Troisième chapitre

Fichiers et entrées/sorties

- Fichiers et systèmes de fichiers
  - Introduction
  - Désignation des fichiers
- Interface d'utilisation des fichiers
  - Interface en langage de commande
  - Interface de programmation
  - Flots d'entrée/sortie et tubes
- Réalisation des fichiers
  - Implantation des fichiers
  - Manipulation des répertoires
  - Protection des fichiers
- Conclusion

Fichiers

Définitions

- Fichier : un ensemble d'informations groupées pour conservation et utilisation
- Système de gestion de fichiers (SGF) : partie de l'OS conservant les fichiers et permettant d'y accéder

Fonctions d'un SGF

- Conservation permanente des fichiers (*ie*, après la fin des processus, sur disque)
- Organisation logique et désignation des fichiers
- Partage et protection des fichiers
- Réalisation des fonctions d'accès aux fichiers

Unix et les fichiers

Les fichiers jouent un rôle central dans Unix

- Support des données
- Support des programmes exécutables
- Communication avec l'utilisateur : fichiers de config, `stdin`, `stdout`
- Communication entre processus : sockets, tubes, etc.
- Interface du noyau : `/proc`
- Interface avec le matériel : périphériques dans `/dev`

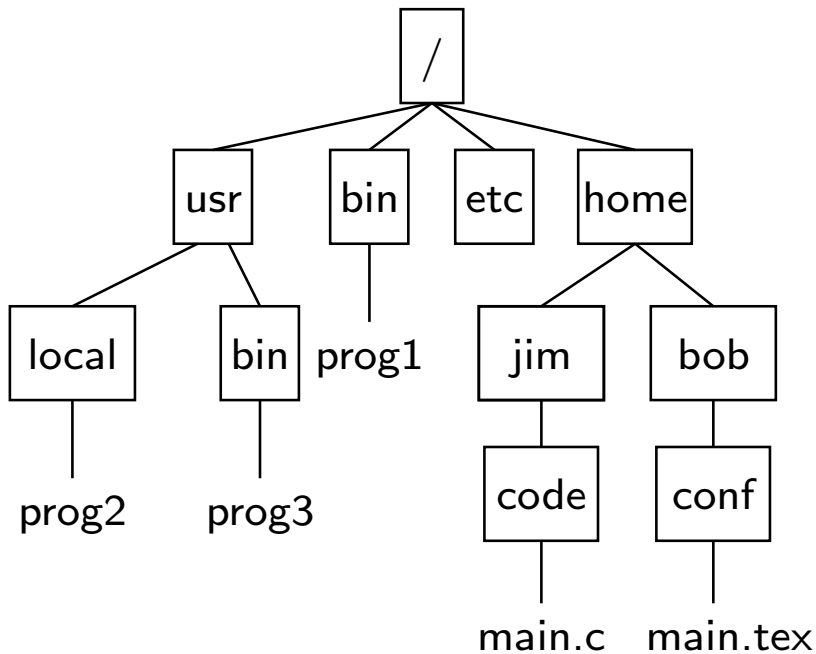
Désignation des fichiers

Désignation symbolique (nommage) : Organisation hiérarchique

- Noeuds intermédiaires : répertoires (*directory* – ce sont aussi des fichiers)
- Noeuds terminaux : fichiers simples
- Nom absolu d'un fichier : le chemin d'accès depuis la racine

Exemples de chemins absolus :

```
/
/bin
/usr/local/bin/prog
/home/bob/conf/main.tex
/home/jim/code/main.c
```



Raccourcis pour simplifier la désignation

Noms relatifs au répertoire courant

- Depuis `/home/bob`, `conf/main.tex` = `/home/bob/conf/main.tex`

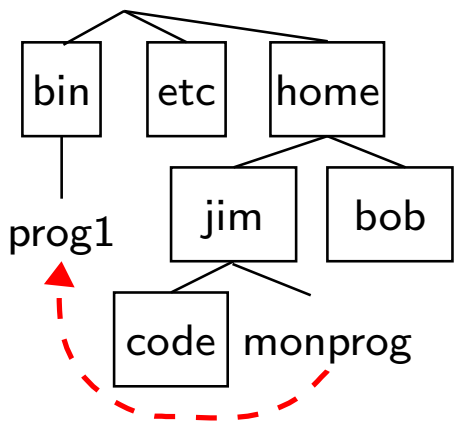
Abréviations

- Répertoire père : depuis `/home/bob`, `../jim/code` = `/home/jim/code`
- Répertoire courant : depuis `/bin`, `./prog1` = `/bin/prog1`
- Depuis n'importe où, ~bob/ = /home/bob/ et ~/ = /home/<moi>/

Liens symboliques

Depuis `/home/jim`

- Création du lien : `ln -s` *cible nom_du_lien*
  Exemple : `ln -s /bin/prog1 monprog`
- `/home/jim/prog1` désigne `/bin/prog1`
- Si la cible est supprimée, le lien devient invalide



Liens durs : `ln cible nom`

- Comme un lien symbolique, mais copies indifférenciables (ok après `rm cible`)
- Interdit pour les répertoires (cohérence de l'arbre)

Règles de recherche des exécutables

- Taper le chemin complet des exécutable (/bin/ls) est lourd
- $\Rightarrow$ on tape le nom sans le chemin et le shell cherche
- Variable environnement PATH : liste de répertoires à examiner successivement
  `/usr/local/bin:/usr/local/sbin:/sbin:/usr/sbin:/bin:/usr/bin:/usr/bin/X11`
- Commande `which` indique quelle version est utilisée

Exercice : Comment exécuter un script nommé gcc dans le répertoire courant ?

