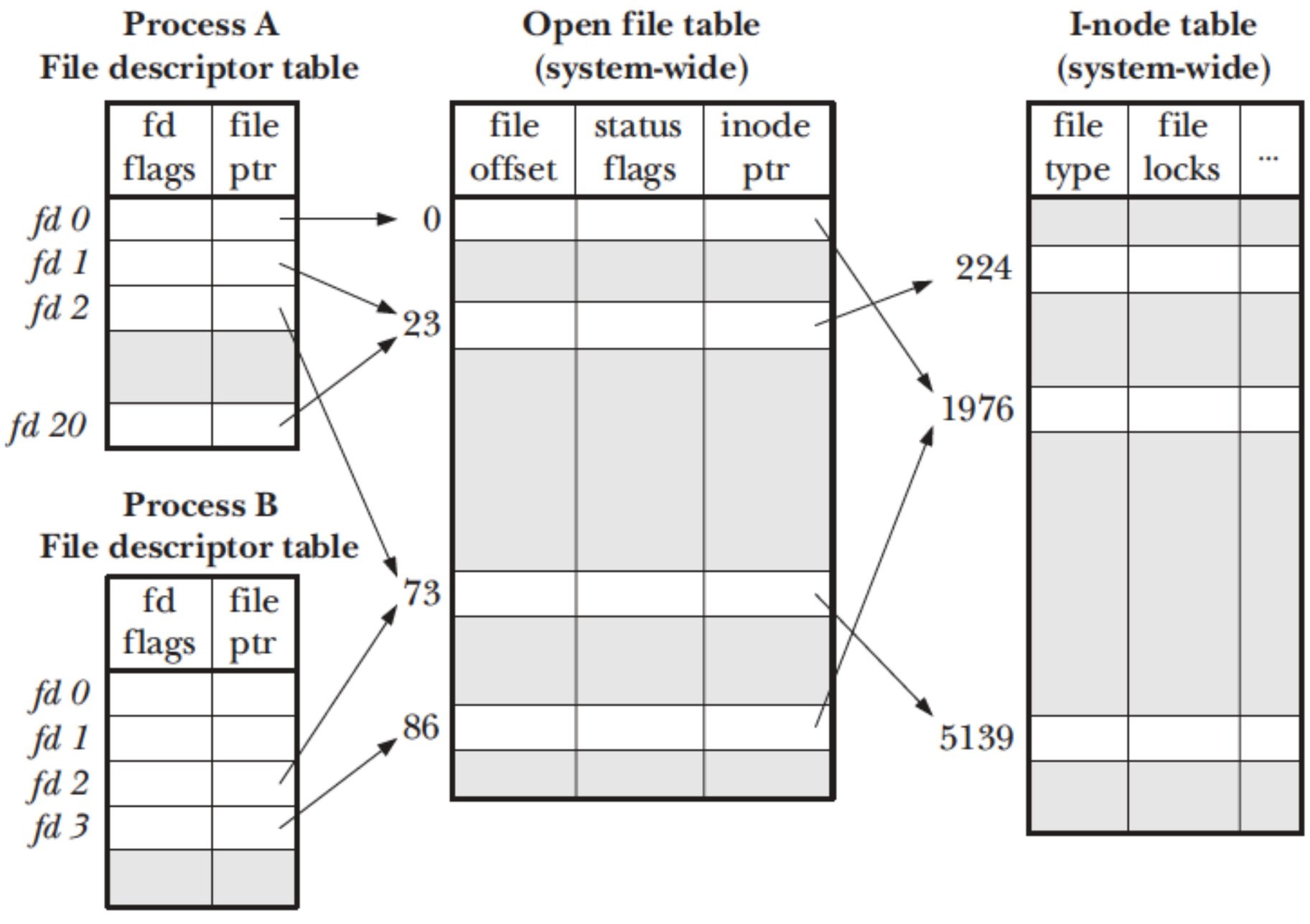
Systèmes avancées
Communication par les fichiers

Wieslaw Zielonka
zielonka@irif.fr

tableaux de descripteurs, de fichiers ouverts et de i-nodes



tableaux de descripteurs

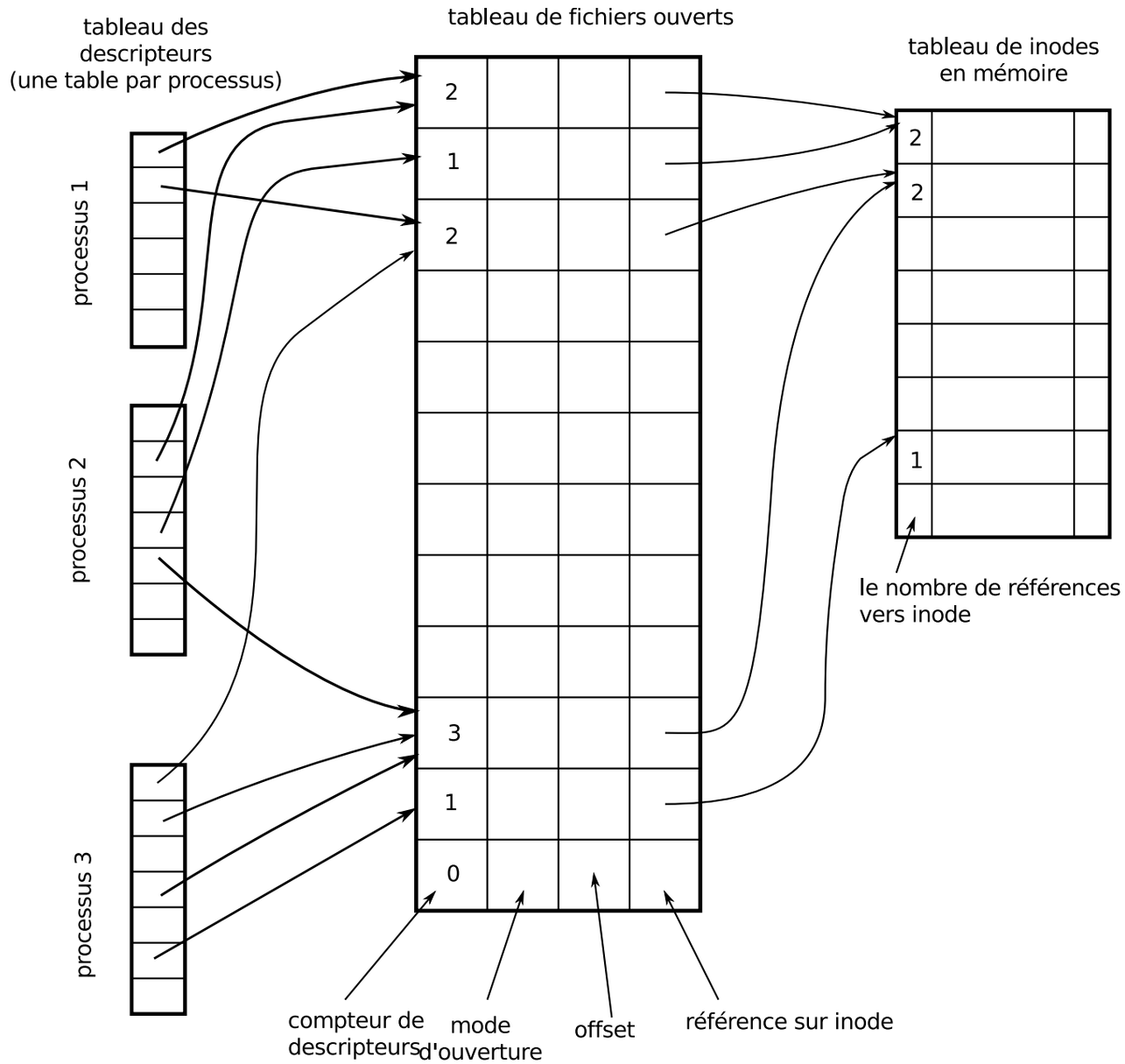


tableau de descripteurs

Le tableau de descripteurs contient pour chaque descripteur ouvert :

- le pointeur vers un élément de tableau de fichiers ouverts
- la valeur du flag close-on-exec O_CLOEXEC. C'est le seul flag associé à un descripteur.

