Programmation système : i-nœuds

Juliusz Chroboczek

12 Octobre 2009

Le disque d'un ordinateur est une ressource partagée : il est utilisé par plusieurs programmes et, sur un système multi-utilisateur, par plusieurs utilisateurs.

Les données sur le disque sont stockées dans des *chiers* (*le* en anglais), des structures de données qui apparaissent aux programmes comme des suites nies d'octets. La structure de données qui organise les chiers sur le disque s'appelle le *système de chiers (le system)*.

1 Vision abstraite du système de fichiers

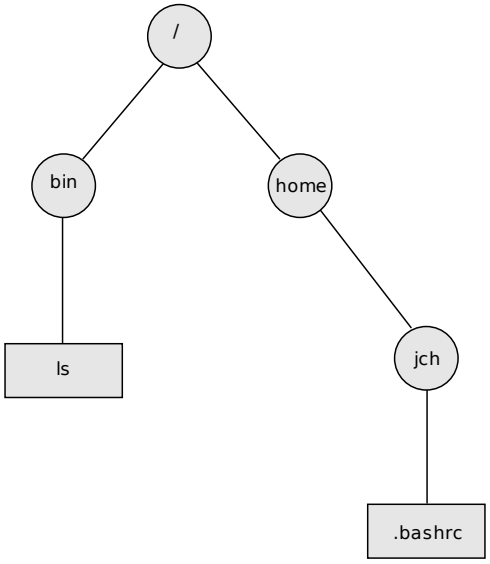

Figure 1: Vision abstraite du système de chiers

Du point de vue du programmeur et de l'utilisateur, le système de chiers apparaît comme un arbre dont les feuilles sont les chiers (gure 1). Un nœud interne de cet arbre s'appelle un *répertoire (directory)*.

Un chier est identié par son *chemin d'accès (pathname)*, qui est constitué du chemin depuis la racine dont les composantes sont séparées par des *slashes* « / ». Par exemple, le chemin d'accès du chier en bas à droite de la gure 1 est `/home/jch/.bashrc`.

2 Vision concrète du système de fichiers

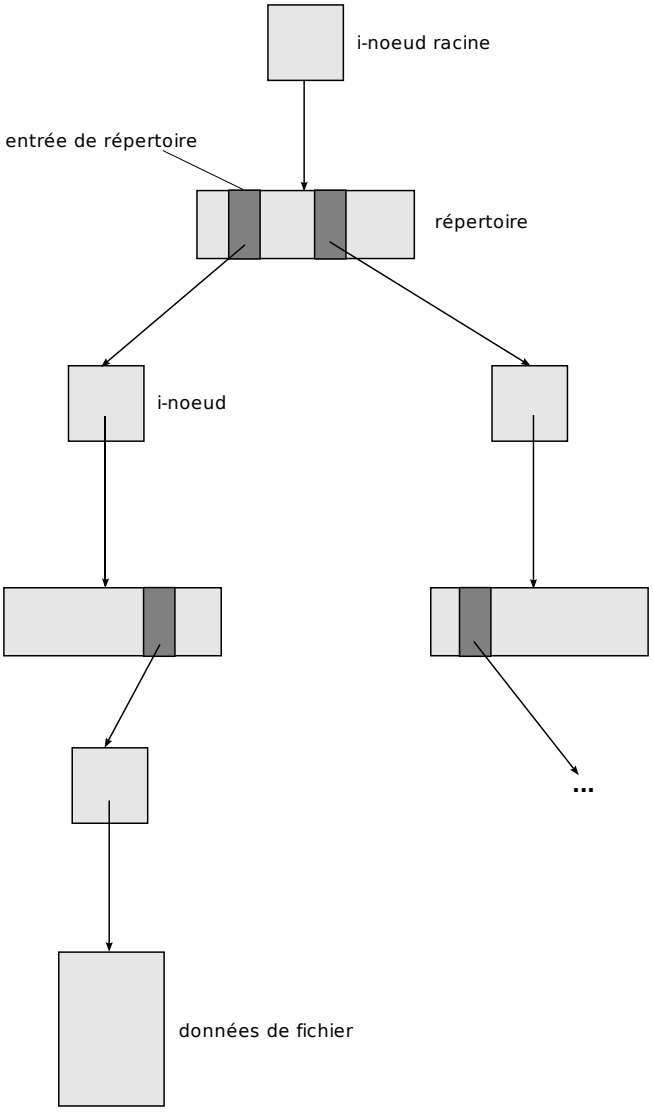

Figure 2: Vision concrète du système de chiers

Concrètement (gure 2), le système de chiers est constitué de deux types de structures de données : les *i-nœuds (i-nodes)* et les *données de chier*. Ces dernières peuvent être soit des *données*

de chier ordinaire, ou des *répertoires*.

