Le FileSystem

Table des matiéres

I) GÉNÉRALITÉS

Toutes les opérations d'entrée-sortie sont réalisées en langage C via des appels à des fonctions et sont considérés comme mettant en œuvre des fichiers.
Le système de gestion des entrées-sorties sur fichiers est lui-même subdivisé en deux sous-systèmes bien distincts :

- ✓ un sous-système d'entrée-sortie en mode caractère,
- ✓ un sous-système d'entrée-sortie en mode bloc.

On peut réaliser ces opérations en invoquant :

- ✓ des fonctions système, pour réaliser des opérations dites de bas niveau,
- ✓ des fonctions de la bibliothèque C, pour réaliser des opérations dites de haut niveau.

Dans ce chapitre, nous ne nous intéresserons qu'aux fonctions de bas niveau, les autres étant étudiées lors de l'étude du langage C.

On distingue les appels système permettant de manipuler les références des fichiers et celles permettant d'agir sur leur contenu.

Rappel : Sous UNIX, un fichier est constitué par une suite de caractères adressables sans aucune structure interne. La taille du fichier correspond exactement au nombre de caractères qu'il contient.

II) MANIPULATION DE FICHIERS

La structure "stat"La manipulation "externe" des fichiers (sans préoccupation du contenu) se fait en utilisant son inode. La taille de cet inode est de 64 octets. Il contient divers renseignements sur le fichier en lui-même tels que ses droits d'accès, ses dates de modification, sa taille et ses adresses de stockage sur le disque. La structure ***stat*** définie dans ***sys/stat.h*** permet d'accéder à ces informations.

```
struct stat {
   dev_t st_dev;              /* device de montage */
   ino_t st_ino;              /* numéro de l'inode */
   ushort st_mode;            /* mode de l'inode (type et droits) */
   short st_nlink;            /* nombre de liens */
   ushort st_uid;             /* numéro du propriétaire */
   ushort st_gid;             /* numéro du groupe */
   dev_t st_rdev;             /* noeuds majeur/mineur */
   off_t st_size;             /* taille du fichier en octet */
   blksize_t st_blksize;      /* taille d'un bloc pour le filesystem */
   blkcnt_t  st_blocks;       /* nombre de blocs de 512o alloués */
   time_t st_atime;           /* date du dernier accès */
   time_t st_mtime;           /* date de la dernière modification */
   time_t st_ctime;           /* date de création/modif. d'inode */
};
```

Explication des éléments :

✓ dev_t st_dev : concaténation sur un entier (2 octets) des nœuds majeurs et mineur du device auquel appartient le fichier

✓ ino_t st_ino : numéro d'inode du fichier

✓ ushort st_mode : mode du fichier. Ce champ regroupe sur 4 bits le type du fichier, et sur 12 bits les droits du fichier. Ceux-ci sont définis dans <***sys/mode.h***>. La vérification de ces bits se fait grâce à l'opérateur binaire ***&*** ("ET" bit à bit) avec les constantes suivantes :

- S_IFREG : fichier "régulier" (classique)
- S_IFDIR : fichier "répertoire"
- S_IFCHR : fichier "spécial" mode "caractère"
- S_IFBLK : fichier "spécial" mode "bloc"
- S_IFIFO : fichier "pipe" (first-in, first-out)
- S_IFLNK : fichier "lien symbolique"
- S_IFSOCK : fichier "socket"
- S_IFMT : masque symbolique permettant d'isoler les 4 bits correspondants au type du fichier
- S_ISUID : droits contenant "setuid"
- S_ISGID : droits contenant "setgid"
- S_ISTICK : droits contenant "sticky bit"
- S_IRUSR : droit de lecture pour le propriétaire
- S_IWUSR : droit d'écriture pour le propriétaire
- S_IXUSR : droit d'exécution pour le propriétaire
- S_IRWXU : groupement de lecture, écriture, exécution pour le propriétaire
- S_IRGRP : droit de lecture pour le groupe
- S_IWGRP : droit d'écriture pour le groupe
- S_IXGRP : droit d'exécution pour le groupe
- S_IRWXG : groupement de lecture, écriture, exécution pour le groupe
- S_IROTH : droit de lecture pour les autres
- S_IWOTH : droit d'écriture pour les autres
- S_IXOTH : droit d'exécution pour les autres
- S_IRWXO : groupement de lecture, écriture, exécution pour les autres

