Systèmes d'exploitation

TP 7 - Interface du système de fichiers

2023-06-08

Ce TP est librement adapté du chapitre 39 d'Operating Systems: Three Easy Pieces.

Dans ce TP, nous allons étudier l'interface du système de fichiers Unix.

1 Créer des fichiers

Nous commencerons par l'opération la plus élémentaire : la création d'un fichier. Elle peut être réalisée avec l'appel système `open`. En appelant `open()` et en lui passant l'option `O_CREATE`, un programme peut créer un nouveau fichier.

Créez maintenant un fichier appelé `foo` dans le répertoire de travail actuel :

```c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

int main() {
    int fd = open("foo", O_CREAT|O_WRONLY,S_IRUSR|S_IWUSR);
    printf("file descriptor: %d", fd);
    return EXIT_SUCCESS
}
```

La fonction `open()` prend un certain nombre d'arguments différents. Dans cet exemple, le second argument crée le fichier (`O_CREAT`) s'il n'existe pas et s'assure que le fichier ne peut être qu'écrit (`O_WRONLY`). Le troisième argument spécifie les permissions, dans ce cas en rendant le fichier lisible et inscriptible par le propriétaire.

Effacer le contenu du fichier à l'ouverture

Quelle option faut-il ajouter à `O_CREAT` et `OWRONLY` pour faire en sorte que si le fichier existe déjà, celui-ci soit tronqué à une taille de zéro octet, supprimant ainsi tout contenu existant ? Référez-vous aux *man pages*.

Un aspect important de `open()` est ce qu'il renvoie : un **descripteur de fichier (*file descriptor*)**. Un descripteur de fichier est juste un entier, propre au processus, qui est utilisé dans les systèmes Unix pour manipuler les fichiers. Ainsi, une fois qu'un fichier est ouvert, vous utilisez le descripteur de fichier pour lire ou écrire le fichier, en supposant que vous avez la permission de le faire. De cette manière, un descripteur de fichier est une pointeur opaque vers un objet de type fichier qui vous donne la possibilité d'effectuer certaines opérations en utilisant des fonctions telles que `read()` et `write()`.

Gestion des descripteurs de fichiers par le noyau

Les descripteurs de fichiers sont gérés par le système d'exploitation par processus. Cela signifie qu'une sorte de structure simple (par exemple, un tableau) est conservée dans la structure du processus sur les systèmes UNIX. Voici un extrait du noyau xv6 [CK+08] :

```
struct proc {
  ...
  struct file *ofile[NOFILE]; // Open files
  ...
};
```

Un simple tableau (avec un maximum de fichiers ouverts `NOFILE`) permet de savoir quels fichiers sont ouverts par un processus. Chaque entrée du tableau est en fait un pointeur vers une structure `file`, qui sera utilisée pour conserver les informations relatives au fichier lu ou écrit.

2 Lire et écrire des fichiers

Une fois que nous avons des fichiers, nous voulons bien sûr les lire ou les écrire. Commençons par créer un fichier `foo` avec du contenu puis lisons-le.

```
$ echo hello > foo
$ cat foo
```

Dans cet extrait de code, nous redirigeons la sortie du programme `echo` vers le fichier `foo`, qui contiendra alors le mot "hello". Nous utilisons ensuite `cat` pour afficher le contenu du fichier. Mais comment le programme `cat` accède-t-il au fichier `foo` ?

Pour le savoir, nous utiliserons un outil incroyablement utile pour identifier les appels système effectués par un programme. Sous Linux, cet outil s'appelle `strace` ; d'autres systèmes disposent d'outils similaires (voir `dtruss` sur Mac, ou `truss` sur certaines anciennes variantes d'Unix). `strace` permet de tracer chaque appel système effectué par un programme pendant qu'il s'exécute, et d'afficher la trace à l'écran.

Utilisons `strace` pour comprendre comment fonctionne `cat`:

```
$ strace cat foo
```

Reconnaissez-vous certains appels systèmes réalisés par `cat` ?

Essayons maintenant d'identifier l'appel système responsable de l'ouverture du fichier `foo`.

```
strace cat foo 2>&1 | grep foo
```

L'outil `grep` nous permet de chercher un motif dans un texte. Il est nécessaire de rediriger l'erreur standard de `strace` vers la sortie standard (`2>&1`) pour pouvoir ensuite la traiter avec `grep`. Vous devriez obtenir un résultat semblable à la ligne suivante.

```
execve("/usr/bin/cat", ["cat", "foo"], 0x7ffe75af5f38 /* 54 vars */) = 0
openat(AT_FDCWD, "foo", O_RDONLY)       = 3
```

C'est l'appel système `openat()` (assez similaire à `open()`) qui est ici utilisé par `cat` pour ouvrir le fichier foo.

