Chapitre 5

Appels système du Système de Gestion de Fichier

Les appels système d'entrées-sorties ou entrées-sorties de bas niveau sont rudimentaires mais polymorphes, en effet c'est eux qui permettent d'écrire des programmes indépendamment des supports physiques sur lesquels se font les entrées/sorties et de pouvoir facilement changer les supports physiques associés a une entrée-sortie.

Les appels système du système de gestion de fichier sont :

`open/creat` ouverture/création d'un fichier

`read/write` lecture/ecriture sur un fichier ouvert

`lseek` déplacement du pointeur de fichier

`dup,dup2` copie d'ouverture de fichier

`close` fermeture d'un fichier

`mount` chargement d'un disque

`mknode` création d'un inode de fichier spécial

`pipe` création d'un tube

`fcntl` manipulation des caractéristiques des ouvertures de fichiers

Les appels système sont réalisés par le noyau et retournent -1 en cas d'erreur.

5.1 `open`

```
#include <fcntl.h>
int open(char *ref, int mode, int perm);
```

Ouverture du fichier de référence (absolue ou relative à ".") *ref.*
Le *mode* d'ouverture est une conjonction des masques suivants :

```
O_RDONLY  /* open for reading */
O_WRONLY  /* open for writing */
O_RDWR    /* open for read & write */
O_NDELAY  /* non-blocking open */
O_APPEND  /* append on each write */
O_CREAT   /* open with file create */
O_TRUNC   /* open with truncation */
O_EXCL    /* error on create if file exists*/
```

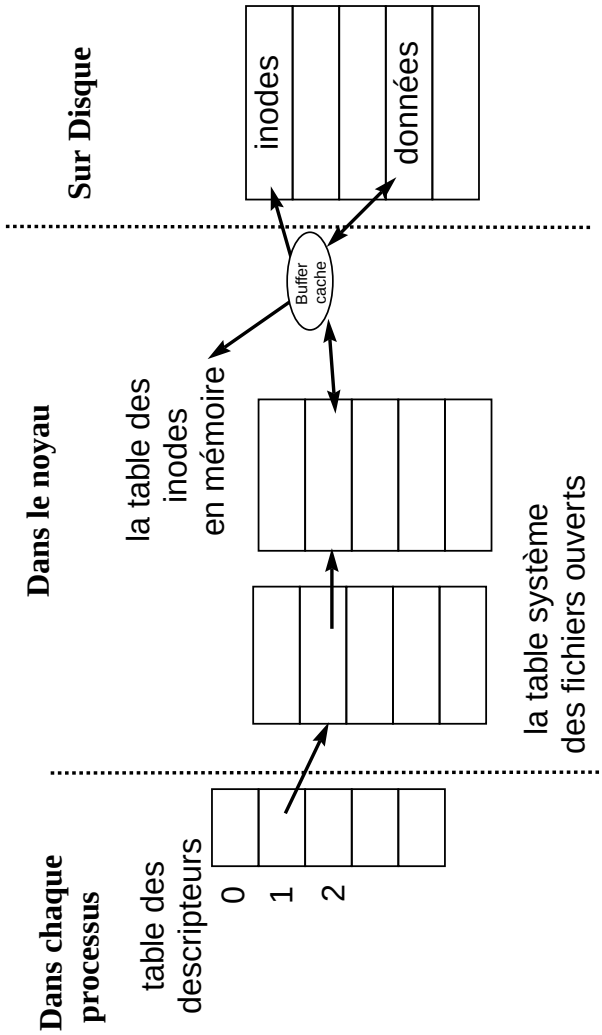

FIG. 5.1 – Tables du système de fichiers.

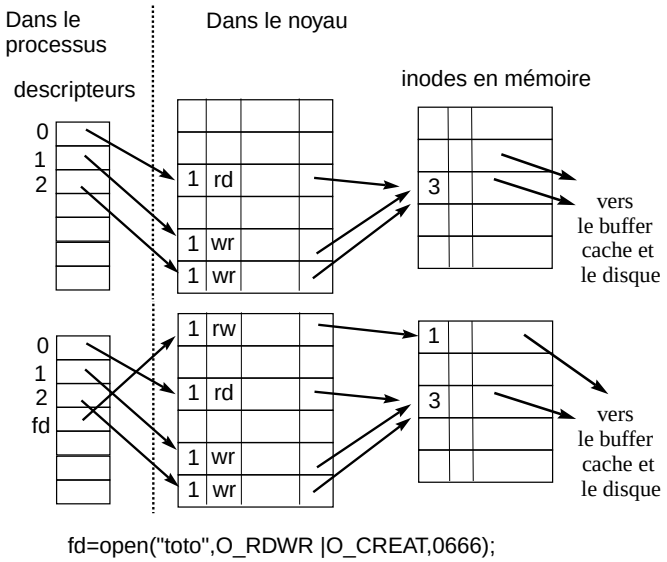

FIG. 5.2 – Avant l'ouverture, descripteurs standard ouverts, puis après l'ouverture de "toto".