- Solution 1 : export PATH=". :$PATH" ; gcc <bla>
- Solution 2 : ./gcc <bla>

Utilisations courantes des fichiers

- Unix : fichiers = suite d'octets sans structure (interprétée par utilisateur)
- Windows : différencie fichiers textes (où \n est modifié) des fichiers binaires

Programmes exécutables

- Commandes du système ou programmes créés par un utilisateur
- Exemple : `gcc -o test test.c ; ./test`
  Produit programme exécutable dans fichier `test` ; exécute le programme test
- Question : pourquoi `./test` ? (car test est un binaire classique : `if test $n -le 0;`)

Fichiers de données

- Documents, images, programmes sources, etc.
- Convention : le suffixe indique la nature du contenu
  Exemples : .c (programme C), .o (binaire translatable, cf. plus loin), .h (entête C), .gif (un format d'images), .pdf (Portable Document Format), *etc.*
  Remarque : ne pas faire une confiance aveugle à l'extension (*cf.* `man file`)

Fichiers temporaires servant pour la communication

- Ne pas oublier de les supprimer après usage
- On peut aussi utiliser des tubes (*cf.* plus loin)

Troisième chapitre

Fichiers et entrées/sorties

Utilisation des fichiers dans le langage de commande

Créer un fichier

- Souvent créés par applications (éditeur, compilateur, etc), pas par shell
- On peut néanmoins créer explicitement un fichier (cf plus loin, avec flots)

Quelques commandes utiles

créer un répertoire `mkdir <nom du répertoire>` (initialement vide)

détruire un fichier `rm <nom du fichier>` (option `-i` : interactif)

détruire un répertoire `rmdir <rep>` s'il est vide ; `rm -r <rep>` sinon

chemin absolu du répertoire courant `pwd` (*print working directory*)

contenu du répertoire courant `ls` (*list* – `ls -l` plus complet)

Expansion des noms de fichiers (*globbing*)

- * désigne n'importe quelle chaîne de caractères :
  `rm *.o` : détruit tous les fichiers dont le nom finit par .o
  `ls *.c` : donne la liste de tous les fichiers dont le nom finit par .c
  `*.[co][A-Z]` : fichiers dont le nom termine par 'c' ou 'o' puis une majuscule

Interface de programmation POSIX des fichiers

Interface système

- Dans l'interface de programmation, un fichier est représenté par descripteur
  Les descripteurs sont de (petits) entiers (indice dans des tables du noyau)
- Il faut **ouvrir** un fichier avant usage :
  `fd = open("/home/toto/fich", O_RDONLY, 0);`
  - Ici, ouverture en lecture seule (écriture interdite)
  - Autres modes : O_RDWRITE, O_WRONLY
  - fd stocke le descripteur alloué par le système (ou –1 si erreur)
  - Fichier créé s'il n'existe pas (*cf.* aussi creat(2))
  - `man 2 open` pour plus de détails
- Il faut **fermer** les fichiers après usage :
  `close (fd)`
  - Descripteur `fd` plus utilisable ; le système peut réallouer ce numéro

Interface standard

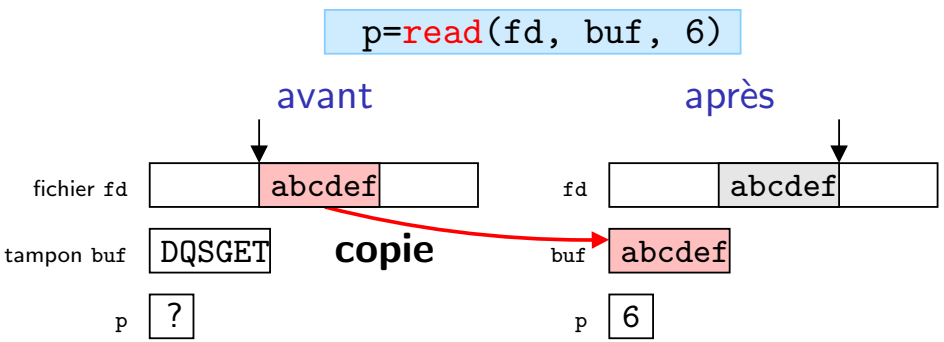
- Il existe une autre interface, plus simple (on y revient)

