

Système de gestion de fichiers

Programmation Système — R3.05

C. Raïevsky

2023



Département Informatique
BUT Informatiques 2^{ème} année

Fichier

Entité logique qui regroupe des données

Système (de gestion) de fichier – *File System*

Ensemble des outils fournis par l'OS pour manipuler les fichiers

Rôles du système de fichiers

- Abstraction du matériel
- Permettre de manipuler les fichiers

Opérations sur les fichiers

Opérations simples

- Création
- Lecture
- Écriture
- Positionnement
- Suppression

Opérations complexes

- Gestion des droits d'accès
- Gestion des accès concurrents
- Gestion de version

Deux modes d'accès : séquentiel et aléatoire

Pour certains types de fichiers, il existe deux modes d'accès :

Séquentiel

Lecture et écriture se font uniquement à partir du début

Aléatoire

Il est possible de déplacer un "curseur" permettant de lire et/ou d'écrire à une position arbitraire

Types de fichiers

Fichiers ordinaires :

Contiennent simplement des données

Répertoires

- Fichiers spéciaux gérés par le système de fichier
- Servent à organiser d'autres fichiers
- De manière hiérarchique

Fichiers spéciaux “caractères” et “bloc”

- Correspondent à un périphérique d'entrée-sortie
- Adressés caractère par caractère (clavier, tubes, sockets)
→ *character device file*
- Adressés bloc par bloc (disques) → *block device file*

Types de fichiers ordinaires

Les fichiers ordinaires peuvent être typés :

Par leur extension (.exe, .bin) - Windows

- L'OS gère en partie les extensions
- Les actions possibles sur un fichier sont conditionnées par cette extension

Par leur contenu - Linux - Unix (cf. `man file`)

- Les extensions sont indicatives
- Les actions possibles sont déterminées par
 - les droits (exécution par exemple) et
 - par le contenu (par exemple `#!/bin/bash` comme première ligne du fichier)

Répertoire – *Directory*

Fichier particulier

- Ne peut pas être manipulé directement
- Constitué d'une liste d'entrées associant
 - un nom de fichier et
 - un numéro de *inode*

→ structure noyau représentant un fichier

Un répertoire peut contenir d'autres répertoires

⇒ Structure hiérarchique

Deux noms peuvent être associés au même inode

⇒ *hard link*

Un nom peut faire référence à un autre nom

⇒ *symbolic link*

inode (ou *i-nud*)

Chaque inode contient des informations sur un fichier

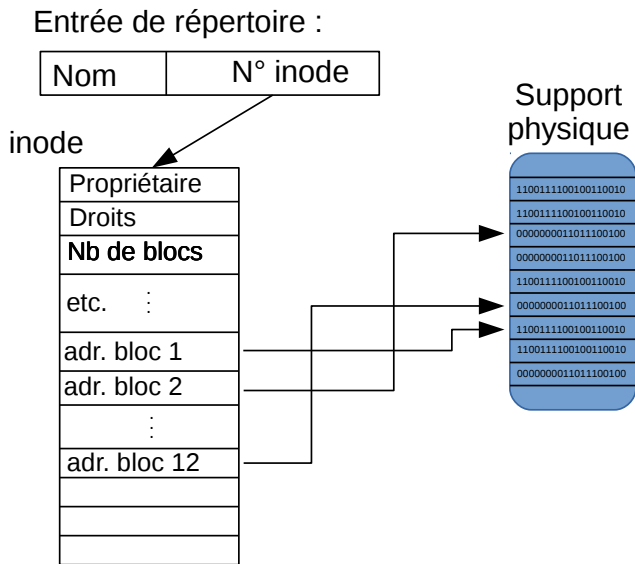
- Propriétaire et groupe
- Droits d'accès
- Taille
- Nombre de lien pointant dessus
- Adresse des blocs de données
- Identifiant du périphérique (pour les *device files*)
- Identifiant du support (disque, partition)

