# Cours Système n°3: les fichiers 2/2

Éric Gaudefroy

École Hexagone
Cursus Cyberdéfense - M1

## 1. Principes d'accès aux ressources

Cette section introduit les deux grands types de logiques sous-jacents aux accès aux objets du système, qu'il s'agisse de fichiers du système de fichiers, d'objets génériques manipulables par des descripteurs de fichiers (tubes, sockets, etc...) ou encore d'objets divers (paramètres du noyau, files de messages, etc...).

Afin de préserver la cohérence des ressources dans un environnement où plusieurs tâches sont susceptibles de s'exécuter de façon concurrente, des modes d'accès particuliers sont proposés ou imposés par le noyau. La présentation qui suit, orientée ressources, est à articuler avec les cours suivants sur les systèmes multi-tâches (parallélisme, ordonnancement, interruptions) et les questions de synchronisation (mutex, sémaphores), cours qui adopteront une approche plus orientée système.

*Remarque* : Les files d'attente associées aux différents objets de synchronisation du système qui seront présentés ici (verrous, ressources producteur/consommateur, etc...) sont, dans la pratique, gérées selon les priorités de l'ordonnanceur (cf. cours sur les processus). Pour simplifier, on pourra en première approximation se les représenter comme des files d'attente de type « Premier arrivé, premier servi » (First In First Out, ou FIFO en anglais).

### 1.1. Logique lecteur / écrivain

L'accès à une ressource peut s'effectuer en mode lecteur/écrivain. On distingue alors deux rôles :

- le lecteur, qui accède à tout ou partie de la ressource sans la modifier (même transitoirement) ;
- l'écrivain, qui est susceptible de modifier transitoirement ou durablement la ressource.

La ressource elle-même est généralement vue « à plat » , c'est-à-dire qu'il y a une notion sous-jacente de contenu persistant et que chaque acteur est susceptible d'accéder à l'ensemble de ce contenu et de s'y déplacer (*lseek*) [1]. Pour conserver la cohérence de la ressource, on essaye généralement d'éviter les accès concurrents en écriture. De même, pour éviter qu'un lecteur ne puisse voir une version incohérente de l'objet (c'est-à-dire une version intermédiaire qui ne correspond ni à l'état cohérent avant écriture ni à l'état cohérent après écriture), on évite d'autoriser l'accès en lecture pendant une écriture.

1. Cas particulier : sur un fichier ouvert en mode O_APPEND, chaque opération d'écriture est automatiquement recalée en fin de fichier.

L'algorithme de synchronisation le plus élémentaire est le suivant :

- gérer une file d'attente pour l'accès à la ressource ;
- lorsque quelqu'un se présente et que la file est non vide, le positionner en queue de file ;
- lorsqu'un lecteur arrive et que la file est vide :
  si un écrivain utilise la ressource, mettre le lecteur dans la file ;
  sinon, faire entrer le lecteur (même s'il y a déjà d'autres lecteurs) ;
- lorsqu'un écrivain arrive et que la file est vide :
  si quelqu'un utilise la ressource (lecteur(s) ou écrivain), mettre l'écrivain dans la file ;
  sinon, faire entrer l'écrivain ;
- lorsque le dernier lecteur ou l'écrivain sort, faire rentrer l'écrivain ou les lecteurs en tête de file d'attente.

On peut facilement démontrer que les requêtes sont traitées dans l'ordre d'arrivée (caractère équitable de l'algorithme) et qu'à un instant donné il y a 0-1 écrivain ou 0-n lecteurs qui accèdent à la ressource. Par ailleurs, et sous ces contraintes, personne n'attend inutilement.

Un algorithme plus simpliste consiste à ne faire entrer qu'une personne à la fois. Ceci risque de bloquer inutilement de nombreux lecteurs.

On peut aussi imaginer des algorithmes plus élaborés permettant de gérer des priorités dans la file d'attente ou encore de limiter le nombre de lecteurs autorisés à accéder simultanément à la ressource pour ne pas trop faire attendre les écrivains. Enfin, on a intérêt à trouver la bonne granularité lors de la définition des ressources afin de ne pas multiplier le nombre de files d'attente tout en évitant de bloquer inutilement les accès parallèles à des portions indépendantes de la ressource.

Se prêtent naturellement aux accès en mode lecteur/écrivain :

- les fichiers réguliers ;
- les répertoires ;
- les périphériques en mode bloc(voir plus bas) ;
- par extension, les liens symboliques pointant vers les types de fichiers mentionnés ci-dessus.

La synchronisation des accès à ces fichiers devra généralement être réalisée de façon explicite (utilisation de verrous ou de sémaphores dans les programmes qui y accèdent, voir les cours suivants), l'OS se bornant au mieux à sérialiser les accès bas niveau individuels (*read*, *write*, etc...), ce qui n'est généralement pas suffisant.

### 1.2. Logique producteur / consommateur

Un autre mode d'accès aux ressources est le mode producteur / consommateur. On y distingue également deux rôles :

- le producteur, qui dépose des objets dans la ressource ;
- le consommateur, qui ramasse les objets présents dans la ressource.

On peut voir la ressource comme un buffer synchronisé de $n$ cases qui sont vides ou contiennent les objets déposés et non encore ramassés. Généralement, ce buffer est géré comme une file (FIFO), ce qui signifie que les objets sont ramassés dans l'ordre où ils ont été déposés. Il n'y a pas de notion de déplacement dans la ressource (*lseek*). Selon les cas, le consommateur peut aussi observer ou compter les objets sans les ramasser.

Pour conserver la cohérence de la ressource, on peut synchroniser les accès de la façon suivante :

- on gère deux files d'attente pour l'accès à la ressource (une pour les producteurs, l'autre pour les consommateurs) ;
- si quelqu'un arrive et que sa file n'est pas vide, on le positionne en queue de file ;
- si un producteur arrive et que sa file est vide :

  s'il reste au moins une case vide, on place sa production dans la première [2] telle case (en la marquant pleine) et on le libère ;

  sinon, on le met dans sa file d'attente.
- si un consommateur arrive et que sa file est vide :

  s'il y a au moins une case pleine, on lui donne le contenu de la première telle case (en la marquant vide) et on le libère ;

  sinon, on le met dans sa file d'attente ;
- lorsqu'on libère un producteur, si la file d'attente des consommateurs est non vide, on fait rentrer le premier consommateur de cette file (cf. traitement de l'entrée d'un consommateur quand sa file est vide) ;
- lorsqu'on libère un consommateur, si la file d'attente des producteurs est non vide, on fait rentrer le premier producteur de cette file (cf. traitement de l'entrée d'un producteur quand sa file est vide).

