TD 1 : Manipulation du système de fichiers

Université Paris 7 — L3MI & EIDD — Systèmes et Réseaux

1 Via la bibliothèque standard du C

Toutes les fonctions sont accessibles via `#include <stdio.h>`

1.1 Ouverture

```
FILE *fopen(char *chemin, char *mode);
```

Ouvrir un fichier, renvoie un descripteur de fichier (*file handler*) ou NULL si erreur. `path` peut être absolu ou relatif. `mode` peut être :

- `"r"` lecture. Tête en début de fichier.
- `"w"` écriture. Écrase un fichier préexistant (taille ramenée à 0)
- `"a"` écriture sans écraser. Tête en fin de fichier
- `"r+"` lecture/écriture. Tête en début de fichier.
- `"w+"` lecture/écriture. Écrase un fichier préexistant.
- `"a+"` lecture/écriture. Tête en début de fichier.

1.2 Mode binaire

```
size_t fread(void *ptr, size_t taille_elem, size_t nombre_elem, FILE *stream);
size_t fwrite(const void *ptr, size_t taille_elem, size_t nombre_elem, FILE *stream);
```

le type `size_t` est un entier (32 ou 64 bit selon l'architecture). Les deux fonctions font un échange entre un tableau et un fichier (qui est respectivement lu et écrit ; donc pour le tableau c'est l'inverse !). Ledit tableau :

- a comme adresse `ptr`
- possède `nombre_elem` cases
- la taille (`sizeof()`) de chaque case étant `taille_elem`

Ainsi pour lire (ou écrire) un tableau T de 100 int, les trois premiers champs vaudront T, `sizeof(int)` et 100, respectivement. Il faut remarquer que pour la lecture depuis un disque (`fread()`), il faut avoir déjà alloué le tableau (pour l'écrire aussi bien sûr !). La fonction renvoie le nombre de *cases* lues ou écrites. Ainsi si le fichier fait 202 octets, `fread()` lira 200 octets (50 ints) et renverra 50, même si on lui demande d'en lire 100. Une valeur de retour nulle signale une erreur ou une fin de fichier.

1.3 Mode texte

Ce sont les même que les entrées/sorties au terminal, mais en spécifiant un `FILE *`, alors que les fonctions que vous connaissez utilisent implicitement `stdin` ou `stdout`

`int fprintf(FILE *stream, const char *format, ...);` au lieu de `int printf(const char *format, ...);`. Notez que c'est une fonction à nombre d'arguments variable, d'où les "..." dans le prototype.

`int fscanf(FILE *stream, const char *format, ...);` au lieu de `int scanf(const char *format, ...);`

`char *fgets(char *s, int size, FILE *stream);` lit au plus `size` caractères et les copies dans `s` qui doit avoir été allouée. Noter que `gets` a une syntaxe un peu différente.

`int fputs(const char *s, FILE *stream);` écrit une chaîne dans le fichier. On n'a pas à donner le nombre de caractères car la chaîne est terminée par un '\0'.

`int fgetc(FILE *stream);` est comme `int getchar(void);` et `int fputc(int c, FILE *stream);` comme `int putchar(int c);`. Noter qu'un caractère est ici un `int` et non un `char`.

1.4 Manipulation de la tête de lecture

Les fonctions de lecture/écriture sont séquentielles, c'est-à-dire que la tête de lecture/écriture virtuelle est déplacée après chaque lecture/écriture de sorte que la suivante commencera à la fin de la précédente. Si l'on veut déplacer la tête ailleurs :

`long ftell(FILE *stream)` donne l'*offset* (position de la tête) courant.

`int fseek(FILE *fich, long offset, int mode);` déplace la tête du fichier `fich` de `offset octets`, un nombre positif signifiant "vers la fin du fichier" et un négatif "vers le début de fichier". Le mode peut être
- `SEEK_CUR` : depuis la position courante.
- `SEEK_SET` : depuis le début du fichier.
- `SEEK_END` : depuis la fin du fichier.

2 Appels système pour les fichiers

Attention, les appels systèmes utilisent un entier comme descripteur de fichier et non un `FILE *`.