A quoi sert O_CLOEXEC ?

Normalement les appels de la famille exec conservent la table de descripteurs de fichiers du processus. **Une exception :** quand O_CLOEXEC est activé sur un descripteur alors ce descripteur sera automatiquement fermé quand le processus effectue exec.

tableau de fichiers ouverts

Le tableau de fichiers ouverts contient pour chaque fichier ouvert :

- la position courante (offset)
- le mode d'accès spécifié par open() : read, write, read et write
- d'autres flags spécifiés dans open() à l'exception de O_CLOEXEC
- une référence à un i-node
- le compteur du nombre de descripteurs qui pointent vers cet élément

modifier les drapeaux de fichiers ouverts

On peut obtenir la valeur de différents drapeaux de fichier ouvert avec `fcntl`.

Certains drapeaux peuvent être modifiés avec `fcntl` (`O_APPEND`, `O_NONBLOCK` sont les seuls modifiables parmi les drapeaux qu'on utilisera).

duplication de descripteurs

```
./mon_prog  <in.txt  >out.txt
```

Supposons que *mon_prog* lit depuis l'entrée standard (le descripteur 0) et écrit à la sortie standard (descripteur 1).

On demande que le processus exécutant le programme mon_prog lise à partir de fichier in.txt et qu'il écrive dans out.txt sans qu'on change le code source du programme.

Solution : dupliquer les descripteurs avant d'effectuer exec pour exécuter le programme mon_prog

dup()

```
#include <unistd.c>

int dup(int oldfd);

oldfd – un descripteur valide
```

RETOUR : nouveau descripteur si OK, -1 si erreur

La fonction retourne un nouveau descripteur de fichier qui pointe **vers le même élément du tableau de fichiers ouverts** que le descripteur *oldfr*.

Le système garantie que le descripteur retourné par `dup() soit` le plus petit descripteur libre dans le tableau de descripteurs.

