

I/O (entrées/sorties)

Guillaume Chanel - guillaume.chanel@hesge.ch

Florent Glück - florent.gluck@hesge.ch

May 28, 2025

ISC - HEPIA

Descripteurs de fichiers

Les fichiers dans UNIX

Dans un OS de type UNIX, presque toutes les ressources sont représentées par des fichiers¹ :

- Les fichiers stockés
- Les fichiers périphériques réels : `/dev/sda`, `/dev/rtc`, etc.
- Les fichiers périphériques virtuels : `/dev/urandom`, `/dev/stdout`, etc.
- Les fichiers virtuels (données du noyau présentées comme des fichiers) :
 - `/proc` : informations sur le noyau et les processus
 - `/sys` : informations sur les bus, drivers, etc.
- Les sockets et pipes
- etc.

¹Mais aussi accédées via l'API des fichiers

Descripteur de fichiers

Un descripteur de fichier est représenté par **un entier non-négatif** qui correspond à un **index dans la table des fichiers ouverts du processus**

- Trois descripteurs de fichiers sont automatiquement créés par le noyau à la création d'un processus :
 - 0 → entrée standard (stdin)
 - 1 → sortie standard (stdout)
 - 2 → erreur standard (stderr)

Descripteurs de fichiers et fichiers ouverts : exemple

Processus 1 :

```
int fd1 = open("msg.txt", O_RDONLY);  
int fd2 = open("/var/local/db.dat", O_RDWR);  
int fd3 = open("msg.txt", O_WRONLY);
```

Processus 2 :

```
int fd1 = open("/tmp/pipo", O_RDWR);  
int fd2 = open("msg.txt", O_RDONLY);
```

Processus 3 :

```
printf("hello\n");    // write(1, "hello\n", 6);  
close(1);  
printf("houba\n");    // write(1, "houba\n", 5);  
open("/home/zoe/hop", O_RDWR); // existe  
printf("yes!\n");     // write(1, "yes!\n", 4);
```

Descripteurs de fichiers et fichiers ouverts : exemple

Structures internes du **noyau**

Table des **descripteurs de fichiers**
(locale à chaque processus)

Processus 1

0	
1	
2	
3	
4	
5	

Processus 2

0	
1	
2	
3	
4	

Processus 3

0	
1	
2	

Table des **fichiers ouverts**
(globale au système)

O_RDONLY	1	n/a
O_RDWR	1	0
O_WRONLY	1	n/a
O_WRONLY	1	0
O_RDONLY	1	n/a
O_WRONLY	1	n/a
O_WRONLY	1	n/a
O_RDONLY	1	0
O_WRONLY	1	n/a
O_RDWR	1	0
O_RDONLY	1	0
O_RDONLY	1	n/a
O_RDWR	1	4
O_WRONLY	1	n/a

Table des **inodes**
(globale au système)

/dev/pts/5
/var/local/db.dat
/dev/pts/11
msg.txt
/tmp/pipo
/dev/pts/7
/home/zoe/hop

compteur réf → position courante dans le fichier

Ouverture d'un fichier

A l'ouverture d'un fichier :

- Un descripteur est alloué dans la **table des descripteurs du processus**
 - la valeur du descripteur est l'**indice** de la première entrée libre dans la **table**
 - le descripteur pointe vers l'entrée dans la **table des fichiers ouverts**
- Une entrée est allouée dans la **table des fichiers ouverts** du noyau
 - cette entrée contient : mode d'accès, position, etc.
 - cette entrée pointe sur l'entrée dans la **table des inodes**
- À la première ouverture du fichier, une entrée est allouée dans la **table des inodes** du noyau

Table des fichiers ouverts

Une entrée de la **table des fichiers ouverts** du noyau contient :

- Mode d'accès aux données (`O_RDONLY`, `O_WRONLY`, `O_RDWR`)
- Pour les fichiers normaux : la **position courante**
- L'état du descripteur (`O_APPEND`, `O_NONBLOCK`, `O_ASYNC`)
- Un compteur de références pointant sur l'entrée
- Le numéro d'inode du fichier
- etc.