Interface de programmation POSIX : `read()`

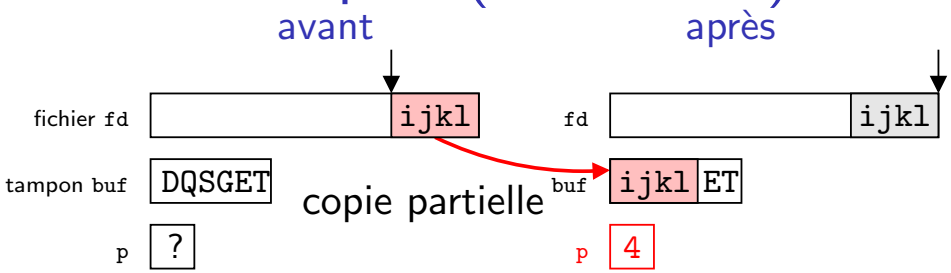
L'objet fichier

- Chaque fichier a un pointeur courant : tête de lecture/écriture logique
- Les lectures et écritures le déplacent implicitement
- On peut le déplacer explicitement (avec `lseek()` – voir plus loin)

Lecture normale (`read()`)



S'il n'y a pas assez de données : lecture tronquée (*short read*)



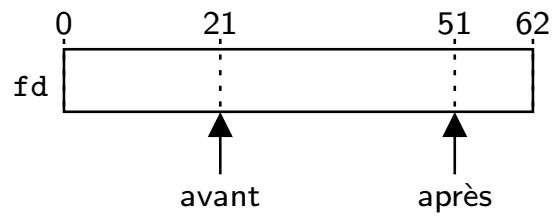
Interface de programmation POSIX : `lseek()`

Objectif

- Permet de déplacer le pointeur de fichier explicitement

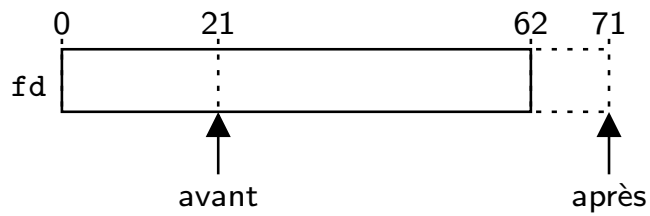
Exemples

`lseek(fd, 30, SEEK_CUR)`



+30 octets depuis position courante

`lseek(fd, 71, SEEK_SET)`



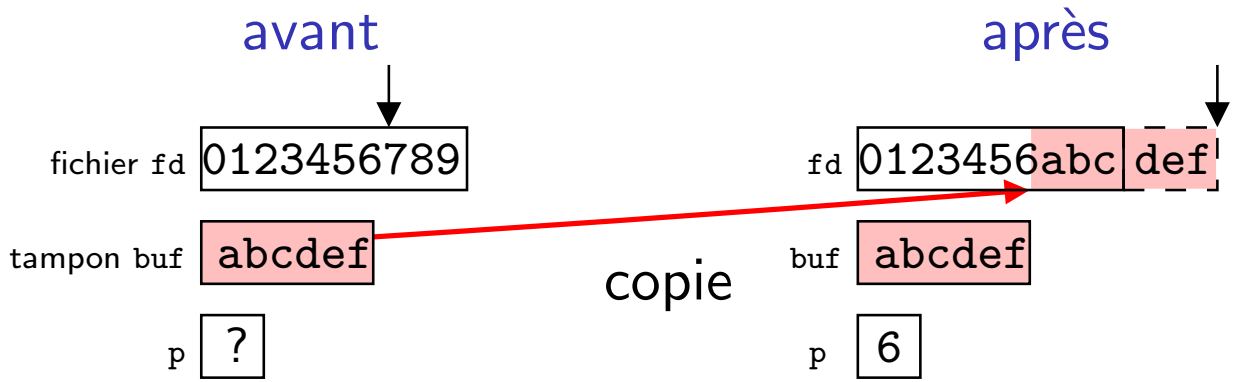
place le pointeur à la position 71

- Le pointeur peut être placé après la fin du fichier

Interface de programmation POSIX : `write()`

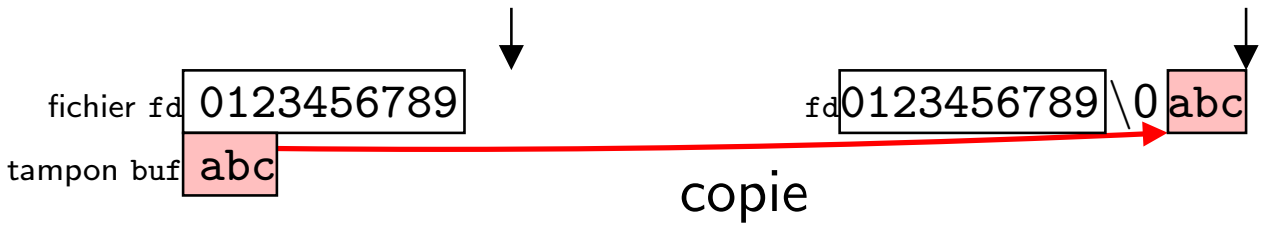
- Écriture dans les conditions normales :

```
p=write(fd, buf, 6)
```



Le fichier a été rallongé

- Si un `lseek()` a déplacé le pointeur après la fin, remplissage avec des zéros (qui ne sont pas immédiatement alloués par le SGF)



- Possibilité d'écriture tronquée (*short-write*) si le disque est plein, ou si descripteur est un tube ou une socket (*cf.* plus loin)