exemple

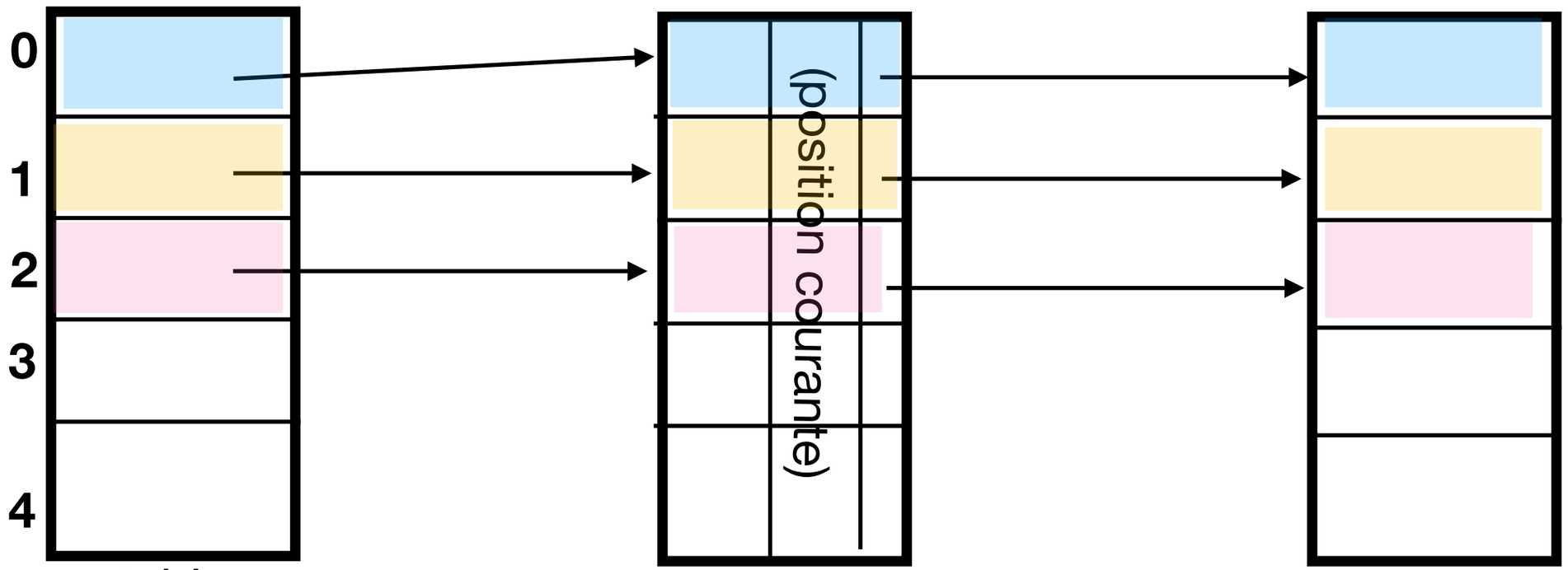


table
de descripteurs
du processus

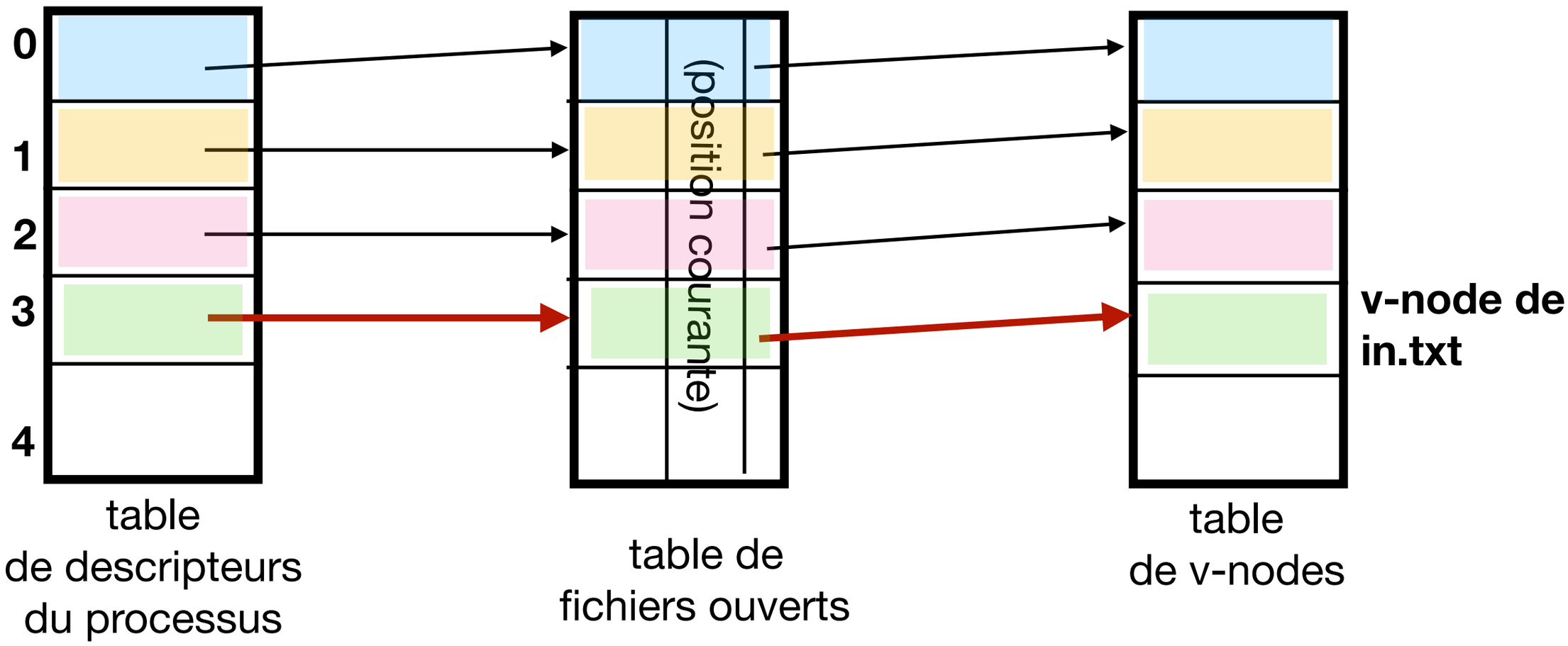
table de
fichiers ouverts

table
de v-nodes

Configuration initiale, les descripteurs 0, 1, 2 ouverts sur l'entrée standard, sortie standard et sortie d'erreurs standard.

