Le système de fichiers : utilisation

Louis Mandel
louis.mandel@lri.fr
Laboratoire de Recherche en Informatique
Université Paris-Sud 11

IFIPS
Cycle ingénieur de la filière étudiant
année 2008/2009

Le système de fichiers

Il permet de stocker les données de façon persistante.

Un programme ne voit pas directement les informations telles qu'elles sont stockées sur le disque, mais par l'intermédiaire du système de fichiers qui lui en donne une vue abstraite :

- les détails d'implantation sont cachés,
- la présentation à l'utilisateur est indépendante de leur représentation sur machine,
- le système peut préserver des invariants (car l'utilisateur n'a pas accès aux détails de la représentation).

Pluralité des systèmes de fichiers

- Différents types de systèmes de fichiers
  - exemple : fat (ms-dos), ufs (Unix), ext2 (Linux), NTFS (Windows NT), HFS (MacOs), etc.
- Découplage système d'exploitation / système de fichiers
  - Un système d'exploitation peut fonctionner avec des systèmes de fichiers différents.
- Système de fichiers = abstraction d'un disque
  - Plusieurs disques
  - Plusieurs systèmes de fichiers
  - Éventuellement différents
  - Vue unifiée des systèmes de fichiers présents (une seule hiérarchie)

Le système de fichiers

Point de vue externe :

- Le système de fichiers présente une hiérarchie
  - Les répertoires « contiennent » des fichiers et des répertoires.
- La notion de fichier ne se limite pas à la notion de fichier sur disque.
  - Les périphériques sont aussi vus comme des fichiers.
- À chaque fichier est associé l'ensemble de ses attributs
  - propriétaire
  - droits d'accès
  - etc.

Le système de fichiers

Point de vue interne :

- Le système de fichier n'est pas une hiérarchie
- À chaque fichier correspond un i-nœud (*i-node* ou *index node*).
- Un fichier est identifié par une paire :
  - identification de la table qui contient son i-nœud
  - index dans cette table de son i-nœud (`ls -i`)
- Un i-nœud contient :
  - les attributs du fichier
  - et le moyen d'accéder à son contenu.

Hiérarchie

- Hiérarchie = arbre

Hiérarchie

- Hiérarchie = arbre + les répertoires `.` et `..`

Hiérarchie

- Hiérarchie = arbre + les répertoires `.` et `..` + liens
- Hiérarchie = graphe orienté acyclique (DAG)

Hiérarchie

- Hiérarchie = arbre + les répertoires `.` et `..` + liens + liens symboliques
- Hiérarchie = graphe orienté

Système de fichiers : un graphe

- Le système de fichiers est un graphe :
  - Un seul point d'entrée appelé la racine, désigné par `/`
  - Les arcs sont étiquetés par des noms
  - Deux arcs issus d'un même nœud ont des étiquettes différentes.
- Chemins :
  - chaque nœud peut être désigné de façon univoque au moins par un chemin à partir de la racine.
  - Chemin absolu, exemples : `/bin/ls` et `/tmp/../usr/lib`
  - Chemin relatif, exemples : `bin/gcc` et `../tmp/foo`
    Pour interpréter un chemin relatif, il faut préciser un nœud de départ (déterminé par le contexte).

Position dans la hiérarchie

- Chaque processus Unix a une position dans la hiérarchie.
- Les chemins relatifs sont interprétés à partir de cette position.
- Connaître sa position :
  - En shell : `pwd`
  - En C :

```
#include <unistd.h>
char *getcwd(char *tampon, size_t taille);
```

- Changer sa position sa position :
  - En shell : `cd`
  - En C :

```
#include <unistd.h>
int chdir(const char *reference);
```

Position dans la hiérarchie : Exemple (mpwd)

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

#define BUF_SIZE 1024
char buf[BUF_SIZE];
int main () {
  if (NULL == getcwd(buf, BUF_SIZE)) {
    perror("getcwd");
    exit(EXIT_FAILURE);
  }
  printf("%s", buf);
  exit(EXIT_SUCCESS);
}
```

Monter une hiérarchie dans une autre

- Monter un système de fichiers : `mount`
- Afficher les systèmes de fichiers : `mount` et `df`

Création de fichiers

```
--> ls -ia
  53026 .  1123984 ..
--> touch a
--> touch b
--> ls -ia
  53026 .  1123984 ..    53027 a    53028 b
```

Copie et déplacement de fichiers

