

# Fichiers et entrées-sorties

---

**Vania Marangozova**

**Université Grenoble Alpes**

**2023-2024**

# Fichiers

---

## ► Définitions

- Fichier : ensemble d'informations regroupées en vue de leur **conservation** et de leur utilisation dans un système informatique
  - La plus petite unité de stockage persistant
- **Système de gestion de fichiers (SGF)** : partie du système d'exploitation qui conserve les fichiers et permet d'y accéder

## ► Fonctions d'un SGF

- Conservation permanente des fichiers
  - permanente = indépendamment de l'exécution des programmes et de l'intégrité de la mémoire principale)
  - conservation en mémoire secondaire (disque)
- Organisation logique et désignation des fichiers
- Partage et protection des fichiers
- Réalisation des fonctions d'accès aux fichiers

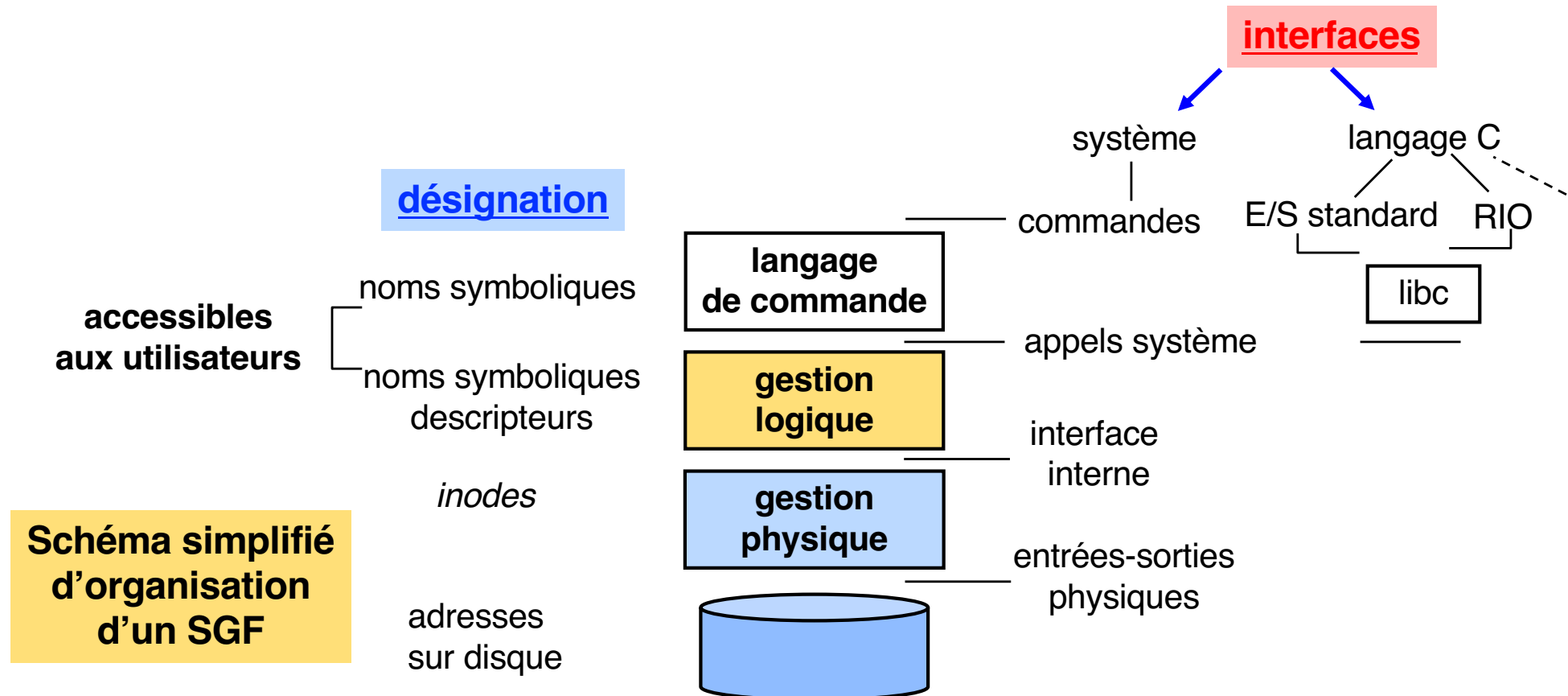
# Plan d'étude

---

- ▶ **Illustré par le SGF d'Unix**
  - ▶ Désignation
  - ▶ Fonctions d'accès aux fichiers
  - ▶ Entrées-sorties, flots et tubes
  - ▶ Protection
  - ▶ Notions sur la réalisation

# Place du SGF dans un système d'exploitation

- ▶ **Les fichiers jouent un rôle central dans un système d'exploitation**
  - ▶ Support des programmes exécutables
  - ▶ Support des données
  - ▶ Communication entre processus et entre utilisateurs
- ▶ **Il existe un lien étroit entre fichiers et entrées-sorties**



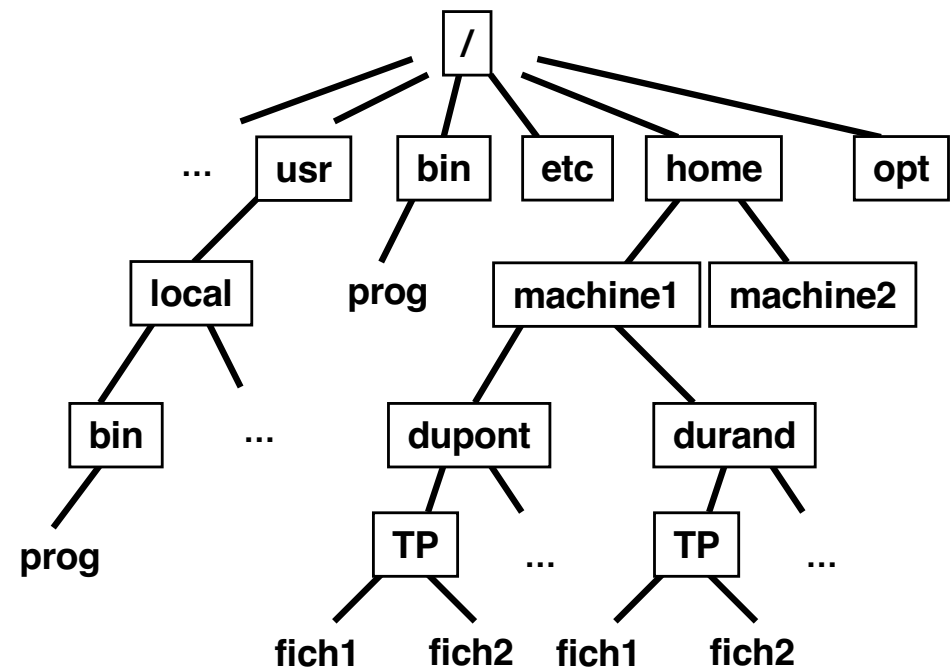
# Désignation des fichiers (1)

