R3.05 - TP 1 - File System et fonctions système

Rappels de 1ère année

Arborescence

En 1ère année, vous avez appris à manipuler le Système de Fichiers qu'on appelle aussi File System (FS) dans le jargon du système d'exploitation. C'est ainsi qu'on va le nommer dans ce qui suit.

On vous a d'abord expliqué ce qu'est une arborescence, qui est une façon hiérarchique de classer les objets (fichiers et dossiers) du FS sur un support de stockage, afin d'en simplifier l'accès pour nous, êtres humains.

Ce support de stockage est soit local (le disque dur de votre ordinateur, un CD, une clé USB, etc.), soit distant (un volume réseau, le cloud).

Fichiers

Les fichiers sont les éléments terminaux de l'arborescence. Comme les feuilles d'un arbre, on ne peut pas aller plus loin, plus profondément dans l'arborescence.

Nous avons vu exclusivement des fichiers dits "standards" en 1ère année. Ces fichiers contiennent des données qui peuvent être formatées et lisibles directement (texte), formatées et lisibles par un logiciel adapté (image, son, vidéo, etc.) ou qui peuvent être des données brutes, sans cohérence apparente et qui servent généralement au système ou à certains logiciels (BDD par exemple) et sont rarement manipulés directement par l'humain.

Certains fichiers, quand ils sont associés à une application particulière, peuvent aussi être appelés des documents (Word, Excel, Figma, etc.)

Un fichier peut ne rien contenir, on dit qu'il est vide, sa taille (`ls -l`) fait zéro octets.

Cette année nous allons voir d'autres types de fichiers qu'on nomme "spéciaux". Nous allons voir ce qu'ils représentent et ce qu'on peut en faire, et ils sont une grande force d'Unix comme nous allons l'expérimenter.

Dossiers

Les dossiers sont des conteneurs, des branches de l'arborescence. Comme dans un arbre dans la nature, ces branches peuvent être le point d'attachement d'autres branches, de feuilles (les fichiers) ou des deux à la fois.

Même si c'est un arbre qui vient naturellement à l'idée quand on parle d'arborescence, on parle aussi de conteneur, comme s'il s'agissait de boîtes imbriquées les unes dans les autres. En 1ère année on avait même fait l'analogie avec des armoires contenant des boîtes à archives contenant des dossiers contenant des feuilles de papier, pour illustrer cette imbrication d'objets, avec la feuille de papier comme élément terminal, le fichier.

De cette notion de "conteneur", découle donc l'expression qu'on utilise généralement : "un dossier contient …".

Un dossier peut aussi être vide mais on a vu en 1ère année que l'information correspondant à sa taille (**`ls -l`**) ne semble pas refléter une réelle information sur la quantité contenue par un dossier. On va voir pourquoi cette année et ce que signifie réellement l'information de "taille" qui est affichée.

Racine

L'arborescence possède un point de départ qu'on appelle la racine, comme pour un arbre dans la nature, sauf qu'il n'y a pas "des" racines mais "une" racine.

On peut le considérer comme un dossier qui contient tout le reste.

Chemins

Pour qu'un objet (fichier ou dossier) puisse être accédé, il doit apparaître quelque part dans l'arborescence.

Un chemin est une façon de décrire où se trouve un objet (fichier ou dossier) dans l'arborescence.

Un chemin est une succession de noms de dossiers séparés par des slashes (**/**).

Après le dernier slash, et uniquement à cet endroit, on peut avoir un nom de fichier. Dans ce cas, on dit que le chemin mène vers un fichier. Sinon il mène obligatoirement vers un dossier.

Il existe deux types de chemins : les chemins absolus et les chemin relatifs

Un **chemin absolu** est un chemin qui part de la racine (**/**). Il est dit "absolu" car il décrit la façon d'y accéder, peu importe où on se trouve actuellement.

Attention, l'usage d'un tilde (**~**) qui est un nom raccourci pour dire "mon répertoire home", est aussi un chemin absolu car il se traduit toujours par un chemin absolu vers votre dossier home.

Un **chemin relatif** est un chemin qui ne part pas de la racine, c'est-à-dire qui ne commence pas par un slash. Il est relatif à l'endroit où on se trouve actuellement, au

moment où on fait usage du chemin. Ainsi, un chemin relatif pour aboutir à des endroits différents, localiser des objets différents, en fonction de notre position courante dans l'arborescence.

`.` et `..`

Deux noms de dossiers ont une signification spéciale :

`.` signifie : le dossier courant, celui où je me trouve actuellement.

`..` signifie : le dossier parent de celui où je me trouve actuellement.

C'est un nommage pratique car il permet de nommer le dossier courant et son parent, sans avoir ni à connaître leur nom, ni leur position absolue dans l'arborescence.

Et comme tout dossier à toujours un parent (y compris **`/`**), on peut remonter l'arborescence en cumulant les **`..`** comme par exemple : **`../../..`** pour dire le parent du parent du parent du dossier où je me trouve.

Dossier courant

On a utilisé plusieurs fois l'expression "le dossier où je me trouve", mais qu'est-ce que ça signifie précisément ?

Ceci est évidemment compréhensible dans un Terminal. On est toujours dans un dossier et d'ailleurs il est souvent affiché, au moins partiellement, dans le prompt.

Mais qu'en est-il d'un programme qui s'exécute ? Notamment des programmes graphiques, VSC ou Firefox par exemple. On les exécute souvent depuis l'interface graphique et rien ne nous indique clairement où ils se trouvent au démarrage.

