Système d'exploitation – avancé

Programmation système

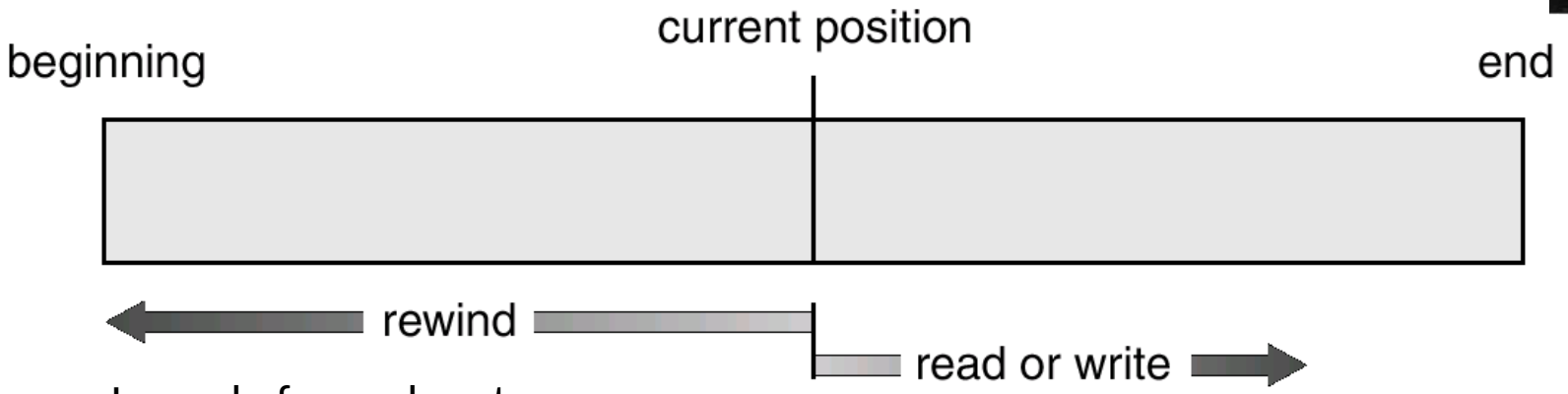
Laurent Mascarilla

2024-2025

« Sous Unix tout est fichier…»

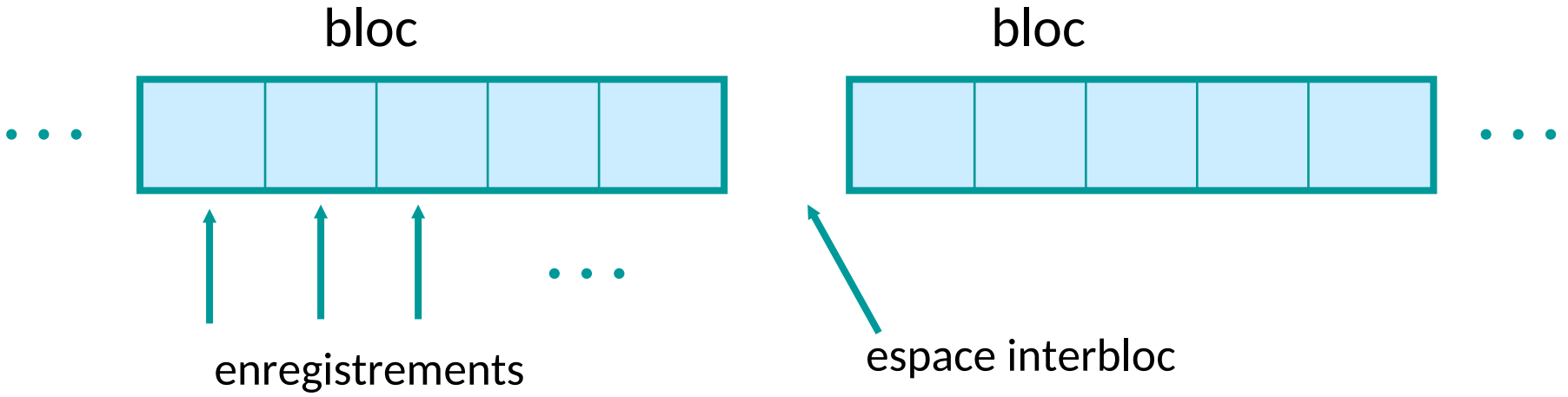
Fichiers à accès séquentiel

(archétype: rubans)



La seule façon de retourner en arrière est de retourner au début (rembobiner, rewind)

En avant seulement, 1 seul enreg. à la fois



Bibliothèque standard d'E/S

- **stdio.h**
  - fopen()
  - fread()
  - fwrite()
  - fclose()
  - fflush()
  - ……

    Déjà vue en cours de C : manipulent des **FILE * aussi appelés flux**
- Structure **FILE** dans stdio.h
  - pointeurs du tampon du fichier
  - n° de descripteur