✓ short st_nlink : nombre de liens (noms logiques reliés au fichier)

✓ short st_uid : numéro du propriétaire du fichier

✓ short st_gid : numéro du groupe auquel appartiens le fichier

✓ dev_t st_rdev : concaténation sur un entier (2 octets) des nœuds majeur et mineur du fichier (ce champ n'est significatif que pour un fichier spécial)

✓ off_t st_size : taille du fichier en octet

✓ time_t st_atime : date (en secondes depuis le 1/1/1970) de dernier accès (lecture-écriture) du fichier

✓ time_t st_mtime : date (en secondes depuis le 1/1/1970) de dernière modification (écriture) du fichier

✓ time_t st_ctime : date (en secondes depuis le 1/1/1970) de création ou de dernière modification de statut (droits, propriétaire, etc.) du fichier

1) Accès au contenu de l'inode

Les primitives ***stat()***, ***fstat()***, ***lstat()*** permettent d'accéder au contenu de l'inode d'un fichier.

✓ la primitive ***stat()*** accède à un fichier par son nom

✓ la primitive ***fstat()*** accède à un fichier par son descripteur (renvoyé par la primitive ***open()***)

✓ la primitive ***lstat()*** renvoie des informations sur le fichier cible d'un lien symbolique

```
#include <sys/stat.h>
int stat (char *nom, struct stat *buf);
int fstat (int file, struct stat *buf);
int lstat (char *nom, struct stat *buf);
```

Explication des paramètres :

✓ char *nom : nom du fichier

✓ int file : descripteur du fichier ouvert au préalable par la primitive ***open()***

✓ struct stat *buf : pointeur vers une variable de type ***struct stat*** qui sera remplie avec les éléments récupérés par la fonction

2) Modification des attributs du fichier

Les primitives ***chmod()*** et ***fchmod()*** permettent de modifier les droits d'un fichier

✓ la primitive ***chmod()*** modifie les droits d'un fichier par son nom

✓ la primitive ***fchmod*** modifie les droits d'un fichier par son descripteur (renvoyé par la primitive ***open()***)

```
#include <sys/mode.h>
int chmod (char *nom, int droits);
int fchmod (int file, int droits);
```

Explication des paramètres :

✓ char *nom : nom du fichier

✓ int file : descripteur du fichier ouvert au préalable par la primitive ***open()***

✓ int droits : droits attribués au fichier (utiliser au choix une valeur **octale** ou bien une association des bits vus dans la structure ***stat***).

Les primitives ***chown()*** et ***fchown()*** permettent de modifier les numéros de propriétaire et groupe d'un fichier (uid et gid)

✓ la primitive ***chown()*** modifie les uid/gid d'un fichier par son nom

✓ la primitive ***fchown()*** modifie les uid/gid d'un fichier par son descripteur (renvoyé par la primitive ***open()***)

```
#include <sys/mode.h>
int chown (char *nom, int uid, int gid);
int fchown (int file, int uid, int gid);
```

Explication des paramètres :

✓ char *nom : nom du fichier

✓ int file : descripteur du fichier ouvert au préalable par la primitive ***open()***

✓ int uid : numéro du futur propriétaire à qui appartiendra le fichier

✓ int gid : numéro du futur groupe auquel appartiendra le fichier

Remarque : Dans le cas où le programmeur ne désirerait modifier que le ***uid*** (ou ***gid***) du fichier, il suffit de mettre la valeur spéciale ***–1*** dans le paramètre qu'il ne veut pas changer.