2.1 Ouverture

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode\_t mode);
```

Les flags sont une combinaison booléenne (par |) de constantes. Au moins l'un des trois premiers ce cette liste doit être donné. Il y en a d'autres, voir `man 2 open`.
- `O_RDONLY` lecture seule
- `O_WRONLY` écriture seule
- `O_RDWR` lecture/écriture
- `O_APPEND` Ajoute en queue d'un fichier préexistant
- `O_CREAT` Crée un fichier en cas de besoin (en absence de ce flag : erreur). Le mode dit alors les permissions du fichier créé, voir `man 2 open`
- `O_TRUNC` Écrase un fichier préexistant (necessite ouverture en écriture)

Retourne un descripteur de fichier entier, -1 en cas d'erreur.

2.2 Lecture/écriture

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nb);
ssize_t write(int fd, const void *buf, size_t nb);
```

Respectivement, tente de lire ou d'écrire `nb` octets entre le fichier de descripteur `fd` et la mémoire pointée par `buf`. La valeur de retour est le nombre d'octets effectivement lus/écrits ; -1 en cas d'erreur.

2.3 fermeture

```
#include <unistd.h>
int close(int fd);
```

Ferme le fichier. Renvoie -1 ou 0 selon qu'il y a erreur ou non.

2.4 Manipulation de la tête de lecture

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int mode);
```

Déplace la tête de lecture. `offset` et `mode` fonctionnent exactement comme `fseek()`. La valeur de retour est le nouvel offset (depuis le début du fichier ; -1 en cas d'erreur), ce qui permet aussi d'avoir le même comportement que `ftell()`.

2.5 Obtention des attributs du fichier

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

Ces deux fonctions remplissent une `struct stat` avec des informations sur les attributs du fichier (voir `man 2 stat` pour les détails de cette structure). Elle doit avoir été allouée. `stat()` prend un nom de fichier et `fstat()` un descripteur de fichier ouvert. La valeur de retour est 0 ou -1 en cas d'erreur.

2.6 Modification de l'accès au fichier

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd,...);
```

Fonction qui permet de modifier plein de choses pour l'accès au fichier : verrouillage exclusif ou partagé, accès bloquants ou non bloquants, gestion des signaux, notification en cas de modification du fichier/repertoire, ... Voir la page de manuel.

3 Appels système de manipulation de répertoire

Les fonctions du shell de manipulation de répertoire correspondent à des appels système. Ci-dessous `chemin` est un nom de fichier relatif ou absolu. Ces cinq fonctions renvoient -1 ou 0 selon qu'il y a eu une erreur ou non.

`mkdir (char *chemin, mode_t mode)`	crée un nouveau répertoire
`rmdir (char *chemin)`	supprime un répertoire vide
`link (char *source, char *dest)`	crée un lien physique comme `ln source dest`
`unlink (char *chemin)`	supprime un lien physique comme `rm chemin`
`chdir (char *chemin)`	modifie le répertoire de travail, comme `cd chemin`

Quatre autres appels systèmes sont utilisés pour lister le contenu d'un répertoire : `opendir()`, `readdir()`, `closedir()` et `rewinddir()`. Voir exercice 3 du TP (L3MI) ou page de manuel. En particulier, chaque appel à `readdir()` va renvoyer l'*entrée de répertoire* suivante du répertoire passé en paramètre, ce qui permet de le lister.

4 Exercices

1. Donner cinq façons d'accéder au fichier `/tmp/truc.txt` depuis un repertoire donné
2. Pourrait-on imaginer qu'au lieu de faire un `open()` préalable, on puisse passer directement à `read()` ou `write()` le chemin d'accès à un fichier ? Quelles différences cela ferait-il ?
3. Pourquoi `tell()` ou `ftell()` permettent-elles d'obtenir des tas d'informations sur un fichier, mais pas son nom ?
4. Pourquoi `rm` (shell) ou `unlink()` (appel système) effacent-ils seulement le lien, et pas le fichier ?
5. Il n'y a pas d'appel système correspondant à la commande `mv` du shell, que ce soit pour déplacer ou renommer un fichier. Pourquoi ?
6. L'appel système `lseek()` est-il vraiment indispensable, ou bien on pourrait s'en passer ? Comment ?
7. Pourquoi n'y a-t-il pas d'appel système `writedir()`, en symmétrie à `readdir()` ?
8. Pourquoi ne peut-on supprimer que des répertoires vides ?
9. Quels sont les avantages et inconvéniants respectifs des liens physiques et des liens symboliques ?
10. Sous MS-DOS il existe un attribut *archive* pour les fichiers. Un programme d'archivage n'archive que les fichiers qui ont ce flag à 1 et le remet à 0. Toute création ou modification de fichier le met à 1. Sous Unix, ce flag n'existe pas. Pourquoi, et par quoi le remplace-t-on ?
11. Pourquoi la taille d'un fichier est-elle un attribut de ce fichier ? Cela ne crée-t-il pas une possible incohérence ? Ne pourrait-on pas la déduire de ses données ?