



BUT 2 / R3.05

# PROGRAMMATION SYSTÈME

ENTRÉES-SORTIES



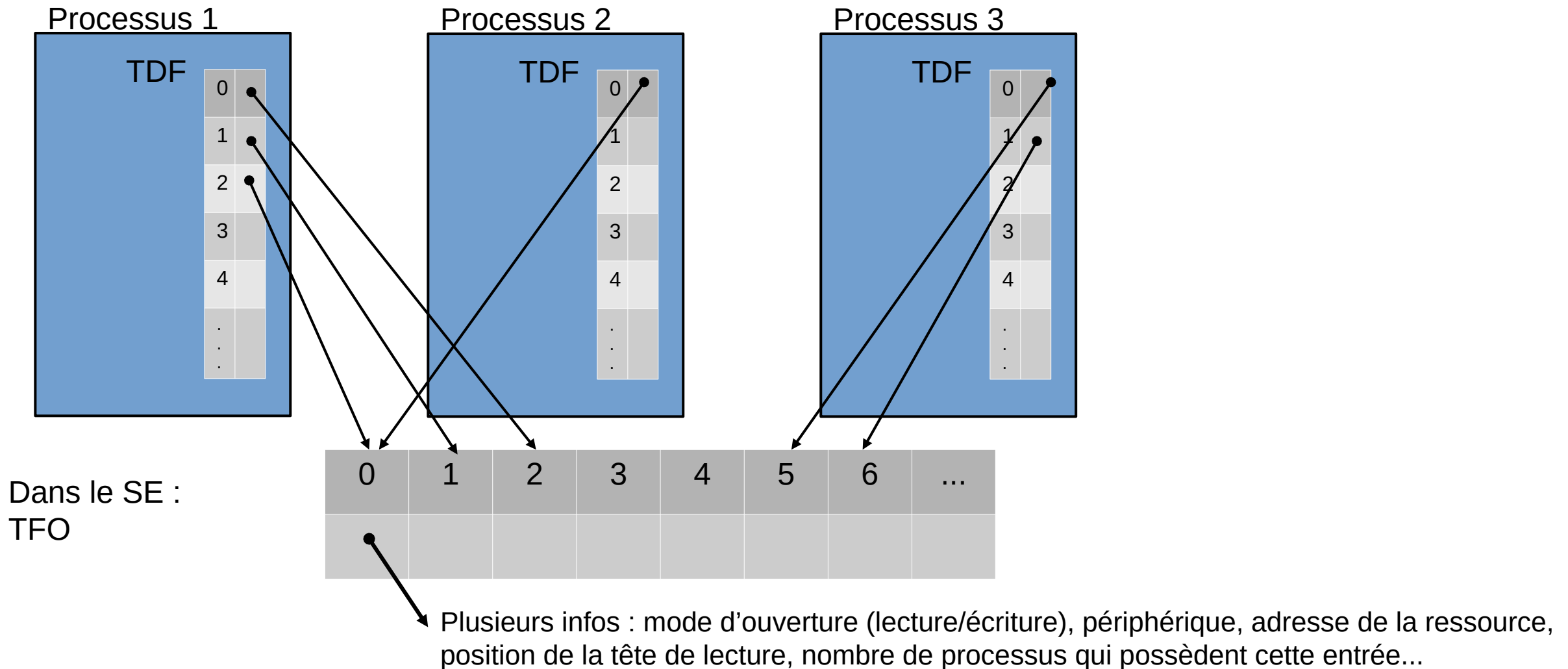
# INTRODUCTION

- **Abstraction** : exposer au programme une abstraction des périphériques
- Les entrées-sorties sous Linux sont gérées via la notion abstraite de **Fichier** : permet de manipuler
  - Des « vrais » fichiers, mais aussi :
  - Périphériques
  - Réseau