exemple

```
int d = open("in.txt", O_RDONLY);
```

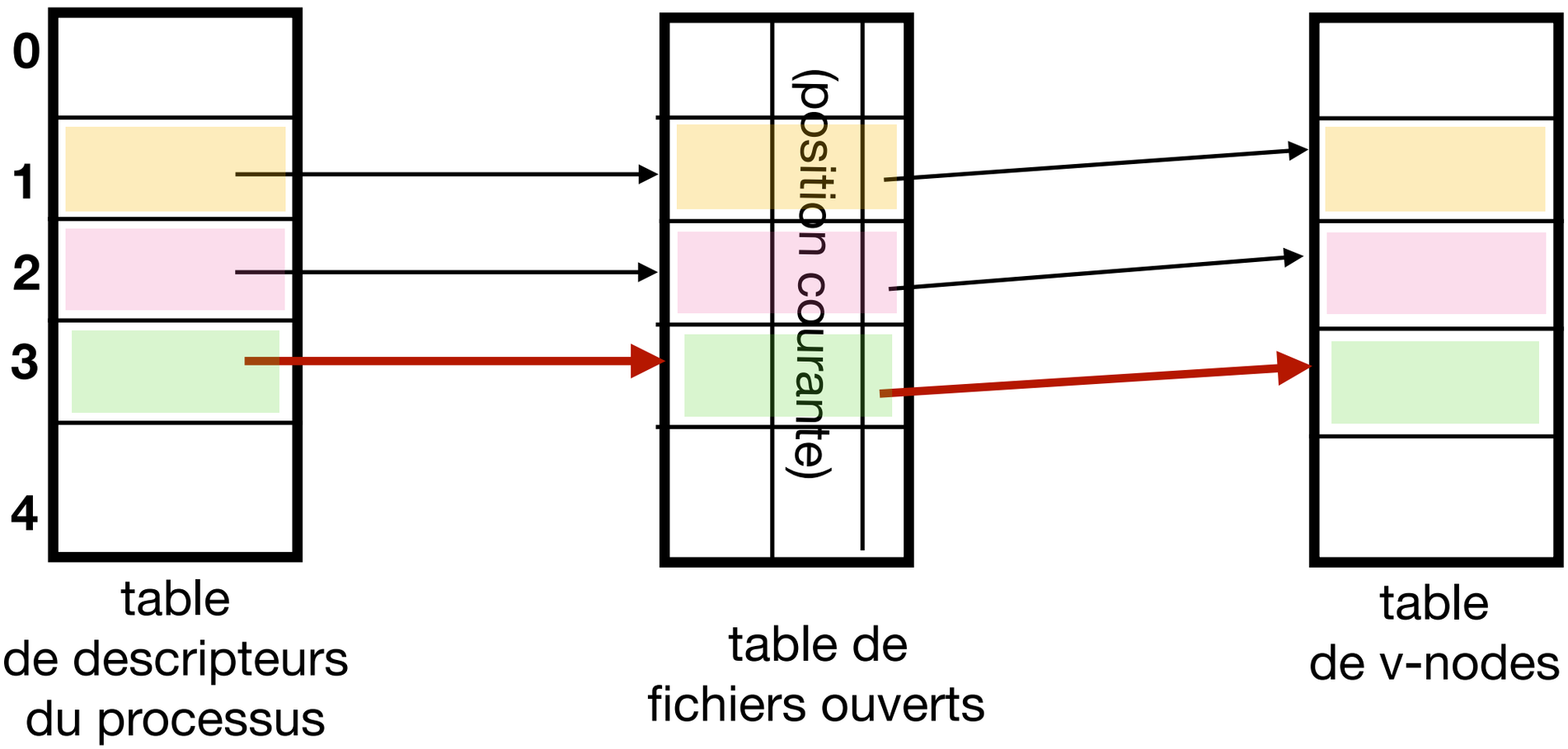


d == 3, nouveau descripteur ouvert

exemple

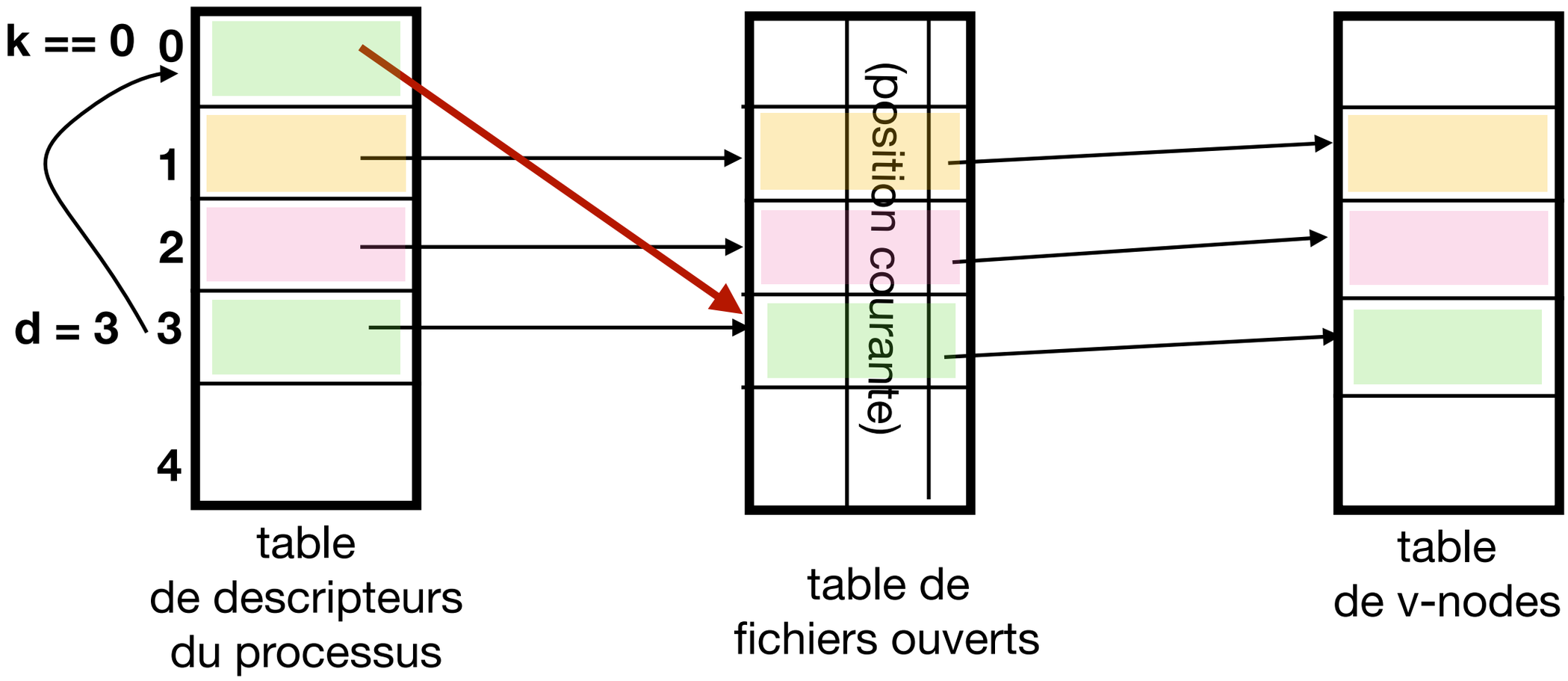
```
close( 0 );
```

Descripteur 0 devient libre.



exemple dup()

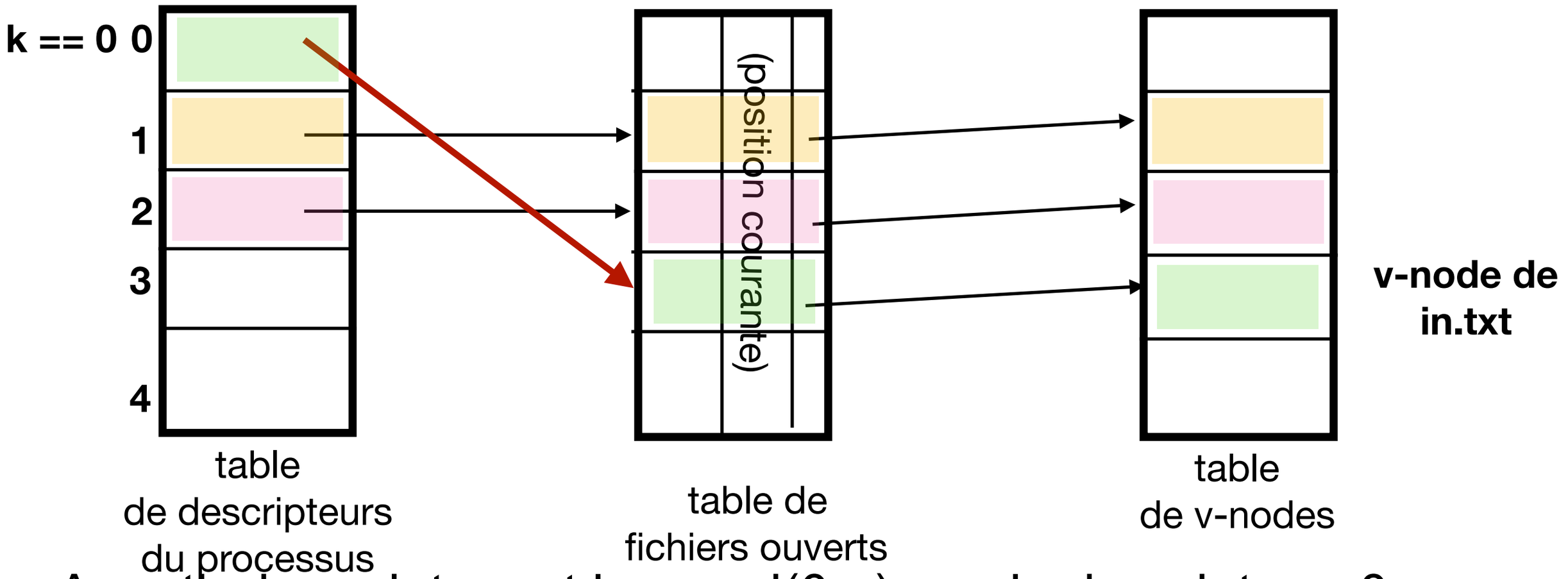
```
int k = dup( d );
/*  avec  d == 3,  k <-- 0 */
```



Le descripteur 3 dupliqué dans 0.

exemple

```
close( d );
/*  fermer le descripteur 3 non utilisé */
```



A partir de maintenant les read(0,) sur le descripteur 0 provoquent la lecture du fichier `in.txt.` Donc on peut faire exec...() pour exécuter le programme mon_prog.

dup()

Inconvénient de dup() :

il faut être sûr que le descripteur "cible" (celui que nous voulons ouvrir) soit le premier descripteur libre.

`Le remède : utilisez dup2() à la place de dup()`

dup2()

```
#include <unistd.h>

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

RETOUR: le nouveau descripteur si OK, -1 si erreur

(si `oldfd` n'est pas un descripteur valide alors errno == EBADF)

- si le descripteur `newfd` est ouvert avant l'appel alors il sera d'abord fermé.
- ensuite le descripteur `oldfd` est recopié dans `newfd`

exemple

Redirection de descripteurs :

1) 0 vers le fichier `in.txt`,
2) 1 vers le fichier `out.txt` :

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

int out = open("out.txt", O_WRONLY);

dup2(in, 0);

dup2(out, 1);

close( in ); close(out); // fermer les descripteurs qui ne sont
                         //plus utiles
```

l'écriture dans le descripteur 1 entraînera l'écriture dans le fichier `out.txt`.

