Programmation Système
Travaux Pratiques (2), Licence 2 Informatique
Gestion de fichiers

Les exercices suivants ont pour but de vous familiariser avec les commandes de base de manipulation des répertoires et fichiers.
Il vous est recommandé de consulter les pages man pour de plus amples informations sur leur syntaxe, leur sémantique et les éventuelles options qu'elles offrent.

Les instructions des exercices se repèrent par des icônes, qui sont les suivantes :

	Information	Information concernant l'usage ou le rôle d'une commande, par exemple. Dans certains cas, il s'agit d'une information sur ce que vous êtes en train de faire ou sur ce qui se passe.
	Exemple	Exemple d'utilisation.
	Contrôle	Vérifier le résultat d'une (ou plusieurs) action(s).
	Action	Effectuer la ou les action(s) décrite(s).
	Question	Questions auxquelles vous devez répondre.

De plus, un `texte en police courier` correspond soit à une sortie écran soit à des noms spécifiques (menus, fenêtre, icône, processus, commandes…).
Un **texte en police times gras** correspond à ce que l'utilisateur doit introduire comme valeur de paramètre, ou encore, est utilisé pour attirer l'attention de l'utilisateur.

La gestion des erreurs

Les appels système apparaissent, traditionnellement, comme des fonctions à valeur entière et l'échec d'un appel est matérialisé par une valeur de retour égale à -1. En cas d'échec d'un appel à l'une de ces fonctions, il est possible d'obtenir une information plus précise sur la nature de l'erreur rencontrée par l'intermédiaire de la variable entière externe `errno`. La valeur de la variable `errno` n'est significative qu'au retour d'un appel à une fonction ayant échoué (la variable n'est pas affectée au cours des appels réussis). La variable `errno` est définie dans le fichier `errno.h` qui contient la liste des erreurs susceptibles de provoquer l'échec d'un appel système. Dans ce même fichier, pour chacune des valeurs possibles de `errno`, on trouve la définition d'une constante symbolique et un texte en commentaire indiquant la nature de l'erreur.

Mohamed Taghelit , taghelit@univ-tours.fr

Il est possible, lorsqu'une fonction échoue (valeur de retour `-1` pour les fonctions à valeur numérique ou `NULL` pour celles renvoyant un pointeur), de visualiser le message correspondant à l'erreur rencontrée au moyen de la fonction

```
void perror (const char *p_chaine)
```

qui affiche le message associé à l'erreur, précédé de la chaîne pointée par `p_chaine` (définie par l'utilisateur) suivie du séparateur `:<space>`.

```
void main (int nb_args, char *args[])
{
…
if ( open (args[1], O_RDONLY) == -1) {
      perror("Erreur d'Ouverture");
      exit(1);
      }
…
```

Par ailleurs, deux variables `sys_nerr` et `sys_errlist` sont accessibles si elles ont été déclarées externes dans le programme, sous la forme

```
extern int sys_nerr;
extern char *sys_errlist[];
```

`sys_errlist` est un tableau de `sys_nerr` pointeurs et pour chaque entier `i`, `sys_errlist[i]` est un pointeur sur le message correspondant à l'erreur `i`.

Utiliser la gestion des erreurs dans les programmes à écrire dans la suite.

Les variables d'environnement

La forme la plus générale de la fonction principale d'un programme (en langage C) correspond au prototype suivant

```
int main (int argc, char *argv[], char **arge);
```

Le paramètre `argc` est le nombre total de paramètres. C'est à dire, le nombre de composantes de la commande `shell` correspondante.

Le paramètre `argv` est un tableau contenant les différents paramètres de la commande. Il contient `argc` pointeurs (plus un pointeur `NULL` pour marquer la fin). Le i-ème élément du tableau est le i-ème argument de la commande (le premier élément du tableau, `argv[0]`, pointe sur le nom de la commande).

Le paramètre arge est une liste de pointeurs permettant l'accès à l'environnement dans lequel le processus s'exécute. Chacun des pointeurs permet d'accéder à une chaîne de caractères de la forme

```
nom_variable=chaine_valeur
```

Ceci permet à un processus correspondant à l'exécution d'une commande tapée sous un `shell` de s'exécuter dans l'environnement défini par l'utilisateur (par exemple, les variables `TERM` ou `PATH`).

Accès à l'environnement

L'accès aux variables de l'environnement d'un processus peut se faire, dans un programme C, de différentes façons.

Accès par le paramètre `arge` de la fonction `main`

```
#include<stdio.h>
void main (int argc, char *argv[], char **arge) {
int i;
for ( i=0; arge[i] != NULL; i++)
      puts(arge[i]);
}
```

Tester l'accès aux variables de l'environnement selon la méthode précédente.

