Programmation C et Système
Fonctions d'accès au disque

Régis Barbanchon

L2 Informatique

Volume et device number

Les volumes d'un système de fichier sont identifiés par :

- un nom de **device** dans le répertoire `/dev`,
- un **numéro de device** de forme `major,minor` en hexa,
- un **point de montage**, le chemin pour accéder à son contenu.

Exemple : un système sur lequel on trouve les volumes...

volume	device	major,minor	mount point
partition de disque	`/dev/sda2`	`8,2`	`/`
partition de clé USB	`/dev/sdb1`	`8,11`	`/media/usb`

ainsi que deux fichiers dans ces volumes :

- un fichier `/home/regis/photo.jpg` sur le disque,
- un fichier `/media/usb/pics/logo.gif` sur la clé.

Status de fichier, i-node et i-number

Le contenu d'un fichier occupe une série de **blocs** sur le volume.
Le **status** du fichier est stocké hors de ces blocs dans une structure appelée **i-node**, qui contient les infos suivantes :

- type de fichier (régulier, répertoire, lien symbolique. . .),
- *user id* et *group id* du propriétaire,
- droits d'accès pour le propriétaire, son groupe, et les autres,
- taille en bytes et nombre de blocs (de 512 bytes) du contenu,
- adresses des blocs stockant le contenu,
- nombre de *hard links* vers ce fichier,
- dernières dates de lecture, d'écriture, de modif de status,
- (*mais pas le nom du fichier*).

Les i-nodes sont regroupés en une table créée à la génération du système de fichiers. L'index d'un fichier dans cette table est son **i-number** qui l'identifie de façon unique à l'intérieur du volume. Avec le **device-number** du volume, il l'identifie de façon unique à l'intérieur du système de fichier.

Status d'un fichier avec la commande `stat` (1/2)

La commande `ls -i FILE` affiche l'i-node d'un fichier.

```
$ ls -i /home/regis/photo.jpg
14178043 /home/regis/photo.jpg
```

```
$ ls -i /media/usb/logo.gif
4635 /media/usb/pics/logo.gif
```

La commande `stat` permet d'obtenir le status complet :

```
$ stat /home/regis/photo.jpg
  File: '/home/regis/photo.jpg'
  Size: 1775713         Blocks: 3472       IO Block: 4096   regular file
Device: 802h/2050d      Inode: 14178043    Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/   regis)   Gid: ( 1001/   prof)
Access: 2019-03-22 14:19:34.897225079 +0100
Modify: 2019-02-14 21:37:29.914343410 +0100
Change: 2019-02-14 21:37:29.914343410 +0100
```

- Sauf si on a créé des hard links avec la cmd `ln FILE LINK`, le nombre de liens pour un fichier régulier est toujours 1.
- Il peut y avoir plus de blocs de 512 que nécessaire (ici $+4$), ou moins si le fichier a des trous (suites de zéros non codées).
- Ici, le groupe et les autres peuvent lire `(r)` le fichier, et le proprio peut écrire `(w)`, s'ils ont le droit d'exécuter `(x)` tous les répertoires le long du chemin accédant au fichier.
- Effacer $\neq$ modifier. Pour effacer le fichier, il faut le droit d'écrire `(w)` sur le répertoire contenant le fichier.

Status d'un fichier avec la commande `stat` (2/2)

Sous Unix, un répertoire est une forme particulière de fichier, contenant une liste de couples `(i-number, filename)`.
La commande `ls -i -d DIR` affiche l'i-number d'un répertoire.

```
$ ls -i -d /home/regis
14155778 /home/regis
```

```
$ ls -i -d /media/usb/
1280 /media/usb
```

La commande `stat` permet d'obtenir le status complet :

```
$ stat /home/regis
  File: '/home/regis/'
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: 802h/2050d      Inode: 14155778    Links: 66
Access: (0755/drwxr-xr-x)  Uid: ( 1000/   regis)   Gid: ( 1001/   prof)
Access: 2019-03-22 11:35:49.886104434 +0100
Modify: 2019-03-22 11:30:48.615388035 +0100
Change: 2019-03-22 11:30:48.615388035 +0100
```

- Créer des hard links avec la cmd `ln DIR LINK` est interdit, et le nb de liens est 2+ le nb de sous-répertoires `DIR/SUBDIR`, car `DIR` a les liens `"DIR"`, `"DIR/."` et `"DIR/SUBDIR/.."`. On peut donc supposer qu'il y a $66 - 2 = 64$ sous-répertoires.
- Tout le monde a le droit `(r)` et peut faire `ls DIR`.
- Tout le monde a le droit `(x)` et peut faire `cd DIR`.
- Seul le proprio a le droit `(w)` et peut faire `rm DIR/FILE`.

