** ne ferme pas les fichiers, ce qui permet entre autres à un processus d'hériter des 0, 1 et 2 de son père

open

- **`flags`**=mode d'ouverture:
  - **`O_RDONLY`**, **`O_WRONLY`** et **`O_RDWR`**
- peut se combiner avec un OU binaire avec un attribut de création:
  - **`O_CREAT`**: créer si nécessaire le fichier
  - **`O_APPEND`**: ouvrir en mode ajout
  - **`O_TRUNC`**: tronquer le fichier s'il existe
  - **`O_EXCL`**: combiné avec **`O_CREAT`**, vérifie que le fichier n'existait pas
  - etc

open

- **mode**=droits d'accès à utiliser si le fichier doit être créé grâce à **O_CREAT**
- Utiliser et combiner avec | les constantes RWX ou R W X, pour USR GRP OTH → S_IRUSR(400), S_IRWXO (007) ou S_IWGRP(020)
- se combinent avec le umask
- si on oublie, **open** donnera des résultats bizarres...
- if (-1==(pfd = open(„test“, O_WRONLY | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR | S_IRGRP))) {...

Etapes d'un open

- trouver l'inode du fichier
- la charger en mémoire si elle n'y est pas
- vérifier les droits d'accès
  - la vérification n'est faite qu'à l'ouverture
  - si on enlève les droits sur un fichier ouvert, ça ne bloque pas le processus qui utilise le fichier !
- allouer une entrée dans la table des fichiers ouverts

Etapes d'un open

- positionner l'offset courant (0 ou fin de fichier en cas d'ouverture en ajout)
- allouer une place dans la table des descripteurs de fichiers du processus
- renvoyer au processus appelant le descripteur obtenu

open et les tables

- 3 tables:

table des inodes

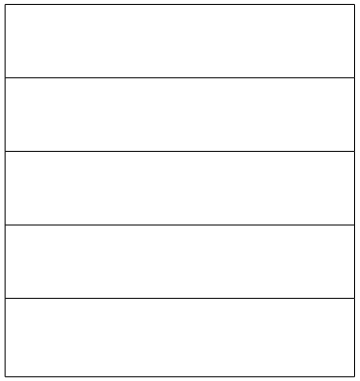
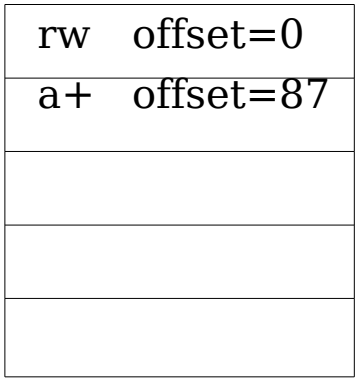
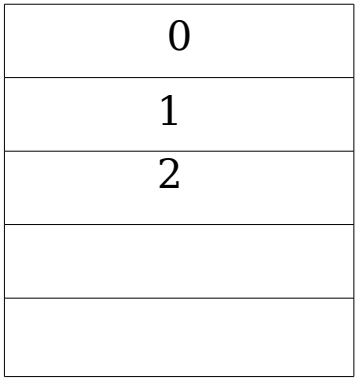


table des ouvertures de fichiers



dans le noyau

table des descripteurs



dans chaque processus

creat

- `int creat(const char *pathname, mode_t mode);`
- équivalent à `open` avec `flags`=`O_CREAT | O_WRONLY | O_TRUNC`

access

- `int access(const char *pathname, mode_t mode);`
- teste les droits d'accès de l'utilisateur réel
- dangereux: les droits peuvent changer juste après qu'on les ait consultés !
- ne peut être utilisé sans risque que dans le noyau, donc ne devrait pas être un appel système visible

unlink

- `int unlink(const char *pathname);`
- décrémente le compteur de liens physiques
- s'il tombe à zéro, le fichier sera détruit dès que plus aucun processus n'aura de descripteur ouvert
- application: fichier qui disparaît automatiquement à la fin d'un programme

close

- `int close(int fd);`
- ferme le fichier et libère le descripteur
- si `fd` était le dernier descripteur sur le fichier (à l'intérieur du processus):
  - le compteur associé à l'inode en mémoire est décrémenté; s'il vaut 0, l'inode est déchargée
  - les ressources associées à l'ouverture du fichier sont libérées

close

- si le compteur des liens physiques vaut 0, le fichier est effacé
  - son inode est recyclée, ainsi que ses blocs disques
- lève les éventuels verrous, même s'ils n'ont pas été posés sur `fd`

read

- `ssize_t read(int fd, void *buf, size_t count);`
- peut retourner moins que demandé en cas:
  - de fin de fichier
  - de lecture sur un tube ou un terminal
  - d'interruption par un signal

read

- vérification de la validité du descripteur et du bon mode d'ouverture
- mais les droits d'accès de l'utilisateur ne sont pas testés

`fs/read_write.c`

`write_no_check.cpp`

write

- `ssize_t write(int fd, const void *buf, size_t count);`
- peut écrire moins que demandé si:
  - plus de place sur le périphérique
  - quota disque du processus dépassé
  - interruption par un signal
- dans un fichier, la modification de la position et l'écriture sont faites de façon atomique

lseek

- `off_t lseek(int fd, off_t offset, int whence);`
- ne marche pas pour tout type de descripteur ! (cf man)
- `whence`=`SEEK_SET`, `SEEK_CUR` ou `SEEK_END`
- peut aller au-delà de la taille du fichier (sans la modifier); une écriture modifiera alors la taille réelle et on lira des 0 dans le trou intermédiaire

dup

- `int dup(int oldfd);`
- duplique un descripteur de fichier valide
- la copie reçoit le plus petit numéro de descripteur disponible
- pénible quand on veut un numéro précis
- deprecated!!

`init/main.c`

dup2

- **`int dup2(int oldfd, int newfd);`**
- duplique un descripteur de fichier valide
- la copie reçoit le numéro **`newfd`**
- si **`newfd`** était ouvert, il est fermé automatiquement : le old devient le new.
- très utile pour les redirections:

```
int fd=open("toto",O_RDONLY);
if (fd!=-1) {
    dup2(fd,0); /* utilise "toto" comme entrée standard */
}
```

dup et dup2

- le descripteur et sa copie partagent la position modifiable par **`lseek`**
  - risque de problèmes de concurrence
- ils ne partagent pas l'attribut close-on-exec (pour ça, utiliser **`dup3`**)
- après une duplication, il faut fermer deux fois le descripteur pour fermer complètement

fcntl

- `int fcntl(int fd, int cmd, ... /* arg */ );`
- fonction fourre-tout permettant de manipuler des descripteurs de fichiers:
  - changer la position courante
  - poser des verrous
  - dupliquer un descripteur
  - faire de la veille sur des modifications de fichiers ou de répertoires
  - etc

stat, fstat, lstat

- `int stat(const char *path, struct stat *buf);`
- `int fstat(int fd, struct stat *buf);`
- `int lstat(const char *path, struct stat *buf);`
- obtenir les infos sur un fichier
- `lstat` ne suit pas les liens symboliques

ftw

- `int ftw(const char *dirpath, int (*fn) (const char *fpath, const struct stat *sb, int typeflag), int nopenfd);`
- file tree walk: parcourir une arborescence
- `nopenfd`=nombre maximum de descripteurs utilisables, pour ne pas saturer la table des descripteurs du processus

ftw.cpp
ftw2.cpp