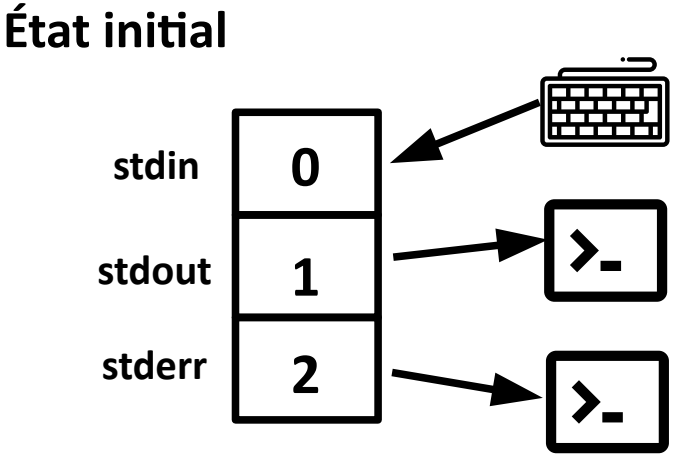
- Principe du tampon (buffer)
  - en lecture:
    - tampon se remplit par une lecture
    - les caractères sont récupérés dans ce tampon jusqu ’à épuisement
  - en écriture
    - les caractères remplissent le tampon
    - le tampon plein est vidé par une écriture
  - Mémoire cache avant écriture disque

Fonctions systèmes d'E/S

- Appels système: read(), write(), open(), close()
  - accès par les descripteurs
    - Descripteur = entier != FILE *
  - définition des zones de réception/émission
  - nb d 'octets transmis
  - pas de formatage des E/S, pas de conversion

- Ces fonctions sont utilisées par les fonctions E/S standards

- Par défaut un processus dispose des descripteurs suivants :
  - stdin dont le fd est 0 : entrée standard (clavier)
  - stdout dont le fd est 1 : sortie standard (terminal)
  - stderr dont le fd est 2 : sortie d'erreur standard (terminal)



Par habitude en C un file descriptor est appelé fd ou fildes

Fonctions système E/S sur un fichier

- Ouverture d'un fichier : **open**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *chemin,
         int typeOpen,
         mode_t droitsFic);
```

Nom du fichier à ouvrir

Mode d'ouverture :
O_RDONLY
O_WRONLY
O_RDWR
O_APPEND
O_CREAT

Si création : droits appliqués à la création (p. ex. 0750)

Descripteur du fichier
ou **–1** en cas d'erreur
et errno est mis à jour

mode/flags

Mode fopen()	Flags open()	Effet
r		
w		
a		
r+		
w+		
a+		

- O_APPEND

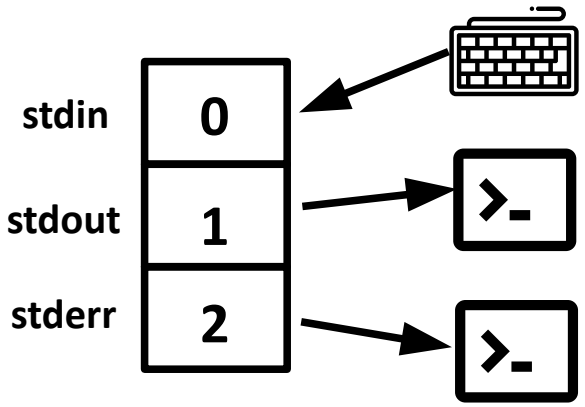
  Le fichier est ouvert en mode « ajout »: la tête de lecture/écriture est placée à la fin du fichier
- O_CREAT

  Créer le fichier s'il n'existe pas.
- O_EXCL

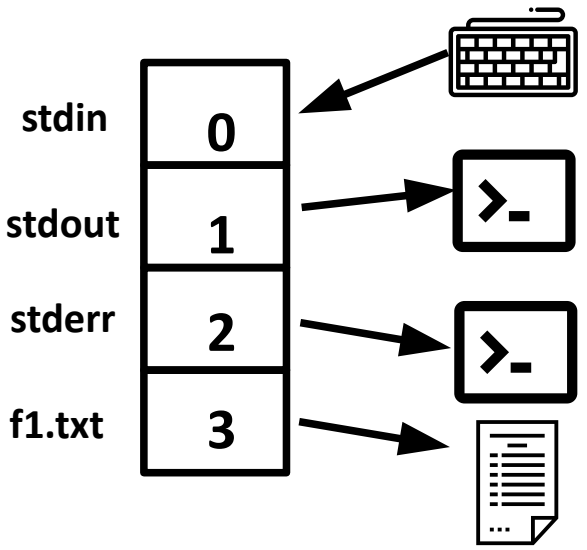
  S'assurer que cet appel crée le fichier : si cet attribut est spécifié en conjonction avec O_CREAT et si le fichier existe déjà, open() échouera.
- O_TRUNC

  Si le fichier existe, est un fichier régulier, et est ouvert en écriture (O_RDWR ou O_WRONLY), il sera tronqué à une longueur nulle.
- ....

État initial



fildes=open("f1.txt", O_WRONLY |O_CREAT,0666) ;



- Les descripteurs sont stockés dans une table (table des descripteurs!) propre à chaque processus

- Création d'un fichier : **creat**

creat() est équivalent à open() avec l'attribut flags égal à **O_CREAT | O_WRONLY | O_TRUNC** .

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

Nom du fichier à créer

Droits appliqués à la création

Descripteur ou –1 en cas d'erreur et errno est mis à jour

- Lecture dans un fichier : **read**

