Introduction aux systèmes d ’exploitation

Partie 5 : La gestion de fichiers

Jalil BOUKHOBZA

UBO / Lab-STICC

Email : boukhobza@univ-brest.fr

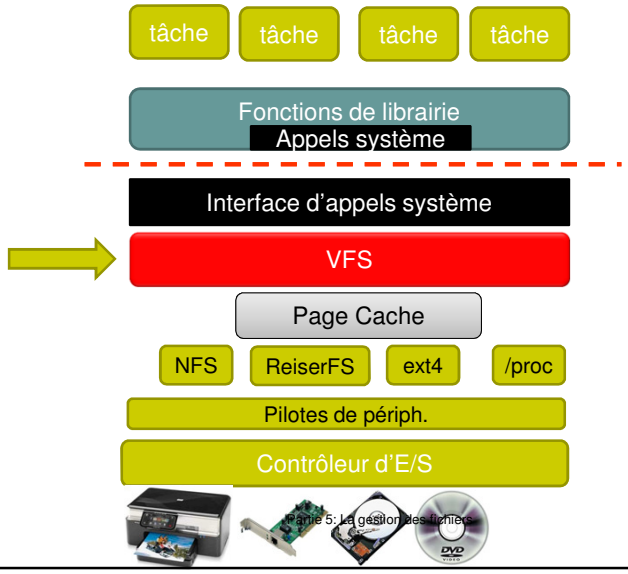
Partie 7 : La gestion de fichiers

1. *Principes généraux*
2. *Le VFS*
3. *Les descripteurs de fichiers*
4. *Opérations sur les fichiers*

Principe général

- De manière générale : “tout est fichier” !
  - Mise en œuvre homogène des entrées/sorties
  - Se résume à des lectures/écritures dans des flots
  - Réutilisation des traitements dans divers contextes
  - Généralisation du terme “fichier”
    - Fichier “réel” d’un système de fichiers
    - Terminal
    - Tube de communication
    - Socket
    - . . .
- Deux notions principales
  - Les descripteurs de fichiers (système)
  - Les flux de données (espace utilisateur)

Le système de fichiers virtuel VFS, un système pivot



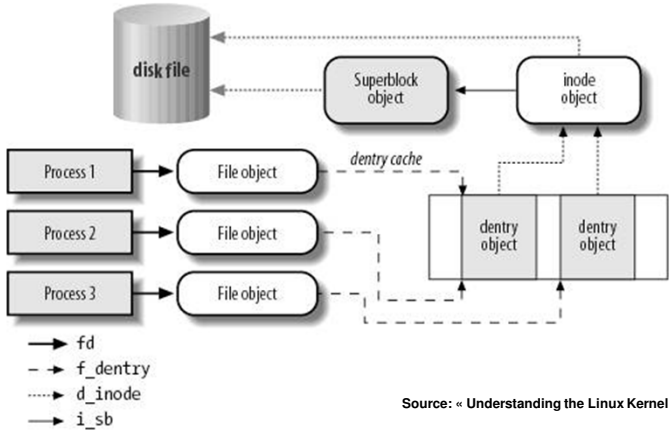
Systèmes de fichiers et VFS

- 3 classes de systèmes de fichiers supportés:
  - Systèmes de fichiers disque (concrets)
    - Gestion d'espace sur stockage secondaire: systèmes de fichiers Linux (ext3, ext4, reiserFS, etc), systèmes de fichiers UNIX (UFS, MINIX, VxFS, etc). Systèmes de fichier Microsoft (VFAT, NTFS, etc), ...
  - Systèmes de fichiers réseaux
    - Accès à des systèmes de fichiers distants: NFS, AFS, CIFS, etc.
  - Systèmes de fichiers spéciaux:
    - Ne gère pas de support de stockage, par exemple: /proc, /sys, /dev ...