Pourquoi l'appel à `openat()` renvoie-t-il `3`, et non `0` ou `1` comme on pourrait s'y attendre ? Il s'avère que chaque processus en cours d'exécution a déjà trois fichiers ouverts, l'**entrée standard** (*stdin*) (que le processus peut lire pour recevoir des données), la **sortie standard** (*stdout*) (sur laquelle le processus peut écrire pour afficher des informations à l'écran) et l'**erreur standard** (*stderr*) (sur laquelle le processus peut écrire des messages d'erreur). Ces éléments sont représentés par les descripteurs de fichiers `0`, `1` et `2`, respectivement. Ainsi, lorsque le programme ouvre un autre fichier pour la première fois (comme le fait `cat` ci-dessus), il s'agira presque certainement du descripteur de fichier `3`.

Filtrez maintenant les résultats de `strace` pour identifier les lignes dans lequel des références à `hello` sont faites.

Une fois l'ouverture du fichier réussie, `cat` utilise l'appel système `read()` pour lire à certains octets d'un fichier. Vous devriez aussi voir un autre résultat intéressant du `strace` : un seul appel à l'appel système `write()`, vers le descripteur de fichier `1`. Comme nous l'avons mentionné plus haut, ce descripteur est connu sous le nom de sortie standard et est donc utilisé pour écrire le mot "hello" à l'écran, comme le programme `cat` est censé le faire.

L'écriture d'un fichier s'effectue de la même manière. Tout d'abord, un fichier est ouvert pour l'écriture, puis l'appel système `write()` est appelé, peut-être à plusieurs reprises pour les gros fichiers, et enfin `close()` sert à fermer le descripteur de fichier pour qu'il ne référence plus le fichier.

Écriture dans un fichier

Écrivez un programme qui reproduit le fonctionnement de `cat foo`. Ouvrez d'abord le fichier `foo` à l'aide d'`open()`, lisez ensuite 6 octets du fichier à l'aide de `read()` et écrivez ensuite le contenu lu dans la sortie standard avec `write()`. Finalement, fermez le fichier avec `close()`.

3 Lire et écrire des fichiers de façon non-séquentielle

Jusqu'à présent, nous avons vu comment lire et écrire des fichiers, mais tous les accès ont été séquentiels, c'est-à-dire que nous avons soit lu un fichier du début à la fin, soit écrit un fichier du début à la fin.

Par exemple, si vous créez un index sur un document texte et que vous l'utilisez pour rechercher un mot spécifique, il se peut que vous lisiez à partir d'un certain nombre d'emplacements aléatoires dans le document. Pour ce faire, nous utiliserons l'appel système `lseek()`. Voici le prototype de la fonction :

```
off_t lseek(int fildes, off_t offset, int whence);
```

Le premier argument est un descripteur de fichier. Le deuxième argument est `offset`, qui positionne l'*offset* du fichier à un emplacement particulier dans le fichier. Le troisième argument, appelé `whence` pour des raisons historiques, détermine exactement comment la recherche est effectuée.

Extrait de la page du manuel :

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
    location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
    the file plus offset bytes.
```

Pour chaque fichier ouvert par un processus, le système d'exploitation conserve un offset "actuel", qui détermine l'endroit où la lecture ou l'écriture suivante commencera dans le fichier. Ainsi, **un fichier ouvert dans un processus possède un offset actuel**, qui est mis à jour de deux manières. La première est que lorsqu'une lecture ou une écriture de N octets a lieu, N est ajouté à l'offset actuel ; ainsi, chaque lecture ou écriture met implicitement à jour l'offset. La seconde est explicitement avec `lseek`, qui modifie l'offset comme spécifié ci-dessus.

Attention : Appeler `lseek()` ne conduit pas à un seek sur le disque. Les deux concepts n'ont en commun que leur nom.

lseek

Écrivez un programme qui :

- ouvre un fichier `monfichier.txt` en mode lecture-écriture;
- écrit dans le fichier le texte suivant : `Hello, World`;
- utilise `lseek` pour placer l'offset du fichier au niveau du commencement du mot `World`;
- lit le fichier à partir de l'offset et affiche sur l'écran le mot `World` lu depuis le fichier.

Structure `file`

L'offset est conservé dans la structure `file` dans le noyau, que nous avons vu plus tôt. Voici une definition xv6 (simplifiée) de la structure :

```
struct file {
    int ref;
    char readable;
    char writable;
    struct inode *ip;
    uint off;
};
```

4 Renommer des fichiers

Une fois que nous avons un fichier, il est parfois utile de pouvoir lui donner un nom différent. Utilisons la commande `mv` pour renommer le fichier `foo`.