Pourtant, tout processus qui s'exécute est obligatoirement, dès son lancement, "dans" un répertoire qu'on va nommer son "répertoire de travail".

Ainsi, dans un code en C que vous avez écrit et compilé, vous pouvez faire un **`fopen()`** pour ouvrir un fichier **`../docs/liste.txt`**, ce qui est bien un chemin relatif vers un fichier. Ce chemin est relatif par rapport à son répertoire de travail. Évidemment, ça fonctionne aussi avec des chemins absolus.

Actions sur le File System

Commandes

Vous savez déjà faire des actions et manipulé le FS, en ligne de commande, dans un Terminal :

- **`cp`** : copier un fichier sous un autre nom et/ou vers un autre endroit. Avec certaines options (**`-a`**, **`-R`**), on peut aussi copier un dossier complet, avec son contenu, c'est-à-dire récursivement.

- **mv** : renommer un objet si la destination reste la même, ou le déplacer et éventuellement le renommer en même temps si la destination est différente.
- **rm** : supprimer un fichier, ou un dossier complet (avec l'option **-r**).
- **mkdir** : créer un dossier, ou une arborescence de dossiers (avec l'option **-p**)
- **rmdir** : supprimer un dossier qui doit obligatoirement être vide.
- **ls** : afficher la liste des objets, avec quelques informations supplémentaires (**-l**)

Fonctions système

Ces commandes ont leur équivalent sous forme de fonctions système, en C[1].

Certaines ont une équivalence directe (même nom, même rôle) et d'autres ont soit un autre nom, soit sont une combinaison de plusieurs fonctions système.

Par exemple **mkdir()** et **rmdir()** existent mais **mv()** et **cp()** n'existent pas.

Les fonctions système ne se limitent pas aux actions sur le FS, mais on va découvrir ce que sont les fonctions système au travers de quelques unes dont les actions touchent au FS. Dans de futurs TP, nous verrons d'autres fonctions système, non liées au FS mais à la mécanique similaire. Vous pourrez alors revenir ici en cas de besoin d'aide.

Manuel

Vous le savez, le **man**uel est votre meilleur ami pour le cours de Système d'exploitation.

Le manuel est découpé en volumes. Il y en a officiellement 8 (9 sous Linux). On les nomme par leur numéro. On dit par exemple "regarde dans le **man 3**".

Voici la liste :

1. Commandes et programmes système. Ex : **ls**, **mkdir**, **vi**
2. Fonctions système. Notre terrain de jeu de cette année
3. Fonctions de bibliothèques. Ex : **sin()**, **strlen()**, **printf()** ou **qsort()**
4. Fichiers spéciaux situés dans **/dev**
5. Formats de fichiers et conventions. Ex : **/etc/passwd**, **/etc/hosts**, **sudo.conf**
6. Jeux (si si... mais pas Fortnite ou LoL !)
7. Divers (fourre tout). Ex : regex, signaux
8. Commandes d'administration système (pour **root**). Ex : **mkfs**, **mount**

Attention, sans précision sur le numéro de volume, **man** vous donnera la 1ère page qu'il trouve en allant dans l'ordre croissant des numéros de volume.

Vous avez noté que le manuel qui va nous intéresser cette année est le n°2.

Or, vous avez aussi noté que certaines commandes ont leur équivalent en fonction système, comme par exemple la commande **mkdir** et la fonction système **mkdir()**.

Ainsi, si vous faites un **man mkdir**, vous allez atterrir sur le **man 1** de **mkdir** alors que c'est celui du **man 2** qui nous intéresse. Il faudra donc vous assurer d'être sur le bon volume de manuel, sur la "bonne page" (c'est ainsi qu'on dit).

[1] On se rappellera qu'Unix est écrit en C, les librairies système le sont donc aussi.

Pour s'assurer qu'on est sur la fonction système et non pas sur une commande, il suffit de regarder le numéro de volume qui est indiqué entre **()** en 1ère ligne. Exemple : **MKDIR(2)**

Un autre moyen d'en être sûr est que le **SYNOPSIS** d'une fonction système indique toujours les includes (**#include**) à utiliser avec la fonction en question. C'est donc un indice.

Pour cibler le bon manuel, on indique le numéro de volume en 1er paramètre. Exemple :

```
man 2 mkdir
```

Si vous ne savez pas dans quel volume se trouve ce que vous cherchez, vous pouvez éventuellement faire ceci :

```
man -k mkdir
```

qui vous affichera toutes les pages de manuel parlant de **mkdir**. Attention, on peut avoir une liste assez longue en fonction de ce qu'on cherche.

Composition d'une page du man 2

Le manuel 2, qui regroupe toutes les fonctions système, a une structure de page particulière qu'il faut connaître afin de savoir où chercher l'information utile.

Attardons-nous sur les sections les plus importantes :

- **SYNOPSIS** : donne les éléments clés pour utiliser la fonction dans du code en C :
  - Le ou les includes (**#include**). N'utilisez pas la fonction sans placer ces includes en tête de fichier C.
  - Le prototype de la fonction : les paramètres et leur type ainsi que ce qu'elle retourne. Une fonction système retourne toujours quelque chose. On en parle juste en dessous.
- **DESCRIPTION** : une partie de texte détaillant ce que fait la fonction.
- **RETURN VALUE** : ce que retourne la fonction. Une fonction système retourne presque toujours un entier qui vaut **-1** en cas d'erreur et une valeur **≥ 0** sinon. On reparle de ça un peu plus loin (voir le paragraphe sur **errno**)
- **ERRORS** : les différents codes d'erreur possibles (voir le paragraphe sur **errno**)

Il y a d'autres sections mais les plus importantes pour coder seront celles-là.