- ▶ **Principe de la désignation symbolique : organisation hiérarchique**
  - ▶ Les noms forment une arborescence
  - ▶ Nœuds intermédiaires = catalogues/répertoires/dossiers (*directory*)
    - les répertoires sont aussi des fichiers
  - ▶ Nœuds terminaux = fichiers simples
  - ▶ Nom (universel ou **absolu**) d'un fichier = le chemin d'accès depuis la racine (en anglais : *path*)

## Exemples de noms universels :

```

/
/bin
/usr/local/bin/prog
/home/machine1/dupont/TP/fich1
/home/machine1/durand/TP/fich1
  
```



# Désignation des fichiers (2)

## ► Divers raccourcis simplifient la désignation

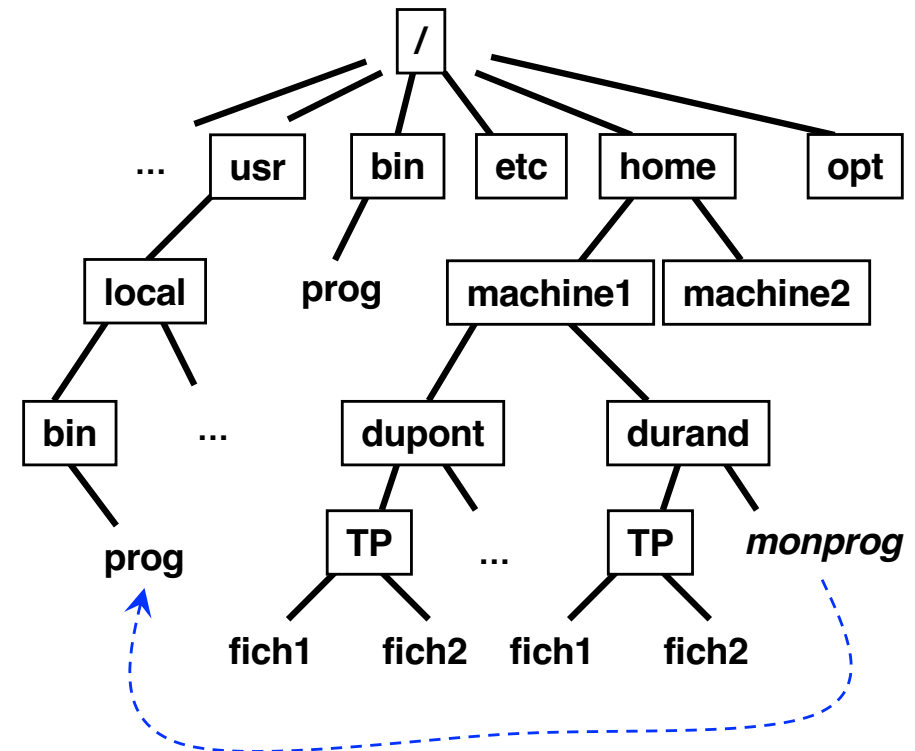
- **noms relatifs** au répertoire courant

Si répertoire courant = `/home/machine1/dupont/`  
 alors on peut utiliser les noms relatifs  
`TP/fich1`, `TP/fich2`

- désignation du **père**

Si répertoire courant = `/home/machine1/dupont/`  
 alors on peut utiliser  
`../durand/TP/fich1`

- **liens symboliques**



Si répertoire courant = `/home/machine1/durand/`

- création du lien : `ln -s /usr/local/bin/prog monprog`
- dans le répertoire courant, le nom `monprog` désigne maintenant le fichier `/usr/local/bin/prog`
- un lien n'est qu'un raccourci : si le fichier cible est supprimé, le lien devient invalide

# Désignation des fichiers (3)

## ► Répertoire courant

- par défaut, tout usager a un répertoire courant de base
  - *home directory*
  - par exemple `/home/machine/dupont`
  - un raccourci est `~dupont`
- on peut changer de répertoire courant au moyen de la commande
 

`cd <nom du répertoire destination>`

  - `cd` sans paramètres ramène au répertoire de base
- le nom `.` désigne le répertoire courant
- **pwd** : connaître le nom absolu du répertoire courant
- **ls** : connaître le contenu du répertoire courant par (`ls -l` est plus complet)

```
<unix> ls -l
total 16
lrwxrwxrwx   1 krakowia sardes   19 Jul  9  2004 isr-l3-td.cls -> ../../isr-l3-td.cls
-rw-r--r--   1 krakowia sardes   94 Jul  9  2004 Makefile
drwxr-xr-x   2 krakowia sardes 4096 Dec 20 16:20 Old/
-rw-r--r--   1 krakowia sardes 4320 Jan 18 10:17 td3.tex
```

# Désignation des fichiers : règles de recherche

---

- ▶ Pour exécuter un programme (fichier exécutable), il suffit d'entrer une commande avec son nom simple.  
**Comment le système trouve-t-il le fichier correspondant ?**
- ▶ Règle de recherche
  - ▶ le système explore dans l'ordre une suite de répertoires
  - ▶ cette suite est enregistrée dans une variable d'environnement **PATH**

```
<unix> echo $PATH  
/usr/local/bin:/bin:/usr/bin:/opt/gnu/arm/bin:/usr/j2se/bin
```

- ▶ La commande `which` indique le nom absolu du fichier qui sera exécuté par défaut

```
<unix> which gcc  
/usr/local/bin/gcc
```



# Règle de recherche : exemple d'application

---

- ▶ **Vous avez créé votre propre exécutable, dans le répertoire courant**

```
<unix> ls -l  
-rwxr-xr-x  1 vania  staff  55872  6 fév 07:32 alarm
```

- ▶ **Pour l'exécuter**

- ▶ soit changer le PATH

```
[unix] alarm  
-bash: alarm: command not found  
[unix] export PATH=.:$PATH  
[unix] alarm  
bip  
bip  
bip  
bip  
^C
```

- ▶ soit plus simplement

```
<unix> ./alarm
```

# Utilisations courantes des fichiers

---

Dans Unix, le contenu d'un fichier est simplement une **suite d'octets**, sans autre structure. L'interprétation de ce contenu dépend de l'utilisation.

## ► Programmes exécutables

- Commandes du système ou programmes créés par un usager

### ► Exemple

```
gcc -o prog prog.c.    produit le programme exécutable dans un fichier prog
./prog                exécute le programme prog
```

## ► Fichiers de données

- Documents, images, programmes sources, etc.

### ► **Convention** : mettre au nom **un suffixe** indiquant la nature du contenu

- .c (programme C), .o (binaire translatable), .h (inclusions), .gif (un format d'images), .ps (PostScript), .pdf (Portable Document Format), etc.

