

# Chapitre 2 (Partie 2) - Système

## Système de Gestion des Fichiers

Vincent Colotte

*Département Informatique - Faculté des Sciences  
Université de Lorraine - Nancy*



*mars 2023*

## 5 Introduction

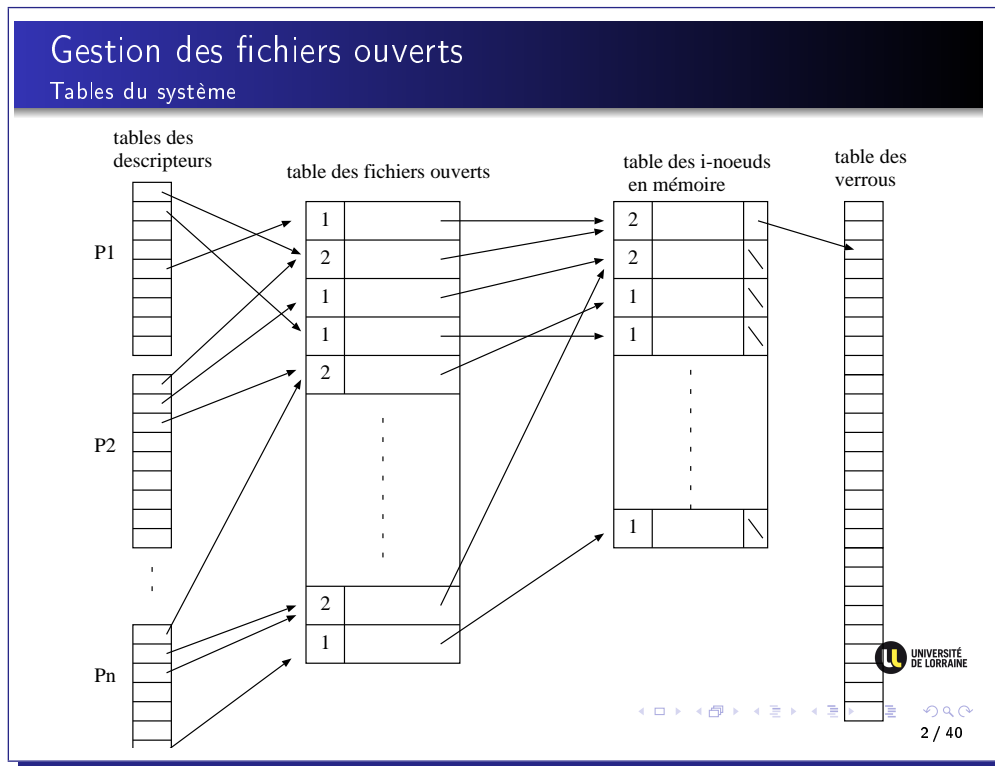
Le fichier est un élément central dans un système d'exploitation. Dans la partie précédente de ce chapitre nous avons vu comment les systèmes en général géraient les fichiers. Dans cette partie nous allons nous focaliser sur l'utilisation des fichiers sous Linux.

Tout d'abord nous aborderons un point important sous Unix concernant la gestion des **fichiers ouverts** qui conditionne la manipulation des fichiers. Nous verrons notamment le mécanisme utilisé pour gérer les utilisateurs. Nous aborderons ensuite les primitives de base associées à la manipulation des fichiers. Enfin, nous décrirons un mécanisme important de communication sous Linux : le **tube**.

## 6 Gestion des fichiers ouverts

### 6.1 Les tables sous Linux

Le schéma suivant illustre l'organisation générale des tables mises en jeu dans un système Linux : les tables des descripteurs, la table des fichiers ouverts, la table des i-noeuds en mémoire, et la table des verrous.



#### 6.1.1 La table des i-noeuds en mémoire

### La table des i-noeuds

i-noeud

Structure sur le **disque dur** contenant l'ensemble des informations d'un fichier présent sur le disque.  
(appelé aussi *i-node* ou noeud d'index)

*Seuls les fichiers ouverts par le système ont leur **i-noeud** chargé en mémoire et sous la forme d'une structure **v-noeud** (\*).*

(\*) le v-noeud (noeud virtuel) est une forme plus abstraite du i-noeud pour gérer plusieurs types de système de fichiers.

Une entrée dans la table contient :

- Le nombre total de façons dont le fichier a été ouvert,
- L'identification du disque logique auquel appartient le i-noeud,
- Les données liées au i-noeud (état du noeud, ...)