Opérations sur les fichiers

Les mêmes fonctions sont utilisées, quel que soit le type de fichier :

Opération	Appel système
Ouverture	open
Lecture de données	read
Écriture de données	write
Déplacement pointeur lecture/écriture ¹	lseek
Contrôle du fonctionnement	fcntl, dup, dup2
Fermeture	close

¹Si supporté par le type d'objet

Ouverture, fermeture et contrôle

Ouvrir un fichier (open)

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

- `pathname` est le nom du fichier
- `flags` est un champ de bits indiquant le mode d'accès au fichier
- `mode` indique les permissions si le fichier est créé par open (cf. `O_CREAT`)
- Retourne soit le descripteur de fichier créé, soit -1 en cas d'erreur
- Le descripteur retourné est le plus petit descripteur possible (le premier disponible)

Flags

- Les *flags* suivants peuvent être passés à la fonction `open` :

<code>O_RDONLY</code>	Lecture seule
-----------------------	---------------

<code>O_WRONLY</code>	Écriture seule
-----------------------	----------------

<code>O_RDWR</code>	Lecture et écriture
---------------------	---------------------

<code>O_APPEND</code>	Écrit à la fin
-----------------------	----------------

<code>O_CREAT</code>	Crée le fichier s'il n'existe pas
----------------------	-----------------------------------

<code>O_TRUNC</code>	Efface le fichier s'il existe
----------------------	-------------------------------

<code>O_EXCL</code>	Erreur si <code>O_CREATE</code> est spécifié et que le fichier existe
---------------------	---

- Les trois premiers *flags* sont particuliers :
 - on ne peut pas les combiner entre eux
 - on doit en passer au moins un

Mode (permissions)

Lorsque que le flag `O_CREAT` est passé, il faut spécifier les permissions :

```
int fd = open("/tmp/foo.txt", O_RDWR | O_CREAT | O_EXCL, 0640);
```

Si `O_CREAT` est absent, le mode est ignoré

Ouverture multiple

Si un même fichier est ouvert **plusieurs fois** par un ou plusieurs processus :

- Plusieurs descripteur de fichiers sont créés
- Dans le cas d'un fichier normal, chaque descripteur aura sa position dans le fichier (et éventuellement ses propres *buffer*)
- Toutes les opérations s'effectuent indépendamment et en parallèle
- **Attention :**
 - Le développeur est responsable de coordonner l'accès aux fichiers
 - Pour en savoir plus : voir les verrous (*lock*)

Fermer un fichier (close)

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

- Le descripteur de fichier est libéré et pourra être recyclé
- Retourne 0 en cas de succès et -1 en cas d'erreur
- Le compteur de référence de l'entrée dans la table des fichiers ouverts est décrémenté
 - s'il atteint 0, l'entrée est supprimée et les ressources libérées
- Lors de la terminaison d'un processus via les fonctions `exit` ou `abort`, le noyau ferme tous les descripteurs de fichiers ouverts (équivalent à `close`)

Lecture/écriture

Lecture : bas niveau (read)

Pour lire les données d'un fichier :

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

- Essaie de lire jusqu'à **count** bytes depuis le descripteur **fd**
- Copie les bytes lus dans **buf**
- Retourne le nombre de bytes **effectivement lus** en cas de succès
- Retourne **-1** en cas d'erreur (cf. **errno**)
- Le nombre de bytes lus peut être **plus petit** que ce qui est demandé
- L'indicateur de position du fichier **avance** d'autant de bytes

Ecriture : bas niveau (write)

Pour écrire des données dans un fichier :