On peut remarquer que chaque file d'attente est elle-même une FIFO de taille illimitée. L'OS gère cette FIFO (synchronisation) en empêchant matériellement les accès concurrents à certaines de ses propres ressources internes.

Là encore, un algorithme plus simpliste consiste à interdire tout accès concurrent à la ressource. Ceci empêche plusieurs producteurs et consommateurs de travailler en parallèle sur des cases distinctes. Des algorithmes plus élaborés permettent de gérer des priorités dans les files d'attente ou encore de permettre la production/consommation de plusieurs cases à la fois.

Sont naturellement accessibles en mode producteur/consommateur :

- la plupart des périphériques en mode caractère (voir plus bas ; ces fichiers représentent en fait deux ressources distinctes s'ils sont accessibles en lecture et en écriture depuis l'espace utilisateur) ;
- les tubes (pipes) nommés ou anonymes (voir plus bas) ;
- les sockets locales (nommées ou anonymes) ou réseau (voir le cours sur les sockets) ;
- par extension, les liens symboliques pointant vers des fichiers de type périphérique en mode caractère, *tubes* nommés ou *sockets* locales nommées.

2. Ceci suppose implicitement que les productions sont sérialisées, comme les accès en écriture de la logique lecteur / écrivain, pour éviter tout problème lors de l'attribution des cases. Le même problème se pose pour les consommations.

Tous ces types de fichiers sont gérés comme des files d'attente et la synchronisation élémentaire est réalisée de façon transparente par le noyau[3]. À ce titre, ils peuvent eux-mêmes être utilisés comme matière première en vue de réaliser des fonctions de synchronisation de plus haut niveau. En contrepartie, les accès en lecture à une ressource vide ou en écriture à une ressource pleine sont bloquants, c'est-à-dire que le programme réalisant une telle opération sera interrompu jusqu'à ce que la ressource soit disponible. L'option d'ouverture O_NONBLOCK permet des accès non bloquants. Dans ce cas, les opérations qui auraient dû être bloquantes renverront un retour d'erreur et la variable *errno* sera positionnée à EAGAIN (cf. pages de manuel de *read* et *write*).

## 2. Index des types de fichiers UNIX

La section qui suit présente les différents types de fichiers susceptibles d'être rencontrés dans le VFS UNIX[4].

Il existe une commande *Shell* Linux non portable permettant de copier un fichier en préservant toutes ses propriétés, pourvu que la cible soit située sur un système de fichiers suffisamment riche pour le permettre (EXTx). En particulier, si l'original est un fichier spécial (périphérique, tube nommé, lien symbolique, etc...), c'est le fichier qui sera dupliqué plutôt que son contenu apparent.

```
cp -a <src> <dst>
```

On fera ainsi la différence entre **cp /dev/zero /tmp/dump** (qui ne se termine que lorsque la partition est pleine !) et **cp -a /dev/zero /tmp/dump** (qui réalise un mknod). Ce mode est un raccourci pour **-dpR**, et réalise en particulier une copie récursive sur les répertoires.

### 2.1. Fichiers réguliers

Les opérations élémentaires sur les fichiers réguliers ont été décrites dans le cours n°2. La création d'un tel fichier peut s'effectuer par une ouverture si l'option O_CREAT est sélectionnée. Il existe d'autres fonctions plus spécifiques :

- **int fd = create(const char *name, mode_t mode)** : création et ouverture (en écriture seule) d'un nouveau fichier en précisant les permissions à appliquer, moins éventuellement les permissions de umask (POSIX) ;
- **int ret = truncate(const char *name, off_t len)** : troncature d'un fichier existant (BSD, SYSV) ;
- **int ret = ftruncate(int fd, off_t len)** : troncature d'un fichier ouvert en écriture (POSIX) ;
- **FILE * f = tmpfile(void)** : choix d'un nom et ouverture d'un fichier temporaire original qui sera détruit à la fermeture (POSIX ; voir aussi **mkstemp**).

Pour la commande **ls -l**, le premier caractère de la ligne indique le type de fichier. Pour un fichier régulier, ce caractère est « - ».

```
bash$ ls -l /etc/passwd
-rw-r--r-- 1 root root 1111 mai 10 11:35 /etc/passwd
```

3. On peut aussi souhaiter y superposer d'autres formes de synchronisation, de façon explicite, par exemple pour garantir un certain séquencement si plusieurs producteurs ou plusieurs consommateurs accèdent à la même ressource.

4. Il existe, sous Linux, d'autres types d'objets manipulables par l'intermédiaire de descripteurs de fichiers : les *sockets* IP, les queues de message POSIX, etc.... Toutefois, ces objets ne sont pas directement associés à l'espace de nommage du VFS et s'exploitent au moyen de fonctions spécifiques.

Rappelons que les liens durs ne constituent pas des fichiers à part entière, mais représentent simplement la possibilité d'accrocher un même fichier à plusieurs endroits dans un même FS. Il existe une fonction POSIX pour ajouter un lien dur en C :

```
int ret = link(const char *src, const char *dst)
```

## 2.2. Répertoires

Les répertoires ont été présentés dans le cours n°2. Le caractère de type affiché par **ls -ld**[5] est « d » :

```
bash$ ls -ld /tmp
drwxrwxrwt 20 root root 4096 oct 5 09:33 /tmp
```

Dans le système de fichiers EXT, un répertoire est un fichier dont le contenu est le tableau des entrées du répertoire. La norme POSIX définit une API de haut niveau pour un accès structuré en lecture seule aux répertoires :

- **DIR *d = opendir(const char *name)** : ouverture ;
- **int ret = closedir(DIR *d)** : fermeture ;
- **struct dirent *entry = readdir(DIR *d)** : lecture d'un enregistrement ( NULL en fin de répertoire).