Pour les curieux

```
ls -id / /home  
mount | egrep "/ |home"
```

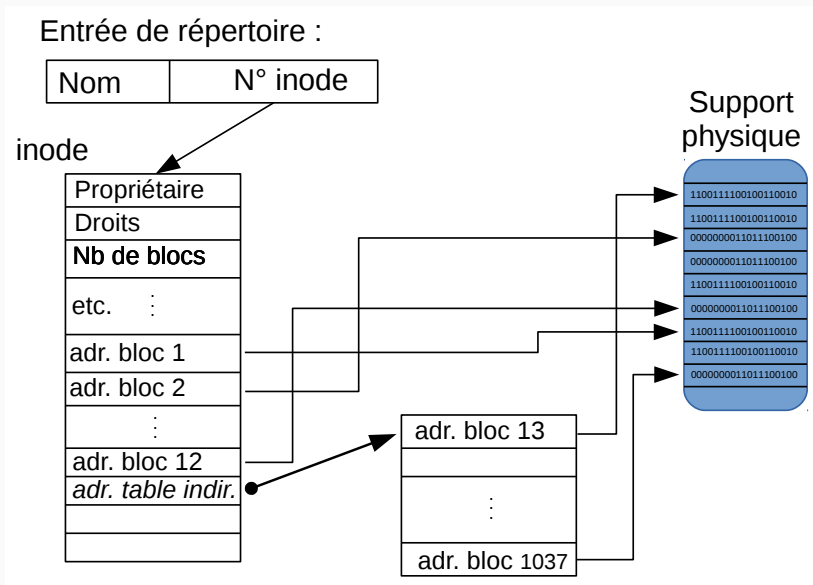

Accès aux données à partir d'un inode

(1/4)



Accès aux données à partir d'un inode

(2/4)



Taille des blocs et taille permise pour les fichiers

Taille typique de bloc : 4Ko

Tailles pour un fichier :

- Accès direct : $12 * 4\text{Ko} \rightarrow 48\text{Ko}$
- Indirection simple : 4 243 456 octets (4Mo)
- Indirection double : 4 297 162 752 octets (4Go)
- Indirection triple : 4 402 347 966 464 octets (4 To)

La taille des blocs peut être adaptée à l'utilisation

Machine de dev ↔ serveur de fichiers vidéos

Manipulation des fichiers : deux interfaces

Appels systèmes

`open, read,`
`write, lseek,`
`close, ...`

*Descripteur de
fichier - int*

Librairie C standard

`fopen, fscanf, fgets,`
`fread, fprintf,`
`fclose, fseek, ...`

*Flux - FILE**

Interfaces pour manipuler les fichiers

Descripteur de fichier - `fd`

- type `int`
- Référence vers une donnée noyau
- *file descriptor*
- `open`, `read`, `write`, `lseek`, `close`, ...

Flux - FLIE*

- Structure opaque de la **librairie C**
- Contient un *file descriptor*
- Outils facilitant la manipulation efficace de fichiers :
 - Tampons (*buffers*)
 - Écriture formatée
→ `fprintf`, `sprintf`, `fwrite`, ...
 - Lecture formatée
→ `scanf`, `fscanf`, `fread` ...

Descripteur de Fichier – *File Descriptor* – `fd`

Un Descripteur de Fichier est :

- un identifiant (de type entier)
- associé indirectement à un fichier (sockets, drivers, tubes, etc. sont des fichiers)
- construit/assigné par le noyau

