CPE Lyon - 3ICS - Année 2021/22 Administration des systèmes Linux

TD6 – Gestion des fichiers

Emile METRAL

Ce TD aborde les différentes approches permettant de stocker des données dans un fichier ainsi que quelques notions concernant la gestion des périphériques et certains fichiers spéciaux.

1. Les périphériques

Les techniques présentées dans cette partie fournissent un accès direct aux pilotes de périphériques s'exécutant au sein du noyau Linux, et à travers eux aux dispositifs matériels connectés au système. Une mauvaise manipulation peut altérer ou endommager le système GNU/Linux.

Linux, comme la plupart des systèmes d'exploitation, interagit avec les périphériques matériels via des composants logiciels modulaires appelés pilotes de périphériques.

Un pilote masque les particularités des protocoles de communication utilisés par un dispositif matériel au système d'exploitation et lui permet d'interagir avec le périphérique par le biais d'une interface standardisée.

Sous Linux, les pilotes de périphériques font partie du noyau et peuvent être intégrés de façon statique à celui-ci ou chargés à la demande sous forme de modules. Les pilotes de périphériques s'exécutent comme s'ils faisaient partie du noyau et ne sont pas accessibles directement aux processus utilisateurs.

Cependant, Linux propose un mécanisme à ces processus pour communiquer avec un pilote – et par là même avec le dispositif matériel – via des objets semblables aux fichiers.

Ces objets apparaissent dans le système de fichiers et des applications peuvent les ouvrir, les lire et y écrire pratiquement comme s'il s'agissait de fichiers normaux.

Vos programmes peuvent donc communiquer avec des dispositifs matériels via des objets semblables aux fichiers soit en utilisant les opérations d'E/S de bas niveau de Linux soit les opérations de la bibliothèque d'E/S standard du C.

Linux fournit également plusieurs objets semblables à des fichiers qui communiquent directement avec le noyau plutôt qu'avec des pilotes de périphériques. Ils ne sont pas liés à des dispositifs matériels ; au lieu de

cela, ils fournissent différents types de comportements spécialisés qui peuvent être utiles aux applications et aux programmes systèmes.

Les fichiers de périphériques se divisent en deux types :

- Un périphérique caractère représente un dispositif matériel qui lit ou écrit en série un flux d'octets. Les ports série et parallèle, les lecteurs de cassettes, les terminaux et les cartes son sont des exemples de périphériques caractères.
- Un périphérique bloc représente un dispositif matériel qui lit ou écrit des données sous forme de blocs de taille fixe. Contrairement aux périphériques caractère, un périphérique bloc fournit un accès direct aux données stockées sur le périphérique. Un lecteur de disque est un exemple de périphérique bloc.

Les programmes traditionnels n'utiliseront jamais de périphériques blocs. Bien qu'un lecteur de disque soit représenté comme un périphérique matériel, le contenu de chaque partition contient habituellement un système de fichiers monté sur l'arborescence racine de GNU/Linux. Seul le code du noyau qui implémente le système de fichiers a besoin d'accéder au périphérique bloc directement ; les programmes d'application accèdent au contenu du disque via des fichiers et des répertoires normaux.

Un fichier de périphérique ressemble beaucoup à un fichier classique. Vous pouvez créer un fichier de périphérique en utilisant la commande **mknod** (man 1 mknod) ou l'appel système **mknod** (man 2 mknod). Créer un fichier de périphérique n'implique pas automatiquement que le pilote ou le dispositif matériel soit présent ou disponible ; le fichier de périphérique est en quelque sort un portail pour communiquer avec le pilote, s'il est présent.

La commande suivante crée un fichier de périphérique caractère appelé lp0 dans le répertoire courant. Ce périphérique a le numéro de périphérique majeur 6 et le numéro mineur 0. Ces nombres correspondent au premier port parallèle sur le système Linux :

```
mknod ./lp0 c 6 0
```

Les liens suivants fournissent une explication sur la notion de numéro majeur/mineur pour un périphérique, ainsi que la liste des numéros majeurs/mineurs utilisés :

- https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch03s02.html
- https://www.kernel.org/doc/html/latest/admin-guide/devices.html

Par convention, un système GNU/Linux inclut un répertoire /dev contenant tous les fichiers de périphériques caractère ou bloc des périphériques détectés. Les entrées de /dev ont des noms standardisés correspondants au numéros de périphérique majeur et mineur. Typiquement, seuls les administrateurs système et les développeurs utilisant des périphériques spécifiques ont besoin de créer leurs propres fichiers de périphérique. La plupart des distributions GNU/Linux proposent des utilitaires d'aide à la création de fichiers de périphérique standards avec les noms corrects.