3) Création/Effacement de fichier

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

mode_t umask(mode_t mask);
```

umask() fixe le masque de création de fichiers du processus appelant à la valeur *mask* & 0777 (c'est-à-dire, seuls les bits relatifs aux permissions du fichier sont utilisés) et renvoie la valeur précédente du masque.

Ce masque est utilisé par **open**, **mkdir** et d'autres appels système qui créent des fichiers pour positionner les permissions d'accès initiales sur les fichiers ou répertoires nouvellement créés. Les bits contenus dans le umask sont éliminés de l'argument *mode* des appels **open** et **mkdir**.

Les constantes qui peuvent être utilisées pour définir *mask* sont décrites dans **stat**.

La valeur typique par défaut pour le umask du processus est *S_IWGRP* | *S_IWOTH* (022 en octal). Dans le cas général où l'argument *mode* de **open** vaut :

S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH

(0666 en octal) lors de la création d'un nouveau fichier, les permissions sur le fichier créé seront :

S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH

(car 0666 & ~022 = 0644 ; c'est-à-dire, rw-r--r--).

Les primitives ***creat()***, ***mkdir()***, ***mkfifo()*** et ***mknod()*** permettent de créer respectivement un fichier "classique", un fichier "répertoire", un fichier "pipe" ou bien un fichier de n'importe quel type.

```
#include <sys/mode.h>
int creat (char *nom, int droits);
int mkdir (char *nom, int droits);
int mkfifo (char *nom, int droits);
int mknod (char *nom, int droits, int dev);
```

Explication des paramètres :

✓ char *nom : nom du fichier à créer. Tous les répertoires précédant le nom doivent exister et être accessibles (droit "x").

✓ int droits : droits attribués au fichier (utiliser les constantes vues dans la structure ***stat***). Pour la primitive ***mknod()***, il faut ajouter au droits le type du fichier voulu (pris parmi les constantes de la structure ***stat***).

✓ int dev : variable contenant dans son premier octet le nœud majeur et dans son second octet le nœud mineur du fichier "spécial" que l'on veut créer. Remplir cette variable nécessite de savoir manipuler les bits, ou les ***union*** ; mais cette variable n'a de signification que si la primitive est utilisée pour créer un fichier spécial.

Les primitives ***link()*** et ***symlink()*** permettent respectivement de créer un lien physique ou un lien symbolique vers un fichier.

```
#include <unistd.h>
int link (char *source, char *cible);
int symlink (char *source, char *cible);
```

Explication des paramètres :

- ✓ char *source : nom du fichier qui sera à l'origine du lien
- ✓ char *cible : nom du lien créé

La primitive ***rmdir()*** permet d'effacer un fichier de type "répertoire" à condition que celui-ci soit vide.

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

Explication des paramètres :

- ✓ char *nom : nom du fichier

La primitive ***unlink()*** permet d'effacer un lien physique sur un fichier. Si le compteur de liens présent dans l'inode tombe à zéro, le fichier est physiquement effacé.

```
#include <unistd.h>
int unlink (char *nom);
```

Explication des paramètres :

- ✓ char *nom : nom du fichier

4) Autres fonctions de manipulation de fichiers

La primitive ***access()*** permet de tester les droits d'accès à un fichier par rapport au propriétaire réel du processus utilisant cette primitive.

```
#include <unistd.h>
int acces (char *nom, int droit);
```

Explication des paramètres :

- ✓ char *nom : nom du fichier
- ✓ int droit : droit à tester. Celui-ci peut être combiné avec les valeurs suivantes :
  - R_OK : droit en lecture
  - W_OK : droit en écriture
  - X_OK : droit en exécution
  - F_OK :simple existence

Valeur renvoyée (int) : 0 si le test demandé réussi, (-1) sinon

III) ACCÈS AU CONTENU D'UN FICHIER

1) Ouverture/Fermeture d'un fichier

La primitive ***open()*** permet à un processus de réaliser une ouverture de fichier régulier (fichier normal uniquement), c'est à dire de demander l'allocation d'une nouvelle entrée dans la table des fichiers ouverts du système. Elle renvoie un descripteur de fichier, sous la forme d'un entier, pour le fichier dont le nom est passé en paramètre. Ce descripteur est garanti comme étant le plus petit descripteur disponible. La primitive permet l'ouverture d'un fichier existant ou la création d'un nouveau fichier au besoin (évite d'avoir à utiliser la primitive ***creat()***).