UNIVERSITÉ DE LORRAINE  
3 / 40

### 6.1.2 La table des fichiers ouverts

#### La table des fichiers ouverts

Chaque entrée correspond à une façon dont le fichier a été ouvert (mode d'ouverture).

Chaque entrée (structure *file*) contient :

- Le nombre total de descripteurs pointant sur cette entrée,
- Le mode d'ouverture (O\_RDONLY, O\_WRONLY, O\_APPEND, ...),
  - Un fichier peut être ouvert en lecteur et/ou écriture indépendamment.
  - cas particulier : *socket\** (1 entrée : lecture et écriture)
- La position courante (offset),
- Un pointeur sur le i-noeud en mémoire.



### 6.1.3 La table des descripteurs

#### La table des descripteurs

- Une table par processus
- Le programme manipule les fichiers via un indice dans cette table : **le descripteur**  
3 descripteurs particuliers :
  - ★ 0 : l'entrée standard (STDIN\_FILENO)
  - ★ 1 : la sortie standard (STDOUT\_FILENO)
  - ★ 2 : la sortie-erreur standard (STDERR\_FILENO)
- Chaque entrée de la table contient :
  - ★ un pointeur sur le table des fichiers ouverts.

Un processus obtient des descripteurs :

- par héritage (père/fils : *fork*)
- par l'intermédiaire de primitives (*open*, *dup...*)



### 6.1.4 Verrouillage des fichiers

(pour la mise en place voir 6.3.6)

#### Verrouillage des fichiers

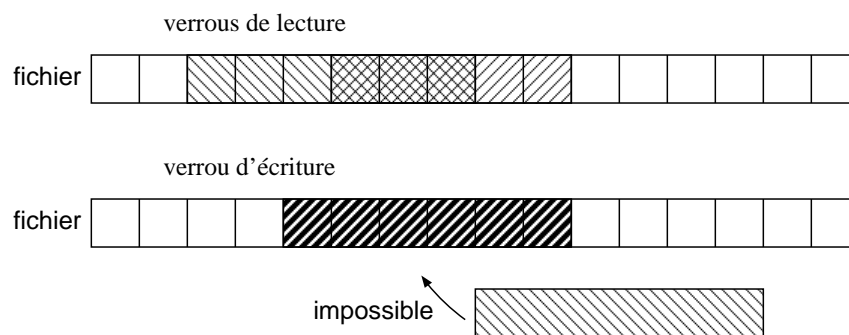
*Un verrou permet de limiter ou d'interdire certaines opérations sur tout ou une partie d'un fichier.*

- Il appartient à un processus **propriétaire** et associé à un **i-noeud**.
- Il existe 2 types de verrous :
  - **partagés ou de lecture** : plusieurs verrous de ce type peuvent cohabiter (plusieurs processus peuvent lire mais pas écrire).
  - **exclusifs ou d'écriture** : aucune cohabitation possible (même avec un partagé) (un seul processus peut écrire les autres sont bloqués même pour une lecture)
- 2 modes :
  - **impératif** : la pose d'un verrou peut être bloquée et bloquante sur une lecture ou une écriture par un *read* ou *write*.
  - **consultatif** : seule la pose du verrou est gérée, n'influe pas la lecture et l'écriture (processus coopératif).



8 / 40

#### verrouillage des fichiers



9 / 40

## 6.2 Droits et sécurité des fichiers


Droits sous Linux


Utilisateurs

- Tout utilisateur a un identifiant : UID (*UserIDentification*)
- Il appartient à un ou plusieurs groupe : GID
- Tous les processus et fichiers créés reçoivent l'UID et GID de leur propriétaire.

- Les droits sur les fichiers : 3 catégories  
⇒ Propriétaire (*user*) / Groupe (*group*) / Autres (*other*)
- Pour chaque catégorie : rwx (3 bits)
  - r (*read*) : lecture
  - w (*write*) : écriture
  - x (*execution*) : exécution  
(pour un répertoire : x= droits de traverser le répertoire)





10 / 40

Exemples :

rw- r-- r-- (0644)	
rwX rwx --- (0770)	

Les 3 bits rwx peuvent être représentés par un nombre octal en utilisant la somme, si le symbole est présent, de  $r=2^2=4$ ,  $w=2^1=2$ ,  $x=2^0=1$ . Ainsi  $rwX=1*4+1*2+1*1=7$ ,  $r--=1*4+0*2+0*1=4$ , ...