Pour mémoire, actuellement lorsque vous partitionnez un nouveau disque dur, le système créer automatiquement les points d'accès au périphérique... Cela n'a pas toujours été le cas.

Il est possible d'accéder à un périphérique par deux méthodes :

- Vous pouvez même utiliser des commandes conçues pour les fichiers traditionnels comme cat ou less
  `cat document.txt > /dev/lp0`
- Dans un programme vous pouvez envoyer des données à un périphérique :

```
int fd = open ( "/dev/lp0" , O_WRONLY ) ;
write ( fd , buffer , buffer_length ) ;
close ( fd ) ;
```

Linux fournit également divers périphériques caractère ne correspondant à aucun périphérique matériel (périphériques spéciaux). Ces fichiers ont tous le numéro de périphérique majeur 1, qui est associé à la mémoire du noyau Linux et non à un pilote de périphérique :

- /dev/null
- /dev/zero
- /dev/random
- /dev/urandom
- /dev/loop0
- [1A] A quoi correspondent ces périphériques ?

```
- /dev/null (Le périphérique null):
    -   Linux rejette toutes les données écrites sur /dev/null. Un truc courant
est de spécifier /dev/null comme fichier de sortie dans un contexte où la sortie
est indésirable.
    - La lecture à partir de /dev/null entraîne toujours une fin de fichier. Par
exemple, si vous ouvrez un descripteur de fichier dans /dev/null en utilisant open
et que vous essayez de lire à partir du descripteur de fichier, read ne lira aucun
octet et retournera 0. Si vous copiez depuis /dev/null vers un autre fichier, la
destination sera un fichier de longueur zéro
- /dev/zero : L'entrée du périphérique /dev/zero se comporte comme s'il s'agissait
d'un fichier infiniment long rempli de 0 octets. Autant de données que vous
essayez de lire à partir de /dev/zero, Linux "génère" suffisamment de 0 octets.

Les périphériques spéciaux /dev/random et /dev/urandom permettent d'accéder à
l'installation de génération de nombres aléatoires intégrée au noyau Linux. La
différence entre les deux dispositifs s'affiche lorsque Linux épuise son magasin
de hasard.
- /dev/random : Si vous essayez de lire un grand nombre d'octets de /dev/random
mais ne générez aucune action en entrée (vous ne tapez pas, ne déplacez pas la
souris ou effectuez une action similaire), Linux bloque l'opération de lecture.
- /dev/urandom : Une lecture de /dev/urandom, en revanche, ne bloquera jamais. Si
Linux fonctionne au hasard, il utilise un algorithme cryptographique pour générer
des octets pseudoaléatoires à partir de la séquence passée d'octets aléatoires.
Bien que ces octets soient suffisamment aléatoires à de nombreuses fins, ils ne
passent pas autant de tests de hasard que ceux obtenus à partir de /dev/random.
- /dev/loop0 : Un périphérique loopback vous permet de simuler un périphérique
```

```
bloc en utilisant un fichier disque ordinaire. Imaginez un périphérique de disque
pour lequel les données sont écrites et lues à partir d'un fichier nommé disk-
image plutôt que vers et à partir des pistes et des secteurs d'un disque physique
réel ou d'une partition de disque. Chacun peut être utilisé pour simuler un seul
dispositif de bloc à la fois.
```

- [1B] En utilisant les fonctions open (man 3 open) et read (man 3 read), écrivez un programme qui affiche un nombre aléatoire sur un int.

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

/**
 * @brief Return a random integer between MIN and MAX, inclusive.  Obtain
randomness from /dev/random.
 *
 * @param min
 * @param max
 * @return int
 */
int random_number()
{
    static int dev_random_fd = -1;

    char *next_random_byte;

    int bytes_to_read;

    unsigned random_value;

    if (dev_random_fd == -1)
    {

        dev_random_fd = open("/dev/random", O_RDONLY);

        if(dev_random_fd == -1){
            return -1;
        }
    }

    next_random_byte = (char *)&random_value;

    bytes_to_read = sizeof(random_value);

    do
    {

        int bytes_read;
```

```
        bytes_read = read(dev_random_fd, next_random_byte, bytes_to_read);

        bytes_to_read -= bytes_read;

        next_random_byte += bytes_read;

    } while (bytes_to_read > 0);


    return random_value;
}

int main(int argc, char const *argv[])
{
    printf("%d\n", random_number());
    return 0;
}
```

Pour aller plus loin...

L'appel système `ioctl` est une interface destinée au contrôle de dispositifs matériels. Le premier argument de ioctl est un descripteur de fichier qui doit pointer sur le périphérique que vous voulez contrôler. Le second argument est un code de requête indiquant l'opération que vous souhaitez effectuer. Différents codes de requêtes sont disponibles pour chacun des périphériques. La plupart des codes de requête disponibles pour les différents périphériques sont listés sur la page de manuel de ioctl_list . L'utilisation de ioctl nécessite généralement une connaissance approfondie du pilote de périphérique du matériel que vous souhaitez contrôler.