VFS

- **Objectif**: Fournir à l'utilisateur une API uniforme pour l'accès aux périphériques et systèmes de fichiers
- Structures principales et opérations associées:
  - **Superblock**: données du système de fichiers (tailles de blocs, nombres de blocs libres, nombre d'i-nœuds, etc.)
  - **i-node** (i-nœuds): un i-nœud par fichier, attributs du fichier
  - **Dentry**: résolution de chemins d'accès (cat /home/coco/kiki.txt : 4 dentries (/, home, coco, kiki.txt)
  - **File**: structure utilisée lorsqu'un processus fait des accès à un fichier

VFS -2-



Source: « Understanding the Linux Kernel », 3rd edition

Structures de données de VFS
L'objet *Superblock*

- Les objets « Superblock »
- Objet: attributs + méthodes (pointeurs de fonction)
- Structure super_block:
  - VFS utilise une double liste chainée circulaire pour relier l'ensemble des super_block
  - Méta données sur le système de fichiers:
    - Taille des blocs, nombre max de fichiers, type du sys.
  - La liste des **méthodes** (struct super_operations): fonctions:
    - Opérations sur les inodes, sur les métadonnées du super block
  - Méta données sur le contenu du sys. de fichiers:
    - Liste des inodes, inodes modifiés, objets fichiers, quota disque, etc.
  - Informations sur les pilotes et périphériques:
    - Pointeur sur le descripteur du pilote de périph. block
    - Nom du périphérique bloc

L'object *inode*

- Informations dont le système à besoin pour manipuler un fichier
- Les inodes d'un super block donné sont relié via des listes chainées.
- Une structure contenant
  - Numéro/identifiant, compteur d'utilisation, mode d'accès, type de fichier, uid, gid, périphérique, taille, taille et nombre de blocks, sémaphore, pointeur sur super-block, pointeur sur pilote de périph.,
  - Des **méthodes**: (struct inode_operations): création, recherche, lien (sym et hard), opérations sur répertoire, permissions, etc.

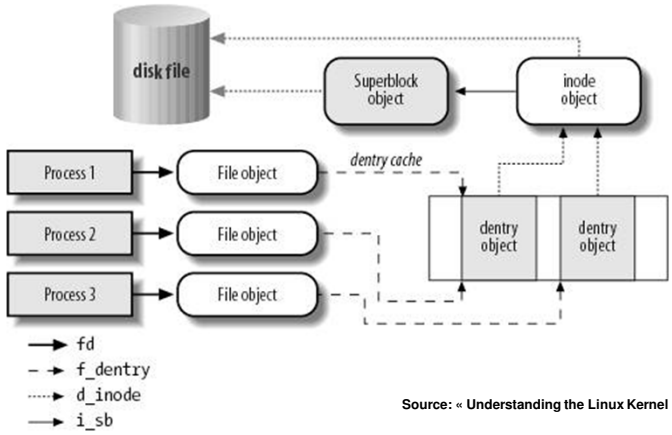
L'objet « *File* »

- Décrit comment un **processus interagit avec un fichier** qu'il ouvre. L'objet est crée à l'ouverture du fichier
- Une structure « file »:
  - Un pointeur sur le dentry, sur le sys. de fichiers monté, compteur d'utilisation, flags d'ouverture, modes, offset, uid, gid, mapping, etc.
  - **Méthodes** (struct file_operations)
    - Open, read, write, aio_read, aio_write, llseek, readdir, ioctl, mmap, fsync, ...,
    - Ensemble de fonctions disponibles à tous les types de fichiers, mais seul un sous ensemble est utilisable

Objets « dentry »

- Un objet dentry est crée pour chaque **composant d'un chemin**: associe chaque composant avec son inode.
  - Exemple: /home/jalil: 3 dentry: « / », « home» et « jalil »
- Structure dentry
  - Inode, dentry parent, nom de fichier, pointeurs sur répertoires et sous répertoires, superblock, etc.
  - **Méthodes**: comparaison de nom, suppression, libération, etc.
- **Cache de dentry**: maintient des objets dentry dans un cache pour réutilisations futures

