Programmation C et Système
Entrées/sorties sur descripteur de fichier

Régis Barbanchon

L2 Informatique

Descripteurs de fichier

Sous Unix, tout **fichier ouvert dans un processus** est identifié par son `fd` (*file descriptor*), un entier **local au processus** qui est ≥ 0.

Les fonctions `open()` et `creat()` qui ouvrent et créent un fichier retournent donc un `fd`, que l'on utilise avec `read()` et `write()` pour lire et écrire des données dans le fichier ainsi ouvert.
Enfin, on utilise `close()` pour fermer le fichier associé à un `fd`.

Par convention, les shells UNIX associent aux numéros `0`, `1`, `2` les `fds` des flux standards du processus :

- `fd 0` (ou `FILENO_STDIN`) : l'entrée standard `stdin`,
- `fd 1` (ou `FILENO_STDOUT`) : la sortie standard `stdout`,
- `fd 2` (ou `FILENO_STDERR`) : la sortie d'erreur `stderr`.

La fonction `open()` retourne toujours le plus petit entier `fd` libre.
Lorsqu'un `fd` est fermé, il redevient libre et réutilisable par `open()`.

La fonction d'ouverture `open()`

`open()` ouvre un fichier de chemin `path` et retourne son `fd` :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (char const path[], int flags [, mode_t mode] );
```

Il faut spécifier exactement un flag parmi les trois suivants : `O_RDONLY` (*read only*), `O_WRONLY` (*write only*), `O_RDWR` (*read/write*).

Il peuvent être combinés en OU bit-à-bit avec (entre autres) :

- `O_TRUNC` : écrase le fichier s'il existe déjà (*truncation*),
- `O_CREAT` : crée le fichier s'il n'existe pas (*creation*),
- `O_CREAT | O_EXCL` : échoue si existe déjà (*exclusive creation*),
- `O_APPEND` : écrit toujours à la fin du fichier.

Avec le flag `O_CREAT`, le 3$^{\text{ème}}$ paramètre `mode` doit être spécifié, indiquant les droits pour le propriétaire, son groupe, et les autres. Cependant, ces indications sont altérées par un masque (`umask`).

Les modes d'ouverture `open()` avec `O_CREAT`

Le mode est une combinaison en OU bit-à-bit des droits suivants, et qui correspondent à ce que la cmd `ls -l -d path` affiche :

- `S_IRUSR = 0400` : `[r-- --- ---]` *owner user can read*,
- `S_IWUSR = 0200` : `[-w- --- ---]` *owner user can write*,
- `S_IXUSR = 0100` : `[--x --- ---]` *owner user can exec* ;
- `S_IRGRP = 0040` : `[--- r-- ---]` *group of owner can read*,
- `S_IWGRP = 0020` : `[--- -w- ---]` *group of owner can write*,
- `S_IXGRP = 0010` : `[--- --x ---]` *group of owner can exec* ;
- `S_IROTH = 0004` : `[--- --- r--]` *others can read*,
- `S_IWOTH = 0002` : `[--- --- -w-]` *others can write*,
- `S_IXOTH = 0001` : `[--- --- --x]` *others can exec*.

Par exemple, la combinaison `S_IRUSR | S_IWUSR | S_IRGRP` correspond aux droits `[rw- r-- ---]` affichés par `ls -l`, sous réserve que le masque `umask` n'altère pas ces indications.

La fonction de masquage des droits `umask()`

`umask()` change le masque des droits non-honorés pour `open()`.
Elle retourne aussi l'ancien masque des droits non-honorés.

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask (mode_t mask);
```

Le masque `mask` est un OU bit-à-bit des droits vus précédement.
Ces droits ne seront pas honorés dans le mode indiqué à `open()`.

Un masque typique est `S_IWGRP | S_IWOTH = 0022`, qui refuse les droits en écriture pour le groupe du propriétaire et les autres.
Donc une indication de droit d'écriture ne sera honorée par `open()` que si c'est celle du propriétaire, c'est-à-dire `S_IWUSR`.

En d'autre termes, lorqu'on spécifie un mode `mode` à `open()`, la fonction n'honore que `mode & ~mask`, le mode privé du masque, soit le ET bit-à-bit du mode avec le NON bit-à-bit du masque.

La fonction historique de création `creat()`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (char const path[], mode_t mode);
```

À l'époque où la fonction `open()` n'avait pas le flag `O_CREAT`, la fonction `creat()` était le seul moyen de créer un fichier. Elle est obsolète et équivalente à `open()` avec les flags : `O_WRONLY | O_CREAT | O_TRUNC`.