Toutes les opérations noyau sur les fichiers utilisent un `fd`

Création et ouverture - `open`, `creat`

Retournent un `fd`, assigné par le noyau

Lecture, écriture, fermeture, placement - `read`, `write`, `close`, `lseek`

Prennent un `fd` en paramètre

Descripteurs de fichiers

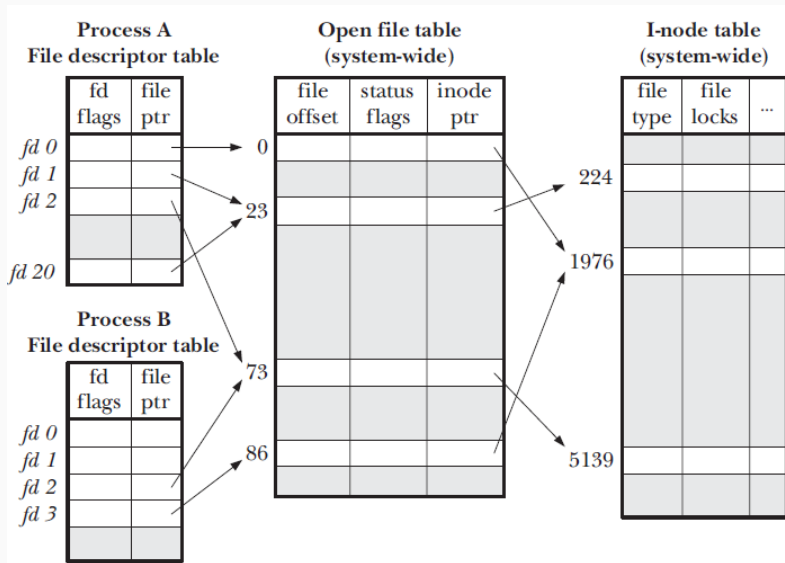
Chaque processus possède une table de descripteurs de fichier

- Chaque entrée dans cette table associe
 - Un *file descriptor*
 - Un pointeur vers une entrée de la **table des fichiers ouverts**
- Initialement il y a 3 fd dans la table : 0, 1, 2
- Cette table survie à un `fork`

Au niveau système : table des fichiers ouverts

- Partagée par tous les processus
- Chaque entrée de cette table contient :
 - Les informations sur l'état du fichier (position du curseur, indicateurs d'erreur et de fin de fichier)
 - Un pointeur vers un i-node

Descripteur de fichier et table des fichiers ouverts



Source : *The Linux Programming Interface*

Flux – *File Stream* ou *File Handle* (FILE*)

Structure de la bibliothèque C **opaque**

- Manipulable uniquement via les fonctions associées
- Encapsulation d'un fd
- Utilisation de buffers
- Informations concernant le flux :
 - position courante
 - indicateur d'erreur
 - indicateur de fin de fichier

Création d'un fichier

Noyau : `open`, `creat`

- retourne un `fd`
- utilisable par les autres fonctions systèmes :
 - `read`, `write`, `pread`, etc.

Librairie C : `fopen`, `fdopen`, `fmemopen`

- retourne un *file stream* (`FILE*`)
- utilisable par les fonctions de `stdio.h` :
 - `fscanf`, `fprintf`, `fread`, `fwrite`, `fgets`, etc.

Options d'ouverture (open, creat)

(1/2)

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

Modes d'accès - requis dans flags

- Lecture seule (O_RDONLY)
- Écriture seule (O_WRONLY)
- Lecture-Écriture (O_RDWR)

Option d'ouverture - autres options pour flags

- O_APPEND
- O_CLOEXEC

...

Options d'ouverture (open, creat)

(2/2)

Options de création

- `O_CREAT` : création du fichier s'il n'existe pas
- `O_DIRECTORY` : erreur si le nom n'est pas un répertoire
- `O_EXCL` : assure la création du fichier (erreur sur fichier existant)
- `O_TRUNC` : efface le contenu du fichier après ouverture
- ...

Gestion des droits à la création - paramètre `mode`

- Pour `O_CREAT` et `O_TMPFILE` uniquement
 - `S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR` → 0700, 0400, 0200, 0100, → `rwX -- --, r- -- --, -w- -- --, -x -- --`
 - `S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP` → 0070, 0040, 0020, 0010, → `-- rwX --, -- r- --, -- -w- --, -- -x --`
 - `S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH` → 0007, 0004, 0002, 0001, → `-- -- rwX, -- -- r-, -- -- -w-, -- -- -x`

Option d'ouverture pour les fonctions de `stdio.h`

```
FILE *fopen(const char *path, const char *mode)
```

man fopen - mode :

r Ouvre le fichier en lecture, curseur positionné au début

r+ Ouvre le fichier en **lecture et écriture**, curseur positionné au début

w **Efface** ou le crée le fichier, l'ouvre en écriture seule, curseur positionné au début

w+ Idem mais ouverture en lecture-écriture

a Ouvre ou crée le fichier en écriture seule, curseur à la fin du fichier.

a+ Ouvre ou crée le fichier en lecture-écriture, curseur d'écriture à la fin du fichier, curseur de lecture au début.

```
fopen("unFichier.ppm", "r+");
```