Ces fonctions sont rarement utilisées puisque les fonctions d'ouverture de fichiers (**open**) permettent de traverser les répertoires en spécifiant un chemin complet. Les opérations de bas niveau correspondantes, moins portables, sont **open** (en mode O_RDONLY), **close** et **getdents**/**getdirentries**. La création d'un sous-répertoire s'effectue par la fonction POSIX :

```
int ret = mkdir(const char *name, mode_t mode)
```

où l'on précise les permissions à appliquer (modulo le umask). La suppression d'entrées dans un répertoire peut s'effectuer par :

- **int ret = remove(const char *name)** pour les fichiers ou les répertoires vides (ANSI) ;
- **int ret = unlink(const char *name)** pour tout fichier non répertoire (POSIX) ;
- **int ret = rmdir(const char *name)** pour les répertoires vides (POSIX).

Le mot fichier est à prendre au sens large et couvre ici le cas des liens symboliques (suppression du lien).

La suppression d'une entrée dans un répertoire s'accompagne d'une décrémentation du compteur de liens de l'inode correspondant. L'inode n'est complètement libéré que si ce compteur de références passe à 0 (absence d'autres liens durs vers l'inode) et que tous les processus disposant de références dynamiques (notamment via des descripteurs de fichiers) vers l'inode au moment de la suppression les ont refermées. D'un point de vue pratique, cela signifie que la suppression d'un fichier empêche toute nouvelle ouverture à travers le chemin VFS concerné mais que les processus qui l'utilisent déjà peuvent continuer à le faire (en mémoire). Attention, la libération physique et/ou en mémoire de l'inode n'implique pas l'écrasement physique des données et méta-données sur le disque, même après synchronisation des caches. Dans le cas de EXT, l'inode et ses blocs sont simplement marqués libres dans les cartes (*bitmaps*) correspondantes (et le *dtime* est renseigné, etc...).

Enfin, le déplacement ou le renom mage d'éléments d'un répertoire s'effectue par :

5. L'option -d indique à ls de ne pas lister le contenu des répertoires.

```
int ret = rename(const char *oldname, const char *newname)
```

(renommage et/ou déplacement d'un fichier (au sens large) sans changement de partition, en supprimant la cible si elle existe (POSIX)).

## 2.3. Liens symboliques

Les liens symboliques ont été introduits dans le cours n°2. Pour ls -l, leur caractère de type est « l » :

```
bash$ ls -l /etc/rc3.d/S10network
lrwxrwxrwx 1 root root 17 mai 10 13:09 /etc/rc3.d/S10network -> ../init.d/ne
```

Normalement, on n'accède jamais au lien symbolique en tant que tel mais au fichier pointé s'il existe. Pour cette raison, les permissions sur le lien lui-même ont peu d'importance, le contrôle d'accès pertinent ayant lieu lors de l'ouverture du fichier pointé. L'invocation de **chmod** sur un lien symbolique résulte d'ailleurs en un changement de permissions sur le fichier pointé. À noter que sous Linux, la fonction **open** admet une option non standard O_NOFOLLOW si l'on souhaite explicitement ouvrir un fichier en évitant d'être redirigé s'il s'agit d'un lien symbolique [6] : s'il s'agit d'un lien symbolique, **open** génère alors une erreur. Dans le système de fichiers EXT, le lien symbolique est en principe un fichier dont le contenu est le chemin vers le fichier pointé. Si le chemin est suffisamment court, il est en réalité stocké dans le descripteur d'inode, en lieu et place des pointeurs sur les blocs de données. On parle alors de « lien symbolique rapide » (*Fast symlink*) (reconnaissable par un nombre de blocs de données égal à zéro). Il existe deux fonctions C pour manipuler les liens symboliques :

- **int ret = symlink(const char *src, const char *dst)** : permet la création d'un lien (POSIX) ;
- **ssize_t size = readlink(const char *name, char *buf, size_t bufsize)** : lecture du contenu du lien (BSD).

## 2.4. Périphériques

Une section de ce document est consacrée aux fichiers de type « périphérique » . Du point de vue du système de fichier EXT, il s'agit de fichiers d'un type particulier (bloc ou caractère), vides (taille nulle) et caractérisés par un numéro majeur et un numéro mineur stockés dans l'inode à la place du pointeur sur le premier bloc de données. La commande *shell* **mknod** permet à l'administrateur (*root*) de créer de tels fichiers.

### 2.4.1. Périphériques en mode bloc

Les périphériques en mode bloc sont créés de la façon suivante :

```
mknod <nom> b <maj> <min>
```

La commande **ls -l** leur attribue le caractère de type « b » :

```
bash$ ls -l /dev/hda
brw------- 1 root root 3, 0 jan 1 1970 /dev/hda
```

6. Si le nom de fichier est précédé par un chemin, absolu ou relatif, les éventuels liens symboliques (vers des répertoires) qu'il contient seront suivis.

Les deux numéros figurant après le propriétaire et le groupe sont, dans l'ordre, les numéros majeur et mineur. Les fonctions C spécifiques aux périphériques en mode bloc sont les **ioctl**[7] associés à chaque type de périphérique, ainsi que la fonction de création :

```
int ret = mknod(const char *name, S_IFBLK | mode_t mode, dev_t dev)
```

où l'on précise les permissions à appliquer au nouveau périphérique (modulo le umask).

*Remarque* : Une variable de type dev_t se construit via la macro **makedev(major, minor)** et s'analyse avec les macros major(dev) et minor(dev).

### 2.4.2. Périphériques en mode caractère

Les périphériques en mode caractère sont créés de la façon suivante :

```
mknod <nom> <maj> <min>
```

La commande **ls -l** leur attribue le caractère de type « c » :

```
bash$ ls -l /dev/null
rw-rw-rw- 1 root root 1, 3 jan 1 1970 /dev/null
```

Les deux numéros figurant après le propriétaire et le groupe sont, dans l'ordre, les numéros majeur et mineur. Les fonctions C spécifiques aux périphériques en mode caractère sont les **ioctl** associés à chaque type de périphérique, ainsi que la fonction de création :

```
int ret = mknod(const char *name, S_IFCHR | mode_t mode, dev_t dev)
```

où l'on précise les permissions à appliquer (modulo le umask).