```
// File open.c
#include <stdlib.h>
#include <stdio.h>
```

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

```
int main (void) {
  umask (0);
  int flags=  O_WRONLY | O_CREAT | O_TRUNC;
  mode_t mode= S_IRUSR | S_IWUSR | S_IRGRP; // the octal value 0640
  int fd= open ("./hello.txt", flags, mode);
  if (fd < 0) { perror ("open()"); exit (1); }
  return 0; // all fds are closed when process terminates
}
```

```
$ ./open
$ ls -l ./hello.txt
-rw-r----- 1 regis prof 0 Feb 23 16:28 ./hello.txt
```

La fonction de positionnement `lseek()` (1/2)

`lseek()` manipule la position de la prochaine lecture/écriture.
À l'ouverture, la position est soit à 0, soit à la fin avec `O_APPEND`.

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek (int fd, off_t offset, int from);
```

La position est déplacée par un `offset` depuis une orgine `from` :
Si `from == SEEK_SET`, alors `offset` est relatif au début du fichier.
Si `from == SEEK_CUR`, alors `offset` est relatif à la pos actuelle.
Si `from == SEEK_END`, alors `offset` est relatif à la fin du fichier.

La nouvelle position est retournée, ou `-1` en cas d'erreur.

Cas particulier :
`lseek(fd, 0, SEEK_CUR)` retourne la position actuelle,
ou en cas d'erreur, détecte que le `fd` n'est pas *seekable*,
comme par exemple pour les terminaux ou les pipes.

La fonction de positionnement `lseek()` (2/2)

Ce programme teste si `stdin`, `stdout`, et `stderr` sont *seekable*.

```c
// File: lseek.c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <sys/types.h>
#include <unistd.h>

int main (void) {
  bool canSeekStdin = lseek (STDIN_FILENO , 0, SEEK_CUR) != -1;
  bool canSeekStdout= lseek (STDOUT_FILENO, 0, SEEK_CUR) != -1;
  bool canSeekStderr= lseek (STDERR_FILENO, 0, SEEK_CUR) != -1;
  printf ("can seek stdin ? %s\n", canSeekStdin ? "yes" : "no");
  printf ("can seek stdout? %s\n", canSeekStdout? "yes" : "no");
  printf ("can seek stderr? %s\n", canSeekStderr? "yes" : "no");
  return 0;
}
```

```
$ ./lseek
can seek stdin ? no
can seek stdout? no
can seek stderr? no
```

```
$ ./lseek < ./hello.txt | nl
     1  can seek stdin ? yes
     2  can seek stdout? no
     3  can seek stderr? no
```

```
$ ./lseek  2> errors.txt
can seek stdin ? no
can seek stdout? no
can seek stderr? yes
```

On voit que les IO sur le terminal ne sont pas *seekable*.

De même, les IO en pipe ne sont pas *seekable*.

Mais quand un flux est redirigé sur un fichier, il est *seekable*.

La fonction d'écriture `write()` (1/2)

`write()` écrit une série de bytes sur un descripteur de fichier :

```
#include <unistd.h>
ssize_t write (int fd, void const * buffer, size_t n);
```

On demande d'écrire `n` bytes à partir de l'addresse `buffer` sur `fd`.

Le nombre de bytes `m` réellement écrits est retourné,
ou `-1` en cas d'erreur : `fd` invalide, pipe fermé par le lecteur...
Normalement `m == n`, mais on peut parfois avoir `m < n` :
disque plein, quota dépassé, fichier trop long, ...

Pour un fichier régulier, l'écriture se fait depuis la position actuelle,
sauf avec `O_APPEND` où la position se déplace à la fin du fichier.
La position actuelle est ensuite déplacée de `m` bytes.

Remarque : la taille `n` est non-signée de type `size_t`, alors que
la taille retournée est signée de type `ssize_t` (elle peut valoir `-1`).

La fonction d'écriture `write()` (2/2)

Voici un exemple combinant des appels à `write()` et `lseek()` :

```
// file: write.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main (void) {
  char * lines[2]= { "hello world!\n", "HELLO\n" };
  write (STDOUT_FILENO, lines[0], strlen(lines[0]));
  bool couldSeek= lseek (STDOUT_FILENO, 0, SEEK_SET) != -1;
  write (STDOUT_FILENO, lines[1], strlen(lines[1]));
  if ( ! couldSeek) fprintf (stderr, "could not seek stdout.\n");
  return 0;
}
```

```
$ ./write 1> ./hello.txt
$ cat ./hello.txt
HELLO
world!
```

```
$ ./write
hello world!
HELLO
could not seek stdout.
```

```
$ ./write | nl
could not seek stdout.
     1  hello world!
     2  HELLO
```

Quand `stdout` est redirigé vers un fichier, `lseek()` réussit et `"HELLO\n"` écrase `"hello "` en début de fichier.