```
#include <unistd.h>
int read(int desc,
         void *buffer,
         size_t nb);
```

Descripteur du fichier

Adresse de la variable dans laquelle on stocke le résultat de la lecture

Nombre d'octets à lire

Nb d'octets lu
-1 en cas d'erreur
0 si fin de fichier
et errno est mis à jour

Attention : **read** lit et déplace le curseur !

- Ecriture dans un fichier : **write**

```
#include <unistd.h>
int write(int desc,
          void *buffer,
          size_t nb);
```

Descripteur du fichier

Adresse de la variable qui contient ce que l'on veut écrire

Nombre d'octets à écrire

Nb d'octets écrits
-1 en cas d'erreur
et errno est mis à jour

Attention : **write** écrit et déplace le curseur !

- Déplacement de la position courante : **lseek**

```
#include <unistd.h>
off_t lseek(int desc,
            off_t depl,
            int vers);
```

Descripteur du fichier

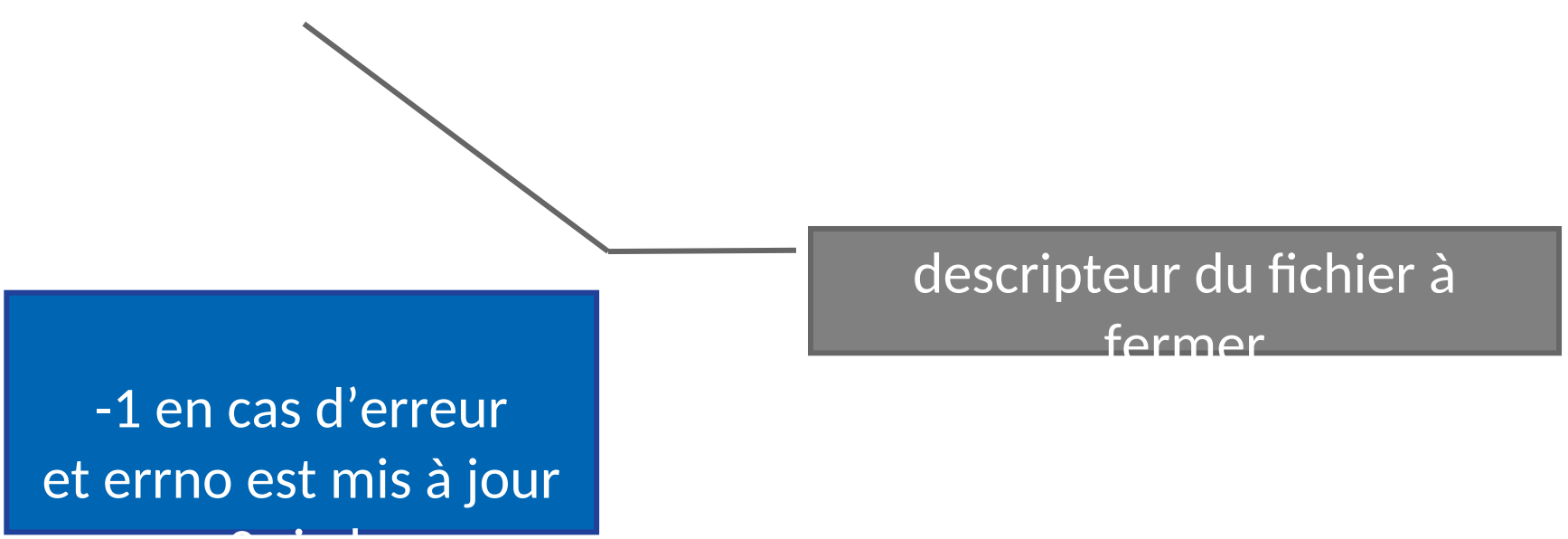
Valeur du déplacement (en octet) par rapport à *vers*

-**SEEK_SET** : déplacement par rapport au début du fichier
- **SEEK_CUR** : déplacement par rapport à la position courante
- **SEEK_END** : déplacement par rapport à la fin du fichier

Nouvelle position par rapport au début du fichier
-1 en cas d'erreur
et errno est mis à jour

- Fermeture : **close**

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



```c
#include<stdio.h>

#include <fcntl.h>

int main()