```
--> ls -ia
  53026 .  1123984 ..    53027 a    53028 b
--> cp b c
--> ls -ia
  53026 .  1123984 ..    53027 a    53028 b    53029 c
--> mv c d
--> ls -ia
  53026 .  1123984 ..    53027 a    53028 b    53029 d
```

Les primitives de base

Les primitives de base : open

- Ouverture d'un fichier :

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

  - `pathname` désigne le fichier à ouvrir
  - `flags` spécifie les opérations que l'on désire réaliser :
    - lecture seule (`O_RDONLY`), écriture seule (`O_WRONLY`), lecture et écriture (`O_RDWR`)
    - créer le fichier s'il n'existe pas (`O_CREAT`), écriture à la fin du fichier (`O_APPEND`), etc.
  - `mode` est utile que lors de la création de fichier
  - La fonction retourne un entier que nous appellerons descripteur.

Les primitives de base : `create`

- Création d'un fichier :

```
int creat(const char *pathname, mode_t mode);
```

- Cette fonction est équivalente à :

```
open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode);
```

Les primitives de base : `read`

- Lecture d'un fichier :

  ```
  ssize_t read(int fd, void *buf, size_t count);
  ```

  - `fd` : descripteur correspondant au fichier que l'on veut lire.
  - `buf` : adresse de la mémoire où les caractères lus sont écrits.
  - `count` : nombre maximum de caractères lus.
  - valeur de retour :
    - nombre de caractères effectivement lus.
    - `-1` en cas d'erreur. Dans ce cas, la variable `errno` contient le code d'erreur

Les primitives de base : `read`

- La fonction `read` recopie dans `buf` les n premiers caractères à partir de la position courante (`offset`) où

$$n = min(\texttt{count}, \texttt{longueur du ficher} - \texttt{offset})$$

- Elle incrémente l'`offset` de n.

Remarque : il n'y a pas de marqueur de fin de fichier.

Les primitives de base : read (lire-fichier.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#define TAILLE_TAMPON 256
char tampon[TAILLE_TAMPON];

int main(int argc, char *argv[]) {
  int desc, lus;
  if (1 == argc) exit(1);
    if (-1 == (desc = open(argv[1], O_RDWR))) {
    perror("open"); exit(2);
  }
  do {
    if (-1 == (lus = read(desc, tampon, TAILLE_TAMPON))) {
      perror("read"); exit(3);
    }
    printf("lecture de %d caracteres\n", lus);
  } while (lus != 0);
  exit(0);
}
```

Les primitives de base : `write`

- Écriture d'un fichier :

  ```
  ssize_t write(int fd, const void *buf, size_t count);
  ```

  - `fd` : descripteur correspondant au fichier où l'on veut écrire.
  - `buf` : adresse de la mémoire où les caractères à écrire sont lus.
  - `count` : nombre maximum de caractères à écrire.
  - valeur de retour :
    - nombre de caractères effectivement écrits.
    - `-1` en cas d'erreur. Dans ce cas, la variable `errno` contient le code d'erreur
  - l'écriture change l'`offset` et éventuellement la taille du fichier

Les primitives de base : `lseek`

- Déplacement de la position courante :

  ```
  off_t lseek(int fd, off_t delta, int whence);
  ```

  - `offset` : déplacement de `delta` octets (peut être négatif).
  - `whence` : position à partir de laquelle le déplacement est fait.
    - `SEEK_SET` : début du fichier
    - `SEEK_CUR` : position courante
    - `SEEK_END` : fin du fichier
  - retourne la nouvelle position absolue
- Positionnement possible au delà de la fin du fichier
  - ne change pas la taille
  - écriture à cette position change la taille
  - le « trou » est rempli de zéros

Les primitives de base : `close`

- Fermeture d'un fichier :

```
int close(int fd);
```

- Libération des ressources.
- Bonne pratique : ouvrir et fermer les fichiers dans la même fonction.

Attributs des fichiers

Les i-nœuds

- Un i-nœud contient les attributs suivants :
  - Le type du fichier et des droits d'accès des différents utilisateurs
  - L'identification du propriétaire du fichier
  - L'identification du groupe propriétaire du fichier
  - La taille du fichier (en nombre de caractères)
  - Le nombre de liens physiques sur le fichier
  - La date de dernière modification
  - La date de dernière consultation
  - Pour les fichiers sur disque : l'adresse des blocs utilisés sur le disque
  - Pour les fichiers spéciaux : l'identification de la ressource associée
  - etc.
- Un i-nœud contient les adresses des blocs de données du fichier.
- Le système de fichier conserve sur disque une table des i-nœuds.