Vous voilà bien armé maintenant pour trouver de l'aide dans le manuel…

… et pas dans GOOGLE

ni STACKOVERFLOW ou ChatGPT !!!!! 😡

Expérimentations

Exercice sur la création d'un dossier

Pour créer un dossier en ligne de commande, on utilise **`mkdir`**.

La fonction système (en C) s'appelle aussi **`mkdir()`**. Notez bien la présence des parenthèses pour distinguer la commande et la fonction.

Faites un **`man`** de la fonction **`mkdir()`**. Même s'il s'agit d'une fonction, on ne précise pas les **`()`** pour chercher dans **`man`**. Attention à ce qu'on vient de dire à propos du manuel, vérifiez que vous affichez la bonne page, celle de la fonction **`mkdir()`** !

> Vous avez peut-être un exemple de code qui vous est donné dans le manuel. Faire du copier-coller est très tentant mais SVP réfléchissez bien à ce que fait ce code et ne faites pas du copier-coller bêtement sans rien comprendre.

Créez un programme **`mk_dir.c`** permettant de créer un dossier nommé **`doss_sys2A`**.

Pour renseigner le second argument de la fonction **`mkdir()`**, qui s'appelle **`mode`**, et qui correspond aux droits sur le dossier, il existe des constantes au lieu d'utiliser des valeurs numériques, mais on verra cela plus loin dans ce TP.

Pour le moment vous avez le droit d'utiliser la notation octale[2] : **`0NNN`** (exemples : **`0777`** ou encore **`0522`**).

Pour cet exercice, vous allez fixer ces droits à **`RWX`** pour **`User`** et **`Group`** et rien pour **`Other`**.

Compilez et testez. Dans un Terminal, n'oubliez pas de vérifier la justesse des droits du dossier créé. Qu'en est-il ? Est-ce conforme à ce que vous attendiez[3] ?

Pour comprendre ce qui s'est passé, dans votre Terminal tapez ceci :

```
umask
```

Voyez-vous une relation entre ce qui s'affiche et l'absence de certains droits que vous deviez vous attendre à trouver sur votre dossier ?

`umask` est un "masqueur" de droits, il permet de retirer automatiquement des droits lors de la création de fichiers ou de dossiers. C'est fait dans un objectif sécuritaire. Vous pouvez dans un second temps re-forcer les droits par un **`chmod`** (commande) ou **`chmod()`** (fonction système). **`umask`** est une sorte de variable d'environnement.

[2] Important : n'oubliez pas de mettre le préfixe **`0`**, sinon ce sera du décimal et pas de l'octal !
[3] A priori ça ne devrait pas l'être.

Cette valeur octale (sans doute **0022** chez vous) représente un tableau de bits et les droits réellement placés sont ceux que vous spécifiez desquels on ôte les bits présents dans le masque. L'opération binaire qui est faite est ainsi :

droits_réels = droits_fixés & !masque

Par exemple, pour un **mkdir (..., 0770)** les droits demandés sont **770**. Mais les droits réellement fixés à la création seront :

770 & !022 → 770 & 755 → 750

Soit **RWX** pour **User**, **RX** pour **Group** et rien pour **Other**. Notez que dans **0022**, le 1er **0** ne fait pas partie du tableau de bits. Il est simplement là pour indiquer que c'est de l'octal.

Si vous ne comprenez pas le fonctionnement de cet **umask**, interrogez votre enseignant.

Vous pouvez le modifier ponctuellement (jusqu'au logout) ainsi, par exemple :

```
umask 0000
```

retire tout le masque et donc tout "effacement" automatique des droits en création d'un fichier ou d'un dossier, tandis qu'un :

```
umask 0077
```

"efface" automatiquement tous les droits fixés pour **Group** et **Other**.

Au lieu de modifier **umask**, vous pouvez aussi (c'est mieux) passer un appel à la fonction **chmod()** pour forcer les droits définitifs que vous souhaitez, à la suite de votre appel à **mkdir()**. Modifiez votre code **mk_dir.c** pour fixer les droits demandés initialement dans le sujet.

errno

On a évoqué un peu plus tôt que presque toutes les fonctions systèmes retournent un entier, et que celui-ci vaut **-1** en cas d'erreur.

Avec ça, on n'est pas très avancé pour savoir ce qui s'est passé quand il y a un problème, et c'est là qu'entre en jeu **errno**, qui est une variable globale entière, définie par l'OS pour chaque processus, et qui contient le code de la dernière erreur rencontrée lors d'un appel à une fonction système.

Donc, si un appel d'une fonction système retourne **-1**, alors **errno** nous donnera le code de l'erreur.

Attention, **errno** sera écrasée par une nouvelle valeur au prochain appel d'une fonction système quelle qu'elle soit et quel que soit le résultat (succès ou échec). Donc, si besoin, sauvegardez la valeur de **errno** dans une autre variable si vous comptez faire un usage ultérieur de sa valeur.