la lecture dans 0 provoque la lecture `in.txt`

exemple

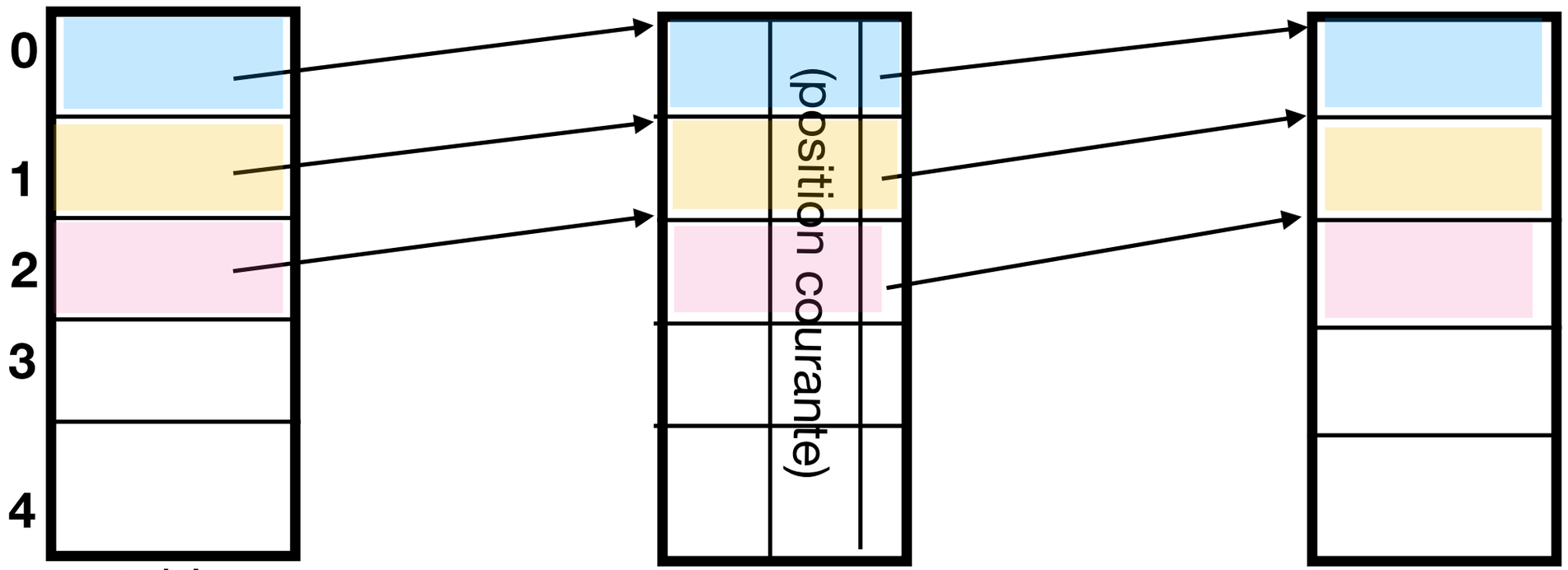


table
de descripteurs
du processus

table de
fichiers ouverts

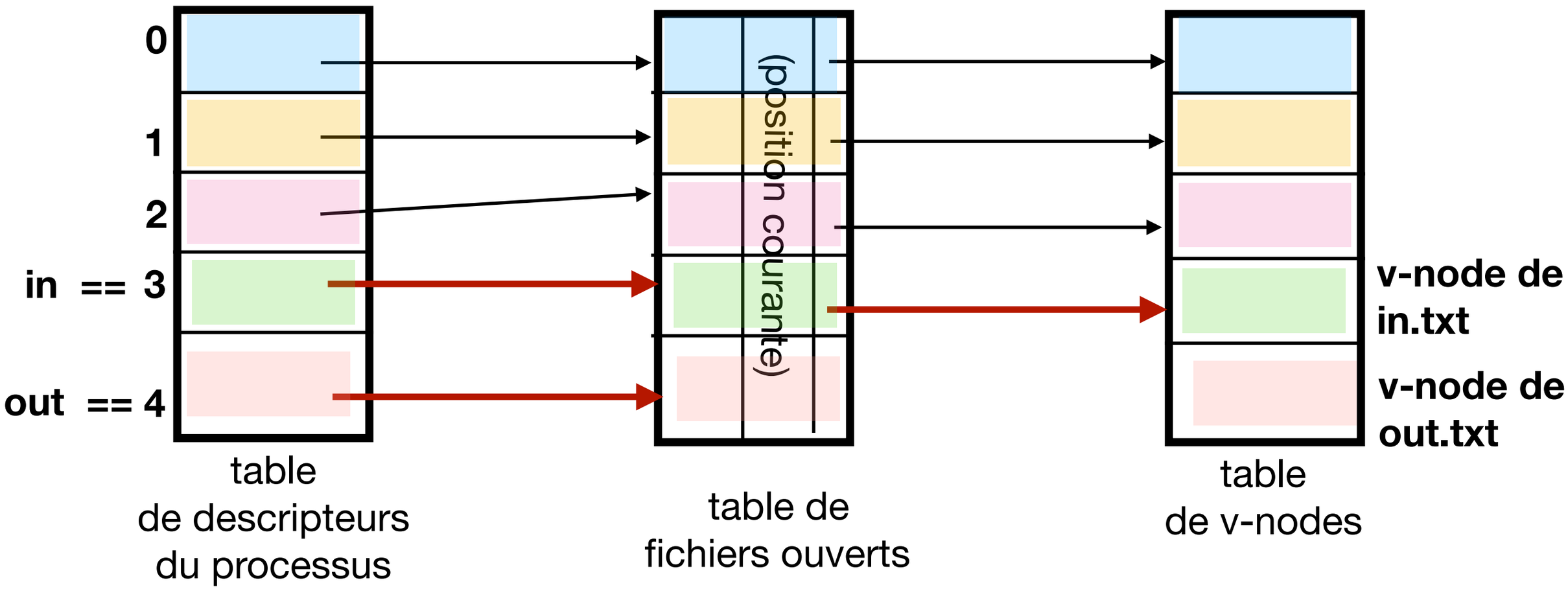
table
de v-nodes

Configuration initiale, les descripteurs 0, 1, 2 ouverts sur l'entrée standard, sortie standard et sortie d'erreurs standard.