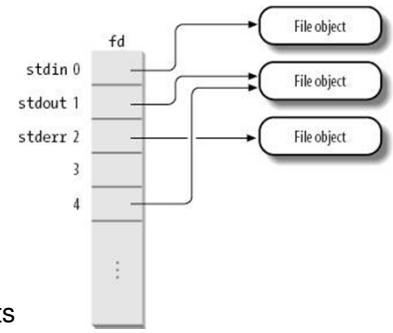
VFS



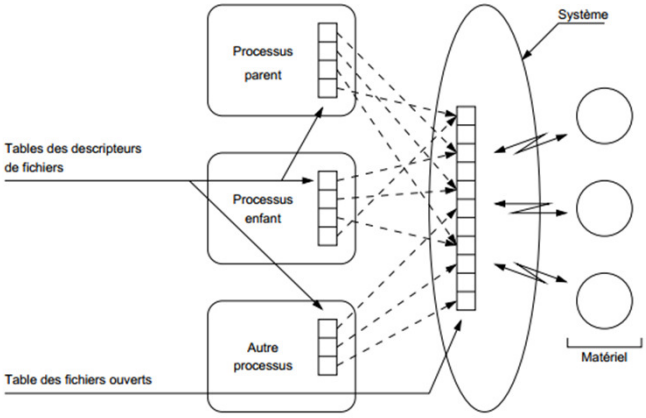
Source: « Understanding the Linux Kernel », 3rd edition

Interaction Processus / fichiers

- Ex: répertoire de travail, répertoire racine
- Le système maintient
  - Une structure par processus: **fs_struct** qui contient:
    - Dentry du rep. De travail et racine, compteur du nombre de processus partageant la table ...
  - Une autre structure (champs files) contenant la liste des fichiers ouverts (struct file**fd) → voir figure



Source: « Understanding the Linux Kernel », 3rd edition



Descripteurs de fichiers

- Opérations spécifiques
  - Ouverture du fichier
  - Traitements particuliers (positionnement . . .)
- Opérations génériques
  - Lecture/écriture
  - Scrutation
  - Paramétrage
  - Fermeture

Fichiers ouverts par défaut

- Ouverture implicite de 3 flots pour chaque processus
  - Entrée standard → lecture depuis le terminal
    - Descripteur de fichier 0 (STDIN_FILENO dans unistd.h)
  - Sortie standard → écriture vers le terminal
    - Descripteur de fichier 1 (STDOUT_FILENO)
  - Sortie d'erreurs → écriture vers le terminal
    - Descripteur de fichier 2 (STDERR_FILENO)
- Modifications explicites
  - Redirection depuis la ligne de commande
  - Redirection/fermeture par le processus parent
  - Redirection/fermeture par le processus lui-même

Ouverture et fermeture d'un fichier

- Cas particulier ici pour l'ouverture
  - Un fichier "réel" dans un système de fichiers
  - Plus général qu'il n'y parait (/dev/ttyS0, /dev/audio ...)
- Ouverture d'un fichier existant
  - Appel système open() (man 2 open)

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

  - path : chemin absolu ou relatif du fichier à ouvrir

Ouverture de fichiers

- Ouverture d'un fichier existant
  - flags : type d'ouverture et paramètres (combinaison bit à bit)
    - Ouverture O_RDONLY, O_WRONLY ou O_RDWR
    - O_CREAT : création si inexistant
    - O_EXCL : échec si O_CREAT mais existant (exclusif)
    - O_APPEND : ajout en fin de fichier
    - O_NONBLOCK, O_SYNC, O_APPEND . . .
  - mode : droits du fichier si O_CREAT
    - Valeur entière (octale) ou combinaison bit à bit
    - S_ISUID, S_ISGID, S_ISVTX

    S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR
    S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP
    S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH

- Ouverture d'un fichier existant
  - Retour de open()
    - Un descripteur de fichier si ouverture correcte
    - -1 si une erreur survient
  - Nombreuses causes d'erreurs (consulter errno)
    - Fichier inexistant, droits insuffisants, flags incorrects ...
- Création d'un nouveau fichier
  - Appel système creat() (man 2 creat)
  - int creat(const char * path,mode_t mode);
    ≡open(path,O_CREAT|O_WRONLY|O_TRUNC,mode);

Synchronisation et fermeture

- Synchronisation d'un flot
  - Appel système fsync() (man 2 fsync)

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

  - fd : descripteur de fichier quelconque
  - Retour : 0 si OK, -1 si mauvais fd ou flot inadapé
- Fermeture d'un flot
  - Trés général (pas uniquement les fichiers)
  - Appel système close() (man 2 close)

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

  - fd : descripteur de fichier quelconque
  - Retour : 0 si OK, -1 si mauvais fd

Lecture de fichier

- Lecture depuis un flot
  - Trés général (pas uniquement les fichiers)
  - Appel système read() (man 2 read)

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

  - Extraction de count octets de fd vers buf (préallablement alloué !)
  - Retour : nombre d'octets extraits, 0 si fin de fichier, ou -1 si erreur
  - Nombreuses causes d'erreurs (consulter errno)
    - EBADF si mauvais fd
    - EINTR si interrompu → relance
    - EAGAIN si ouverture O_NONBLOCK (ou vérouillage) et rien à lire
    - Dépendant de la nature exacte de fd . . .

Écriture de fichier

- Ecriture vers un flot
  - Trés général (pas uniquement les fichiers)
  - Appel système write() (man 2 write)

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

  - Envoi des count octets de buf dans fd
  - Retour : nombre d'octets envoyés ou -1 si erreur
  - Nombreuses causes d'erreurs (consulter errno)
    - EBADF si mauvais fd
    - EPIPE si extémité fermée (tube , socket)
    - EINTR si interrompu → relance
    - EAGAIN si ouverture O_NONBLOCK (ou vérouillage) → relance
  - Dépendant de la nature exacte de fd . .

Paramètrage de fichiers

- L'appel système fcntl() (man 2 fcntl)

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd,int cmd,...);
```

  - fd : le flot à paramétrer
  - cmd : la commande à appliquer (peut nécessiter des arguments)
  - Influer sur les paramètres d'un flot ouvert
    - F_DUPFD, F_GETFD, F_SETFD, F_GETFL, F_SETFL
  - Vérouiller des portions de fichier
    - F_GETLK, F_SETLK, F_GETLKW
  - Communication asynchrone
    - F_GETOWN, F_SETOWN, F_GETSIG, F_SETSIG

Positionnement

- L'appel système lseek() (man 2 lseek)
  - Déplacer/consulter la position courante dans un flot de données
  - Position : le type off_t (≃ entier long)

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

  - Se déplacer de offset octets dans le flot fd selon whence :
    - SEEK_SET : depuis le début du flot
    - SEEK_CUR : depuis la position courante
    - SEEK_END : depuis la fin du flot
  - Retour : nouvelle position(depuis le d´ebut) ou -1 si erreur

Positionnement -2-

- L'appel système lseek()
  - Causes d'erreur (consulter errno) :
    - Mauvais arguments
    - Flot inadapté (tube , socket . . .)
    - Position finale négative
  - Le déplacement ne modifie pas le contenu !
  - On peut dépasser la fin !
    - On peut écrire à cette position
    - read() donne des 0 dans l'espace
  - Consulter la position courante : pos=lseek(fd,0,SEEK_CUR);
  - Consulter la taille du flot : size=lseek(fd,0,SEEK_END);

Parcours des répertoires

- Parcours séquentiel (principe)
  - Ouverture du répertoire par opendir() → structure DIR
  - Lecture des entrées successives par readdir() → structure dirent
  - Lecture du nom de l'entrée par le champ d_name
  - Retour éventuel à la première entrée par rewinddir()
    - (seule fonctionnalité de déplacement POSIX)
  - Fermeture du répertoire par closedir()

- La fonction opendir() (man 3 opendir)