Différentes interfaces d'usage des fichiers

Interface POSIX système

- Appels systèmes `open`, `read`, `write`, `lseek` et `close`.
- Utilisation délicate : lecture/écriture tronquée, traitement d'erreur, *etc.*
- Meilleures performances après réglages précis

Bibliothèque C standard

- Fonctions `fopen`, `fscanf`, `fprintf` et `fclose`.
- Plus haut niveau (utilisation des formats d'affichage)
- Meilleures performances sans effort (POSIX : perfs au tournevis)
- Un syscall $\approx$ 1000 instructions $\Rightarrow$ écritures dans un tampon pour les grouper

```
#include <stdio.h>

FILE *fd = fopen("mon_fichier","w");
if (fd != NULL) {
  fprintf(fd,"Résultat %d\n", entier);
  fprintf(fd,"un autre %f\n", reel);
  fclose(fd);
}
```

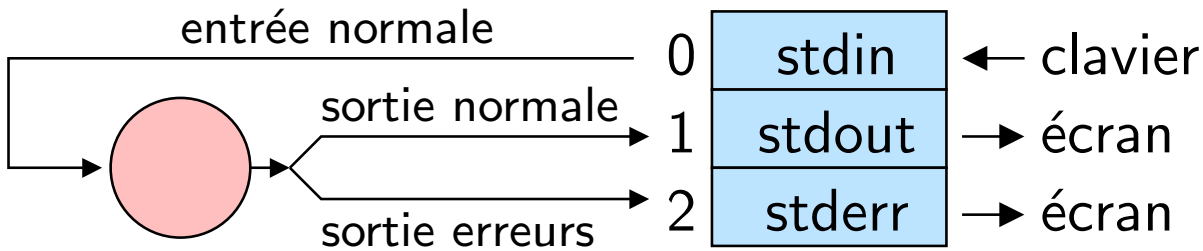
```
#include <stdio.h>

FILE *fd = fopen("mon_fichier","r");
if (fd != NULL) {
  char toto[50];
  fscanf(fd,"%s%d",toto, &entier);
  fscanf(fd,"%s%s%f",toto,toto, &reel);
  fclose(fd);
}
```

Fichiers et flots d'entrée sortie

Sous Unix, tout est un fichier

- Les périphériques sont représentés par des fichiers dans `/dev`
- Tout processus est créé avec des flots d'entrée/sortie standard :
  - stdin : entrée standard (connecté au terminal)
  - stdout : sortie standard (connecté à l'écran)
  - stderr : sortie d'erreur (connecté à l'écran)
- Ces flots sont associés aux descripteurs 0, 1 et 2
- Ils peuvent être fermés ou redirigés vers des ≪vrais≫ fichiers



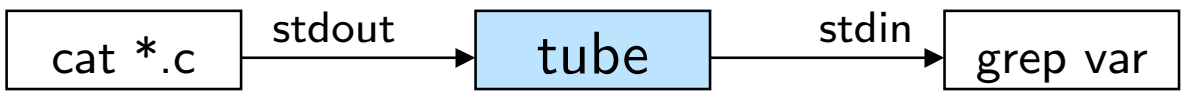
Flots et interface de commande

Rediriger les flots en langage de commande : < et >

```
cat fich # écrit le contenu de fich sur la sortie standard (l'affiche à l'écran)
cat fich > fich1 # copie fich dans fich1 (qui est créé s'il n'existe pas)
./prog1 < entrée # exécute prog en écrivant le contenu de entrée sur son entrée standard
```

Tubes (*pipes*) : moyen de communication inter-processus

- Branche la sortie standard d'un processus sur l'entrée standard d'un autre
- Exemple : `cat *.c | grep var`
  - crée deux processus : `cat *.c` et `grep var`
  - connecte la sortie du premier à l'entrée du second à travers un tube



Exercice : que fait la commande suivante ?

```
cat f1 f2 f3 | grep toto | wc -l > resultat
```

Interface de programmation des tubes

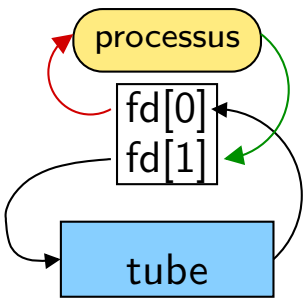
Appel système pipe()

- Crée un tube : deux descripteurs (l'entrée et la sortie du tube)

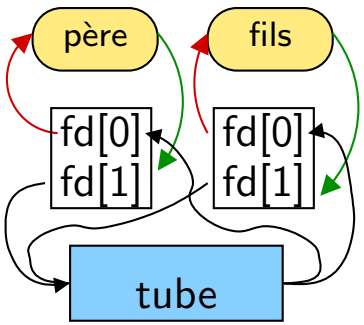
```
int fd[2];
pipe(fd);
```

- Ce qui est écrit sur l'entrée (`fd[1]`) est disponible sur la sortie (`fd[0]`)
  par défaut, cela permet de se parler à soi-même
  Mnémotechnique : On lit sur 0 (stdin), et on écrit sur 1 (stdout)

