Chap. VIII : Manipulation des fichiers en C

Laurent Poinsot

UMR 7030 - Université Paris 13 - Institut Galilée

Cours “Architecture et Système”

Nous avons déjà abordé le sujet des fichiers que ce soit en cours (partie “ Architecture ”, chap. IV “ Le système de fichiers ”), ainsi qu’en travaux dirigés et pratiques. Nous savons donc qu’il est possible de manipuler les fichiers (en langage C) *via* des pointeurs de lecture/écriture. Dans ce chapitre, nous verrons une autre manière de gérer les fichiers à l’aide de descripteurs de fichiers.

Nous avons déjà abordé le sujet des fichiers que ce soit en cours (partie “ Architecture ”, chap. IV “ Le système de fichiers ”), ainsi qu’en travaux dirigés et pratiques. Nous savons donc qu’il est possible de manipuler les fichiers (en langage C) *via* des pointeurs de lecture/écriture. Dans ce chapitre, nous verrons une autre manière de gérer les fichiers à l’aide de descripteurs de fichiers.

Nous avons déjà abordé le sujet des fichiers que ce soit en cours (partie “ Architecture ”, chap. IV “ Le système de fichiers ”), ainsi qu’en travaux dirigés et pratiques. Nous savons donc qu’il est possible de manipuler les fichiers (en langage C) *via* des pointeurs de lecture/écriture. Dans ce chapitre, nous verrons une autre manière de gérer les fichiers à l’aide de descripteurs de fichiers.

Les types de fichiers

Il y a trois types de fichiers UNIX :

1. les fichiers ordinaires : tableaux linéaires d'octets identifiés par leur i-node ;
2. les répertoires : ces fichiers permettent de repérer un fichier par un nom plutôt que par son i-node dans la table de noeud d'index ; le répertoire est donc grossièrement constitué d'une table à deux colonnes contenant d'un côté le nom que l'utilisateur donne au fichier, et de l'autre, le numéro d'i-node donné par le système qui permet d'accéder à ce fichier. Cette paire est appelée un lien ;
3. les fichiers spéciaux, périphériques, tubes, sockets, ..., que nous aborderons plus loin.

Les types de fichiers

Il y a trois types de fichiers UNIX :

1. les fichiers ordinaires : tableaux linéaires d'octets identifiés par leur i-node ;
2. les répertoires : ces fichiers permettent de repérer un fichier par un nom plutôt que par son i-node dans la table de noeud d'index ; le répertoire est donc grossièrement constitué d'une table à deux colonnes contenant d'un côté le nom que l'utilisateur donne au fichier, et de l'autre, le numéro d'i-node donné par le système qui permet d'accéder à ce fichier. Cette paire est appelée un lien ;
3. les fichiers spéciaux, périphériques, tubes, sockets, ..., que nous aborderons plus loin.

Les types de fichiers

Il y a trois types de fichiers UNIX :

1. les fichiers ordinaires : tableaux linéaires d'octets identifiés par leur i-node ;
2. les répertoires : ces fichiers permettent de repérer un fichier par un nom plutôt que par son i-node dans la table de noeud d'index ; le répertoire est donc grossièrement constitué d'une table à deux colonnes contenant d'un côté le nom que l'utilisateur donne au fichier, et de l'autre, le numéro d'i-node donné par le système qui permet d'accéder à ce fichier. Cette paire est appelée un lien ;
3. les fichiers spéciaux, périphériques, tubes, sockets, ..., que nous aborderons plus loin.

Les types de fichiers

Il y a trois types de fichiers UNIX :

1. les fichiers ordinaires : tableaux linéaires d'octets identifiés par leur i-node ;
2. les répertoires : ces fichiers permettent de repérer un fichier par un nom plutôt que par son i-node dans la table de noeud d'index ; le répertoire est donc grossièrement constitué d'une table à deux colonnes contenant d'un côté le nom que l'utilisateur donne au fichier, et de l'autre, le numéro d'i-node donné par le système qui permet d'accéder à ce fichier. Cette paire est appelée un lien ;
3. les fichiers spéciaux, périphériques, tubes, sockets, ..., que nous aborderons plus loin.

Les types de fichiers

Il y a trois types de fichiers UNIX :

1. les fichiers ordinaires : tableaux linéaires d'octets identifiés par leur i-node ;
2. les répertoires : ces fichiers permettent de repérer un fichier par un nom plutôt que par son i-node dans la table de noeud d'index ; le répertoire est donc grossièrement constitué d'une table à deux colonnes contenant d'un côté le nom que l'utilisateur donne au fichier, et de l'autre, le numéro d'i-node donné par le système qui permet d'accéder à ce fichier. Cette paire est appelée un lien ;
3. les fichiers spéciaux, périphériques, tubes, sockets, ..., que nous aborderons plus loin.

Les types de fichiers

Il y a trois types de fichiers UNIX :

1. les fichiers ordinaires : tableaux linéaires d'octets identifiés par leur i-node ;
2. les répertoires : ces fichiers permettent de repérer un fichier par un nom plutôt que par son i-node dans la table de noeud d'index ; le répertoire est donc grossièrement constitué d'une table à deux colonnes contenant d'un côté le nom que l'utilisateur donne au fichier, et de l'autre, le numéro d'i-node donné par le système qui permet d'accéder à ce fichier. Cette paire est appelée un lien ;
3. les fichiers spéciaux, périphériques, tubes, sockets, ..., que nous aborderons plus loin.

Les types de fichiers

Il y a trois types de fichiers UNIX :

1. les fichiers ordinaires : tableaux linéaires d'octets identifiés par leur i-node ;
2. les répertoires : ces fichiers permettent de repérer un fichier par un nom plutôt que par son i-node dans la table de noeud d'index ; le répertoire est donc grossièrement constitué d'une table à deux colonnes contenant d'un côté le nom que l'utilisateur donne au fichier, et de l'autre, le numéro d'i-node donné par le système qui permet d'accéder à ce fichier. Cette paire est appelée un lien ;
3. les fichiers spéciaux, périphériques, tubes, sockets, ..., que nous aborderons plus loin.

Les types de fichiers

Il y a trois types de fichiers UNIX :

1. les fichiers ordinaires : tableaux linéaires d'octets identifiés par leur i-node ;
2. les répertoires : ces fichiers permettent de repérer un fichier par un nom plutôt que par son i-node dans la table de noeud d'index ; le répertoire est donc grossièrement constitué d'une table à deux colonnes contenant d'un côté le nom que l'utilisateur donne au fichier, et de l'autre, le numéro d'i-node donné par le système qui permet d'accéder à ce fichier. Cette paire est appelée un lien ;
3. les fichiers spéciaux, périphériques, tubes, sockets, ..., que nous aborderons plus loin.

Descripteurs de fichier

Nous avons vu que l'i-node d'un fichier est la structure d'identification du fichier vis-à-vis du système. Cependant lorsqu'un processus veut manipuler un fichier, il va utiliser plus simplement un entier appelé descripteur de fichier. L'association de ce descripteur à l'i-node du fichier se fait lors de l'appel à la primitive `open()` (introduite plus loin dans ce chapitre). Le descripteur devient alors le nom local du fichier dans le processus.

Descripteurs de fichier

Nous avons vu que l'i-node d'un fichier est la structure d'identification du fichier vis-à-vis du système. Cependant lorsqu'un processus veut manipuler un fichier, il va utiliser plus simplement un entier appelé descripteur de fichier. L'association de ce descripteur à l'i-node du fichier se fait lors de l'appel à la primitive `open()` (introduite plus loin dans ce chapitre). Le descripteur devient alors le nom local du fichier dans le processus.

Descripteurs de fichier

Nous avons vu que l'i-node d'un fichier est la structure d'identification du fichier vis-à-vis du système. Cependant lorsqu'un processus veut manipuler un fichier, il va utiliser plus simplement un entier appelé descripteur de fichier. L'association de ce descripteur à l'i-node du fichier se fait lors de l'appel à la primitive `open()` (introduite plus loin dans ce chapitre). Le descripteur devient alors le nom local du fichier dans le processus.

Descripteurs de fichier

Nous avons vu que l'i-node d'un fichier est la structure d'identification du fichier vis-à-vis du système. Cependant lorsqu'un processus veut manipuler un fichier, il va utiliser plus simplement un entier appelé descripteur de fichier. L'association de ce descripteur à l'i-node du fichier se fait lors de l'appel à la primitive `open()` (introduite plus loin dans ce chapitre). Le descripteur devient alors le nom local du fichier dans le processus.

Chaque processus UNIX dispose de 20 descripteurs de fichier, numérotés de 0 à 19. Par convention, les trois premiers sont toujours ouverts au début de la vie d'un processus :

- le descripteur de fichier 0 est l'entrée standard (généralement le clavier) ;
- le descripteur de fichier 1 est la sortie standard (généralement l'écran) ;
- le descripteur de fichier 2 est la sortie erreur standard (généralement l'écran) ;

Les 17 autres sont disponibles pour les fichiers et les fichiers spéciaux que le processus ouvre lui-même. Cette notion de descripteur de fichier est utilisée par l'interface d'entrée/sortie de bas niveau, principalement avec les primitives `open()`, `write()`, `read()`, `close()`.

Chaque processus UNIX dispose de 20 descripteurs de fichier, numérotés de 0 à 19. Par convention, les trois premiers sont toujours ouverts au début de la vie d'un processus :

- le descripteur de fichier 0 est l'entrée standard (généralement le clavier) ;
- le descripteur de fichier 1 est la sortie standard (généralement l'écran) ;
- le descripteur de fichier 2 est la sortie erreur standard (généralement l'écran) ;

Les 17 autres sont disponibles pour les fichiers et les fichiers spéciaux que le processus ouvre lui-même. Cette notion de descripteur de fichier est utilisée par l'interface d'entrée/sortie de bas niveau, principalement avec les primitives `open()`, `write()`, `read()`, `close()`.

Chaque processus UNIX dispose de 20 descripteurs de fichier, numérotés de 0 à 19. Par convention, les trois premiers sont toujours ouverts au début de la vie d'un processus :

- le descripteur de fichier 0 est l'entrée standard (généralement le clavier) ;
- le descripteur de fichier 1 est la sortie standard (généralement l'écran) ;
- le descripteur de fichier 2 est la sortie erreur standard (généralement l'écran) ;

Les 17 autres sont disponibles pour les fichiers et les fichiers spéciaux que le processus ouvre lui-même. Cette notion de descripteur de fichier est utilisée par l'interface d'entrée/sortie de bas niveau, principalement avec les primitives `open()`, `write()`, `read()`, `close()`.

Chaque processus UNIX dispose de 20 descripteurs de fichier, numérotés de 0 à 19. Par convention, les trois premiers sont toujours ouverts au début de la vie d'un processus :

- le descripteur de fichier 0 est l'entrée standard (généralement le clavier) ;
- le descripteur de fichier 1 est la sortie standard (généralement l'écran) ;
- le descripteur de fichier 2 est la sortie erreur standard (généralement l'écran) ;

Les 17 autres sont disponibles pour les fichiers et les fichiers spéciaux que le processus ouvre lui-même. Cette notion de descripteur de fichier est utilisée par l'interface d'entrée/sortie de bas niveau, principalement avec les primitives `open()`, `write()`, `read()`, `close()`.

Chaque processus UNIX dispose de 20 descripteurs de fichier, numérotés de 0 à 19. Par convention, les trois premiers sont toujours ouverts au début de la vie d'un processus :

- le descripteur de fichier 0 est l'entrée standard (généralement le clavier) ;
- le descripteur de fichier 1 est la sortie standard (généralement l'écran) ;
- le descripteur de fichier 2 est la sortie erreur standard (généralement l'écran) ;

Les 17 autres sont disponibles pour les fichiers et les fichiers spéciaux que le processus ouvre lui-même. Cette notion de descripteur de fichier est utilisée par l'interface d'entrée/sortie de bas niveau, principalement avec les primitives `open()`, `write()`, `read()`, `close()`.

Chaque processus UNIX dispose de 20 descripteurs de fichier, numérotés de 0 à 19. Par convention, les trois premiers sont toujours ouverts au début de la vie d'un processus :

- le descripteur de fichier 0 est l'entrée standard (généralement le clavier) ;
- le descripteur de fichier 1 est la sortie standard (généralement l'écran) ;
- le descripteur de fichier 2 est la sortie erreur standard (généralement l'écran) ;

Les 17 autres sont disponibles pour les fichiers et les fichiers spéciaux que le processus ouvre lui-même. Cette notion de descripteur de fichier est utilisée par l'interface d'entrée/sortie de bas niveau, principalement avec les primitives `open()`, `write()`, `read()`, `close()`.

Chaque processus UNIX dispose de 20 descripteurs de fichier, numérotés de 0 à 19. Par convention, les trois premiers sont toujours ouverts au début de la vie d'un processus :

- le descripteur de fichier 0 est l'entrée standard (généralement le clavier) ;
- le descripteur de fichier 1 est la sortie standard (généralement l'écran) ;
- le descripteur de fichier 2 est la sortie erreur standard (généralement l'écran) ;

Les 17 autres sont disponibles pour les fichiers et les fichiers spéciaux que le processus ouvre lui-même. Cette notion de descripteur de fichier est utilisée par l'interface d'entrée/sortie de bas niveau, principalement avec les primitives `open()`, `write()`, `read()`, `close()`.

