

Cours n°2 : Appels Systèmes et Entrée/Sortie

Victor Poupet

12 Février 2018

Système d'exploitation

- Le système d'exploitation (OS) est le premier programme exécuté au démarrage de l'ordinateur
- C'est l'OS qui gère toute l'activité de l'ordinateur (mémoire, utilisateurs, processeur, périphériques, réseau, etc.)
- En cours : *Unix* (mais les autres sont similaires)
- Les programmes communiquent avec l'OS par l'intermédiaire d'appels systèmes (*system call*)

Appels systèmes

Les appels systèmes ressemblent à des procédures (appels de fonction), avec toutefois des différences importantes :

- Les appels systèmes sont très coûteux en ressources :
 - le système sauvegarde l'état
 - l'OS prend le contrôle du CPU
 - l'OS exécute une tâche spécifique
 - l'OS sauvegarde son état
 - le contrôle est rendu au processus qui a lancé l'appel
- Les appels systèmes dépendent du système d'exploitation
- Il ne faut pas les utiliser directement si le programme doit être portable
- La réalisation détaillée d'un appel système demande une connaissance précise des registres du processeur
- Il faut être très prudent pour ne pas provoquer d'erreurs
- On utilise en général une fonction qui se charge de faire l'appel

Entrée/Sortie

```
int open(char *path, int flags[, int
mode]);
int close(int fd);
int read(int fd, char *buf, int size);
int write(int fd, char *buf, int
size);
off_t lseek(int fd, off_t offset, int
whence);
```

Il existe 5 appels systèmes principaux pour manipuler des fichiers.

- ressemblent à des procédures, mais sont envoyés directement au système souvent on utilise des fonctions qui font un appel système (ex : **fopen** qui appelle **open**)
- les entrées/sorties sont gérées par l'OS pour éviter que des erreurs dans les programmes aient des conséquences sur le système de fichiers
- chaque appel est décrit dans sa page de manuel (ex : **man 2 open**)

man 2 open

OPEN(2)

BSD System Calls Manual

OPEN(2)

NAME

open -- open or create a file for reading or writing

SYNOPSIS

```
#include <fcntl.h>
```

```
int open(const char *path, int oflag, ...);
```

DESCRIPTION

The file name specified by `path` is opened for reading and/or writing, as specified by the argument `oflag`; the file descriptor is returned to the calling process.

The `oflag` argument may indicate that the file is to be created if it does not exist (by specifying the `O_CREAT` flag). In this case, `open` requires a third argument `mode_t mode`; the file is created with `mode`

`mode` as described in `chmod(2)` and modified by the process' `umask` value (see `umask(2)`).

Open

```
int open(char *path, int flags, int mode);
```

```
#include <fcntl.h>
```

```
main()
{
    int fd;

    fd = open("test.txt", O_RDONLY);
    printf("%d\n", fd);
}
```

L'appel `open` sert à demander l'accès à un fichier

- `path` indique le chemin du fichier
- `flags` indique comment on veut manipuler le fichier
- `mode` définit les permissions à donner en cas de création d'un nouveau fichier
- le résultat est un entier correspondant à un descripteur de fichier (ou `-1` en cas d'erreur)
- Tous les autres appels E/S utilisent un descripteur de fichier

Open(flags)

O_RDONLY open for reading only
O_WRONLY open for writing only
O_RDWR open for reading and writing
O_NONBLOCK do not block on open or for data to become available
O_APPEND append on each write
O_CREAT create file if it does not exist
O_TRUNC truncate size to 0
O_EXCL error if **O_CREAT** and the file exists
O_SHLOCK atomically obtain a shared lock
O_EXLOCK atomically obtain an exclusive lock
O_NOFOLLOW do not follow symlinks
O_SYMLINK allow open of symlinks
O_EVTONLY descriptor requested for event notifications only
O_CLOEXEC mark as close-on-exec

- L'argument **flags** est un ou bit-à-bit de valeurs
- Les valeurs possibles sont définies dans `/usr/include/sys/fcntl.h`