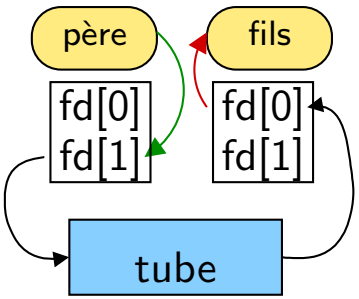
- Application classique : communication père–fils



Après pipe(fd)



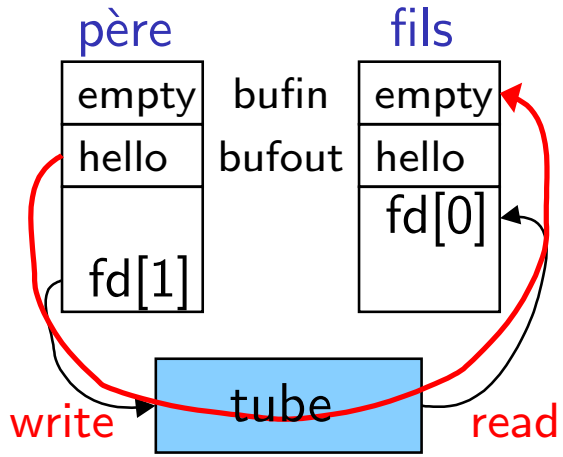
Après fork()
les descripteurs sont copiés



Après fermeture des descripteurs inutiles

Programmation d'un tube père/fils

```
#define BUFSIZE 10
int main(void) {
  char bufin[BUFSIZE] = "empty";
  char bufout[ ] = "hello";
  pid_t childpid;
  int fd[2];
  pipe(fd);
  if (fork() > 0) { /* père */
    close(fd[0]);
    write(fd[1], bufout, strlen(bufout)+1);
  } else {          /* fils */
    close(fd[1]);
    read(fd[0], bufin, BUFSIZE);
  }
  printf("[%d]: bufin = %.*s, bufout = %s\n",
         getpid(), BUFSIZE, bufin, bufout);
  return 0;
}
```




```
$ ./parentwritepipe
[29196]: bufin=empty, bufout=hello
[29197]: bufin=hello, bufout=hello
$
```

Exercice : modifier le programme pour tenir compte des lectures/écritures tronquées

```
int todo = strlen(bufout)+1, done;
char *p=bufout;
while (todo) {
  done = write(fd[1],p,todo);
  todo -= done; p += done;   }
```

```
int done=0; bufin[0]='a';
while (bufin[done]) {
  done += read(fd[0],
               bufin+done,
               BUFSIZE-done);  }
```

Capacité d'un tube

`write(fd, &BUFF, TAILLE)` : écriture d'au plus TAILLE caractères

- S'il n'y a plus de lecteur :
  - Écriture inutile : on ne peut pas ajouter de lecteurs après la création du tube
  - Signal SIGPIPE envoyé à l'écrivain (mortel par défaut)
- Sinon si le tube n'est pas plein : Écriture atomique
- Sinon : Écrivain bloqué tant que tous les caractères n'ont pas été écrits (tant qu'un lecteur n'a pas consommé certains caractères)

`read(fd, &BUFF, TAILLE)` : lecture d'au plus TAILLE caractères

- Si le tube contient n caractères : Lecture de $min(n, TAILLE)$ caractères.
- Sinon s'il n'il a plus d'écrivains : Fin de fichier ; `read` renvoie 0.
- Sinon : Lecteur bloqué jusqu'à ce que le tube ne soit plus vide (jusqu'à ce qu'un écrivain ait produit suffisamment de caractères)

Parenthèse : opérations non-bloquantes

- Drapeau `O_NONBLOCK` $\Rightarrow$ read/write ne bloquent plus jamais :
- Retournent -1 (et errno=`EAGAIN`) si elles auraient dû bloquer
- `fnctl(fd, F_SETFL, fcntl(fd[1], F_GETFL) | O_NONBLOCK);`

Tubes nommés (FIFOs)

- Problème des tubes : seulement entre descendants car passage par clonage
- Solution : tubes nommés (même principe, mais nom de fichier symbolique)

```
#include <sys.types.h>
#include <sys/stat.h>
int mkfifo(char *nom, mode_t mode);
```

  - renvoie 0 si OK, -1 si erreur
  - mode est numérique (on y revient)

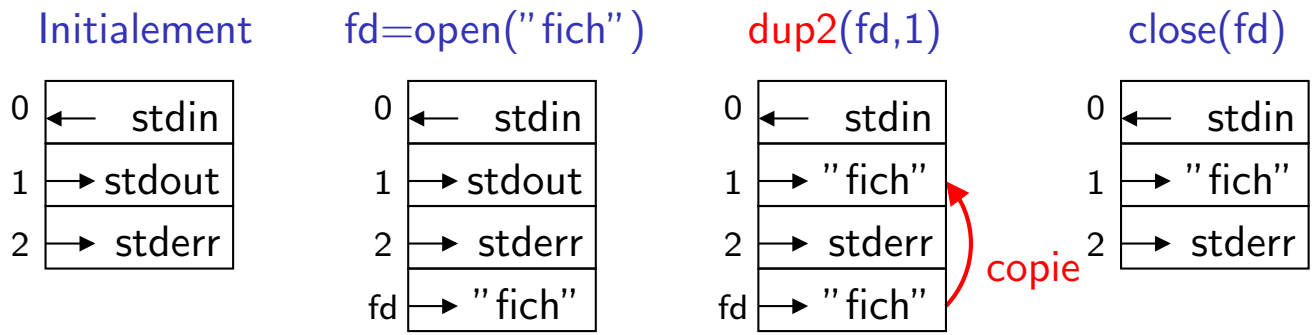
- Après création, un processus l'ouvre en lecture, l'autre en écriture
- Il faut connecter les deux extrémités avant usage (bloquant sinon)

