Programmation Système

Les fichiers

Emmanuel Bouthenot (emmanuel.bouthenot@u-bordeaux.fr)

Licence professionnelle ADSILLH - Université de Bordeaux

2020/2021

Les fichiers sous UNIX

Dans la philosophie d'UNIX, 'tout est fichier'.

Les entrées/sorties (input/output ou I/O), sont symbolisées par des manipulations de fichiers.

La manipulation se fait au travers de **descripteur de fichiers** et d'appels systèmes.

Descripteurs de fichiers

Définitions

- Un descripteur de fichier (**file descriptor** ou **fd**) est entier positif affecté par le noyau pour référencer un fichier utilisé par un programme en cours d'exécution

Un descripteur de fichier est unique au sein d'un même processus.

A chaque fois que le noyau ouvre ou créé un fichier pour un processus, il lui retourne le descripteur de fichier associé.

Les actions ultérieures sur le fichier requiert du processus qu'il passe ce descripteur de fichier au noyau.

Les descripteurs de fichiers standards

Descripteur	Utilisation	Nom POSIX	Nom Python
0	Entrée standard	STDIN_FILENO	sys.stdin.fileno()
1	Sortie standard	STDOUT_FILENO	sys.stdout.fileno()
2	Sortie erreur	STDERR_FILENO	sys.stderr.fileno()

Par convention, au démarrage du programme, ces descripteurs de fichiers sont déjà ouverts et prêts à être utilisé. Ils sont en fait hérités dans cet état depuis le shell qui a lancé le programme.

Même s'il est possible d'utiliser le numéro du descripteur du fichier, il reste préférable d'utiliser le nom POSIX.

L'appel système open

```c
#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);

// renvoie le nouveau descripteur de fichier en cas de
// succès, ou -1 en cas d'échec, auquel cas errno contient
// le code d'erreur
```

Appel système qui ouvre le fichier identifié par *pathname* et retourne un descripteur de fichier. En fonction de l'argument *flags*, le fichier peut être ouvert en lecture, écriture ou les 2. L'argument *mode* indique les permissions à placer sur le fichier en cas de création.

```
> man 2 open
```

L'appel système open - flags

L'argument *flags* de *open* est un masquage de bit (bit mask) qui spécifie le mode d'accès au fichier.

Flag	**Utilisation**
O_RDONLY	Lecture seule
O_WRONLY	Écriture seule
O_RDWR	Lecture et écriture
O_CREAT	Création du fichier s'il n'éxiste pas
O_TRUNC	Troncature du fichier
...	...
O_APPEND	Écriture à la fin du fichier
...	...

L'appel système open - mode

L'argument *mode* de *open* est un masquage de bit les droits à utiliser si un nouveau fichier est créé (entre autres si O_CREAT est présent dans *flags*).

Mode	**Octal**	**Utilisation**
S_IRWXU	00700	L'utilisateur (propriétaire) a le droit de lire, ecrire, exécuter
S_IRUSR	00400	L'utilisateur a le droit de lire
S_IWUSR	00200	L'utilisateur a le droit d'ecrire
S_IXUSR	00100	L'utilisateur a le droit d'exécuter
S_IRWXG	00070	Le groupe a le droit de lire, ecrire, exécuter
S_IRGRP	00040	Le groupe a le droit de lire
S_IWGRP	00020	Le groupe a le droit d'ecrire
S_IXGRP	00010	Le groupe a le droit d'exécuter
S_IRWXO	00007	Tout le monde a le droit de lire, ecrire, exécuter
S_IROTH	00004	Tout le monde a le droit de lire
S_IWOTH	00002	Tout le monde a le droit d'ecrire
S_IXOTH	00001	Tout le monde a le droit d'exécuter

L'appel système open - exemples

```
fd = open("test1.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
```

- ouvre le fichier existant ou créé le fichier *test1.txt*
- ouverture en lecture et écriture
- seul le propriétaire peut lire et écrire le fichier

```
fd = open("test2.txt", O_WRONLY | O_CREAT | O_APPEND,
                       S_IRUSR | S_IWUSR | S_IRGRP);
```

- ouvre le fichier existant ou créé le fichier *test2.txt*
- ouverture en écriture seule
- l'écriture se fera toujours à la fin du fichier
- seul le propriétaire peut lire et écrire le fichier, le groupe peut lire le fichier

L'appel système open (Python)

```
import os, stat

# Method
os.open(path, flags, mode=0o777, dir_fd=None)
# Flags
os.O_RDONLY
os.O_WRONLY
os.O_RDWR
os.O_CREAT
os.O_TRUNC
os.O_APPEND
# Modes
stat.S_IRWXU
stat.S_IRUSR
stat.S_IWUSR
```