Vous n'avez pas besoin de traiter tous les codes d'erreur. Vous pouvez décider de faire une action spéciale sur une erreur précise et avoir un traitement commun pour toutes les autres erreurs. Dans ce cas, il existe la fonction **perror()** qui, en s'appuyant sur la valeur de **errno**, permet d'afficher à votre place un message d'erreur formaté par le système. C'est très pratique ! Il est temps de faire un **man perror** si ce n'est pas déjà fait…

Ainsi, voici une proposition de la structure que doit avoir votre code pour un appel à une fonction système. Prenons **rmdir()** comme exemple, mais tous les appels système doivent être traités de la même façon, en fonction des erreurs que vous souhaitez considérer et avec un appel à **perror()** pour toutes les autres :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>   // voir SYNOPSIS dans man 2 rmdir
#include <errno.h>    // voir SYNOPSIS dans man 3 errno

int main() {
     int ret;

     ret = rmdir("D1");
     if (ret == -1) {  // Erreur
          if (errno == ENOENT) {  // On traite manuellement celle-ci
               printf("D1 n'existe pas !\n");
          } else {     // On confie les autres à perror()
               perror("Erreur inconnue");   // Mon msg + celui d'errno
          }
          return EXIT_FAILURE;   // 1 seul return serait mieux ! :-)
     } else {  // C'est forcément 0, signifiant que tout est OK
          printf("D1 supprimé\n");
     }
     return EXIT_SUCCESS;
}
```

Testez ce code dans les conditions suivantes et observez le message qui s'affiche :

1) Le dossier **D1** n'existe pas
2) Le dossier **D1** existe et est vide
3) Le dossier **D1** existe mais n'est pas vide
4) **D1** existe sous forme d'un fichier

Reprenez maintenant votre code de l'exercice 1 et traitez, de la même façon, les erreurs possibles.

Consultez le **man** de **mkdir()** pour identifier le code d'erreur concerné (retournez voir le paragraphe **Composition d'une page du man 2**). Testez en lançant deux fois votre

programme afin qu'il provoque une erreur la seconde fois puisque le dossier aura été créé lors du 1er lancement.

L'i-node ou inode

Chaque objet du FS (fichiers et dossiers) est attaché à une structure d'information qu'on appelle un **i-node** (ou plus rarement **i-nœud** en français).

Note : dans ce qui suit, pour simplifier les explications, on va parler uniquement de fichiers mais c'est exactement la même chose pour les dossiers. Car on sait depuis la 1ère année que sous Unix un dossier est comme un fichier dont le contenu est juste le catalogue, la liste des objets qu'il contient.

Un inode possède un identifiant numérique unique sur son espace de stockage. Pour faire simple, on va dire que l'espace de stockage est un disque dur ou une clé USB par exemple.

Alors que l'utilisateur connaît un fichier par son nom et sa position dans l'arborescence, l'OS connaît ce fichier uniquement par son numéro d'inode. En fait l'OS se moque bien de sa localisation dans l'arborescence, c'est d'ailleurs un détail qui a son importance, comme on va le voir un peu plus loin.

En ligne de commande, on peut afficher les inodes avec l'option **`-i`** de **`ls`**. Testez, c'est le nombre dans la 1ère colonne.

Contenu d'un inode

Un inode contient des métadonnées, c'est-à-dire des informations à propos du fichier. Ce ne sont pas les données du fichier lui-même.

Les principales métadonnées sont :

- Nature de l'objet : fichier standard, dossier, fichier spécial (on en verra plus tard, dans d'autres TP)
- Droits : **`RWX`** sur **`UGO`**
- Propriétaire : **`UID`**
- Groupe propriétaire : **`GID`**
- Date de création : secondes écoulées depuis l'Epoch
- Date de dernier accès : en lecture ou en écriture
- Date de dernière modification des données
- Date de dernière modification de l'inode : changement des droits par exemple
- Taille : pour les fichiers uniquement, en octets
- Nombre de liens : voir plus loin "le chien et les laisses"
- Localisation des données sur le support physique (le disque dur)

Notez que nulle part on n'a parlé d'un nom de fichier.

Un inode **ne contient pas** de nom de fichier. On peut donc se poser la question : où sont donc indiqués les noms des objets du FS ?

Dossiers : retour en 1ère année

On a vu en 1ère année, et on l'a rappelé il y a quelques instants en introduisant les inodes : un dossier est une sorte de fichier spécial listant son contenu.

On l'avait évoqué ainsi pour expliquer la façon dont sont traités les droits sur les dossiers. Par exemple, pour supprimer un fichier il suffit d'avoir les droits d'écriture sur le dossier, c'est-à-dire sur le listing du dossier puisqu'on le considère ainsi. Peu importe les droits qu'on possède sur le fichier, si on a le droit de modifier le listing (en supprimant une ligne), on a donc le droit de supprimer le fichier.

On s'était arrêtés là en 1ère année car ça suffisait à nos besoins de l'époque de voir le dossier sous cette forme de fichier-listing.

Maintenant que nous sommes en 2ème année, nous allons donc reprendre cette vision des choses car elle va nous permettre de comprendre la relation entre les inodes et les noms des objets.

