



Programmation Système

Gestion des Fichiers

Licence Sciences et Technologies Mention : Informatique
2ème Année

Les appels Système Relatifs aux Fichiers

- Types et Attributs
- Permissions
- Ouverture/Création/Suppression
- Lecture/Écriture
- Déplacement du Pointeur
- Duplication des Descripteurs
- Les verrous, Types et Modes Opératoires
- Modification des Attributs
- Manipulation des Répertoires

Types des Fichiers

Différents types de fichiers existent sous Unix

- Fichiers réguliers (-) // archives, multimédia, documents, ...
- Répertoires (d) // dossiers
- Fichiers spéciaux caractères (c) // écran, souris, webcam, imprimante, ...
- Fichiers spéciaux blocs (b) // disques, périphériques optiques, mécanismes de communication
- FIFO (p) // points de communication
- Socket (s) // liés à d'autres fichiers
- Liens symboliques (l)

Le type de fichier est encodé dans le champs **st_mode** de la structure **stat**.

```
$ ls -l
drwxrwxr-- 2 root      root     4096  2 août   15:11 DONNEES
-rw-rw-r-- 1 student   student    71   2 août   15:06 fichier
lrwxrwxrwx 1 student   student     4   2 août   15:04 file -> pipe
crw-rw---- 1 root      lp        6, 0  2 août   15:00 lp0
prw-rw---- 1 student   student    0   2 août   14:56 pipe
brw-rw---- 1 root      disk     8, 0  2 août   15:05 sda
$ ls -l DONNEES
-rw-rw-r-- 1 root      root    37   2 sept. 15:10 notes
$
```



Les Fichiers

Aux fichiers sont associés des attributs (contenus dans l'inode) qui les caractérisent.

Obtention des caractéristiques d'un fichier :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
    int stat (const char *path, struct stat *buf);
    int lstat(const char *path, struct stat *buf);
    int fstat(int fd, struct stat *buf);
```

The diagram illustrates the parameters for the `stat`, `lstat`, and `fstat` functions. It shows three boxes with arrows pointing to specific parameters in the function prototypes:

- A box labeled "Nom de fichier" has an arrow pointing to the `*path` parameter.
- A box labeled "Descripteur de fichier ouvert" has an arrow pointing to the `int fd` parameter.
- A box labeled "Tampon de sauvegarde de la structure stat" has an arrow pointing to the `struct stat *buf` parameter.

stat et **fstat** récupèrent la structure **stat** associée au fichier indiqué.
lstat est identique mais s'applique au lien indiqué plutôt qu'au fichier pointé.

Dans tous les cas, la structure **stat** correspondante est sauvegardée dans le tampon pointé par **buf**.

Tous les appels renvoient **0** en cas de succès et **-1** en cas d'erreur (et **errno** est modifiée en conséquence).

Les Fichiers

Aucun droit d'accès n'est nécessaire sur le fichier pour récupérer ses informations (uniquement droit de parcours de tous les répertoires dans le chemin menant au fichier).

La structure **stat**

```
struct stat{  
    dev_t      st_dev;          // identifiant du périphérique  
    ino_t      st_ino;         // numéro inode  
    mode_t     st_mode;        // type de fichier et protection (droits)  
    nlink_t    st_nlink;       // nombre liens matériels  
    uid_t      st_uid;         // UID propriétaire  
    gid_t      st_gid;         // GID propriétaire  
    dev_t      st_rdev;        // type périphérique (fichiers spéciaux)  
    off_t      st_size;        // taille totale en octets  
    blksize_t  st_blksize;     // taille de bloc pour E/S  
    blkcnt_t   st_blocks;      // nombre de blocs actuellement alloués  
    time_t     st_atime;       // heure dernier accès  
    time_t     st_mtime;       // heure dernière modification  
    time_t     st_ctime;       // heure dernier changement état  
}
```

Détermination du Type d'un Fichier

Des macros permettant de définir le type des fichiers sont disponibles (définies dans **sys/stat.h**)

- **S_ISREG**(buf->st_mode) : rend “vrai” (valeur = 1) s'il s'agit d'un fichier régulier
- **S_ISDIR**(buf->st_mode) : rend “vrai” s'il s'agit d'un fichier répertoire
- **S_ISCHR**(buf->st_mode) : rend “vrai” s'il s'agit d'un fichier spécial caractère
- **S_ISBLK**(buf->st_mode) : rend “vrai” s'il s'agit d'un fichier spécial bloc
- **S_ISFIFO**(buf->st_mode) : rend “vrai” s'il s'agit d'un fichier FIFO
- **S_ISLNK**(buf->st_mode) : rend “vrai” s'il s'agit d'un fichier lien symbolique
- **S_ISSOCK**(buf->st_mode) : rend “vrai” s'il s'agit d'un fichier socket

L'argument de chacune des macros est le champs **st_mode** de la structure **stat**.

Exemple : Détermination du Type

Programme `affiche_type.c`

```
int affiche_type(char *nom_fichier) {
    struct stat buf;
    if (stat(nom_fichier, &buf) == -1) {
        perror(nom_fichier); return 0;
    }
    if (S_ISREG(buf.st_mode))
        printf("Type fichier \"%s\" : regulier.\n", nom_fichier);
    else if (S_ISDIR(buf.st_mode))
        printf("Type fichier \"%s\" : repertoire.\n", nom_fichier);
    else if (S_ISCHR(buf.st_mode))
        printf("Type fichier \"%s\" : special caracteres.\n", nom_fichier);
    else if (S_ISBLK(buf.st_mode))
        printf("Type fichier \"%s\" : special bloc.\n", nom_fichier);
    else if (S_ISFIFO(buf.st_mode))
        printf("Type fichier \"%s\" : fifo.\n", nom_fichier);
    else if (S_ISLNK(buf.st_mode))
        printf("Type fichier \"%s\" : lien symbolique.\n", nom_fichier);
    else if (S_ISSOCK(buf.st_mode))
        printf("Type fichier \"%s\" : socket.\n", nom_fichier);
    return 1;
}
int main(int argc, char* argv[]) {
    int i;
    for (i = 1; i < argc; i++)
        affiche_type(argv[i]);
    return 1;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```

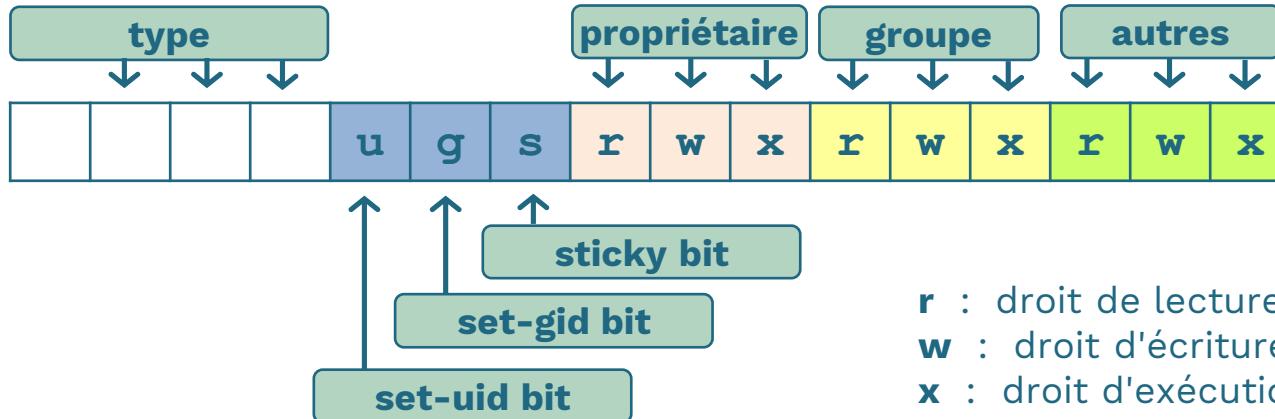
Exemple : Détermination du Type

Exécution `affiche_type.c`

```
$ ls -l
total 12
-rw-rw-r-- 1 student student  977  5 août 17:51 affiche_type.c
drwxrwxr-- 2 root      root   4096  2 août 15:11 DONNEES
-rw-rw-r-- 1 student student    71  2 août 15:06 fichier
lrwxrwxrwx 1 student student     4  2 août 15:04 file -> pipe
crw-rw---- 1 root      lp      6,  0  2 août 15:00 lp0
prw-rw---- 1 student student     0  2 août 14:56 pipe
brw-rw---- 1 root      disk    8,  0  2 août 15:05 sda
$  
$ gcc affiche_type.c -o affiche_type
$  
$ ./affiche_type affiche_type.c DONNEES fichier file lp0 pipe sda
Type fichier "affiche_type.c" : regulier.  
Type fichier "DONNEES" : repertoire.  
Type fichier "fichier" : regulier.  
Type fichier "file" : fifo. ← si utilisation de lstat, le résultat aurait été :  
Type fichier "file" : lien symbolique.  
Type fichier "lp0" : special caracteres.  
Type fichier "pipe" : fifo.  
Type fichier "sda" : special bloc.  
$
```

Droits d'Accès aux Fichiers

Champs `st_mode`



Mode	Description
<code>S_ISUID</code>	set_user_ID on execution
<code>S_ISGID</code>	set-group-ID on execution
<code>S_ISVTX</code>	saved-text (sticky bit)
<code>S_IRWXU</code>	read, write, and execute by user (owner)
<code>S_IRUSR</code>	read by user (owner)
<code>S_IWUSR</code>	write by user (owner)
<code>S_IXUSR</code>	execute by user (owner)

Mode	Description
<code>S_IRWXG</code>	read, write, execute by group
<code>S_IRGRP</code>	read by group
<code>S_IWGRP</code>	write by group
<code>S_IXGRP</code>	execute by group
<code>S_IRWXO</code>	read, write, execute by other (world)
<code>S_IROTH</code>	read by other (world)
<code>S_IWOTH</code>	write by other (world)
<code>S_IXOTH</code>	execute by other (world)

Rappel du rôle du set-uid bit

Lorsque le **set-uid** bit d'un exécutable est positionné, l'utilisateur qui demande son exécution aura comme **UID effectif** l'**UID** du propriétaire de cet exécutable.