```
mv foo bar
```

En utilisant strace, nous pouvons voir que `mv` utilise l'appel système `rename()`.

Une garantie intéressante fournie par l'appel `rename()` est qu'il est (généralement) implémenté comme un appel atomique en ce qui concerne les pannes du système. Si le système tombe en panne pendant le renommage, le fichier sera nommé soit avec l'ancien nom, soit avec le nouveau nom, et aucun état intermédiaire étrange ne peut survenir. Ainsi, `rename()` est essentiel pour supporter certains types d'applications qui requièrent une mise à jour atomique de l'état du fichier.

Création atomique d'un fichier

Écrivez un programme qui crée un fichier et y écrit le texte suivant :

```
Un bon programmeur est quelqu'un qui regarde toujours
des deux côtés avant de traverser une rue à sens unique.
```

Le nom du fichier est passé en argument au programme. Ce programme présente une petite subtilité : **en cas de panne, le fichier est soit créé complétement avec le texte intégral, soit pas du tout**.

Indice : utilisez un fichier temporaire.

5 Supprimer des fichiers

Comment supprimer des fichiers ? Vous pensez probablement le savoir : il suffit de lancer le programme `rm`. Mais quel appel système `rm` utilise-t-il pour supprimer un fichier ?

Utilisons à nouveau notre vieil ami strace pour le découvrir. Ici, nous supprimons le fichier `bar` :

```
strace rm bar
...
unlink("bar")                           = 0
...
```

Nous avons supprimé tout un tas d'éléments sans rapport avec la sortie tracée, ne laissant qu'un seul appel à l'appel système mystérieusement nommé `unlink()`. Comme vous pouvez le voir, `unlink()` prend simplement le nom du fichier à supprimer, et renvoie zéro en cas de succès. Mais cela nous amène à une grande énigme : pourquoi cet appel système s'appelle-t-il unlink ? Pour obtenir la réponse à cette question, nous devons d'abord comprendre comment les fichiers sont organisés sur le disque.

6 Liens physiques (*hard links*)

Dans un système Unix, les fichiers et les répertoires (qui sont des fichiers spéciaux) sont organisés sous forme d'une arborescence avec une racine unique. Chaque fichier possède un nom de bas niveau, **le numéro d'inode**, ainsi qu'un nom de haut niveau qui est exposé à l'utilisateur (par exemple `bar.txt`).

Étudions une nouvelle façon d'ajouter une entrée dans l'arborescence du système de fichiers, via un appel système connu sous le nom de `link()`. L'appel système `link()` prend deux arguments, un ancien nom de chemin et un nouveau ; lorsque vous "liez" un nouveau nom de fichier à un ancien, vous créez essentiellement une autre façon de faire référence au même fichier. Le programme de ligne de commande `ln` est utilisé pour créer des liens et fait appel à `link()`.

Créez maintenant un fichier nommé `file` contenant la chaîne de caractère `hello`. Ensuite, utilisez `ln` pour créer une nouvelle référence `file2` vers le fichier `file`.

```
$ ln file file2
```

Que se passe-t-il lorsque vous affichez le contenu de `file2` avec `cat` ?

Le fonctionnement de `link()` consiste simplement à créer un autre nom dans le répertoire vers lequel vous créez le lien, et à le référencer au même numéro d'inode (c'est-à-dire au nom de bas niveau) du fichier original. Le fichier n'est en aucun cas copié ; vous avez simplement deux noms lisibles par l'utilisateur (`file` et `file2`) qui renvoient tous deux au même fichier.

Nous pouvons même le constater dans le répertoire lui-même, en affichant le numéro d'inode de chaque fichier :