# DESCRIPTEURS DE FICHIERS

- Dans les programmes, on manipule des **DESCRIPTEURS DE FICHIERS** (file descriptor)
  - Un entier qui identifie, au sein d'un processus, un fichier
  - Chaque processus contient une **table des descripteurs de fichiers (TDF)**
    - Commence à l'indice 0, et le maximum est généralement 1024 (ou une autre valeur choisie par le SE)
  - Chaque entrée peut être vue comme un **pointeur** vers une entrée de la **table des fichiers ouverts (TFO)**
    - Cette structure est **commune à tous les processus**

# DESCRIPTEURS DE FICHIERS



# POUR OUVRIR UN FICHER

**man 2 open** : 2 signatures

```
int open(const char *pathname, int flags);
```

```
int open(const char *pathname, int flags, mode_t mode);
```

- Premier argument : adresse du fichier à ouvrir
- Second argument : des *drapeaux* qui précisent le mode d'ouverture. On peut en ajouter plusieurs grâce l'opérateur |
  - Il faut absolument en choisir un parmi :
    - O\_RDONLY (lecture seulement)
    - O\_WRONLY (écriture seulement)
    - O\_RDWR (lecture et écriture)
  - On peut en rajouter d'autres (voir page suivante)
- La fonction retourne -1 en cas d'erreur, ou le **descripteur de fichier** correspondant en cas de succès

# POUR OUVRIR UN FICHER

■ Quelques drapeaux à retenir :

- **O\_APPEND** : si on écrit dans le fichier, on se placera à la fin de celui-ci (sinon, on sera au début, et donc on écrasera les éventuelles données)
- **O\_TRUNC** : on efface tout le contenu précédent du fichier
- **O\_CREAT** : si le fichier n'existe pas, on le crée (s'il n'y a pas ce drapeau et qu'il n'existe pas, la fonction retourne -1). Mais alors il faut utiliser la seconde signature

```
int open(const char *pathname, int flags, mode_t mode);
```

et le mode d'ouverture correspond aux droits (unix) du fichier en octal (ex : 0644, 0777...)

rappel : chaque chiffre : droit en écriture (4) – lecture (2) – exécution (1), puis

chiffre de droite : droit des autres utilisateurs

chiffre du milieu : droit des utilisateurs du même groupe

que celui qui a créé

chiffre de gauche : droit de celui qui l'a créé

# GESTION DES DESCRIPTEURS DE FICHIERS

- À chaque open, on retourne le premier descripteur de fichier disponible (qui pointe vers rien)
- Pour fermer un fichier : `int close(int fd);`
- Remarque : à la terminaison du processus, la TDF est supprimée, donc tous les fichiers sont « fermés »
- Par défaut, tout nouveau processus a ses 3 premières entrées dans la TDF déjà attribuées :
  - 0 : ouvert en lecture seulement : entrée standard (saisie clavier)
  - 1 : ouvert en écriture seulement : sortie standard (dans la console qui a créé le processus)
  - 2 : ouvert en écriture seulement : sortie d'erreur (dans la console qui a créé le processus)

Vous ne le voyez pas, mais par exemple :

- printf écrit sur 1
- perror écrit sur 2

# EXEMPLES

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

Ouverture en lecture seulement

```
int fd2 = open("toto.txt", O_RDWR | O_CREAT, 0644) ;
```

Ouverture en lecture/écriture, s'il n'existe pas on le crée avec les droits 644

```
int fd3 = open("toto.txt", O_RDWR | O_TRUNC) ;
```

Ouverture en lecture/écriture, on efface le contenu du fichier

```
int fd4 = open("/dev/input/mice", O_RDONLY) ;
```

Ouverture en lecture du fichier associé à la souris (périphérique)



# LECTURE

man 2 read :

```
ssize_t read(int fd, void *buf, size_t count);
```

- Premier paramètre : un descripteur de fichier
- Second paramètre : un pointeur qui va accueillir les données lues
- Troisième paramètre : le nombre d'octets que l'on souhaite lire
- Retourne :
  - -1 en cas d'erreur (ex : problème de mode d'ouverture, fichier terminé), ou
  - le nombre d'octets effectivement lus ( $\leq$  **count** : on peut en lire moins si le fichier est terminé)
- C'est un appel système **bloquant** : on peut éventuellement attendre que des données arrivent dans le fichier/périphérique/réseau
- En cas de succès, cela fait avancer la **tête de lecture** associée au fichier ouvert

# LECTURE : EXEMPLE

```
int fd = open("toto.txt", O_RDONLY) ;
if (fd < 0) {
    perror("ouverture") ;
    exit(0) ;
}
char buf[10] ;
if (read(fd, buf, 10) < 0) {
    perror("lecture") ;
    exit(0) ;
}
printf("données lues : %s \n", buf) ;
```

toto.txt

Coucou comment allez  
vous ?

