Les fichiers

Les programmes que vous avez développés jusqu'à maintenant sont, certes, intéressants, mais ils sont très peu adaptatifs. Vous savez interagir avec l'utilisateur via le terminal, mais il n'est pas encore capable de vous fournir un grand nombre de données. Nous allons donc aborder dans ce cours les fichiers, ou plus précisément, nous allons étudier comment transférer des données du disque dur vers la mémoire utilisée par l'exécutable : la RAM. Ce chapitre est assez succinct puisque la théorie liée aux fichiers est elle-même réduite. En revanche, il vous faudra réaliser de nombreux TP pour être totalement à l'aise avec cette notion.

1 Introduction et rappels

Définition 1 (Fichier). Un fichier est une série d'octets enregistrés sur une mémoire de masse, et est identifié par à un nom.

Autrement dit, tout élément présent sur votre disque dur est un fichier. Ils sont, comme déjà vu, organisés en dossier et possède donc un *chemin absolu* Pour les repérer. Ce chemin se termine par le nom du fichier : il peut donc exister sur le disque plusieurs fichiers au nom identique mais au contenu différent (ou pas).

Exemple 1. Considérons le fichier ayant le chemin absolu suivant : `/home/TeamIbijau/Images/Potoo.jpg`. Ce fichier est vraisemblablement une image (car terminé par l'extension `jpg`) et il est situé dans le dossier `Images` de l'utilisateur `TeamIbijau`. Il est constitué d'une série d'octets qui, une fois interprétés par un logiciel de visualisation d'images, permet certainement d'afficher un bel oiseau. ◢

Un fichier n'a donc d'intérêt que lorsqu'il est *utilisé* par un logiciel. Cela signifie que le logiciel doit, à un moment donné, *accéder* à ce fichier sur le disque. Par accès, on entend en réalité transférer le fichier de l'espace de stockage vers la mémoire de travail : la mémoire vive. C'est le seul moyen pour le programme d'utiliser les octets du fichier afin de les interpréter. Voilà donc l'objectif de ce chapitre : accéder à un fichier afin de transférer son contenu dans la mémoire vive et interpréter ses octets.

Le processus d'accès à un fichier est similaire à celui d'un livre dans la vie courante. Une fois le livre choisi, on l'ouvre pour accéder à la page qui nous intéresse, on le parcourt séquentiellement pour le lire puis une fois terminé, on ferme le livre pour en laisser l'accès aux autres utilisateurs. En C, le principe est identique et on utilise des fonctions de la bibliothèque `stdio.h`.

Voyons maintenant la pratique avec les trois parties du traitement d'un fichier :

- l'ouverture afin de verrouiller les droits d'accès sur le fichier ;
- l'accès en lecture ou écriture ;
- la fermeture du fichier.

2 Ouverture et fermeture du fichier

La partie ouverture du fichier a deux utilités principales. La première est de réserver l'accès au fichier auprès du système d'exploitation. En effet, ce dernier vérifie que les droits d'accès nous sont bien attribués puis nous autorise à y accéder en verrouillant potentiellement l'accès aux autres programmes. La seconde utilité est d'initialiser la gestion de l'accès pour notre programme.

L'accès au fichier se fait grâce à la manipulation d'un *pointeur sur fichier*. Ce pointeur, de type `FILE*` est en réalité un pointeur sur une structure de type `FILE`. Ce type structuré contient différents champs qui, pour une fois, ne nous regardent pas ! En effet, nous manipulerons les pointeurs sur fichier sans jamais accéder au contenu du pointeur. Pour simplifier, il vous suffit de considérer un pointeur sur un fichier comme un curseur d'un éditeur de document.

L'ouverture se réalise donc à l'aide de la fonction `fopen`. Rappelons que cette fonction est disponible via la bibliothèque `stdio`. Étudions sa syntaxe et son utilisation.

```
FILE * fopen ( const char * filepath , const char * mode );
```

La fonction retourne un pointeur de type `FILE`, qui est donc notre fameux pointeur sur fichier. La variable qui va récupérer cette adresse sera donc celle que l'on réutilisera dans les fonctions de lecture et écriture. La paramètre `filepath` est le chemin d'accès au chemin. Comme tout chemin, il peut être absolu ou relatif. Si le chemine est valide, la fonction retourne un pointeur valide, sinon la fonction retourne `NULL`. Il est donc simple de tester le retour de la fonction pour éviter de manipuler un pointeur nul par la suite.