- la *commande* `mkfifo`(1) existe aussi

Copie de descripteurs : dup() et dup2()

`ls > fich`

- Cette commande inscrit dans `fich` ce que `ls` aurait écrit.
- Comment ça marche ?
  On utilise les appels systèmes `dup()` et `dup2()` pour copier un descripteur :

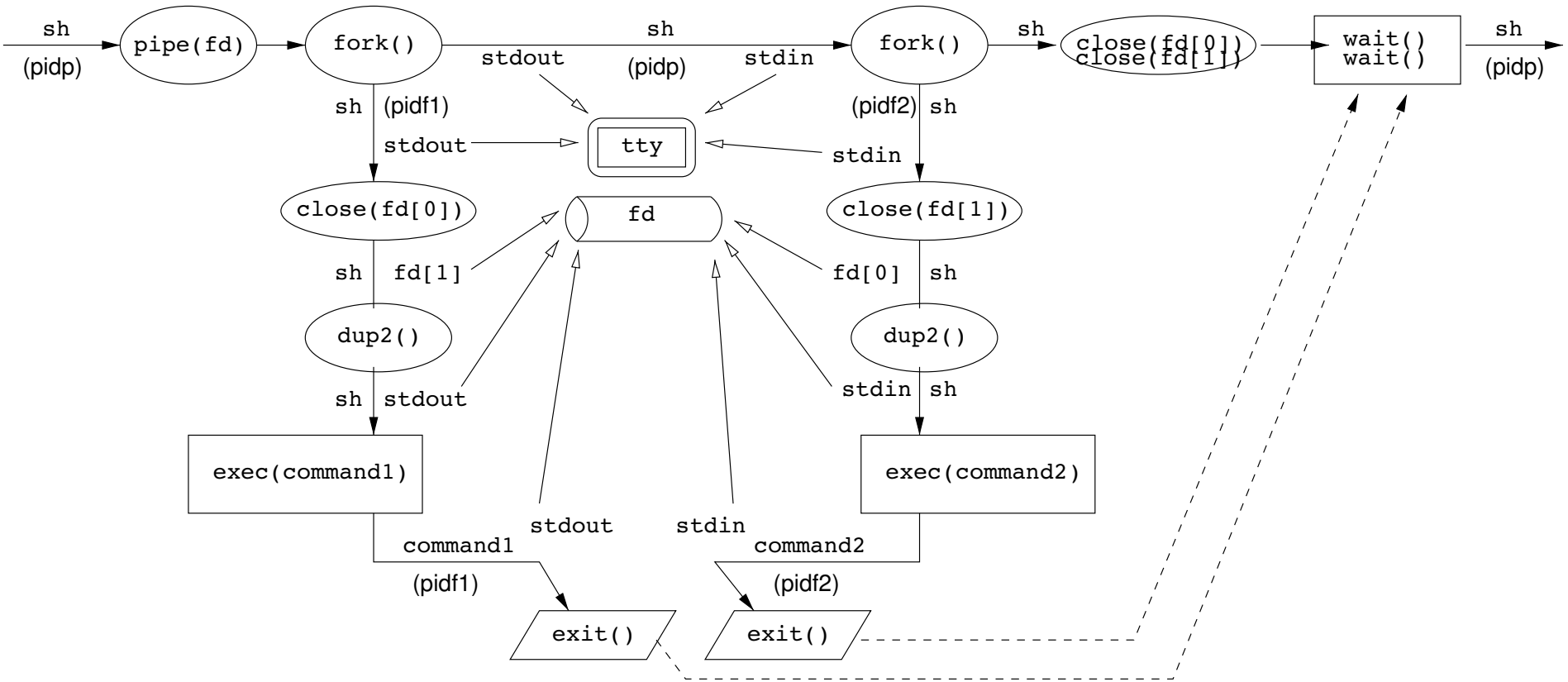


- Appel système dup(fd) duplique `fd` dans le premier descripteur disponible
- Appel système dup2(fd1, fd2) recopie le descripteur `fd1` dans `fd2`
  on dit "dup to", duplicate to somewhere

L'exemple du shell

Création d'un tube entre deux commandes

```
$ commande1 | commande2
```



- (Un peu touffu, mais pas si compliqué à la réflexion :)
- Si on oublie des `close()`, les lecteurs ne s'arrêtent pas (reste des écrivains)