```

```

2 Le système de fichiers /proc

Le pseudo-répertoire /proc est une fenêtre sur le noyau Linux en cours d'exécution. Les fichiers de /proc ne correspondent pas à des fichiers réels sur un disque physique. Les fichiers de /proc ne correspondent pas à des fichiers réels sur un disque physique. Il s'agit plutôt « d'objets magiques » qui se comportent comme des fichiers mais donnent accès à des paramètres, des structures de données et des statistiques du noyau. Le « contenu » de ces fichiers n'est pas composé de blocs de données, comme celui des fichiers ordinaires. Au lieu de cela, il est généré à la volée par le noyau Linux lorsque vous lisez le fichier. Vous pouvez également changer la configuration du noyau en cours d'exécution en écrivant dans certains fichiers du système de fichiers /proc.

La commande :

cat /proc/cpuinfo

affiche les informations concernant le CPU.

La plupart des fichiers de /proc donnent des informations formatées pour pouvoir être lues par des humains, cependant leur présentation reste suffisamment simple pour qu'elles puissent être analysées automatiquement

- [2A] A quoi correspondent les répertoires nommés par des numéros ? `Le nom de chaque répertoire est l'ID du processus correspondant.`
- [2B] A quoi correspond le fichier cmdline a l'intérieur d'un de ces répertoires ? `cmdline contient la liste des arguments du processus.`
- [2C] A quoi correspond le fichier cwd a l'intérieur d'un de ces répertoires ? `cwd est un lien symbolique qui pointe vers le répertoire de travail actuel du processus.`
- [2D] A quoi correspond le fichier exe a l'intérieur d'un de ces répertoires ? `exe est un lien symbolique qui pointe vers l'image exécutable en cours d'exécution dans le processus.`
- [2E] A quoi correspond le fichier root a l'intérieur d'un de ces répertoires ? `root est un lien symbolique vers le répertoire racine de ce processus. Habituellement, il s'agit d'un lien symbolique vers /, le répertoire racine du système. Le répertoire racine d'un processus peut être modifié en utilisant l'appel chroot ou la commande chroot.`
- [2F] Que contient le fichier /proc/devices ? `Le fichier /proc/devices répertorie les numéros de périphériques principaux pour les périphériques de caractères et de blocs disponibles pour le système.`
- [2G] Que pouvez vous dire du répertoire /proc/self ? `L'entrée /proc/self est un lien symbolique vers le répertoire /proc correspondant au process courant. La destination du lien /proc/self dépend du processus qui le regarde : chaque processus voit son propre répertoire process comme la cible du lien.`

3- Test des permissions d'accès a un fichier

L'appel système `access` détermine si le processus appelant à le droit d'accéder à un fichier. Il peut vérifier toute combinaison des permissions de lecture, écriture ou exécution ainsi que tester l'existence d'un fichier. L'appel `access` prend deux arguments. Le premier est le chemin d'accès du fichier à tester. Le second un OU binaire entre R_OK, W_OK et X_OK, qui correspondent aux permissions en lecture, écriture et exécution. La valeur de retour est zéro si le processus dispose de toutes les permissions passées en paramètre. Si le fichier existe mais que le processus n'a pas les droits dessus, access renvoie -1 et positionne errno à EACCES (ou EROFS si l'on a testé les droits en écriture d'un fichier situé sur un système de fichiers en lecture seule).

4- Verrous et opérations sur les fichiers

L'appel système `fcntl` est le point d'accès de plusieurs opérations avancées sur les descripteurs de fichiers. Le premier argument de `fcntl` est un descripteur de fichiers ouvert et le second est une valeur indiquant quelle opération doit être effectuée. Pour certaines d'entre elles, `fcntl` prend un argument supplémentaire. Nous verrons ici l'une des opérations les plus utiles de `fcntl` : le verrouillage de fichier. Consultez la page de manuel de `fcntl` pour plus d'informations sur les autres opérations.

- [4A] Pourquoi peut-on avoir besoin de verrouiller un fichier en écriture ? `Afin de signaler que le verrou a déjà été donné à un autre processus pour que aucun autre processus puisse écrire ou pour permettre de communiquer avec un autre processus qui aurait au préalable eut le verrou de lecture.`

- [4B] Écrire un programme qui écrit des valeurs aléatoires dans un fichier. Ce programme devra placer un verrou en écriture (F_WRLCK) sur le fichier. Que ce passe-t-il si vous exécutez deux fois le même programme depuis 2 shells différents. ? `Le deuxième processus reste bloqué à l'étape d'ouverture du fichier car il n'a pas le verrou.`

5- Buffers (Mémoire tampon)

Sur la plupart des systèmes d'exploitation, lorsque vous écrivez dans un fichier, les données ne sont pas immédiatement écrites sur le disque. Au lieu de cela, le système d'exploitation met en cache les données écrites dans un tampon en mémoire, pour réduire le nombre d'écritures disque requises et améliorer la réactivité du programme. Lorsque le tampon est plein ou qu'un événement particulier survient (par exemple, au bout d'un temps donné), le système écrit les données sur le disque.

Linux fournit un système de mise en cache de ce type. Normalement, il s'agit d'une bonne chose en termes de performances. Cependant, ce comportement peut rendre instables les programmes qui dépendent de l'intégrité de données stockées sur le disque. Si le système s'arrête soudainement – par exemple, en raison d'un crash du noyau ou d'une coupure de courant – toute donnée écrite par le programme qui réside dans le cache en mémoire sans avoir été écrite sur le disque est perdue.

La fonction fsync permet de « flusher » les tampons mémoires. L'appel fsync ne se termine pas tant que les données n'ont pas été physiquement écrites.

6 Parcours d'un répertoire

La fonction `stat` permet de récupérer des informations sur un fichier, comme sa taille , ses droits d'accès...

Toutes les informations renvoyées par `stat` sont stockées dans la structure `stat`, dont la déclaration est la suivante (voir man 2 `stat`) :

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */
    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */
#define st_atime st_atim.tv_sec      /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
```

```
            };
```