En effet, le listing du dossier contient non seulement des noms d'objets mais chaque nom est associé à un inode ! Ceci permet donc à l'OS de trouver la localisation physique des données d'un fichier grâce à l'inode dont il va trouver la relation avec un nom qu'on lui indique. Relisez cette phrase et/ou faites-vous expliquer sa signification par l'enseignant.e si besoin. C'est très important de bien comprendre cette relation.

Le chien et les laisses

Maintenant qu'on a éclairci comment est entretenue la relation **`nom_objet` → `inode`**, on va faire une analogie avec, comme le titre du paragraphe l'indique, un chien !

Un objet est attaché à un inode.

Un objet se trouve toujours dans un dossier, ça on le sait depuis la 1ère année.

Mais, ce que vous savez sans doute moins, c'est qu'un objet peut être dans plusieurs dossiers en même temps !

La relation **`nom_objet` → `inode`**, c'est comme un chien et une laisse. Un chien peut être rattaché à plusieurs laisses et tant qu'il existe une laisse attachée au chien, il ne peut pas se libérer. Tant qu'il existe, dans un quelconque dossier, une relation **`nom_objet` → `inode`**, l'inode n'est pas libéré et l'objet existe. Mais, dès que la dernière laisse est lâchée, le chien disparaît. Dès qu'il n'existe plus aucun dossier ayant l'inode dans sa liste, l'inode est libéré, le fichier et ses données disparaissent.

Fonctionnement de **`rm`** (ou **`rmdir`**)

Quand on supprime un objet, on modifie le catalogue du dossier qui contient l'objet en supprimant la ligne où il apparaît.

Cependant, comment fait l'OS pour savoir si l'inode associé à ce nom (qu'on est en train de supprimer) n'existe pas encore dans un ou plusieurs autres dossiers ? Voire dans le même dossier mais sous un autre nom, ce qui est tout à fait autorisé.

Retournez voir le paragraphe **Contenu d'un inode**. Y voyez-vous un indice ?

Effectivement, vous avez dû identifier qu'il y a une information concernant le "nombre de liens" vers l'inode. Si vous n'avez pas trouvé cet indice (ou que vous n'avez pas eu le courage d'aller chercher), maintenant que vous avez la réponse vous pouvez retourner vérifier dans le paragraphe **Contenu d'un inode**

Chaque fois que l'inode est attaché à un nom dans un dossier, ce nombre augmente de 1. Chaque fois qu'on libère l'inode, ce nombre diminue de 1. Quand il arrive à 0, donc quand le chien n'a plus de laisse attachée à lui, l'inode est libéré par l'OS ainsi que l'espace sur disque occupé par les données de fichier. Le fichier disparaît.

Fonctions bas niveau d'accès aux fichiers

Avant de parler de fonctions bas niveau, parlons des fonctions de plus haut niveau. Qu'entend-on par là ?

Fonctions haut niveau

Depuis votre passage en 1ère année, vous connaissez déjà les fonctions haut niveau d'accès aux fichiers :

- **`fopen()`** : ouverture d'un fichier. Toutes les autres fonctions nécessitent l'ouverture préalable du fichier pour fonctionner.
- **`fclose()`** : fermeture d'un fichier (hé oui, ça existe, même si vous n'en faites pas usage, ce qui n'est pas bien car tout fichier ouvert doit être refermé !)
- **`fread()`**, **`fgets()`**, **`fgetc()`**, **`fscanf()`** : lecture dans un fichier
- **`fwrite()`**, **`fputs()`**, **`fputc()`**, **`fprintf()`** : écriture dans un fichier
- **`fseek()`**, **`ftell()`**, etc. : il y a plein d'autres fonctions que vous n'avez peut-être jamais utilisées...

On les qualifie de *haut niveau* car elle apportent un certain confort d'utilisation :

- Gestion d'un buffer (sorte de cache) d'entrée/sortie
- **`fscanf()`** ou **`fprintf()`** permettent notamment des lectures et écritures suivant un format assez souple.

Ces fonctions ne sont pas des fonctions système.

Vous aurez noté (ou pas) qu'elles portent toutes un nom commençant par un **`f`**, comme **`file`**.

Ces fonctions sont un enrobage des fonctions système qu'on nomme *fonctions bas niveau*. Elles font appel aux fonctions système qu'on va voir maintenant.

Fonctions bas niveau

Les fonctions système (bas niveau) sont beaucoup plus rudimentaires :

- Pas de gestion d'un buffer d'entrée/sortie : ce que vous écrivez dans un fichier avec une fonction système d'écriture (qu'on va voir juste après), sera écrit immédiatement dans le fichier sur disque[4].
- Une seule fonction de lecture et une seule fonction d'écriture, toutes deux très basiques.
- Un fonctionnement sur les fichiers physiques (sur disque ou sur des volumes disque réseau) mais aussi sur un ensemble d'autres mécanismes Unix non "fichiers" mais traités de façon similaire : connexions réseaux (sockets), tubes (pipes), queues de messages (IPC). On en verra certains cette année.

ATTENTION : Sauf si on vous indique le contraire, à partir de maintenant, dans ce module R3.05, vous n'utiliserez **que des fonctions système**, donc des **fonctions bas niveau**.

Voici la liste des fonctions système que vous devez connaître et utiliser. A vous d'aller dans le **man** pour trouver du détail sur leur utilisation ! Attention il faut consulter le **man 2** !

Open(...) : ouverture d'un fichier

Tout comme pour les fonctions haut niveau, les fonctions système sur les fichiers nécessitent souvent l'ouverture du fichier dans un mode particulier, en fonction de ce qu'on souhaite faire sur le fichier (lire ou écrire).

Pour cette 1ère fonction, on va voir ensemble ce que dit le **man 2 open**, ensuite débrouillez-vous pour les autres fonctions.

"Voir ensemble" ne vous empêche pas d'aller voir et suivre les indications dans le **man**.