Les fonctions `stat` et `lstat`

Les fonctions C suivantes sont les analogues de la cmd `stat` :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int  stat (char const path[], struct stat * info_ptr);
int lstat (char const path[], struct stat * info_ptr);
```

`lstat ()` évite de traverser `path` si c'est un lien symbolique.
Ces fonction retournent 0 en cas de succès et renseignent
la structure de type `struct stat` pointée par `info_ptr` :

```
struct stat {
   dev_t     st_dev;          // device number (encoding varies across systems)
   ino_t     st_ino;          // i-number of i-node
   mode_t    st_mode;         // file type and protections
   nlink_t   st_nlink;        // number of hard links
   uid_t     st_uid;          // user  ID of owner
   gid_t     st_gid;          // group ID of owner

   off_t     st_size;         // file size in byte for regular file
   blksize_t st_blksize;      // hinted blocksize for read()/write()
   blkcnt_t  st_blocks;       // number of 512B blocks allocated

   struct timespec st_atim;   // time of last access
   struct timespec st_mtim;   // time of last modification
   struct timespec st_ctim;   // time of last status change (chown, chmod...)
   ...
};
```

Le champ `st_dev` de la structure `struct stat`

Si `stat (path, & info)` réussit alors l'entier `info.st_dev` contient le major et le minor du device number. Historiquement :

- le minor se trouvait dans les 8 bits de poids faible, et
- le major se trouvait dans les 8 bits suivants,

de sorte que l'on pouvait extraire le minor et major avec :

```
unsigned devNumber= info.st_dev; // always use unsigned when shifting bits
unsigned devMinor = (devNumber >> 0) & 0xFF;
unsigned devMajor = (devNumber >> 8) & 0xFF;
```

Remarque : `0xFF` en hexa, ou `0377` en octal, ou `255` en décimal, représentent le même nombre qui a ses 8 bits de poids faible à 1. Les notations hexa et octale permettent de le voir facilement, car on peut grouper les bits par 4 et 3 pour obtenir les chiffres.

Sur 12 bits : $\underbrace{0000}_{0}\ \underbrace{1111}_{F}\ \underbrace{1111}_{F} = \underbrace{000}_{0}\ \underbrace{011}_{3}\ \underbrace{111}_{7}\ \underbrace{111}_{7}$

```
#define _BSD_SOURCE
#include <sys/types.h>
unsigned devMinor= minor (info.st_dev);
unsigned devMajor= major (info.st_dev);
```

Cela n'est plus portable.
On doit passer par ces macros.

Le champ st_mode de la structure struct stat (1/2)

Si `stat (path, & info)` réussit alors `m= info.st_mode` contient à la fois le type de `path` et ses permissions.
Les macros suivantes sur `m` permettent de tester le type de `path` :

- `S_ISREG(m)` : est un fichier régulier ?
- `S_ISDIR(m)` : est un répertoire ?
- `S_ISCHR(m)` : est un char-device (comme un terminal) ?
- `S_ISBLK(m)` : est un block-device (comme un disque) ?
- `S_ISFIFO(m)` : est un pipe nommé (fifo) ?
- `S_ISLNK(m)` : est un lien symbolique ? avec `lstat()`.

Chaque macro est un masquage de bits avec la constante `S_IFMT` combiné avec un test d'égalité sur les constantes `S_IFxxx` :

```
#define S_ISREG(m)  (((m) & S_IFMT) == S_IFREG )
#define S_ISDIR(m)  (((m) & S_IFMT) == S_IFDIR )
#define S_ISCHR(m)  (((m) & S_IFMT) == S_IFCHR )
#define S_ISBLK(m)  (((m) & S_IFMT) == S_IFBLK )
#define S_ISFIFO(m) (((m) & S_IFMT) == S_IFFIFO)
#define S_ISLNK(m)  (((m) & S_IFMT) == S_IFLNK )
```

Le champ st_mode de la structure struct stat (2/2)

Si stat (path, & info) réussit alors m= info.st_mode contient à la fois le type de path et ses permissions.
Les constantes octales suivantes permettent de masquer les bits de m pour en isoler le bit d'un droit spécifique sur path :