En parallèle, les fonctions `opendir` et `readdir` permettent respectivement d'ouvrir un répertoire et de lire le contenu du répertoire ouvert.

- [6A] En utilisant les fonctions précédentes, créez un programme qui affiche la taille cumulée de tous les fichiers contenus dans un répertoire.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <time.h>
#include <dirent.h>
#include <string.h>

int getSizeDirectory(char path[100], struct stat stats );
int getFileSize(struct stat stats);

int main()
{
    char path[100];
    struct stat stats;

    printf("Enter source file path: ");
    scanf("%s", path);


    // stat() returns 0 on successful operation,
    // otherwise returns -1 if unable to get file properties.
    if (stat(path, &stats) == 0)
    {
        printf("Size of the repertory : %d\n", getSizeDirectory(path, stats));
    }
    else
    {
        printf("Unable to get file properties.\n");
        printf("Please check whether '%s' file exists.\n", path);
    }

    return 0;
}

int getSizeDirectory(char path[100], struct stat stats ){
    int size = 0;
    DIR *dp;
    struct dirent *dirp;
    char pathFile[200];

    printf("path : %s\n", path);
```

```
    dp = opendir(path);

    while ((dirp = readdir(dp)) != NULL){
        strcpy(pathFile, path);
        strcat(pathFile, "/");
        strcat(pathFile, dirp->d_name);
        stat(pathFile, &stats);
        size += getFileSize(stats);
        //printf("size: %d name: %s\n", size, pathFile );
    }

    closedir(dp);

    return size;
}

int getFileSize(struct stat stats)
{
    // File size
    int size =  512 * stats.st_blocks;
    //printf("%d\n", size );
    return  size; // size of an block * numbers of block
}
```

7 Descripteurs de fichiers et fichiers binaires

Un descripteur de fichier est un entier qui identifie un fichier dans un programme C. Ne pas confondre un descripteur de fichier avec un pointeur de fichier. Les fonctions `fdopen`, `fopen`, `freopen` permettent d'obtenir un pointeur de fichier à partir d'un descripteur. Les fonctions `fread` et `fwrite` utilisent un pointeur de fichier.

La fonction `open` permet d'obtenir un descripteur de fichier à partir du nom de fichier sur le disque, de la même façon que `fopen` permet d'obtenir un pointeur de fichier. Une grande différence est que la fonction `open` offre beaucoup plus d'options pour tester les permissions. Le prototype de la fonction `open` est le suivant :

```
int open(const char *pathname, int flags, mode_t mode);
```

La fonction retourne une valeur strictement négative en cas d'erreur.

- [7A] Quelles sont les différentes valeurs (avec leurs symboles) possibles pour le paramètre flags ?