On peut donc représenter des droits sur 9 bits par un octal ( $rw- r-- r-- = 0644$ ).

Exemple :

```
ls -la fichier.txt :
```

```
-rwX r-x r-x 1 kevin etud 896 janv. 29 20:51 fich.txt
```



Modification des droits (en Shell) :

```
chmod 664 fich.txt ou chmod u=rwx, g=rwx,o=rx fich.txt
```

```
-rwX rwx r-x 1 kevin etud 896 janv. 29 20:51 fich.txt
```

## Protection sous Linux

### Protection ressources système

- Accès à des ressources critiques ???
- Normalement seul le super-utilisateur (root)! [UID=0]
- Exemple : fichier `/etc/shadow` contenant les mots de passe.

### Bit de protection : set-uid

- À l'exécution d'un fichier binaire (si positionné)
- Donne UID du propriétaire du fichier binaire
- On parle alors UID **effective\*** du processus ( $\neq$  UID réelle).
- Les droits sont conditionnés par l'UID effective d'un processus.

[si le bit set-uid n'est pas positionné l'UID effective reste identique à l'UID réelle]



## Protection sous Linux

**Exemple :** le binaire `passwd` change le mot de passe dans `/etc/shadow`

```
⇒ -rws r-x r-x root root /usr/bin/passwd
⇒ -rw- r-- --- root shadow /etc/shadow
```

Processus qui fait `passwd` :

- UID réelle = 1234, donc normalement `/etc/shadow` est inaccessible (---)
- set-uid positionné  $\Rightarrow$  UID effective devient 0 (=root)
- `/etc/shadow` est accessible et modifiable pour `root` et donc aussi pour le processus  
... mais seulement par cette fonction (l'accès est donc sous "contrôle").

(le `s` a été placé par root avec : `chmod u+s passwd`)



Lors de la vérification d'accès à un fichier, le système regarde l'UID effective du processus qui demande cette accès. En d'autres termes, si le bit set-uid n'est pas positionné, l'UID effective est donc l'UID réelle du processus.

Le bit `set - gid` existe aussi pour donner une **GID effective**. Mais il est corrélé au bit set-uid et rarement utilisé seul.

## Fonctions C

Quelques appels système pour accéder/modifier les droits :

- `chmod(...)` : Change les protections (existe comme commande en shell)
- `access(...)` : vérifie l'accès avec les UID et GID réels
- `getuid()`, `geteuid()` : récupère UID réelle et effective
- `getgid()`, `getegid()` : récupère GID réelle et effective
- `setuid(...)` : place le bit de protection

## 6.3 Les primitives de base d'opérations sur les fichiers (POSIX)

### 6.3.1 Ouverture d'un fichier : *open*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char * ref, int mode_ouv, ... /* les droits */);
```

#### Open

```
int open (const char * ref, int mode_ouv, ... /* les droits */);
```

#### Open

Déclare au système le type des opérations réalisables sur le fichier en cas de réussite de l'ouverture :

- Charge dans la table des i-noeuds en mémoire (i-noeud désigné par *ref*).
- Allocation d'une nouvelle entrée dans la table des fichiers ouverts du système.
- allocation d'un descripteur dans la table lié au processus.

Il renvoie le **descripteur** (-1 sinon).

Les constantes du mode d'ouverture sont ajoutées par disjonction (ex : O\_WRONLY|O\_APPEND|O\_CREAT|0777).



Les constantes du mode d'ouverture sont ajoutées par disjonction et sont de 3 types :

- Obligatoire : O\_RDONLY, O\_WRONLY ou O\_RDWR (inscrit dans la table de fichiers ouverts mais non modifiable)
- Optionnelle : utile à l'ouverture seulement (non inscrit dans la table des fichiers ouverts) (O\_TRUNC le fichier est mis à zéro, O\_CREAT création avec les droits)
- Optionnelle mémorisable : inscrit dans la table et modifiable par la suite (O\_APPEND écriture à la fin, O\_NONBLOCK open non bloquant (pour tube ou terminaux), O\_SYNC bloque le processus jusqu'à l'écriture effective sur le disque)



### 6.3.2 Fermeture d'un fichier : *close*

#### Close