Les i-nœuds : les attributs

```
struct stat {
  dev_t   st_dev;    // identificateur du disque logique du fichier
  ino_t   st_ino;    // numéro du fichier sur le disque
  mode_t  st_mode;   // type du fichier et droits des utilisateurs
  nlink_t st_nlink;  // nombre de liens physiques
  uid_t   st_uid;    // propriétaire du fichier
  gid_t   st_gid;    // groupe propriétaire du fichier
  off_t   st_size;   // taille du fichier (si cela a un sens)
  quad_t  st_blocks; // blocs alloues
  time_t  st_atime;  // date de dernier accès
  time_t  st_mtime;  // date de dernier modification du contenu
  time_t  st_ctime;  // date de dernier modification du noeud
  ...
};
```

Les i-nœuds (`stat.c`)

- Pour accéder aux informations sur les i-nœuds :
  - En shell : `ls -il` ou `stat`
  - En C :

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

Utilisation typique :

```
struct stat statut;
if (-1 == stat("/tmp", &statut)) {
  perror("stat()"); exit(EXIT_FAILURE);
} ...
```

Types de fichiers

- Les fichiers réguliers ou ordinaires
  - Suite de caractères caractérisée par sa longueur.
  - Ces fichiers peuvent être des programmes ou des données.
- Les répertoires
- Les fichiers spéciaux
  - Associé à une ressource physique ou logique (disques, imprimantes, terminaux physique ou virtuels, etc.).
  - Mode bloc (comme les disques) : les entrées/sorties transitent par des caches ou *buffer caches* (tampons de la zone de données du noyau).
  - Mode caractère (comme les terminaux) : les entrées/sorties sont réalisées caractère par caractère et ne passent pas par les caches.
- Les liens symboliques
- Les tubes nommés et les sockets

Types de fichiers : `ls`

- `ls -l` : le premier caractère indique le type de fichier

```
--> ls -ld /dev/ /dev/cdrom /dev/sda1 /dev/tty /etc/passwd
drwxr-xr-x 13 root root 14400 2008-02-04 17:43 /dev/
lrwxrwxrwx  1 root root     4 2008-02-04 16:42 /dev/cdrom -> scd0
brw-rw----  1 root disk  8, 1 2008-02-04 17:42 /dev/sda1
crw-rw-rw-  1 root root  5, 0 2008-02-04 16:50 /dev/tty
-rw-r--r--  1 root root  1554 2007-12-21 15:36 /etc/passwd
```

Types de fichiers : le champ `st_mode`

- Les types de fichier peuvent être testés par les fonctions suivantes :
  - `S_ISREG(mode)` : fichier régulier
  - `S_ISBLK(mode)` : fichier spécial bloc
  - `S_ISCHR(mode)` : fichier spécial caractère
  - `S_ISDIR(mode)` : répertoire
  - `S_ISLNK(mode)` : lien symbolique
  - `S_ISFIFO(mode)` : tube nommé
  - `S_ISSOCK(mode)` : socket

- Le champ `st_mode` contient aussi les droits d'accès.

Les droits d'accès aux fichiers en Unix

- Trois types d'utilisateurs :
  - Propriétaire (`u`)
  - Membres du groupe propriétaire sauf le propriétaire (`g`)
  - Les autres (`o`)
- Un utilisateur privilégié : numéro de compte `0` (en général `root`)
- Trois types d'accès à un fichier :
  - lecture (`r`)
  - écriture (`w`)
  - exécution (`x`)
- Exemple :

```
--> ls -l aucun-droit tous-les-droits
---------- 1 mandel demons 0 2008-02-05 19:12 aucun-droit
-rwxrwxrwx 1 mandel demons 0 2008-02-05 19:12 tous-les-droits
```

Les droits d'accès : codage

- Les droits pour chaque type d'utilisateurs sont codés sur trois bits.
  - Exemple : `r-x` = 101, `rw-` = 110
- Les droits pour un fichier sont codés sur neuf bits.
  - Exemple : `rw-r-xr--` = 110101100
- Chaque droit est défini dans dans une constante.
  - droits du propriétaire (position `S_IRWXU`) :
    - `S_IRUSR` : lecture
    - `S_IWUSR` : écriture
    - `S_IXUSR` : exécution

Les droits d'accès : exemple (est-executable.c)

```
int main (int argc, char **argv) {
  struct stat statut;

  if (1 == argc) exit(1);
  if (-1 == stat(argv[1], &statut)) {
    perror("stat()"); exit(EXIT_FAILURE);
  }
  if (S_ISREG(statut.st_mode)) {
    printf("Fichier ordinaire");
    if (statut.st_mode & S_IXUSR)
      printf(", executable par son proprietaire");
  }
  putchar('\n');
  exit(EXIT_SUCCESS);
}
```