Effets du set-uid bit

```
[student]$ id  
uid=1011(student) gid=1011(student) groupes=1011(student),10(wheel),18(dialout),1006(vboxsf)  
[student]$ ls -l  
total 12  
-rwxr-xr-x 1 root      root      6762  3 sept. 13:37 affiche_UID  
-rw-rw-r-- 1 root      root      152   3 sept. 13:34 affiche_UID.c  
[student]$ ./affiche_UID  
UID Réel = 1011  
UID Effectif = 1011  
[student]$  
[student]$ sudo ./affiche_UID  
UID Réel = 0  
UID Effectif = 0  
[student]$ sudo chmod u+s affiche_UID  
[student]$ ls -l  
total 12  
-rwsr-xr-x 1 root      root      6762  3 sept. 13:37 affiche_UID  
-rw-rw-r-- 1 root      root      152   3 sept. 13:34 affiche_UID.c  
[student]$ ./affiche_UID  
UID Réel = 1011  
UID Effectif = 0  
[student]$ sudo ./affiche_UID  
UID Réel = 0  
UID Effectif = 0
```

```
/* Programme affiche_UID.c */  
#include <stdio.h>  
  
int main(int argc, char* argv[]) {  
    printf("UID Réel = %d\n", getuid());  
    printf("UID Effectif = %d\n", geteuid());  
    return 1;
```

→ 's' ⇒ droit d'exécution ET set-UID bit positionnés

Le processus s'exécute avec un UID effectif égal à celui du propriétaire (root) de l'exécutable (affiche_UID).

← Aucun changement pour le propriétaire de l'exécutable.

Test des Droits d'Accès

Chaque fois qu'un processus ouvre, crée ou supprime un fichier, le noyau procède aux tests suivants :

1. si **ID user effectif** du processus est **0** (super user) l'accès est autorisé,
2. si **ID user effectif** du processus est égal à **ID propriétaire** du fichier :
 - a) si le mode d'accès correspond aux droits d'accès propriétaire, l'accès est autorisé,
 - b) sinon l'accès est refusé,
3. si **ID groupe effectif** du processus, ou l'un des IDs groupes supplémentaires du processus, est égal à ID groupe du fichier :
 - a) si le mode d'accès correspond aux droits d'accès du groupe, l'accès est autorisé,
 - b) sinon l'accès est refusé,
4. si le mode d'accès correspond aux droits d'accès des autres, l'accès est autorisé, sinon l'accès est refusé.

Test d'Accessibilité

Test d'accessibilité basé sur les ID utilisateur et de groupe réels

```
#include <unistd.h>
```

Fichier testé

Vérifications d'accès à effectuer

```
int access (const char *pathname, int mode);
```

mode	Description
F_OK	teste si le fichier existe
R_OK	teste si le fichier existe et autorise la lecture
W_OK	teste si le fichier existe et autorise l'écriture
X_OK	teste si le fichier existe et autorise l'exécution

Valeur de retour :

- **0** en cas de succès (toutes les permissions demandées sont autorisées),
⇒ possibilité de masque contenant plusieurs valeurs du mode (avec des **OU** (**|**) binaires)
- **-1** en cas d'échec (au moins une permission de **mode** est interdite ou d'autres erreurs se sont produites).

Rappel : Le test est effectué avec les **UID** et **GID** réels du processus appelant

Exemple : Test d'Accessibilité

Programme **access.c**

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Nbr arguments insuffisant !!\n"); return 0; }
    if (access(argv[1], F_OK) == 0) {
        printf("Le fichier \"%s\" existe !\n", argv[1]);
        if (access(argv[1], R_OK) == 0) printf("\tLa lecture est autorisée !\n");
        if (access(argv[1], W_OK) == 0) printf("\tL'écriture est autorisée !\n");
        if (access(argv[1], X_OK) == 0) printf("\tL'exécution est autorisée !\n");}
    else printf("Le fichier \"%s\" n'existe pas !\n", argv[1]);
    return 1;
}
```

```
#include <stdio.h>
#include <unistd.h>
```

Exécution de **access.c**

```
[student]$ gcc access.c -o access
[student]$ ls -l
-rwxrwxr-x 1 student student 7082  3 sept. 22:23 access
-rw-rw-r-- 1 student student  563  3 sept. 22:06 access.c
[student]$
[student]$ ./access access.c
Le fichier "access.c" existe !
    La lecture est autorisée !
    L'écriture est autorisée !
[student]$
```



Les Descripteurs Standards

Les fichiers en cours d'accès (ouverts) sont référencés par des descripteurs → table de descripteurs locale à chaque processus (u_ofile)

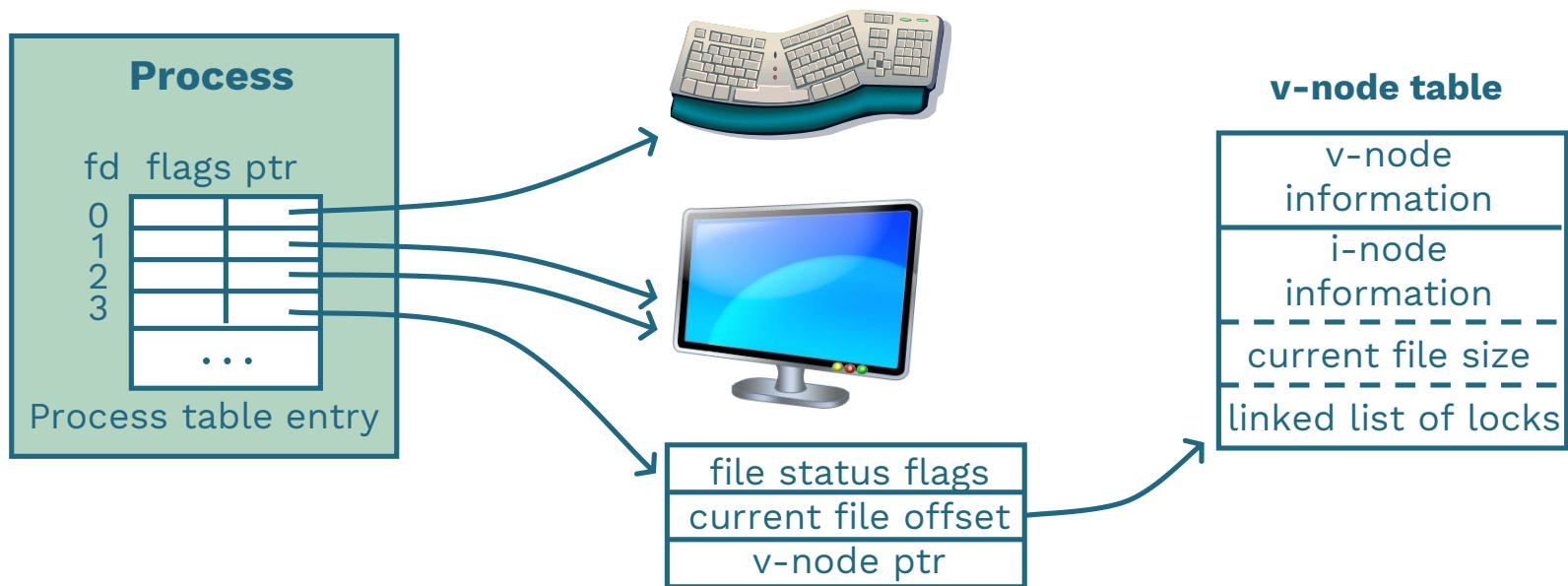
Descripteurs standards

- **0** (stdin) : flux correspondant à l'entrée standard
- **1** (stdout) : flux correspondant à la sortie standard
- **2** (stderr) : flux correspondant à la sortie d'erreur standard

Définition POSIX

STDIN_FILENO
STDOUT_FILENO
STDERR_FILENO

<unistd.h>



Exemple : Les Descripteurs Standards

Programme `desc_std.c`

```
int main() {
    int count = 128; char buf[count]; int nb;

    write(STDOUT_FILENO/* ou 1 */, "Entrez texte : \n", strlen("Entrez texte : \n"));
    if ((nb = read(STDIN_FILENO/* ou 0 */, buf, count)) != -1) {
        write(STDOUT_FILENO/* ou 1 */, "Texte lu = ", strlen("Texte lu = "));
        write(STDOUT_FILENO/* ou 1 */, buf, nb);
        return 0;
    } else {
        write(STDERR_FILENO/*ou 2*/, "Erreur Lecture !\n", strlen("Erreur Lecture !\n"));
        return -1;
    }
}
```

Exécution de `desc_std.c`