{

    int fd1 = open("foo.txt", O_RDONLY);

    if (fd1 == -1) {

        perror("open");

        exit(1);

    }

    printf("fd = % d\n", fd1);



    if (close(fd1) == -1) {

        perror("close");

        exit(1);

    }

    printf("Bye.\n");

}
```

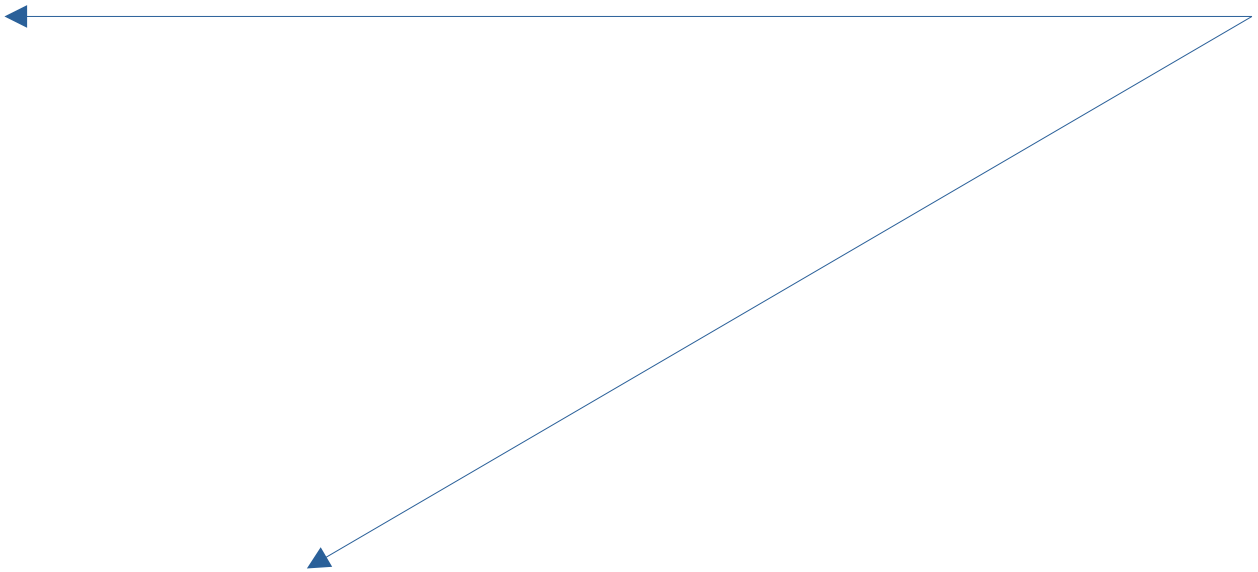
- Le système utilise le premier descripteur libre : 3
- Le système ferme les fichiers ouverts à la fin du processus (sauf si on utilise _exit) mais la bonne pratique est de fermer les fichiers quand on ne les utilise plus :
  - le nombre d'entrées est limité (mais grand. Ex : 1073741816)
  - c'est une bonne pratique générale : on libère toujours les ressources qu'on a réquisitionné dès que possible

```
fd = 3
```

```c
#include<stdio.h>
#include <fcntl.h>
main()
{
    int sz;
    char *ch = "hello geeks\n" ;

    int fd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("Open");
        exit(1);
    }

    sz = write(fd, ch, strlen(ch));

    printf("On vient d'écrire %d caractères\n", sz);

    close(fd);
}
```

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  int fd, retval;
  char buffer;

  if (argc < 2) {
    printf("Usage : %s pathname\n", argv[0]);
    exit(1);
  }

  if ((fd = open(argv[1], O_RDONLY)) == -1) {
    perror("Problème d'ouverture du fichier");
    exit(1);
  }

  while ((retval = read(fd, &buffer, 1)) > 0)
    printf("%c", buffer);

  if (retval < 0) {
    perror("Erreur de lecture");
    exit(1);
  }

  close(fd);
}
```

Ne pas oublier d'allouer la taille nécessaire pour le buffer :
Ici on utilise l'adresse d'une variable de type char car on lit octet par octet mais ce n'est pas le cas général



- Dupliquer un descripteur de fichier : **dup et dup2**

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Ancien descripteur à dupliquer

nouveau descripteur

dup et dup2 renvoient le nouveau descripteur, ou -1 s'ils échouent et errno est mis à jour

- **dup** et **dup2** créent une copie du descripteur de fichier oldfd.
- Après un appel réussi à **dup** ou **dup2**, l'ancien et le nouveau descripteurs peuvent être utilisés de manière interchangeable. Par exemple si le pointeur de position est modifié en utilisant **lseek** sur l'un des descripteurs, la position est également changée pour l'autre.
- **dup** utilise le plus petit numéro inutilisé pour le nouveau descripteur.

- **dup2** transforme newfd en une copie de oldfd, et ferme auparavant newfd si nécessaire.