Quand `stdout` est le terminal ou un pipe, `lseek()` échoue et `"HELLO\n"` est écrit à la suite de `"hello world!\n"`.

Noter aussi la course entre `stderr` de `./write` et `stdout` de `nl` pour l'écriture sur le terminal : le msg d'erreur est arrivé ici en 1er.

La fonction de lecture `read()` (1/2)

`read()` lit une série de bytes sur un descripteur de fichier.

```
#include <unistd.h>
ssize_t read (int fd, void * buffer, size_t n);
```

La fonction peut lire jusqu'à `n` bytes, qu'elle écrit dans `buffer`.
Ce dernier doit donc avoir au moins une capacité de `n` bytes,
voire `n+1` bytes si l'on souhaite rajouter un `'\0'` ensuite.

Le nombre `m <= n` de bytes réellement lus est retourné,
ou `0` en fin de lecture (fin de fichier, pipe fermé par l'écrivain...),
ou `-1` en cas d'erreur (`fd` invalide, ...).
Remarques : la taille `n` est non-signée de type `size_t`, alors que la taille retournée est signée de type `ssize_t` (elle peut valoir `-1`).

La fonction de lecture `read()` (2/2)

Voici un prog où `read()` lit `stdin` par paquets de 8 bytes max :

```c
// File: read.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main (void) {
  for (;;) {
    size_t n= 8;
    char buffer [n+1];  // extra-byte for terminal '\0'
    ssize_t m= read (STDIN_FILENO, buffer, n);
    if (m == -1) { perror ("read()"); exit (1); }
    if (m ==  0) break; // end of file detected
    buffer[m]= '\0';    // make buffer a proper string
    printf("%d bytes read: <%s>\n", (int) m, buffer);
  }
  return 0;
}
```

```
$ printf 'hello world!' | ./read
8 bytes read: <hello wo>
4 bytes read: <rld!>
```

```
$ printf 'hello world!\nCIAO!\n' | ./read
8 bytes read: <hello wo>
8 bytes read: <rld!
CIA>
3 bytes read: <O!
>
```

`"hello world!"` est lu en 2 paquets de 8+4 bytes.

`"hello world!\nCIAO!\n"` est lu en 3 paquets de 8+8+3 bytes.

Un `read()` suplémentaire retourne `0`, détectant la fin du pipe.

La fonction de fermeture `close()` (1/2)

La fonction `close()` ferme un descripteur de fichier.

```
#include <unistd.h>
int close (int fd);
int fsync (int fd);
```

Le descripteur devient libre est peut être réutilisé par `open()`.
La fonction retourne `0` si elle réussit, et `-1` sinon.

La fonction peut échouer pour de multiple raisons :
par exemple à cause d'une erreur générée par `write()` antérieur, mais dont l'exécution effective a été retardée par le système, entrainant une detection de l'erreur tardive lors du `close()`.

Il ne faut pas tenter de rappeler `close()` si l'appel a échoué.

Une fermeture réussie ne garantit pas que l'écriture physique a été complètement effectuée, sauf si un appel à `fsync()` est effectué après la dernière écriture et avant la fermeture.

La fonction de fermeture `close()` (2/2)

Voici un prog créant 3 fichiers et les fermant après une écriture :

```c
// File close.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

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

```c
int main (void) {
  char * filenames[3]= { "hello.txt", "all-fine.txt", "bye.txt" };
  char * texts    [3]= { "Hello!\n" , "All fine?\n" , "Bye!\n"  };
  for (int k= 0; k < 3; k++) {
    int flags=  O_WRONLY | O_CREAT | O_TRUNC;
    mode_t mode= S_IRUSR | S_IWUSR | S_IRGRP; // the octal value 0640
    int fd= open (filenames[k], flags, mode);
    if (fd < 0) { perror ("open()"); exit (1); }
    fprintf (stderr, "using fd %d for file <%s>\n", fd, filenames[k]);
    ssize_t n= write (fd, texts[k], strlen (texts[k]));
    if (n < 0)          { perror ("write()"); }
    if (close (fd) < 0) { perror ("close()"); }
  }
  return 0;
}
```

On voit que le `fd 3` est réutilisé 3 fois, car libéré par `close()` :

```
$ ./close
using fd 3 for file <hello.txt>
using fd 3 for file <all-fine.txt>
using fd 3 for file <bye.txt>
```

```
$ cat hello.txt all-fine.txt bye.txt
Hello!
All fine?
Bye!
```