```
#include <fcntl.h>
int open (char *nom, int mode[, int droits]);
```

Explication des paramètres :

✓ char *nom : nom du fichier à ouvrir

✓ int mode : mode d'ouverture demandé. Celui-ci est composé impérativement d'une des trois constantes suivantes :

- O_RDONLY : ouverture en lecture seule
- O_WRONLY : ouverture en écriture seule
- O_RDWR : ouverture en lecture-écriture

et facultativement de une ou plusieurs des quatre constantes suivantes :

- O_CREAT : création du fichier s'il n'existe pas
- O_TRUNC : vidage au préalable du contenu du fichier si celui-ci n'est pas vide
- O_APPEND : écriture à la fin du fichier
- O_EXCL : ne pas ouvrir le fichier s'il existe déjà (protection)

L'association de ces constantes se fait par l'opérateur binaire de manipulation de bits"|".

✓ int droits (facultatif) : droits attribués au fichier (utiliser au choix une valeur **octale** ou bien une association des bits vus dans la structure ***stat***). Ce paramètre n'a de signification que si on demande une création avec O_CREAT. De plus, ce paramètre n'aura d'effet que si le fichier est effectivement créé. Enfin, si ce paramètre est pris en compte, il sera néanmoins filtré par le masque de création de fichiers (***umask***).

Valeur renvoyée (int) : descripteur du fichier ouvert. Ce descripteur servira de référence ultérieure chaque fois qu'il faudra accéder au fichier.

La primitive ***close()*** ferme un fichier ouvert préalablement par ***open()***.

```
#include <fcntl.h>
int close (int desc);
```

Explication des paramètres :

✓ int desc : descripteur du fichier renvoyé par la primitive ***open()***

2) Lecture/Ecriture dans un fichier

Les primitives ***read()*** et ***write()*** ont pour but d'aller respectivement lire et écrire des octets dans un fichier préalablement ouvert avec la primitive ***open()***.

```
#include <fcntl.h>
int read (int desc, char *buffer, int nb);
int write (int desc, char *buffer, int nb);
```

Explication des paramètres :

✓ int desc : descripteur du fichier renvoyé par la primitive ***open()***

✓ char *buffer : pointeur vers une zone mémoire dans laquelle seront rangés (ou pris) les caractères lus (ou écrits) du fichier

✓ int nb : nombre de caractères à lire (ou écrire) dans le fichier

Valeur renvoyée (int) :

✓ Le nombre d'octets réellement lus (ou écrits) dans le fichier s'il y en a. Ce nombre peut être inférieur à ***nb*** (par exemple si on demande de lire plus que ce qu'il n'y a), mais jamais supérieur.

✓ 0 s'il n'y a plus rien à lire

3) Positionnement dans un fichier

La primitives ***lseek()*** permet de déplacer le pointeur interne de lecture/écriture d'un fichier préalablement ouvert avec la primitive ***open()***.

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

Explication des paramètres :

✓ int desc : descripteur du fichier renvoyé par la primitive ***open()***

✓ off_t offset : position de placement dans le fichier

✓ int whence : directive de positionnement. Cette directive doit être prise parmi l'une des trois constantes suivantes :

- SEEK_SET : la position demandée sera prise en compte à partir du début du fichier
- SEEK_CUR : la position demandée sera prise en compte à partir de la position courante
- SEEK_END : la position demandée sera prise en compte à partir de la fin du fichier

Valeur renvoyée (int) :

✓ La nouvelle position mesurée en octets depuis le début du fichier.

4) Manipulation de descripteurs

Les primitives ***dup()*** et ***dup2()*** permettent de manipuler des descripteurs de fichiers ouverts. La primitive ***dup()*** fournit un nouveau descripteur sur un fichier déjà ouvert, pris parmi les premiers descripteurs non-utilisés du système. La primitive ***dup2()*** force un descripteur précis, même s'il est

utilisé, à pointer sur un fichier déjà ouvert. dans le cas d'un descripteur déjà utilisé, celui-ci sera d'abord libéré (primitive ***close()***) avant d'aller référencer le descripteur demandé.