```
[student]$ gcc desc_std.c -o desc_std
[student]$ ./desc_std
Entrez texte :
Bonjour ! ←
Texte lu = Bonjour !
[student]$
```

Texte introduit par
l'utilisateur
avec retour à la ligne
suivante

Exécution identique si : STDIN_FILENO = 0
STDOUT_FILENO = 1
STDERR_FILENO = 2

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
```

Exécution de `desc_std.c` avec `STDIN_FILENO=3`

```
[if ((nb = read(3, buf, count)) != -1)]
```

```
[student]$ gcc desc_std.c -o desc_std
[student]$ ./desc_std
Entrez texte :
Erreur Lecture ! ←
Le descripteur '3'
n'est pas valide.
[student]$ ./desc_std 2> erreur.log
Entrez texte :
[student]$ cat erreur.log
Erreur Lecture !
[student]$
```

Ouverture/Création de Fichiers

Ouverture/création d'un fichier

```
#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);
```

Nom de fichier

Ouverture demandée en lecture seule : **O_RDONLY**
Ouverture demandée en écriture seule : **O_WRONLY**
Ouverture demandée en lecture/écriture : **O_RDWR**

Ajout d'autres flags à l'aide du **OU** binaire (||)

Demande de création du fichier s'il n'existe pas : **O_CREAT**
Demande de création du fichier, échec s'il existe déjà : **O_EXCL**
Demande d'ouverture du fichier en mode "ajout" : **O_APPEND**
Si fichier ordinaire et ouvert en écriture, il sera tronqué : **O_TRUNC**
Demande d'ouverture en mode non bloquant : **O_NONBLOCK**

...

Droits d'accès à attribuer au nouveau fichier : si flag **O_CREAT** utilisé.

Droits modifiés par le **umask**.

le plus petit descripteur de fichier non actuellement ouvert par le processus

Valeur de retour :

- Descripteur local de fichier (entier positif), en cas de succès,
- -1 en cas d'échec (et **errno** est modifiée en conséquence).

Exemple : Ouverture/Création de Fichiers

Programme **open.c**

```
int main(int argc, char **argv) {  
    int vr;  
    vr = open(argv[1], O_RDWR | O_CREAT | O_EXCL, S_IRWXU | S_IRWXG | S_IRWXO );  
    if (vr != -1) {  
        printf("Valeur de retour open = %d\n", vr);  
        return 0;  
    }  
    else {  
        perror("Erreur open ");  
        return -1;  
    }  
}
```

```
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <stdio.h>  
#include <stdlib.h>
```



Exécution de **open.c**

```
[student]$ ls -l  
-rwxrwxr-x 1 student student 6776  5 sept. 14:20 open  
-rw-rw-r-- 1 student student   363  5 sept. 14:19 open.c  
[student]$ ./open file1  
Erreur open : No such file or directory  
[student]$ ./open file1 (avec option ① uniquement)  
Valeur de retour open = 3  
[student]$ ls -l  
-r--r-s--- 1 student student     0  5 sept. 14:24 file1  
-rwxrwxr-x 1 student student 6776  5 sept. 14:24 open  
-rw-rw-r-- 1 student student   363  5 sept. 14:24 open.c  
[student]$
```

Droits imprévisibles car le champs mode n'a pas été utilisé.

Exemple : Ouverture/Création de Fichiers

Exécution de `open.c`

```
[student]$ ./open file1 (avec option ② uniquement)
```

```
Erreur open : Permission denied ← Le flag O_EXCL n'est pas pris en compte.
```

```
[student]$
```

```
[student]$ ./open file1 (avec options ① et ②)
```

```
Erreur open : File exists ← Le flag O_EXCL est pris en compte car utilisé avec le flag O_CREAT.
```

```
[student]$
```

```
[student]$ rm file1 ← Suppression du fichier file1.
```

```
rm : supprimer fichier vide (protégé en écriture) « file1 » ? y
```

```
[student]$ ls -l
```

```
total 12
```

```
-rwxrwxr-x 1 student student 6776 5 sept. 14:24 open
```

```
-rw-rw-r-- 1 student student 363 5 sept. 14:24 open.c
```

```
[student]$
```

```
[student]$ ./open file1 (avec options ①, ② et ③)
```

```
Valeur de retour open = 3
```

```
[student]$ ls -l
```

```
total 12
```

```
rwxrwxr-x 1 student student 0 5 sept. 14:47 file1
```

```
-rwxrwxr-x 1 student student 6776 5 sept. 14:46 open
```

```
-rw-rw-r-- 1 student student 359 5 sept. 14:46 open.c
```

```
[student]$ umask
```

```
0002
```

```
[student]
```

Attribution des bonnes permissions en fonction du **mode** spécifié et de l'**umask**.

L'**umask** interdit le droit d'écriture pour les utilisateurs other.

Lecture/Écriture de Fichiers

Lecture à partir d'un fichier

```
#include <unistd.h>
    ssize_t read(int fd, void *buf, size_t count);
```

tampon de stockage des octets lus

nombre d'octets à lire

descripteur du fichier à partir duquel se fait la lecture

Valeur de retour :

- nombre d'octets lus (**0** en cas de fin de fichier) en cas de succès, et avance la tête de lecture de ce nombre,
- **-1** en cas d'échec (et **errno** est modifiée en conséquence).

Écriture dans un fichier

```
#include <unistd.h>
    ssize_t write(int fd, const void *buf, size_t count);
```

zone mémoire à partir de laquelle se fait la lecture des octets à écrire dans le fichier

nombre d'octets à écrire

descripteur du fichier dans lequel se fait l'écriture

Valeur de retour :

- nombre d'octets écrits (**0** ⇒ aucune écriture) en cas de succès. L'écriture a lieu à la position courante et la tête d'écriture est avancée de ce nombre si l'appel **lseek** est possible pour le fichier,
- **-1** en cas d'échec (et **errno** est modifiée en conséquence).

Exemple : Lecture/Écriture de Fichiers

Programme `read_write.c`

```
int main(int argc, char **argv) {  
  
    int fd; // variable de sauvegarde du descripteur du fichier  
    char c; // variable caractère servant à la lecture/écriture d'un caractère à la fois  
  
    if ( argc < 2 ) {  
        write(STDOUT_FILENO, "Nbr arguments insuffisant !\n",  
              strlen("Nbr arguments insuffisant !\n"));  
        return -1;  
    }  
    if ( (fd = open(argv[1], O_RDWR | O_CREAT, 0744)) == -1) {  
        perror("Erreur Ouverture : ");  
        return -1;  
    }  
  
    write(fd, "Mon premier texte\n", strlen("Mon premier texte\n"));  
  
    write(fd, "écrit dans un fichier\n", strlen("écrit dans un fichier\n"));  
  
    while ( read(fd, &c, 1) != 0 )  
        write(STDOUT_FILENO, &c, 1);  
  
    return 0;  
}
```

```
#include <unistd.h>  
#include <string.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>
```

Ouverture en lecture/écriture (après éventuelle création) du fichier dont le nom est passé en paramètre sur la ligne de commande.

Lecture depuis le fichier et écriture sur la sortie standard, un caractère à chaque itération, jusqu'à atteindre la fin de fichier.

Écriture de deux chaînes de caractères dans le fichier.

Exemple : Lecture/Écriture de Fichiers

Exécution de `read_write.c`

```
[student]$ gcc read_write.c -o read_write
[student]$ ls -l
total 12
-rwxrwxr-x 1 student student 7055 6 sept. 15:33 read_write
-rw-rw-r-- 1 student student 608 6 sept. 15:33 read_write.c
[student]$ ./read_write file
[student]$ 
[student]$ ls -l
total 16
-rwxr--r-- 1 student student 40 6 sept. 15:41 file
-rwxrwxr-x 1 student student 7055 6 sept. 15:33 read_write
-rw-rw-r-- 1 student student 608 6 sept. 15:33 read_write.c
[student]$
```

(mettre en commentaire les 2 instructions `write(fd, ...)`)

```
[student]$
[student]$ gcc read_write.c -o read_write
[student]$ ./read_write file
Mon premier texte
ecrit dans un fichier
[student] cat file
```

```
[student] cat -n file
1 Mon premier texte
2 ecrit dans un fichier
[student]
```

Fermeture de Fichiers

Un fichier doit être fermé quand on n'y accède plus ou avant la terminaison du processus

```
#include <unistd.h>
    ↓
int close(int fd);
```

descripteur du fichier à fermer

La fermeture implique :

- le déréférencement du fichier, le descripteur peut être réutilisé
- les verrous du processus sur le fichier sont supprimés (quel que soit le descripteur qui fut utilisé pour poser ces verrous)
- si ce descripteur est le dernier qui référence un fichier supprimé, alors le fichier est effectivement supprimé.

Valeur de retour :

- **0** en cas de succès,
- **-1** en cas d'échec (et **errno** est modifiée en conséquence).

Pas de vérification de la valeur de retour ⇒ perte silencieuse possible de données (cas de systèmes de fichiers NFS ou d'utilisation de quotas de disques).