## ► Fichiers temporaires servant pour la communication

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

# Utilisation des fichiers dans le langage de commande

---

## ► Créer un fichier

- Le plus souvent, les fichiers sont créés par les applications, non directement dans le langage de commande. Exemple : éditeur de texte, compilateur, etc
- On peut néanmoins créer explicitement un fichier
  - `touch toto`

## ► Créer un répertoire

- `mkdir <nom du répertoire>` le répertoire est initialement vide

## ► Détruire un fichier

- `rm <nom du fichier>`
- `rm -i` va demander une confirmation

## ► Détruire un répertoire

- `rmdir <nom du répertoire>` le répertoire doit être vide

## ► Conventions pour les noms de fichiers

- \* 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`

# Interface système pour l'utilisation des fichiers (1)

---

- ▶ Dans l'interface des appels système, un fichier est représenté par un **descripteur**. Les descripteurs sont numérotés par des (petits) entiers.

- ▶ Pour utiliser un fichier, il faut l'ouvrir pour allouer un descripteur

```
fd = open ("/home/machine/toto/fich", O_RDONLY, 0)
```

- ▶ Fichier ouvert en lecture seule (on ne peut pas y écrire)
- ▶ le numéro de descripteur alloué par le système est fd (renvoie -1 si erreur).
- ▶ Les autres modes d'ouverture possibles sont O\_RDWR et O\_WRONLY.
- ▶ Le fichier pourra être créé s'il n'existe pas

```
fd = open ("/path/to/file",  
          O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
```

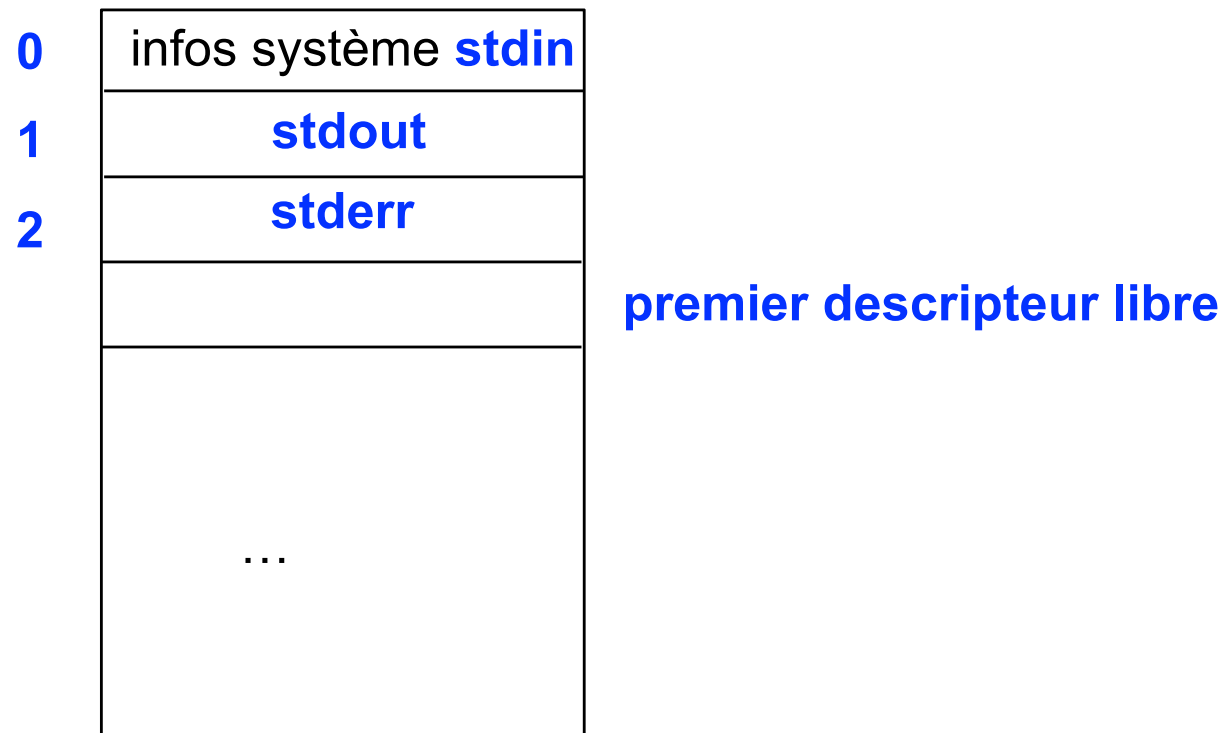
- ▶ Les **bits de permission** (S\_...) définissent quel utilisateur a quels droits d'utilisation du fichier

- voir p.ex. [https://www.gnu.org/software/libc/manual/html\\_node/Permission-Bits.html](https://www.gnu.org/software/libc/manual/html_node/Permission-Bits.html)

- ▶ Quand on a fini d'utiliser un fichier, il faut le fermer `close (fd)`

# Kesako les descripteurs de fichier?

- ▶ Pour manipuler un fichier, le système gère une structure de données dédiés
  - ▶ inode dans Unix
  - ▶ contient les informations sur le fichier, en particulier où sont les données
- ▶ Cette structure ne doit pas être montrée à l'utilisateur (**sécurité**)
- ▶ Le système gère une table de fichiers ouverts



# Interface système pour l'utilisation des fichiers (2)

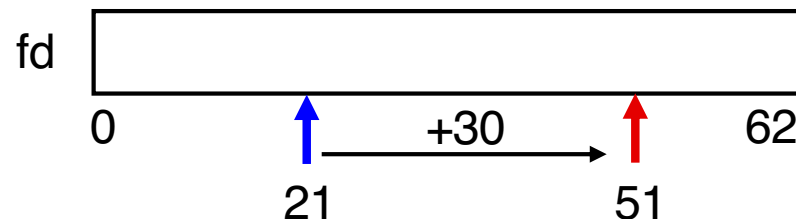
L'ouverture crée un **pointeur courant** (position dans le fichier), initialisé à 0.  
Ce pointeur (invisible directement) est déplacé

- indirectement, par les opérations de lecture (`read`) et d'écriture (`write`).  
(cf détails plus loin)
- directement, par l'opération `lseek` (ci-dessous)

`lseek()` déplace le pointeur courant. Exemples :