Projection mémoire

Motivation

- Adresser le fichier dans l'espace d'adressage virtuel
  Pratique pour sérialiser des données complexes de mémoire vers disque
- Le fichier n'est pas lu/écrit au chargement, mais à la demande

Réalisation

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

- `addr` : où le mettre, quand on sait. NULL très souvent
- `prot` : protection (PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE)
- `flags` : visibilité (MAP_SHARED entre processus ; MAP_PRIVATE copy-on-write, etc)
- `fd`, offset : Quel fichier, quelle partie projeter.
- Retourne l'adresse (virtuelle) du début du block

Troisième chapitre

Fichiers et entrées/sorties

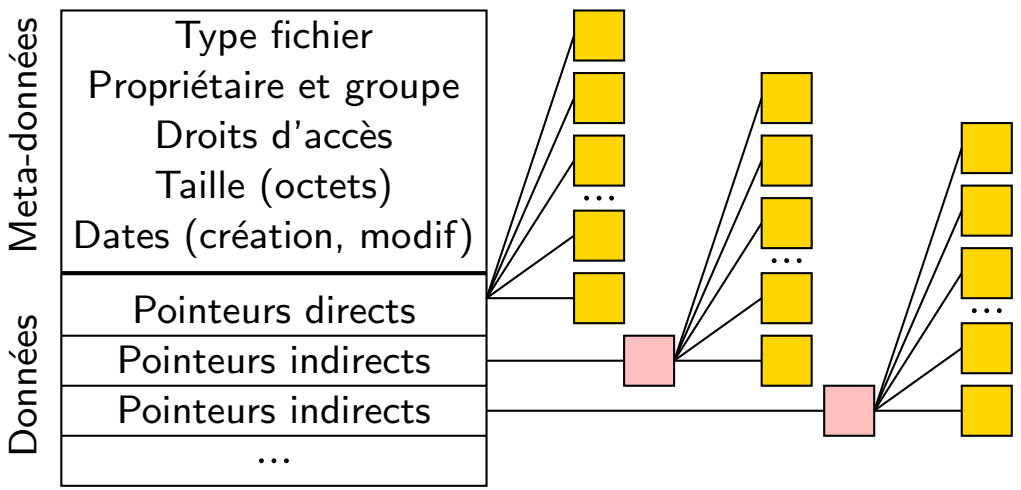
Représentation physique et logique d'un fichier

Représentation physique d'un fichier

- Le disque est découpé en *clusters* (taille classique : 4ko)
- Les fichiers sont écrits dans un ou plusieurs *clusters*
- Un seul fichier par *cluster*

Structure de données pour la gestion interne des fichiers

- Chaque fichier sur disque est décrit par un **inode**
  Structure d'un inode :



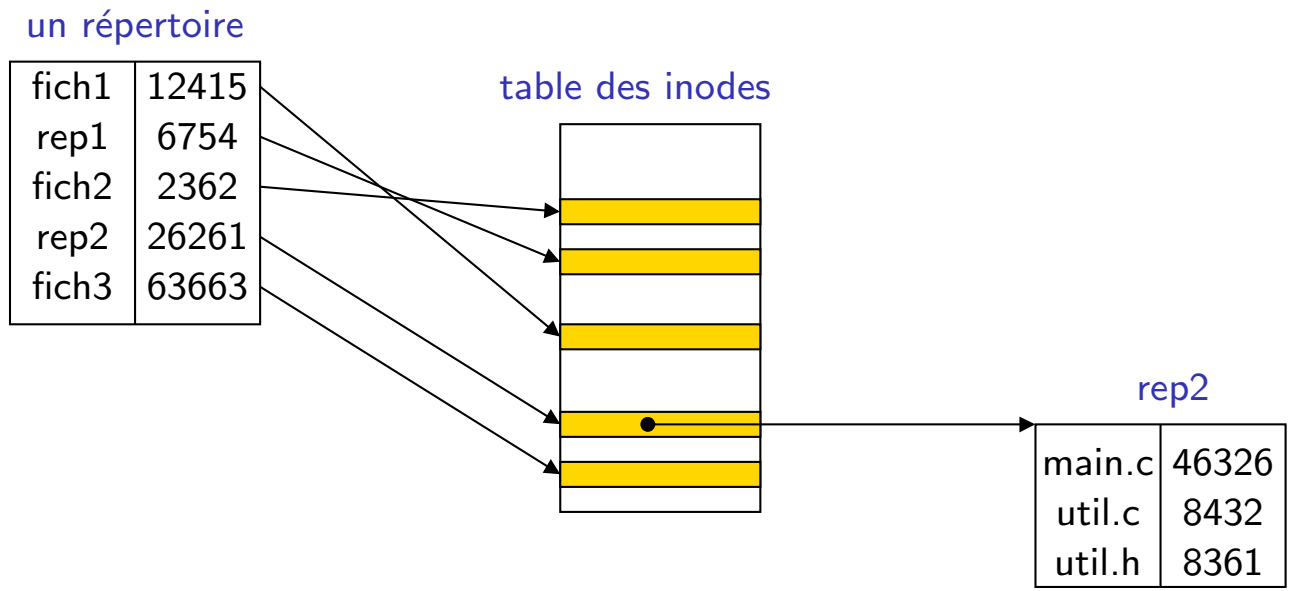
- Pointeurs et tables d'indirection contiennent adresses de *clusters*
- $\approx$ 10 blocs adressés directement $\Rightarrow$ accès efficace petits fichiers (les plus nombreux)
- `stat(2)` : accès à ces info