À l'exécution, on affichera :

**données lues : Coucou com**

# ÉCRITURE

man 2 write: `ssize_t write(int fd, const void *buf, size_t count);`

- Premier paramètre : un descripteur de fichier
- Second paramètre : un pointeur qui pointe vers les données à écrire
- Troisième paramètre : le nombre d'octets que l'on souhaite écrire
- Retourne :
  - -1 en cas d'erreur (ex : problème de mode d'ouverture, disque plein), ou
  - le nombre d'octets effectivement écrits ( $\leq \text{count}$  : on peut en écrire moins si le disque est plein)
- C'est un appel système **bloquant** : on peut éventuellement attendre que des données soient écrites dans le fichier/périphérique/réseau
- En cas de succès, cela fait avancer la **tête de lecture** associée au fichier ouvert
- Si on écrit alors que la tête de lecture est à la fin du fichier, **le fichier s'agrandit**

# LECTURE : EXEMPLE

```
int fd = open("toto.txt", O_WRONLY) ;
if (fd < 0) {
    perror("ouverture") ;
    exit(0) ;
}
char buf[] = "BONJOUR" ;
if (write(fd, buf, 7) < 0) {
    perror("écriture") ;
    exit(0) ;
}
```

toto.txt

Coucou comment allez  
vous ?

Après l'exécution, le fichier contiendra

toto.txt

BONJOURcomment  
allez vous ?

# TÊTE DE LECTURE

- Dans une entrée de la TFO figure une « tête de lecture » (offset), qui représente l'endroit actuel où l'on se trouve dans le fichier (là où on va lire/écrire)
- Par défaut, après un open, la tête de lecture est placée au début du fichier
- S'il y a le flag `O_APPEND` alors celle-ci est placée à la fin du fichier à chaque write
- À chaque lecture ou écriture, la tête de lecture avance
  - read et write font avancer la tête de **count** octets
  - Si fichier texte : 1 caractère ASCII = 1 octet = un entier entre 0 et 255

# EXEMPLE

```
int fd = open("toto.txt", O_RDWR) ;  
char c1 = 'a' ;  
char c2 ;  
read(fd, &c2, 1) ;  
write(fd, &c1, 1) ;  
read(fd, &c2, 1) ;  
write(fd, &c1, 1) ;
```

Avant l'exécution

toto.txt

0123456789

Après l'exécution :

toto.txt

0a2a456789

# EXERCICE 7 DU TP1

Écrire un programme qui reproduit le comportement de `cat fichier`, c'est à dire :

- 1) On récupère en paramètre du programme (via argv) le chemin d'un fichier
- 2) On ouvre ce fichier en lecture
- 3) On lit un caractère du fichier
- 4) On l'affiche sur la sortie standard (descripteur de fichier 0)
- 5) On recommence 3) et 4) tant qu'il reste des caractères à lire