```
O_RDONLY, O_WRONLY, O_RDWR, O_TRUNC O_APPEND O_CREAT O_EXCL
```

- [7B] Que permet de faire le paramètre mode ? `Il permet de determiner les permissions.`

Le rôle de la fonction read est de lire au plus « count » octets de données depuis le fichier spécifié par le descripteur de fichiers fd et de les stocker à l'adresse buf.

L'appel `read` retourne le nombre d'octets effectivement lus, 0 à la fin du fichier, ou -1 en cas d'erreur.

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

Le rôle de la fonction write est d'écrire au plus « count » octets de données qui se trouvent à l'adresse « buf » dans le fichier spécifié par le descripteur de fichiers fd.

L'appel write retourne le nombre d'octets effectivement écrits, ou -1 en cas d'erreur.

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

Enfin, un appel a la fonction `fflush` permet, comme dans le cas de fsync, de « flusher les buffers»

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

- [7C] En utilisant les fonctions précédentes, écrire un programme qui sauvegarde la valeur 19496893802562113L dans un fichier binaire. Ouvrez le fichier. Qu'observez-vous ?

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

int main(int argc, char const *argv[])
{
    char *file = "BinaryFile7C.bin";
    FILE *pFile;
    pFile = fopen(file, "w");
    long int ByteArray = 19496893802562113L;
    if (pFile != NULL)
    {
        fwrite(&ByteArray, sizeof(ByteArray), 1, pFile);

        fclose(pFile);
    }
    return 0;
}
```

`J'observe un message "ABSURDE".`

```
emile@emile-virtual-machine:~/Documents/C_Syst-me/TD6/Exercice7$ cat
BinaryFile7C.bin
> ABSURDE
```

- [7D] De même, créez un programme qui enregistre la valeur 0x4142434451525354L dans un fichier, en utilisant les fonctions précédentes. Affichez la valeur avec un printf en décimal et hexadécimal ? Que contient le fichier binaire ?

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

int main(int argc, char const *argv[])
{
    char *file = "BinaryFile7D.bin";
    FILE *pFile;
    pFile = fopen(file, "w");
    long int ByteArray = 0x4142434451525354L;
    printf("Hexadecimal : %lx \n Decimal : %ld \n",ByteArray,ByteArray);
    if (pFile != NULL)
    {
        fwrite(&ByteArray, sizeof(ByteArray), 1, pFile);

        fclose(pFile);
    }
    return 0;
}
```

```
Le fichier binaire contient : TSRQDCBA et avec le printf il est affiche : Hexadecimal : 4142434451525354 Decimal : 4702394921629406036
```

- [7E] Enregistrez la valeur précédente dans un fichier en utilisant la fonction fprintf. Que constatez-vous ?

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

int main(int argc, char const *argv[])
{
    char *file = "BinaryFile7E.bin";
    FILE *pFile;
    pFile = fopen(file, "w");
    long int ByteArray = 0x4142434451525354L;
    printf("Hexadecimal : %lx \n Decimal : %ld \n",ByteArray,ByteArray);
    if (pFile != NULL)
    {
        fprintf(pFile, "%lo", ByteArray);

        fclose(pFile);
    }
    return 0;
}
```

```
$ cat BinaryFile7E.bin
405022064212124451524
```

On constate que le contenu est un chiffre different que le chiffre en decimal ou en hexadecimal.

- [7F] Quelle est la différence essentielle entre un fichier binaire et un fichier texte ?

```
Les fichiers binaires contiennent généralement une séquence d'octets, ou des
regroupements ordonnés de huit bits.Les formats de fichiers binaires peuvent
inclure plusieurs types de données dans le même fichier, tels que l'image, la
vidéo et les données audio. Ces données peuvent être interprétées en supportant
les programmes, mais apparaîtront comme du texte brouillé dans un éditeur de
texte.

Les fichiers texte sont plus restrictifs que les fichiers binaires puisqu'ils ne
peuvent contenir que des données textuelles. Cependant, contrairement aux fichiers
binaires, ils sont moins susceptibles de devenir corrompus. Alors qu'une petite
erreur dans un fichier binaire peut le rendre illisible, une petite erreur dans un
fichier texte peut simplement apparaître une fois que le fichier a été ouvert.
```