exemple

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

int out = open("out.txt", O_WRONLY);
```

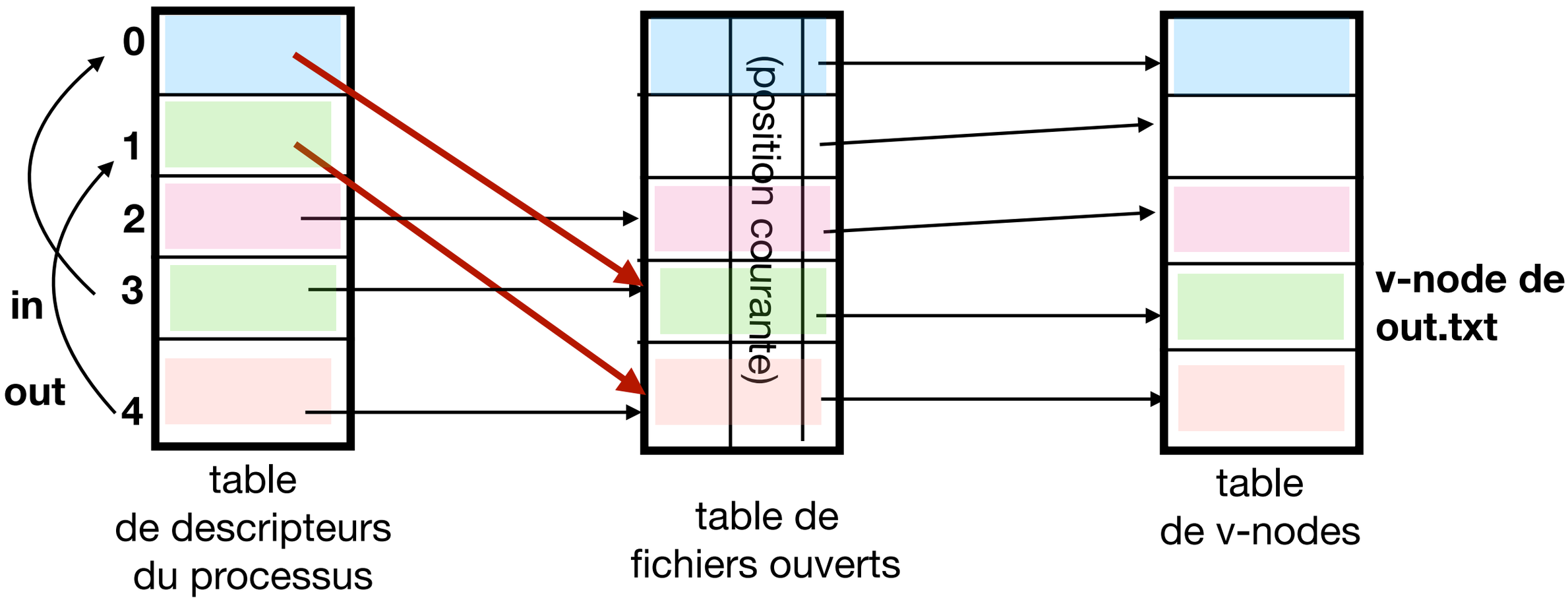


in == 3 et out == 4 nouveaux descripteurs ouverts

exemple

```
dup2(in, 0);

dup2(out, 1);
```

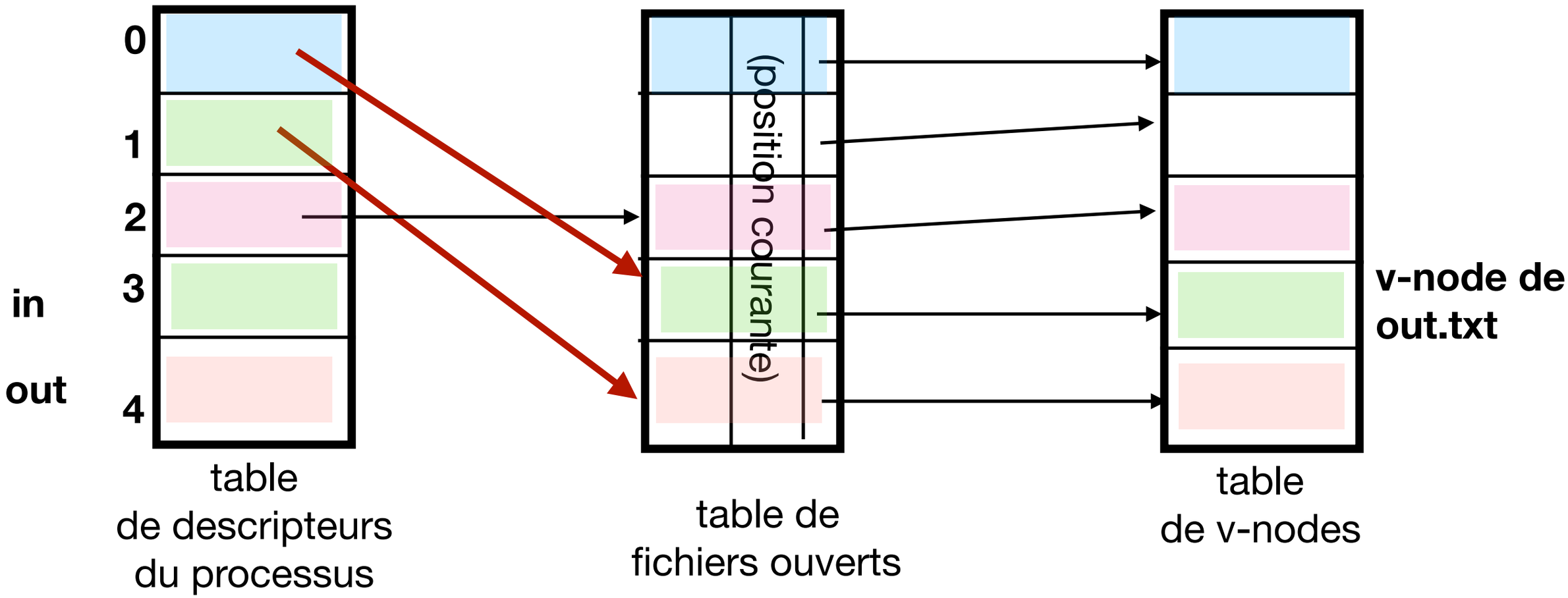


in copié dans 0
out copié dans 1

exemple

```
close(in);

close(out);
```



fermer in et out

descripteurs de fichiers et exec..()

Rappelons que le processus enfant créé par fork() possède son propre tableau de descripteurs qui est une copie exacte du tableau de descripteurs du processus père. La copie s'effectue au moment de la création du processus enfant.

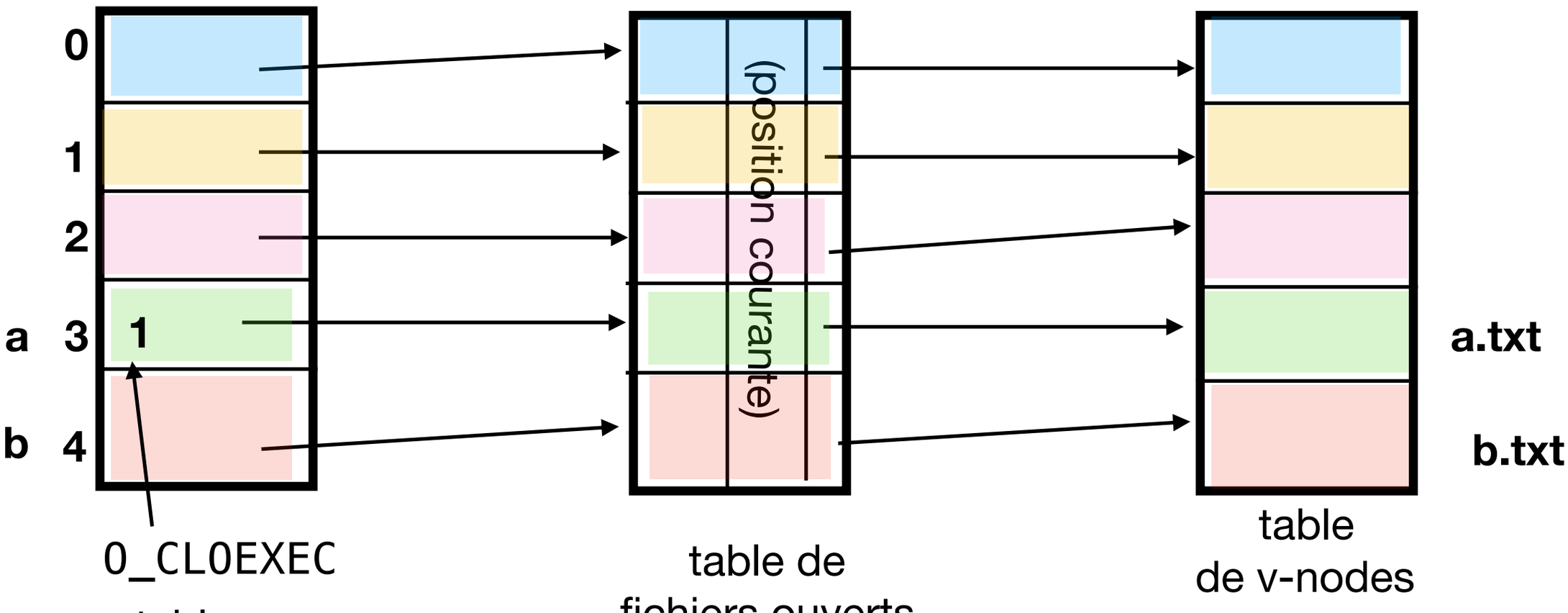
Après fork() les deux tableaux de descripteurs, celui du père et celui de l'enfant, sont différents. Le père et l'enfant peuvent ouvrir et fermer les fichiers de façon indépendante.

le tableau de descripteurs et exec..()

Le tableau de descripteur d'un processus est préservé par l'appel à exec..(), sauf pour les descripteurs pour lesquels le drapeau close-on-exec O_CLOEXEC est activé.

Le drapeau O_CLOEXEC est le seul drapeau associé a un descripteur de fichiers.

exemple