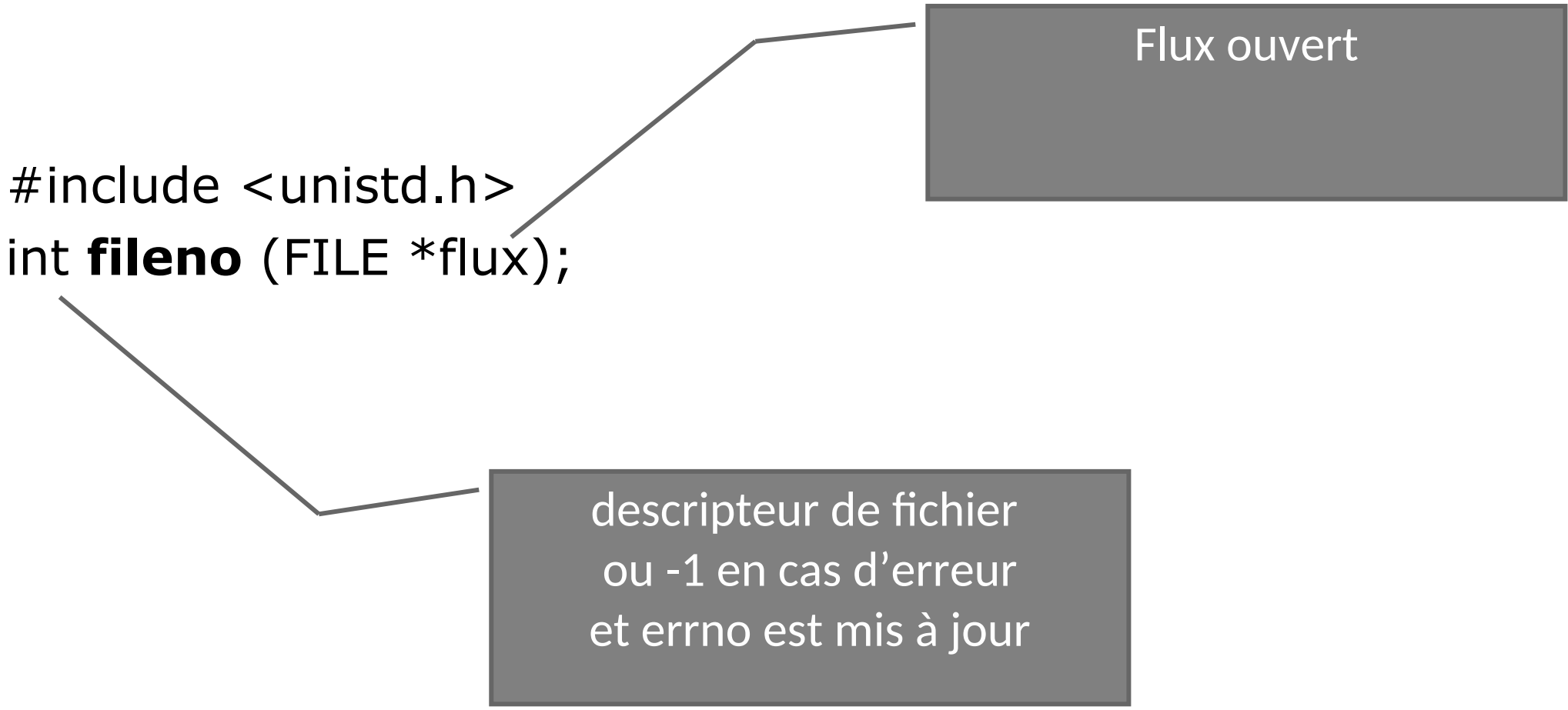
- Association flux / descripteur de fichier : **fdopen**

```
#include <unistd.h>
FILE *fdopen (int desc, const char *mode);
```