- [7G] Que pouvez-vous dire du principe 'little endian' et 'big endian' ?

```
Little Endian: Dans ce schéma, l'octet d'ordre inférieur est stocké sur l'adresse
de départ (A) et l'octet d'ordre supérieur est stocké sur l'adresse suivante (A +
1).

Big Endian : Dans ce schéma, l'octet d'ordre supérieur est stocké sur l'adresse de
départ (A) et l'octet d'ordre inférieur est stocké sur l'adresse suivante (A + 1).

adresse big-endian  petit endian
0x0000  0x12        0xcd
0x0001  0x34        0xab
0x0002  0xab        0x34
0x0003  0xcd        0x12
```

- [7H] A quelle groupe appartiennent les processeurs de la famille des Intel/AMD ? `Les processeurs Intel ont toujours été little-endian.`
- [7I] Donnez un modèle de processeur appartenant a l'autre groupe. `ARM CORTEX M4`
- [7J] Il existe d'autres fonctions permettant de lire et d'écrire dans un fichier, qui sont respectivement `fread` et `fwrite`. Quelles sont les différences entre `read` et `fread` ou `write` et `fwrite` ?

```
Fonctions fopen : fread, fwrite;
Fonctions open : read, write;

Les fonctions fopen sont des fonctions standard de la bibliothèque C, et les
```

```
fonctions Open sont définies par POSIX et sont des appels système dans les
systèmes UNIX.
C’est-à-dire que la fonction fopen est plus portable, alors que la fonction Open
ne peut être utilisée que dans les systèmes d’exploitation POSIX.

Lorsque qu'on utilise les fonctions fopen, on doit définir un objet qui se réfère
à un fichier. Il est appelé "file handler" et est une struct; la fonction Open
utilise un entier int appelé "descripteur de fichier.

Les fonctions fopen sont de haut niveau I / O, et des buffers sont utilisés pour
la lecture et l’écriture. Les fonctions open sont de niveau relativement basse,
plus proche du système d’exploitation, et il n’y a pas de buffer pour la lecture
et l’écriture. Parce qu'elles peuvent traiter avec plus de systèmes
d’exploitation, les fonctions open peuvent accéder et modifier certaines
informations qui ne peuvent pas être accessibles par les fonctions fopen, comme la
visualisation des permissions de lecture et d’écriture des fichiers. Ces
caractéristiques supplémentaires sont généralement propres au système.
```

Les fonctions de « stdio » n’opèrent pas sur des descripteurs de fichiers, qui sont une notion spécifique à Unix, mais sur des pointeurs sur une structure qui représente un flot, la structure FILE .

La définition de FILE dépend du système ; sur un système POSIX elle pourrait par exemple être définie comme suit :

```
#define BUFSIZ 4096
#define FILE_ERR 1
#define FILE_EOF 2
typedef struct _FILE {
16int fd;
int flags;
char *buffer
int buf_ptr;
int buf_end;
} FILE;
```

- [7K] Quelles informations importantes pouvez vous tirer du code précédent ? `On peut notifier la presence d'un file descriptor, d'un buffer avec deux entier pour definir le debut et la fin du buffer et enfin aue le flags est stocke dans las structure du pointeur du fichier.`
- [7L] En utilisant `fwrite`, écrire un programme qui enregistre 100 valeurs (de 0 a 100) de type `int` en binaire dans un fichier et les affiche simultanément. Que pouvez vous observer dans le fichier ?

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

int main(int argc, char const *argv[])
{
```

```
    char *file = "BinaryFile7L.bin";
    FILE *pFile;
    pFile = fopen(file, "wb+");

    if (pFile != NULL)
    {
        for (int i = 0; i < 100; i++)
        {
            fwrite(&i, sizeof(i), 1, pFile);
            printf("%d", i);
        }
        fclose(pFile);
    }
    return 0;
}
```

```
cat BinaryFile7L.bin



123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abc
```

On remarque que le contenu du fichier est les elements de la table ASCII de 0 a 100

- [7M] Écrire un second programme qui lit les valeurs précédentes du fichier et les affiche ?

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

int main(int argc, char const *argv[])
{
    char *file = "BinaryFile7L.bin";
    FILE *pFile;
    pFile = fopen(file, "rb");
    int read =  0;

    fseek(pFile, 0, SEEK_SET);

    if (pFile != NULL)
    {
        for (int i = 0; i < 100; i++)
        {
            fread(&read, sizeof(int), 1, pFile);
            printf("%d ", read);
            fseek(pFile, +1/2*sizeof(int), SEEK_CUR);
        }
        printf("\n");
        fclose(pFile);
    }
```

```
    return 0;
}
```

8- Fichiers séquentiels et fichiers a accès direct

Jusqu'à maintenant, vous avez lu les données d'un fichier les une après les autres.