Les droits d'accès : trois bits supplémentaires

- Le set-uid bit (`s`) :
  - Modifie le comportement des fichiers binaires exécutables.
  - À l'exécution, le processus a les droits du propriétaire du fichier (qui est le propriétaire effectif du processus) et non de l'utilisateur qui le lance (qui en est le propriétaire réel).
  - Exemple :

```
--> ls -l /etc/passwd /usr/bin/passwd
-rw-r--r-- 1 root root  1554 2007-12-21 15:36 /etc/passwd
-rwsr-xr-x 1 root root 29104 2007-05-18 11:59 /usr/bin/passwd
```

- Le set-gid bit (`s`) : idem pour le groupe

Les droits d'accès : trois bits supplémentaires

- Le sticky bit (`t`) :
  - Modifie le comportement des répertoires
  - Interdit la suppression du répertoire et des fichiers contenu dans le répertoire à tout utilisateur autre que le propriétaire (même s'il a les droits en écriture)
  - Exemple :

```
--> ls -ld /tmp
drwxrwxrwt 16 root root 36864 2008-02-05 18:34 /tmp
```

Remarque : il existe d'autres modèles de gestion des droits.

Les droits d'accès : `chmod`

- Changer les droits
  - En shell : `chmod a+rwxt dir`
  - En C :

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

- Tester les droits d'accès
  - En shell : `ls -l`
  - En C :

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

    - `mode` est défini par combinaison de `R_OK`, `W_OK`, `X_OK` et `F_OK`.

Le propriétaire : le champ `uid_t`

- Informations sur le propriétaire

```
struct passwd *getpwuid(uid_t uid);


struct passwd {
  char *pw_name   // Nom de login
  uid_t pw_uid    // Numero d'utilisateur
  gid_t pw_gid    // Numero de groupe
  char *pw_dir    // Repertoire initial
  char *pw_shell  // Shell a utiliser
}
```

Le propriétaire : les champs `uid_t` et `gid_t`

- Changement de propriétaire (utilisateur et groupe)
  - En shell : `chown`
  - En C :

```
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Changement d'autres attributs d'un i-nœud

- Changement des dates de dernière modification
  - En C :

```
int utime(const char *filename, const struct utimbuf *buf);
```

- changement de la longueur :
  - En C :

```
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

Les différents types de fichiers

Les répertoires

- Le contenu d'un répertoire est un catalogue de paires :
  - Numéro d'i-nœud
  - Nom de fichier

- Les droits d'accès pour un répertoire correspondent à :
  - lecture : lister le contenu d'un répertoire
  - écriture : creation et suppression de fichiers dans le repertoire
  - exécution : se positionner dans le répertoire ou faire figurer ce répertoire dans une référence

Les répertoires

- Création d'un répertoire
  - En shell : `mkdir a a/b`
  - En C :

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

- Destruction d'un répertoire
  - En shell : `rmdir a/b a`
  - En C :

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

  - Peut seulement supprimer un répertoire vide.

Les répertoires

- Ouverture :

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

- Fermeture :

```
int closedir(DIR *dir);
```

- Lecture :

```
struct dirent {
   ino_t d_ino;        /* numero de l'inode */
   char  d_name[];     /* nom du fichier */
}
struct dirent *readdir(DIR *dir);
```

  - ▷ Lit une entrée du répertoire

- Rembobinage :

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

Les répertoires : Exemple (`mls.c`)

```
int main(int argc, char **argv) {
  int ind;
  struct stat s;
  DIR *dir;
  char ref[1024];
  struct dirent *entree;
  for (ind = 1; ind < argc; ind++) {
    if ( -1 == stat(argv[ind], &s) ) {
      fprintf(stderr, "%s fichier inconnu\n", argv[ind]); continue;
    }
    if ( !S_ISDIR(s.st_mode) ) { printf("%s\n", argv[ind]); continue; }
    printf("%s : repertoire\n", argv[ind]);
    if ( NULL == (dir = opendir(argv[ind])) ) {
      fprintf(stderr, "%s repertoire : ouverture impossible", argv[ind]); continue;
    }
    while ( NULL != (entree = readdir(dir)) ) {
      sprintf(ref, "%s/%s", argv[ind], entree->d_name);
      if ( -1 == stat(ref, &s) ) { perror("stat"); continue; }
      if ( S_ISDIR(s.st_mode) ) { printf("r : %s\n", entree->d_name); }
      else { printf("    %s\n", entree->d_name); }
    }
  }
  exit(EXIT_SUCCESS);
}
```