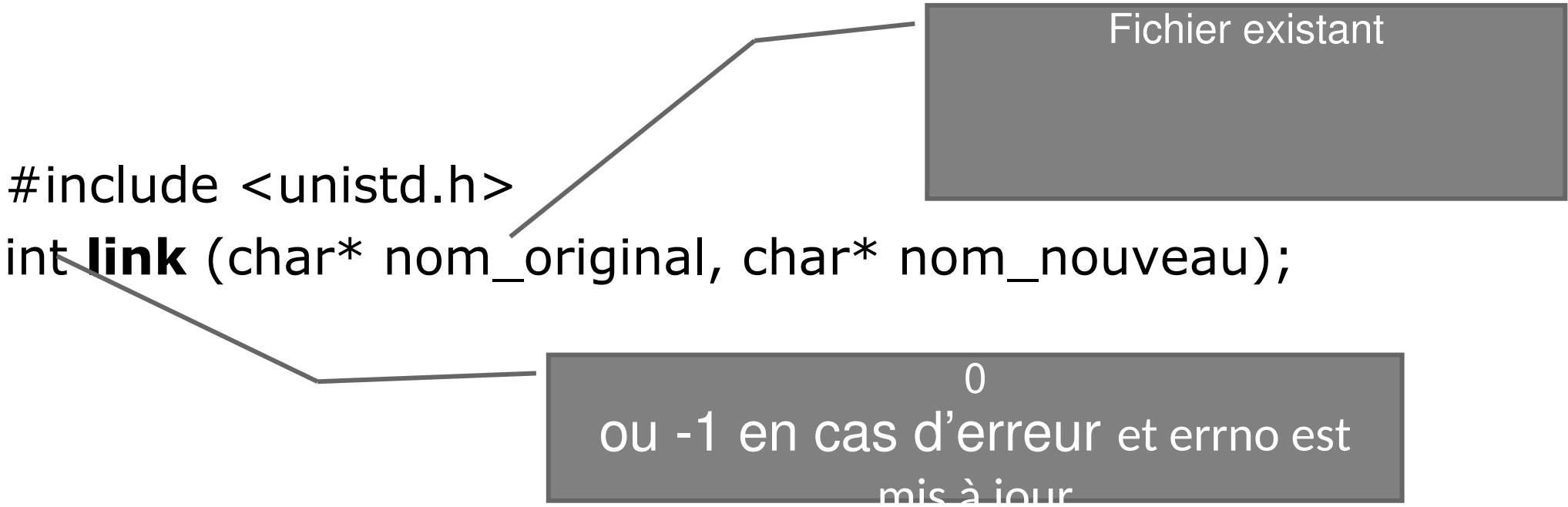
("r", "r+", "w", "w+", "a", ou "a+")
doit être compatible avec celui du descripteur de fichier

descripteur de fichier ouvert

- **Association descripteur de fichier/flux : fileno**



Liens matériels (hard links) entre fichiers



« nouveau » correspond à un nom différent mais au même **inode** que « original » : c'est le même contenu
D'où la notion de **compteur de lien** (incrémenté à chaque nouveau lien)

```
En shell :
$ ln nom_original nom_nouveau
```

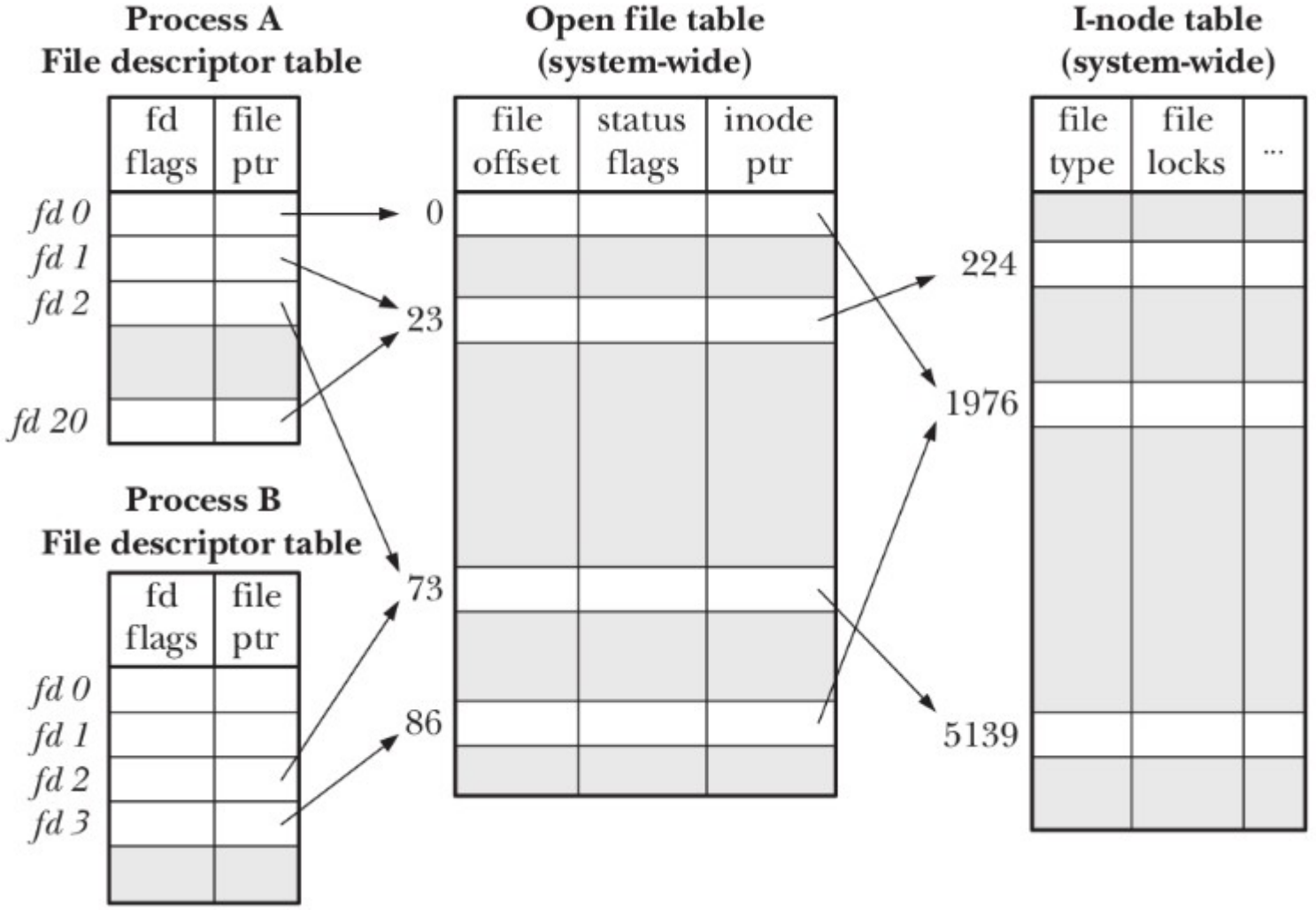
inodes (inoeuds, nœuds d'index)