```
#include <sys/types.h>
#include <dirent.h>
DIR * opendir(const char * path);
```

  - Crée la structure DIR associée au répertoire désigné par path (équivalent du FILE* pour les fichiers)
  - Retour : adresse de cette structure ou pointeur nul si erreur
  - Causes d'erreur (consulter errno) :
    - Répertoire inexistant
    - Droits insuffisants
    - …

- La fonction readdir() (man 3 readdir)

```
#include <sys/types.h>
#include <dirent.h>
struct dirent * readdir(DIR * dir);
```

  - Initialise dans *dir* la structure *dirent* décrivant l'entrée suivante

```
struct dirent {
    ino_t d_ino; /* inode number */
    off_t d_off; /* offset to the next dirent */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type; /* type of file; not supported by all file system types */
    char d_name[256]; /* filename */ };
```

  - Retour :
    - Adresse de cette structure si ok
    - Pointeur nul si erreur ou dernière entrée dépassée
  - Causes d'erreur (consulter errno) : dir incorrect

- La fonction rewinddir() (man 3 rewinddir)

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR * dir);
```

  - Repositionne la lecture du répertoire au début
  - Les fonctions seekdir() et telldir() ne sont pas POSIX
- La fonction closedir() (man 3 closedir)

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR * dir);
```

  - Ferme le répertoire et libère la structure dir
  - Retour : 0 si ok, -1 si erreur (dir incorrect)

La vie d'une requête d'E/S

- Un read() traverse plusieurs couches, plusieurs des services/couches de l'OS sont sollicitées:
  - Interface d'appels système
  - VFS
  - Page cache
  - Système de fichiers concret
  - Pilote de périph. bloc générique
  - Ordonnanceur d'E/S
  - Pilote de périph. bas niveau
  - Gestion des interruptions

La vie d'une requête d'E/S

- Appel à la primitive système read()
- Traversée de **l'interface d'appels systèmes**, porte d'entrée du noyau →transformation de l'appel en syscall(SYS_read, ...) ....
- Entrée dans le **VFS**: vfs_read(....)
- Vérification de chaque page (4KO) de Linux dans le page cache ...
- **Page cache**: un tampon contenant les données (fichiers) lus/écrits ou à lire

Page cache

- Cache logiciel en RAM
- Mécanisme de lectures anticipées (read ahead) → localité spatiale
  - Précharger des pages à lire dans le future
  - Basé sur la détection de séquentialité des accès
- Mécanisme de write-back
  - Retarder les écriture sur disque
    - Afin de minimiser le nombre d'écritures et les déplacements de la tête du disque (très lent).
    - O_SYNC (write through), O_DIRECT (minimise l'effet de cache) → voir man

- Appel au **sys. de fichiers** concret → do_sync_read()
  - Initialisaiton des buffers d'E/S
  - do_generic_file_read( ) → readpage() : copie d'une page
  - → mapping page / block du périphérique
- Couche bloc générique
  - → création de la structure bio (Block I/O)
  - Vérification de la requête par rapport au périphérique
  - Insertion dans la bonne file d'E/S

- **Ordonnanceur d'E/S**
  - Plusieurs politiques possibles: *Anticipatory*, *Deadline*, *CFQ* (Complete Fairness Queueing), *Noop* (No Operation)
  - Objectif → réduire les déplacement de la tête de lecture/écriture tout en restant équitable avec l'ensemble des processus réalisant des E/S
- Envoi de la requête via le pilote bas niveau
  - Copie les données via DMA et est notifié par interruption

Résumé

- Chemin complexe comprenant plusieurs couches:
  - VFS et le page cache
  - Le système de fichiers
  - La hiérarchie des périph. (pilotes) blocs
    - Pilote générique
    - Ordonnanceur
    - Pilote bas niveau

Source: *Understanding the Linux kernel* 3rd edition, O'Reilly ed, D.P. Bovet, M.Cesati