Pointeurs vers un fichier

En revanche, lorsqu'on utilise les primitives de la bibliothèque standard d'entrés/sorties `fopen`, `fread`, `fscanf`, ..., les fichiers sont repérés par des pointeurs vers des objets de type `FILE` (type défini dans le fichier `<stdio.h>`). Il y a trois pointeurs prédéfinis :

1. `stdin` qui pointe vers le tampon (buffer) de l'entrée standard (généralement le clavier) ;
2. `stdout` qui pointe vers le tampon de la sortie standard (généralement l'écran) ;
3. `stderr` qui pointe vers le tampon de la sortie d'erreur standard (généralement l'écran).

Pointeurs vers un fichier

En revanche, lorsqu'on utilise les primitives de la bibliothèque standard d'entrés/sorties `fopen`, `fread`, `fscanf`, ..., les fichiers sont repérés par des pointeurs vers des objets de type `FILE` (type défini dans le fichier `<stdio.h>`). Il y a trois pointeurs prédéfinis :

1. `stdin` qui pointe vers le tampon (buffer) de l'entrée standard (généralement le clavier) ;
2. `stdout` qui pointe vers le tampon de la sortie standard (généralement l'écran) ;
3. `stderr` qui pointe vers le tampon de la sortie d'erreur standard (généralement l'écran).

Pointeurs vers un fichier

En revanche, lorsqu'on utilise les primitives de la bibliothèque standard d'entrés/sorties `fopen`, `fread`, `fscanf`, ..., les fichiers sont repérés par des pointeurs vers des objets de type `FILE` (type défini dans le fichier `<stdio.h>`). Il y a trois pointeurs prédéfinis :

1. `stdin` qui pointe vers le tampon (buffer) de l'entrée standard (généralement le clavier) ;
2. `stdout` qui pointe vers le tampon de la sortie standard (généralement l'écran) ;
3. `stderr` qui pointe vers le tampon de la sortie d'erreur standard (généralement l'écran).

Pointeurs vers un fichier

En revanche, lorsqu'on utilise les primitives de la bibliothèque standard d'entrés/sorties `fopen`, `fread`, `fscanf`, ..., les fichiers sont repérés par des pointeurs vers des objets de type `FILE` (type défini dans le fichier `<stdio.h>`). Il y a trois pointeurs prédéfinis :

1. `stdin` qui pointe vers le tampon (buffer) de l'entrée standard (généralement le clavier) ;
2. `stdout` qui pointe vers le tampon de la sortie standard (généralement l'écran) ;
3. `stderr` qui pointe vers le tampon de la sortie d'erreur standard (généralement l'écran).

Pointeurs vers un fichier

En revanche, lorsqu'on utilise les primitives de la bibliothèque standard d'entrés/sorties `fopen`, `fread`, `fscanf`, ..., les fichiers sont repérés par des pointeurs vers des objets de type `FILE` (type défini dans le fichier `<stdio.h>`). Il y a trois pointeurs prédéfinis :

1. `stdin` qui pointe vers le tampon (buffer) de l'entrée standard (généralement le clavier) ;
2. `stdout` qui pointe vers le tampon de la sortie standard (généralement l'écran) ;
3. `stderr` qui pointe vers le tampon de la sortie d'erreur standard (généralement l'écran).

creat()

```
int creat(const char *path, mode_t perm)
/* crée un fichier */
/* path = nom du fichier */
/* perm = droits d'accès */
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Cette primitive réalise la création d'un fichier, dont le nom est donné dans le paramètre `path`. L'entier `perm` définit les droits d'accès. Si le fichier n'existait pas, il est ouvert en écriture. Ce n'est pas une erreur de créer un fichier qui existait déjà. Reportez vous au manuel en ligne pour interpréter correctement les droits d'accès (tapez " `man creat` " dans un terminal sous linux). Pour créer un fichier de nom " `essai_creat` " avec les autorisations lecture et écriture pour le propriétaire et le groupe, on écrira :

```
if ((fd = creat("essai_creat",0660)) == -1)
perror("Erreur creat()");
```

creat()

```
int creat(const char *path, mode_t perm)
/* crée un fichier */
/* path = nom du fichier */
/* perm = droits d'accès */
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Cette primitive réalise la création d'un fichier, dont le nom est donné dans le paramètre `path`. L'entier `perm` définit les droits d'accès. Si le fichier n'existait pas, il est ouvert en écriture. Ce n'est pas une erreur de créer un fichier qui existait déjà. Reportez vous au manuel en ligne pour interpréter correctement les droits d'accès (tapez " `man creat` " dans un terminal sous linux). Pour créer un fichier de nom " `essai_creat` " avec les autorisations lecture et écriture pour le propriétaire et le groupe, on écrira :

```
if ((fd = creat("essai_creat",0660)) == -1)
perror("Erreur creat()");
```

creat()

```
int creat(const char *path, mode_t perm)
/* crée un fichier */
/* path = nom du fichier */
/* perm = droits d'accès */
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Cette primitive réalise la création d'un fichier, dont le nom est donné dans le paramètre `path`. L'entier `perm` définit les droits d'accès. Si le fichier n'existait pas, il est ouvert en écriture. Ce n'est pas une erreur de créer un fichier qui existait déjà. Reportez vous au manuel en ligne pour interpréter correctement les droits d'accès (tapez " `man creat` " dans un terminal sous linux). Pour créer un fichier de nom " `essai_creat` " avec les autorisations lecture et écriture pour le propriétaire et le groupe, on écrira :

```
if ((fd = creat("essai_creat",0660)) == -1)
perror("Erreur creat()");
```

creat()

```
int creat(const char *path, mode_t perm)
/* crée un fichier */
/* path = nom du fichier */
/* perm = droits d'accès */
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Cette primitive réalise la création d'un fichier, dont le nom est donné dans le paramètre `path`. L'entier `perm` définit les droits d'accès. Si le fichier n'existait pas, il est ouvert en écriture. Ce n'est pas une erreur de créer un fichier qui existait déjà. Reportez vous au manuel en ligne pour interpréter correctement les droits d'accès (tapez " `man creat` " dans un terminal sous linux). Pour créer un fichier de nom " `essai_creat` " avec les autorisations lecture et écriture pour le propriétaire et le groupe, on écrira :

```
if ((fd = creat("essai_creat",0660)) == -1)
perror("Erreur creat()");
```

creat()

```
int creat(const char *path, mode_t perm)
/* crée un fichier */
/* path = nom du fichier */
/* perm = droits d'accès */
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Cette primitive réalise la création d'un fichier, dont le nom est donné dans le paramètre `path`. L'entier `perm` définit les droits d'accès. Si le fichier n'existait pas, il est ouvert en écriture. Ce n'est pas une erreur de créer un fichier qui existait déjà. Reportez vous au manuel en ligne pour interpréter correctement les droits d'accès (tapez " `man creat` " dans un terminal sous linux). Pour créer un fichier de nom " `essai_creat` " avec les autorisations lecture et écriture pour le propriétaire et le groupe, on écrira :

```
if ((fd = creat("essai_creat",0660)) == -1)
perror("Erreur creat()");
```

creat()

```
int creat(const char *path, mode_t perm)
/* crée un fichier */
/* path = nom du fichier */
/* perm = droits d'accès */
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Cette primitive réalise la création d'un fichier, dont le nom est donné dans le paramètre `path`. L'entier `perm` définit les droits d'accès. Si le fichier n'existait pas, il est ouvert en écriture. Ce n'est pas une erreur de créer un fichier qui existait déjà. Reportez vous au manuel en ligne pour interpréter correctement les droits d'accès (tapez " `man creat` " dans un terminal sous linux). Pour créer un fichier de nom " `essai_creat` " avec les autorisations lecture et écriture pour le propriétaire et le groupe, on écrira :

```
if ((fd = creat("essai_creat",0660)) == -1)
perror("Erreur creat()");
```

creat()

```
int creat(const char *path, mode_t perm)
/* crée un fichier */
/* path = nom du fichier */
/* perm = droits d'accès */
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Cette primitive réalise la création d'un fichier, dont le nom est donné dans le paramètre `path`. L'entier `perm` définit les droits d'accès. Si le fichier n'existait pas, il est ouvert en écriture. Ce n'est pas une erreur de créer un fichier qui existait déjà. Reportez vous au manuel en ligne pour interpréter correctement les droits d'accès (tapez " `man creat` " dans un terminal sous linux). Pour créer un fichier de nom " `essai_creat` " avec les autorisations lecture et écriture pour le propriétaire et le groupe, on écrira :

```
if ((fd = creat("essai_creat",0660)) == -1)
perror("Erreur creat()");
```

creat()

```
int creat(const char *path, mode_t perm)
/* crée un fichier */
/* path = nom du fichier */
/* perm = droits d'accès */
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.
Cette primitive réalise la création d'un fichier, dont le nom est donné dans le paramètre `path`. L'entier `perm` définit les droits d'accès. Si le fichier n'existait pas, il est ouvert en écriture. Ce n'est pas une erreur de créer un fichier qui existait déjà. Reportez vous au manuel en ligne pour interpréter correctement les droits d'accès (tapez " `man creat` " dans un terminal sous linux). Pour créer un fichier de nom " `essai_creat` " avec les autorisations lecture et écriture pour le propriétaire et le groupe, on écrira :

```
if ((fd = creat("essai_creat",0660)) == -1)
perror("Erreur creat()");
```

creat()

```
int creat(const char *path, mode_t perm)
/* crée un fichier */
/* path = nom du fichier */
/* perm = droits d'accès */
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Cette primitive réalise la création d'un fichier, dont le nom est donné dans le paramètre `path`. L'entier `perm` définit les droits d'accès. Si le fichier n'existait pas, il est ouvert en écriture. Ce n'est pas une erreur de créer un fichier qui existait déjà. Reportez vous au manuel en ligne pour interpréter correctement les droits d'accès (tapez " `man creat` " dans un terminal sous linux). Pour créer un fichier de nom " `essai_creat` " avec les autorisations lecture et écriture pour le propriétaire et le groupe, on écrira :