```
$ ls -i file file2
```

En passant l'option `-i` à `ls`, la commande affiche le numéro d'inode de chaque fichier (ainsi que le nom du fichier). Vous pouvez ainsi voir ce que `ln` a réellement fait : créer une nouvelle référence vers le même numéro d'inode.

Vous commencez peut-être à comprendre pourquoi `unlink()` s'appelle ainsi. Lorsque vous créez un fichier, vous faites en réalité deux choses. Tout d'abord, vous créez une structure (l'inode) qui contiendra pratiquement toutes les informations pertinentes sur le fichier, y compris sa taille, l'emplacement de ses blocs sur le disque, et ainsi de suite. Deuxièmement, vous associez un nom lisible par l'utilisateur à ce fichier et placez ce lien dans un répertoire.

Après avoir créé un lien physique vers un fichier, il n'y a pas de différence entre le nom de fichier original (`file`) et le nom de fichier nouvellement créé (`file2`). En effet, tous deux ne sont que des liens vers les métadonnées sous-jacentes du fichier, qui se trouvent dans un inode donné.

Supprimons à présent le fichier `file` grâce à la commande `rm` qui fait usage de l'appel système `unlink()`.

```
$ rm file
```

Essayez à présent d'afficher le contenu du fichier `file2` avec `cat` ? Qu'observez-vous ?

La raison pour laquelle cela fonctionne est que, lorsque le système de fichiers *unlink* un fichier, il vérifie un nombre de références dans le numéro d'inode. Ce nombre de références (parfois appelé *link count*) permet au système de fichiers de savoir combien de noms de fichiers différents ont été liés à cet inode particulier. Lorsque la fonction `unlink()` est appelée, elle supprime le "lien" entre le nom lisible par l'utilisateur (le fichier à supprimer) et le numéro d'inode donné, et décrémente le nombre de références ; **ce n'est que lorsque le nombre de références atteint zéro que le système de fichiers libère également l'inode et les blocs de données associés, et donc "supprime" réellement le fichier**.

Vous pouvez afficher le nombre de liens physiques vers un fichier donné avec la commande `stat` qui utilise l'appel système `stat()`.

```
$ echo hello > file
$ stat -f "filename: %N - inode number: %i - reference count: %l" file
```

Création de liens physiques

Créez de nouveaux liens physiques vers le fichier `file` et vérifiez que le nombre de références augmente bien en accord avec chaque nouveau lien créé.

7 Liens symboliques (*symbolic links*)

Il existe un autre type de lien très utile, appelé lien symbolique ou *soft link*. Les liens physiques sont quelque peu limités : vous ne pouvez pas en créer un vers un répertoire (pour éviter les cycles dans l'arborescence des répertoires) ; vous ne pouvez pas créer de lien en dur vers des fichiers situés dans d'autres partitions de disque (parce que les numéros d'inode ne sont uniques qu'à l'intérieur d'un système de fichiers particulier, et non d'un système de fichiers à l'autre) ; etc. Un nouveau type de lien, appelé lien symbolique, a donc été créé.

Pour créer un tel lien, vous pouvez utiliser le même programme `ln`, mais avec l'option `-s`. Voici un exemple :

```
$ ln -s file file_s
```

Essayez alors d'afficher le contenu du fichier `file` en utilisant le lien symbolique `file_s`.

Comme vous pouvez le constater, la création d'un lien symbolique se déroule de la même manière, et le fichier d'origine est désormais accessible par le nom du fichier ainsi que par le nom du lien symbolique `file_s`. Cependant, au-delà de cette similitude superficielle, les liens symboliques sont en fait très différents des liens physique. La première différence est qu'un lien symbolique est en fait un fichier lui-même, d'un type différent. Nous avons déjà parlé des fichiers et des répertoires ordinaires ; les liens symboliques constituent un troisième type de fichier connu du système de fichiers.

Vérifiez qu'on observe bien ce qui est évoqué juste avant en affichant le numéro d'inode du fichier `file` puis celui du lien symbolique `file_s`.

L'exécution de `ls` révèle également ce fait.