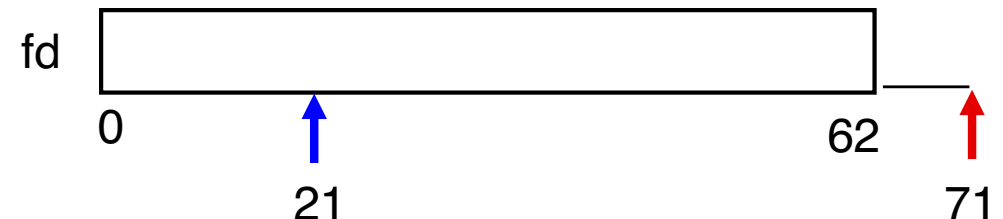
**dépl. relatif**

`lseek(fd, 30, SEEK_CUR)`  
+30 octets depuis position courante



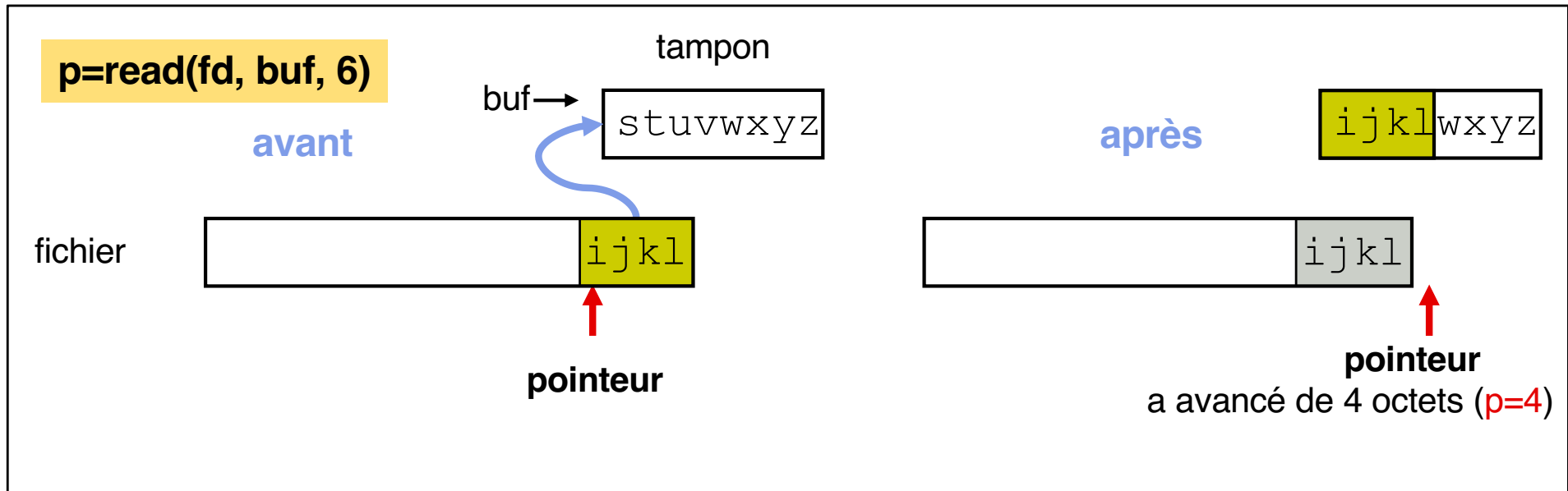
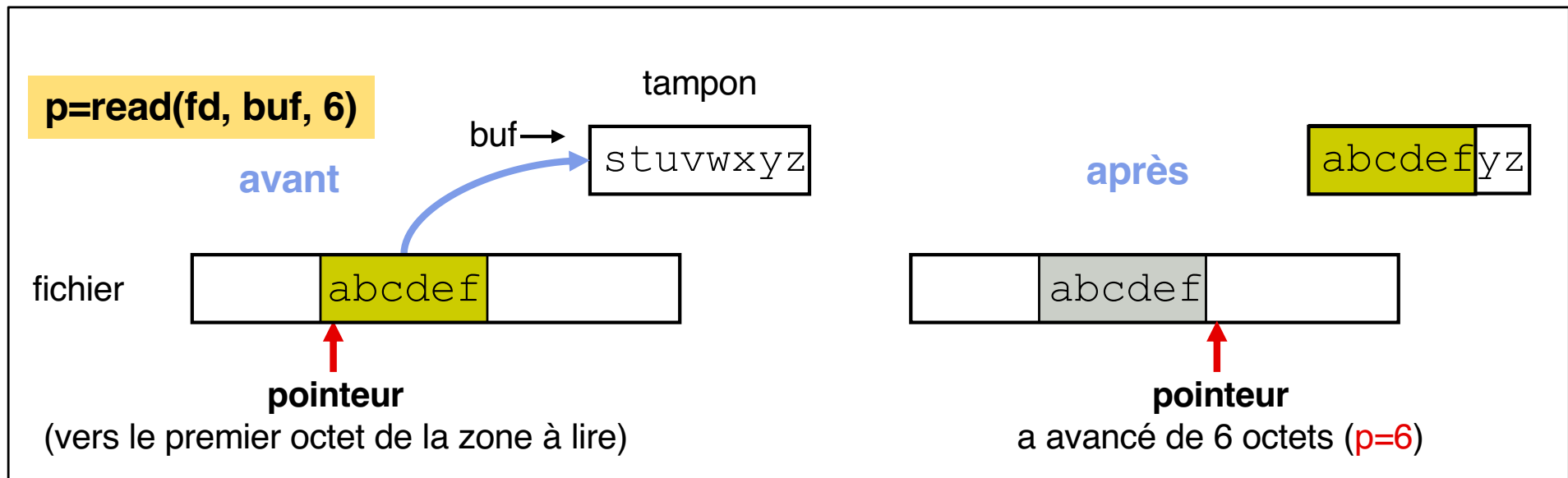
**dépl. absolu**