Conversion `fd` ↔ `FILE*`

`fd` → `FILE*`

```
FILE *fdopen(int fd, const char *mode)
```

- mode identiques à ceux de `fopen`
- doit être compatible avec les options utilisées pour obtenir le `fd`

`FILE*` → `fd`

```
int fileno(FILE *stream)
```

Retourne le `fd` encapsulé dans le *File Stream*

Quelques constantes intéressantes

`fd` ↔ **FILE***

- 0 ↔ `stdin` : Entrée standard
- 1 ↔ `stdout` : Sortie standard
- 2 ↔ `stderr` : Sortie des erreurs standard

Écriture - *file descriptor*

write

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

- Écrit au plus `count` octets depuis le tampon `buf` vers le fichier associé à `fd`
- Retourne le nombre d'octets effectivement écrits
- Écrit à la position courante d'écriture et
- incrémente cette position du nombre d'octets effectivement écrits

En cas d'erreur, moins de '`count`' octets peuvent être écrits

Écriture - *file stream* - FILE*

Écriture non formatée - octets bruts - **fputc**, **fputs**

```
int fputc(int c, FILE *stream);
```

- Écrit le caractère `c` à la position d'insertion courante de `stream`
- Retourne le caractère écrit ou EOF en cas d'erreur

```
int fputs(const char *s, FILE *stream);
```

- Écrit la chaîne `s` à la position d'insertion courante de `stream`
- **Sans son caractère de fin de chaîne** (`\0`)
- Retourne EOF en cas d'erreur, un entier positif sinon

`putchar` et `puts` écrivent un caractère et une chaîne sur `stdout`, respectivement

Écriture sur un *file stream* (FILE*)

Écriture formatée - **fprintf**, **printf**

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

- Prend en paramètre :
 - Un *file stream*
 - Une chaîne de formatage
 - Des pointeurs vers les données à afficher (...)
- Écrit les données fournies selon le format spécifié
- Retourne le nombre de caractères écrit (sans le caractère de fin de chaîne)

Exemple de **fprintf**

```
char nom[20];  
int age;  
...  
fprintf(fileStream, "nom : %s, age : %d\n", nom, age);
```

Écriture sur un *file stream* (FILE*)

Écriture de tableaux de données brutes - **fwrite**

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

- Écrit `nmemb` éléments dans `stream`
- chaque élément ayant une taille de `size` octets
- le premier éléments étant pointé par `ptr`
- Retourne le nombre d'éléments écrits

Exercice

Créer un tableau de 2 Personnes et l'écrire sur la sortie standard à l'aide d'un seul appel à `fwrite`

```
1 typedef struct {
2     int age;
3     char nom[20];
4 } Personne;
```

Prototype de `fwrite` :

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

Écriture sur un *file stream* (FILE*)

(3/3)

```
1 typedef struct {
2     int age;
3     char nom[20];
4 } Personne;
5
6 int main(int argc, const char *argv[]) {
7
8     Personne liste[2];
9
10    liste[0].age = 23;
11    strcpy(liste[0].nom, "Durand");
12
13    liste[1].age = 22;
14    strcpy(liste[1].nom, "Liu");
15
16    ssize_t written = fwrite(liste, sizeof(Personne), 2, stdout);
17
18    printf("\nWrote_%ld_elements.\n", written);
19
20    return EXIT_SUCCESS;
21 }
```

Lecture

fd

- `ssize_t read(int fd, void *buf, size_t count)`
 - Lit **au plus** `count` octets depuis `fd` et les écrit dans `buf`
 - Retourne le nombre d'octet effectivement lus (\leq `count`)
 - Retourne 0 à la fin du fichier

File Stream, non formatée

- `fgetc(FILE*)`, `getc(FILE*)` : Récupère le caractère suivant
- `char *fgets(char *s, int size, FILE *stream)`
 - Lit **au plus** `size` octets depuis `stream` vers `s`
 - La lecture s'arrête en cas de EOF ou fin de ligne ('\n')
 - En cas de fin de ligne, celle-ci est copiée dans `s`
 - Un caractère de fin de chaîne est ajouté à la fin de `s`

Lecture formatée dans un *file stream* (FILE*)

(1/3)