Le paramètre *perm*ission n'a de sens qu'à la création du fichier, il permet de positionner les valeurs du champ `mode` de l'inode. Les droits effectivement positionnés dépendent de la valeur de *umask*, grace à la formule `droits = perm & ~ umask`. La valeur par défaut de `umask` est 066 (valeur octale).
La valeur de retour de `open` est le numéro dans la table de descripteurs du processus qui a été utilisé par `open`. Ce numéro est appelé descripteur de l'ouverture. Ce descripteur est utilisé dans les autres appels système pour spécifier l'ouverture de fichier sur laquelle on veut travailler[1], et -1 en cas d'échec de l'ouverture.

5.1.1 Déroulement interne d'un appel de `open`

1. Le système détermine l'inode du fichier référence (*namei*).
2. – Soit l'inode est dans la table des inodes en mémoire.
   – Soit il alloue une entrée et recopie l'inode du disque (*iget*).
3. Le système vérifie les droits d'accès dans le mode demandé.
4. Il alloue une entrée dans la table des fichiers ouverts du système, et positionne le curseur de lecture écriture dans le fichier (offset = 0, sauf dans le cas du mode `O_APPEND` offset=taille du fichier).
5. Le système alloue une place dans la table des descripteurs `_iob` du fichier.
6. Il renvoie au processus le numéro de descripteur, c'est à dire le numéro de l'entrée qu'il vient d'allouer dans le tableau `_iob`.

Si l'opération a échoué dans une des étapes le système renvoie `-1`.

[1] Un même fichier peut être ouvert plusieurs fois.

5.2 creat

Création d'un fichier et ouverture en écriture.

```
int creat(char *reference, int permissions);
```

1. Le système détermine l'inode du catalogue où l'on demande la création du fichier.
   (a) Si il existe déjà une inode pour le fichier
   - Le noyau lit l'inode en question (allocation dans la table des inodes en mémoire), vérifie que c'est un fichier ordinaire autorisé en écriture par le propriétaire effectif du processus, sinon échec.
   - Le système libère les blocs de données et réduit la taille du fichier à zéro, il ne modifie pas les droits qu'avait le fichier antérieurement.

   (b) Si n'existait pas d'inode pour le fichier
   - Le système teste les droits en écriture sur le catalogue
   - Il alloue une nouvelle inode (*ialloc*)
   - Il alloue une nouvelle entrée dans la table des inodes en mémoire.

Même suite que pour `open`.

5.3 read

```
int nbcharlus = read(int d, char *tampon, int nbalire)
```

***d*escripteur** entrée de la table des descripteurs correspondante au fichier dans lequel doit être effectuée la lecture (fourni par `open`).

nbalire nombre de caractères à lire dans le fichier.

tampon un tableau de caractères alloué par l'utilisateur. Les caractères lus sont placés dans ce tampon.

nbcharlus nombre de caractères effectivement lus, ou -1 en cas d'échec de l'appel système, (droits, ...), la fin de fichier est atteinte quand le nombre de caractères lus est inférieur au nombre de caractères demandés.

Déroulement :

1. Vérification du descripteur $\longrightarrow$ accès aux tables système.
2. Droits (mode adéquat)
3. Grâce à l'inode le système obtient les adresses du (des) bloc(s) contenant les données à lire. Le système effectue la lecture de ces blocs.
4. Le système recopie les données du buffer cache vers le tampon de l'utilisateur.
5. Le curseur dans le fichier est remit à jour dans l'entrée de la table des fichiers ouverts.
6. Le système renvoie le nombre de caractères effectivement lus.

5.4 write

```
int nbcecrits = write(int desc, char *tampon, int nbaecrire);
```

Même déroulement que `read` mais avec une allocation éventuelle de bloc-disque dans le cas d'un ajout au-delà de la fin du fichier.
Dans le cas où l'appel concerne un périphérique en mode caractère : le système active la fonction `write` (réciproquement `read` pour une lecture) du périphérique qui utilise directement l'adresse du tampon utilisateur.

Remarquons ici encore le polymorphisme de ces deux appels système qui permet de lire et d'écrire sur une grande variété de périphériques en utilisant une seule syntaxe. Le code C utilisant l'appel système marchera donc indifféremment sur tous les types de périphériques qui sont définis dans le système de fichier. Par exemple, il existe deux périphériques "logiques" qui sont /dev/null et /dev/zéro (que l'on ne trouve pas sur toutes les machines). Le premier est toujours vide en lecture et les écritures n'ont aucun effet (il est donc possible de déverser n'importe quoi sur ce périphérique). Le deuxième fournit en lecture une infinité de zéro et n'accepte pas l'écriture.

5.5 lseek

```
#include <fcntl.h>
off_t  lseek(int d, off_t offset, int direction)
```

`lseek` permet de déplacer le curseur de fichier dans la **table des fichiers ouverts** du système.
offset un déplacement en octets.
d le descripteur.
direction une des trois macros L_SET, L_INCR, L_XTND.