```
man 2 open
```

affiche :

```
SYNOPSIS
     #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);
... partie coupée ...
```

[4] Physiquement ce n'est pas tout à fait juste car l'OS met en place un mécanisme de cache pour optimiser les performances, mais c'est totalement neutre d'un point de vue technique. Donc on considérera que c'est effectivement vrai.

```
DESCRIPTION
     O_APPEND...
     ...
     O_CREAT...
     ...
RETURN VALUE
...
ERRORS
...
```

On l'a vu précédemment, le **SYNOPSIS** concerne les **#include** que vous devez utiliser dans votre code ainsi que les signatures de la fonction (parfois on y trouve une famille de fonctions). Il apporte aussi une explication sur le rôle et l'usage de la fonction.

On peut donc observer que **open()** a 2 signatures, donc qu'elle accepte 2 façons d'être appelées : avec ou sans le 3ème argument, **mode** (de type **mode_t**, défini dans **sys/types**, c'est pour cela qu'on a un **#include <sys/types.h>** avant).

Détaillons un peu :

- **const char *pathname** : chemin (absolu ou relatif) qui mène au fichier à ouvrir ou à créer, en fonction de la situation.
- **int flags** : le mode d'ouverture ou de création du fichier. C'est un tableau de bits dont on trouve les valeurs définies par des **#define** dans un des fichiers **#include**. Pas besoin d'aller chercher dans les fichiers includes, la liste est intégralement donnée, avec les explications, dans cette page de manuel. Voici quelques exemples utiles pour ce TP. Le préfixe **O_** est là pour signifier "**O**pen mode" :
  - **O_RDONLY** : ouverture pour uniquement de la lecture dans le fichier
  - **O_WRONLY** : ouverture pour uniquement de l'écriture dans le fichier
  - **O_RDWR** : ouverture pour de la lecture et/ou de l'écriture
  - **O_CREAT** (sans "E") : créera le fichier s'il n'existe pas. Sans ce flag, pas de création, le fichier doit exister sinon on aura une erreur.

  Il faut obligatoirement un et un seul flag parmi : **O_RDONLY**, **O_WRONLY** ou **O_RDWR**. Les autres **O_XXX** sont des options.

  Comme il s'agit d'un "tableau de bits", les constantes **O_XXX** doivent être combinées à l'aide de **|** (pipe = ou binaire). **Attention** : ni des **+** ni des **||** (ou logiques). Exemple : **O_WRONLY | O_CREAT**
- **mode_t mode** : droits à affecter au fichier (uniquement en création d'un nouveau fichier). Ces droits sont aussi spécifiés par un tableau de bits (**RWX** sur **UGO**) dont les valeurs sont définies par des **#define** dans un des fichiers **#include**. Pas besoin d'aller chercher dans les fichiers includes, la liste est intégralement donnée, avec les explications, dans cette page de manuel. Le préfixe **S_I** est là pour signifier "**S**et **I**node rights". Voici quelques exemples :
  - **S_IRUSR** : fixer **R** pour **USeR**

- **`S_IWUSR`** : fixer **`W`** pour **`USeR`**
- **`S_IXUSR`** : fixer **`X`** pour **`USeR`**
- **`S_IRWXU`** : fixer **`RWX`** pour **`User`** (c'est un raccourci pour la combinaison de **`S_IRUSR + S_IWUSR + S_IXUSR`**)

Consultez le **`man`** pour les autres valeurs (celles qui concernent **`G`** et **`O`**).

Comme il s'agit d'un "tableau de bits", les constantes **`S_IXXX`** doivent aussi être combinée par des **`|`**.

- **`RETURN VALUE`** ne doit pas être une surprise pour vous depuis les pages 5 et 6 de ce TP. Retournez en arrière si besoin…
- Idem pour **`ERRORS`** qui donne la liste exhaustive des erreurs que cette fonction **`open()`** peut produire (via **`errno`**).

ATTENTION :

On a vu en 1ère année que **`fopen()`** renvoie une structure **`FILE *`**.

Mais **`fopen()`** est une fonction haut niveau.

Comme l'immense majorité des fonctions système, **`open()`** renvoie simplement un entier. Cet entier est (voir le **`man`**) soit un identifiant permettant de travailler (lire/écrire) avec le fichier ouvert, soit une erreur (**`-1`**) complétée par une valeur de **`errno`** décrivant cette erreur.

Toutes les fonctions qui suivent utilisent donc cet identifiant entier renvoyé par **`open()`** pour fonctionner. On appelle cet identifiant un *handler* [5].

`close(...)` : fermeture d'un fichier ouvert ou créé

Comme toujours, refermez vos fichiers quand vous n'en avez plus besoin, y compris avant de quitter le programme. Ne laissez pas ce rôle au système. On éteint la lumière et on ferme la porte en partant, même si on sait que maman ou papa passera derrière…

`read(...)` : lecture dans un fichier ouvert

Notes utiles :

- **`ssize_t`** est un **`#define`** qui est défini, dans l'un des **`#include`**, comme étant un **`int`**. C'est donc finalement un entier maquillé sous un autre nom de type.
- **`buf`** est un pointeur **`void`**, c'est-à-dire que c'est un pointeur de type indéfini. C'est donc simplement une adresse en mémoire. En fonction de vos besoins, vous choisirez vous-même la nature de ce qu'il pointe : une chaîne, une structure, un tableau, peu importe car vous passerez ici l'adresse d'un "truc" que vous aurez alloué (par un **`malloc()`** ou par la déclaration d'une variable locale ou globale).

[5] En anglais *a handle* signifie *une poignée*. Ça sert donc à manipuler (*to handle*).

Vous devez donc passer l'adresse de quelque chose, ce n'est pas **`read()`** qui allouera de la mémoire pour vous !

- **`count`** est la taille du "truc" alloué. La lecture dans le fichier ne dépassera pas cette taille et sera exactement de cette taille s'il y a suffisamment de données dans la source (le fichier). Pour une structure ça peut être un **`sizeof()`** de la structure. Pour un **`malloc()`** ce sera la taille de l'allocation mais en aucun cas un **`sizeof()`** du "truc" retourné par **`malloc()`**[6] !
- La valeur retournée est soit la taille de ce qui a été réellement lu, donc au mieux **`count`**, voire moins, voire même **`0`** s'il n'y avait rien à lire, ou encore **`-1`** en cas d'erreur et **`errno`** sera là pour compléter l'info…

  J'arrête ici de parler de `errno`, à vous de savoir qu'il est là maintenant

`write(...)` : écriture dans un fichier ouvert

Les arguments de la fonction **`write()`** sont similaires à ceux de **`read()`**.

`write()` retourne aussi la taille de ce qui a été écrit (ou **`-1`** si erreur, bla bla bla).

`stat(...)` : récupération des informations d'un inode

Cette fonction ne nécessite pas d'ouvrir le fichier, mais il existe une version de cette fonction, nommée **`fstat()`** qui opère sur un fichier ouvert.

Vous allez en avoir besoin dans un prochain exercice.

> Ne confondez pas les fonctions système avec les fonctions haut niveau.
>
> Généralement[7], les noms des fonctions système **ne commencent pas** par un **`f`** !

Exercices

Dans les exercices suivants, toutes les erreurs doivent afficher un message explicite à l'écran.

Exercice 1 - cp

A l'aide des fonctions système **`open()`**, **`close()`**, **`read()`** et **`write()`**, écrivez un programme **`exo1.c`** qui copie un fichier **`ORIG`** en un fichier **`OLD`**.

[6] Si ceci vous trouble, demandez des explications, c'est important !
[7] Toute règle à ses exceptions et **`fstat()`** en est un parfait exemple.

Vous créerez le fichier **ORIG** à la main, avec VSC par exemple ou par toute autre technique à votre convenance. Préférez un contenu texte, qui sera plus facile à vérifier lors de vos tests.

Vous devez respecter ces instructions :

- Le fichier **OLD** est créé s'il n'existe pas
- Le fichier **OLD** est écrasé s'il existe déjà. Vous vérifierez qu'il a bien été écrasé grâce à la date de dernière modification avec un **ls -l**.
- Pour tester un cas d'erreur, créez un dossier **OLD** pour qu'un fichier de ce nom ne puisse plus être créé au même endroit.

Vérifiez enfin que **OLD** est strictement identique à **ORIG** grâce à la commande :