```
if ((fd = creat("essai_creat",0660)) == -1)
perror("Erreur creat()");
```

Les permissions sont soit données sous leur forme hexadécimale usuelle, soit en utilisant la syntaxe suivante (plusieurs permissions doivent être séparées par un “ | ”) :

- `S_IUSR, S_IRGRP, S_IROTH` : 00400, 00040, 00004 (lecture pour utilisateur, groupe et autres) ;
- `S_IWUSR, S_IWGRP, S_IWOTH` : 00200, 00020, 00002 (écriture pour utilisateur, groupe et autres) ;
- `S_IXUSR, S_IXGRP, S_IXOTH` : 00100, 00010, 00001 (exécution pour utilisateur, groupe et autres) ;
- `S_IRWXU, S_IRWXG, S_IRWXO` : 00700, 00070, 00007 (R, W, X pour utilisateur, groupe et autres).

Les permissions sont soit données sous leur forme hexadécimale usuelle, soit en utilisant la syntaxe suivante (plusieurs permissions doivent être séparées par un “ | ”) :

- `S_IUSR, S_IRGRP, S_IROTH` : 00400, 00040, 00004 (lecture pour utilisateur, groupe et autres) ;
- `S_IWUSR, S_IWGRP, S_IWOTH` : 00200, 00020, 00002 (écriture pour utilisateur, groupe et autres) ;
- `S_IXUSR, S_IXGRP, S_IXOTH` : 00100, 00010, 00001 (exécution pour utilisateur, groupe et autres) ;
- `S_IRWXU, S_IRWXG, S_IRWXO` : 00700, 00070, 00007 (R, W, X pour utilisateur, groupe et autres).

Les permissions sont soit données sous leur forme hexadécimale usuelle, soit en utilisant la syntaxe suivante (plusieurs permissions doivent être séparées par un " | ") :

- `S_IUSR, S_IRGRP, S_IROTH` : 00400, 00040, 00004 (lecture pour utilisateur, groupe et autres) ;
- `S_IWUSR, S_IWGRP, S_IWOTH` : 00200, 00020, 00002 (écriture pour utilisateur, groupe et autres) ;
- `S_IXUSR, S_IXGRP, S_IXOTH` : 00100, 00010, 00001 (exécution pour utilisateur, groupe et autres) ;
- `S_IRWXU, S_IRWXG, S_IRWXO` : 00700, 00070, 00007 (R, W, X pour utilisateur, groupe et autres).

Les permissions sont soit données sous leur forme hexadécimale usuelle, soit en utilisant la syntaxe suivante (plusieurs permissions doivent être séparées par un " | ") :

- `S_IUSR, S_IRGRP, S_IROTH` : 00400, 00040, 00004 (lecture pour utilisateur, groupe et autres) ;
- `S_IWUSR, S_IWGRP, S_IWOTH` : 00200, 00020, 00002 (écriture pour utilisateur, groupe et autres) ;
- `S_IXUSR, S_IXGRP, S_IXOTH` : 00100, 00010, 00001 (exécution pour utilisateur, groupe et autres) ;
- `S_IRWXU, S_IRWXG, S_IRWXO` : 00700, 00070, 00007 (R, W, X pour utilisateur, groupe et autres).

Les permissions sont soit données sous leur forme hexadécimale usuelle, soit en utilisant la syntaxe suivante (plusieurs permissions doivent être séparées par un " | ") :

- `S_IUSR, S_IRGRP, S_IROTH` : 00400, 00040, 00004 (lecture pour utilisateur, groupe et autres) ;
- `S_IWUSR, S_IWGRP, S_IWOTH` : 00200, 00020, 00002 (écriture pour utilisateur, groupe et autres) ;
- `S_IXUSR, S_IXGRP, S_IXOTH` : 00100, 00010, 00001 (exécution pour utilisateur, groupe et autres) ;
- `S_IRWXU, S_IRWXG, S_IRWXO` : 00700, 00070, 00007 (R, W, X pour utilisateur, groupe et autres).

Primitive `open()`

La fonction `open()` a deux profils : avec 2 ou 3 paramètres.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags,
mode_t mode);
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Primitive `open()`

La fonction `open()` a deux profils : avec 2 ou 3 paramètres.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags,
mode_t mode);
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Primitive `open()`

La fonction `open()` a deux profils : avec 2 ou 3 paramètres.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags,
mode_t mode);
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Cette primitive permet d'ouvrir (ou de créer) le fichier de nom `pathname`. L'entier `flags` détermine le mode d'ouverture du fichier. Le paramètre optionnel `mode` n'est utilisé que lorsque `open()` réalise la création du fichier. Dans ce cas, il indique pour celui-ci les droits d'accès donnés à l'utilisateur, à son groupe et au reste du monde. Le paramètre `flags` peut prendre une ou plusieurs des constantes symboliques (qui sont dans ce cas séparées par des "|"), définies dans le fichier d'inclusion `fcntl.h`.

Cette primitive permet d'ouvrir (ou de créer) le fichier de nom `pathname`. L'entier `flags` détermine le mode d'ouverture du fichier. Le paramètre optionnel `mode` n'est utilisé que lorsque `open()` réalise la création du fichier. Dans ce cas, il indique pour celui-ci les droits d'accès donnés à l'utilisateur, à son groupe et au reste du monde. Le paramètre `flags` peut prendre une ou plusieurs des constantes symboliques (qui sont dans ce cas séparées par des "|"), définies dans le fichier d'inclusion `fcntl.h`.

Cette primitive permet d'ouvrir (ou de créer) le fichier de nom `pathname`. L'entier `flags` détermine le mode d'ouverture du fichier. Le paramètre optionnel `mode` n'est utilisé que lorsque `open()` réalise la création du fichier. Dans ce cas, il indique pour celui-ci les droits d'accès donnés à l'utilisateur, à son groupe et au reste du monde. Le paramètre `flags` peut prendre une ou plusieurs des constantes symboliques (qui sont dans ce cas séparées par des "|"), définies dans le fichier d'inclusion `fcntl.h`.

Cette primitive permet d'ouvrir (ou de créer) le fichier de nom `pathname`. L'entier `flags` détermine le mode d'ouverture du fichier. Le paramètre optionnel `mode` n'est utilisé que lorsque `open()` réalise la création du fichier. Dans ce cas, il indique pour celui-ci les droits d'accès donnés à l'utilisateur, à son groupe et au reste du monde. Le paramètre `flags` peut prendre une ou plusieurs des constantes symboliques (qui sont dans ce cas séparées par des "|"), définies dans le fichier d'inclusion `fcntl.h`.

Cette primitive permet d'ouvrir (ou de créer) le fichier de nom `pathname`. L'entier `flags` détermine le mode d'ouverture du fichier. Le paramètre optionnel `mode` n'est utilisé que lorsque `open()` réalise la création du fichier. Dans ce cas, il indique pour celui-ci les droits d'accès donnés à l'utilisateur, à son groupe et au reste du monde. Le paramètre `flags` peut prendre une ou plusieurs des constantes symboliques (qui sont dans ce cas séparées par des "|"), définies dans le fichier d'inclusion `fcntl.h`.

Les valeurs possibles des `flags` sont les suivantes :

- `O_RDONLY` ouverture en lecture seule ;
- `O_WRONLY` ouverture en écriture seule ;
- `O_RDWR` ouverture en lecture et écriture ;
- `O_APPEND` ouverture en écriture à la fin du fichier (ajout en fin de fichier) ;
- `O_CREAT` création du fichier s'il n'existe pas ;
- `O_TRUNC` troncature à la longueur zéro (vider) si le fichier existe déjà ;
- `O_NONBLOCK` ouverture non bloquante : s'il n'y a rien à lire, on n'attend pas.

Les valeurs possibles des `flags` sont les suivantes :

- `O_RDONLY` ouverture en lecture seule ;
- `O_WRONLY` ouverture en écriture seule ;
- `O_RDWR` ouverture en lecture et écriture ;
- `O_APPEND` ouverture en écriture à la fin du fichier (ajout en fin de fichier) ;
- `O_CREAT` création du fichier s'il n'existe pas ;
- `O_TRUNC` troncature à la longueur zéro (vider) si le fichier existe déjà ;
- `O_NONBLOCK` ouverture non bloquante : s'il n'y a rien à lire, on n'attend pas.

Les valeurs possibles des `flags` sont les suivantes :

- `O_RDONLY` ouverture en lecture seule ;
- `O_WRONLY` ouverture en écriture seule ;
- `O_RDWR` ouverture en lecture et écriture ;
- `O_APPEND` ouverture en écriture à la fin du fichier (ajout en fin de fichier) ;
- `O_CREAT` création du fichier s'il n'existe pas ;
- `O_TRUNC` troncature à la longueur zéro (vider) si le fichier existe déjà ;
- `O_NONBLOCK` ouverture non bloquante : s'il n'y a rien à lire, on n'attend pas.

Les valeurs possibles des `flags` sont les suivantes :

- `O_RDONLY` ouverture en lecture seule ;
- `O_WRONLY` ouverture en écriture seule ;
- `O_RDWR` ouverture en lecture et écriture ;
- `O_APPEND` ouverture en écriture à la fin du fichier (ajout en fin de fichier) ;
- `O_CREAT` création du fichier s'il n'existe pas ;
- `O_TRUNC` troncature à la longueur zéro (vider) si le fichier existe déjà ;
- `O_NONBLOCK` ouverture non bloquante : s'il n'y a rien à lire, on n'attend pas.

Les valeurs possibles des `flags` sont les suivantes :

- `O_RDONLY` ouverture en lecture seule ;
- `O_WRONLY` ouverture en écriture seule ;
- `O_RDWR` ouverture en lecture et écriture ;
- `O_APPEND` ouverture en écriture à la fin du fichier (ajout en fin de fichier) ;
- `O_CREAT` création du fichier s'il n'existe pas ;
- `O_TRUNC` troncature à la longueur zéro (vider) si le fichier existe déjà ;
- `O_NONBLOCK` ouverture non bloquante : s'il n'y a rien à lire, on n'attend pas.

Les valeurs possibles des `flags` sont les suivantes :

- `O_RDONLY` ouverture en lecture seule ;
- `O_WRONLY` ouverture en écriture seule ;
- `O_RDWR` ouverture en lecture et écriture ;
- `O_APPEND` ouverture en écriture à la fin du fichier (ajout en fin de fichier) ;
- `O_CREAT` création du fichier s'il n'existe pas ;
- `O_TRUNC` troncature à la longueur zéro (vider) si le fichier existe déjà ;
- `O_NONBLOCK` ouverture non bloquante : s'il n'y a rien à lire, on n'attend pas.

Les valeurs possibles des `flags` sont les suivantes :

- `O_RDONLY` ouverture en lecture seule ;
- `O_WRONLY` ouverture en écriture seule ;
- `O_RDWR` ouverture en lecture et écriture ;
- `O_APPEND` ouverture en écriture à la fin du fichier (ajout en fin de fichier) ;
- `O_CREAT` création du fichier s'il n'existe pas ;
- `O_TRUNC` troncature à la longueur zéro (vider) si le fichier existe déjà ;
- `O_NONBLOCK` ouverture non bloquante : s'il n'y a rien à lire, on n'attend pas.

Les valeurs possibles des `flags` sont les suivantes :

- `O_RDONLY` ouverture en lecture seule ;
- `O_WRONLY` ouverture en écriture seule ;
- `O_RDWR` ouverture en lecture et écriture ;
- `O_APPEND` ouverture en écriture à la fin du fichier (ajout en fin de fichier) ;
- `O_CREAT` création du fichier s'il n'existe pas ;
- `O_TRUNC` troncature à la longueur zéro (vider) si le fichier existe déjà ;
- `O_NONBLOCK` ouverture non bloquante : s'il n'y a rien à lire, on n'attend pas.

Si `O_CREAT` est positionné, alors il faut préciser les permissions en utilisant les mêmes constantes symboliques de `mode` que pour `creat` ou bien les notations hexadécimales usuelles.

Exemple :

Pour effectuer la création et l'ouverture du fichier "`essai_open`" en écriture avec les autorisations de lecture et écriture pour le propriétaire et le groupe, il faut écrire :

```
if ((fd = open("essai_open" , O_WRONLY |
O_CREAT, 0660)) == -1)
perror("Erreur open()");
```

Remarque : l'inclusion du fichier `<sys/types.h>` est nécessaire, car des types utilisés dans `<fcntl.h>` y sont définis.

Exemple :
Pour effectuer la création et l'ouverture du fichier "`essai_open`" en écriture avec les autorisations de lecture et écriture pour le propriétaire et le groupe, il faut écrire :

```
if ((fd = open("essai_open" , O_WRONLY |
O_CREAT, 0660)) == -1)
perror("Erreur open()");
```

Remarque : l'inclusion du fichier `<sys/types.h>` est nécessaire, car des types utilisés dans `<fcntl.h>` y sont définis.

Exemple :
Pour effectuer la création et l'ouverture du fichier "`essai_open`" en écriture avec les autorisations de lecture et écriture pour le propriétaire et le groupe, il faut écrire :

```
if ((fd = open("essai_open" , O_WRONLY |
O_CREAT, 0660)) == -1)
perror("Erreur open()");
```

Remarque : l'inclusion du fichier `<sys/types.h>` est nécessaire, car des types utilisés dans `<fcntl.h>` y sont définis.

Exemple :
Pour effectuer la création et l'ouverture du fichier "`essai_open`" en écriture avec les autorisations de lecture et écriture pour le propriétaire et le groupe, il faut écrire :

```
if ((fd = open("essai_open" , O_WRONLY |
O_CREAT, 0660)) == -1)
perror("Erreur open()");
```

Remarque : l'inclusion du fichier `<sys/types.h>` est nécessaire, car des types utilisés dans `<fcntl.h>` y sont définis.

Fonction `fdopen()`

Cette fonction permet de faire la jonction entre les appels système de manipulation de fichiers de la librairie standard C, qui utilisent des pointeurs vers des objets de type `FILE` (`fclose()`, `fflush()`, `fprintf()`, `fscanf()`), et les primitives de bas niveau `open()`, `write()`, `read()` qui utilisent des descripteurs de fichiers de type `int`.

Fonction `fdopen()`

Cette fonction permet de faire la jonction entre les appels système de manipulation de fichiers de la librairie standard C, qui utilisent des pointeurs vers des objets de type `FILE` (`fclose()`, `fflush()`, `fprintf()`, `fscanf()`), et les primitives de bas niveau `open()`, `write()`, `read()` qui utilisent des descripteurs de fichiers de type `int`.

Son profil est :

```
#include <stdio.h>
FILE* fdopen(int fd, const char *mode)
/* convertit un descripteur de fichier en */
/* un pointeur sur un fichier */
/* fd : descripteur concerné par la conversion
*/
/* mode : mode d'ouverture désiré */
```

Valeur retournée : un pointeur sur le fichier associé au descripteur `fd`, ou la constante `NULL` (prédéfinie dans `<stdio.h>`) en cas d'erreur.

Son profil est :

```
#include <stdio.h>
FILE* fdopen(int fd, const char *mode)
/* convertit un descripteur de fichier en */
/* un pointeur sur un fichier */
/* fd : descripteur concerné par la conversion
*/
/* mode : mode d'ouverture désiré */
```

Valeur retournée : un pointeur sur le fichier associé au descripteur `fd`, ou la constante `NULL` (prédéfinie dans `<stdio.h>`) en cas d'erreur.

Son profil est :