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

- Lecture formatée à partir de `stream`
- Indicateurs de format précisant ce qui doit être lu dans la chaîne `format`
- Stockage des valeurs lues dans les pointeurs fournis après `format`

Exemple de `fscanf`

```
char str1[10], str2[10], str3[10];  
int year;  
fscanf(fileStream, "%s %s %s %d", str1, str2, str3, &year);
```

`gets` → interdit pour cause de sécurité

Lecture formatée dans un *file stream* (FILE*)

(2/3)

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

- Lit nmemb éléments depuis stream
- chaque élément étant de taille size octets
- Stocke les éléments lus à l'adresse ptr

Exercice

Créer un tableau de 2 Personnes à partir du contenu d'un fichier à l'aide d'un seul appel à fread

- `size_t fread(const void *ptr, size_t size, size_t nmemb, FILE *stream);`
- `FILE *fopen(const char *path, const char *mode)`

```
1 typedef struct {  
2     int age;  
3     char nom[20];  
4 } Personne;
```

Lecture formatée dans un *file stream* (FILE*)

(3/3)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5
6 typedef struct {
7     int age;
8     char nom[20];
9 } Personne;
10
11
12 int main(int argc, const char *argv[]) {
13
14     FILE* fileStream = fopen("./listePersonnes.dat", "r");
15
16     if (fileStream == NULL) { perror("File_open_failed."); exit(EXIT_FAILURE); }
17
18     Personne liste[2];
19
20     ssize_t read = fread(liste, sizeof(Personne), 2, fileStream);
21
22     printf("\nRead %ld elements.\n", read);
23
24     printf("Personnel: age: %d, nom: %s\n", liste[0].age, liste[0].nom );
25     printf("Personne2: age: %d, nom: %s\n", liste[1].age, liste[1].nom );
26
27     return EXIT_SUCCESS;
28 }
```

Positionnement

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

- Change la position associée au fichier référencé par `fd`
- Suivant `whence` :
 - `SEEK_SET` : établit la position à `offset`
 - `SEEK_CUR` : ajoute `offset` à la position courante
 - `SEEK_END` : établit la position à la taille du fichier **plus** `offset`
- Retourne la nouvelle position ou `-1` en cas d'erreur

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

- Même comportement que `lseek` sauf
 - retourne `0` en cas de succès et `-1` en cas d'erreur
-
- `ftell` pour obtenir la position courante pour un *file stream*
 - `rewind` permet de remettre la position à `0`

Suppression

```
int remove(const char *pathname)
```

- Détruit l'entrée de répertoire portant le nom `pathname`
- Si `pathname` correspond à un fichier, il est détruit si :
 - Plus aucune entrée de répertoire ne pointe dessus
 - Plus aucun processus ne l'a ouvert
- Si `pathname` correspond à un répertoire, il est détruit si :
 - il est vide
- Retourne 0 en cas de succès, -1 sinon

`remove` appelle `unlink` ou `rmdir` suivant `pathname`

Partage entre processus

Un *file descriptor* survit à un `fork`

- Le processus fils possède une copie de la table des fd de son père
- ⇒ Accès concurrent potentiels
- POSIX précise que les opérations `read` et `write` doivent être atomiques
- Cette exigence est respectée dans Linux depuis la version 3.14

Partage d'un *File Descriptor* – Exemple minimal

```
1 int main(int argc, const char *argv[]) {
2
3     int fd = open("./essai.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
4
5     if (fd == -1) { perror("File_open_failed."); exit(EXIT_FAILURE);}
6
7     ssize_t written;
8
9     pid_t childId = fork(); /* Creation d'un processus fils */
10
11    if (childId == -1) { perror("fork_failed."); exit(EXIT_FAILURE); }
12
13    if (childId == 0) {
14        written = write(fd, "chaine_du_fils\n", strlen("chaine_du_fils\n"));
15        if (written == -1) { perror("Write_to_file_failed."); }
16
17    } else {
18
19        written = write(fd, "chaine_du_pere\n", strlen("chaine_du_pere\n"));
20        if (written == -1) { perror("Write_to_file_failed.");}
21        wait(NULL);
22    }
23    return EXIT_SUCCESS;
24 }
```