```
diff -sq ORIG OLD
```

Exercice 2 - inode

A l'aide de la fonction système **stat()**, écrivez un programme **exo2.c** qui affiche les informations suivantes concernant le fichier **ORIG** (celui de l'exercice précédent, à recréer si vous l'avez supprimé) :

- Numéro d'inode
- Droits sur le fichier (au format octal[8])
- User ID (**UID**) du propriétaire
- Groupe ID (**GID**) du groupe propriétaire
- Taille du fichier (en octets)

Vous vérifierez votre affichage en le comparant à ce qu'un **ls -lin ORIG** vous affiche[9].

Exercice 3 - rm

A l'aide de la fonction système **unlink()**, écrivez un programme **exo3.c** qui supprime le fichier **ORIG**.

Exercice 4 - Parcours du contenu d'un dossier

Il n'existe pas réellement de fonctions système pour lire le contenu (catalogue) d'un dossier. Ca ne signifie pas que c'est impossible mais ça nécessite l'utilisation d'une fonction qui s'appelle **syscall()** est qui est hors cadre à notre niveau.

[8] **man 3 printf**
[9] Un petit **man ls** peut-être ?

Cependant, comme il n'existe pas de fonctions système dédiées, la libc **POSIX** a prévu quelque chose pour nous : des fonctions qui encapsulent les appels **syscall()** pour nous simplifier la tâche et ainsi contourner l'absence de fonctions dédiées.

De ce fait, vous utiliserez un **man 3** au lieu d'un **man 2** (fonctions système).

A l'aide des fonctions **opendir()**, **readdir()** et **closedir()**, écrivez un programme **exo4.c** qui affiche le catalogue du dossier courant[10].

Vous afficherez seulement les noms des objets visibles, comme avec une simple commande **ls** (sans option). Vous afficherez aussi, en dernière ligne, le nombre d'objets listés.

Exercice 5 - Parcours détaillé d'un dossier

Reprenez votre exercice précédent et créez un programme **exo5.c** qui affiche des informations détaillées du dossier courant :

- Nom de l'objet
- Type de l'objet (on se limitera à fichier standard ou dossier). Ici vous pourrez utiliser les constantes **DT_REG** et **DT_DIR** qui sont évoquées dans le **man readdir**

Exercice 6 - Complément exercice 2

Dupliquez votre **exo2.c** en **exo6.c** et complétez-le pour afficher aussi les dates de :

- Dernier accès (exprimée en valeur brute en secondes)
- Dernière modification (exprimée en valeur brute en secondes)

ainsi que le nombre de **hard links**. Un **hard link** correspond au comptage du nombre de liens vers cet inode (voir paragraphe **Le chien et les laisses** à propos des inodes). D'ailleurs, la fonction système **unlink()** (exercice 3) porte bien son nom !

Exercice 7 - rmdir

La commande **rmdir** ne permet pas de supprimer un dossier non vide. La fonction **rmdir()** ne fait pas mieux.

Cependant, la commande **rm -r** permet de le faire, en supprimant le contenu d'un dossier récursivement. Nous avons vu que la fonction système qui correspond à la commande **rm** est **unlink()**.

Vérifiez dans **man** si cette fonction **unlink()** dispose d'une option pour supprimer récursivement un dossier et son contenu.

[10] Astuce : Comment nomme-t-on le dossier courant de façon générique ?

Proposez un code **exo7.c** permettant de combler ce manque et de retrouver la même possibilité qu'une commande **rm -r**. Par sécurité, durant la phase de test, il est conseillé de ne pas réellement supprimer (**unlink()**) quoi que ce soit et de privilégier plutôt un simple affichage "je supprime xxx".

Vous testerez votre code sur un dossier **bidon/** nommé en dur dans votre code, et qu'il vous faudra créer et remplir manuellement en amont.

Tip : **chdir()** est une fonction système qui peut vous être utile. On peut aussi s'en passer.

Exercice 8 - argc et argv

Pour terminer, arrêtons-nous sur un détail qui a été occulté depuis que vous avez écrit votre premier **main()** en 1ère année.

Jusqu'à présent, voici ce que vous avez dû (apprendre à) écrire dans vos codes :