`lseek(fd, 71, SEEK_SET)`  
place le pointeur à la position 71

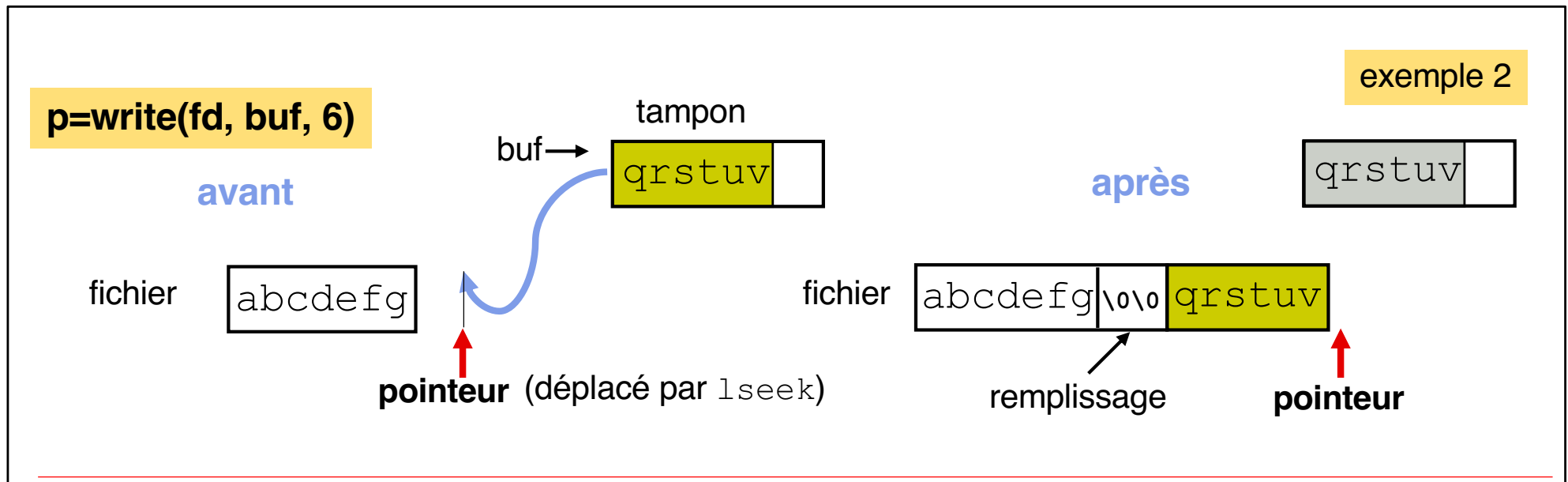
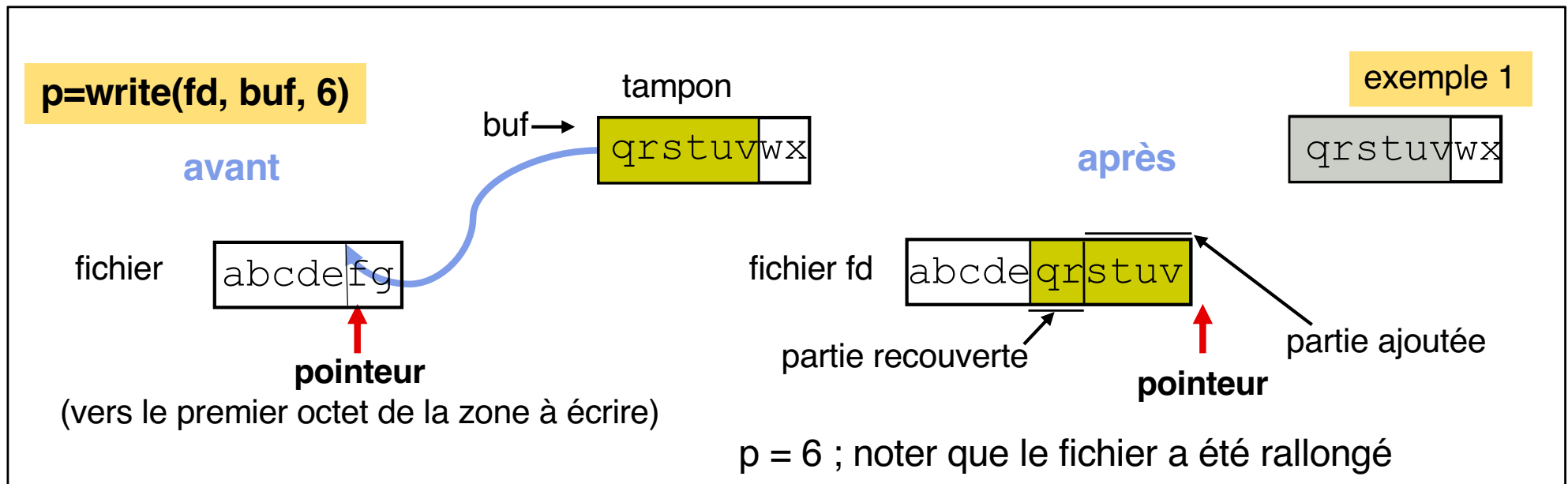


Le pointeur peut être placé au-delà de la fin du fichier.

# Interface système pour l'utilisation des fichier : `read()`



# Interface système pour l'utilisation des fichiers : `write()`





# Interfaces des fichiers (1)

---

Les primitives fournies par le noyau (`open`, `close`, `lseek`, `read`, `write`) sont celles de plus bas niveau. Leur utilisation est souvent **délicate** (gestion des erreurs, lectures tronquées, etc.)

Il est en général préférable de les utiliser à travers des bibliothèques qui facilitent leur usage. Deux bibliothèques sont recommandées :

## Interfaces des fichiers (2)

---

Les primitives fournies par le noyau (`open`, `close`, `lseek`, `read`, `write`) sont celles de plus bas niveau. Leur utilisation est souvent **délicate** (gestion des erreurs, lectures tronquées, etc.)

Il est en général préférable de les utiliser à travers des bibliothèques qui facilitent leur usage. Deux bibliothèques sont recommandées :

La **bibliothèque dite “standard”**, ensemble de fonctions d'accès de plus haut niveau, avec formats, inclus dans la bibliothèque C : `fopen`, `fread`, `fwrite`, `fscanf`, `fprintf`, `fflush`, `fseek`, `fclose` (et fonctions analogues pour les chaînes : `sprintf`, `sscanf`). Voir man.

## Interfaces des fichiers (3)

---

Les primitives fournies par le noyau (`open`, `close`, `lseek`, `read`, `write`) sont celles de plus bas niveau. Leur utilisation est souvent **délicate** (gestion des erreurs, lectures tronquées, etc.)

Il est en général préférable de les utiliser à travers des bibliothèques qui facilitent leur usage. Deux bibliothèques sont recommandées :

Une **bibliothèque appelée `RIO`** (*Robust Input-Output*), développée par W. R. Stevens et améliorée par R. Bryant et D. O'Hallaron. Voir documentation dans le Moodle. Recommandée pour les tubes et les *sockets* (cf plus loin).

# Interfaces des fichiers (4)

---

Les primitives fournies par le noyau (`open`, `close`, `lseek`, `read`, `write`) sont celles de plus bas niveau. Leur utilisation est souvent **délicate** (gestion des erreurs, lectures tronquées, etc.)

Il est en général préférable de les utiliser à travers des bibliothèques qui facilitent leur usage. Deux bibliothèques sont recommandées :

La **bibliothèque dite “standard”**, ensemble de fonctions d'accès de plus haut niveau, avec formats, inclus dans la bibliothèque C : `fopen`, `fread`, `fwrite`, `fscanf`, `fprintf`, `fflush`, `fseek`, `fclose` (et fonctions analogues pour les chaînes : `sprintf`, `sscanf`). Voir `man`.