Une fermeture sans erreur \Rightarrow les données ont été écrites sur le disque (**fsync**).

Exemple : Fermeture de Fichiers

Programme **close.c**

```
int main(int argc, char **argv){  
    int fd; int i = 0;  
    fd = open(argv[1], O_RDWR | O_CREAT | O_APPEND, 0744)) == -1)  
    while ( i < 100 ){  
        write(fd, &i, sizeof(int)); i++;  
    }  
    if ( close(fd) != 0){  
        perror("Erreur Fermeture : "); return -1;  
    }  
    return 0;  
}
```

Ouverture en mode ajout.

```
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>
```

Écriture des 100 premiers entiers dans le fichier.

Test de fermeture du fichier.

Exécution de **close.c**

```
[student]$ ./close f_entiers  
[student]$ ls -nG  
total 12  
-rwxrwxr-x 1..6915 7 sept. 13:10 close  
-rw-rw-r-- 1.. 323 7 sept. 13:10 close.c  
-rwxr--r-- 1.. 400 7 sept. 13:02 f_entiers  
[student]$ ./close f_entiers  
[student]$ ls -nG  
total 12  
-rwxrwxr-x 1..6915 7 sept. 13:10 close  
-rw-rw-r-- 1.. 323 7 sept. 13:10 close.c  
-rwxr--r-- 1.. 800 7 sept. 13:04 f_entiers  
[student]$
```

1011

```
[student]$ cat f_entiers
```

!"#\$%&' ()*+,-.0123456789
:;<=>?@ABCDEFGHIJKLMNPQRSTUVWXYZ[\]^_`abc



!"#\$%&' ()*+,-.0123456789
:;<=>?@ABCDEFGHIJKLMNPQRSTUVWXYZ[\]^_`abc
[student] \$

Manipulation de l'*Offset*

Déplacement de la tête de lecture/écriture d'un fichier régulier

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

Descripteur du fichier

Position par rapport à whence.

SEEK_SET (0) : / au début du fichier
SEEK_CUR (1) : / à la position courante
SEEK_END (2) : / à la fin du fichier

`lseek` place la tête de lecture/écriture à la position **`offset`** en fonction de la directive **`whence`**.

Déplacement possible de la tête de lecture/écriture au-delà de la fin actuelle du fichier :

- la taille du fichier n'est pas modifiée
- si écriture à cet emplacement, une lecture de l'espace intermédiaire retournera des zéros ('\0')

Valeur de retour :

- le nouvel emplacement, mesuré en octets depuis le début du fichier, en cas de succès,
- **-1** en cas d'échec (et **`errno`** est modifiée en conséquence).

Exemple : Fermeture de Fichiers

Programme lseek.c

```
int main(int argc, char *argv[]) {
    int fd;
    if ( (fd=open(argv[1], O_RDWR)) == -1 ){
        perror("Erreur Ouverture\n");
        return -1;
    }
    printf("Nouvelle position = %d\n", lseek(fd, atoi(argv[2])*1024, SEEK_SET));
    printf("\tValeur Ecriture dans bloc %d = %d\n", atoi(argv[2]),
           write(fd, "debut bloc", strlen("debut bloc")));
    close(fd);
return 0;
}
```

Nom du fichier à ouvrir, passé en 1^{er} paramètre de la ligne de commande.

Déplacement de la tête de L/E en début du bloc de N° argv[2].

Exécution de lseek.c

```
[student]$ touch file
[student]$ ls -nG
-rw-rw-r-- 1 1011 0 8 sept. 12:03 file
[student]$ du file
0 file
[student]$ ~/bin/lseek file 0
Nouvelle position = 0
    Valeur Ecriture dans bloc 0 = 10
[student]$ ls -nG
-rw-rw-r-- 1 1011 10 8 sept. 12:04 file
[student]$
```

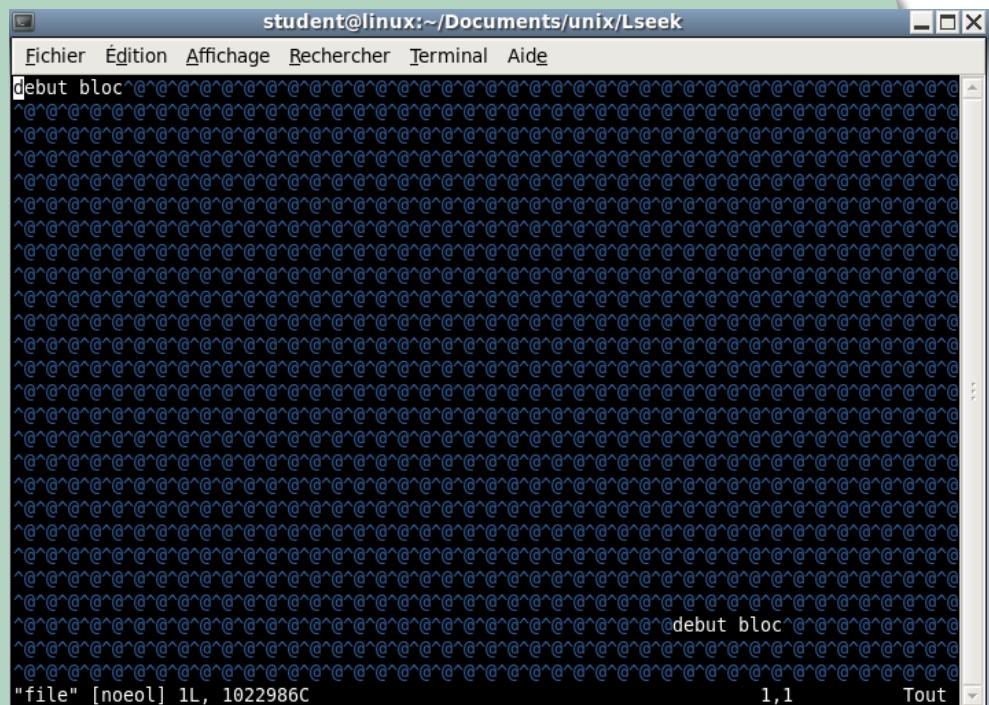
```
[student]$ du file
file
[student]$ ~/bin/lseek file 1
Nouvelle position = 1024
    Valeur Ecriture dans bloc 1 = 10
[student]$ ~/bin/lseek file 2
Nouvelle position = 2048
    Valeur Ecriture dans bloc 2 = 10
[student]$ ~/bin/lseek file 3
Nouvelle position = 3072
    Valeur Ecriture dans bloc 3 = 10
[student]$ ls -nG
-rw-rw-r-- 1 1011 3082 8 sept. 12:05 file
[student]$
```



Exemple : Fermeture de Fichiers

Suite de l'exécution de **lseek.c**

```
[student]$ du file  
4      file  
[student]$ ~/bin/lseek file 4  
Nouvelle position = 4096  
    Valeur Ecriture dans bloc 4 = 10  
[student]$ du file  
8      file  
[student]$ ~/bin/lseek file 999  
Nouvelle position = 1022976  
    Valeur Ecriture dans bloc 999 = 10  
[student]$ ls -nG  
-rw-rw-r-- 1 1011 1022986 8 sept. 12:05 file  
[student]$ du file  
12      file  
[student]$ cat file  
debut blocdebut blocdebut blocdebut blocdebut blocdebut bloc[student]
```



A screenshot of a terminal window titled "student@linux:~/Documents/unix/Lseek". The window contains the output of the lseek command. The first few lines show the creation of a file named "file" and seeking to position 4, writing the value 10. The next few lines show seeking to position 999 and writing the value 10. Finally, the "cat" command is run on the file, displaying the entire content of the file as "debut blocdebut blocdebut blocdebut blocdebut blocdebut bloc". The terminal window has a standard Linux-style interface with menu bars like "Fichier", "Édition", "Affichage", "Rechercher", "Terminal", and "Aide". The status bar at the bottom shows "1,1 Tout".

Duplication de Descripteurs

Duplication de descripteurs de fichiers ouverts

```
#include <unistd.h>
    ↓
    Descripteur de fichier à dupliquer
    ↓
int dup(int oldfd);
    Copie du descripteur de fichier à dupliquer
    ↓
int dup2(int oldfd, int newfd);
```

Les appels dup() et dup2() créent des copies du descripteur oldfd :

- **dup()** utilise le plus petit numéro non utilisé pour le nouveau descripteur
- **dup2()** force **newfd** (en le fermant éventuellement au préalable si en cours d'utilisation) à devenir une copie de **oldfd**

Les appels dup() et dup2() sont utilisés pour rediriger les entrées/sorties standards vers des fichiers ou vers des tubes.

Les deux descripteurs réfèrent le même fichier et peuvent être utilisés de manière interchangeable :

- ils partagent le même pointeur de position (tête de lecture/écriture) et les flags associés au fichier,
- ils ne partagent pas le flag close-on-exec (**O_CLOEXEC**), désactivé pour le nouveau descripteur.

Valeur de retour :

- le nouvel descripteur, en cas de succès,
- **-1** en cas d'échec (et **errno** est modifiée en conséquence).