```
#include <stdio.h>
FILE* fdopen(int fd, const char *mode)
/* convertit un descripteur de fichier en */
/* un pointeur sur un fichier */
/* fd : descripteur concerné par la conversion
*/
/* mode : mode d'ouverture désiré */
```

Valeur retournée : un pointeur sur le fichier associé au descripteur `fd`, ou la constante `NULL` (prédéfinie dans `<stdio.h>`) en cas d'erreur.

Remarque :

Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre `mode` choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`. Ce paramètre peut prendre les valeurs suivantes :

- `"r"` : le fichier est ouvert en lecture ;
- `"w"` : fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro ;
- `"a"` : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Remarque :
Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre `mode` choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`. Ce paramètre peut prendre les valeurs suivantes :

- `"r"` : le fichier est ouvert en lecture ;
- `"w"` : fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro ;
- `"a"` : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Remarque :
Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre `mode` choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`. Ce paramètre peut prendre les valeurs suivantes :

- `"r"` : le fichier est ouvert en lecture ;
- `"w"` : fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro ;
- `"a"` : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Remarque :
Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre `mode` choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`. Ce paramètre peut prendre les valeurs suivantes :

- `"r"` : le fichier est ouvert en lecture ;
- `"w"` : fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro ;
- `"a"` : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Remarque :
Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre `mode` choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`. Ce paramètre peut prendre les valeurs suivantes :

- `"r"` : le fichier est ouvert en lecture ;
- `"w"` : fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro ;
- `"a"` : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Remarque :
Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre `mode` choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`. Ce paramètre peut prendre les valeurs suivantes :

- `"r"` : le fichier est ouvert en lecture ;
- `"w"` : fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro ;
- `"a"` : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Remarque :
Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre `mode` choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`. Ce paramètre peut prendre les valeurs suivantes :

- `"r"` : le fichier est ouvert en lecture ;
- `"w"` : fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro ;
- `"a"` : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Remarque :
Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre `mode` choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`. Ce paramètre peut prendre les valeurs suivantes :

- `"r"` : le fichier est ouvert en lecture ;
- `"w"` : fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro ;
- `"a"` : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Remarque :
Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre `mode` choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`. Ce paramètre peut prendre les valeurs suivantes :

- `"r"` : le fichier est ouvert en lecture ;
- `"w"` : fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro ;
- `"a"` : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Remarque :
Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre `mode` choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`. Ce paramètre peut prendre les valeurs suivantes :

- `"r"` : le fichier est ouvert en lecture ;
- `"w"` : fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro ;
- `"a"` : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Remarque :
Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre `mode` choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`. Ce paramètre peut prendre les valeurs suivantes :

- `"r"` : le fichier est ouvert en lecture ;
- `"w"` : fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro ;
- `"a"` : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Exemple :

```
inf fd; FILE* fp;
/* ouverture préalable par open() par exemple
en lecture */
if ((fd = open("mon_fichier", O_RDONLY, 0666))
== -1) {
perror("Erreur open()");
exit(-1);
}
/* association de fp (de type FILE*) à fd (de
type int) */
if ((fp = fdopen(fd, "r")) == NULL) {
perror("Erreur fdopen()");
exit(-1);
}
```

Exemple :

```
inf fd; FILE* fp;
/* ouverture préalable par open() par exemple
en lecture */
if ((fd = open("mon_fichier", O_RDONLY, 0666))
== -1) {
perror("Erreur open()");
exit(-1);
}
/* association de fp (de type FILE*) à fd (de
type int) */
if ((fp = fdopen(fd, "r")) == NULL) {
perror("Erreur fdopen()");
exit(-1);
}
```

Exemple :

```
inf fd; FILE* fp;
/* ouverture préalable par open() par exemple
en lecture */
if ((fd = open("mon_fichier", O_RDONLY, 0666))
== -1) {
perror("Erreur open()");
exit(-1);
}
/* association de fp (de type FILE*) à fd (de
type int) */
if ((fp = fdopen(fd, "r")) == NULL) {
perror("Erreur fdopen()");
exit(-1);
}
```

Primitive `close()`

Son profil :

```
#include <unistd.h>
int close(int fd) /* fermeture de fichier */
/* fd est le descripteur de fichier */
```

Valeur retournée : 0 ou -1 en cas d'échec.

Cette primitive libère le descripteur de fichier pour une éventuelle réutilisation.

Primitive `close()`

Son profil :

```
#include <unistd.h>
int close(int fd) /* fermeture de fichier */
/* fd est le descripteur de fichier */
```

Valeur retournée : 0 ou -1 en cas d'échec.

Cette primitive libère le descripteur de fichier pour une éventuelle réutilisation.

Primitive `close()`

Son profil :

```
#include <unistd.h>
int close(int fd) /* fermeture de fichier */
/* fd est le descripteur de fichier */
```

Valeur retournée : 0 ou -1 en cas d'échec.

Cette primitive libère le descripteur de fichier pour une éventuelle réutilisation.

Primitive `close()`

Son profil :

```
#include <unistd.h>
int close(int fd) /* fermeture de fichier */
/* fd est le descripteur de fichier */
```

Valeur retournée : 0 ou -1 en cas d'échec.

Cette primitive libère le descripteur de fichier pour une éventuelle réutilisation.

Primitive `write()`

```
#include <unistd.h>
size_t write(int fd, const void *buf, size_t
nbytes)
/* écriture dans un fichier */
/* fd : descripteur de fichier */
/* buf : adresse du tampon */
/* nbytes : nombre d'octets à écrire */
```

Valeur retournée : nombre d'octets écrits ou -1 en cas d'erreur.
Cette primitive écrit dans le fichier ouvert représenté par `fd` les `nbytes` octets sur lesquels pointe `buf`. Il faut noter que l'écriture ne se fait pas directement dans le fichier, mais passe par un tampon du noyau. Remarque : ce n'est pas une erreur que d'écrire moins d'octets que souhaités.

Primitive `write()`

```
#include <unistd.h>
size_t write(int fd, const void *buf, size_t
nbytes)
/* écriture dans un fichier */
/* fd : descripteur de fichier */
/* buf : adresse du tampon */
/* nbytes : nombre d'octets à écrire */
```

Valeur retournée : nombre d'octets écrits ou -1 en cas d'erreur.

Cette primitive écrit dans le fichier ouvert représenté par `fd` les `nbytes` octets sur lesquels pointe `buf`. Il faut noter que l'écriture ne se fait pas directement dans le fichier, mais passe par un tampon du noyau. Remarque : ce n'est pas une erreur que d'écrire moins d'octets que souhaités.

Primitive `write()`

```
#include <unistd.h>
size_t write(int fd, const void *buf, size_t
nbytes)
/* écriture dans un fichier */
/* fd : descripteur de fichier */
/* buf : adresse du tampon */
/* nbytes : nombre d'octets à écrire */
```

Valeur retournée : nombre d'octets écrits ou -1 en cas d'erreur.

Cette primitive écrit dans le fichier ouvert représenté par `fd` les `nbytes` octets sur lesquels pointe `buf`. Il faut noter que l'écriture ne se fait pas directement dans le fichier, mais passe par un tampon du noyau. Remarque : ce n'est pas une erreur que d'écrire moins d'octets que souhaités.

Primitive `write()`

```
#include <unistd.h>
size_t write(int fd, const void *buf, size_t
nbytes)
/* écriture dans un fichier */
/* fd : descripteur de fichier */
/* buf : adresse du tampon */
/* nbytes : nombre d'octets à écrire */
```

Valeur retournée : nombre d'octets écrits ou -1 en cas d'erreur.
Cette primitive écrit dans le fichier ouvert représenté par `fd` les `nbytes` octets sur lesquels pointe `buf`. Il faut noter que l'écriture ne se fait pas directement dans le fichier, mais passe par un tampon du noyau. Remarque : ce n'est pas une erreur que d'écrire moins d'octets que souhaités.

Primitive `write()`

```
#include <unistd.h>
size_t write(int fd, const void *buf, size_t
nbytes)
/* écriture dans un fichier */
/* fd : descripteur de fichier */
/* buf : adresse du tampon */
/* nbytes : nombre d'octets à écrire */
```

Valeur retournée : nombre d'octets écrits ou -1 en cas d'erreur.
Cette primitive écrit dans le fichier ouvert représenté par `fd` les `nbytes` octets sur lesquels pointe `buf`. Il faut noter que l'écriture ne se fait pas directement dans le fichier, mais passe par un tampon du noyau. Remarque : ce n'est pas une erreur que d'écrire moins d'octets que souhaités.

Primitive `read()`

Profil :

```
#include <unistd.h>
size_t read(int fd, void *buf, size_t nbytes)
/* lecture dans un fichier */
/* fd : descripteur de fichier */
/* buf : adresse du tampon */
/* nbytes : nombre d'octets à lire */
```

Valeur retournée : nombre d'octets lus, 0 si la fin de fichier a été atteinte, ou -1 en cas d'erreur.

La primitive essaie de lire les `nbytes` octets (caractères) dans le fichier ouvert représenté par `fd`, et les place dans le tampon sur lequel pointe `buf` (l'allocation de `buf` est à la charge de l'utilisateur et doit être de taille convenable).

Primitive `read()`

Profil :

```
#include <unistd.h>
size_t read(int fd, void *buf, size_t nbytes)
/* lecture dans un fichier */
/* fd : descripteur de fichier */
/* buf : adresse du tampon */
/* nbytes : nombre d'octets à lire */
```

Valeur retournée : nombre d'octets lus, 0 si la fin de fichier a été atteinte, ou -1 en cas d'erreur.

La primitive essaie de lire les `nbytes` octets (caractères) dans le fichier ouvert représenté par `fd`, et les place dans le tampon sur lequel pointe `buf` (l'allocation de `buf` est à la charge de l'utilisateur et doit être de taille convenable).

Primitive `read()`

Profil :

```
#include <unistd.h>
size_t read(int fd, void *buf, size_t nbytes)
/* lecture dans un fichier */
/* fd : descripteur de fichier */
/* buf : adresse du tampon */
/* nbytes : nombre d'octets à lire */
```

Valeur retournée : nombre d'octets lus, 0 si la fin de fichier a été atteinte, ou -1 en cas d'erreur.

La primitive essaie de lire les `nbytes` octets (caractères) dans le fichier ouvert représenté par `fd`, et les place dans le tampon sur lequel pointe `buf` (l'allocation de `buf` est à la charge de l'utilisateur et doit être de taille convenable).

Primitive `read()`

Profil :

```
#include <unistd.h>
size_t read(int fd, void *buf, size_t nbytes)
/* lecture dans un fichier */
/* fd : descripteur de fichier */
/* buf : adresse du tampon */
/* nbytes : nombre d'octets à lire */
```

Valeur retournée : nombre d'octets lus, 0 si la fin de fichier a été atteinte, ou -1 en cas d'erreur.

La primitive essaie de lire les `nbytes` octets (caractères) dans le fichier ouvert représenté par `fd`, et les place dans le tampon sur lequel pointe `buf` (l'allocation de `buf` est à la charge de l'utilisateur et doit être de taille convenable).

Primitive dup ()

```
#include <unistd.h>
int dup(int fd) /* fd est le descripteur donné
*/
```

Cette primitive duplique un descripteur de fichier existant et fournit donc un descripteur ayant exactement les mêmes caractéristiques que celui passé en argument. Il correspond donc au même fichier. Cette primitive garantit que la valeur retournée est la plus petite possible parmi les valeurs de descripteurs disponibles.
Valeur retournée : nouveau descripteur de fichier ou -1 en cas d'erreur.

Primitive `dup()`

```
#include <unistd.h>
int dup(int fd) /* fd est le descripteur donné
*/
```

Cette primitive duplique un descripteur de fichier existant et fournit donc un descripteur ayant exactement les mêmes caractéristiques que celui passé en argument. Il correspond donc au même fichier. Cette primitive garantit que la valeur retournée est la plus petite possible parmi les valeurs de descripteurs disponibles.
Valeur retournée : nouveau descripteur de fichier ou -1 en cas d'erreur.

Primitive `dup()`

```
#include <unistd.h>
int dup(int fd) /* fd est le descripteur donné
*/
```

Cette primitive duplique un descripteur de fichier existant et fournit donc un descripteur ayant exactement les mêmes caractéristiques que celui passé en argument. Il correspond donc au même fichier. Cette primitive garantit que la valeur retournée est la plus petite possible parmi les valeurs de descripteurs disponibles.
Valeur retournée : nouveau descripteur de fichier ou -1 en cas d'erreur.

Primitive `dup()`

```
#include <unistd.h>
int dup(int fd) /* fd est le descripteur donné
*/
```

Cette primitive duplique un descripteur de fichier existant et fournit donc un descripteur ayant exactement les mêmes caractéristiques que celui passé en argument. Il correspond donc au même fichier. Cette primitive garantit que la valeur retournée est la plus petite possible parmi les valeurs de descripteurs disponibles.

Valeur retournée : nouveau descripteur de fichier ou -1 en cas d'erreur.

Primitive `dup()`

```
#include <unistd.h>
int dup(int fd) /* fd est le descripteur donné
*/
```

Cette primitive duplique un descripteur de fichier existant et fournit donc un descripteur ayant exactement les mêmes caractéristiques que celui passé en argument. Il correspond donc au même fichier. Cette primitive garantit que la valeur retournée est la plus petite possible parmi les valeurs de descripteurs disponibles.
Valeur retournée : nouveau descripteur de fichier ou -1 en cas d'erreur.

Primitive `dup2()`

```
#include <unistd.h>
int dup2(int fd1,int fd2) /* force fd2 comme
synonyme de fd1 */
/* fd1 : descripteur à dupliquer */
/* fd2 : nouveau descripteur */
```

Valeur retournée : -1 en cas d'erreur.

Cette primitive oblige le descripteur `fd2` à devenir synonyme du descripteur `fd1`. Notons que `dup2()` ferme le descripteur `fd2` si celui-ci était ouvert. Par exemple, supposons que le descrispteur `d` résulte de l'ouverture d'un fichier, disons `fic_sortie`. La commande `dup2(d,1);` a pour effet de faire pointer le descripteur `d` et le descripteur de la sortie standard 1 sur le même fichier `fic_sortie`. Écrire l'appel système suivant `write(d,...);` ou `write(1,...);` a le même effet. Ceci permet d'effectuer une redirection.

Primitive `dup2()`