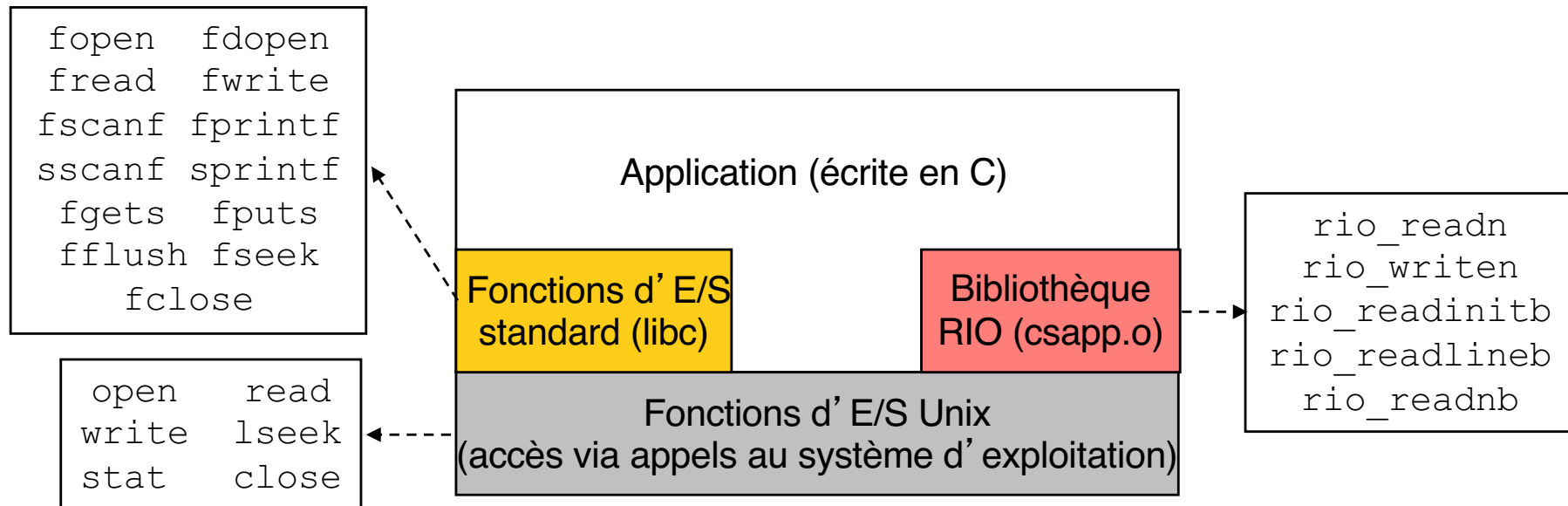
Une **bibliothèque appelée RIO** (*Robust Input-Output*), développée par W. R. Stevens et améliorée par R. Bryant et D. O'Hallaron. Voir documentation dans le Moodle. Recommandée pour les tubes et les *sockets* (cf plus loin).

W. R. Stevens, *Unix Network Programming*, vol. 1, Prentice Hall, 1998

R. E. Bryant, D. O'Hallaron : *Computer Systems, A Programmer's Perspective*, Prentice Hall, 2003

# Interfaces des fichiers (5)

## ► Récapitulatif



Source : R. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Prentice-Hall, 2003

# Protection des fichiers (1)

---

## ► Définition (générale) de la sécurité

- confidentialité : informations accessibles aux seuls usagers autorisés
- intégrité : pas de modifications non désirées
- contrôle d'accès : seuls certains usagers sont autorisé à faire certaines opérations
- 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 (mécanismes de protection) pour assurer que ces règles sont respectées

## ► Sécurité des fichiers (dans Unix)

- On définit
  - des types d'opérations sur les fichiers : lire, écrire, exécuter (contraintes de confidentialité, intégrité, contrôle d'accès)
  - des classes d'usagers
    - usager propriétaire du fichier
    - groupe propriétaire
    - tous les autres

# Protection des fichiers (2)

## ► Fichiers ordinaires

<b>r w x</b>	<b>r w x</b>	<b>r w x</b>	<b>   </b>
propriétaire (u)	groupe (g)	autres (o)	setuid setgid

exemple (fichier fich) : rwx r-- r-- : tout accès pour le propriétaire, lecture seule pour tous les autres

chmod go+w fich : donne le droit w au groupe et aux autres  
chmod o-w fich : retire le droit w aux autres

- r = lecture, w = écriture, x = exécution

## ► répertoires

- même chose, mais le droit x signifie “recherche dans le répertoire”

## ► Un mécanisme de délégation

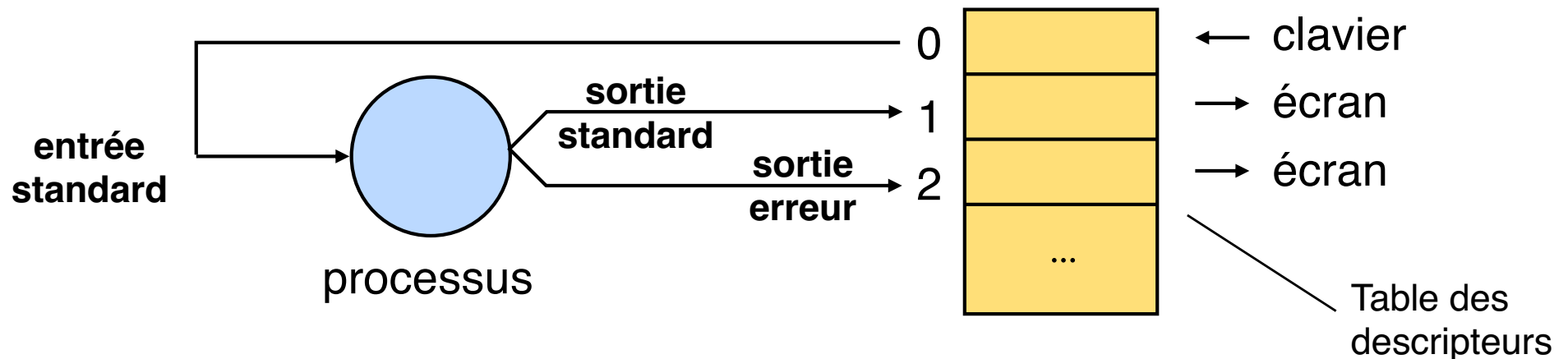
- Le problème : partager un programme dont l'exécution nécessite des droits d'accès que n'ont pas les usagers potentiels
- Solution (setuid ou setgid) : pour l'exécution de ce programme (et uniquement pour cette exécution), un usager quelconque reçoit temporairement les droits de l'utilisateur ou du groupe propriétaire si le bit correspondant est à 1

## ► Règles d'éthique

- protéger vos informations confidentielles
- ne pas tenter de contourner les mécanismes de protection
- les règles de bon usage s'appliquent indépendamment de la protection (ce n'est pas parce qu'un fichier n'est pas protégé qu'il est licite de le lire)

# Fichiers et flots d'entrée-sortie

- ▶ Il y a un lien étroit entre **fichiers** et **entrées-sorties**
- ▶ Les organes d'entrée-sortie sont représentés par des fichiers particuliers (sous Unix, dans le répertoire /dev)
- ▶ Tout processus utilise des **flots** d'entrée-sortie qui peuvent être dirigés soit vers un fichier, soit vers un organe d'entrée-sortie : entrée standard, sortie standard, et sortie erreur
- ▶ Par convention, ces flots sont associés aux descripteurs 0, 1 et 2



- ▶ Les flots d'entrée-sortie peuvent être réorientés vers des fichiers



# Manipuler les flots d'entrée-sortie (commandes)

- Dans le langage de commande, on réoriente les flots standard au moyen de `<` 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)
cat /dev/null > fich crée le fichier vide fich s'il n'existe pas, sinon le rend vide
```

- Les **tubes** (pipes) permettent de faire communiquer des processus.
- Un tube est un fichier anonyme qui sert de tampon entre deux processus fonctionnant en producteur-consommateur.