```
int main() {
...
}
```

Avant d'entrer en détail, rappelons-nous que les commandes qu'on exécute dans un Terminal sont, par exemple, de la forme :

```
nom_commande param1 param2 param3
```

Vous êtes-vous déjà demandé comment une commande, qui est finalement un programme écrit en C, reçoit les paramètres passés derrière son nom, depuis le Shell ?

Vous commencez à comprendre où on va en arriver ?!

Et là vous vous dites : ce sont des paramètres passés au **main()**, et vous avez raison, sauf que vous en avez certainement une idée trop hâtive.

Vous pensez peut-être que dans l'exemple précédent, le main serait de la forme :

```
int main(char *param1, char *param2, char *param3) {
...
}
```

C'est une idée intéressante mais elle pose un problème de taille. Comment gère-t-on ainsi des listes de paramètres de longueur variable ? Par exemple avec un **ls -l *.c**, la

liste de fichiers qui résulte du développement de ***.c** est très variable et peut même être très longue !

Ca n'est donc pas la solution mais on est sur la voie.

En fait, il faut savoir que **main()** accepte 3 arguments, qu'on a omis jusqu'à présent, simplement parce qu'on en n'avait pas l'usage, mais qui sont pourtant très utiles !

Voici la vraie syntaxe d'un **main()** :

```
int main(int argc, char *argv[], char *envp[]) {
...
}
```

qu'on peut aussi trouver écrite ainsi :

```
int main(int argc, char **argv, char **envp) {
...
}
```

On va privilégier la 1ère forme qui est peut-être plus explicite.

Voici donc la signification de chacun des 3 arguments du **main()** :

- **argc** (entier) : nombre de chaînes de caractères dans le tableau **argv[]** ci-après.
- **argv[]** (tableau de chaînes de caractères) : les arguments passés au lancement du programme, généralement en ligne de commande. Ce sont les chaînes **"param1"**, **"param2"** et **"param3"** de notre exemple précédent, mais comme il s'agit d'un tableau, il n'y a pas de limite. C'est l'OS qui alloue autant de place (**argc** cellules) pour ce tableau qu'il y a d'arguments passés en ligne de commande au lancement du programme.
  Même si on passe des arguments numériques, ils sont toujours passés sous la forme de chaînes de caractères, à vous de les convertir[11] si besoin.
- **envp[]** (tableau de chaînes de caractères) : les variables d'environnement du programme. Vous pouvez en avoir un aperçu en tapant la commande **env** dans un Terminal.
  Ce tableau est aussi alloué par l'OS et contient un nombre de cellules (une chaîne de caractère par variable) de taille indéfinie. Par contre, ici il n'y a pas de variable comme **argc** pour savoir la taille du tableau **envp[]**. Ce tableau se termine quand un pointeur **NULL** est détecté comme valeur d'une cellule.

Écrivez un programme exo8.c qui affiche :

- Le nombre d'arguments passés à l'appel du programme en ligne de commande. **IMPORTANT** : vous devrez tester votre programme en ligne de commande dans un Terminal, pas depuis VSC !

[11] **man 3 atoi** ou **atof** par exemple.

- La liste des arguments
- La liste des variables d'environnement

Quelques essais :

```
./exo8
./exo8 a b c
./exo8 9 80 toto 700
```

Combien vaut **argc** au minimum ?

Qu'observez-vous pour **argv[0]** ?

Reprenez le code de votre 1er exercice (**mk_dir.c**) et faites-en un **new_mkdir.c** qui crée un dossier dont le nom est passé en argument du programme, en ligne de commande.

Exercice 9 - ls -l

Écrivez un programme **exo9.c** qui reproduit le fonctionnement d'un **ls -l** acceptant une liste de fichier en paramètres, exemple :

```
./exo9 *.c
```

Notez que c'est le Shell qui s'occupe de développer le ***.c** en autant de fichiers qu'il y en a dans votre répertoire courant qui correspondent à ce motif. Vous recevrez donc une liste exhaustive de noms de fichiers et pas la chaîne ***.c**.[12]. Ce n'est pas à vous de faire ce travail, heureusement et merci le Shell !!!

Observez l'affichage d'un ls -l pour vous adapter. Voici quelques pistes utiles :

- **getpwuid()** permet de retrouver un nom d'utilisateur depuis son **UID**
- **getgrgid()** permet de retrouver un nom de groupe depuis son **GID**
- **strftime()**[13] permet de formater une date+heure

[12] Sauf si aucun fichier ne correspond au motif.
[13] Ce n'est pas une fonction système mais une fonction utilitaire de la **libc**.