## 2.5. Tubes nommés

Les tubes (nommés on non) sont présentés plus loin dans ce document. La commande *shell* **mkfifo** permet de créer un tube nommé :

```
mkfifo <nom>
```

La commande **ls -l** leur attribue le caractère de type « p » :

```
bash$ ls -l /var/run/xdmctl/xdmctl-:0
prw--w---- 1 prof root 0 oct 5 09:32 xdmctl-:0
```

L'ouverture d'un tube nommé en lecture ou en écriture est bloquante tant qu'il n'a pas été ouvert dans l'autre mode (sauf si O_NONBLOCK est précisé ; l'ouverture renvoie alors un code d'erreur ENXIO s'il s'agissait d'une tentative d'ouverture en écriture seule). La fonction C correspondante est tout simplement :

```
int ret = mkfifo(const char *pathname, mode_t mode)
```

où l'on précise les permissions à appliquer (modulo le umask).

7. Contrôleur de périphérique

### 2.6. Sockets

Un cours ultérieur sera dédié aux *sockets*. Comme les tubes, les *sockets* locales peuvent être associées à des fichiers. Il n'y a pas de commande *shell* pour créer ou exploiter de tels fichiers. **ls -l** utilise la lettre « s » pour désigner les fichiers de type *socket* nommée :

```
bash$ ls -l /tmp/.X11-unix/X0
srwxrwxrwx 1 root root 0 oct 5 09:33 /tmp/.X11-unix/X0
```

## 3. Opérations spéciales sur les fichiers

### 3.1. Accès aux méta-données

Il existe trois fonctions permettant d'obtenir des méta-informations sur un fichier donné. Il s'agit essentiellement des informations stockées au niveau de l'inode EXT.

- **int ret = stat(const char *name, struct stat *buf)** qui suit les liens symboliques (POSIX) ;
- **int ret = fstat(int fd, struct stat *buf)** similaire à stat mais fonctionnant sur un descripteur de fichier ouvert (POSIX) ;
- **int ret = lstat(const char *name, struct stat *buf)** variante de stat qui, si la cible est un lien symbolique, s'applique au lien plutôt qu'au fichier vers lequel il pointe (BSD+SYSV).

La page de manuel décrit la structure de la réponse (struct stat).

*Remarque* : Les attributs spéciaux (fichier non modifiable, etc...), dont l'existence et la nature dépendent de l'OS, ne sont pas accessibles par ce moyen (Sous Linux, on utilisera **ioctl**, voir plus bas).

Il existe également une fonction **access** permettant de tester les modes d'ouverture autorisés sur un fichier en utilisant les uid et gid réels (plutôt qu'effectifs) du processus appelant. Elle permet à un programme avec bit setuid de déterminer si l'utilisateur qui l'a lancé a ou non le droit d'accéder à ce fichier. Cependant, son utilisation pour la sécurité (vérification des privilèges de l'utilisateur réel par **access** puis ouverture par **open**) est dangereuse, un attaquant pouvant substituer le fichier entre les deux appels (*race condition*) : mieux vaut utiliser **seteuid** et **open**.

Pour changer les propriétés des inodes, on peut utiliser :

- **chmod**, **chown** et **utime**, qui travaillent sur des noms de fichiers (POSIX) ;
- **lchown**, variante de **chown** qui s'applique aux liens symboliques sans les suivre (Linux) ;
- **fchmod** et **fchown**, qui travaillent sur des descripteurs de fichiers ouverts (BSD, SYSV).

Rappel : Le contrôle d'accès ayant lieu à l'ouverture d'un fichier, les changements de permissions ou de propriétaire n'empêchent pas un processus ayant déjà fait **open** de continuer à accéder à ce fichier selon les modes d'accès obtenus. Il est donc important de créer directement le fichier avec les bons droits. À défaut, il reste possible de le supprimer et d'en recréer un nouveau avec des droits différents (si un processus conserve une référence sur l'ancien inode, il ne sera pas libéré, donc le nouveau prendra bien un numéro différent).

### 3.2. Verrous de fichiers

Lorsqu'elle est nécessaire, la synchronisation des accès aux ressources de type lecteur / écrivain doit généralement être gérée explicitement par le programme utilisant ces ressources. Dans le cas des fichiers, l'OS met à la disposition du développeur des fonctions permettant de gérer des verrous de façon fiable. Ces fonctions peuvent également être utilisées pour synchroniser les accès aux ressources de type producteur / consommateur si l'on souhaite une granularité différente de celle offerte par défaut (ex : pour garantir qu'un consommateur récupèrera deux productions consécutives). Comme tout mécanisme de verrouillage, on prendra garde à éviter les cas d'interblocage lorsqu'on y fait appel (cf. cours sur la synchronisation). Il existe deux API simples pour manipuler des verrous sur les fichiers en C (la prise de verrou est par défaut une opération bloquante si le verrou n'est pas disponible) :

- **int ret = flock(int fd, LOCK_SH / LOCK_EX / LOCK_UN [| LOCK_NB])** pour obtenir un verrou en lecture / en écriture / relâcher un verrou ; LOCK_NB permet de rendre l'appel non bloquant dans le cas où le verrou n'est pas disponible (BSD) ;
- int ret = lockf(int fd, F_LOCK / F_ULOCK / F_TLOCK, 0) pour obtenir un verrou exclusif / le relâcher / tenter de l'obtenir de façon non bloquante à partir de la position courante dans le fichier (SYSV).

Ces verrous portant sur le fichier, et non sur le descripteur, on a la garantie que deux programmes indépendants couvrant chacun le fichier de leur côté ne pourront pas détenir simultanément un verrou exclusif sur ce fichier. **lockf** est typiquement une surcouche des verrous POSIX gérés par **fcntl** (cf. plus bas). Les verrous POSIX sont les plus riches (possibilité de verrouiller une partie d'un fichier, de savoir qui possède un verrou, etc...) mais l'API est plus complexe.

Moralement, les verrous BSD sont détenus par la structure décrivant les propriétés de l'accès à un fichier, alors que les verrous POSIX (ou SYSV) sont détenus par le processus qui les a posés. Concrètement, cela signifie :

- qu'un verrou BSD peut être relâché à travers n'importe quel *file descriptor* lié (par dup/dup2 ou fork) à celui utilisé pour le poser - il sera d'ailleurs automatiquement relâché lorsque le dernier tel descripteur de fichier sera fermé (**close**, ou terminaison du processus) ;
- qu'un verrou POSIX ou SYSV peut être relâché par le processus qui l'a posé à travers n'importe quel fd associé au même fichier - il sera d'ailleurs automatiquement relâché lorsque le premier tel descripteur de fichier sera fermé.