```
cat *.c | grep var
```

a) crée un tube et deux processus : p1 qui exécute `cat *.c`, p2 qui exécute `grep var`

b) connecte la sortie de p1 à l'entrée du tube et l'entrée de p2 à la sortie du tube



**Question :** que fait la commande suivante ?

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

# Manipuler les flots d'entrée-sortie (primitives)

Les tubes et les flots peuvent être manipulés au niveau des appels système.

Créer un tube : la primitive `pipe()` crée un tube, dont l'entrée et la sortie sont associées à des descripteurs choisis par le système

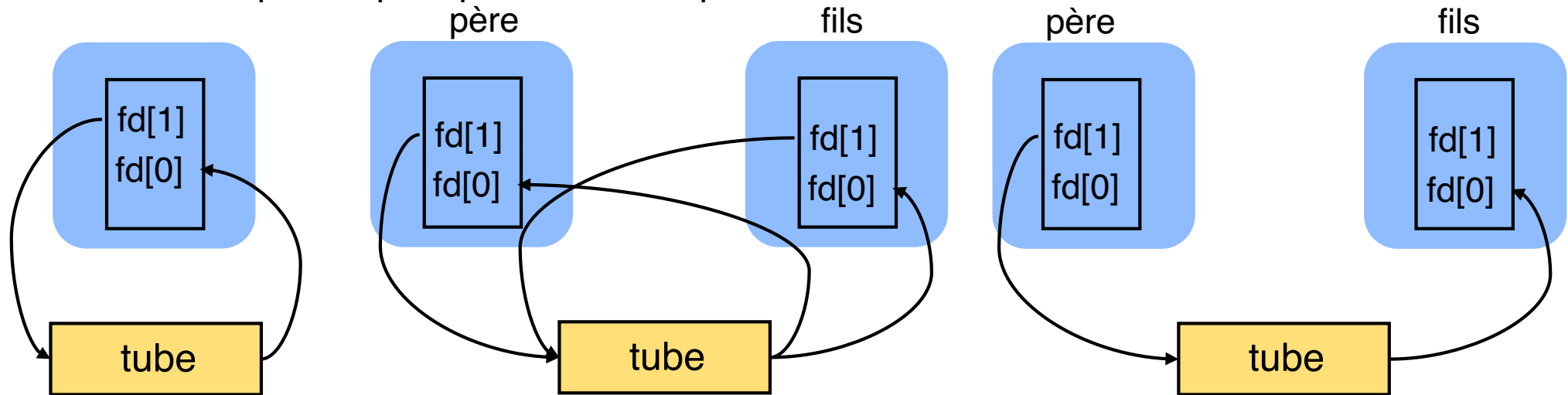
```
int fd[2]; pipe(fd);
```

Si la primitive réussit, elle crée un tube, renvoie 0, et met à jour le tableau `fd` :

**`fd[0]` = desc. sortie du tube, `fd[1]` = desc. entrée du tube.**

Si elle échoue, elle renvoie `-1`

Par exemple, un père peut communiquer avec un fils à travers un tube.



après `pipe(fd)`

après `fork()` (descripteurs copiés)

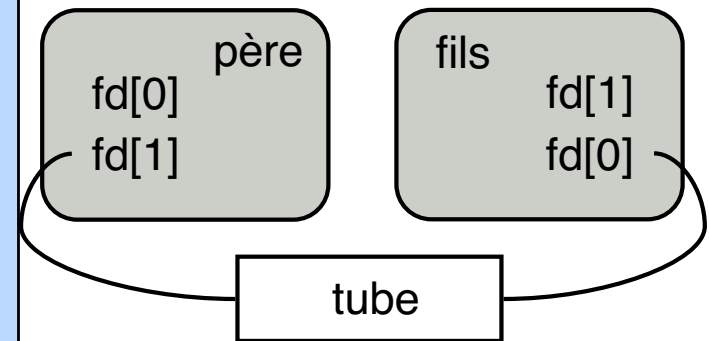
après fermeture descripteurs  
inutiles

# Programmation d'un tube père -> fils

```
#include "csapp.h"
#define BUFSIZE 10
int main(void) {
    char bufin[BUFSIZE] = "empty";
    char bufout[] = "hello";
    int bytesin;  pid_t childpid;
    int fd[2];
    pipe(fd);
    bytesin = strlen(bufin);
    childpid = Fork();
    if (childpid != 0) {          /* père */
        Close(fd[0]); write(fd[1], bufout, strlen(bufout)+1);
    } else {                     /* fils */
        Close(fd[1]); bytesin = read(fd[0], bufin, BUFSIZE);
        printf("[%d]:%d bytes, my bufin is {%s}, my bufout is {%s}\n", getpid(), bytesin, bufin, bufout);
        exit(0);
    }
}
```

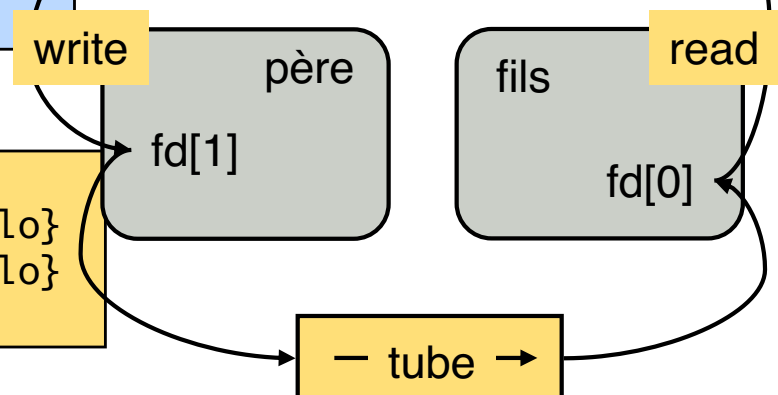
```
<unix>./parentwritepipe
[29196]:5 bytes, my bufin is {empty}, my bufout is {hello}
[29197]:6 bytes, my bufin is {hello}, my bufout is {hello}
<unix>
```

bufin : "empty"    bufin : "empty"  
bufout : "hello"    bufout : "hello"



après `Fork()`

bufin : "empty"    bufin : "hello"  
bufout : "hello"    bufout : "hello"



# Tubes nommés (FIFOs)

---

Un tube ne peut être utilisé qu'entre un processus et ses descendants (ou entre descendants d'un même processus). En effet, ses extrémités ne sont désignées que par des **descripteurs**, qui ne peuvent se transmettre qu'entre père et fils.

Pour faire communiquer deux processus quelconques, on utilise des **tubes nommés**, ou **FIFOs**, tubes possédant un **nom symbolique** dans le répertoire des fichiers. Un tel tube est créé par la primitive `mkfifo`

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(char *nom, mode_t mode) /* renvoie 0 si OK, -1 si erreur */
```

crée un FIFO appelé `nom` avec le mode de protection `mode` (comme pour un fichier, cf plus loin).