```
#include <unistd.h>
int dup2(int fd1,int fd2) /* force fd2 comme
synonyme de fd1 */
/* fd1 : descripteur à dupliquer */
/* fd2 : nouveau descripteur */
```

Valeur retournée : -1 en cas d'erreur.

Cette primitive oblige le descripteur `fd2` à devenir synonyme du descripteur `fd1`. Notons que `dup2()` ferme le descripteur `fd2` si celui-ci était ouvert. Par exemple, supposons que le descrispteur `d` résulte de l'ouverture d'un fichier, disons `fic_sortie`. La commande `dup2(d,1);` a pour effet de faire pointer le descripteur `d` et le descripteur de la sortie standard 1 sur le même fichier `fic_sortie`. Écrire l'appel système suivant `write(d,...);` ou `write(1,...);` a le même effet. Ceci permet d'effectuer une redirection.

Primitive `dup2()`

```
#include <unistd.h>
int dup2(int fd1,int fd2) /* force fd2 comme
synonyme de fd1 */
/* fd1 : descripteur à dupliquer */
/* fd2 : nouveau descripteur */
```

Valeur retournée : -1 en cas d'erreur.

Cette primitive oblige le descripteur `fd2` à devenir synonyme du descripteur `fd1`. Notons que `dup2()` ferme le descripteur `fd2` si celui-ci était ouvert. Par exemple, supposons que le descrispteur `d` résulte de l'ouverture d'un fichier, disons `fic_sortie`. La commande `dup2(d,1);` a pour effet de faire pointer le descripteur `d` et le descripteur de la sortie standard 1 sur le même fichier `fic_sortie`. Écrire l'appel système suivant `write(d,...);` ou `write(1,...);` a le même effet. Ceci permet d'effectuer une redirection.

Primitive `dup2()`

```
#include <unistd.h>
int dup2(int fd1,int fd2) /* force fd2 comme
synonyme de fd1 */
/* fd1 : descripteur à dupliquer */
/* fd2 : nouveau descripteur */
```

Valeur retournée : -1 en cas d'erreur.

Cette primitive oblige le descripteur `fd2` à devenir synonyme du descripteur `fd1`. Notons que `dup2()` ferme le descripteur `fd2` si celui-ci était ouvert. Par exemple, supposons que le descriptpteur `d` résulte de l'ouverture d'un fichier, disons `fic_sortie`. La commande `dup2(d,1);` a pour effet de faire pointer le descripteur `d` et le descripteur de la sortie standard 1 sur le même fichier `fic_sortie`. Écrire l'appel système suivant `write(d,...);` ou `write(1,...);` a le même effet. Ceci permet d'effectuer une redirection.

Primitive `dup2()`

```
#include <unistd.h>
int dup2(int fd1,int fd2) /* force fd2 comme
synonyme de fd1 */
/* fd1 : descripteur à dupliquer */
/* fd2 : nouveau descripteur */
```

Valeur retournée : -1 en cas d'erreur.

Cette primitive oblige le descripteur `fd2` à devenir synonyme du descripteur `fd1`. Notons que `dup2()` ferme le descripteur `fd2` si celui-ci était ouvert. Par exemple, supposons que le descrispteur `d` résulte de l'ouverture d'un fichier, disons `fic_sortie`. La commande `dup2(d,1);` a pour effet de faire pointer le descripteur `d` et le descripteur de la sortie standard 1 sur le même fichier `fic_sortie`. Écrire l'appel système suivant `write(d,...);` ou `write(1,...);` a le même effet. Ceci permet d'effectuer une redirection.

Primitive `dup2()`

```
#include <unistd.h>
int dup2(int fd1,int fd2) /* force fd2 comme
synonyme de fd1 */
/* fd1 : descripteur à dupliquer */
/* fd2 : nouveau descripteur */
```

Valeur retournée : -1 en cas d'erreur.

Cette primitive oblige le descripteur `fd2` à devenir synonyme du descripteur `fd1`. Notons que `dup2()` ferme le descripteur `fd2` si celui-ci était ouvert. Par exemple, supposons que le descrispteur `d` résulte de l'ouverture d'un fichier, disons `fic_sortie`. La commande `dup2(d,1);` a pour effet de faire pointer le descripteur `d` et le descripteur de la sortie standard 1 sur le même fichier `fic_sortie`. Écrire l'appel système suivant `write(d,...);` ou `write(1,...);` a le même effet. Ceci permet d'effectuer une redirection.

Primitive `dup2()`

```
#include <unistd.h>
int dup2(int fd1,int fd2) /* force fd2 comme
synonyme de fd1 */
/* fd1 : descripteur à dupliquer */
/* fd2 : nouveau descripteur */
```

Valeur retournée : -1 en cas d'erreur.

Cette primitive oblige le descripteur `fd2` à devenir synonyme du descripteur `fd1`. Notons que `dup2()` ferme le descripteur `fd2` si celui-ci était ouvert. Par exemple, supposons que le descrispteur `d` résulte de l'ouverture d'un fichier, disons `fic_sortie`. La commande `dup2(d,1);` a pour effet de faire pointer le descripteur `d` et le descripteur de la sortie standard 1 sur le même fichier `fic_sortie`. Écrire l'appel système suivant `write(d,...);` ou `write(1,...);` a le même effet. Ceci permet d'effectuer une redirection.

Primitive `dup2()`

```
#include <unistd.h>
int dup2(int fd1,int fd2) /* force fd2 comme
synonyme de fd1 */
/* fd1 : descripteur à dupliquer */
/* fd2 : nouveau descripteur */
```

Valeur retournée : -1 en cas d'erreur.

Cette primitive oblige le descripteur `fd2` à devenir synonyme du descripteur `fd1`. Notons que `dup2()` ferme le descripteur `fd2` si celui-ci était ouvert. Par exemple, supposons que le descrispteur `d` résulte de l'ouverture d'un fichier, disons `fic_sortie`. La commande `dup2(d,1);` a pour effet de faire pointer le descripteur `d` et le descripteur de la sortie standard 1 sur le même fichier `fic_sortie`. Écrire l'appel système suivant `write(d,...);` ou `write(1,...);` a le même effet. Ceci permet d'effectuer une redirection.

Exemple : Redirection de la sortie standard

Ce programme exécute la commande shell `ps`, puis redirige le résultat vers un fichier `fic_sortie`. Ainsi l'exécution de ce programme ne devrait plus rien donner à l'écran. La primitive `execl()` exécute la commande passée en argument.

Exemple : Redirection de la sortie standard

Ce programme exécute la commande shell `ps`, puis redirige le résultat vers un fichier `fic_sortie`. Ainsi l'exécution de ce programme ne devrait plus rien donner à l'écran. La primitive `execl()` exécute la commande passée en argument.

Fichier `test_dup2.c` :

```
#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
main() { int fd;
/* affecte au fichier fic_sortie le
descripteur fd */
if ((fd = open("fic_sortie",O_CREAT | O_WRONLY
| O_TRUNC, 0666)) == -1) {
perror("Erreur sur l'ouverture de
fic_sortie");
exit(1);}
dup2(fd,1); /* duplique la sortie standard */
execl("/bin/ps","ps",NULL); }
```

Fichier `test_dup2.c` :

```
#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
main() { int fd;
/* affecte au fichier fic_sortie le
descripteur fd */
if ((fd = open("fic_sortie",O_CREAT | O_WRONLY
| O_TRUNC, 0666)) == -1) {
perror("Erreur sur l'ouverture de
fic_sortie");
exit(1);}
dup2(fd,1); /* duplique la sortie standard */
execl("/bin/ps","ps",NULL); }
```

Résultat de l'exécution :

```
Systeme> test_dup2
Systeme> more fic_sortie
PID TTY TIME COMMAND
3954 ttyp3 0 :03 csh
```

Résultat de l'exécution :

```
Systeme> test_dup2
Systeme> more fic_sortie
PID TTY TIME COMMAND
3954 ttyp3 0 :03 csh
```

Résultat de l'exécution :