On s'attachera néanmoins à relâcher explicitement les verrous obtenus, y compris dans la gestion des cas d'erreur.

Il est déconseillé d'utiliser simultanément les différentes API sur un même fichier, le résultat dépendant de la façon dont l'OS gère les différents types de verrous (séparément, ou à travers un même verrou sous-jacent).

Le principe de verrouillage décrit plus haut est de nature coopérative, dans la mesure où les différents acteurs souhaitant synchroniser leurs accès respectifs aux fichiers concernés utilisent les verrous de leur plein gré. Rien n'empêche un programme concurrent d'accéder à un fichier verrouillé s'il ne demande pas le verrou. De même, il n'y a aucun contrôle de cohérence entre la nature d'un verrou pris (exclusif ou partagé) et les opérations (lecture ou écriture) qui sont réalisées ensuite. SYSV a défini un mécanisme de verrou obligatoire permettant de bloquer les accès classiques (par read, write etc...) par d'autres tâches lorsqu'un verrou a été posé. Ce mécanisme consiste à marquer le fichier

comme relevant d'un verrou obligatoire en activant le bit setgid et en désactivant le bit exécutabilité pour le groupe (!). Les verrous classiques SYSV (**lockf**) / POSIX (**fcntlx**) posés sur ce fichier deviennent alors des verrous obligatoires. Ce mécanisme est dangereux du point de vue des effets de bord potentiels et peut entraîner le blocage durable de certains fichiers critiques. Par ailleurs, les effets exacts des verrous obligatoires dépendent des versions d'UNIX qui les implémentent... Sous Linux, les verrous posés sur les fichiers dont les permissions indiquent qu'ils relèvent d'un verrou obligatoire ne sont effectivement obligatoires que si leur partition d'appartenance a été montée avec l'option **mand** (ce n'est pas le cas par défaut).

*Remarque* : Plusieurs programmes utilisent des « fichiers-verrous » pour éviter les accès concurrents à certaines ressources. Ce mécanisme consiste à invoquer **open** sur un fichier particulier (le verrou) avec les options O_CREAT et 0_EXCL : cet appel échoue si, et seulement si, le fichier existe déjà, c'est-à-dire si le verrou est posé. Le verrou se relâche par suppression du fichier. Parmi les défauts de cette approche, on peut citer le caractère non bloquant de la tentative de prise de verrou (comment gérer proprement l'attente ?), les restrictions liées aux permissions sur les divers répertoires, le risque de multiplication des fichiers-verrous dans le FS et la rémanence du verrou en cas d'interruption accidentelle du programme ayant posé le verrou...

*NB* : Sous Linux, cat /proc/locks montre les verrous de fichiers détenus par les divers processus. Les enregistrements adoptent le format suivant :

- <numéro> : // référence du verrou
- POSIX/FLOCK ADVISORY/MANDATORY // type de verrou
- READ/WRITE <pid> // état et possesseur
- <majeur> :<mineur> :<n_inode> // device et inode concernés
- <début>/0 <fin>/EOF // section verrouillée

### 3.3. Duplication de fd

Il existe des fonctions POSIX de duplication des descripteurs de fichiers. La copie ainsi créée partage avec l'original la structure système décrivant les propriétés de l'accès au fichier (en particulier, les déplacements par **lseek** sont liés, cf. cours précédent) mais la fermeture de l'un n'entraîne pas celle de l'autre (simple décrémentation du nombre de références à la structure) :

- **int fd2 = dup(int fd)** (utile si la suite du programme prévoit la fermeture de fd, notamment du fait d'un dup2) ;
- **int fd2 = dup2(int fd, int fd2)** ferme fd2 s'il est déjà attribué et le remplace par une copie de fd.

dup2 joue un rôle clé dans les redirections puisqu'il permet de remplacer les descripteurs standard 0, 1 et 2 par des descripteurs renvoyant vers des fichiers, des tubes ou même des *sockets*.

### 3.4. fcntl

Cette fonction permet d'effectuer des actions particulières sur les descripteurs de fichiers (application d'un verrou, configuration des signaux pour la signalisation asynchrone des entrées/sorties, paramétrage de la transmissibilité à travers l'appel **exec**, etc...). Certaines de ces fonctionnalités sont couvertes par d'autres fonctions, mais **fcntl** offre souvent un contrôle plus fin. La portabilité de **fcntl** est à examiner au cas par cas, selon les types de paramètres qui lui sont passés.

### ioctl

L'appel système **ioctl** (*input output control*) est une fonction permettant d'effectuer des opérations spécifiques sur un fichier spécial (périphériques en mode bloc ou en mode caractère, *sockets*). Pour chaque type de périphériques, les opérations que l'on peut réaliser seront différentes. L'utilisation de **ioctl** sur une *socket* permet par exemple de modifier l'état d'une carte réseau ; à partir d'un descripteur de fichier associé à un lecteur CDROM, on peut provoquer l'ouverture du tiroir ; et ainsi de suite. Sous Linux, voir **man ioctl_list** pour une liste pas toujours exhaustive des possibilités offertes. La portabilité des appels à **ioctl** est variable selon les paramètres, mais il ne faut pas trop en attendre de ce point de vue.

## 4. Les tubes (*pipes*)

Les tubes constituent un exemple parfait de l'application de la logique producteur / consommateur. Sous Linux, la taille de la ressource associée à un tube est typiquement d'une page mémoire, soit 4Ko[8]. Les lectures sont bloquantes lorsque le tube est vide et les écritures deviennent bloquantes si le tube est plein. Il existe deux types de tubes :

- les tubes nommés, qui sont associés à un fichier de VFS (le FS sous-jacent doit supporter ce type de fichier) ;
- les tubes anonymes, également manipulés via des descripteurs de fichiers mais qui ne sont pas rattachés à VFS.