```
#include <unistd.h>
int close (int descr);
```

- enlève les verrous mis par le processus sur le fichier,
- décrémentation de l'entrée dans la table des fichiers ouverts,
- si nul, l'entrée est libérée et le i-noeuds est décrémentée.
- ...



### 6.3.3 Lecture d'un fichier : *read*

#### Read

```
#include <unistd.h>
ssize_t read(int descr, void *buffer, size_t nb_octets);
```

#### Read

Demande la lecture d'au plus *nb\_octets* caractères dans le fichier ayant pour descripteur *descr*. Les caractères lus sont placés dans *buffer*. La fonction renvoie le nombre de caractères effectivement lus.

Remarques :

- lit à partir de la position courant
- l'**offset** est incrémenté du nombre de caractères effectivement lus.
- S'il y a un verrou exclusif impératif sur le fichier dans la portée de lecture : le processus est bloqué (sauf si **O\_NONBLOCK** ou **O\_NDELAY**).



### 6.3.4 Ecriture d'un fichier : *write*

#### Write

```
#include <unistd.h>
ssize_t write (int descr, void *buffer, size_t nb_octets);
```

#### Write

Demande d'écriture des `nb_octets` caractères, lus dans `buffer`, dans le fichier dont le descripteur est `descr`. La fonction renvoie le nombre de caractères effectivement écrits dans le cache du noyau (ou sur le disque si `O_SYNC`).

Remarque :

- S'il y a un verrou exclusif impératif sur le fichier dans la portée d'écriture : le processus est bloqué (sauf si `O_NONBLOCK` ou `O_NDELAY`).



### 6.3.5 Manipulation de la position : *lseek*

#### lseek

```
#include <unistd.h>
off_t lseek (int descr, off_t position, int origine);
```

#### lseek

- Modifier la position courante (ou offset) de l'entrée de la table des fichiers ouverts
- Position courante = *origine* + *position*
- *origine* : `SEEK_SET`, `SEEK_CUR`, `SEEK_END`
- Valeur renvoyée : nouvelle position courante (entier relatif).  
(`off_t`) -1 : erreur (dépassement de la fin du fichier ou avant 0)

#### À noter

- Certains fichiers n'autorisent pas son utilisation (tube, terminaux...)
- Cette fonction est utilisée pour accéder à des données non séquentiellement.

Valeur des constantes pour l'argument *origine* :

SEEK_SET	début de fichier (0)
SEEK_CUR	position courante (avant l'appel)
SEEK_END	taille du fichier (avant l'appel)

### 6.3.6 Contrôle des fichiers : *fcntl* et *fstat*

fcntl


```
#include <fcntl.h>
int fcntl (int descr, int commande, ...);
```

**fcntl**

permet de retrouver ou de modifier les attributs des différentes entrées liées à *descr*.

- La fonction *fcntl* permet de pratiquement tout faire sur les fichiers.
- Le verrouillage des fichiers est réalisé avec cette fonction.

Exemple : *F\_GETFL*, *F\_SETFL* : Récupération/modifications des attributs liés au mode d'ouverture.



UNIVERSITÉ DE LORRAINE

21 / 40

Le verrouillage des fichiers (voir 6.1.4) est réalisé avec cette fonction :

La structure avec les attributs du verrou :

```
#include <sys/types.h>
#include <fcntl.h>
struct flock {
    short l_type      /* valeur : F_RDLCK (verrou partagé)
                      F_WRLCK (verrou exclusif)
                      F_UNLCK (dévrouillage) */
    short l_whence    /* SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start      /* pos. relative de début de la portée par rapport à l_whence */
    off_t l_len        /* longueur ( 0 = fin du fichier ) */
    pid_t l_pid        /* pid du processus propriétaire du verrou */
};
```

## Utilisation

```
struct flock verrou;  
int rep, descr, commande;  
.  
.  
.  
rep = fcntl(descr, commande, &verrou);
```

avec pour l'argument *commande* :

**F\_SETLK** : demande de pose non-bloquante (0 :ok, -1 :échec)

**F\_SETLKW** : demande de pose bloquante (0 :ok, -1 :échec (DEAD LOCK))

**F\_GETLK** : teste la incompatibilité du verrou. (si incompatible, infos dans *verrou*)

## fstat et stat