```
#include <fcntl.h>
int dup (int desc);
int dup2 (int desc1, int desc2);
```

Explication des paramètres :

- ✓ int desc : descripteur du fichier à dupliquer, renvoyé par la primitive ***open()***
- ✓ int desc1 : descripteur du fichier à dupliquer, renvoyé par la primitive ***open()***
- ✓ int desc2 : descripteur qui sera le synonyme de ***desc1***

Valeur renvoyée (int) : Nouveau descripteur synonyme de ***desc*** (primitive ***dup()*** uniquement).

Ces primitives servent notamment à remplacer un des périphériques standards (clavier, écran, erreurs) par un fichier sur disque.

IV) LES RÉPERTOIRES

1) Lire le contenu d'un répertoire

GNU/Linux dispose de fonctions pour lire le contenu des répertoires. Bien qu'elles ne soient pas directement liées aux fonctions de bas niveau, elles sont souvent utiles.

Pour lire le contenu d'un répertoire, les étapes suivantes sont nécessaires :

1. Appelez opendir en lui passant le chemin du répertoire que vous souhaitez explorer. opendir

renvoie un descripteur DIR* , dont vous aurez besoin pour accéder au contenu du répertoire.

Si une erreur survient, l'appel renvoie NULL ;

2. Appelez readdir en lui passant le descripteur DIR* que vous a renvoyé opendir . À chaque appel, readdir renvoie un pointeur vers une instance de struct dirent correspondant à l'entrée suivante dans le répertoire. Lorsque vous atteignez la fin du contenu du répertoire, readdir renvoie NULL .

La struct dirent que vous optenez via readdir dispose d'un champ d_name qui contient le nom de l'entrée.

3. Appelez closedir en lui passant le descripteur DIR* à la fin du parcours.

Incluez <sys/types.h> et <dirent.h> si vous utilisez ces fonctions dans votre programme.

Au niveau applicatif, les fonctions opendir( ), readdir( ), closedir( ) nous permettent d'accéder au contenu d'un répertoire sous forme de structures dirent. Pour assurer la portabilité d'une application, nous nous limiterons à l'utilisation du seul champ qui soit défini par Posix, char d_name[ ], qui contient le nom du fichier ou du sous-répertoire.

Ces fonctions sont définies dans <dirent. h>:

```
DIR * opendir (const char * repertoire);

struct dirent * readdir (DIR * dir);

int closedir(DIR * dir);
```

Le type DIR, défini dans <sys/types.h>, est une structure opaque, comparable au flux FILE, mais on l'emploie sur des répertoires au lieu des fichiers. A la manière de fopen( ), la fonction opendir( ) renvoie un pointeur NULL en cas d'échec. La fonction readdir( ) renvoie l'entrée suivante ou NULL une fois arrivée à la fin du répertoire. Lorsqu'on a fini d'utiliser le répertoire. on le referme avec closedir( ):

exemple_opendir.c :

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


void
 affiche_contenu (const char * repertoire){
DIR * dir;
struct dirent * entree;
dir = opendir (repertoire);

if (dir == NULL)  return;

fprintf (stdout, "%s :\n", repertoire);

while ((entree = readdir (dir)) != NULL)
 fprintf (stdout, " %s\n", entree -> d_name);

fprintf (stdout, "\n");

closedir (dir);
}

Int main (int argc, char * argv []){
int i;
```

```
if (argc < 2) {affiche_contenu (".");}
else
{for (i = 1; i < argc; i++)
 affiche_contenu (argv [i]);}

return (0);
}
```

On observe que readdir( ) renvoi les entrées « . » et « .. » correspondant respectivement au répertoire courant et à son parent. Ce comportement n'est pas garanti par Posix. Par contre, ces deux entrées sont toujours valables pour opendir( ) ou pour des commandes de changement de répertoire de travail.