Le tube nommé est plus lourd en ce sens qu'il nécessite la création préalable et la gestion d'un fichier réel. L'avantage est qu'il peut être partagé par des entités indépendantes pourvu qu'elles connaissent le chemin d'accès et aient les bonnes permissions. La fonction C permettant de créer un tel tube est la suivante :

```
int ret = mkfifo(const char *name, mode_t mode)
```

permettra la création d'un tube nommé dans le système de fichiers en précisant les permissions à appliquer modulo le umask (POSIX).

Une fois créé, le tube nommé se manipule comme un fichier classique (**open**, **read**, **write**, **close**) mais selon une logique producteur / consommateur.

NB : En tant que fichier du FS, le tube nommé est toujours vide. Son contenu est stocké en mémoire système (allocation lors de la première ouverture), et il est détruit lorsque son dernier utilisateur a fait **close** (pas de persistance du contenu). Le fichier lui-même ne sert donc que de « point de rencontre » et de support pour la gestion des permissions d'accès.

Le tube anonyme est plus maniable mais ne peut essentiellement[9] être partagé qu'entre descendants de son créateur (cf. fork et le cours suivant) pour des questions de transmission des entrées correspondantes dans la table des descripteurs de fichiers. La création d'un tel tube s'effectue en C par l'appel suivant :

8. Depuis le noyau 2.6.11, cette taille est ajustée dynamiquement en fonction des besoins des producteurs, et peut croître jusqu'à un maximum de 16 pages mémoire (64Ko). Lorsque toutes ces pages sont remplies, les opérations de production deviennent bloquantes.

9. Les deux exceptions sont le transfert de descripteurs de fichiers par *sockets* UNIX (cf. cours sur les *sockets*) et l'existence d'un nommage indirect via le /proc/<pid>/fd/ sous Linux (cf. cours sur les processus).

```
int ret = pipe(int fd[2])
```

Création d'un tube anonyme en couche noyau et récupération d'un descripteur pour la consommation (fd[0]) et d'un descripteur pour la production (fd[1]) (POSIX).

Les producteurs agissent en appelant **write** sur fd[1] et les consommateurs en appelant **read** sur fd[0]. La fermeture des extrémités s'effectue simplement par **close** et le tube est détruit quand plus personne ne conserve d'extrémité ouverte. Enfin, la norme POSIX définit deux fonctions de haut niveau utilisant les tubes anonymes :

- **FILE *f = popen(const char *cmd, "r"/"w")** : exécution d'une ligne de commande via /bin/sh et redirection de son entrée standard ("w") ou de sa sortie standard ("r") dans un tube anonyme dont on récupère l'autre extrémité ;
- **int ret = pclose(FILE *f)** : fermeture du tube et attente de la terminaison de la commande.

Ces appels peuvent s'avérer pratiques mais leur utilisation est déconseillée dans la mesure où elle repose sur l'invocation d'un *shell* externe. On peut obtenir le même résultat de façon plus satisfaisante (en faisant l'économie du *shell* et des risques d'injection de commandes indésirables dans la ligne de commande) avec **pipe**, **fork** et **exec** (cf. cours suivant).

## 5. Périphériques

Les systèmes UNIX offrent un accès banalisé à la plupart des périphériques réels (ex : disque) ou virtuels (ex : générateur de pseudo-aléa) sous la forme de fichiers spéciaux [10]. De cette manière, un programme utilisateur souhaitant accéder au périphérique peut utiliser les fonctions d'accès classiques (**open**, **read**, **write**, **close**, éventuellement **lseek**) pour réaliser les opérations élémentaires ainsi que des **ioctl** pour les opérations spécifiques au périphérique. Ainsi, l'accès bas niveau à un disque se réalise aussi simplement que l'accès à un fichier régulier.

En réalité, le traitement de ces fonctions n'est pas classique, et le noyau répond à ces requêtes au cas par cas. L'accès aux fichiers de type périphérique est donc à voir comme une communication programme / OS, le « contenu apparent » de ces fichiers dépendant de ce que l'OS souhaite montrer. La correspondance entre un fichier de type périphérique et les fonctions de traitement dont dispose l'OS se fait non pas sur le nom du fichier [11] mais sur son type (bloc ou caractère), ses numéros majeur (qui désigne généralement le pilote concerné) et mineur (qui identifie précisément les fonctions à appliquer parmi celles du pilote). Les numéros majeur et mineur forment un couple d'entiers, les différentes combinaisons envisageables ne correspondant pas toutes à des pilotes disponibles ou existants [12]. Si le nom du fichier n'est pas déterminant du point de vue du système, l'usage veut que l'on respecte un certain nommage et qu'on les regroupe dans le répertoire /dev de telle sorte que les applications sachent où les trouver. Les noms utilisés ainsi que les numéros diffèrent cependant d'un type d'UNIX à l'autre.

L'option de montage **nodev** permet d'ignorer les éventuels périphériques liés dans le système de fichiers concerné. Elle est souvent utilisée conjointement avec l'option **nosuid** (cf. cours précédent).

10. Ce n'est pas le cas des périphériques réseau, qui sont exploités sous la forme d'« interfaces réseau » accessibles via les *sockets* (cf. cours sur les *sockets*).
11. Ce serait le cas pour un fichier régulier au sein d'un FS virtuel comme celui du /proc.
12. Sous Linux, la liste des numéros réservés et des noms des périphériques associés est fournie avec le code source du noyau, dans le fichier */usr/src/linux/Documentation/devices.txt*.

*Remarque* (historique) : Le noyau Linux disposait d'un système de fichiers virtuel (aujourd'hui obsolète), devfs, montable sur /dev. Ce système contenait les fichiers de type périphérique associés à tous les périphériques effectivement reconnus par le système à un instant donné, son contenu étant mis à jour automatiquement lors de l'ajout ou de la suppression d'un périphérique matériel (ou du pilote correspondant). Son utilisation pouvait, selon les configurations, entraîner des instabilités.