```
#include <sys/types.h>  
#include <sys/stat.h>  
#include <unistd.h>  
int fstat (int descr, struct stat *pStat);  
int stat (const char *ref, struct stat *pStat);
```

## fstat

permet d'obtenir la structure *stat* à partir de *descr*.

- La structure *stat* : contient les informations propriétaire, droits, date d'accès, numéro d'i-node. . .
- La fonction *stat* : permet d'accéder au fichier par son chemin/nom.
- *fstat* et *stat* renvoient 0 si la récupération est réussie (-1 sinon)

(voir TP)

### 6.3.7 Duplication de descripteur : *dup* et *dup2*

#### Exemple en shell

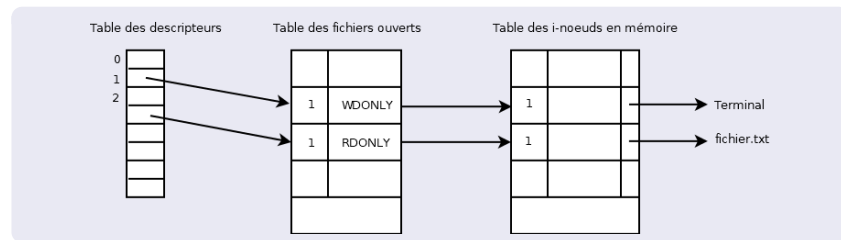
```
cat -n fichier.txt
```

```
cat -n fichier.txt > fichier2.txt
```

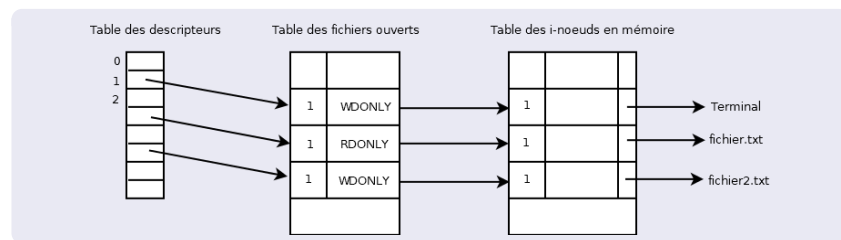


#### Exemple en shell

```
(cat -n fichier.txt)
```



```
(cat -n fichier.txt > fichier2.txt)
```



## Duplication

```
#include <unistd.h>
int dup (int descr); int dup2 (int descr, int descr2);
```

### dup et dup2

Duplique le descripteur *descr*, en lui attribuant le plus petit descripteur disponible ou *descr2*.

⇒ Il crée un descripteur synonyme qui pointe vers la même entrée.

### Mécanisme de redirection

- Modification de l'association des descripteurs avec les fichiers physiques.
- Par exemple rediriger la sortie standard (ou d'erreur) dans un fichier sans modifier le programme initial. ⇒ La redirection est transparente pour le programme.

VERSITÉ  
ORRAINE

25 / 40

## 6.4 Fonctions de la bibliothèque standard : fopen, fscanf, fprintf et fclose

La bibliothèque C d'entrées/sorties proposent la manipulation des fonctions précédentes à un niveau plus élevé. Le fichier n'est plus directement manipulé par un descripteur et offre ainsi une portabilité plus importante. Leur utilisation reste très proche des fonctions systèmes.

## Bibliothèque standard C

Fonctions de manipulation des fichiers :

- Appel aux fonctions système (open, read, ...) : portabilité plus importante.
  - fopen → open
  - fprintf → write
  - fscanf → read
  - fclose → close
  - ...
- Manipulation d'un objet de type FILE (lié au descripteur)  
*FILE \* file = fopen("monfichier.txt", "r");*
- Permet l'utilisation de formatage  
*fprintf(file, "%s \n", chaine );*
- Sorties standard : stdin, stdout, stderr

### À noter

- La bibliothèque C a son propre cache par utilisateur : *fflush()* pour le transférer sur celui du système.
- Ne pas mélanger les utilisations (write/fprintf/put) (effets de bord possible).

VERSITÉ  
ORRAINE

26 / 40

## 7 Communication par tube

### Communication par un tube

#### Idée

- Échange d'information entre 2 processus à l'aide d'un fichier
- Principe simple à mettre en oeuvre.
- Linux : propose de faciliter ce mode de communication
- Utilisation dans le Shell avec `|` (pipe)

#### Exemple