```
#define S_IRUSR 00400  // owner has read permission
#define S_IWUSR 00200  // owner has write permission
#define S_IXUSR 00100  // owner has execute permission

#define S_IRGRP 00040  // group has read permission
#define S_IWGRP 00020  // group has write permission
#define S_IXGRP 00010  // group has execute permission

#define S_IROTH 00004  // others have read permission
#define S_IWOTH 00002  // others have write permission
#define S_IXOTH 00001  // others have execute permission
```

Contrairement aux types, il n'y a pas de macros de test.
On doit écrire nos propres prédicats, par exemple :

```
bool Mode_ownerCanRead   (mode_t m) { return (m & S_IRUSR) == S_IRUSR; }
bool Mode_groupCanWrite  (mode_t m) { return (m & S_IWGRP) == S_IWGRP; }
bool Mode_otherCanExecute(mode_t m) { return (m & S_IXOTH) == S_IXOTH; }
```

Les champs `st_*id` de la structure `struct stat`

Si `stat (path, & info)` réussit alors :

- `info.st_uid` contient l'ID du propriétaire, et
- `info.st_gid` contient l'ID de son groupe.

Pour obtenir leurs noms, voir `getpwuid()` et `getgrgid()` :

```
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>
struct passwd * getpwuid (uid_t uid);
struct group  * getgrgid (gid_t gid);
```

```
struct passwd {
  char * pw_name;   // user name
  char * pw_passwd; // user password
  uid_t  pw_uid;    // user ID
  gid_t  pw_gid;    // group ID
  char * pw_gecos;  // user info
  char * pw_dir;    // home directory
  char * pw_shell;  // default shell
};
```

```
struct group {
  char  * gr_name;   // group name
  char  * gr_passwd; // group password
  gid_t   gr_gid;    // group ID

  // NULL-terminated array of pointers
  // to all names of group members
  char ** gr_mem;
};
```

Exemple :

```
 struct passwd * p= getpwuid (info.st_uid);
 struct group  * g= getgrgid (info.st_gid);
 char *  userName= (p != NULL) ? p->pw_name : "(unknown)";
 char * groupName= (g != NULL) ? g->gr_name : "(unknown)";
```

Les champs st_*tim, de la structure struct stat

Les 3 champs d'horloge sont de type struct timespec :

```
struct stat {
   struct timespec st_atim, st_mtim, st_ctim; // Access, Modif, Status Change
   ...
};
```

Cette structure encapsule un *Unix time stamp* mesurant le temps en secondes depuis l'Epoch Unix, plus un laps en nano-secondes :

```
struct timespec {
  time_t tv_sec; // Unix Time Stamp: seconds since the Epoch (1/1/1970 00:00:00)
  long  tv_nsec; // Extra nano-seconds after the Unix Time Stamp
};
```

Pour convertir un time_t en une date struct tm, et en chaîne. . .

```
struct tm {
  int tm_sec , tm_min, tm_hour; // Seconds (0-60), Minutes (0-59), Hours (0-23)
  int tm_mday, tm_mon, tm_year; // Month day (1-31), Month (0-11), Year - 1900
  int tm_wday, tm_yday; // Week day (0-6, Sun = 0), Year day (0-365, 1 Jan = 0)
  int tm_isdst;          // is Daylight Saving Time ?
};
```

. . . voir gmtime/localtime(), puis strftime() de <time.h> :

```
struct tm *    gmtime (const time_t * time_ptr); // Univeral Time Coordinates
struct tm * localtime (const time_t * time_ptr); // User Local Time
size_t strftime (char s[], size_t n, char const fmt[], struct const tm * date);
```

Fonctions C analogues aux commandes (voir man pages)

```
# Print Current Working Dir
$ pwd
/home/regis/public_html
# Change Current Working Dir
$ cd ../Documents
```

```
#include <unistd.h>
char * getcwd (char dirPath[], size_t size);
int chdir (char const dirPath[]);
```

```
# Make New Directory
$ mkdir ../Dir1
$ mkdir /home/regis/Dir2
```

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