### 5.1. Périphériques en mode bloc

Les périphériques en mode bloc sont par définition les périphériques auxquels le noyau accède (lectures / écritures) par blocs entiers de données, en gérant des caches [13] pour les différents blocs en cours de manipulation.
Dans la pratique, ils sont à voir comme des ressources étendues (ex : un disque), virtuelles ou non, auxquelles le noyau donne accès via le fichier correspondant. Ce fichier, vide (au sens du FS), n'héberge pas réellement la ressource. De par sa nature, un périphérique en mode bloc est compatible avec la fonction **lseek**. La ressource associée peut généralement être formatée et contenir un FS, montable dans le VFS. Exemples de périphériques en mode bloc (Linux) :

- **/dev/ram*** : disques virtuels en RAM ;
- **/dev/hd*** : périphériques IDE (disques durs, cdroms) et leurs éventuelles partitions ;
- **/dev/loop*** : périphériques loop (voir plus bas) ;
- **/dev/sd*** : disques durs SCSI ou assimilés (SATA, USB, etc...) ;
- **/dev/scd* (ou /dev/sr*)** : cdroms SCSI (ou assimilés) ;
- **/dev/mapper/*** (sauf control) : périphériques gérés par le *device-mapper* (cf. **man dmsetup**).

Leur validité dépend de la présence du matériel correspondant et des pilotes de périphériques présents en mémoire. Les périphériques loop permettent de voir des fichiers normaux ou d'autres périphériques en mode bloc (éventuellement après transformation) comme des périphériques en mode bloc, donc des disques. L'association d'un fichier à un périphérique loop se fait par la commande :

```
losetup [-e <algo>] /dev/loop<n> <fichier>
```

L'association est directe ou traverse une couche de chiffrement à la volée (algorithme précisé par l'option -e). La désassociation du périphérique s'effectue par losetup -d /dev loop<n>. L'option -f permet de trouver (**find**) un périphérique loop inutilisé. Les opérations sur un /dev/loop associé sont similaires à celles sur /dev/hda1 (montage / démontage dans VFS, formatage, etc...). On peut ainsi examiner la copie bas niveau d'un CDROM stocké dans un fichier régulier :

```
$ losetup -f
/dev/loop0
$ losetup /dev/loop0 cd_img.iso
$ mount /dev/loop0 /mnt/disk
$ [...]
$ umount /mnt/disk
$ losetup -d /dev/loop0
```

13. Dans le cas d'une partition montée, il n'y a aucune garantie de cohérence entre le cache VFS et le cache bloc utilisé en cas d'accès bas niveau à la partition. Ces deux caches sont structurés en blocs et peuvent directement être synchronisés avec le disque correspondant. Il est donc très fortement déconseillé d'accéder en écriture à une partition montée, surtout si elle est montée en lecture/écriture.

*Remarque* : **losetup** réalise les associations demandées par des ioctl sur les /dev/loop. Quelques éléments supplémentaires sur **losetup** :

- l'option -o permet d'ignorer les premiers octets du fichier et de faire commencer /dev/loopX uniquement à partir de l'offset indiqué ;
- une méthode plus rapide de réaliser le montage d'une image ISO d'un CDROM est d'utiliser l'option loop de mount, par exemple **mount -o loop cd_img.iso /dev/loop0** ;
- l'utilisation de **losetup** pour réaliser du chiffrement de partition est aujourd'hui déconseillé, au profit des méthodes utilisant le *device mapper*.

Les périphériques **loop** sont spécifiques à Linux. Voir aussi **dmsetup** et **cryptsetup** (Linux, plus récent), **vnconfig** (*BSD), **mdconfig** (FreeBSD), **gbde** (FreeBSD), **gdconfig** (NetBSD).

### 5.2. Périphériques en mode caractère

Les périphériques en mode caractère sont par définition les périphériques auxquels le noyau accède caractère par caractère, sans gérer de cache. Dans la plupart des cas, ils peuvent être vus comme des ressources accessibles selon une logique producteur / consommateur, une des extrémités étant détenue par le noyau de l'OS et l'autre par le programme utilisateur de la ressource. Si le périphérique est accessible en lecture et en écriture, il faut le voir comme deux ressources indépendantes, le noyau jouant le rôle de producteur pour les accès en lecture à la première ressource et de consommateur pour les accès en écriture à la seconde. En particulier, une application ne pourra pas consommer dans un périphérique en mode caractère ce qu'elle-même ou une autre application y a produit (ce n'est pas un tube !). Les ressources triviales (null, zero, etc...) peuvent être considérées comme infinies et jamais bloquées, le noyau étant en mesure de produire ou de consommer au rythme des accès au périphérique. Pour les ressources plus complexes (génération d'aléa, etc...) ou associées à des entités physiques (carte son, console, etc...), des limitations en bande passante ou en disponibilité des données à lire peuvent rendre bloquants les accès en lecture ou en écriture. À noter que le noyau ne se bloque jamais sur la ressource, même s'il n'y a pas de producteur ou de consommateur côté applicatif.
De par sa nature, un périphérique en mode caractère ne supporte généralement pas la fonction **lseek** (exception : les périphériques comme /dev/zero qui peuvent l'ignorer silencieusement).
Exemples de périphériques en mode caractère (Linux) :

- **/dev/null** : puits sans fond pour l'écriture, vide pour la lecture ;
- **/dev/zero** : producteur infini de caractères nuls ;
- **/dev/random** : producteur infini d'aléa (lent, moyenne qualité)[14] et consommateur de graines d'aléa ;
- **/dev/urandom** : producteur infini d'aléa (rapide, basse qualité) et consommateur de graines d'aléa ;
- **/dev/pty[p-za-e]*** : pseudo-terminaux maîtres (BSD) ;
- **/dev/tty[p-za-e]*** : pseudo-terminaux esclaves (BSD) ;
- **/dev/tty0** : la console virtuelle (tty) active[15] ;

14. Sur d'autres UNIX, /dev/random représente un générateur physique d'aléa, et /dev/srandom un générateur logiciel d'aléa de moyenne qualité.
15. Un descripteur de fichier obtenu sur /dev/tty0 ne suit pas les éventuels changements ultérieurs de console active. Les écritures sur /dev/tty0 peuvent par ailleurs être redirigées vers un terminal particulier au moyen d'un **ioctl**.