Close

```
int close(int fd);
```

```
#include <fcntl.h>
#include <unistd.h>
main() {
    int fd1, fd2;

    fd1 = open("test.txt", O_RDONLY);
    if (fd1 < 0) {
        exit(1);
    }
    fd2 = open("test.txt", O_RDONLY);
    if (fd2 < 0) {
        exit(1);
    }
    if (close(fd1) < 0) {
        exit(1);
    }
}
```

L'appel `close` permet de libérer un descripteur de fichier

- Le résultat est 0 en cas de réussite, -1 en cas d'erreur
- Les descripteurs de fichier sont tous libérés à la fin d'un processus
- Le nombre de fichiers ouverts par un processus est limité (dépend du système, varie d'une centaine à plusieurs milliers)

Read

```
int read(int fd, char *buf, int size);
```

```
int main() {  
    char *b;  
    int fd, sz;  
    b = malloc(sizeof(char) * 11);  
  
    fd = open("test.txt", O_RDONLY);  
    if (fd < 0) {  
        exit(1);  
    }  
  
    do {  
        sz = read(fd, b, 10);  
        b[sz] = '\0';  
        printf("lu: %s\n", b);  
    } while (sz == 10);  
  
    close(fd);  
}
```

L'appel `read` sert à lire des octets dans un fichier

- `fd` est le descripteur du fichier à lire
- `buf` est un pointeur vers un tableau où mettre les caractères lus
- `size` est le nombre d'octets à lire
- le résultat est le nombre d'octets effectivement lus (peut être inférieur à `size` si on est en fin de fichier)

Write

```
int write(int fd, char *buf, int
size);
```

```
int main() {
    int fd, sz;
    char *txt;

    fd = open("test.txt", O_WRONLY |
O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        exit(1);
    }

    txt = "Bonjour\n";
    sz = write(fd, txt, 8);

    close(fd);
}
```

■ L'appel `write` permet d'écrire des octets dans un fichier

- `fd` est le descripteur du fichier où écrire
- `buf` est un pointeur vers un tableau de caractères à écrire
- `size` correspond au nombre de caractères du tableau `buf` à écrire
- le résultat est le nombre d'octets effectivement écrits (devrait toujours être égal à `size`)

Pointeur de fichier

```
int main() {
    char *c;
    int fd, sz;
    c = malloc(sizeof(char) * 10);

    fd = open("test.txt", O_RDWR |
O_APPEND);
    if (fd < 0) {
        exit(1);
    }

    while(1) {
        sz = read(fd, c, 10);
        write(fd, c, sz);
        if (sz < 10) break;
    }

    close(fd);
}
```

Les fichiers ouverts sont tous associés à un *pointeur de fichier* qui indique un emplacement dans le fichier

- Initialement, le pointeur est au début du fichier
- Lorsque l'on utilise `read` ou `write`, la lecture et l'écriture se font au niveau du pointeur
- Le pointeur est automatiquement avancé par la lecture et l'écriture

Ouvertures multiples

```
int main() {  
    int fd1, fd2;  
  
    fd1 = open("test.txt", O_WRONLY |  
    O_CREAT | O_TRUNC);  
    fd2 = open("test.txt", O_WRONLY);  
  
    write(fd1, "un ", 3);  
    write(fd2, "deux ", 5);  
    write(fd1, "trois ", 6);  
}
```

On peut ouvrir plusieurs fois un même fichier (même en écriture)

- chaque descripteur a son propre pointeur de fichier
- les ouvertures en mode **O_APPEND** restent à la fin du fichier

Ouvertures multiples

```
int main() {  
    int fd1, fd2;  
  
    fd1 = open("test.txt", O_WRONLY |  
    O_CREAT | O_TRUNC);  
    fd2 = open("test.txt", O_WRONLY);  
  
    write(fd1, "un ", 3);  
    write(fd2, "deux ", 5);  
    write(fd1, "trois ", 6);  
}
```