Exemple : Fermeture de Fichiers

Programme dup_dup2.c

```
int main(void) {
    int erreur_fd;

    erreur_fd = open("erreur.log", O_WRONLY | O_CREAT | O_APPEND, 0666);
    close (STDERR_FILENO); ← Fermeture du descripteur STDERR_FILENO.

    if (dup(erreur_fd) == -1) { ← Duplication du descripteur erreur_fd en
        perror("dup "); return -1; choisissant le plus petit numéro non utilisé.
    }
    close(erreur_fd); ← Fermeture du descripteur erreur_fd, devenu inutile.

    if (open("fichier_inexistant", O_RDWR) == -1) ← On génère volontairement une erreur.
        perror("Erreur Ouverture ");

    if (dup2(STDOUT_FILENO, erreur_fd) == -1) { ← Duplication du descripteur
        perror("dup2 "); return -1; STDOUT_FILENO en une copie erreur_fd.
    }
    close (STDOUT_FILENO); ← Fermeture du descripteur STDOUT_FILENO.

    write(erreur_fd, "Fin du processus !\n", strlen("Fin du processus !\n")); ←
    return (0); ← Écriture via le descripteur erreur_fd qui est une copie de STDOUT_FILENO.
```

Exécution de dup_dup2.c

```
[student]$ ~/bin/dup_dup2
Fin du processus !
[student]$ ls -ng
-rw-rw-r-- 1 1011 47 8 sept. 14:04 erreur.log
[student]$ cat erreur.log
Erreur Ouverture : No such file or directory
[student]$
```



Le Verrouillage - Caractéristiques

Le verrouillage s'applique au fichier et non à l'un de ses descripteurs

- un verrou posé sur un fichier via un descripteur est "visible" via un autre descripteur de ce même fichier.

Un verrou est associé à un processus et à un fichier

- seul le propriétaire du verrou peut le modifier ou le supprimer,
- lorsqu'un processus se termine, tous les verrous qu'il détient sont supprimés,
- chaque fois qu'un descripteur est fermé par un processus, tous les verrous sur le fichier référencé par le descripteur pour le processus donné sont supprimés.

Héritage des verrous

- les verrous ne sont jamais hérités par les processus fils à travers un **fork()**,
- les verrous peuvent être hérités par un nouveau programme au travers d'un **exec()** (cas de **SVR4** et **4.3+BSD**, **POSIX.1** ne le requiert pas).

La portée du verrou

Le Verrouillage - Compatibilité

Un verrou a un type

- **partagé** (*shared*) : plusieurs verrous de ce type peuvent cohabiter (avoir des portées sur une région commune)
- **exclusif** (*exclusive*) : un verrou de ce type ne peut cohabiter avec aucun autre verrou (exclusif ou partagé)

		Requête pour verrou partagé verrou exclusif	
La région possède déjà	aucun verrou	OK	OK
	un ou plusieurs verrous partagés	OK	interdit
	un verrou exclusif	interdit	interdit

Compatibilité entre les types de verrou

Le Verrouillage : Le Mode Opératoire

Un verrou a un mode opératoire

- **consultatif** (*advisory*) : ce mode opératoire n'a pas d'influence sur les entrées/sorties
- **impératif** (*mandatory*) : ce mode influence les entrées/sorties

Descripteur bloquant, tentative de		Descripteur non bloquant, tentative de	
lecture	écriture	lecture	écriture
OK bloquée	bloquée bloquée	OK EAGAIN	EAGAIN EAGAIN
<i>un verrou partagé existe sur la région</i>			
<i>un verrou exclusif existe sur la région</i>			

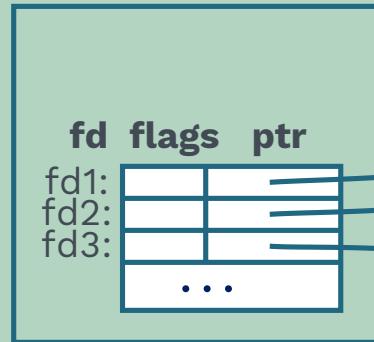
Effets du verrouillage impératif sur les lectures et écritures des autres processus

L'indication du mode opératoire est mémorisée dans les i-nœuds (setgid bit)

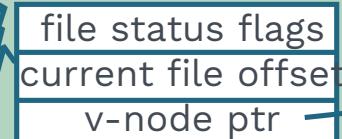
- Pour un fichier donné, soit tous les verrous posés sont de type consultatif, soit ils sont tous de type impératif.

Implémentation 4.3+BSD

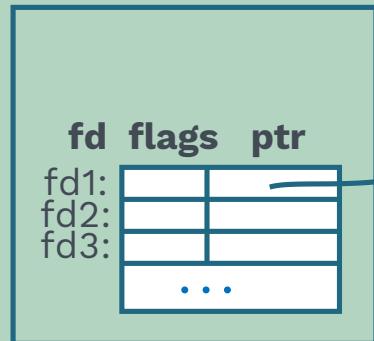
parent process table entry



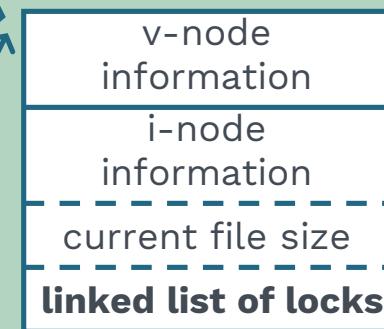
file table



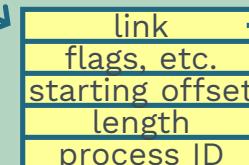
child process table entry



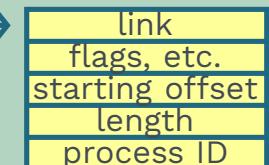
v-node table



struct flock



struct flock



Les Différentes Formes de Verrouillage

System	Advisory	Mandatory	fcntl	lockf	flock
POSIX.1	▪		▪		
XPG3	▪		▪		
SVR2	▪		▪	▪	
SVR3, SVR4	▪	▪	▪	▪	
4.3BSD	▪				▪
4.3BSD Reno	▪		▪		▪

Advanced Programming in the UNIX Environment, W. Richard Stevens, Addison-Wesley Professional Computing Series

- **flock** : le verrouillage s'applique à la totalité du fichier
- Verrouillage plus fin, voir **fcntl()**

Modification des Caractéristiques d'un Fichier

Réalisation d'opérations diverses sur un descripteur de fichier

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */);
```

↑
paramètre dont le type dépend de cmd ou void si non nécessaire

F_DUPFD : trouver le plus petit numéro de descripteur libre \geq à **arg**

F_GETFD, **F_SETFD** : lecture/modification attributs du descripteur (**FD_CLOEXEC**)

F_GETFL, **F_SETFL** : lecture/modification attributs d'état du fichier (**O_APPEND**, **O_NONBLOCK**, ...)

F_GETLK, **F_SETLK**, **F_SETLKW** : gestion des verrous

F_GETOWN, **F_SETOWN** : gestion des signaux de disponibilité d'E/S (socket)

La primitive **fcntl** permet, en fonction de la valeur du paramètre **cmd**, de réaliser un certain nombre d'opérations sur le descripteur de fichier **fd**.

Valeur de cmd	Type de arg	Valeur de retour
F_DUPFD	long	le nouveau descripteur
F_GETFD , F_GETFL	void	la valeur des attributs
F_GETOWN	void	le propriétaire du descripteur de fichier
F_SETOWN	long	zéro
F_GETLK , F_SETLK , F_SETLKW	struct flock *	zéro
Toutes (presque) les autres commandes	void	zéro

Verrouillage par la Primitive fcntl()

La primitive **fcntl()** permet, via la pose de verrous, de gérer les accès concurrents (sur des régions/portions de fichiers)

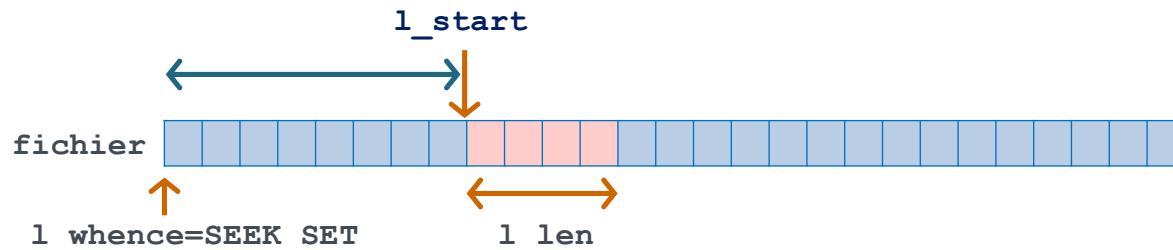
- Le troisième paramètre de la primitive est un pointeur sur une structure **flock**.