L'appel système open - exemples (Python)

```
import os, stat
f = os.open("test1.txt",
    os.O_RDWR | os.O_CREAT | os.O_TRUNC,
    stat.S_IRUSR | stat.S_IWUSR)
f = open("test1.txt", 'w+', 0o600) # umask issue
```

- ouvre le fichier existant ou créé le fichier *test1.txt*
- ouverture en lecture et écriture
- seul le propriétaire peut lire et écrire le fichier

```
import os, stat
f = os.open("test2.txt",
    os.O_WRONLY | os.O_CREAT | os.O_APPEND,
    stat.S_IRUSR | stat.S_IWUSR | stat.S_IRGRP)
f = open("test2.txt", 'a', 0o640) # umask issue
```

- ouvre le fichier existant ou créé le fichier *test2.txt*
- ouverture en écriture seule
- l'écriture se fera toujours à la fin du fichier
- seul le propriétaire peut lire et écrire le fichier, le groupe peut lire le fichier

L'appel système read

```c
#include <unistd.h>

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

// renvoie le nombre d'octets lus ou -1 s'il échoue,
// auquel cas errno contient le code d'erreur
```

Appel système qui lit au moins *count* octets depuis le fichier référencé par le descripteur de fichier *fd* et qui les stocke dans *buffer*. S'il n'y a plus d'octets qui puissent être lu (fin de fichier) l'appel renvoie 0.

```
> man 2 read
```

L'appel système write

```c
#include <unistd.h>

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

// renvoie le nombre d'octets écrits ou -1 s'il échoue,
// auquel cas errno contient le code d'erreur
```

Appel système qui lit au plus *count* octets depuis la zone mémoire pointée par *buf* et les écrit dans dans le fichier référencé par le descripteur *fd*. Le nombre d'octets écrits peut être inférieur à *count* (ex: place disponible insuffisante).

```
> man 2 write
```

L'appel système close

```c
#include <unistd.h>

int close(int fd);

// renvoie 0 en cas de réussite ou -1 s'il échoue,
// auquel cas errno contient le code d'erreur
```

Appel système qui libère le descripteur de fichier *fd* et toutes ressources allouées par le noyau qui lui sont associées.

```
> man 2 close
```

Exemple de lecture / écriture

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

#define BUFMAX 32

int main (int argc, char **argv) {
    char buffer[BUFMAX];
    int count;

    count = read(STDIN_FILENO, buffer, BUFMAX);
    if (count == -1) {
        write(STDERR_FILENO, "Unable to read stdin\n", 21);
        exit(EXIT_FAILURE);
    }
    write(STDOUT_FILENO, "Input data was: ", 16);
    write(STDOUT_FILENO, buffer, count);

    exit(EXIT_SUCCESS);
}
```

Exemple de lecture / écriture (Python)

```python
#!/usr/bin/python3

import os, sys

def main():
    #buf = sys.stdin.read() # Ctrl-D for EOF
    try:
        buf = input()
    except EOFError:
        buf = ''
    except Exception as e:
        sys.stderr.write("Unable to read stdin: %s", e)
        sys.exit(os.EX_DATAERR)

    sys.stdout.write("Input data was: %s" % (buf))
    sys.exit(os.EX_OK)

main()
```

Déplacement dans un fichier

Pour chaque programme en cours d'exécution, le noyau maintient une table des fichiers ouverts qui contient entre autre le descripteur de fichier et la position du curseur (file offset).

Ce curseur est la position où aura lieu la prochaine lecture ou écriture. Cette position est exprimé en octet depuis le début du fichier.

A l'ouverture du fichier le curseur est positionné au début du fichier et chaque lecture ou écriture dans le fichier déplace le curseur du nombre d'octets lus ou écrits.

L'appel système lseek

```
#include <unistd.h>

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

// renvoie le nouvel emplacement (en octets depuis le début du
// fichier) en cas de réussite ou -1 en cas d'échec
// auquel cas errno contient le code d'erreur
```

Position	Utilisation
SEEK_SET	Le curseur est placé à offset octets depuis le début du fichier
SEEK_CUR	Le curseur de lecture/écriture est avancé de offset octets
SEEK_END	Le curseur est placé à la fin du fichier plus offset octets

Si *whence* vaut *SEEK_CUR* ou *SEEK_END*, *offset* peut être négatif.