→ deux fois

On peut ouvrir plusieurs fois un même fichier (même en écriture)

- chaque descripteur a son propre pointeur de fichier
- les ouvertures en mode **O_APPEND** restent à la fin du fichier

Ouvertures multiples

```
int main() {  
    int fd1, fd2;  
  
    fd1 = open("test.txt", O_WRONLY |  
    O_CREAT | O_TRUNC | O_APPEND);  
    fd2 = open("test.txt", O_WRONLY |  
    O_APPEND);  
  
    write(fd1, "un ", 3);  
    write(fd2, "deux ", 5);  
    write(fd1, "trois ", 6);  
}
```

On peut ouvrir plusieurs fois un même fichier (même en écriture)

- chaque descripteur a son propre pointeur de fichier
- les ouvertures en mode **O_APPEND** restent à la fin du fichier

Ouvertures multiples

```
int main() {  
    int fd1, fd2;  
  
    fd1 = open("test.txt", O_WRONLY |  
        O_CREAT | O_TRUNC | O_APPEND);  
    fd2 = open("test.txt", O_WRONLY |  
        O_APPEND);  
  
    write(fd1, "un ", 3);  
    write(fd2, "deux ", 5);  
    write(fd1, "trois ", 6);  
}
```

→ un deux trois

On peut ouvrir plusieurs fois un même fichier (même en écriture)

- chaque descripteur a son propre pointeur de fichier
- les ouvertures en mode **O_APPEND** restent à la fin du fichier

stdin, stdout, stderr

```
main() {  
    char c;  
    while (read(0, &c, 1) == 1) {  
        write(1, &c, 1);  
    }  
}
```

Les trois premiers descripteurs de fichiers sont automatiquement attribués à chaque processus :

- 0 : entrée standard (**stdin**)
- 1 : sortie standard (**stdout**)
- 2 : sortie d'erreur (**stderr**)

On peut directement utiliser ces descripteurs de fichiers sans avoir à utiliser **open**.

lseek

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

```
int main() {  
    char c[10];  
    int fd, sz;  
    fd = open("test.txt", O_RDWR);  
    if (fd < 0) exit(1);  
  
    int rp = 0;  
    int wp = lseek(fd, 0, SEEK_END);  
    do {  
        lseek(fd, rp, SEEK_SET);  
        sz = read(fd, c, 10);  
        rp = lseek(fd, 0, SEEK_CUR);  
  
        lseek(fd, wp, SEEK_SET);  
        write(fd, c, sz);  
        wp = lseek(fd, 0, SEEK_CUR);  
    } while (sz == 10);  
    close(fd);  
}
```

lseek permet de déplacer manuellement le pointeur de fichier

- fd désigne le descripteur de fichier
- offset indique le nombre d'octets de déplacement
- whence permet de décrire la référence du déplacement
 - SEEK_SET début du fichier
 - SEEK_CUR position actuelle
 - SEEK_END fin du fichier
- Le résultat est la position absolue du curseur après déplacement

Il est possible de déplacer le curseur au-delà de la fin du fichier

stdio.h

```
int main() {  
    FILE *fpin, *fpout;  
    int i=0;  
  
    fpin = fopen("test.txt", "r");  
    fpout = fopen("res.txt", "w");  
    char s[80];  
  
    while(fgets(s, 80, fpin)) {  
        fprintf(fpout, "ligne %d: %s", i,  
            s);  
        i++;  
    }  
  
    fclose(fpin);  
    fclose(fpout);  
}
```

En général, on n'appelle pas directement les appels systèmes mais on utilise des fonctions de la librairie C qui les appellent indirectement :

- `fopen`, `fclose`
- `fgetc`, `fgets`, `fscanf`, etc.
- `fputc`, `fprintf`, `fputs`, etc.

Ces fonctions manipulent les fichiers par l'intermédiaire de pointeurs de fichiers (*file pointers*) de type `FILE*` (structure contenant un descripteur de fichier, et des informations supplémentaires)