Noms symboliques et inodes

- Les inodes sont rangés dans une table au début de la partition
- On peut accéder aux fichiers en donnant leur inode (*cf.* `ls -i` pour le trouver)
- Les humains préfèrent les noms symboliques aux numéros d'identification

$\Rightarrow$ notion de répertoire, sous forme d'arborescence (chainage avec racine)

- Les répertoires sont stockés sous forme de fichiers «normaux»
- Chaque entrée d'un répertoire associe nom symbolique $\leftrightarrow$ numéro d'inode



Manipulation de répertoires

Appels systèmes opendir, readdir et closedir

- Équivalent de `open`, `read` et `close` pour les répertoires

Exemple : Implémentation de `ls -i .`

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int main() {
  DIR *dirp;
  struct dirent *dp;

  dirp = opendir(".");
  while ((dp = readdir(dirp)) != NULL) {
    printf ("%s\t%d\n", dp->d_name, dp->d_ino);
  }
  closedir(dirp);
}
```

- On pourrait compléter cette implémentation avec `stat`(2)

Protection des fichiers : généralités

Définition (générale) de la sécurité

- confidentialité : informations accessibles aux seuls usagers autorisés
- intégrité : pas de modifications indésirées
- contrôle d'accès : qui a le droit de faire quoi
- authentification : garantie qu'un usager est bien celui qu'il prétend être

Comment assurer la sécurité

- Définition d'un ensemble de règles (politiques de sécurité) spécifiant la sécurité d'une organisation ou d'une installation informatique
- Mise en place de mécanismes de protection pour faire respecter ces règles

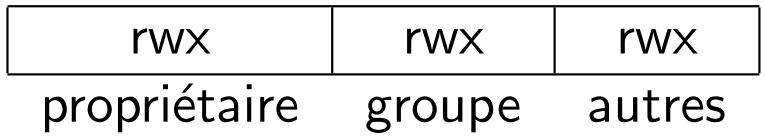
Règles d'éthique

- Protéger ses informations confidentielles (comme les projets et TP notés !)
- Ne pas tenter de contourner les mécanismes de protection (c'est la loi)
- Règles de bon usage avant tout :
  La possibilité technique de lire un fichier ne donne pas le droit de le faire

Protection des fichiers sous Unix

Sécurité des fichiers dans Unix

- Trois types d'opérations sur les fichiers : lire (r), écrire (w), exécuter (x)
- Trois classes d'utilisateurs vis à vis d'un fichier :
  propriétaire du fichier ; membres de son groupe ; les autres



  Granularité plus fine avec les Access Control List (cf `getfacl(1)`)
  - Liste d'utilisateurs
- Pour les répertoires, `r` = `ls`, `w` = créer des éléments et `x` = `cd`.
- `ls -l` pour consulter les droits ; `chmod` pour les modifier

Mécanisme de délégation : `setuid`, `setgid`

- Problème : programme dont l'exécution nécessite des droits que n'ont pas les usagers potentiels (exemple : gestionnaire d'impression, d'affichage, `ping`)
- Solution (setuid ou setgid) : ce programme s'exécute toujours sous l'identité du propriétaire du fichier ; identité utilisateur *effective* momentanément modifiée
  - Granularité plus fine : *capabilities* (`getcap(1)`)
- Source de nombreux problèmes de sécurité
  CVE-2005-2748 : The malloc function in the libSystem library in Apple Mac OS X 10.3.9 and 10.4.2 allows local users to overwrite arbitrary files by setting the MallocLogFile environment variable to the target file before running a setuid application.
  CVE-2001-1015 : Buffer overflow in Snes9x 1.37, when installed setuid root, allows local users to gain root privileges via a long command line argument.
- Ou souvent utilisé pour escalader à partir d'une faille existante
  CVE-2004-2768 : dpkg 1.9.21 does not properly reset the metadata of a file during replacement of the file in a package upgrade, which might allow local users to gain privileges by creating a hard link to a vulnerable (1) setuid file, (2) setgid file, or (3) device

Résumé du troisième chapitre

- Désignation des fichiers
  - Chemin relatif *vs.* chemin absolu
  - $PATH
- Utilisation des fichiers
  - Interface de bas niveau : open, read, write, close
    Problèmes : I/O tronquée, perfs par manque de tampon
  - Interface standard : fopen, fprintf, fscanf, fclose
  - Trois descripteurs par défaut : stdin (0), stdout (1), stderr (2)
  - Rediriger les flots en shell : $<$, $>$ et $|$
  - Tubes, tubes nommés et redirection en C : pipe(), mkfifo(), dup() et dup2()
- Réalisation et manipulation des fichiers et répertoires
  - Notion d'inode
  - Manipulation des répertoires : opendir, readdir, closedir
- Quelques notions et rappels de protection des fichiers