- Un inode est un entier (unique) qui référence un fichier dans le système de fichier.
- Permet de localiser le fichier dans le système de fichiers.

```
$ ln cours1_2022_2023.odp lien.odp
$ ls -ail
6816330 .rw-r--r-- 2.6M lmascari  9 Jan 16:33  cours1_2022_2023.odp
6816536 .rw-r--r-- 990k lmascari  9 Jan 16:33   cours1_2022_2023.pdf
6816416 .rw-r--r-- 1.0M lmascari 11 Jan 16:35 cours2_2022_2023.odp
6816330 .rw-r--r-- 2.6M lmascari  9 Jan 16:33   lien.odp
```

- Les descripteurs de fichier (propres à chaque processus) sont associés à des entrées dans une table des fichiers ouverts (globale au système), ces entrées sont associées aux inodes.

Relations table des descripteurs/table des fichiers ouverts/inodes



D'après Michael Kerrisk

Table des fichiers ouverts

- Table globale / système, une entrée pour chaque fichier ouvert :
  - Décalage par rapport au début du fichier (offset)
  - Mode d'accès au fichier (RW ...,, cf open())
  - Indicateurs d'état du fichier (cf open())
  - Référence à l’ inode du fichier

```
#include <unistd.h>
int unlink (char* nom_fichier);
```

0
ou -1 en cas d'erreur et errno est mis à jour

Le **compteur de lien** est décrémenté et le fichier détruit quand il est à 0.

```
En shell :
$ rm nom_fichier
```

Liens symboliques (soft links) entre fichiers

Fichier existant

```
#include <unistd.h>
int symlink (char* nom_original, char* nom_nouveau);
```

0
ou -1 en cas d'erreur et errno est mis à jour

Ici le nouveau nom contient seulement le chemin vers l'original :

- On peut effacer le nouveau sans affecter l'original.
- Si l'original est effacé, le nouveau existe toujours mais indique un fichier qui n'existe plus...

Plus souple (fonctionne même entre partitions distantes)

En shell :