```
# Remove Empty Directory
$ rmdir ../Dir
# Remove File
$ unlink ../photo1.jpg
$ rm /home/regis/photo2.jpg
```

```
#include <unistd.h>
int rmdir  (char const dirPath[]);
int unlink (char const path[]);
```

```
# Rename or Move File
$ mv noise.wav pouet.wav
$ mv pouet.wav ../Music
```

```
#include <stdio.h>
int rename (char const old[], char const new[]));
```

```
# Create Hard Link
$ ln    ../Music link
# Create Symbolic Link
$ ln -s ../Music symlink
```

```
#include <unistd.h>
int   link (char const old[], char const link[]);
int symlink(char const old[], char const link[]);
```

```
# Read Symbolic Link Content
$ readlink symlink
../Music
```

```
#include <unistd.h>
ssize_t readlink // no '\0' added in target
  (char const path[], char target[], size_t n);
```

```
# Change File Permissions
$ chmod 0750           ../Pics
$ chmod u=rwx,g=rx,o= ../Pics
```

```
#include <sys/stat.h>
int chmod(char const path[], mode_t mode);
```

Différence en hard link et symbolic link (1/2)

Sous Unix, un répertoire est une forme particulière de fichier, contenant une liste de couples `(i-number, filename)`.

- Créer dans un répertoire un hard link de nom `name` sur un fichier d'i-number `i` consiste à :
  - rajouter le couple `(i, name)` dans le fichier du répertoire,
  - incrémenter le compteur de hard links dans l'i-node.

```
$ touch toto
$ ln toto toto-link
$ stat --format "i-number=%i hard-links=%h name=%n" toto*
i-number=15864353 hard-links=2 name=toto
i-number=15864353 hard-links=2 name=toto-link
$ ls -l toto*
-rw-r--r-- 3 regis prof 0   mars 23 01:53 toto
-rw-r--r-- 3 regis prof 0   mars 23 01:53 toto-link
```

- Alors que créer un symbolic link `name` sur un chemin `path` revient à créer un fichier spécial contenant la chaîne `path`.

```
$ touch tata
$ ln -s tata tata-symlink
$ stat --format "i-number=%i hard-links=%h name=%n" tata*
i-number=15864354 hard-links=1 name=tata
i-number=15864355 hard-links=1 name=tata-symlink
$ ls -l tata*
-rw-r--r-- 1 regis regis 0   mars 23 01:54 tata
lrwxrwxrwx 1 regis regis 4   mars 23 01:54 tata-symlink -> tata
```

Différence en hard link et symbolic link (2/2)

La cible d'un **hard link** :

- doit être un fichier existant
- ne peut pas être un répertoire ou être dans un autre volume.

A contrario, la cible d'un **symbolic link** :

- peut être un chemin inexistant (lien mort),
- peut être un répertoire ou un chemin vers un autre volume.

Un hard link a le même i-number que sa cible, pas le symbolic link.

Supprimer un **hard link ou sa cible** revient à :

- décrémenter le compteur de hard link de leur i-node commun,
- effacer l'entrée dans le répertoire,
- effacer le contenu du fichier lorsque ce compteur tombe à 0 (mais après que tous ses descripteurs ouverts sont fermés).

A contrario, pour le **symbolic link** :

- supprimer la cible ne supprime pas le symbolic link,
- supprimer le symbolic link ne supprime pas la cible.

Parcours de répertoire avec <dirent.h>

On manipule un répertoire ouvert avec le type `DIR *`,
type pointeur sur une structure opaque analogue à `FILE *`.
On ouvre un répertoire avec `opendir()`, l'analogue de `fopen()`.
On le ferme avec `closedir()`, l'analogue de `fclose()` :

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

On lit la prochaine entrée avec la fonction `readdir()`
qui retourne un `struct dirent *` ou `NULL` en fin de répertoire :

```
struct dirent { char d_name[/* unspecified */]; };
struct dirent * readdir (DIR *dir);
```

Les entrées ne sont pas triées, et `"."` et `".."` sont dans la liste.
Enfin, les répertoires sont seekable avec les fonctions suivantes :

```
long telldir   (DIR * dir);
void seekdir   (DIR * dir, long pos); // where pos was returned by telldir()
void rewinddir (DIR * dir);
```

Attention : `pos` n'est pas un offset, mais un entier opaque
obligatoirement retourné par un appel antérieur à `telldir()`.