```
Systeme> test_dup2
Systeme> more fic_sortie
PID TTY TIME COMMAND
3954 ttyp3 0 :03 csh
```

Au lieu d'afficher le résultat de la commande `ps` sur la sortie standard (l'écran), on l'écrit dans `"fichier_sortie"` : on a redirigé la sortie standard vers `"fichier_sortie"`.

Un répertoire est un fichier contenant, entre autres, un ensemble de couples (nom de fichier, numéro d'inode). Il est possible d'ouvrir un répertoire pour le parcourir. C'est ce que fait la commande `ls`. Unix fournit plusieurs fonctions pour parcourir un répertoire. Rappelons que Unix hiérarchise l'ensemble des répertoires comme un arbre dont la racine (`root`) est notée " / ". Par convention, le répertoire courant s'appelle " . " et son père dans la hiérarchie s'appelle " .. ". Ainsi par exemple, si on tape `cd ..`, on remonte d'un cran dans la hiérarchie arborescente des répertoires. De même `cd ../..` nous permet de remonter de deux crans, *etc.*

Un répertoire est un fichier contenant, entre autres, un ensemble de couples (nom de fichier, numéro d'inode). Il est possible d'ouvrir un répertoire pour le parcourir. C'est ce que fait la commande `ls`. Unix fournit plusieurs fonctions pour parcourir un répertoire. Rappelons que Unix hiérarchise l'ensemble des répertoires comme un arbre dont la racine (`root`) est notée " / ". Par convention, le répertoire courant s'appelle " . " et son père dans la hiérarchie s'appelle " .. ". Ainsi par exemple, si on tape `cd ..`, on remonte d'un cran dans la hiérarchie arborescente des répertoires. De même `cd ../..` nous permet de remonter de deux crans, *etc.*

Un répertoire est un fichier contenant, entre autres, un ensemble de couples (nom de fichier, numéro d'inode). Il est possible d'ouvrir un répertoire pour le parcourir. C'est ce que fait la commande `ls`. Unix fournit plusieurs fonctions pour parcourir un répertoire. Rappelons que Unix hiérarchise l'ensemble des répertoires comme un arbre dont la racine (`root`) est notée " / ". Par convention, le répertoire courant s'appelle " . " et son père dans la hiérarchie s'appelle " .. ". Ainsi par exemple, si on tape `cd ..`, on remonte d'un cran dans la hiérarchie arborescente des répertoires. De même `cd ../..` nous permet de remonter de deux crans, *etc.*

Un répertoire est un fichier contenant, entre autres, un ensemble de couples (nom de fichier, numéro d'inode). Il est possible d'ouvrir un répertoire pour le parcourir. C'est ce que fait la commande `ls`. Unix fournit plusieurs fonctions pour parcourir un répertoire. Rappelons que Unix hiérarchise l'ensemble des répertoires comme un arbre dont la racine (`root`) est notée " / ". Par convention, le répertoire courant s'appelle " . " et son père dans la hiérarchie s'appelle " .. ". Ainsi par exemple, si on tape `cd ..`, on remonte d'un cran dans la hiérarchie arborescente des répertoires. De même `cd ../..` nous permet de remonter de deux crans, *etc.*

Un répertoire est un fichier contenant, entre autres, un ensemble de couples (nom de fichier, numéro d'inode). Il est possible d'ouvrir un répertoire pour le parcourir. C'est ce que fait la commande `ls`. Unix fournit plusieurs fonctions pour parcourir un répertoire. Rappelons que Unix hiérarchise l'ensemble des répertoires comme un arbre dont la racine (`root`) est notée " / ". Par convention, le répertoire courant s'appelle " . " et son père dans la hiérarchie s'appelle " .. ". Ainsi par exemple, si on tape `cd ..`, on remonte d'un cran dans la hiérarchie arborescente des répertoires. De même `cd ../..` nous permet de remonter de deux crans, *etc.*

Un répertoire est un fichier contenant, entre autres, un ensemble de couples (nom de fichier, numéro d'inode). Il est possible d'ouvrir un répertoire pour le parcourir. C'est ce que fait la commande `ls`. Unix fournit plusieurs fonctions pour parcourir un répertoire. Rappelons que Unix hiérarchise l'ensemble des répertoires comme un arbre dont la racine (`root`) est notée " / ". Par convention, le répertoire courant s'appelle " . " et son père dans la hiérarchie s'appelle " .. ". Ainsi par exemple, si on tape `cd ..`, on remonte d'un cran dans la hiérarchie arborescente des répertoires. De même `cd ../..` nous permet de remonter de deux crans, *etc.*

Un répertoire est un fichier contenant, entre autres, un ensemble de couples (nom de fichier, numéro d'inode). Il est possible d'ouvrir un répertoire pour le parcourir. C'est ce que fait la commande `ls`. Unix fournit plusieurs fonctions pour parcourir un répertoire. Rappelons que Unix hiérarchise l'ensemble des répertoires comme un arbre dont la racine (`root`) est notée " / ". Par convention, le répertoire courant s'appelle " . " et son père dans la hiérarchie s'appelle " .. ". Ainsi par exemple, si on tape `cd ..`, on remonte d'un cran dans la hiérarchie arborescente des répertoires. De même `cd ../..` nous permet de remonter de deux crans, *etc.*

Un répertoire est un fichier contenant, entre autres, un ensemble de couples (nom de fichier, numéro d'inode). Il est possible d'ouvrir un répertoire pour le parcourir. C'est ce que fait la commande `ls`. Unix fournit plusieurs fonctions pour parcourir un répertoire. Rappelons que Unix hiérarchise l'ensemble des répertoires comme un arbre dont la racine (`root`) est notée " / ". Par convention, le répertoire courant s'appelle " . " et son père dans la hiérarchie s'appelle " .. ". Ainsi par exemple, si on tape `cd ..`, on remonte d'un cran dans la hiérarchie arborescente des répertoires. De même `cd ../..` nous permet de remonter de deux crans, *etc.*

Un répertoire est un fichier contenant, entre autres, un ensemble de couples (nom de fichier, numéro d'inode). Il est possible d'ouvrir un répertoire pour le parcourir. C'est ce que fait la commande `ls`. Unix fournit plusieurs fonctions pour parcourir un répertoire. Rappelons que Unix hiérarchise l'ensemble des répertoires comme un arbre dont la racine (`root`) est notée " / ". Par convention, le répertoire courant s'appelle " . " et son père dans la hiérarchie s'appelle " .. ". Ainsi par exemple, si on tape `cd ..`, on remonte d'un cran dans la hiérarchie arborescente des répertoires. De même `cd ../..` nous permet de remonter de deux crans, *etc*.

Les fonctions `opendir()` et `closedir()`

Profil :

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

La fonction `opendir()` ouvre le répertoire `name` et retourne un pointeur vers la structure `DIR` qui joue le même rôle que `FILE`. La valeur `NULL` est renvoyée si l'ouverture n'est pas possible.

Profil :

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

La fonction `closedir()` ferme un répertoire ouvert par `opendir`.

Les fonctions `opendir()` et `closedir()`

Profil :

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

La fonction `opendir()` ouvre le répertoire `name` et retourne un pointeur vers la structure `DIR` qui joue le même rôle que `FILE`. La valeur `NULL` est renvoyée si l'ouverture n'est pas possible.

Profil :

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

La fonction `closedir()` ferme un répertoire ouvert par `opendir`.

Les fonctions `opendir()` et `closedir()`

Profil :

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

La fonction `opendir()` ouvre le répertoire `name` et retourne un pointeur vers la structure `DIR` qui joue le même rôle que `FILE`. La valeur `NULL` est renvoyée si l'ouverture n'est pas possible.

Profil :

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

La fonction `closedir()` ferme un répertoire ouvert par `opendir`.

Les fonctions `opendir()` et `closedir()`

Profil :

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

La fonction `opendir()` ouvre le répertoire `name` et retourne un pointeur vers la structure `DIR` qui joue le même rôle que `FILE`. La valeur `NULL` est renvoyée si l'ouverture n'est pas possible.

Profil :

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

La fonction `closedir()` ferme un répertoire ouvert par `opendir`.

Les fonctions `opendir()` et `closedir()`

Profil :

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

La fonction `opendir()` ouvre le répertoire `name` et retourne un pointeur vers la structure `DIR` qui joue le même rôle que `FILE`. La valeur `NULL` est renvoyée si l'ouverture n'est pas possible.

Profil :

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

La fonction `closedir()` ferme un répertoire ouvert par `opendir`.

La fonction `readdir()`

Profil :

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

La fonction `readdir()` retourne un pointeur vers une structure de type `struct dirent` représentant le prochain couple (nom de fichier, inode) dans le répertoire. La structure `dirent` contient deux champs `d_name` et `d_ino` qui représentent respectivement le nom du fichier et son numéro d'inode. Quand la fin du répertoire est atteinte, `readdir()` renvoie `NULL`.

La fonction `readdir()`

Profil :

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

La fonction `readdir()` retourne un pointeur vers une structure de type `struct dirent` représentant le prochain couple (nom de fichier, inode) dans le répertoire. La structure `dirent` contient deux champs `d_name` et `d_ino` qui représentent respectivement le nom du fichier et son numéro d'inode. Quand la fin du répertoire est atteinte, `readdir()` renvoie `NULL`.

La fonction `readdir()`

Profil :

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

La fonction `readdir()` retourne un pointeur vers une structure de type `struct dirent` représentant le prochain couple (nom de fichier, inode) dans le répertoire. La structure `dirent` contient deux champs `d_name` et `d_ino` qui représentent respectivement le nom du fichier et son numéro d'inode. Quand la fin du répertoire est atteinte, `readdir()` renvoie `NULL`.

La fonction `readdir()`

Profil :

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

La fonction `readdir()` retourne un pointeur vers une structure de type `struct dirent` représentant le prochain couple (nom de fichier, inode) dans le répertoire. La structure `dirent` contient deux champs `d_name` et `d_ino` qui représentent respectivement le nom du fichier et son numéro d'inode. Quand la fin du répertoire est atteinte, `readdir()` renvoie `NULL`.

Unix étant un système multi-utilisateur, il se peut que deux utilisateurs veuillent accéder en même temps au même fichier. Si les fichiers sont consultés en lecture, il n'y a pas de conflit. Les problèmes surgissent lorsqu'un des processus modifie le fichier pendant que d'autres le lisent. Sans précaution, des informations érronées peuvent être enregistrées. Un processus lit un enregistrement dans son espace d'adressage, le modifie et le réécrit sur disque. Si deux processus lisent une même zone du fichier, la modifient et la réécrivent, les résultats dépendront de l'ordre de réécriture, mais les modifications d'un des processus seront perdues. On a un problème d'accès concurrent, et il faut édicter des règles pour que tout se passe bien.

Unix étant un système multi-utilisateur, il se peut que deux utilisateurs veuillent accéder en même temps au même fichier. Si les fichiers sont consultés en lecture, il n'y a pas de conflit. Les problèmes surgissent lorsqu'un des processus modifie le fichier pendant que d'autres le lisent. Sans précaution, des informations érronées peuvent être enregistrées. Un processus lit un enregistrement dans son espace d'adressage, le modifie et le réécrit sur disque. Si deux processus lisent une même zone du fichier, la modifient et la réécrivent, les résultats dépendront de l'ordre de réécriture, mais les modifications d'un des processus seront perdues. On a un problème d'accès concurrent, et il faut édicter des règles pour que tout se passe bien.

Unix étant un système multi-utilisateur, il se peut que deux utilisateurs veuillent accéder en même temps au même fichier. Si les fichiers sont consultés en lecture, il n'y a pas de conflit. Les problèmes surgissent lorsqu'un des processus modifie le fichier pendant que d'autres le lisent. Sans précaution, des informations érronées peuvent être enregistrées. Un processus lit un enregistrement dans son espace d'adressage, le modifie et le réécrit sur disque. Si deux processus lisent une même zone du fichier, la modifient et la réécrivent, les résultats dépendront de l'ordre de réécriture, mais les modifications d'un des processus seront perdues. On a un problème d'accès concurrent, et il faut édicter des règles pour que tout se passe bien.

Unix étant un système multi-utilisateur, il se peut que deux utilisateurs veuillent accéder en même temps au même fichier. Si les fichiers sont consultés en lecture, il n'y a pas de conflit. Les problèmes surgissent lorsqu'un des processus modifie le fichier pendant que d'autres le lisent. Sans précaution, des informations érronées peuvent être enregistrées. Un processus lit un enregistrement dans son espace d'adressage, le modifie et le réécrit sur disque. Si deux processus lisent une même zone du fichier, la modifient et la réécrivent, les résultats dépendront de l'ordre de réécriture, mais les modifications d'un des processus seront perdues. On a un problème d'accès concurrent, et il faut édicter des règles pour que tout se passe bien.

Unix étant un système multi-utilisateur, il se peut que deux utilisateurs veuillent accéder en même temps au même fichier. Si les fichiers sont consultés en lecture, il n'y a pas de conflit. Les problèmes surgissent lorsqu'un des processus modifie le fichier pendant que d'autres le lisent. Sans précaution, des informations érronées peuvent être enregistrées. Un processus lit un enregistrement dans son espace d'adressage, le modifie et le réécrit sur disque. Si deux processus lisent une même zone du fichier, la modifient et la réécrivent, les résultats dépendront de l'ordre de réécriture, mais les modifications d'un des processus seront perdues. On a un problème d'accès concurrent, et il faut édicter des règles pour que tout se passe bien.

Unix étant un système multi-utilisateur, il se peut que deux utilisateurs veuillent accéder en même temps au même fichier. Si les fichiers sont consultés en lecture, il n'y a pas de conflit. Les problèmes surgissent lorsqu'un des processus modifie le fichier pendant que d'autres le lisent. Sans précaution, des informations érronées peuvent être enregistrées. Un processus lit un enregistrement dans son espace d'adressage, le modifie et le réécrit sur disque. Si deux processus lisent une même zone du fichier, la modifient et la réécrivent, les résultats dépendront de l'ordre de réécriture, mais les modifications d'un des processus seront perdues. On a un problème d'accès concurrent, et il faut édicter des règles pour que tout se passe bien.

Unix étant un système multi-utilisateur, il se peut que deux utilisateurs veuillent accéder en même temps au même fichier. Si les fichiers sont consultés en lecture, il n'y a pas de conflit. Les problèmes surgissent lorsqu'un des processus modifie le fichier pendant que d'autres le lisent. Sans précaution, des informations érronées peuvent être enregistrées. Un processus lit un enregistrement dans son espace d'adressage, le modifie et le réécrit sur disque. Si deux processus lisent une même zone du fichier, la modifient et la réécrivent, les résultats dépendront de l'ordre de réécriture, mais les modifications d'un des processus seront perdues. On a un problème d'accès concurrent, et il faut édicter des règles pour que tout se passe bien.

Pour un fichier sous Unix, il y a deux façons de procéder. Un processus peut poser un verrou sur tout le fichier ou sur une partie seulement d'un fichier si celui-ci est très souvent utilisé. Il y a deux sortes de verrou :

- le verrou en lecture ou verrou partagé : on lit le fichier sans le modifier ; plusieurs utilisateurs peuvent poser un verrou en lecture ;
- le verrou en écriture ou verrou exclusif : il ne peut y avoir qu'un seul processus en écriture ; de plus aucun verrou en lecture ne doit être posé pour pouvoir modifier le fichier, d'où l'exclusivité du verrou.

Pour un fichier sous Unix, il y a deux façons de procéder. Un processus peut poser un verrou sur tout le fichier ou sur une partie seulement d'un fichier si celui-ci est très souvent utilisé. Il y a deux sortes de verrou :

- le verrou en lecture ou verrou partagé : on lit le fichier sans le modifier ; plusieurs utilisateurs peuvent poser un verrou en lecture ;
- le verrou en écriture ou verrou exclusif : il ne peut y avoir qu'un seul processus en écriture ; de plus aucun verrou en lecture ne doit être posé pour pouvoir modifier le fichier, d'où l'exclusivité du verrou.

Pour un fichier sous Unix, il y a deux façons de procéder. Un processus peut poser un verrou sur tout le fichier ou sur une partie seulement d'un fichier si celui-ci est très souvent utilisé. Il y a deux sortes de verrou :

- le verrou en lecture ou verrou partagé : on lit le fichier sans le modifier ; plusieurs utilisateurs peuvent poser un verrou en lecture ;
- le verrou en écriture ou verrou exclusif : il ne peut y avoir qu'un seul processus en écriture ; de plus aucun verrou en lecture ne doit être posé pour pouvoir modifier le fichier, d'où l'exclusivité du verrou.

Pour un fichier sous Unix, il y a deux façons de procéder. Un processus peut poser un verrou sur tout le fichier ou sur une partie seulement d'un fichier si celui-ci est très souvent utilisé. Il y a deux sortes de verrou :

- le verrou en lecture ou verrou partagé : on lit le fichier sans le modifier ; plusieurs utilisateurs peuvent poser un verrou en lecture ;
- le verrou en écriture ou verrou exclusif : il ne peut y avoir qu'un seul processus en écriture ; de plus aucun verrou en lecture ne doit être posé pour pouvoir modifier le fichier, d'où l'exclusivité du verrou.

Pour un fichier sous Unix, il y a deux façons de procéder. Un processus peut poser un verrou sur tout le fichier ou sur une partie seulement d'un fichier si celui-ci est très souvent utilisé. Il y a deux sortes de verrou :

- le verrou en lecture ou verrou partagé : on lit le fichier sans le modifier ; plusieurs utilisateurs peuvent poser un verrou en lecture ;
- le verrou en écriture ou verrou exclusif : il ne peut y avoir qu'un seul processus en écriture ; de plus aucun verrou en lecture ne doit être posé pour pouvoir modifier le fichier, d'où l'exclusivité du verrou.

Pour un fichier sous Unix, il y a deux façons de procéder. Un processus peut poser un verrou sur tout le fichier ou sur une partie seulement d'un fichier si celui-ci est très souvent utilisé. Il y a deux sortes de verrou :

- le verrou en lecture ou verrou partagé : on lit le fichier sans le modifier ; plusieurs utilisateurs peuvent poser un verrou en lecture ;
- le verrou en écriture ou verrou exclusif : il ne peut y avoir qu'un seul processus en écriture ; de plus aucun verrou en lecture ne doit être posé pour pouvoir modifier le fichier, d'où l'exclusivité du verrou.

Pour un fichier sous Unix, il y a deux façons de procéder. Un processus peut poser un verrou sur tout le fichier ou sur une partie seulement d'un fichier si celui-ci est très souvent utilisé. Il y a deux sortes de verrou :

- le verrou en lecture ou verrou partagé : on lit le fichier sans le modifier ; plusieurs utilisateurs peuvent poser un verrou en lecture ;
- le verrou en écriture ou verrou exclusif : il ne peut y avoir qu'un seul processus en écriture ; de plus aucun verrou en lecture ne doit être posé pour pouvoir modifier le fichier, d'où l'exclusivité du verrou.

Pour un fichier sous Unix, il y a deux façons de procéder. Un processus peut poser un verrou sur tout le fichier ou sur une partie seulement d'un fichier si celui-ci est très souvent utilisé. Il y a deux sortes de verrou :

- le verrou en lecture ou verrou partagé : on lit le fichier sans le modifier ; plusieurs utilisateurs peuvent poser un verrou en lecture ;
- le verrou en écriture ou verrou exclusif : il ne peut y avoir qu'un seul processus en écriture ; de plus aucun verrou en lecture ne doit être posé pour pouvoir modifier le fichier, d'où l'exclusivité du verrou.

Pour un fichier sous Unix, il y a deux façons de procéder. Un processus peut poser un verrou sur tout le fichier ou sur une partie seulement d'un fichier si celui-ci est très souvent utilisé. Il y a deux sortes de verrou :

- le verrou en lecture ou verrou partagé : on lit le fichier sans le modifier ; plusieurs utilisateurs peuvent poser un verrou en lecture ;
- le verrou en écriture ou verrou exclusif : il ne peut y avoir qu'un seul processus en écriture ; de plus aucun verrou en lecture ne doit être posé pour pouvoir modifier le fichier, d'où l'exclusivité du verrou.

Verrou

Le verrouillage du fichier se fait à l'aide de l'appel système `flock` qui pose sur un fichier ouvert de descripteur `fd` un verrou partagé (`LOCK_SH`) ou exclusif (`LOCK_EX`). Le retrait du verrou se fait par `LOCK_UN`. La pose d'un verrou incompatible est bloquante (le processus attend que le verrou soit levé) sauf si on indique `LOCK_NB` dans le champ `operation`. La syntaxe est la suivante :

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

avec `operation` l'un des mots-clefs `LOCK_SH`, `LOCK_EX`, `LOCK_UN`, `LOCK_NB`.

Verrou

Le verrouillage du fichier se fait à l'aide de l'appel système `flock` qui pose sur un fichier ouvert de descripteur `fd` un verrou partagé (`LOCK_SH`) ou exclusif (`LOCK_EX`). Le retrait du verrou se fait par `LOCK_UN`. La pose d'un verrou incompatible est bloquante (le processus attend que le verrou soit levé) sauf si on indique `LOCK_NB` dans le champ `operation`. La syntaxe est la suivante :

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

avec `operation` l'un des mots-clefs `LOCK_SH`, `LOCK_EX`, `LOCK_UN`, `LOCK_NB`.

Verrou

Le verrouillage du fichier se fait à l'aide de l'appel système `flock` qui pose sur un fichier ouvert de descripteur `fd` un verrou partagé (`LOCK_SH`) ou exclusif (`LOCK_EX`). Le retrait du verrou se fait par `LOCK_UN`. La pose d'un verrou incompatible est bloquante (le processus attend que le verrou soit levé) sauf si on indique `LOCK_NB` dans le champ `operation`. La syntaxe est la suivante :

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

avec `operation` l'un des mots-clefs `LOCK_SH`, `LOCK_EX`, `LOCK_UN`, `LOCK_NB`.

Verrou

Le verrouillage du fichier se fait à l'aide de l'appel système `flock` qui pose sur un fichier ouvert de descripteur `fd` un verrou partagé (`LOCK_SH`) ou exclusif (`LOCK_EX`). Le retrait du verrou se fait par `LOCK_UN`. La pose d'un verrou incompatible est bloquante (le processus attend que le verrou soit levé) sauf si on indique `LOCK_NB` dans le champ `operation`. La syntaxe est la suivante :

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

avec `operation` l'un des mots-clefs `LOCK_SH`, `LOCK_EX`, `LOCK_UN`, `LOCK_NB`.

Verrou

Le verrouillage du fichier se fait à l'aide de l'appel système `flock` qui pose sur un fichier ouvert de descripteur `fd` un verrou partagé (`LOCK_SH`) ou exclusif (`LOCK_EX`). Le retrait du verrou se fait par `LOCK_UN`. La pose d'un verrou incompatible est bloquante (le processus attend que le verrou soit levé) sauf si on indique `LOCK_NB` dans le champ `operation`. La syntaxe est la suivante :

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

avec `operation` l'un des mots-clefs `LOCK_SH`, `LOCK_EX`, `LOCK_UN`, `LOCK_NB`.

Les verrous servent donc à empêcher que plusieurs processus accèdent simultanément aux mêmes enregistrements. Considérons le programme suivant qui écrit dans un fichier 10 fois son pid ainsi que l'heure d'écriture.