Comme avec les flux de fichiers, il est possible de se déplacer au sein des répertoires DIR en utilisant rewinddir( ) , qui revient au début du répertoire, telldir( ) , qui renvoie la position courante, ou seekdir( ) , qui permet de sauter à une position donnée, renvoyée précédemment par telldir( ). Les prototypes de ces fonctions sont :

```
void rewinddir (DIR * dir);

void seekdir (DIR * dir, off_t offset);

off_t telldir(DIR * dir);
```

La fonction rewinddir( ) est définie par Posix, mais les deux autres sont spécifiques à

BSD.

2) Changement de répertoire de travail

3)

Chaque processus dispose en permanence d'un répertoire de travail. Ce répertoire est hérité du processus père et peut être modifié avec l'appel-système chdir( ). Le changement n'est toutefois visible que dans le processus courant et ses futurs descendants, pas dans le processus père.

Lors de la connexion d'un utilisateur, login lit dans le fichier /etc/password le répertoire personnel de l'utilisateur et s'y place, avant d'invoquer le shell. Il configure également la variable d'environnement HOME, qui restera donc correctement renseignée, même si l'utilisateur se déplace dans l'arborescence du système de fichiers.

Il existe deux appels-système permettant de modifier le répertoire courant d'un processus chdir( ) , qui prend en argument le nom du répertoire destination, et fchdir( ) , qui utilise un descripteur sur le répertoire cible. Ces deux appels-système sont déclarés dans <unistd.h>. mais seul chdir( ) est défini par Posix.

```
int chdir (const char * nom);

int fchdir(int descripteur);
```

Ils renvoient tous deux 0 en cas de réussite, et -1 en cas d'erreur.

Un processus dispose toujours d'un répertoire de travail, mais aussi surprenant que cela puisse paraître. il n'existe pas d'appel-système permettant d'obtenir directement le nom de ce répertoire. II faut pour cela s'adresser à une fonction de bibliothèque comme getcwd( ), get_current_working_dir_name( ) ou getwd( ).

Seule la première de ces fonctions est définie par Posix. Leurs prototypes sont déclarés dans <unistd.h> :

```
char * getcwd (char * buffer, size_t taille);

char * get_current _working_dir_name(void);

char * getwd(char * buffer);
```

4) Création et suppression de répertoire

5)

Pour créer un nouveau répertoire, ou en supprimer un, il existe deux appels-système, mkdir( ) et rmdir( ), dont le fonctionnement est assez intuitif et rappelle les deux commandes /bin/mkdir et /bin/rmdir qui sont construites à partir de ces fonctions.

Leurs prototypes sont déclarés ainsi dans <unistd.h> :

```
int mkdir (const char * repertoire, mode_t mode);

int rmdir (const char * repertoire) ;
```

L'emploi du type mode_t pour le second argument de mkdir() nécessite l'inclusion supplémentaire de <fcntl.h> et de <sys/types.h>, comme avec open( ).
Ces deux appels-système renvoient 0 s'ils réussissent, et -1 en cas d'échec. En plus des erreurs liées aux autorisations d'accès ou aux irrégularités concernant le nom fourni, mkdir( ) peut échouer avec le code ENOSPC dans errno si le disque est saturé ou si le quota attribué à l'utilisateur est rempli, ou avec l'erreur EEXIST si le répertoire existe déjà.

De son côté, rmdir( ) peut renvoyer surtout les erreurs EACCES ou EPERM liées aux autorisations d'accès, ENOTEMPTY si le répertoire à supprimer n'est pas vide, ou EBUSY si on essaye de supprimer le répertoire de travail courant d'un autre processus. Cette dernière erreur n'est pas respectée sur tous les systèmes.

La profondeur des sous-répertoires dans une arborescence n'est pas limitée. Il est donc possible de créer des sous-répertoires imbriqués jusqu'à la saturation du disque. Il peut toute-fois y avoir des limitations liées au système de fichiers sous-jacent.
Le mode fourni en second argument de mkdir( ) sert à indiquer les autorisations d'accès du répertoire nouvellement créé. Comme pour open( ), on utilise les constantes S_Ixxx ou leurs valeurs octales.