Accès par la variable externe `environ`

L'environnement est également accessible au travers de la variable externe `environ`. Le programme doit contenir la déclaration de cette variable sous la forme

```
extern char **environ;
```

```
#include<stdio.h>
void main (int argc, char *argv[], char **arge) {
extern **environ; char **env;
env = environ;          /* préserve environ */
while ( *env != NULL )
      puts( *env++);
}
```

Tester l'accès aux variables de l'environnement selon la méthode précédente.

Accès par la fonction standard `char * getenv(const char *nom_variable);`

La fonction standard `getenv()` recherche, dans la liste des variables d'environnement, une chaîne de la forme "`VAR=valeur`" et retourne un pointeur sur la chaîne "`valeur`". Si la variable n'est pas définie, la veleur retournée est le pointeur `NULL`.

```
#include<stdio.h>
#include<stdlib.h>
void main (int argc, char *argv[], char **arge) {
char *VAR;
if ( (VAR = getenv("PATH")) != NULL )
      fprintf(stdout, "Valeur PATH = %s", VAR);
else
      fprintf(stderr, "VARIABLE PATH NON DEFINIE !");
}
```

Tester l'accès aux variables de l'environnement selon la méthode précédente.

Création, ou modification de la valeur, d'une variable de l'environnement

Un processus peut modifier la valeur d'une variable de son environnement ou y ajouter une nouvelle variable par l'appel à la fonction standard

```
int putenv(const char *chaine);
```

La chaîne de caractères donnée en paramètre a la forme `VAR=valeur`. La variable de nom `VAR` est soit créée (si elle n'existe pas déjà) soit modifiée dans l'environnement du processus. La valeur de retour de la fonction est nulle si la création ou la modification de la variable a été possible et non nulle sinon.

```
#include<stdio.h>
#include<stdlib.h>
void main (int argc, char *argv[], char **arge) {
putenv("MYVAR=titi");
      fprintf(stdout, "Vefir MYVAR = %s\n", getenv("MYVAR"));
}
```

Tester la création et la modification des variables de l'environnement selon la méthode précédente.

Appels système : Gestion des fichiers

Création d'un fichier

Écrire un programme qui crée, au travers de l'appel

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

un fichier en lecture/écriture.

Si le fichier existe déjà, une erreur doit être retournée.

Quel est le code d'erreur retourné lorsque le fichier existe déjà ?

Si l'argument `mode` spécifié est `755` (`rwxr-xr-x`), est-ce que le fichier est créé avec exactement ces droits ? Expliquer !

Obtention des caractéristiques d'un fichier

Écrire un programme qui récupère les caractéristiques de fichiers donnés, au travers des appels des fonctions `int stat(); int fstat();` et `int lstat;`. Pour un fichier donné, afficher les caractéristiques suivantes :

- le numéro d'inode,
- la protection,
- le nombre de liens physiques,
- l'`ID` du propriétaire,
- l'`ID` du groupe,
- la taille du fichier,
- la taille de bloc,
- le nombre de blocs,
- l'heure du dernier accès.

L'affichage d'une heure dans un format lisible peut être accompli en utilisant la fonction `ctime()`.

Ouverture d'un fichier, lecture et écriture dans un fichier

Cet exercice a pour but d'utiliser les fonctions `int open()`, `ssize_t read()` et `ssize_t write()`.

Écrire un programme qui recopie un fichier source, `fichier_source`, dans un fichier destinataire, `fichier_destinataire`. Le programme doit vérifier que le fichier source est un fichier régulier (utiliser la macro `S_ISREG(m)`, cf. `int stat()`). Le programme doit également vérifier qu'il n'existe pas déjà de fichier de même nom que le fichier destinataire.

Quels sont les temps d'exécution (utiliser `/bin/time`) respectifs si la taille du buffer utilisé dans la fonction `read()` est de 1024 octets puis un octet ? Expliquer !

Duplication de descripteurs

Cet exercice a pour but d'utiliser la fonction `int dup()`.

Écrire un programme qui redirige la sortie d'erreur standard vers un fichier, `fichier_erreur`, préalablement créé. C'est à dire, toute écriture de la forme `write(2, …)` doit se faire dans le fichier `fichier_erreur`. Le descripteur de valeur `2` étant au départ celui de la sortie d'erreur standard.

Quelle est la propriété de la fonction `dup()` qui est exploitée pour ainsi rediriger les E/S standards ?

Positionnement de la tête de lecture/écriture dans un fichier

Cet exercice a pour but d'utiliser la fonction `off_t lseek()`.

Écrire un programme qui crée un fichier vide. Positionner la tête de lecture/écriture sur le 10 000ème octet à partir du début du fichier. Écrire un caractère à cette position.

Quelle doit être la taille du fichier ? Est-ce cette taille qui est retournée par la commande `ls -l` ? Est-ce que les blocs correspondant au trou de 9 999 caractères ont été alloués (utiliser `df`) ?