```
ls /home/user/toto/ | wc -l
```

- `ls` repertoire :
- `wc -l` fichier :
- `|` :

### 7.1 Propriétés

### Communication par un tube

#### Un tube

- fait partie du système de fichier.
- lecture et écriture avec `read()` et `write()`
- unidirectionnel : une extrémité pour l'écriture et une pour la lecture.
- mode *stream* : lecture/écriture par octet (par opposition au mode par structure).
- la lecture est destructrice.
- un tube a une capacité : un tube peut donc être plein.

## Communication par un tube

### Propriétés :

**nbre de lecteurs** : nbre de descripteurs associés à l'entrée en lecture.

→ si nbre de lecteur = 0, alors pas d'écriture (le signal SIGPIPE est envoyé).

**nbre d'écrivains** : nbre de descripteurs associés à l'entrée en écriture.

→ si nbre d'écrivain = 0 et le tube est vide alors c'est équivalent à fin de fichier.  
(s'il ne peut devenir nul, il y a interblocage)



## 7.2 Les tubes « anonymes » : pipe

```
#include <unistd.h>  
int pipe (int *tdesc); /* tdesc[2] : tableau de deux entiers préalablement alloués */
```

## Tubes « anonymes »

*Ces tubes sont des fichiers **sans référence**, ils ne peuvent être utilisés que par des processus de **même filiation** et n'être manipulés que par des descripteurs.*

### Intérêt

- Un processus (père) crée un tube puis crée un processus fils
- Les descripteurs sont hérités  $\Rightarrow$  le fils est directement connecté au tube
- Reste à transmettre le numéro du descripteur pour la lecture ou l'écriture (à la création du fils)
- La destruction est gérée par le système (après la fermeture (*close*) du dernier descripteur en mémoire).





## Tubes « anonymes »

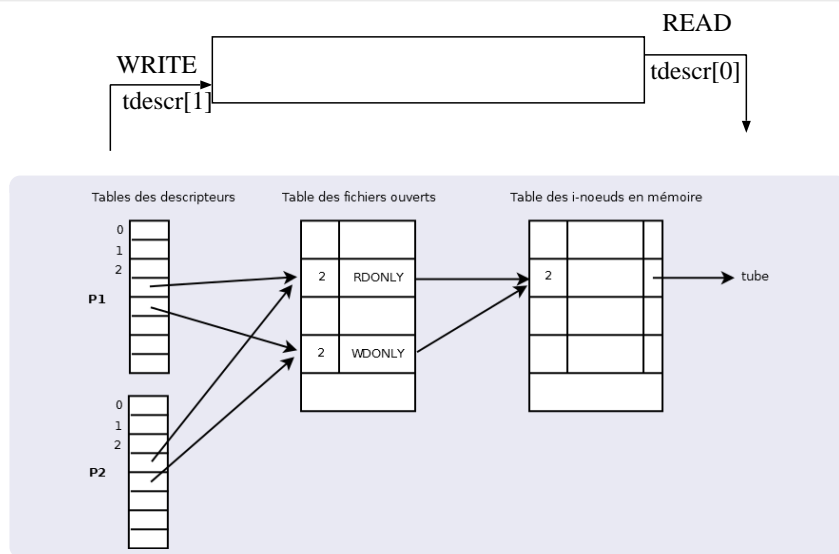
```
int pipe (int *tdesc) ;
```

crée un nouveau tube :

- alloue un (i-)noeud,
- 2 entrées dans la table des fichiers ouverts,
- 2 descripteurs : *tdesc*[0] pour la lecture et *tdesc*[1] pour l'écriture.

### À noter

- L'écriture se fait en fin de tube/fichier.
- La lecture en début de tube/fichier.



Exemple de mise en place d'un tube anonyme pour l'envoi d'un objet de type *char* :

