École Normale Supérieure — Systèmes et réseaux
Année 2016-2017

TP2 : Manipulation du système de fichiers[1]

Attention : les exercices ci-dessous proposent de réaliser une commande de copie de fichiers nommée `mon_cp`. Faites attention de ne pas perdre ou corrompre vos données !

Récupérer les squelettes de programmes à compléter à l'adresse suivante :

`http://www.di.ens.fr/~pouzet/cours/systeme/tp02/tp-02.tgz`

1 Manipulation de fichiers avec la libC

La bibliothèque standard du C (la libC) propose notamment les fonctions suivantes pour accéder aux fichiers : `fopen`, `fread`, `fwrite`, `fclose` *(3)* . Ces fonctions ne manipulent pas directement les fichiers, mais une représentation abstraite de ceux-ci, des *flux*.

- `fopen` prend comme paramètre le chemin d'un fichier et le mode d'accès à ce dernier (lecture, écriture, lecture/écriture) et retourne un pointeur sur un flux.
- `fread` et `fwrite` permettent de lire et d'écrire dans un flux à partir ou vers un fichier.
- `fclose` permet de détruire le flux associé au fichier. Cette fonction écrit le contenu de ce flux dans le fichier si ce dernier était ouvert en écriture.

Le fichier `stdio.h` doit être inclus afin de disposer de ces fonctions. Il est important de noter que l'utilisateur doit passer par ces flux pour lire et écrire dans le fichier alors même que la structure de ces flux lui est totalement inconnue. En effet, l'implémentation des flux est à la charge des programmeurs de la libC utilisée et ils ne doivent être manipulés que par les fonctions ci-dessus depuis un programme utilisateur.

Historiquement, il existe une autre raison à l'utilisation courante des flux : la *bufferisation*, i.e., le stockage temporaire en mémoire de la partie « courante » du fichier, afin d'éviter d'accéder au disque à chaque fois qu'elle est modifiée ou relue, améliorant grandement les performances dans ces cas. Tous les systèmes d'exploitation modernes réalisent eux-mêmes une telle bufferisation, ce qui rend celle-ci inutile de ce point de vue, sauf dans des cas très particuliers. Par contre, la bufferisation présente toujours l'intérêt de diminuer le nombre d'appels systèmes pour transférer la même quantité de données. Les écritures dans un flux peuvent donc être retardée aussi longtemps que le flux n'a pas été fermé. Seules les fonctions `fclose`, qui ferme le flux, ou `fflush` permettent d'assurer que les données ont bien été transmises au système d'exploitation. Cela ne signifie pas pour autant que celles-ci seront immédiatement écrites sur le disque.

2 Manipulation de fichiers avec l'interface POSIX

Concernant la manipulation de fichiers, les fonctions qui nous intéressent sont `open`, `read`, `write` et `close` *(2)* . Elles se distinguent des méthodes de la libC car elles ne manipulent pas des flux, mais des descripteurs de fichiers. Un descripteur de fichier est un entier qui sert à référencer un fichier ouvert par un processus. Il s'obtient par l'utilisation de `open`

1. Ces exercices ont été créés par Philippe Dumont. Le texte a été créé par Louis Mandel.

pour les fichiers réguliers. Il est valable dès l'ouverture du fichier et jusqu'à sa fermeture. Contrairement aux fonctions de la libC, les accès aux fichiers sont immédiatement pris en compte par le système d'exploitation. Ils seront donc notamment immédiatement visibles depuis d'autres processus.

Question 1. Complétez le squelette de bibliothèque (ficher `file_manip.c`) en vous servant des fonctions `open`, `read`, `write` et `close`*(2)*. Testez de la bibliothèque avec le programme `mon_cp` fournit avec les squelettes à compléter.

3 La gestion des erreurs

Notre commande `mon_cp` peut facilement échouer. Par exemple, il est possible de lui fournir un nom de fichier source incorrect ou un répertoire comme fichier de destination. Par ailleurs, d'autres types d'erreurs peuvent survenir dans le code de notre bibliothèque, notamment lors de l'allocation mémoire des buffers. Dans de tels cas, l'erreur est automatiquement détectée dans la bibliothèque par notre fonction `fopen` et plus précisément lors de l'appel des fonctions `open` ou `malloc`. La libC utilise la variable `errno` afin de spécifier l'erreur qui s'est produite.