Intuitivement, l'i-nœud « c'est » le chier. Un i-nœud est une structure de taille xée (quelques centaines d'octets) qui contient un certain nombre de *méta-données* à propos d'un chier — son type (chier ou répertoire), sa taille, sa date d'accès, son propriétaire, etc. — ainsi que su samment d'informations pour retrouver le contenu du chier.

Le contenu du chier est (une structure de données qui code) une suite d'octets. Dans le cas d'un chier ordinaire, ce contenu est interprété par l'application, il n'a pas de signi cation pour le système. Dans le cas d'un répertoire, par contre, c'est le système qui l'interprère comme une suite d'*entrées de répertoire*, dont chacune contient un *nom de chier* et une *référence à un i-nœud*.

3 Structures de données en mémoire

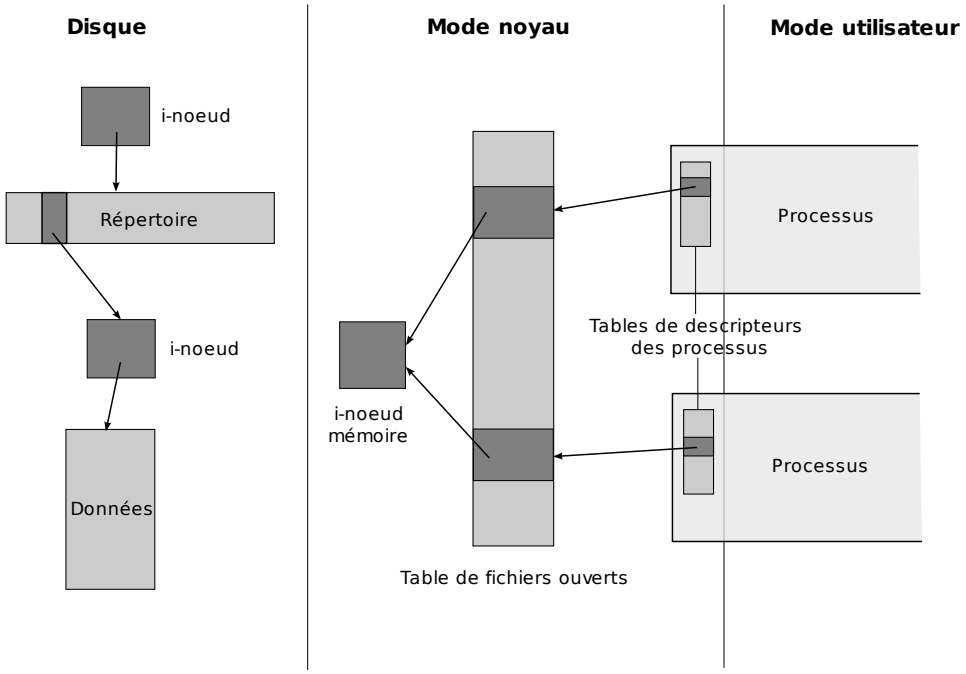

Figure 3: Structures de données sur disque et en mémoire

Lorsque l'utilisateur demande au système de manipuler un chier (en e ectuant l'appel système `open`, voir ci-dessous), celui-ci charge l'i-nœud correspondant en mémore ; il y a alors dans la mémoire du noyau un *i-nœud mémore*, une *entrée de la table de chiers ouverts* et une *entrée de la table de descripteurs de chier du processus*.

L'i-nœud mémoire est une structure de données contenant le contenu de l'i-nœud (disque) ainsi que quelques données comptables supplémentaires (notamment le numéro de périphérique à partir duquel cet i-nœud a été chargé).

L'*entrée de la table de chiers ouverts* est une structure de données globale qui réfère à l'i-nœud

mémoire ; cette structure de données contient elle-même quelques champs supplémentaires, notamment le *pointeur de position courante*, que nous verrons dans un cours futur.

L'*entrée de la table de descripteurs de chiers* est une structure qui réfère à l'entrée de la table de chiers ouverts. À la di érence des deux structures de doonnées ci-dessus, qui sont *globales*, il s'agit d'une structure qui est locale au processus ayant ouvert le chier.

Ces structures de données vivent dans la mémoire du noyau ; le code utilisateur ne peut donc pas y référer directement. Le code utilisateur indique une entrée de la table de chiers à travers un petit entier, l'indice dans la table de descripteurs de chiers, communément appelé lui-même *descripteur de chier*.