```
/*fichier writel.c*/
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
```

Les verrous servent donc à empêcher que plusieurs processus accèdent simultanément aux mêmes enregistrements. Considérons le programme suivant qui écrit dans un fichier 10 fois son pid ainsi que l'heure d'écriture.

```
/*fichier writel.c*/
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
```

Les verrous servent donc à empêcher que plusieurs processus accèdent simultanément aux mêmes enregistrements. Considérons le programme suivant qui écrit dans un fichier 10 fois son pid ainsi que l'heure d'écriture.

```
/*fichier writel.c*/
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
```

Le main :

```
int main(int argc, char **argv) /* argv[1] =
fichier à écrire */{
int desc;
int i;
char buf[1024];
int n;
time_t secnow;
if (argc < 2) {
fprintf(stderr,"Usage : %s filename \n",
argv[0]);
exit(1);
}
if ((desc = open(argv[1], O_WRONLY | O_CREAT |
O_APPEND, 0666)) == -1) {
perror("Ouverture impossible ");
exit(1);}
```

Le main :

```
int main(int argc, char **argv) /* argv[1] =
fichier à écrire */{
int desc;
int i;
char buf[1024];
int n;
time_t secnow;
if (argc < 2) {
fprintf(stderr,"Usage : %s filename \n",
argv[0]);
exit(1);
}
if ((desc = open(argv[1], O_WRONLY | O_CREAT |
O_APPEND, 0666)) == -1) {
perror("Ouverture impossible ");
exit(1);}
```

Verrouillage du fichier :

```
#ifdef VERROU /* on verrouille tout le fichier
*/
if (flock(desc, LOCK_EX) == -1) {
perror("lock");
exit(1);
}
else
printf("processus %ld : verrou posé \n", (long
int)getpid());
#endif
```

Verrouillage du fichier :

```
#ifdef VERROU /* on verrouille tout le fichier
*/
if (flock(desc, LOCK_EX) == -1) {
perror("lock");
exit(1);
}
else
printf("processus %ld : verrou posé \n", (long
int)getpid());
#endif
```

Écritures du pid et de l'heure :

```
for (i =0; i< 10; i++) {
time(&secnow);
sprintf(buf,"%ld : écriture à %s ", (long
int)getpid(), ctime(&secnow));
n = write (desc, buf, strlen(buf));
sleep(1);
}
#ifdef VERROU /* levée du verrou */
flock(desc, LOCK_UN);
#endif
return 0;
}
```

Écritures du pid et de l'heure :

```
for (i =0; i< 10; i++) {
time(&secnow);
sprintf(buf,"%ld : écriture à %s ", (long
int)getpid(), ctime(&secnow));
n = write (desc, buf, strlen(buf));
sleep(1);
}
#ifdef VERROU /* levée du verrou */
flock(desc, LOCK_UN);
#endif
return 0;
}
```

On compile le programme sans définir la constante `VERROU`, donc sans verrou implémenté par la fonction `flock`.

```
Systeme> gcc -Wall writel.c
Systeme> ./a.out essai & ./a.out essai &
./a.out essai & ./a.out essai &
[1] 1959
[2] 1960
[3] 1961
[4] 1962
```

On compile le programme sans définir la constante `VERROU`, donc sans verrou implémenté par la fonction `flock`.

```
Systeme> gcc -Wall writel.c
Systeme> ./a.out essai & ./a.out essai &
./a.out essai & ./a.out essai &
[1] 1959
[2] 1960
[3] 1961
[4] 1962
```

Puis on inspecte le contenu du fichier `essai` :

```
Systeme> cat essai
1959 : écriture à Sat Nov 29 18 :43 :59 1997
1960 : écriture à Sat Nov 29 18 :43 :59 1997
1961 : écriture à Sat Nov 29 18 :43 :59 1997
1962 : écriture à Sat Nov 29 18 :43 :59 1997
1959 : écriture à Sat Nov 29 18 :44 :00 1997
1960 : écriture à Sat Nov 29 18 :44 :00 1997
1961 : écriture à Sat Nov 29 18 :44 :00 1997
...
1962 : écriture à Sat Nov 29 18 :44 :08 1997
```

Toutes les écritures sont mélangées. Voyons comment sérialiser les accès.

Puis on inspecte le contenu du fichier `essai` :

```
Systeme> cat essai
1959 : écriture à Sat Nov 29 18 :43 :59 1997
1960 : écriture à Sat Nov 29 18 :43 :59 1997
1961 : écriture à Sat Nov 29 18 :43 :59 1997
1962 : écriture à Sat Nov 29 18 :43 :59 1997
1959 : écriture à Sat Nov 29 18 :44 :00 1997
1960 : écriture à Sat Nov 29 18 :44 :00 1997
1961 : écriture à Sat Nov 29 18 :44 :00 1997
...
1962 : écriture à Sat Nov 29 18 :44 :08 1997
```

Toutes les écritures sont mélangées. Voyons comment sérialiser les accès.

Puis on inspecte le contenu du fichier `essai` :

```
Systeme> cat essai
1959 : écriture à Sat Nov 29 18 :43 :59 1997
1960 : écriture à Sat Nov 29 18 :43 :59 1997
1961 : écriture à Sat Nov 29 18 :43 :59 1997
1962 : écriture à Sat Nov 29 18 :43 :59 1997
1959 : écriture à Sat Nov 29 18 :44 :00 1997
1960 : écriture à Sat Nov 29 18 :44 :00 1997
1961 : écriture à Sat Nov 29 18 :44 :00 1997
...
1962 : écriture à Sat Nov 29 18 :44 :08 1997
```

Toutes les écritures sont mélangées. Voyons comment sérialiser les accès.

Puis on inspecte le contenu du fichier `essai` :

```
Systeme> cat essai
1959 : écriture à Sat Nov 29 18 :43 :59 1997
1960 : écriture à Sat Nov 29 18 :43 :59 1997
1961 : écriture à Sat Nov 29 18 :43 :59 1997
1962 : écriture à Sat Nov 29 18 :43 :59 1997
1959 : écriture à Sat Nov 29 18 :44 :00 1997
1960 : écriture à Sat Nov 29 18 :44 :00 1997
1961 : écriture à Sat Nov 29 18 :44 :00 1997
...
1962 : écriture à Sat Nov 29 18 :44 :08 1997
```

Toutes les écritures sont mélangées. Voyons comment sérialiser les accès.

On compile notre fichier `writel.c` mais cette fois en définissant la constante `VERROU` dans la ligne de commande.

```
Systeme> gcc -Wall -DVERROU writel.c
Systeme> ./a.out essai & ./a.out essai &
./a.out essai & ./a.out essai &
[1] 2534
[2] 2535
[3] 2536
[4] 2537
Systeme> processus 2534 : verrou posé
processus 2536 : verrou posé
processus 2535 : verrou posé
processus 2537 : verrou posé
[4] Done ./a.out essai
[3] + Done ./a.out essai
[2] + Done ./a.out essai
[1] + Done ./a.out essai
```

On compile notre fichier `writel.c` mais cette fois en définissant la constante `VERROU` dans la ligne de commande.

```
Systeme> gcc -Wall -DVERROU writel.c
Systeme> ./a.out essai & ./a.out essai &
./a.out essai & ./a.out essai &
[1] 2534
[2] 2535
[3] 2536
[4] 2537
Systeme> processus 2534 : verrou posé
processus 2536 : verrou posé
processus 2535 : verrou posé
processus 2537 : verrou posé
[4] Done ./a.out essai
[3] + Done ./a.out essai
[2] + Done ./a.out essai
[1] + Done ./a.out essai
```

Et si on imprime `essai`, on voit que chaque processus a pu écrire sans être interrompu.

```
Systeme> cat essai
2534 : écriture à Sat Nov 29 19 :23 :39 1997
...
2534 : écriture à Sat Nov 29 19 :23 :48 1997
2536 : écriture à Sat Nov 29 19 :23 :49 1997
...
2536 : écriture à Sat Nov 29 19 :23 :58 1997
2535 : écriture à Sat Nov 29 19 :23 :59 1997
...
2535 : écriture à Sat Nov 29 19 :24 :08 1997
2537 : écriture à Sat Nov 29 19 :24 :09 1997
...
2537 : écriture à Sat Nov 29 19 :24 :19 1997
```

Et si on imprime `essai`, on voit que chaque processus a pu écrire sans être interrompu.

