Programmation système : répertoires

Juliusz Chroboczek

25 Novembre 2009

1 Manipulation de répertoires

Un *répertoire* (*directory* en Anglais)[1] est un fichier dont le contenu est une suite d'*entrées de répertoire*, chaque entrée étant composée d'un nom et d'un numéro d'i-nœud. Les répertoires sont identifiés par une valeur spécifique du champ *mode* de leur i-nœud, et le système les traite spécialement : il ne permet pas à l'utilisateur de les modifier directement, et les consulte lors de la résolution de noms.

1.1 Résolution de noms

La *résolution de noms* est le processus qui permet de convertir un nom de fichier en un i-nœud. La résolution de noms est effectuée lors de tout appel système qui prend un nom de fichier en paramètre, par exemple `open` ou `stat`.

Chaque système de fichiers a un i-nœud distingué, appelé son *i-nœud racine*, qui contient un répertoire, appelé le *répertoire racine*. La résolution d'un nom de fichier absolu (un nom qui commence par un *slash* « `/` ») commence au répertoire racine. Le système recherche le premier composant du nom de fichier dans le répertoire racine. S'il est trouvé, et s'il pointe sur un répertoire, le deuxième composant est recherché dans celui-ci. La résolution se poursuit ainsi jusqu'à épuisement du nom de fichier.

La résolution d'un nom *relatif* (un nom qui ne commence pas par un *slash*) se fait de façon analogue, mais en commençant à un i-nœud qui dépend du processus — le *répertoire courant* du processus (voir cours suivant).

1.2 Lecture des répertoires

Il est parfois nécessaire de lire explicitement le contenu d'un nom de répertoire. Par exemple, le programme `ls` lit le contenu d'un répertoire et affiche les noms de fichier qu'il y trouve.

1 Le terme *folder* (dossier) a, à ma connaissance, été introduit par Mac OS (classique). Microsoft l'a adopté ce terme avec la sortie de Windows 95. Il n'est normalement pas utilisé sous Unix.

1.2.1 Les répertoires en 7e édition

Sous Unix 7e édition, les noms de fichier étaient limités à 14 caractères. Un répertoire était simplement une suite de structures de type `dirent` :

```
struct dirent {
    unsigned short d_ino;
    char d_name[14];
};
```

Un programme qui désirait lire le contenu d'un répertoire procédait de la façon habituelle : après un appel à `open`, il lisait simplement le contenu du répertoire à l'aide de `read` ; chaque suite de 16 octets lus était interprétée comme une structure de type `dirent`.

1.2.2 Interface de haut niveau

Sous 4.2BSD, la structure des répertoires est devenue plus compliquée : un répertoire pouvait contenir des noms de fichiers de longueur arbitraire, et il n'était plus possible de simplement interpréter un répertoire comme une suite d'enregistrements de 16 octets chacun. Afin d'éviter au programmeur d'analyser lui-même une structure de données complexe, une interface de haut niveau, analogue à `stdio`, a été définie.

L'incarnation moderne de cette interface s'appelle `dirent`. Elle définit deux structures de données : `DIR` (analogue à `FILE`) est une structure de données opaque représentant un répertoire ouvert. La structure `dirent` représente une entrée de répertoire ; elle contient au moins les champs suivants :

```
struct dirent {
    ino_t d_ino;
    char d_name[];
};
```

Comme dans la version traditionnelle (7e édition), le champ `d_ino` contient un numéro d'i-nœud, et le champ `d_name` le nom d'un fichier. À la différence de celle-ci, cependant, la longueur du champ `d_name` n'est pas spécifiée — il est terminé par un `\0`, comme une chaîne normale en C.

Les répertoires sont manipulés à l'aide de trois fonctions :

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
int closedir(DIR *dir);
```

La fonction `opendir` ouvre un répertoire et retourne un pointeur sur une structure de type `DIR` représentant le répertoire ouvert.

La fonction `readdir` retourne un pointeur sur l'entrée de répertoire suivante. Celui-ci peut pointer sur les tampons internes au `DIR` manipulé, ou même sur un tampon statique ; son contenu cesse donc d'être valable dès le prochain appel à `readdir`, et il faut en faire une copie s'il doit être utilisé par la suite.

Enfin, la fonction `closedir` ferme le répertoire et détruit la structure `DIR`.

2 Liens symboliques

Les liens (dits « durs »), tels qu'on les as vus au cours précédent, sont un concept propre, élégant et sans ambiguïté. Cependant, ils ont certaines limitations qui les rendent parfois peu pratiques :

– il est impossible de faire un lien entre deux systèmes de fichiers distincts ;
– il est impossible (en pratique) de faire un lien sur un répertoire.

Les *liens symboliques* ont été développés[2] pour éviter ces limitations. Il s'agit d'un concept peu élégant[3] et ayant une sémantique souvent douteuse, mais fort utile en pratique.

À la différence d'un lien (dur), un lien symbolique ne réfère pas à un i-nœud, mais à un nom. Lorsque le système rencontre un lien symbolique lors de la résolution d'un nom de fichier, il remplace le nom du lien par son contenu, et recommence la résolution. Par exemple, si `/usr/local` est un lien symbolique contenant `/mnt/disk2/local`, le nom de fichier

```
/usr/local/src/hello.c
```

est remplacé par

```
/mnt/disk2/local/src/hello.c
```

De même, si `/usr/share` est un lien symbolique contenant `../lib`, le nom de fichier

```
/usr/share/doc
```

est remplacé par

```
/usr/share/../lib/doc
```

Que se passe-t-il lorsqu'un lien symbolique contenant un nom relatif est déplacé ?

Création d'un lien symbolique

Un lien symbolique est créé par l'appel système `symlink` :

```
int symlink(const char *contents, const char *name);
```

Un appel à `symlink` construit un lien symbolique nommé `name` dont le contenu est la chaîne contenue dans `contents`.

Utilisation des liens symboliques

Dans la plupart des cas, l'utilisation des liens symboliques se fait de façon transparente par les appels système qui effectuent une résolution de noms. Par exemple, un appel à `open` sur le nom d'un lien symbolique se fera automatiquement sur la cible du lien.

Certains appels système ne traversent pas les liens symboliques. C'est le cas notamment de `unlink`, qui détruit le lien, et pas sa cible. L'appel système `stat` existe en deux variantes : `stat`, qui traverse les liens, et `lstat`, qui ne le fait pas.

On peut lire le contenu d'un lien symbolique à l'aide de l'appel système `readlink` :

```
ssize_t readlink(const char *path, char *buf, size_t bufsiz);
```

2 Et introduits dans 4.2BSD.
3 Moins horrible, toutefois, que les *raccourcis* utilisés sur certains systèmes.