L_SET la nouvelle position est *offset* sauf si *offset* est supérieur à la taille du fichier, auquel cas la position est égale à la taille du fichier. Si l'offset est négatif, alors la position est zéro.

L_INCR la position courante est incrémentée de *offset* place (même contrainte sur la position maximum et la position minimum).

L_XTND Déplacement par rapport à la fin du fichier, cette option permet d'augmenter la taille du fichier (ne pas créer de fichiers virtuellement gros avec ce mécanisme, ils posent des problèmes de sauvegarde).

La valeur de retour de lseek est la nouvelle position du curseur dans le fichier ou -1 si l'appel a échoué.

5.6 dup et dup2

Les appels dup et dup2 permettent de dupliquer des entrées de la table des descripteurs du processus.

```
int  descripteur2 = dup(int descripteur1);
```

1. vérification que descripteur est le numéro d'une entrée non nulle.
2. recopie dans la **première entrée** libre du tableau des descripteurs l'entrée correspondant à descripteur1.
3. le compteur de descripteurs de l'entrée associée à descripteur1 dans la table des ouvertures de fichiers est incrémenté.
4. renvoi de l'indice dans la table des descripteurs de l'entrée nouvellement allouée.

Redirection temporaire de la sortie standard dans un fichier :

```
tempout = open("sortie_temporaire",1);
oldout = dup(1);
close(1);
newout = dup(tempout); /* renvoie 1  */
write(1,"xxxx",4); /* ecriture dans le fichier temporaire */
```

```
close(tempout);
close(1);
newout = dup(oldout);
close(oldout);
```

Il est aussi possible de choisir le descripteur cible avec

```
int ok = dup2(int source, int destination);
```

Recopie du descripteur `source` dans l'entrée `destination` de la table des descripteurs. Si `destination` désigne le descripteur d'un fichier ouvert, celui-ci est préalablement fermé avant duplication. Si `destination` n'est pas un numéro de descripteur valide, il y a une erreur, retour -1.

5.7 close

Fermeture d'un fichier.

```
int ok = close(descripteur);
```

1. si *descripteur* n'est pas un descripteur valide retour -1
2. l'entrée d'indice `descripteur` de la table est libérée.
3. Le compteur de l'entrée de la table des fichiers ouvert associé à `descripteur` est décrémenté.

   Si il passe à Zéro alors
4. l'entrée de la table des fichiers ouverts est libérée et le compteur des ouvertures de l'inode en mémoire est décrémenté.

   Si il passe à Zéro alors
5. l'entrée dans la table des inodes en mémoire est libérée.

   Si de plus le compteur de liens de l'inode est à 0 alors
6. le fichier est libéré : récupération de l'inode et des blocs.

Dans le cas d'une ouverture en écriture : le dernier bloc du buffer cache dans lequel on a écrit est marqué "a écrire".

5.8 fcntl

L'appel système `fnctl` permet de manipuler les ouverture de fichier après l'ouverture, bien sur il n'est pas possible de changer le mode d'ouverture (lecture/écriture/lecture-écriture) après l'ouverture.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int desc, int commande);
int fcntl(int desc, int commande, long arg);
int fcntl(int desc, int commande, struct flock *verrou);
```

L'appel système `fnctl` permet de positionner des verrous de fichier voire le chapitre 12. L'appel système `fnctl` permet la manipulation de certains des drapeaux d'ouverture :

O_APPEND

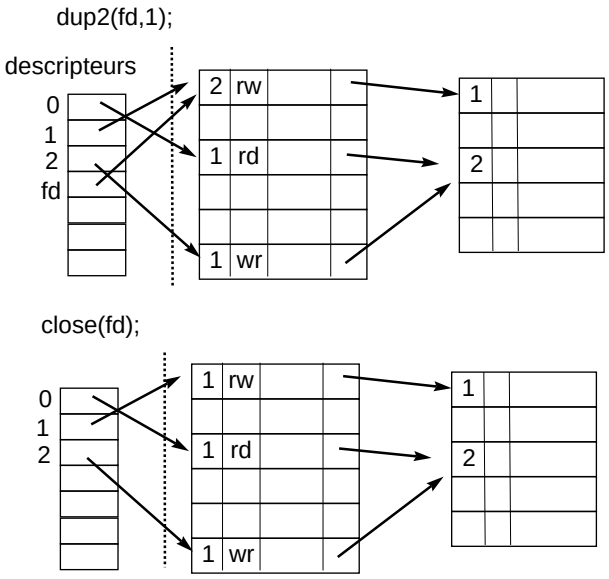

FIG. 5.3 – Redirection de la sortie standard sur "toto".

O_NONBLOCK

O_ASYNC

O_DIRECT

L'appel système `fnctl` permet gerer les signaux associés aux entrée asyncrones.