La paramètre `mode` est une chaine de caractères qui vous permet de spécifier le type d'ouverture à utiliser. Comme pour un livre, il est possible de lire ou écrire, mais l'équipement est différent : il faut un stylo pour écrire. Le principe est ici le même, les caractères présents dans la chaine seront différents selon ce que l'on veut faire du fichier.

Mode	Description	Commentaire	Position pointeur
`r`	Lecture	Erreur si fichier absent	Début
`w`	Écriture	Crée ou écrase	Début
`a`	Ajout	Crée si absent	Fin
`r+`	Lecture + écriture	idem lecture	Début
`w+`	Écriture + lecture	idem écriture	Début
`a+`	Ajout + lecture	idem ajout	Début ou fin
`b`	Données binaires	–	–

FIGURE 1 – Liste des mode d'ouverture disponibles pour `fopen`

La tableau 1 liste les différents modes de lecture disponibles pour le paramètre `mode` de la fonction `fopen`. Par exemple, le mode `r` permet d'ouvrir un fichier en lecture, si le fichier n'est pas accessible, la fonction retourne une erreur et ne crée pas automatiquement le fichier. Une fois le fichier ouvert, le pointeur sur fichier est situé sur le premier octet de ce dernier. Le mode `+` permet de lire et écrire en même temps, mais nous recommandons de NE PAS utiliser ce mode. Le mode `b` permet de lire un fichier sans interpréter les

octets lus. C'est nécessaire pour toute lecture de fichier binaire tels que des images, des exécutables, bref, tout ce qui n'est pas du texte pur.

Exemple 2. Si on souhaite ouvrir le fichier `bibi.jpg` qui est dans le dossier courant, pour simplement parcourir le fichier et stocker ses octets dans la mémoire vive, on écrira les instructions suivantes.

```
FILE *pfile = NULL;
pfile = fopen("bibi.jpg", "rb");
```

Le pointeur `pfile` sera utilisé par la suite pour lire les données du fichier `bibi.jpg`. Le mode binaire est utilisé car il s'agit d'une image. Un usage propre veut que le pointeur `pfile` soit testé afin de vérifier qu'il n'est pas à nul et quitter le programme le cas échéant. ◢

TD **Exercice 1.**

La seconde fonction est celle permettant de libérer l'accès au fichier et de libérer la mémoire allouée pour le pointeur sur fichier. Il s'agit de la fonction `fclose` prenant en paramètres le pointeur sur fichier `stream`.

```
int fclose ( FILE * stream );
```

On appelle cette dernière une fois que l'on a fini de traiter le fichier. En règle générale, le traitement d'un fichier se fait de la façon suivante :

- on ouvre le fichier avec `fopen` et le mode d'ouverture adapté ;
- on traite le fichier avec une des fonctions de lecture ou écriture ;
- on ferme le fichier dès que possible avec `fclose`.

Les données du fichier doivent être lues en une seule fois, on ne réalise pas d'oprations dessus tant que l'intégralité du fichier n'a pas été lu. Une fois que les données sont stockées sur la RAM, on ferme le fichier puis on manipule les données. Cela permet de réduire le temps de réservation au fichier et ainsi éviter les erreurs.

TD **Exercice 2.**

3 Lecture du fichier

Une fois l'ouverture du fichier réussie, vous possédez un pointeur sur fichier contenant une adresse valide. Cette adresse est celle de la structure de type `FILE` qui a été allouée dynamiquement lors de l'ouverture. Inutile de chercher à connaitre les champs présents dans cette structure, il sont modifiés uniquement par le biais de fonctions de lecture et d'écriture, que nous allons lister maintenant.

Les fonctions de lecture suivantes sont accessibles avec les modes `r`, `w+` ou `a+`. Néanmoins, nous recommandons l'usage du mode `r` uniquement, en mode binaire ou pas. Toutes les fonctions suivantes consistent à lire le contenu du fichier et copier des séries d'octets vers la mémoire vive. Il faut pour cela allouer une quantité de mémoire suffisante, sur la pile ou le tas selon la quantité.

Lorsque les derniers octets du fichiers sont lus, les fonctions suivantes peuvent renvoyer la valeur `EOF` (End Of File) qui indique que la fin du fichier est atteinte. Il faut alors stopper la lecture puis fermer le fichier.