```
> man 2 lseek
```

L'appel système lseek - exemples

```
/* Positionnement au début du fichier */
lseek(fd, 0, SEEK_SET);
/* Positionnement à la fin du fichier */
lseek(fd, 0, SEEK_END);
/* Positionnement avant le dernier octet du fichier */
lseek(fd, -1, SEEK_END);
/* Positionnement 10 octets avant la position actuelle du curseur */
lseek(fd, -10, SEEK_CUR);
/* Positionnement 10000 octets après la fin du fichier */
lseek(fd, 10000, SEEK_END);
```

L'appel système *lseek* modifie uniquement la table des fichiers ouverts du processus appelant. Il ne provoque aucun accès physique.

L'appel système lseek - exemples (Python)

```python
import os, io

# Ouverture du fichier
fd = os.open("thatfile", os.O_RDWR | os.O_CREAT | os.O_TRUNC)
# Positionnement au début du fichier
os.lseek(fd, 0, os.SEEK_SET);
# Positionnement 10 octets après la fin du fichier
os.lseek(fd, 10, os.SEEK_END)
# Positionnement 5 octets avant la position actuelle du curseur
os.lseek(fd, -5, io.SEEK_CUR)
# Fermeture du fichier
os.close(fd)
```

Fichiers à trous

Que se passe t'il si on positionne le curseur après la fin du fichier et qu'on essaye d'y écrire des données ?

L'espace entre la précédente fin du fichier et les octets nouvellement écrits est appelé un trou.

Du point de vue du programmeur, les octets dans un trou sont bien présents et lire les octets dans un trou retourne un tampon composé d'octets qui valent 0.

Cependant, les trous de fichiers n'occupe pas d'espace disque. Le système de fichiers, n'alloue pas les blocs tant que des données n'y sont pas écrites.

Fichiers à trous (suite)

Un fichiers à trou (sparse file) permet d'utiliser le système de fichiers de manière plus efficace, surtout quand le fichier est majoritairement vide de données.

Le système de fichiers stocke les blocs vide (trous) sous forme de méta données plutôt que physiquement stocker les octets représentant l'espace vide.

A la lecture de fichiers à trous, le système de fichiers converti silencieusement les méta données représentant les blocs vide en bloc rempli d'octets à 0.

Les programmes ignorent cette conversion.

Les fichiers à trous sont fréquemment utilisé pour des images disques, des instantanés de base de données, des fichiers journaux, des applications scientifiques, etc.

La bibliothèque stdio

La bibliothèque (stdio) est fournie par la librairie standard et permet de manipuler des fichiers avec des fonctions de plus haut niveau que les appels systèmes.

Les fonctions de cette bibliothèque n'opère pas directement sur les descripteurs de fichiers qui sont une notion spécifique à UNIX mais sur des pointeurs sur une structure qui représente un flux : **la structure FILE**.

La structure de données **FILE** contient entre autres :

- le descripteur de fichier
- un champ *flags* qui indique l'état du flux
- une mémoire tampon
- la position courante dans le tampon

stdio.h

```c
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
int fclose(FILE *stream);

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

int fseek(FILE *stream, long offset, int whence);
int fflush(FILE *stream);

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

Chaque appel système de gestion des fichiers à un équivalent dans stdio.

```
> man 3 stdio
```

L'appel système unlink

unlink() détruit un nom dans le système de fichiers.

```c
#include <unistd.h>

int unlink(const char *pathname);

// renvoie 0 en cas de réussite ou -1 en cas d'échec
// auquel cas errno contient le code d'erreur
```

```
> man 2 unlink
```

```python
import os

os.remove(pathname)
os.unlink(pathname)
```

L'appel système unlink (détails)

unlink() détruit un nom dans le système de fichiers.

Si ce nom était le dernier lien sur un fichier, et si aucun processus n'a ouvert ce fichier, ce dernier est effacé, et l'espace qu'il utilisait est rendu disponible.

Si le nom était le dernier lien sur un fichier, mais qu'un processus conserve encore le fichier ouvert, celui-ci continue d'exister jusqu'à ce que le dernier descripteur le référençant soit fermé.

Si le nom correspond à un lien symbolique, le lien est supprimé.

Exercices

Pour chaque progamme écrit, il est important de gérer les ereurs en utilisant errno et la sortie erreur

- **Question 1.** Écrire un programme en Python qui créé un fichier avec 4 trous de 10000 octets et observer la taille du fichier retournée pas les commandes *du* et *ls*
  1. Que remarquez vous ?
  2. Que pouvez vous en déduire ?
- **Question 2.** Écrire un programme en Python (puis en C) qui affiche le nom et la position des arguments qui lui sont passées en ligne de commande.
- **Question 3.** Écrire un programme en Python (puis en C) qui se comporte comme la commande `rm` et qui supprime tous les fichiers qui lui sont passés en arguments.

Bibliographie

[**APUE**] Advanced Programming in the UNIX Environment.
W. Richard Stevens and Stephen A. Rago.
Addison-Wesley Professional, 2005.

[**TLPI**] The Linux Programming Interface.
Michael Kerisk.
No Starch Press, 2010.