La primitive `lseek()` permet de déplacer l'`offset` courant d'un fichier et retourne le nouvel `offset`.

Écrire un programme qui, après un certain nombre de lecture/écriture sur un fichier, retourne la valeur de l'`offset` courant.

L'`offset` est associé à un fichier et non pas à un descripteur. Si deux descripteurs référencent un même fichier, la modification (par lecture/écriture ou `lseek()`) de l'`offset` du fichier via un descripteur est "visible" via l'autre descripteur.

Vérifier l'affirmation précédente.

Verrouillage des fichiers réguliers

Le verrouillage est un mécanisme général qui vise à assurer un contrôle des accès concurrents à un même fichier régulier.

Cet exercice a pour but d'utiliser la fonction `int fcntl()`.

Poser un verrou consultatif (*advisory*) partagé (*shared*) sur une zone d'un fichier.

Vérifier si la zone verrouillée peut être lue par un autre processus ? Si elle peut être écrite ?

Poser un verrou consultatif exclusif (*exclusive*) sur une zone d'un fichier.

Vérifier si la zone verrouillée peut être lue par un autre processus ? Si elle peut être écrite ?

Qu'en déduisez-vous de l'intérêt d'utiliser des verrous dans ce mode opératoire ?

Procéder aux mêmes exercices que précédemment avec un verrou impératif (*mandatory*).

L'indication du mode opératoire des verrous est mémorisée non pas dans les verrous, mais dans les inodes. Donc, les verrous sur un même fichier sont soit tous consultatifs, soit tous impératifs.

La demande de verrouillage en mode impératif de tous les verrous sur un fichier donné est réalisée en positionnant le `set-gid bit` du fichier. Le bit d'exécution sur le groupe du fichier ne l'étant pas.

Vérifier quels sont les différents types de verrous qui peuvent cohabiter, c'est à dire qui peuvent avoir des portées non disjointes.

Quel est le résultat de vos tests ?

Soient deux processus **P1** et **P2** et deux zones, **zone_1** et **zone_2**, d'un même fichier. Le processus **P1** pose un verrou sur la zone **zone_1** puis un autre verrou sur la zone **zone_2**. Le processus **P2** procède inversement (pose de verrou sur **zone_2** puis sur **zone_1**). On s'assurera que l'ordre des poses est soit [(**P1**, **zone_1**), (**P2**, **zone_2**), (**P1**, **zone_2**) et (**P2**, **zone_1**)] soit [(**P2**, **zone_2**), (**P1**, **zone_1**), (**P2**, **zone_1**) et (**P1**, **zone_2**)].

Considérer les cas de demandes de pose bloquantes et où les verrous sont, par exemple, exclusifs.

Quel devrait être le problème d'une telle exécution ?

Vérifier vos conclusions par l'exécution des deux processus.

Le résultat de l'exécution conforte-t-il vos conclusions ?

Un verrou est associé à un fichier et à un processus. Ceci a comme conséquences les points suivants :

- si le processus se termine (sans supprimer les verrous qu'il aura posés), tous ses verrous sont supprimés,
- chaque fois qu'un descripteur est fermé par un processus, tout verrou sur le fichier, référencé par ce descripteur, posé par ce processus est supprimé.

Vérifier les affirmations précédentes.

Appels système : Manipulation des répertoires

Ouverture d'un répertoire

La primitive `DIR *opendir(const char *pathname)` permet d'ouvrir en lecture le répertoire référencé par `pathname`.

La primitive `struct dirent *readdir(DIR *dp)` permet de lire l'entrée suivante du répertoire identifié par `dp`.

Écrire un programme qui ouvre un répertoire (`/tmp` par exemple) et qui affiche tous les fichiers qui y sont contenus ainsi que le type de chacun (champ `d_type` de la strucure `dirent`).

Création et suppression d'un répertoire

La primitive `int mkdir(const char *pathname, mode_t mode)` permet de créer le répertoire de nom référencé par `pathname`.

Modifier le programme précédent en y ajoutant à la fin la création (dans le répertoire `/tmp`) d'un répertoire ayant pour nom votre propre nom.

Vérifier que le répertoire a bien été créé.

Positionnement en début du pointeur de lecture dans un répertoire

La primitive `void rewinddir(DIR *dp)` permet de positionner le pointeur de lecture dans un répertoire sur la première entrée de ce répertoire.

Modifier le programme précédent en y ajoutant à la fin 1) le positionnement en début du pointeur de lecture du répertoire et 2) l'affichage à nouveau du contenu du répertoire (`/tmp`).

Avant chaque exécution, supprimer du répertoire `/tmp` le répertoire que vous avez créé.

Modifier le programme précédent en y ajoutant à la fin la suppression du répertoire créé.

Vérifier que le répertoire a bien été supprimé.