3.1 Lire octet par octet : fgetc()

La fonction `fgetc` permet de récupérer un caractère à partir du pointeur sur fichier `stream`. Comme toutes les autres fonctions de lecture, le curseur est automatiquement incrémenté après la lecture : il passe sur le caractère suivant. La valeur du caractère retourné peut alors être stockée dans une variable de type `int` [1] puis castée en `char` ou `unsigned char` si la valeur est positive. Une valeur négative (EOF) indique une erreur de lecture ou la fin du fichier.

```
int fgetc ( FILE * stream );
```

Exemple 3. Voici un usage correct de la fonction `fgetc` pour récupérer un octet. On suppose, comme précédemment, que le pointeur `pfile` est un pointeur sur fichier correctement initialisé.

```
int buf; unsigned char c;
buf = fgetc (pfile);
if (buf>=0) c = (unsigned char)buf;
```

3.2 Lire une chaine de caractères : fgets()

L'usage de la fonction `fgets` est similaire à celui de la fonction `fgetc` mais permet de récupérer une chaine de caractères (et non un seul caractère) à partir du pointeur sur fichier `stream`. Elle prend donc également en argument le pointeur `str` vers la zone mémoire allouée pour recevoir le contenu lu ainsi que le nombre maximum `num` d'octets à lire. La fonction arrête sa lecture dès qu'une fin de ligne ou que la fin du fichier est atteinte. Attention, la fonction récupérant une chaine de caractères, elle ajoute un `\0` en fin de chaine. La zone mémoire pointée par `str` doit donc avoir une taille supérieure ou égale à `num`. Si la lecture est correctement effectuée, la fonction retourne une adresse valide, `NULL` en cas d'erreur.

```
char * fgets ( char *str , int num , FILE *stream);
```

Exemple 4. Supposons que l'on veuille lire un fichier texte ligne par ligne. On alloue une zone mémoire suffisante pour récupérer le contenu du fichier puis on utilise `fgets` dans une boucle pour récupérer chaque ligne. On part du principe ici que l'on connait à l'avance le nombre de ligne et la taille maximale d'une ligne.

```
char buf[10][128]; // 10 lignes de 128 caractères max
for (int i=0; i<10; i++)
  fgets (buf[i], 127, pfile);
```

1. oui, `int` et pas `char`.

3.3 Lire un paquet d'octets : fread()

La fonction `fread` permet de récupérer `count` blocs, chacun de `size` octets à partir du pointeur sur fichier `stream`. Les données lues sont stockées à l'adresse `ptr` qui doit pointer sur une zone mémoire réservant au moins `count` × `size` octets. La fonction retourne le nombre de blocs correctement lus ; lorsque tout s'est bien passé, la valeur de retour est égal à `count`. Cette fonction est particulièrement utile pour récupérer des données binaire comme le contenu d'une image, d'un fichier ZIP ou autre contenu non purement textuel.

```
size_t fread (void *ptr , size_t size, size_t count, FILE *stream);
```

Exemple 5. Si l'on souhaite récupérer 20 octets du fichier ouvert précédemment, il nous faut dans un premier temps allouer un tableau de 20 octets (de façon statique ou dynamique, peut importe) puis utiliser `fread` sur le pointeur sur fichier pour remplir ce tableau.

```
char *buf = calloc (20, sizeof(char));
fread (buf, 1, 20, pfile);
```

3.4 Lire une chaine formatée : fscanf()

Si le fichier est un de type texte et que le contenu est formaté, la fonction `fscanf` est celle à privilégier. De façon analogue à la fonction `scanf`, elle lit le contenu du fichier désigné par le pointeur sur fichier `stream` et stocke le contenu lu selon le formatage `format` dans les pointeurs donnés dans les arguments variables. La fonction retourne le nombre d'éléments correctement affectés.

```
int fscanf ( FILE *stream , const char *format, ... );
```

Exemple 6. Supposons qu'on veuille lire le contenu d'un fichier CSV dans lequel les données sont séparées par des virgules, le tout sur plusieurs lignes. On suppose que chaque ligne commence par une chaine de caractères (variable selon le numéro de ligne), puis vient le mot clé `pasteque`, et se termine par trois entiers. Une façon de lire une de ces lignes peut être la suivante.

```
char str[32];
int a, b, c;
fscanf (pfile, "%s,pasteque,%d,%d,%d\n", str, &a, &b, &c);
```

Si tout se passe bien sur cet exemple, la fonction retournera 4.