```
struct flock {           // défini dans fcntl.h
    short l_type;        // type de verrou : F_RDLCK partagé, F_WRLCK exclusif,
    short l_whence;      //          // F_UNLCK déverrouillage
    short l_start;        // position (idem lseek())
    short l_len;          // position relative de début par rapport à l_whence
    int l_pid;            // nombre d'octets verrouillés, si 0 => jusqu'à fin du fichier
                          // PID du processus auquel appartient le verrou. Retourné
                          // par F_GETLK
}
```

indiquent la région à verrouiller →

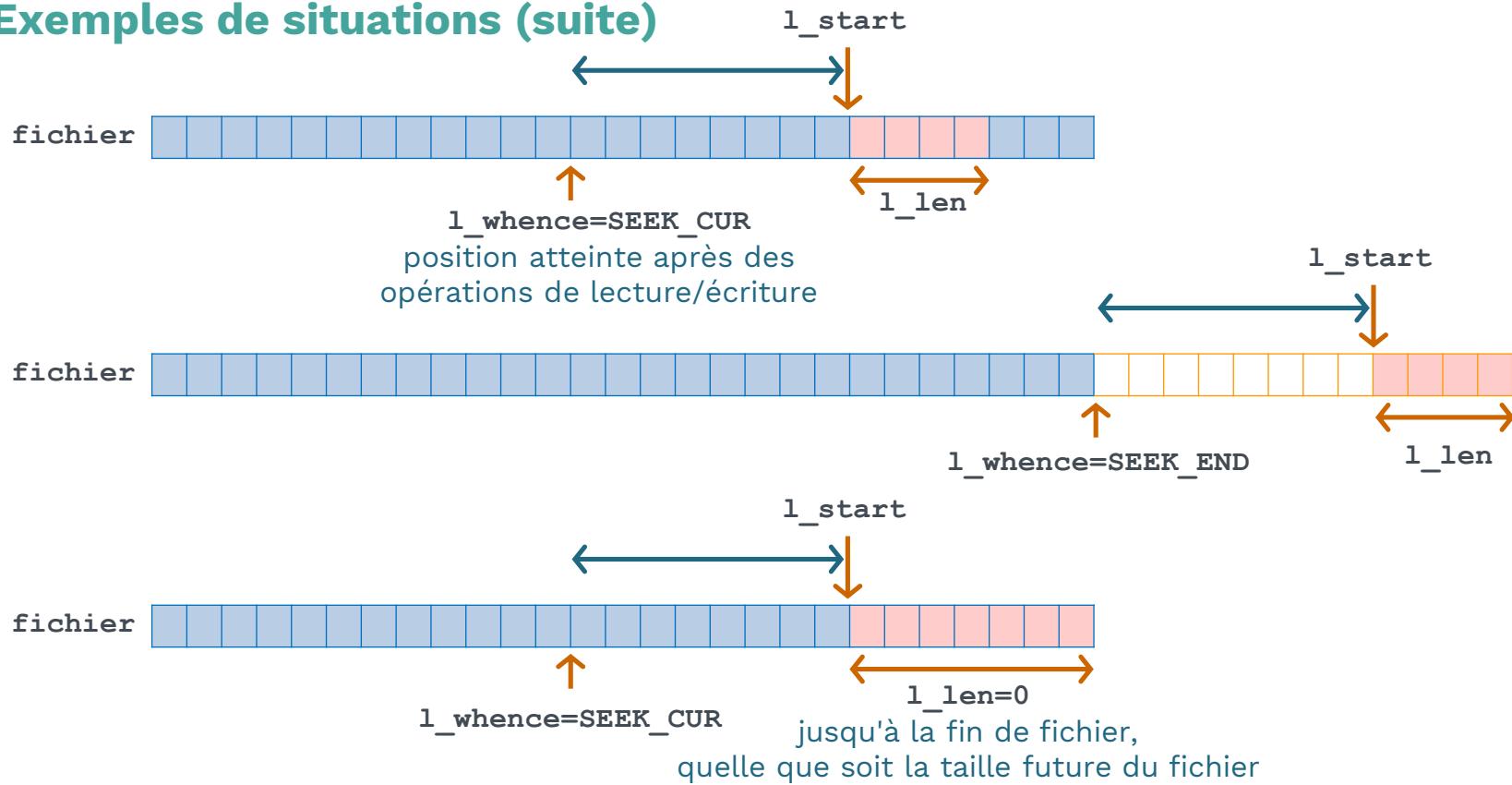
Exemples de situations

- l_start = 8, l_len = 4**



Verrouillage par la Primitive fcntl()

Exemples de situations (suite)



- **POSIX** permet à une implémentation de définir une valeur négative pour `l_len`.
- **Un processus donné ne peut détenir qu'un seul verrou sur une région d'un fichier.**
- **Les situations d'interblocage sont détectées.**

Exemple de Pose de Verrous

Programme verrou_partage.c

```
int main(int argc, char * argv[]) {
    int fd;
    struct flock verrou = { F_RDLCK, SEEK_SET, 0, 0, 0 };
    printf("Descripteur Ouverture = %d\n", fd = open(argv[1], O_RDONLY));

    if (fcntl(fd, F_SETLK, &verrou) == -1) {
        perror("Erreur pose verrou ");
        return -1;
    }
    printf("Pose reussie de verrou partage !\n");
    pause();
    return 0;
}
```

The diagram illustrates the fields of a `flock` structure and their assignments in the code. The `flock` structure is defined as `{ l_start, l_type, l_whence, l_len }`. Arrows point from each field name to its respective assignment in the code:

- `l_start` points to `0`
- `l_type` points to `F_RDLCK`
- `l_whence` points to `SEEK_SET`
- `l_len` points to `0`

A callout box labeled "fonction du verrou à poser" points to the `fcntl` call.

A dashed box labeled "pose du verrou partagé" encloses the error handling code within the `if` block.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
```

Programme verrou_exclusif.c

```
int main(int argc, char * argv[]) {
    int fd;
    struct flock verrou = { F_WRLCK, SEEK_SET, 0, 0, 0 };
    printf("Descripteur Ouverture = %d\n", fd = open(argv[1], O_WRONLY));

    while (fcntl(fd, F_SETLK, &verrou) == -1) {
        perror("Erreur pose verrou ");
        sleep(1);
    }
    printf("Pose reussie de verrou exclusif !\n");
    pause();
    return 0;
}
```

The diagram illustrates the fields of a `flock` structure and their assignments in the code. The `flock` structure is defined as `{ l_start, l_type, l_whence, l_len }`. Arrows point from each field name to its respective assignment in the code:

- `l_start` points to `0`
- `l_type` points to `F_WRLCK`
- `l_whence` points to `SEEK_SET`
- `l_len` points to `0`

A callout box labeled "pose du verrou exclusif" points to the `while` loop.

Exemple de Pose de Verrous

Exécution de **verrou_partage.c**

```
[student]$ ls -nG  
-rw-rw-r-- 1 1011 87 8 sept. 22:08 file  
[student]$ ./verrou_partage file  
Descripteur Ouverture = 3  
Pose réussie de verrou partage !
```

```
^C  
[student]$
```

Exécution de **verrou_exclusif.c**

```
[student]$  
[student]$  
[student]$ ./verrou_exclusif file  
Descripteur Ouverture = 3  
PB Pose : Resource temporarily unavailable  
Pose réussie de verrou exclusif !
```

```
^C  
[student]$
```

↑
Scénario 1 : **verrou_partage** est exécuté en premier

↓
Scénario 2 : **verrou_exclusif** est exécuté en premier

```
[student]$  
[student]$ ./verrou_partage file  
Descripteur Ouverture = 3  
PB Pose : Resource temporarily unavailable  
[student]$
```

```
[student]$ ./verrou_exclusif file  
Descripteur Ouverture = 3  
Pose réussie de verrou exclusif !  
^C  
[student]$
```

Exemple de Pose de Verrous

Programme verrou_partage.c (utilisation de SETLKW)

```
int main(int argc, char * argv[]) {
    int fd; struct flock verrou = { F_RDLCK, SEEK_SET, 0, 0, 0 };
    printf("Descripteur Ouverture = %d\n", fd = open(argv[1], O_RDONLY));
    if (fcntl(fd, F_SETLKW, &verrou) == -1) {
        perror("Erreur pose verrou ");
        return -1;
    }
    printf("Pose reussie de verrou partage !\n");
    pause();
}
```

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
```

Programme verrou_exclusif.c (sans attente active, utilisation de SETLKW)

```
int main(int argc, char * argv[]) {
    int fd; struct flock verrou = { F_WRLCK, SEEK_SET, 0, 0, 0 };
    printf("Descripteur Ouverture = %d\n", fd = open(argv[1], O_WRONLY));
    if (fcntl(fd, F_SETLKW, &verrou) == -1) {
        perror("Erreur pose verrou ");
        return -1;
    }
    printf("Pose reussie de verrou exclusif !\n");
    pause();
}
```

```
[student]$ ./verrou_partage file
Descripteur Ouverture = 3
Pose reussie de verrou partage !
^C
[student]$
```