Question 2. Afin de rendre votre commande `mon_cp` plus facile à utiliser, affichez les erreurs éventuelles en modifiant `mon_cp.c`. Vous utiliserez pour cela le contenu de la fonction `perror`*(3)* ou la variable `errno`*(3)* et la fonction `strerror`*(3)*.

Pour l'instant, nous détectons les erreurs uniquement au niveau des appels systèmes. Mais une mauvaise utilisation de votre bibliothèque peut également entraîner des erreurs qui ne peuvent/doivent être vues par les appels systèmes.

Question 3. Étendez votre bibliothèque pour que cette dernière renvoie les erreurs pouvant survenir à son niveau. Vous devez pour cela utiliser `errno` comme le ferait la libC.

Laissez ces deux questions pour la fin du TD, s'il vous reste du temps.

Certaines erreurs pouvant survenir dans la bibliothèque ne doivent pas être propagées au programme principal. Par exemple, l'appel système `read` peut être interrompu par le système d'exploitation (signal), alors que l'interface de `fread` ne prévoit pas que ce cas se produise. Afin que notre bibliothèque respecte correctement l'interface, il est nécessaire de prendre en compte ce cas.

Question 4. Écrivez un exemple qui met en évidence ce comportement. (Notez que par défaut les signaux tuent un processus. Il faut enregistrer un routine de traitement avec `signal`*(2)* pour rester en vie. Pour faire bloquer `read`, pensez à utiliser `mkfifo`*(1)*.)

Question 5. Étendez votre bibliothèque pour que cette dernière gère automatiquement le cas où les appels systèmes `read` et `write` ont été interrompus par le système d'exploitation.

4 Copie dans un répertoire

Notre commande `mon_cp` se contente pour l'instant de copier un simple fichier dans un nouveau fichier. Ainsi le premier paramètre de notre application est le nom du fichier

à copier et le deuxième le nom du fichier après la copie. Nous voulons désormais que le deuxième paramètre puisse être soit le nom du fichier après la copie, soit le nom du répertoire dans lequel la copie doit avoir lieu (auquel cas le nom du fichier original est conservé).

Notre commande `mon_cp` doit donc déterminer le type du deuxième argument. L'utilisation d'un simple paramètre est exclu afin de respecter le fonctionnement de la commande `cp` originale et la présence ou non d'un `/` à la fin du deuxième argument n'est en rien significative. Nous devons donc nous servir de l'appel système `stat` *(2)* .

Fonctionnement de la fonction stat. La fonction `stat` permet de connaître les informations disponibles sur un fichier. Les informations sont communiquées à travers une structure de données de type `stat` qui doit être alloué par l'appelant. Les principaux champs de la structure sont :

```
struct stat {
    dev_t         st_dev;      /* Périphérique                 */
    ino_t         st_ino;      /* Numéro i-noeud               */
    mode_t        st_mode;     /* Droits                       */
    nlink_t       st_nlink;    /* Nb liens matériels           */
    uid_t         st_uid;      /* UID propriétaire             */
    gid_t         st_gid;      /* GID propriétaire             */
    dev_t         st_rdev;     /* Type périphérique            */
    off_t         st_size;     /* Taille totale en octets      */
    blksize_t     st_blksize;  /* Taille de bloc pour E/S      */
    blkcnt_t      st_blocks;   /* Nombre de blocs alloués      */
    time_t        st_atime;    /* Heure dernier accès          */
    time_t        st_mtime;    /* Heure dernière modification  */
    time_t        st_ctime;    /* Heure dernier changement état */
};
```

Nous n'avons besoin de nous intéresser qu'aux informations de `st_mode`. Pour vous en convaincre, affichez les droits d'un fichier et d'un répertoire avec la commande Unix `ls -l`, que constatez vous ?

Question 6. En vous aidant de la page de manuel de `stat`, modifiez votre programme pour qu'il gère de façon transparente la copie dans un répertoire et qu'il vérifie que le fichier source soit un fichier régulier.

Si le fichier destination se révèle être un lien « dur » sur le fichier source, alors notre programme `mon_cp` lorsqu'il ouvre le fichier destination efface notre fichier source.

Question 7. Toujours à l'aide de `stat`, vérifiez que les fichiers sources et destinations ne sont pas un lien « dur » de l'un sur l'autre. *Il vous suffit pour cela de comparer leur numéro d'inoeud et l'identifiant de leur périphérique de stockage.*