TD **Exercice 3.**

4 Écriture

Les fonctions d'écriture sont des homologues des fonctions de lecture vues précédemment. Elles sont accessible à condition que le fichier ait été ouvert avec un mode d'écriture tel que `w`, `a` ou avec un mode `+`. Là encore, le choix de la fonction dépendra du format des données à écrire sur le fichier :

- pour écrire caractère par caractère : `fputc` ;
- pour écrire une chaine de caractères : `fputs` ;
- pour écrire un paquet d'octets sans les interpréter : `fwrite` ;
- pour écrire des données formatées : `fprintf`.

Notons que, tout comme les fonctions de lecture, il est possible d'utiliser ces fonctions pour lire et écrire des données à partir ou sur le terminal. Pour une fonction de lecture, il suffit de remplacer le paramètre `stream` par le mot-clé `stdin`, et par `stdout` pour l'écriture. Comme précédemment, une fois la fonction terminée, le curseur est incrémenté du nombre de caractères écrits. Voyons maintenant le prototype de ces fonctions d'écriture, dont les paramètres sont très similaires à ceux de la lecture.

```
int fputc ( int car, FILE *stream );
```

La fonction `fputc` permet d'écrire le caractère `car` dans le fichier désigné par le pointeur sur fichier `stream`. La fonction retourne la valeur du caractère écrit ou une valeur négative (EOF) en cas d'échec.

```
int fputs ( const char *str , FILE *stream );
```

La fonction `fputs` permet d'écrire la chaine de caractères `str` dans le fichier désigné par le pointeur sur fichier `stream`. La fonction retourne une valeur positive ou une valeur négative (EOF) en cas d'échec.

```
size_t fwrite ( const void *ptr , size_t size, size_t count, FILE
 ↪  *stream );
```

La fonction `fwrite` permet d'écrire les `count` premiers blocs de la zone mémoire pointée par `ptr`, chacun composés de `size` octets dans le fichier désigné par le pointeur sur fichier `stream`. La fonction retourne le nombre de blocs correctement écrits dans le fichier, si tout se passe bien, cette valeur doit donc être égale à `count`.

```
int fprintf ( FILE *stream , const char *format, ... );
```

La fonction `fprintf` a un fonctionnement et une syntaxe identique à celle de la fonction `printf`, à la différence qu'elle écrit les données dans le flux `stream` et non sur le terminal. La fonction retourne le nombre d'éléments correctement écrits.

5 Autres fonctions

Toutes les fonctions vues précédemment incrémentent le pointeur sur fichier du nombre de caractères lus ou écrits. Lorsque la fin du fichier est atteinte, ces fonctions peuvent retourner la valeur EOF. Notons alors l'existence de la fonction `feof` prenant en paramètres le pointeur sur fichier `stream` et retournant la valeur 0 si la fin de fichier est atteinte.

Il existe également des fonctions permettant de se déplacer dans le fichier. L'objectif est d'emmener le curseur à un autre endroit dans le fichier ou de récupérer le nombre d'octet passés depuis le début du fichier.

```
long int ftell ( FILE * stream );
```

La fonction `ftell` permet de retourner la position du curseur dans le fichier. Elle indique le nombre d'octets présents entre le début du fichier et la position actuelle du pointeur `stream`. Elle est particulièrement utile pour déterminer la taille d'un fichier.

```
int fseek ( FILE * stream, long int offset, int origin );
```

La fonction `fseek` permet de déplacer le pointeur sur fichier `stream` à partir d'une position indiquée par le paramètre `origin` et décalé de `offset` octets. La paramètre `origin` peut être choisi parmi les constantes `SEEK_SET` (début du fichier), `SEEK_CUR` (position actuelle du pointeur), ou `SEEK_END` (fin du fichier). On l'utilise principalement pour emmener le curseur à la fin du fichier pour déterminer par la suite sa taille à l'aide de la fonction `ftell`.

```
void rewind ( FILE * stream);
```

Comme sur les lecteurs cassette[2], cette fonction permet de ramener le pointeur sur fichier `stream` au début du fichier. Particulièrement utile lorsqu'on a déterminé la taille avec les fonctions précédentes pour lire le contenu du fichier à partir du premier octet.

TD **Exercices 4 et 5.**

2. Qui a dit « J'ai pas connu » ?!