Autrement dit vous avez lu de façon séquentielle les données présentes dans le fichier. Les lectures et les écritures dans un fichier sont par défaut séquentielles : les données sont lues ou écrites les unes à la suite de l'autre. Par exemple, lors de l'exécution de la séquence de code

```
rc = write(fd, "a", 1);
rc = write(fd, "b", 1);
```

l'octet « b » est écrit après l'octet « a ».

La position dans un fichier à laquelle se fait la prochaine lecture ou écriture est stockée dans le noyau dans un champ de l'entrée de fichier ouvert qui s'appelle le pointeur de position courante.

Lors de l'appel système open , le pointeur de position courante est initialisé à 0, soit le début du fichier. Le pointeur de position courante associé à un descripteur de fichier peut être lu et modifié à l'aide de l'appel système `lseek` :

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

Le paramètre « fd » est le descripteur de fichier dont l'entrée de la table de fichiers ouverts associée doit être modifiée. Le paramètre « offset » (« déplacement ») identifie la nouvelle position, et le paramètre « whence » (« à partir d'où ») spécifie l'interprétation de ce dernier. Il peut avoir les valeurs suivantes :

- SEEK_SET : offset spécifie un décalage à partir du début du fichier ;
- SEEK_CUR : offset spécifie un décalage à partir de la position courante ;
- SEEK_END : offset spécifie un décalage à partir de la fin du fichier.

L'appel `lseek` retourne la nouvelle valeur du pointeur de position courante, ou -1 en cas d'erreur (et alors errno est positionné). Par exemples, pour déplacer le pointeur de fichier au début d'un fichier, on peut faire lrc = lseek(fd, 0L, SEEK_SET); Notez que dans ce dernier cas, il existe aussi la fonction rewind qui ramène le curseur de lecture au début du fichier :

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

Pour déplacer le pointeur de fichier à la fin d'un fichier, on peut faire size = lseek(fd, 0L, SEEK_END);

NB : si l'appel réussit, la variable « size » contient la taille du fichier dans ce dernier cas.

Enfin, la fonction ftell donne la position courante du curseur de lecture :

```
long ftell(FILE* fd);
```

- [8A] Écrivez un programme qui enregistre les valeurs de 10 a 30 dans un fichier binaire.

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

int main(int argc, char const *argv[])
{
    char *file = "BinaryFile8A.bin";
    FILE *pFile;
    pFile = fopen(file, "wb");

    if (pFile != NULL)
    {
        for (int i = 10; i <= 30; i++)
        {
            fwrite(&i, sizeof(i), 1, pFile);
            printf("%d", i);
        }
        fclose(pFile);
    }
    return 0;
}
```

- [8B] Votre programme doit ensuite relire les données stockées a raison d'une valeur sur trois (vous devez utiliser lseek)

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/file.h>

int main(int argc, char const *argv[])
{
    char *file = "BinaryFile8A.bin";
    int pFile;
    pFile = open(file, O_RDONLY);
    int readVar = 0;

    lseek(pFile, 0, SEEK_SET);

    if (pFile != 0)
    {
        for (int i = 0; i < 7; i++)
        {
            read(pFile, &readVar, sizeof(int));
            printf("%d ", readVar);
            lseek(pFile, +2*sizeof(int), SEEK_CUR);
```

```
        }
        printf("\n");
        close(pFile);
    }
    return 0;
}
```

- [8C] Maintenant votre programme doit, en plus, lire la 5ieme valeur enregistrée dans le fichier.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/file.h>

int main(int argc, char const *argv[])
{
    char *file = "BinaryFile8A.bin";
    int pFile;
    pFile = open(file, O_RDONLY);
    int readVar = 0;

    lseek(pFile, 0, SEEK_SET);

    if (pFile != 0)
    {
        for (int i = 0; i < 7; i++)
        {
            read(pFile, &readVar, sizeof(int));
            printf("%d ", readVar);
            lseek(pFile, +2*sizeof(int), SEEK_CUR);
        }

        lseek(pFile, +5*sizeof(int), SEEK_SET);
        read(pFile, &readVar, sizeof(int));
        printf("5eme valeur : %d ", readVar);

        printf("\n");
        close(pFile);
    }
    return 0;
}
```

9-Sauvegarde d'une structure

L'enregistrement d'une structure dans un fichier doit être l'objet d'une attention particulière.

Prenez la structure suivante :