Les liens physiques

- Les liens physiques permettent de donner plusieurs nom à un même fichier
- Création :
  - En shell : `ln cible nom_du_lien`
  - En C :

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
```

- Renommage :
  - En shell : `mv old new`
  - En C :

```
int rename(const char *from, const char *to);
```

Les liens physiques

- Exemple :

```
--> ls -il
total 0
52927 -rw-r--r-- 1 mandel demons 0 2008-02-06 13:53 a
--> ln a b
--> ls -il
total 0
52927 -rw-r--r-- 2 mandel demons 0 2008-02-06 13:53 a
52927 -rw-r--r-- 2 mandel demons 0 2008-02-06 13:53 b
```

  - a et b ont le même i-nœud (première colonne)
  - Le nombre de liens a augmenté (troisième colonne)
- Un répertoire a toujours au moins deux liens vers son i-nœud.

Les liens physiques : Restrictions

- Il est impossible de créer des liens sur des répertoires
  - ▷ cela permettrait de transformer « l'arborescence » des fichiers en un graphe avec cycles
- Il est impossible de créer des liens entre des fichiers résidant sur des systèmes de fichiers différents.

Les liens symboliques

- Les liens symboliques n'ont pas le contraintes précédentes.
  - Création de cycles.
  - Liens vers fichiers ou répertoires (même inexistants).
- Contenu d'un chemin = lien
  - relatif
  - absolu
- Interprétation du nom
  - le lien symbolique lui-même
  - le fichier qu'il désigne
  - dépend du contexte d'utilisation

    `--> rm symlink`

    `--> cat symlink`

Les liens symboliques

- Créer un lien symbolique
  - En shell : `ln -s cible lien`
  - En C : `int symlink(const char *cible, const char *lien);`
- Lecture
  - En shell : `readlink`
  - En C :

    ```
    int readlink(const char *path, char *buf, int bufsiz);
    ```
  - Utilisation typique :

    ```
    char buf[PATH_MAX+1];
    ssizet_t len;
    if (-1 != (len = readlink(path, buf, PATH_MAX))) {
      buf[len] = '\0';
    } else { perror("error reading link"); }
    ```
- Consultation des attributs (de la cible) :

  ```
  int lstat(const char *path, struct stat *pstat);
  ```

Les liens symboliques : exemple

```
--> ls -il
total 4
53031 -rw-r--r-- 1 mandel demons 5 Feb  7 02:01 a
--> cat a
aaaa
--> ln -s a b
--> ls -il
total 4
53031 -rw-r--r-- 1 mandel demons 5 Feb  7 02:01 a
53032 lrwxrwxrwx 1 mandel demons 1 Feb  7 02:01 b -> a
--> cat b
aaaa
```

Les liens symboliques

- Lien symbolique invalide

```
--> ls -il
total 4
53031 -rw-r--r-- 1 mandel demons 5 Feb  7 02:01 a
53032 lrwxrwxrwx 1 mandel demons 1 Feb  7 02:01 b -> a
--> rm a
--> cat b
cat: b: No such file or directory
```

- Boucle

```
--> ln -s . loop
--> cd loop/loop/loop
```

Parcours d'une hiérarchie

- Traitement récursif
  - commandes `find`, `du`, etc.
- Traitement selon le type de fichier
  - Fichier ordinaire $\Rightarrow$ traitement standard
  - Répertoire $\Rightarrow$ itérer sur toutes les entrées sauf `.` et `..`
  - Lien symbolique : plusieurs choix sont possibles
    - mémoriser les nœuds visités
    - limiter le nombre de liens symboliques traversés

Suppression

- Suppression de fichier = suppression du lien
  - En shell : `rm`
  - En C :

```
int unlink(const char *path);
```

```
--> ls -il
--> rm a
--> ls -il
--> rm b
--> ls -il
```

Suppression

- Un fichier peut être supprimé physiquement lorsqu'il n'est plus pointé par aucun lien ou descripteur.
- Système de fichier = « garbage collector » par comptage de références.
- Garantie d'absence de cycles

Suppression : Conditions de suppressions

- `rm` (sans option) ne peut pas supprimer de répertoires.
- Il faut (et il suffit de) avoir les droits en écriture sur un répertoire pour supprimer un fichier.
  - Rôle du sticky-bit.