# EXERCICE RÉVISION

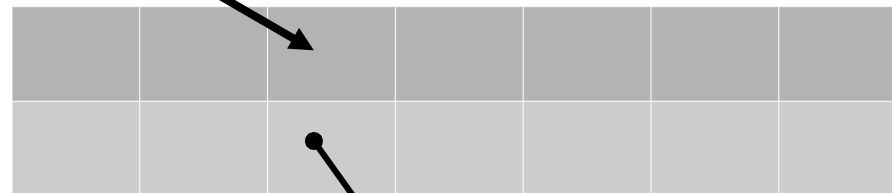
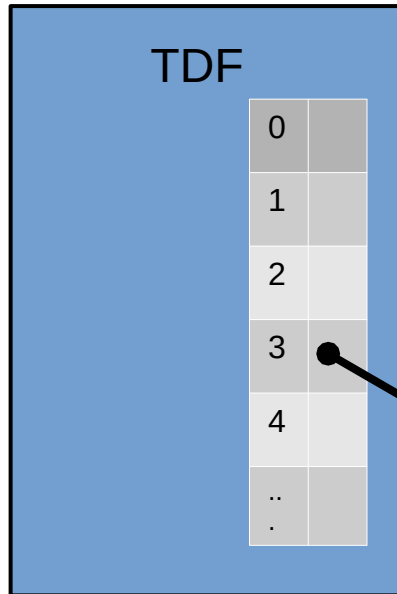
Quel est le contenu du fichier toto.txt après l'exécution de ce programme ?

```
int main() {  
  
    int fd = open("toto.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);  
    char s1[] = "Hello world ! ";  
    write(fd, s1, sizeof(s1)/sizeof(char));  
  
    if (fork() > 0) { //père  
        for (int i = 0 ; i < 20 ; i++) {  
            char s2[] = "père ";  
            write(fd, s2, sizeof(s2)/sizeof(char));  
        }  
  
    } else { //fils  
        for (int i = 0 ; i < 20 ; i++) {  
            char s2[] = "fils ";  
            write(fd, s2, sizeof(s2)/sizeof(char));  
        }  
    }  
  
    return 0;  
}
```



# EXERCICE RÉVISION

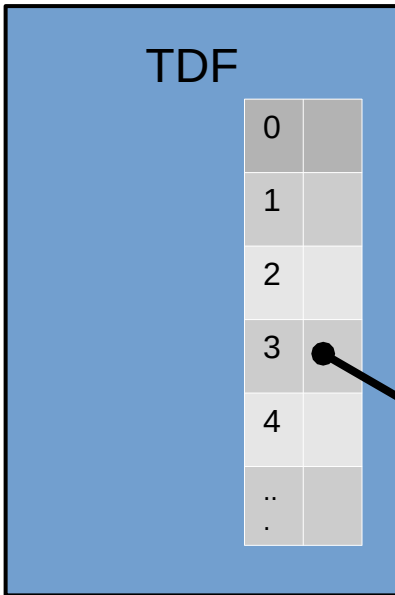
Processus 1



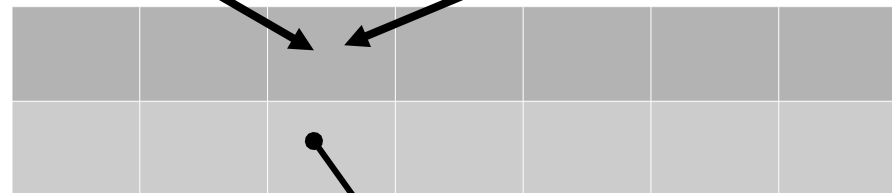
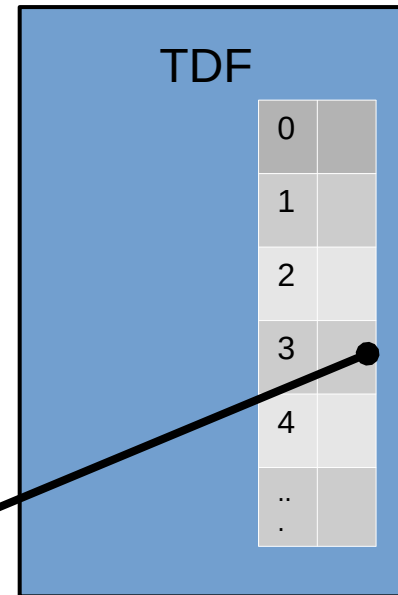
toto.txt