```
$ ls -al
```

Si vous regardez de près le premier caractère de la forme longue de la sortie de `ls`, vous pouvez voir que le premier caractère de la colonne la plus à gauche est un `-` pour les fichiers normaux, un `d` pour les répertoires, et un `l` pour les liens symboliques. Vous pouvez également voir la taille du lien symbolique (4 octets dans ce cas) et ce vers quoi le lien pointe (le fichier nommé `file`). La raison pour laquelle 'file_sù a une taille de 4 octets est que la manière dont un lien symbolique est formé consiste à conserver le nom du chemin d'accès du fichier lié comme données du fichier de lien.

Enfin, supprimez le fichier `file` puis essayer d'accéder au contenu du fichier avec `cat` via le lien symbolique `file_s`. Que se passe-t-il ?

En raison de la manière dont les liens symboliques sont créés, ils laissent la possibilité à ce que l'on appelle une *dangling reference* ; contrairement aux liens physiques, la suppression du fichier d'origine fait pointer le lien vers un chemin qui n'existe plus.

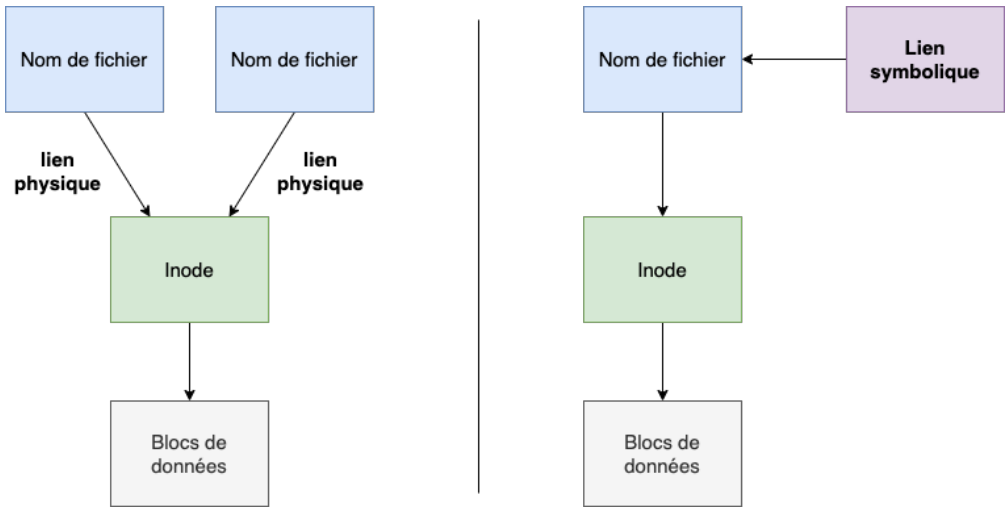

Figure 1: **Lien physique et lien symbolique**

8 Pour aller plus loin

Ce TP ne couvre qu'une partie de l'interface du système de fichiers Unix. Pour en savoir plus, notamment sur les répertoires et la gestion des droits, vous pouvez consulter le chapitre 39 d'Operating Systems: Three Easy Pieces.

9 Références

[CK+08] "The xv6 Operating System" par Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich. Source: https://github.com/mit-pdos/xv6-public.

À retenir

- Un fichier possède un nom de bas niveau, le **numéro d'inode**, qui le désigne de manière unique.
- Pour accéder à un fichier, un processus doit utiliser un appel système (généralement, `open()`) pour demander l'autorisation au système d'exploitation. Si l'autorisation est accordée, le système d'exploitation renvoie un **descripteur de fichier**, qui peut alors être utilisé pour l'accès en lecture ou en écriture, en fonction des autorisations et de l'intention.
- Chaque **descripteur de fichier** est une entité propre à un processus, qui fait référence à une entrée dans la tableau des fichiers ouverts. Cette entrée indique le fichier auquel cet accès se réfère, l'*offset* actuel du fichier (c'est-à-dire la partie du fichier à laquelle la prochaine lecture ou écriture accédera) et d'autres informations pertinentes.
- Les appels à `read()` et `write()` mettent naturellement à jour l'*offset* actuel ; sinon, les processus peuvent utiliser `lseek()` pour modifier sa valeur, ce qui permet un accès aléatoire à différentes parties du fichier.
- L'appel système `rename()` permet de modifier le nom ou le chemin d'un fichier.
- Pour que plusieurs noms lisibles par l'utilisateur dans le système de fichiers renvoient au même fichier sous-jacent, utilisez des **liens physiques** ou des **liens symboliques**. La commande `ln` et l'appel système `link()` permettent de réaliser cela.
- Un **lien physique** est un fichier à part entière, et la référence du fichier renvoit à l'inode du fichier pour lequel le lien physique a été créé.
- Un **lien symbolique** est un type particulier de fichier qui contient le nom (ou chemin) du fichier original. Il ne pointe pas vers l'inode de ce dernier.
- La suppression d'un fichier consiste simplement à effectuer un dernier `unlink()` de ce fichier dans la hiérarchie des répertoires.
- `strace` permet de tracer chaque appel système effectué par un programme pendant qu'il s'exécute, et d'afficher la trace à l'écran.
- La commande `stat` permet d'obtenir des informations sur le statut d'un fichier, notamment le nombre de liens physiques qui pointent vers l'inode de ce fichier.