```
...
char buffer; /* buffer d'un seul caractère */
int descr_tube[2];
...
pipe(descr_tube); /* création d'un tube */
...
/* dans le code du processus écrivain : */
/* Fermeture du bout pour lecture pour lui */
close(descr_tube[0]);
/* écriture (buffer est une suite d'un ou plusieurs octets ici de taille char) */
write(descr_tube[1], &buffer, sizeof(char));
...
/* fin de l'écriture */
close(descr_tube[1]);
/* dans le code du processus lecteur : */
/* Fermeture du bout pour l'écriture pour lui */
close(descr_tube[1]);
/* lecture (buffer est une suite d'un ou plusieurs octets ici de taille char) */
read(descr_tube[0], &buffer, sizeof(char));
...
/* fin de lecture */
close(descr_tube[0]);
```

### 7.2.1 Lecture dans un tube : *read*

Comme des fichiers classiques, un tube est lu avec la primitive *read*. Cette primitive garde les mêmes fonctionnalités et notamment la lecture est bloquante par défaut.

#### Lecture

Avec *read()* :

- ❶ si le tube n'est pas vide  $\Rightarrow$  lecture du nbre de caractères demandés (ou moins si pas assez)
- ❷ si le tube est vide
  - $\rightarrow$  si nbre d'écrivains = 0  $\Rightarrow$  EOF, renvoie 0.
  - $\rightarrow$  si nbre d'écrivains > 0
    - lecture bloquante  $\Rightarrow$  attente de remplissage
    - lecture non-bloquante  $\Rightarrow$  retour et renvoie de -1.

### 7.2.2 Écriture dans un tube : *write*

De la même manière, l'écriture dans un tube est réalisée avec la primitive *write*. La taille du tube est limitée (PIPE\_BUF) cependant le système garantit une écriture atomique (en une fois et une seule) d'un buffer qui est plus petit que PIPE\_BUF. Il est déconseillé d'écrire des buffers plus grand que la taille du tube.

#### Ecriture

Avec *write()* :

- ❶ si **nbre de lecteur = 0**  $\Rightarrow$  envoie du signal SIGPIPE (message "Broken Pipe")  
Comme plus de lecteur et qu'il ne peut plus y en avoir, l'écrivain devient inutile (terminaison du processus ou retour -1).
- ❷ si **nbre de lecteur > 0**
  - écriture bloquante  $\Rightarrow$  écrit en une seule fois sinon attend que le tube se vide suffisamment.
  - écriture non-bloquante :
    - a. si  $n \leq \text{PIPE\_BUF}$ , et au moins  $n$  emplacements libres dans le tube, écriture.
    - b. si  $n \leq \text{PIPE\_BUF}$ , et pas assez de place dans le tube  $\rightarrow$  pas d'écriture et retour -1.
    - c. si  $n > \text{PIPE\_BUF}$ , écriture d'un nbre inférieur à  $n$  (et retour -1!?).



### 7.3 Les tubes nommés : FIFO

#### Tubes nommés

Contrairement aux tubes anonymés, les tubes nommés sont référencés dans le SGF

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

#### Propriétés

- Création avec *mkfifo*.  
(ex : `mkfifo("./fichier.txt", 0666);`)
- Ouverture avec *open\**  
(ex : `open("./fichier.txt", O_RDONLY);`)
- Lecture ou écriture comme tube anonyme
- Il faut supprimer physiquement le tube (*unlink* ou *rm*).
  - *close* ne suffit pas


Contrairement aux tubes anonymes, les tubes nommés sont référencés dans le SGF. Avec le commande *ls -l*, ils apparaissent avec le caractère *p* comme type de fichier.


### Tubes nommés : ouverture

Ouverture avec *open* : ouverture en lecture ou en écriture  
**bloquante**  
⇒ permet la synchronisation à l'ouverture

#### Propriétés

- ouverture en lecture bloquante en cas :
  - d'absence d'écrivain
  - de processus bloqué sur une ouverture en écriture.
- ouverture en écriture bloquante en cas :
  - d'absence de lecteur
  - de processus bloqué sur une ouverture en lecture.

 UNIVERSITÉ DE LORRAINE

 36 / 40

#### 7.4 Accès aux caractéristiques d'un tube par *fcntl* et *fstat*

La fonction *fcntl* peut être utilisée pour changer les attributs d'un tube (nommé ou anonyme), notamment pour rendre une lecture non-bloquante.

Exemple de modification :

```
#include <fcntl.h>;
int status_lecture, tube[2];
:
pipe(tube);
:
status_lecture = fcntl(tube[0], F_GETFL);
fcntl(tube[0], F_SETFL, status_lecture | O_NONBLOCK);
:
```

La fonction *fstat* permet de récupérer les caractéristiques d'un tube (nommé) à partir du descripteur de ce dernier. Le résultat est placé dans une structure classique de fichier : *stat*. Prenons par exemple le pointeur *pStat*, pointant sur une structure *stat* ; on peut tester la nature du fichier (champ *st\_mode* de la structure) et notamment pour savoir si ce dernier est un tube : *S\_ISFIFO(pStat → st\_mode)* renvoie un entier  $> 0$  si vrai.