```
Systeme> cat essai
2534 : écriture à Sat Nov 29 19 :23 :39 1997
...
2534 : écriture à Sat Nov 29 19 :23 :48 1997
2536 : écriture à Sat Nov 29 19 :23 :49 1997
...
2536 : écriture à Sat Nov 29 19 :23 :58 1997
2535 : écriture à Sat Nov 29 19 :23 :59 1997
...
2535 : écriture à Sat Nov 29 19 :24 :08 1997
2537 : écriture à Sat Nov 29 19 :24 :09 1997
...
2537 : écriture à Sat Nov 29 19 :24 :19 1997
```

Remarque : L'ordre d'appropriation du fichier par les processus est quelconque. Ainsi le processus 2536 est passé avant 2535.

Remarque : L'ordre d'appropriation du fichier par les processus est quelconque. Ainsi le processus 2536 est passé avant 2535.

Les pipes (ou tubes ou encore tubes anonymes) sont connus dans le monde d'Unix. Ils permettent, depuis un terminal, de rediriger la sortie d'une commande vers l'entrée d'une autre commande (à l'aide du caractère " | "). On peut également les utiliser entre processus dans un programme écrit en langage C. En C, un tube possède deux extrémités (l'entrée et la sortie du tube), et il n'est possible de faire passer des informations que dans un sens unique (on parle de tube unidirectionnel) : on peut écrire à l'entrée du tube et lire à sa sortie.
Concrétement on utilise l'appel système `pipe` pour créer un tube :

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

De cette façon, on crée un tube dont l'entrée est désignée par le descripteur de fichier `fd[1]` et la sortie par `fd[0]`.

Les pipes (ou tubes ou encore tubes anonymes) sont connus dans le monde d'Unix. Ils permettent, depuis un terminal, de rediriger la sortie d'une commande vers l'entrée d'une autre commande (à l'aide du caractère " | ").

On peut également les utiliser entre processus dans un programme écrit en langage C. En C, un tube possède deux extrémités (l'entrée et la sortie du tube), et il n'est possible de faire passer des informations que dans un sens unique (on parle de tube unidirectionnel) : on peut écrire à l'entrée du tube et lire à sa sortie.

Concrétement on utilise l'appel système `pipe` pour créer un tube :

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

De cette façon, on crée un tube dont l'entrée est désignée par le descripteur de fichier `fd[1]` et la sortie par `fd[0]`.

Les pipes (ou tubes ou encore tubes anonymes) sont connus dans le monde d'Unix. Ils permettent, depuis un terminal, de rediriger la sortie d'une commande vers l'entrée d'une autre commande (à l'aide du caractère " | "). On peut également les utiliser entre processus dans un programme écrit en langage C. En C, un tube possède deux extrémités (l'entrée et la sortie du tube), et il n'est possible de faire passer des informations que dans un sens unique (on parle de tube unidirectionnel) : on peut écrire à l'entrée du tube et lire à sa sortie.
Concrétement on utilise l'appel système `pipe` pour créer un tube :

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

De cette façon, on crée un tube dont l'entrée est désignée par le descripteur de fichier `fd[1]` et la sortie par `fd[0]`.

Les pipes (ou tubes ou encore tubes anonymes) sont connus dans le monde d'Unix. Ils permettent, depuis un terminal, de rediriger la sortie d'une commande vers l'entrée d'une autre commande (à l'aide du caractère " | "). On peut également les utiliser entre processus dans un programme écrit en langage C. En C, un tube possède deux extrémités (l'entrée et la sortie du tube), et il n'est possible de faire passer des informations que dans un sens unique (on parle de tube unidirectionnel) : on peut écrire à l'entrée du tube et lire à sa sortie.
Concrétement on utilise l'appel système `pipe` pour créer un tube :

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

De cette façon, on crée un tube dont l'entrée est désignée par le descripteur de fichier `fd[1]` et la sortie par `fd[0]`.

Les pipes (ou tubes ou encore tubes anonymes) sont connus dans le monde d'Unix. Ils permettent, depuis un terminal, de rediriger la sortie d'une commande vers l'entrée d'une autre commande (à l'aide du caractère " | "). On peut également les utiliser entre processus dans un programme écrit en langage C. En C, un tube possède deux extrémités (l'entrée et la sortie du tube), et il n'est possible de faire passer des informations que dans un sens unique (on parle de tube unidirectionnel) : on peut écrire à l'entrée du tube et lire à sa sortie.
Concrétement on utilise l'appel système `pipe` pour créer un tube :

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

De cette façon, on crée un tube dont l'entrée est désignée par le descripteur de fichier `fd[1]` et la sortie par `fd[0]`.

Les pipes (ou tubes ou encore tubes anonymes) sont connus dans le monde d'Unix. Ils permettent, depuis un terminal, de rediriger la sortie d'une commande vers l'entrée d'une autre commande (à l'aide du caractère " | "). On peut également les utiliser entre processus dans un programme écrit en langage C. En C, un tube possède deux extrémités (l'entrée et la sortie du tube), et il n'est possible de faire passer des informations que dans un sens unique (on parle de tube unidirectionnel) : on peut écrire à l'entrée du tube et lire à sa sortie.

Concrétement on utilise l'appel système `pipe` pour créer un tube :

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

De cette façon, on crée un tube dont l'entrée est désignée par le descripteur de fichier `fd[1]` et la sortie par `fd[0]`.

Les pipes (ou tubes ou encore tubes anonymes) sont connus dans le monde d'Unix. Ils permettent, depuis un terminal, de rediriger la sortie d'une commande vers l'entrée d'une autre commande (à l'aide du caractère " | "). On peut également les utiliser entre processus dans un programme écrit en langage C. En C, un tube possède deux extrémités (l'entrée et la sortie du tube), et il n'est possible de faire passer des informations que dans un sens unique (on parle de tube unidirectionnel) : on peut écrire à l'entrée du tube et lire à sa sortie. Concrétement on utilise l'appel système `pipe` pour créer un tube :

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

De cette façon, on crée un tube dont l'entrée est désignée par le descripteur de fichier `fd[1]` et la sortie par `fd[0]`.

Les pipes (ou tubes ou encore tubes anonymes) sont connus dans le monde d'Unix. Ils permettent, depuis un terminal, de rediriger la sortie d'une commande vers l'entrée d'une autre commande (à l'aide du caractère " | "). On peut également les utiliser entre processus dans un programme écrit en langage C. En C, un tube possède deux extrémités (l'entrée et la sortie du tube), et il n'est possible de faire passer des informations que dans un sens unique (on parle de tube unidirectionnel) : on peut écrire à l'entrée du tube et lire à sa sortie. Concrétement on utilise l'appel système `pipe` pour créer un tube :

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

De cette façon, on crée un tube dont l'entrée est désignée par le descripteur de fichier `fd[1]` et la sortie par `fd[0]`.

Les pipes (ou tubes ou encore tubes anonymes) sont connus dans le monde d'Unix. Ils permettent, depuis un terminal, de rediriger la sortie d'une commande vers l'entrée d'une autre commande (à l'aide du caractère " | "). On peut également les utiliser entre processus dans un programme écrit en langage C. En C, un tube possède deux extrémités (l'entrée et la sortie du tube), et il n'est possible de faire passer des informations que dans un sens unique (on parle de tube unidirectionnel) : on peut écrire à l'entrée du tube et lire à sa sortie.
Concrétement on utilise l'appel système `pipe` pour créer un tube :

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

De cette façon, on crée un tube dont l'entrée est désignée par le descripteur de fichier `fd[1]` et la sortie par `fd[0]`.

Exemple

```
void main (void){
char buffer[30];
int fd[2];
pipe(fd);
write(fd[1],"Hello world",strlen("Hello
world"));
read(fd[0],buffer,29);}
```

Remarque

On ne peut pas relire les informations d'un tube car la lecture est destructive, c'est-à-dire que les caractères lus sont " consommés "

Création d'un tube dans un processus ayant un fils

Supposons que l'on ait un processus qui, en plus de créer un tube, crée un fils. Alors comme tous descripteurs de fichiers, le tube est partagé entre le père et le fils. Ainsi si on écrit dans le tube, alors on ne sait pas lequel du fils ou du père va recevoir l'information. Pour être certain de qui va écrire et qui va lire dans le tube, il faut que les processus ferment les extrémités qu'ils n'utilisent pas. Par exemple, si le père veut communiquer avec le fils en utilisant un tube, il ferme sa sortie du tube et le fils ferme son entrée du tube. Ainsi tout ce qui sera écrit le sera par le père, et tout ce qui sera lu, le sera par le fils. Voyons cela sur un exemple.

Création d'un tube dans un processus ayant un fils

Supposons que l'on ait un processus qui, en plus de créer un tube, crée un fils. Alors comme tous descripteurs de fichiers, le tube est partagé entre le père et le fils. Ainsi si on écrit dans le tube, alors on ne sait pas lequel du fils ou du père va recevoir l'information. Pour être certain de qui va écrire et qui va lire dans le tube, il faut que les processus ferment les extrémités qu'ils n'utilisent pas. Par exemple, si le père veut communiquer avec le fils en utilisant un tube, il ferme sa sortie du tube et le fils ferme son entrée du tube. Ainsi tout ce qui sera écrit le sera par le père, et tout ce qui sera lu, le sera par le fils. Voyons cela sur un exemple.

Création d'un tube dans un processus ayant un fils

Supposons que l'on ait un processus qui, en plus de créer un tube, crée un fils. Alors comme tous descripteurs de fichiers, le tube est partagé entre le père et le fils. Ainsi si on écrit dans le tube, alors on ne sait pas lequel du fils ou du père va recevoir l'information. Pour être certain de qui va écrire et qui va lire dans le tube, il faut que les processus ferment les extrémités qu'ils n'utilisent pas. Par exemple, si le père veut communiquer avec le fils en utilisant un tube, il ferme sa sortie du tube et le fils ferme son entrée du tube. Ainsi tout ce qui sera écrit le sera par le père, et tout ce qui sera lu, le sera par le fils. Voyons cela sur un exemple.

Création d'un tube dans un processus ayant un fils

Supposons que l'on ait un processus qui, en plus de créer un tube, crée un fils. Alors comme tous descripteurs de fichiers, le tube est partagé entre le père et le fils. Ainsi si on écrit dans le tube, alors on ne sait pas lequel du fils ou du père va recevoir l'information. Pour être certain de qui va écrire et qui va lire dans le tube, il faut que les processus ferment les extrémités qu'ils n'utilisent pas. Par exemple, si le père veut communiquer avec le fils en utilisant un tube, il ferme sa sortie du tube et le fils ferme son entrée du tube. Ainsi tout ce qui sera écrit le sera par le père, et tout ce qui sera lu, le sera par le fils. Voyons cela sur un exemple.

Création d'un tube dans un processus ayant un fils

Supposons que l'on ait un processus qui, en plus de créer un tube, crée un fils. Alors comme tous descripteurs de fichiers, le tube est partagé entre le père et le fils. Ainsi si on écrit dans le tube, alors on ne sait pas lequel du fils ou du père va recevoir l'information. Pour être certain de qui va écrire et qui va lire dans le tube, il faut que les processus ferment les extrémités qu'ils n'utilisent pas. Par exemple, si le père veut communiquer avec le fils en utilisant un tube, il ferme sa sortie du tube et le fils ferme son entrée du tube. Ainsi tout ce qui sera écrit le sera par le père, et tout ce qui sera lu, le sera par le fils. Voyons cela sur un exemple.

Création d'un tube dans un processus ayant un fils

Supposons que l'on ait un processus qui, en plus de créer un tube, crée un fils. Alors comme tous descripteurs de fichiers, le tube est partagé entre le père et le fils. Ainsi si on écrit dans le tube, alors on ne sait pas lequel du fils ou du père va recevoir l'information. Pour être certain de qui va écrire et qui va lire dans le tube, il faut que les processus ferment les extrémités qu'ils n'utilisent pas. Par exemple, si le père veut communiquer avec le fils en utilisant un tube, il ferme sa sortie du tube et le fils ferme son entrée du tube. Ainsi tout ce qui sera écrit le sera par le père, et tout ce qui sera lu, le sera par le fils. Voyons cela sur un exemple.

Création d'un tube dans un processus ayant un fils

Supposons que l'on ait un processus qui, en plus de créer un tube, crée un fils. Alors comme tous descripteurs de fichiers, le tube est partagé entre le père et le fils. Ainsi si on écrit dans le tube, alors on ne sait pas lequel du fils ou du père va recevoir l'information. Pour être certain de qui va écrire et qui va lire dans le tube, il faut que les processus ferment les extrémités qu'ils n'utilisent pas. Par exemple, si le père veut communiquer avec le fils en utilisant un tube, il ferme sa sortie du tube et le fils ferme son entrée du tube. Ainsi tout ce qui sera écrit le sera par le père, et tout ce qui sera lu, le sera par le fils. Voyons cela sur un exemple.

Exemple (le fils)

```
int main (){
int desctube [2];
if (pipe(desctube)==-1)
{ printf("Erreur d'ouverture du tube");
return(1);}
int pid;
if((pid=fork())<0)
{ printf("Erreur creation de
fils");return(2);}
if (pid==0){
char buffer[100];
close(desctube[1]);//fermeture entrée tube
while (read(desctube[0],buffer,100) !=0)
printf("Fils a lu %s",buffer);
close(desctube[0]);}
```

Exemple (le père)

```
else {
char buffer [100];
close(desctube[0]);//fermeture sortie tube
strcpy(buffer,"Hello world\n");
write(desctube[1],buffer,strlen(buffer)+1);
close(desctube[1]);}
return (0);} }
```

Remarque

Les tubes anonymes ne peuvent être utilisés pour la communication entre processus que si les processus en question ont un lien de parenté (père, fils).