```
int a = open("a.txt", O_RDWR | O_CLOEXEC);
int b = open("b.txt", O_RDWR);
```

```
pid_t pid = fork() ;
```

table
de descripteurs
du processus

père

	0	
	1	
	2	
a	3	1
b	4	

(position courante)

table de
fichiers ouverts

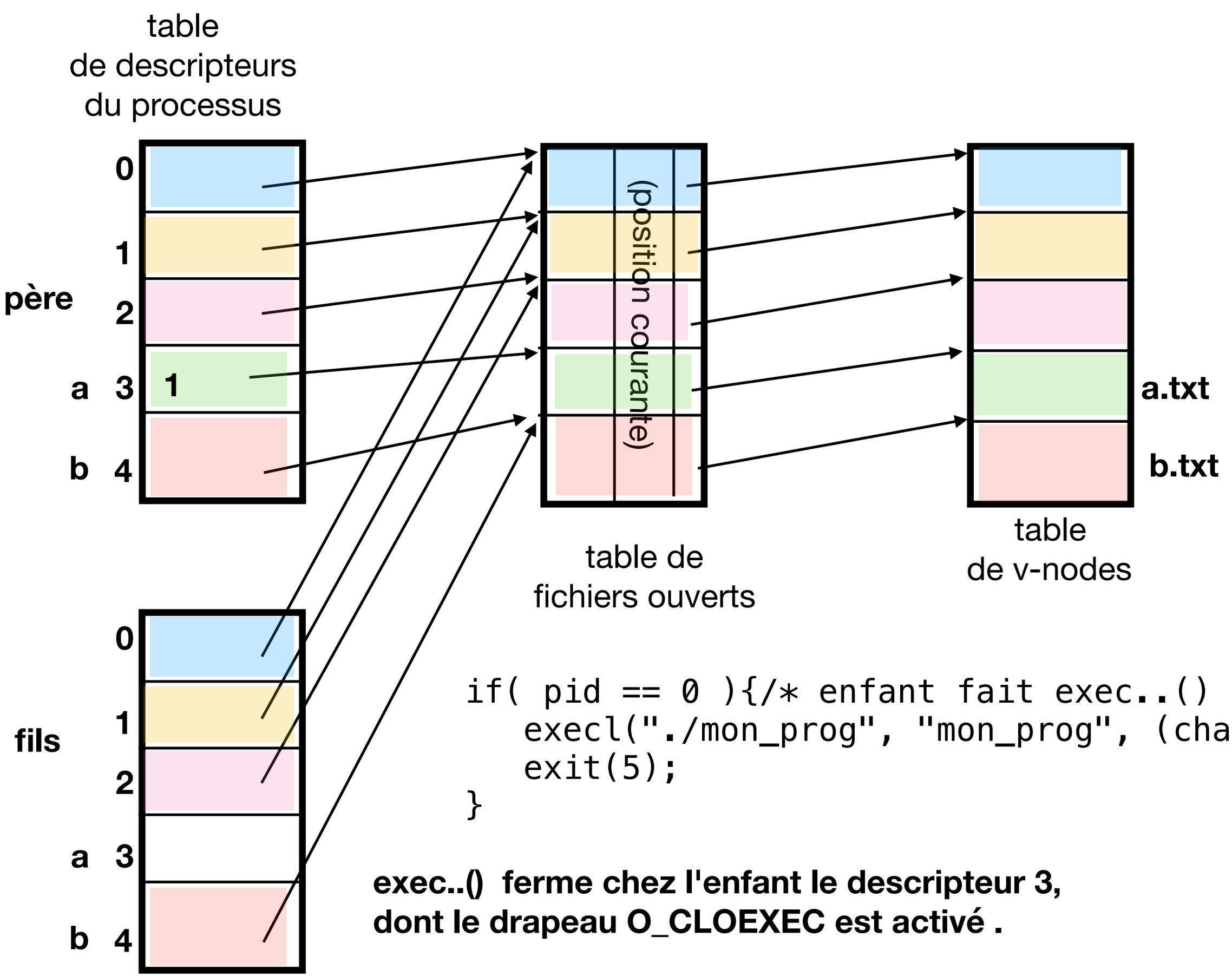
a.txt

b.txt

table
de v-nodes

enfant

	0	
	1	
	2	
a	3	1
b	4	




```
if( pid == 0 ){/* enfant fait exec..()
   execl("./mon_prog", "mon_prog", (char *)0);
   exit(5);
}
```

exec..() ferme chez l'enfant le descripteur 3, dont le drapeau O_CLOEXEC est activé .

Verrous de fichier

Nécessité de verrou : un exemple

Verrouillage de fichier

Verrous :

- poser un verrou
- effectuer une opération (ou des opérations) de lecture/écriture
- enlever le verrou.

Deux types de verrou :

- le verrou consultatif (advisory) : un processus peut ignorer les verrous posés par d'autre processus et accéder au fichier. Pour que les verrous puissent avoir un effet tous les processus doivent coopèrer. C'est le verrouillage par défaut.

- le verrou obligatoire (mandatory) : le processus essayant d'accèder au fichier verrouillé sera contraint à respecter les conditions imposées par le verrouillage même s'il ne veut pas coopérer.

Verrous flock()

Les verrous BSD, ne font pas partie de POSIX mais présent sur presque tous les systèmes UNIX.

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

int flock(int fd, int operation)
```

pose le verrou sur tout le fichier. Retourne 0 en cas de succès, -1 sinon.