```
[student]$
[student]$ ./verrou_exclusif file
Descripteur Ouverture = 3
Pose reussie de verrou exclusif !
^C
[student]$
```

Algorithme de Lecture/Écriture

depuis/dans un fichier ordinaire avec la primitive read/write

Il n'existe pas de verrou exclusif impératif sur le fichier dans la portée de la lecture

- Si non fin de fichier, lecture nombre spécifié de caractères ou jusqu'à la fin du fichier
- Si fin de fichier, aucune lecture (0 en retour)

Il existe un verrou exclusif impératif sur le fichier dans la portée de la lecture

- Si mode de lecture bloquant (O_NONBLOCK et O_NDELAY non positionnés), processus bloqué jusqu'à suppression du verrou ou réception signal
- Si mode de lecture non bloquant (O_NONBLOCK ou O_NDELAY positionnés), retour immédiat et pas de lecture (-1 en retour et errno = EAGAIN)

Il n'existe pas de verrou impératif (partagé ou exclusif) sur le fichier dans la portée de l'écriture

- Écriture dans le fichier (nombre caractères écrits en retour)
- Si l'indicateur O_SYNC non positionné alors écriture dans le cache du noyau, sinon écriture sur le disque

Il existe un verrou impératif (partagé ou exclusif) sur le fichier dans la portée de l'écriture

- Si mode d'écriture bloquant (O_NONBLOCK et O_NDELAY non positionnés), processus bloqué jusqu'à suppression du verrou ou réception signal
- Si mode d'écriture non bloquant (O_NONBLOCK ou O_NDELAY positionnés), retour immédiat et pas d'écriture (-1 en retour et errno = EAGAIN)

Modification de l'Attribut d'un Descripteur de Fichier

Fermeture d'un descripteur lors d'un recouvrement (primitives exec)

- La valeur, par défaut, de l'indication (**FD_CLOEXEC**) de fermeture automatique ou de maintien de l'ouverture d'un descripteur lors d'un recouvrement correspond à un maintien de l'ouverture.
- Cet état peut être modifié après coup (si non spécifié à l'ouverture du fichier)

Programme **coe_proc1.c**

```
int main(int argc, char **argv) {  
    int fd;  
    printf("Desc. Procl = %d\n", fd = open(argv[1], O_WRONLY | O_CREAT | O_APPEND, 0666));  
    write(fd, "Ecrit par Proc1\n", strlen("Ecrit par Proc1\n"));  
    if (execl("coe_proc2", "coe_proc2", NULL) == -1) {  
        perror("Erreur Exec Proc1");  
        return -1;  
    }  
    return 0;  
}
```

```
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <unistd.h>  
#include <string.h>  
#include <stdio.h>
```

Programme **coe_proc2.c**

```
int main(int argc, char **argv) {  
    if (write(3, "Ecrit par Proc2\n", strlen("Ecrit par Proc2\n")))  
        perror("Erreur Ecriture Proc2 "); return -1;  
    }  
    close(3);  
    return 0;  
}
```

```
#include <unistd.h>  
#include <string.h>
```



Exemple : Modification de l'Attribut d'un fd

Exécution de `coe_proc1.c`

```
[student]$ gcc coe_proc2.c -o coe_proc2
[student]$ gcc coe_proc1.c -o coe_proc1
[student]$ ./coe_proc1 file
Desc. Proc1 = 3
[student]ls -l
-rw-rw-r-- 1 student student      32   9 sept. 09:42 file
[student]$ cat file
Ecrit par Proc1
Ecrit par Proc2
[student]
```

Modification de `coe_proc1.c` (positionnement de `FD_CLOEXEC` du descripteur après ouverture du fichier)

```
...
    write(fd, "Ecrit par Proc1\n", strlen("Ecrit par Proc1\n"));

Partie ajoutée à coe_proc1.c
    int attr_fd;
    attr_fd = fcntl(fd, F_GETFD);
    attr_fd = attr_fd | FD_CLOEXEC;
    fcntl(fd, F_SETFD, attr_fd);
    if (execl("coe_proc2", "coe_proc2", NULL) == -1) {
...

```

récupère les attributs du descripteur fd

positionnement de l'indicateur

maj des nouveaux attributs

Exemple : Modification de l'Attribut d'un fd

Exécution de **coe_proc1.c** (après modification)

```
[student]$ gcc coe_proc1.c -o coe_proc1
[student]$ ./coe_proc1 file
Desc. Procl = 3
Erreur Ecriture Proc2 : Bad file descriptor
[student]$ cat file
Ecrit par Procl
[student]
```

L'attribut **FD_CLOEXEC** du descripteur aurait pu être positionné à l'ouverture du fichier (primitives **open()**)

Modification de coe_proc1.c (positionnement de **O_CLOEXEC** du descripteur à l'ouverture du fichier)

```
...
printf("Desc. Procl = %d\n", fd = open(argv[1], O_WRONLY | O_CREAT | O_APPEND
    | O_CLOEXEC, 0666));
```

Exécution de **coe_proc1.c** (après modification)

```
[student]$ gcc coe_proc1.c -o coe_proc1
[student]$ ./coe_proc1 file
Desc. Procl = 3
Erreur Ecriture Proc2 : Bad file descriptor
[student]$ cat file
Ecrit par Procl
[student]
```

Modification des Attributs d'États d'un Fichier

Rendre les entrées/sorties non bloquantes

- La valeur, par défaut, de l'indication (**O_NDELAY** ou **O_NONBLOCK** - POSIX) d'entrée/sortie en mode bloquant ou non bloquant correspond à un mode bloquant.
- Cet état peut être modifié après coup (si non spécifié à l'ouverture du fichier)

Programme **es_block.c**

```
int main() {
    int n; char buf[128];

    while (1) {
        if ((n = read(STDIN_FILENO, buf, sizeof(buf))) != -1)
            write(STDOUT_FILENO, buf, n);
        else
            perror("Erreur lecture ");
    }
}
```

```
#include <fcntl.h>
```

Exécution de **es_block.c**

```
[student]$ gcc es_block.c -o es_block
[student]$ ./es_block
Bonjour
Bonjour
Au revoir
Au revoir
^C
[student]$
```

introduits par
l'utilisateur

affichés par le processus

À chaque appel à **read()**,
le processus est en
attente d'introduction
de données au clavier.

Exemple : Modification des Attributs d'États

Programme **es_nonblock.c** (modification de **es_block.c** pour des entrées/sorties non bloquantes)

```
int main() {
    int n; char buf[128];
    int mode_courant, mode_non_bloquant;
    mode_courant = fcntl(STDIN_FILENO, F_GETFL);
    mode_non_bloquant = mode_courant | O_NONBLOCK;
    fcntl(STDIN_FILENO, F_SETFL, mode_non_bloquant); ←
    while (1) {
        if ((n = read(STDIN_FILENO, buf, sizeof(buf))) != -1)
            write(STDOUT_FILENO, buf, n);
        else {
            perror("Erreur lecture ");
            sleep(10);
        }
    }
}
```

```
#include <unistd.h>
#include <fcntl.h>
```

récupération de l'état de **STDIN_FILENO**

modification du mode des entrées standards

forcer les entrées standards à être réalisées en mode non bloquant

Exécution de **es_nonblock.c**

```
[student]$ ./es_nonblock
Erreur lecture : Resource temporarily unavailable
Erreur lecture : Resource temporarily unavailable
Bonjour : ← - laps de temps entre l'introduction par l'utilisateur et l'affichage par le processus (sleep())
Bonjour :
Erreur lecture : Resource temporarily unavailable
Au revoir
Au revoir
Erreur lecture : Resource temporarily unavailable
^C
[student]
```

À chaque appel à **read()**, le processus passe à l'instruction suivante si aucune donnée n'est disponible (pas d'attente).

Modification des Attributs d'États d'un Fichier

Forcer (après coup) les écritures à se faire en fin de fichier

- La valeur, par défaut, de l'indication (**O_APPEND**) d'écriture en mode ajout (en fin de fichier) ou non correspond à ce dernier mode.
- Cet état peut être modifié après coup (si non spécifié à l'ouverture du fichier)

Programme **ecriture.c**

```
int main(int argc, char **argv) {
    int fd;

    fd = open(argv[1], O_RDWR);
    write(fd, "22222", strlen("22222"));
    write(fd, "22222", strlen("22222"));
}
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
```

Exécution de **ecriture.c**

```
[student]$ cat file
1111111111[student]$ gcc ecriture.c -o ecriture
[student]$ ./ecriture file
[student]$ cat file
2222222222[student]$
```

↑
**Les données contenues dans le fichier file sont "écrasées" par les nouvelles données écrites par le processus.
Le fichier ne change pas de taille.**

Exemple : Modification des Attributs d'États

Programme **ecriture_ajout.c** (modification de **ecriture.c** pour une écriture dans le fichier en mode ajout (à partir de la fin du fichier))

```
int main(int argc, char **argv) {
    int fd;
    int ajout_pos_courante, ajout_fin;

    fd = open(argv[1], O_RDWR);

    ajout_pos_courante = fcntl(fd, F_GETFL);
    ajout_fin = ajout_pos_courante | O_APPEND; ← récupération du mode d'ouverture
    fcntl(fd, F_SETFL, ajout_fin); ← modification du mode d'écriture

    write(fd, "22222", strlen("22222"));
    write(fd, "22222", strlen("22222"));
}
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
```

← forcer les écritures à être réalisées
en fin de fichier

Exécution de **ecriture_ajout.c**

```
[student]$ cat file
1111111111[student]$ gcc écriture_ajout.c -o écriture_ajout
[student]$ ./écriture_ajout file
[student]$ cat file
11111111112222222222 [student]$
```

Les nouvelles données sont écrites à partir de la fin
du fichier, préservant ainsi celles déjà contenues.

Exemple : Modification des Attributs d'États

Forcer les écritures en fin de fichier peut se faire à tout moment et n'importe où.

Programme **écriture_ajout.c** (nouvelle modification)