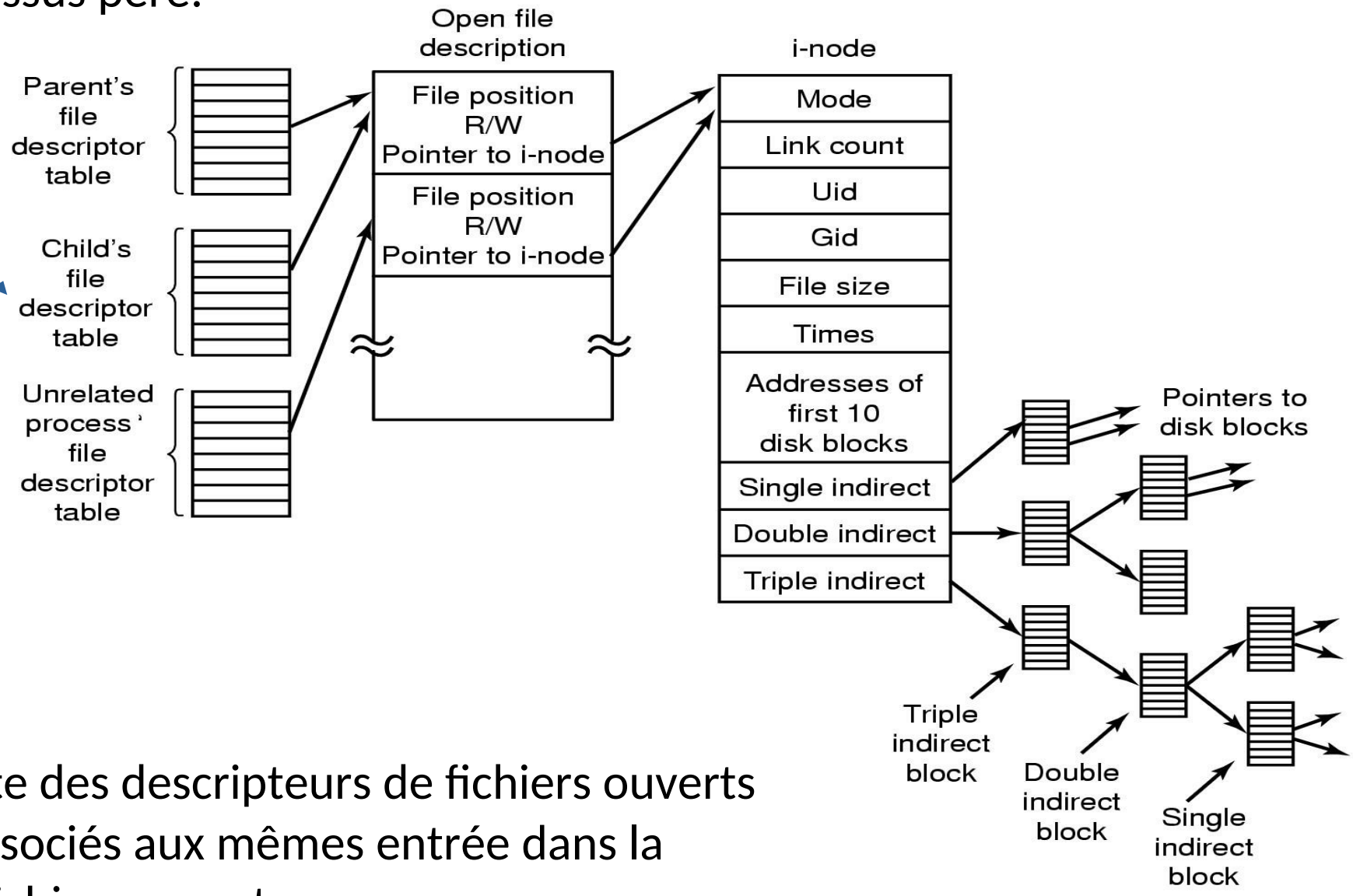
```
$ ln -s nom_original nom_nouveau
```

```
$ ln -s cours1_2022_2023.odp lien_symb.odp
$ ls -il
6816330 .rw-r--r-- 2.6M lmascari  9 Jan 16:33  cours1_2022_2023.odp
6816536 .rw-r--r-- 990k lmascari  9 Jan 16:33   cours1_2022_2023.pdf
6816414 .rw-r--r-- 1.1M lmascari 11 Jan 16:52  cours2_2022_2023.odp
6816330 .rw-r--r-- 2.6M lmascari  9 Jan 16:33   lien.odp
6816510 lrwxrwxrwx   20 lmascari 11 Jan 16:56 lien_symb.odp ->
cours1_2022_2023.odp
```

Dans ce cas les inodes sont différents

Partage de fichiers entre processus père et fils

Le fork duplique la table des descripteurs de fichiers du processus père.



Le fils hérite des descripteurs de fichiers ouverts qui sont associés aux mêmes entrée dans la table des fichiers ouverts

Exemple: partage de fichiers père/fils

```
int outfd = open("fileout",O_RDWR+O_CREAT);
int infd = open("filein", O_RDWR);
FILE* outfile = fdopen(outfd,"w");

fprintf(outfile,"123");
write(outfd,"456",3);
read(infd, buffer, 3);

printf("Before fork read %s\n", buffer);
int p = fork();

if (p > 0) {
    write(outfd,"p7p8",4);
    fprintf(outfile,"p9\n");
    read(infd, buffer, 3);
    printf("Parent after fork read %s\n",
buffer);

} else {
    write(outfd,"c7c8",4);
    fprintf(outfile,"c9\n");
    read(infd, buffer, 3);
    printf("Child after fork read %s\n",
buffer);
}
```

filein :

abcdefghi

Exemple: partage de fichiers

```
int outfd = open("fileout",O_RDWR+O_CREAT);
int infd = open("filein", O_RDWR);
FILE* outfile = fdopen(outfd,"w");

fprintf(outfile,"123");
write(outfd,"456",3);
read(infd, buffer, 3);

printf("Before fork read %s\n", buffer);
int p = fork();

if (p > 0) {
    write(outfd,"p7p8",4);
    fprintf(outfile,"p9\n");
    read(infd, buffer, 3);
    printf("Parent after fork read %s\n",
buffer);

} else {
    write(outfd,"c7c8",4);
    fprintf(outfile,"c9\n");
    read(infd, buffer, 3);
    printf("Child after fork read %s\n",
buffer);
}
```

filein :
abcdefghi

```
$ ./fork_file
Before fork read abc
Parent after fork read def
Child after fork read ghi
$ cat fileout
456p7p8123p9
c7c8123c9
```