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

- Essaie d'écrire jusqu'à **count** bytes sur le descripteur **fd**
- **buf** contient les bytes à écrire
- Retourne le nombre de bytes **effectivement écrits** en cas de succès
- Retourne -1 en cas d'erreur (cf. **errno**)
- Le nombre de bytes écrits peut être **plus petit** que ce qui est demandé
- L'indicateur de position du fichier **avance** d'autant de bytes

read/write : attention

Sur des fichiers réguliers, `read` et `write` retournent moins que demandé quand :

- `read` arrive en fin de fichier
- `write` n'a plus d'espace sur le système de fichiers

Attention :

- Une lecture/écriture peut être interrompue par un signal
- Les utilisateurs peuvent passer à votre programme tous types de fichiers qui se comportent différemment (pipes, sockets, character devices, etc.)
- Attention à gérer les cas où `read/write` réalisent des **opérations partielles**
- Solution alternative : utiliser les I/O bufferisées (`fread/fwrite`)

Positionnement : bas niveau (lseek)

- À l'ouverture d'un fichier, son indicateur de position est initialisé 0
- La position avance à chaque lecture/écriture
- On peut la changer avec (si le type de fichier le permet) :

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

- Modifie la position liée au descripteur `fd`
- `offset` est interprété en fonction de l'argument `whence` :

SEEK_SET `offset`

SEEK_CUR position courante + `offset`

SEEK_END fin du fichier + `offset`

- Si succès → retourne la position en byte depuis le début du fichier
- Si échec → retourne -1 (cf. `errno`)

Lecture : haut niveau (fread)

Pour lire les données d'un fichier :

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

- Lit `nmemb` éléments de données, chacun de `size` bytes, depuis le fichier `stream`, et les stocke dans `ptr`
- Si succès → renvoie le nombre d'éléments lus
 - si `size` vaut 1 → renvoie le nombre de bytes transférés
- Si erreur ou fin de fichier → valeur retour = nombre d'éléments inférieur (ou 0)
- L'indicateur de position du fichier avance du nombre de bytes effectivement lus
- **Pas de distinction entre fin de fichier et erreur !**
 - utiliser `feof(...)` et `ferror(...)` pour déterminer la cause

Fin de fichier vs erreur

Comment distinguer une fin de fichier d'une erreur ?

```
int buffer[42];
size_t nmemb = fread(buffer, sizeof(int), 42, file);
if (nmemb != 42) {
    if (feof(file)) {
        printf("End of file reached.\n");
    }
    else if (ferror(file)) {
        perror("Error reading file");
    }
}
```

Écriture : haut niveau (write)

Pour écrire des données dans un fichier :

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

- Écrit `nmemb` éléments de données, chacun de taille `size` bytes, dans le fichier `stream`, en les obtenant à partir du contenu dans `ptr`
- Si succès → renvoie le nombre d'éléments écrits
 - si `size` vaut 1 → renvoie le nombre de bytes transférés
- Si erreur → valeur retour = nombre d'éléments inférieur (ou 0)
- L'indicateur de position du fichier avance du nombre de bytes effectivement écrits

Positionnement : haut niveau (fseek)

- Fonction similaire à `lseek` mais pour les flux (*stream*)

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

- Modifie la position liée au flux `stream`
- Les autres arguments et comportement sont identiques à `lseek`

Fonctions bas niveau vs haut niveau

- Les fonctions de bas niveau (`open`, `read`, `write`, `close`, `lseek`) s'utilisent via des descripteurs de fichiers
 - ce sont des fonctions *wrapper* qui exécutent directement les appels système correspondants
 - **par abus de langage on les appelle “appels système”**
- Les fonctions de haut niveau (`fopen`, `fread`, `fwrite`, `fclose`, `fseek`) s'utilisent via des flux (*stream*) (type `*FILE`)

- Les fonctions `fread/fwrite` sont **bufférisées** : elles font également appel aux appels système correspondants, **mais** :
 - à chaque appel à la fonction `fread` ou `fwrite` ne correspond pas un appel système
 - en interne elles lisent/écrivent des blocs dans des buffers
 - lorsque les buffers sont pleins, alors l'appel système correspondant est réalisé