```
int main(int argc, char **argv) {
    int fd;
    int ajout_pos_courante, ajout_fin;
    fd = open(argv[1], O_RDWR);
    write(fd, "22222", strlen("22222")); ← première écriture
    {
        ajout_pos_courante = fcntl(fd, F_GETFL);
        ajout_fin = ajout_pos_courante | O_APPEND;
        fcntl(fd, F_SETFL, ajout_fin);
    }
    write(fd, "22222", strlen("22222")); ← deuxième écriture
}
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
```

Modifications
des attributs

Exécution de **écriture_ajout.c**

```
[student]$ cat file
1111111111[student]$ ./écriture_ajout file
[student]$ cat file
22222111122222[student]$
```

Modification des attributs d'états entre les deux écritures.

L'attribut **O_APPEND** du descripteur aurait pu être positionné à l'ouverture du fichier.

Modification de **écriture_ajout.c** (positionnement de **O_APPEND** à l'ouverture) : `fd = open(argv[1], O_RDWR | O_APPEND);`

Ouverture/Fermeture des Répertoires

Ouverture d'un répertoire

```
#include <dirent.h>
DIR *opendir(const char *pathname);
```

nom du répertoire



Ouvre un flux répertoire correspondant au répertoire référencé par **pathname** et renvoie un pointeur sur ce flux.

Valeur de retour :

- un pointeur vers le flux répertoire, en cas de succès,
- **NULL** en cas d'échec (et **errno** est modifiée en conséquence).

Fermeture d'un répertoire

```
#include <dirent.h>
int closedir (DIR *dirp);
```

pointeur rendu par opendir() lors de l'ouverture



Libère les ressources allouées lors de l'appel à opendir().

Valeur de retour :

- **0**, en cas de succès,
- **-1** en cas d'échec (et **errno** est modifiée en conséquence).

Manipulation des Répertoires

Lecture d'une entrée de répertoire

```
#include <dirent.h>
```

pointeur rendu par `opendir()` lors de l'ouverture

```
    struct dirent *readdir(DIR *dirp);
```

Lecture de l'entrée suivante dans le répertoire référencé par dirp.

Valeur de retour :

- un pointeur vers une structure **dirent**, en cas de succès,
- **NULL** en fin de fichier ou en cas d'échec (et **errno** est modifiée en conséquence).

```
struct dirent {  
    ino_t d_ino;           // i-node number  
    off_t d_off;          // offset to this dirent  
    unsigned short d_reclen; // length of this d_name  
    char d_name [NAME_MAX+1]; // null-terminated filename  
    unsigned short d_type; // type  
}
```

Repositionnement du pointeur de lecture

```
#include <dirent.h>
```

pointeur rendu par `opendir()` lors de l'ouverture

```
void rewinddir(DIR *dirp);
```

- Ne renvoie pas de valeur.

Exemple de Manipulation de Répertoires

Programme **affiche_rep.c**

```
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>

int main(int argc, char **argv) {
    DIR *dirp;
    struct dirent *dire;

    if ((dirp = opendir(argv[1])) == NULL) {
        perror("Erreur Ouverture ");
        return -1;
    }
    while ((dire = readdir(dirp)) != NULL) {
        if (strcmp(dire->d_name, ".") == 0 || strcmp(dire->d_name, "..") == 0)
            continue;
        printf("fichier trouvé : %s\t de type : %d\n", dire->d_name, dire->d_type);
    }
    rewinddir(dirp); ← réinitialise la position du pointeur du flux dirp au début du répertoire
    while ((dire = readdir(dirp)) != NULL) {
        if (strcmp(dire->d_name, ".") == 0 || strcmp(dire->d_name, "..") == 0)
            continue;
        printf("fichier trouvé : %s\t de type : %d\n", dire->d_name, dire->d_type);
    }
    closedir(dirp);
    return 0;
}
```



Exemple de Manipulation de Répertoires

Exécution de **affiche_rep.c**

```
[student]$ ls -l ../../TESTS/
total 8
drwxrwxr-- 2 root      root      4096  2 sept. 15:10 DONNEES
-rw-rw-r-- 1 student   student    71   2 août   15:06 fichier
lrwxrwxrwx 1 student   student     4   2 août   15:04 file -> pipe
crw-rw---- 1 root      lp       6, 0  2 août   15:00 lp0
prw-rw-r-- 1 student   student    0   2 août   14:56 pipe
brw-rw---- 1 root      disk     8, 0  2 août   15:05 sda
[student]$ gcc affiche_rep.c -o affiche_rep
[student]$ ./affiche_rep ../../TESTS/
fichier trouvé : DONNEES          de type : 4
fichier trouvé : pipe            de type : 1
fichier trouvé : file            de type : 10
fichier trouvé : lp0             de type : 2
fichier trouvé : fichier         de type : 8
fichier trouvé : sda             de type : 6
fichier trouvé : DONNEES          de type : 4
fichier trouvé : pipe            de type : 1
fichier trouvé : file            de type : 10
fichier trouvé : lp0             de type : 2
fichier trouvé : fichier         de type : 8
fichier trouvé : sda             de type : 6
[student]$
```

Création et Suppression de Répertoires

Création de répertoire

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

Diagramme montrant les paramètres de la fonction mkdir :
- Le paramètre **pathname** est encadré dans un rectangle bleu et pointé par une flèche vers le texte "nom du répertoire".
- Le paramètre **mode** est encadré dans un rectangle bleu et pointé par une flèche vers le texte "permissions à appliquer au répertoire (modifiées par umask du processus)".

Crée un nouveau répertoire vide nommé **pathname**. Les entrées "." et ".." sont automatiquement créées.

Valeur de retour :

- **0**, en cas de succès,
- **-1** en cas d'échec (et **errno** est modifiée en conséquence).

Suppression de répertoire

```
#include <unistd.h>
int rmdir(const char *pathname);
```

Diagramme montrant le paramètre de la fonction rmdir :
- Le paramètre **pathname** est encadré dans un rectangle bleu et pointé par une flèche vers le texte "nom du répertoire".

Le répertoire doit être vide.

Valeur de retour :

- **0**, en cas de succès,
- **-1** en cas d'échec (et **errno** est modifiée en conséquence).

Propriété des Nouveaux Fichiers et Répertoires

Les règles de la propriété d'un nouveau répertoire sont identiques à celles de la propriété d'un nouveau fichier.

- L'identifiant utilisateur (**UID**) d'un nouveau fichier est établi à l'identifiant utilisateur effectif (**EUID**) du processus.
- POSIX.1 permet à toute implémentation de choisir l'une des deux options suivantes pour déterminer l'identifiant de groupe (**GID**) d'un nouveau fichier :
 - l'identifiant de groupe d'un nouveau fichier peut être l'identifiant de groupe effectif (**EGID**) du processus,
 - l'identifiant de groupe d'un nouveau fichier peut être l'identifiant de groupe (**GID**) du répertoire dans lequel le fichier est créé.

SVR4

set-group-ID du répertoire parent positionné
sinon

→ ID de groupe du répertoire parent
→ ID de groupe effectif du processus

(Héritage du set-gid-bit du parent pour le nouveau répertoire suite à un mkdir(...))

4.3+BSD

Utilise toujours l'identifiant de groupe du répertoire parent



Exemple Creation/Suppression de Rertoires

Programme **cree_rep.c**

```
int main(int argc, char **argv) {
    DIR *dirp;
    struct dirent *dire;

    if (mkdir(argv[1], 0777) == -1) {
        perror("Erreur Creation ");
        return -1;
    }
    if ((dirp = opendir(argv[1])) == NULL) {
        perror("Erreur Ouverture ");
        return -1;
    }
    while ((dire = readdir(dirp)) != NULL) {
        printf("fichier trouve : %s\t de type : %d\n", dire->d_name, dire->d_type);
    }
    closedir(dirp);
    return 0;
}
```

```
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <dirent.h>
#include <stdio.h>
```

Programme **suppr_rep.c**

```
int main(int argc, char **argv) {
    if (rmdir(argv[1]) == -1) {
        perror("Erreur Suppression ");
        return -1;
    }
    return 0;
}
```



Exemple Creation/Suppression de Rertoires

Execution de **cree_rep.c** et **suppr_rep.c**

```
[student]$ ls -ld REPERTOIRE
ls: impossible d'accéder à REPERTOIRE: Aucun fichier ou dossier de ce type
[student]$ gcc cree_rep.c -o cree_rep
[student]$ ./cree_rep REPERTOIRE
fichier trouvé : .. de type : 4
fichier trouvé : . de type : 4
[student]$ ls -ld REPERTOIRE
drwxrwxr-x 2 student student 4096 9 sept. 18:00 REPERTOIRE
[student]$ touch REPERTOIRE/file
[student]$ ls -l REPERTOIRE
total 0
-rw-rw-r-- 1 student student 0 9 sept. 18:00 file
[student]$ gcc suppr_rep.c -o suppr_rep
[student]$ ./suppr_rep REPERTOIRE/
Erreur Suppression : Directory not empty
[student]$ \rm REPERTOIRE/file
[student]$ ls -l REPERTOIRE
total 0
[student]$ ./suppr_rep REPERTOIRE/
[student]$ ls -ld REPERTOIRE
ls: impossible d'accéder à REPERTOIRE: Aucun fichier ou dossier de ce type
[student]$
```