# EXERCICE RÉVISION

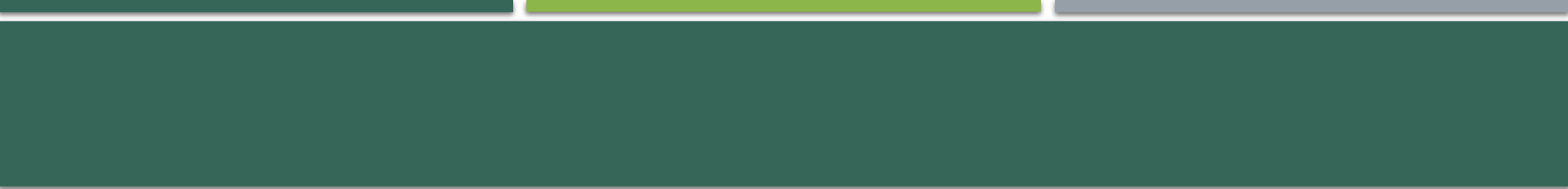
Processus 1



Processus 2



toto.txt



Hello world ! père père père père  
père père père fils père fils père  
fils père fils père fils père fils  
père fils père fils père fils père  
fils père fils père fils père fils  
père fils fils fils fils fils fils

## EXERCICE 8 DU TP1

Écrire un programme `monCp` qui prend trois paramètres en ligne de commandes :

- le chemin d'un fichier `f1` (qui est supposé exister)
- le chemin d'un fichier `f2` (qui n'existe pas forcément : il faut le créer si ce n'est pas le cas)
- un entier `T`. On rappelle que pour transformer une chaîne de caractères représentant un entier en l'entier correspondant, on peut utiliser la fonction `atoi`.

Votre programme copie tout le contenu de `f1` vers `f2` par "blocs" de `T` octets.

Mesurez la différence de temps d'exécution (avec le programme `time`) en utilisant pour `f1` un très gros fichier texte, et en faisant varier la taille de `T` (essayez avec `T` qui vaut 1, 100, 1000). Donnez vos résultats et commentez.

# DUPLICATION DE DESCRIPTEURS DE FICHIERS

```
int dup(int oldfd);  
int dup2(int oldfd, int newfd);
```

## DESCRIPTION

The **dup()** system call creates a copy of the file descriptor `oldfd`, using the lowest-numbered unused file descriptor for the new descriptor.

After a successful return, the old and new file descriptors may be used interchangeably. They refer to the same open file description (see `open(2)`) and thus share file offset and file status flags; for example, if the file offset is modified by using `lseek(2)` on one of the file descriptors, the offset is also changed for the other.

### **dup2()**

The **dup2()** system call performs the same task as `dup()`, but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in `newfd`. If the file descriptor `newfd` was previously open, it is silently closed before being Reused.

## RETURN VALUE

On success, these system calls return the new file descriptor. On error, -1 is returned,

# DUPLICATION DE DESCRIPTEURS DE FICHIERS

```
int main() {  
  
    int fd = open("toto.txt", O_RDWR | O_TRUNC | O_CREAT, 0644);  
  
    int fd2 = dup(fd);  
  
    char chaine[] = "coucou ";  
  
    write(fd, chaine, strlen(chaine));  
    write(fd2, chaine, strlen(chaine));  
  
    return 0;  
}
```

# DUPLICATION DE DESCRIPTEURS DE FICHIERS

```
int main() {  
    int fd = open("toto.txt", O_RDWR | O_CREAT | O_TRUNC, 644);  
  
    dup2(fd, 1);  
  
    printf("hello world !\n");  
  
    return 0;  
}
```

# DUPLICATION DE DESCRIPTEURS DE FICHIERS

**La table des descripteurs de fichiers est :**

- **Dupliquée lors d'un fork**
- **Conservée lors d'un execvp**

→ Utilisation pour le TP du shell : gestion des redirections :

```
>>> ls -al > sortie.txt
```

- Détecter la redirection (caractère '>')
- Duplication du processus (fork)
- Le père attend la terminaison du fils
- Le fils ouvre le fichier "sortie.txt"
- Il duplique ce descripteur de fichier vers le descripteur 1
- Il exécute le programme ls (avec execvp)

→ **conséquence : lorsque le programme ls fera des écritures sur la sortie standard, il écrira**