4 Appels système fichiers

Sous Unix, les entrées/sorties sur les chiers sont réalisées par les appels système `open`, `close`, `read`, `write`.

`open` L'appel système `open` est dé ni[1] par

```
int open(char *pathname, int flags,... /* int mode */);
```

Son rôle est de charger un i-nœud en mémoire, créer une entrée de la table de chiers qui y réfère, et créer une entrée dans la table de descripteurs de chiers du processus courant qui réfère à cette dernière. Il retourne le descripteur de chiers correspondant, ou -1 en cas d'erreur (et la variable globale `errno` est alors positionnée).

Le paramètre `pathname` est le nom du chier à ouvrir. Le paramètre `flags` peut valoir

- `O_RDONLY`, ouverture en lecture seulement ;
- `O_WRONLY`, ouverture en écriture seulement ;
- `O_RDWR`, ouverture en lecture et écriture.

Cette valeur peut en outre être combinée (à l'aide de la disjonction bit-à-bit « | ») avec

- `O_CREAT`, créer le chier s'il n'existe pas ;
- `O_EXCL`, échouer si le chier existe déjà ;
- `O_TRUNC`, tronquer le chier s'il existe déjà ;
- `O_APPEND`, ouvrir le chier en mode « ajout à la n ».

Le troisième paramètre, `mode_t mode`, n'est présent que si `O_CREAT` est dans `flags`. Il spéci e les permissions maximales du chier créé. On pourra le positionner à 666_8 (soit « 0666 » en C).

`close` L'appel système `close` libère une entrée de la table de descripteurs du processus courant, ce qui peut avoir pour e et de libérer une entrée de la table de chiers ouverts et un i-nœud en mémoire. Il est dé ni comme

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

Il retourne 0 en cas de succès, et -1 en cas d'échec[2]. Cependant, la plupart du code Unix traditionnel ne véri e pas le résultat retourné par `close` (ce qui cause parfois des problèmes sur les systèmes de chiers montés à travers le réseau).

1 Plus précisément, c'est la fonction *stub* correspondante qui est dé nie comme-ça.

2 Et la variable `errno` est alors positionnée. Je ne le mentionne plus, désormais.

`read` L'appel système `read` est dé□ni par

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

Sa fonction est de lire au plus `count` octets de données depuis le □chier spéci□é par le descripteur de □chiers `fd` et de les stocker à l'adresse `buf`.

L'appel `read` retourne le nombre d'octets e□ectivement lus, 0 à la □n du □chier, ou -1 en cas d'erreur.

`write` L'appel système `write` est dé□ni comme

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

Sa fonction est d'écrire au plus `count` octets de données qui se trouvent à l'adresse `buf` dans le □chier spéci□é par le descripteur de □chiers `fd`.

L'appel `write` retourne le nombre d'octets e□ectivement écrits[3], ou -1 en cas d'erreur.

Exemple Le programme suivant e□ectue une copie de □chiers. Comme nous le verrons, il est extrêmement ine□cace.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int fd1, fd2, rc;
    char buf;

    if(argc != 3) {
        fprintf(stderr, "Syntaxe: %s f1 f2\n", argv[0]);
        exit(1);
    }

    fd1 = open(argv[1], O_RDONLY);
    if(fd1 < 0) {
        perror("open(fd1)");
        exit(1);
    }

    fd2 = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if(fd2 < 0) {
```

3 Une écriture de moins de `count` octets s'appelle une *écriture partielle*. Une écriture partielle n'est normalement pas possible sur un □chier, mais elle est commune sur d'autres types de descripteurs de □chiers, par exemple les *pipes* ou les *sockets*.

```
        perror("open(fd2)");
        exit(1);
    }

    while(1) {
        rc = read(fd1, &buf, 1);
        if(rc < 0) {
             perror("read");
             exit(1);
        }
        if(rc == 0)
            break;

        rc = write(fd2, &buf, 1);
        if(rc < 0) {
            perror("write");
            exit(1);
        }
        if(rc != 1) {
            fprintf(stderr, "Écriture interrompue");
            exit(1);
        }
    }

    close(fd1);
    close(fd2);
    return 0;
}
```

Le programme décrit ci-dessus e□ectue deux appels système pour chaque octet copié — ce qui est tragique. Mon portable est capable d'e□ectuer jusqu'à 3 000 000 d'appels système par seconde ; ce programme ne sera donc pas capable de copier plus d'1,5 Mo de données par seconde environ. Nous verrons durant la séance prochaine des techniques qui permettront d'éviter ce problème.