Pour pouvoir être utilisé, un FIFO doit avoir été préalablement ouvert par deux processus, l'un en mode écriture, l'autre en mode lecture. Chacun des processus reste bloqué tant que l'autre n'a pas ouvert le FIFO. Détails en TD et TP.

# Copie de descripteurs : dup ()

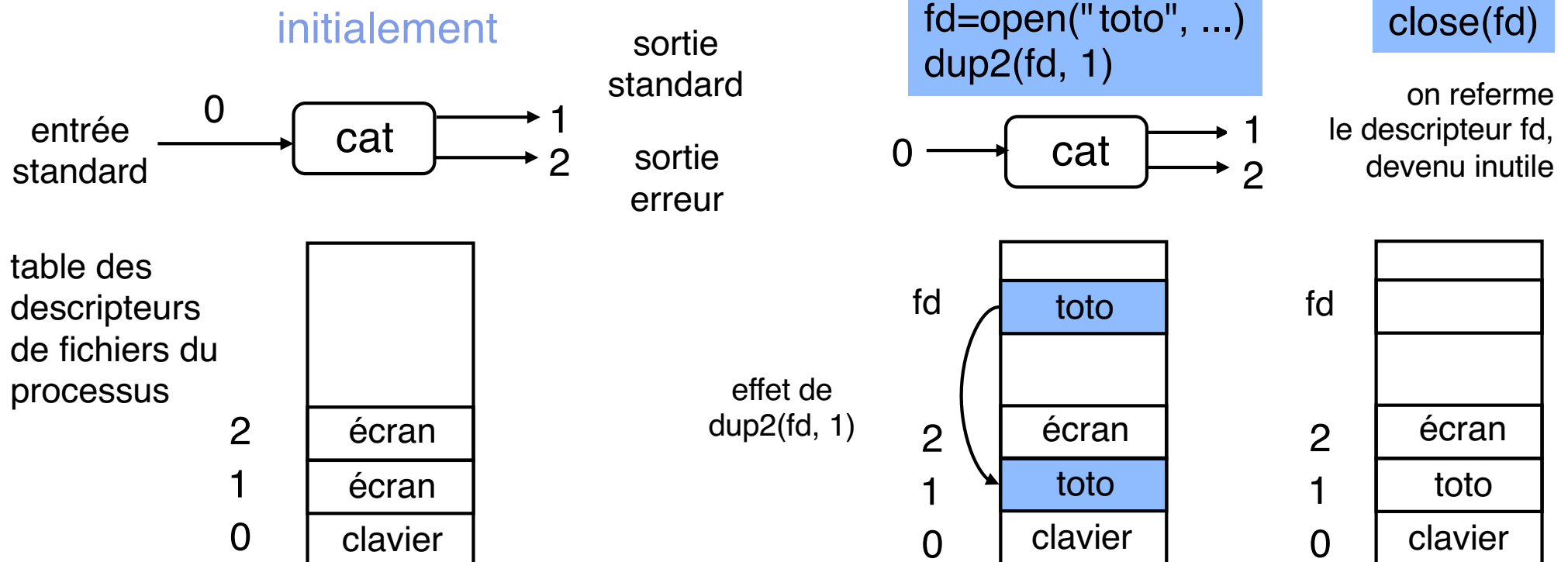
Que fait la commande suivante ?

```
<unix> cat > toto
```

Elle recopie ce qui est lu sur l'entrée standard (clavier) vers le fichier toto (qui est créé s'il n'existe pas). L'entrée doit finir par EOF (control-D)

Que se passe-t-il "derrière" ?

redirection



## Copie de descripteurs : `dup()` (2)

---

La primitive `dup(fd)` recopie le descripteur de numéro `fd` dans le premier descripteur disponible (descripteur disponible de plus petit numéro).

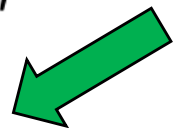
La primitive `dup2(fd, fd1)` recopie le descripteur de numéro `fd` dans le descripteur de numéro `fd1`.

Le rôle de ces primitives est la redirection des flots d'entrée-sortie, cf exemple précédent. Noter aussi qu'un processus fils hérite de la table de descripteurs de son père. Les redirections effectuées se transmettent aux fils.

# Exemple

```
#define BUFSIZE 10
int main(void) {
    char bufin[BUFSIZE] = "empty";
    char bufout[BUFSIZE] = "hello";
    int bytesin, bytesout;
    pid_t childpid;
    int fd[2];

    pipe(fd);
    bytesin = strlen(bufin);
    childpid = Fork();
    if (childpid != 0) {                /* pere */
        Close(fd[0]);
        bytesout = write(fd[1], bufout, strlen(bufout)+1);
        printf("[%d]: write %d bytes, send %s to my child\n",
            getpid(), bytesout, bufout);
    } else {                            /* fils */
        Close(fd[1]); bytesin = read(fd[0], bufin, BUFSIZE);
        printf("[%d]: read %d bytes, my bufin is {%s} \n",
            getpid(), bytesin, bufin);
    }
    exit(0);
}
```



# Communication dans les deux sens

```

..
#define BUFSIZE 10
int main(void) {
    char bufin[BUFSIZE] = "empty";
    char bufout[BUFSIZE] = "hello";
    int bytesin, bytesout;
    pid_t childpid;
    int tube1[2], tube2[2];

    pipe(tube1); pipe(tube2);
    bytesin = strlen(bufin);
    childpid = Fork();
    if (childpid != 0) {                /* père */
        Close(tube1[0]);
        Close(tube2[1]);

        bytesout = write(tube1[1], bufout, strlen(bufout)+1);
        printf("[%d]: J'écris %d bytes, envoie %s à mon fils\n",
                getpid(), bytesout, bufout);

        bytesin = read(tube2[0], bufin, BUFSIZE);
        printf("[%d]: Je lis %d bytes, reçu %s de mon fils\n",
                getpid(), bytesin, bufin);
    }
}

```



# Communication dans les deux sens 2

---

```
    } else {                                     /* fils */
        Close(tube1[1]);
        Close(tube2[0]);

        bytesin = read(tube1[0], bufin, BUFSIZE);
        printf("[%d]: lu %d bytes, my bufin est {%s} \n ",
                getpid(), bytesin, bufin);

        bytesout = write(tube2[1], bufin, strlen(bufin)+1);
        printf("[%d]: J'écris %d bytes, envoie %s au père\n",
                getpid(), bytesout, bufin);

    }
    exit(0);
}
```