- fd - un descripteur de fichier valide

Verrous flock() - opérations

valeur	description
LOCK_SH	placer le verrou partagé sur un fichier dont le descripteur est fd
LOCK_EX	placer le verrou exclusif sur le fichier fd
LOCK_UN	déverrouiller le fichier dont le descripteur est fd
LOCK_NB	faire une demande de verrou non bloquante

Verrous flock()

Plusieurs processus peuvent posséder en même temps les verrous partagés sur le même fichier. On pose le verrou partagé pour effectuer une lecture.

A chaque moment seulement au plus un processus peut posséder un verrou exclusif sur un fichier, d'autres processus ne peuvent pas avoir ni verrou exclusif ni partagé pendant ce temps. On pose le verrou exclusif pour effectuer des écritures et/ou des lectures.

Il est possible de convertir un verrou d'un type (partagé/exclusif) vers un autre type (exclusif/partagé) avec l'appel à flock() en spécifiant le nouveau type de verrou.

Conversion de verrou partagé vers exclusif peut être bloquante (s'il y a d'autres processus possédant les verrous). La conversion de verrou n'est pas atomique, elle se déroule en deux étapes

1. la libération de verrou que le processus possède suivi de
2. la tentative d'obtention de nouveau verrou.

Mais entre 1 et 2 un autre processus peut obtenir le verrou donc notre processus peut perdre le verrou après 1 et être bloqué sur l'opération 2.

Verrous flock()

L'opération d'acquisition de verrou est normalement bloquante (le processus bloque jusqu'à l'obtention de verrou).

On ajoute LOCK_NB pour la rendre non-bloquante:

```
int n = flock( fd, LOCK_EX | LOCK_NB );

if( n == -1 && errno == EWOULDBLOCK ){

    // c'est le cas ou le verrou n'est pas acquis parce que

    // d'autres processus détiennent un verrou non-compatible

    // avec le verrou exclusif demandé.

  .....

}
```

De la même façon LOCK_SH | LOCK_NB rend la demande de verrou partagé non-bloquante.

Verrous flock() et fork, exec

Les verrous flock() sont associés à la table de fichier ouvert.

Donc si deux descripteurs pointent vers la même entrée dans la table de fichiers ouverts alors les deux processus possèdent le même verrou.

Conséquences:

A. processus A obtient le verrou sur un fichier xyz.txt

B. processus A fait fork() et crée un processus enfant B

C. les deux processus A et B possèdent le verrou (même dans le cas de verrou exclusif).

Ce comportement permet de passer le verrou flock() du parent vers l'enfant de manière atomique :

1. processus A obtient le verrou et fait fork() en créant l'enfant B
2. le parent fait close() sur le descripteur, à partir de ce moment seulement l'enfant détient le verrou
3. l'enfant fait exec() -- le verrou est préservé à travers de exec().

Verrous flock() et dup/dup2

Quand un processus duplique un descripteur avec dup() ou dup2() les deux descripteurs partagent toujours le verrou. Quand on libère le verrou sur un descripteur le verrou est automatiquement libéré sur l'autre descripteur.

```
flock( fd, LOCK_EX );

newfd = dup( fd );

flock( newfd, LOCK_UN);  // libère le verrou de newdf

                         // et de fd
```

Si on ouvre le même fichier deux fois avec open() le deux descripteurs sont traités de façon indépendante par les verrous flock.

limitation de flock

- pas de possibilité d'obtenir le verrou sur une partie du fichier, cela limite le parallélisme,
- seulement les verrous consultatifs
- le système de fichier NFS souvent ne prend pas en compte les verrous flock

verrous fcntl

Ce sont les verrous POSIX. Permettent de faire verrouiller une partie du fichier.

struct flock flockstr;

/* remplir la structure flockstr */

fcntl(fd, cmd, &flockstr);

fd - descripteur de fichier dont une partie nous voulons verrouiller / déverrouiller.

verrous fcntl - paramètre cmd

le paramètre cmd de fcntl prend une de trois valeurs:

F_SETLK - obtenir ou libérer un verrou. Si l'acquisition de verrou impossible parce qu'un autre processus détient un verrou incompatible alors fcntl() retourne tout de suite -1 et errno prend la valeur EAGIN ou EACCES

F_SETLKW - comme précédent, mais l'appel est bloquant jusqu'à ce que l'acquisition de verrou devienne possible. Un signal peut interrompre l'attente (errno == EINTR). Utile pour l'alarme ou timeout.

F_GETFLK - permet de vérifier si l'acquisition de verrou est possible (sans le demander). Le type de verrou *l_type* de struct flock doit être F_RDLCK ou F_WRLCK.

Si l'acquisition de verrou est possible *l_type* prendra la valeur F_UNLCK.

Si l'acquisition de verrou est impossible `struct flock` retournera l'information sur un verrou incompatible détenu par un autre processus avec l'identité de ce processus (champs `l_pid`).

verrous fcntl - struct flock

```
struct flock{
   short   l_type; //type de verrou : F_RDLCK, F_WRLCK, F_UNLCK
   short   l_whence ; // valeurs SEEK_SET, SEEK_CUR, SEEK_END
  off_l    l_start; //offset du début de verrou
    off_t    l_len; // longueur de verrou en octets
   pid_t l_pid;  // pid du processus qui empêche la pose de verrou
                  //uniquement avec LOCK_UN
}
```

verrous fcntl- struct flock

`l_len == 0`

indique qu'on demande de verrouiller tous les octets à partir d'une position donnée et le verrou s'étend automatiquement quand le fichier grossit.

verrous fcntl

- libérer un verrou qu'on ne possède pas ne pose pas de problème
- changer le verrou sur certain segment de fichier est atomique (contrairement à `flock`)
- un processus ne peut jamais avoir le conflit de verrou avec le verrou qu'il a posé lui-même (contrairement à flock)
- placer un verrou au milieu de segment déjà verrouillé peut diviser le segment en trois segments avec de verrous distincts.
- avec F_SETLKW un deadlock de processus est possible. Le noyau détecte quand le deadlock survient et `errno == EDEADLK` indique cette situation.

verrous fcntl

- les verrous fcntl **ne sont pas** hérités du père à fils par un fork()
- les verrous fcntl sont préservés par exec (mais attention à close-on-exec flag)
- tous les threads d'un processus partagent les verrous (donc les verrous flock inutile pour synchroniser l'accès entre les threads du même processus)
- les verrous associés au processus et i-node. Conséquence :

  fermeture d'un descripteur libère tous les verrous sur le fichier, peu importe comment obtenus:

```
struct flock f1;
f1.l_whence = SEEK_SET;  f1.l_type = F_WRLCK;
f1.l_start  = 0;         f1.l_len  = 0;
fd1 = fopen("fichier", O_RDWR);
fd2 = fopen("fichier", O_RDWR);
if( fcntl( fd1, cmd, &f1) == -1 ){ exit(1);} //poser le verrou sur fd1
close( fd2 );  // ceci libère le verrou sur fd1!!!
```

La sémantique de héritage et de libération de verrous fcntl limitent **considérablement** leur utilité.

Exemple de problème : les verrous posés dans une librairie peuvent être libérés par inadvertance.