```
typedef struct {
    char nom[20];
```

```
    int age;
    float taille;
} Personne;
```

- [9A] Créez un tableau de 4 « Personne »

```
    Personne tab[4];


    strcpy(tab[0].nom, "eleve1");
    tab[0].age = 15;
    tab[0].taille = 1.98;

    strcpy(tab[1].nom, "eleve2");
    tab[1].age = 16;
    tab[1].taille = 1.58;

    strcpy(tab[2].nom, "eleve3");
    tab[2].age = 15;
    tab[2].taille = 1.65;

    strcpy(tab[3].nom, "eleve4");
    tab[3].age =14;
    tab[3].taille = 1.69;
```

- [9B] Affichez les données stockées dans les structures (nom, age et poids de chaque « Personne »)

```
    for (int i = 0; i < 4; i++)
    {
        printf("nom : %s, age: %d,taille : %.2f\n", tab[i].nom, tab[i].age,
tab[i].taille);
    }
```

- [9C] Créez une fonction qui sauvegarde le contenu du tableau dans un fichier binaire.

```
void saveInFile(Personne *personnes)
{
    char *file = "BinaryFile9C.bin";
    FILE *pFile;
    pFile = fopen(file, "wb");

    if (pFile != NULL)
    {
        fwrite(&personnes, sizeof(personnes), 1, pFile);

        printf("\n");
```

```
        fclose(pFile);
    }
}
```

- [9D] Créez une fonction qui lit les données du fichier précédent et les affiche au fur et a mesure de la lecture. Vous devez bien sur retrouver les données qui étaient stockées dans les structures.
- [9E] Créez une fonction qui lit les données du fichier précédents et les stocke dans un tableau de « Personne »

```
Marche aussi pour la 9D
void readInFile()
{
    char *file = "BinaryFile9C.bin";
    FILE *pFile;
    pFile = fopen(file, "rb");
    Personne *readVar = malloc(4*sizeof(Personne));

    if (pFile != 0)
    {
        fread(&readVar, sizeof(readVar), 1, pFile);
        for (int i = 0; i < 4; i++)
        {
            printf("nom : %s, age: %d,taille : %.2f\n", readVar[i].nom,
readVar[i].age, readVar[i].taille);
        }
        fclose(pFile);
    }

}
```

- [9F] Même exercice que précédemment, mais cette fois avec la structure suivante :

```
typedef struct {
    char *nom;
    int age;
    float taille;
} Personne;
```

CODE :

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/file.h>

typedef struct
```

```
{
    char *nom;
    int age;
    float taille;
} Personne;

void saveInFile(Personne *personnes)
{
    char *file = "BinaryFile9C.bin";
    FILE *pFile;
    pFile = fopen(file, "wb");

    if (pFile != NULL)
    {
        fwrite(&personnes, sizeof(personnes), 1, pFile);

        printf("\n");

        fclose(pFile);
    }
}

void readInFile()
{
    char *file = "BinaryFile9F.bin";
    FILE *pFile;
    pFile = fopen(file, "rb");
    Personne *readVar;

    if (pFile != 0)
    {
        fread(&readVar, sizeof(readVar), 1, pFile);
        for (int i = 0; i < 4; i++)
        {
            printf("nom : %s, age: %d,taille : %.2f\n", readVar[i].nom,
readVar[i].age, readVar[i].taille);
        }
        fclose(pFile);
    }

}

int main(int argc, char const *argv[])
{
    Personne tab[4];

    tab[0].nom = "eleve1";
    tab[0].age = 15;
    tab[0].taille = 1.98;

    tab[1].nom="eleve2";
    tab[1].age = 16;
    tab[1].taille = 1.58;
```

```
    tab[2].nom="eleve3";
    tab[2].age = 15;
    tab[2].taille = 1.65;

    tab[3].nom="eleve4";
    tab[3].age = 14;
    tab[3].taille = 1.69;

    for (int i = 0; i < 4; i++)
    {
        printf("nom : %s, age: %d,taille : %.2f\n", tab[i].nom, tab[i].age,
tab[i].taille);
    }

    saveInFile(tab);
    readInFile();

    return 0;
}
```

Sources :

- http://www.makelinux.net/alp/046.htm
- https://www.programiz.com/c-programming/c-file-input-output
- https://topic.alibabacloud.com/a/difference-between-fopenopen-readwrite-and-freadfont-colorredfwritefont_8_8_31848651.html