- **/dev/tty*** : consoles virtuelles en mode texte ;
- **/dev/ttyS*** : ports série ;
- **/dev/tty** : le terminal de contrôle de l'application [16] ;
- **/dev/console** : la console système [17] ;
- **/dev/lp*** : ports parallèle ;
- **/dev/input/mouse*** : pointeurs (souris) reconnus par le système ;
- **/dev/rtc** : horloge temps réel ;
- **/dev/agpgart** : table de traduction d'adresses AGP (carte graphique) ;
- **/dev/dsp** : carte son ;
- **/dev/ptmx** : pseudo-terminal maître (System V / Unix98, multiplexé) ;
- **/dev/pts/*** : pseudo-terminaux esclaves actifs (System V / Unix98).

Les pseudo-terminaux permettent aux applications graphiques (xterm) ou réseau (ssh) de gérer des sessions à la manière d'une console virtuelle en mode texte. L'utilisateur agit sur l'application côté maître, qui exploite son périphérique (/dev/ptmx en Unix98 ou /dev/ptyp* en BSD) pour envoyer des caractères et lire les réactions de l'application côté esclave (/dev/pts/* ou /dev/ttyp*), typiquement un *shell*. Le couple (maître, esclave) se comporte essentiellement comme un tube bi-directionnel, qui émule des fonctionnalités de terminal côté esclave (gestion des tâches, contrôle du débit, gestion de l'écho, ioctls, etc..). Linux dispose d'un système de fichiers virtuel, **devpts**, généralement monté sur /dev/pts. Ce système contient la liste, mise à jour dynamiquement, des pseudo-terminaux esclaves Unix98 actifs.

Il existe aussi quelques périphériques en mode caractère qui sont associés à des ressources accessibles selon la logique lecteur / écrivain (et supportent donc **lseek**). Ils ne sont pas en mode bloc car on souhaite pouvoir y accéder sans passer par un cache. En voici quelques exemples (Linux) :

- **/dev/mem** : mémoire physique (cf. cours sur la mémoire) ;
- **/dev/kmem** : mémoire noyau (cf. cours sur la mémoire) ;

*Remarque* : Dans les anciennes versions de Linux, les périphériques /dev/raw/raw* permettent d'accéder aux ressources de périphériques en mode bloc en court-circuitant le cache noyau [18]. L'association d'un périphérique en mode bloc à un périphérique *raw* se faisait par la commande :

```
raw /dev/raw/raw<n> /dev/<bloc_dev>
```

Sur les systèmes Linux modernes, on obtient le même comportement en ouvrant le périphérique en mode bloc avec l'option O_DIRECT. Pour préserver la cohérence du périphérique, il faut évidemment éviter d'y accéder autrement qu'en lecture seule s'il est utilisé par ailleurs (partition montée, etc...).

16. Un descripteur de fichier obtenu sur /dev/tty ne suit pas les éventuels changements ultérieurs de terminal de contrôle.
17. Il s'agit par défaut de /dev/tty0.
18. L'accès direct à un disque nécessite que l'application aligne et dimensionne correctement ses lectures et ses écritures, de telle sorte que le contrôleur disque puisse effectuer les transferts correspondants directement en mémoire applicative.

# 6. Le système de fichiers EXTx

L'**extended file system** ou EXT, est le premier système de fichiers créé en avril 1992 spécifiquement pour le système d'exploitation Linux. Il est aujourd'hui en version 4 (EXT4).

## 6.1. Organisation physique

Le système de fichiers EXT est organisé en blocs de taille paramétrable, typiquement 4096 octets (4 Ko), eux-mêmes rassemblés en groupes de blocs. Chaque groupe a la charge d'un certain nombre d'inodes (ou fichiers). Ainsi, si on appelle $n$ le nombre d'inodes par groupe, les inodes 1 à $n$ sont décrits dans le groupe 1, les inodes $n+1$ à $2n$ dans le groupe 2, etc... Le début du FS contient :

- une zone de boot réservée de 1Ko (au début du bloc 0) ;
- un super-bloc de 1Ko qui contient les paramètres caractéristiques du FS :

  taille des blocs,

  nombre de blocs par groupe,

  nombre d'inodes par groupe,

  numéro du bloc dans lequel il se trouve (0 si la taille des blocs est au moins de 2Ko ou 1 si elle est de 1Ko),

  etc...

  un champ « état » indiquant si le FS est « propre » ou pas ; ce champ, qui est modifié à chaque opération de montage / démontage (sauf pour un montage en lecture seule), permet au système de lancer le réparateur de FS (équivalent du ScanDisk sous Windows) en cas de redémarrage après arrêt brutal de la machine.

*Remarque* : Dans EXT3, le champ « état » est toujours marqué « propre » , et le noyau s'appuie en fait sur le flag « recover » lié au journal. C'est une des raisons pour lesquelles il faut éviter de monter un FS EXT3 en EXT2.

Le bloc suivant contient une table de descripteurs de groupe (1 descripteur par groupe) indiquant, pour chaque groupe, les numéros de blocs où se trouvent les méta-données relatives au groupe (voir plus bas). Les informations du super-bloc et de la table de descripteurs de groupe sont reproduites dans les premiers blocs de chaque groupe (à des différences mineures près), de façon à faciliter la réparation du FS en cas d'incident sur cette zone critique.

Dans chaque groupe, les blocs suivants contiennent :

- une carte des blocs alloués et disponibles pour le groupe courant ;
- une carte des numéros d'inodes alloués et disponibles pour le groupe courant ;
- la table des inodes des fichiers du groupe ;
- des blocs de données brutes, associés ou non à des inodes.

Ces données et méta-données sont spécifiques à chaque groupe.

## 6.2. Inodes

Les fichiers (ou répertoires) appartenant au FS sont représentés sous la forme d'inodes. Les inodes sont composés de méta-données relatives au fichier, notamment :

- l'identifiant (uid) du propriétaire ;

- l'identifiant (gid) du groupe d'appartenance ;
- des attributs normaux (type de fichier et permissions [19]) ;
- des attributs spéciaux (*flags* : fichier non modifiable, etc...), rarement utilisés, cf. **man chattr** et **man lsattr** sous Linux (les équivalents BSD sont **chflags** et **ls -lo**) ;
- les éléments nécessaires au support des listes de contrôle d'accès (ACL, non gérées par défaut sous Linux) et autres attributs étendus ;
- un compteur de liens (nombre de fois où cet inode est référencée) ; l'inode n'est libérée que lorsque ce compteur passe à 0 ;
- les dates de dernier changement d'attribut (**time** [20]), de dernière modification de contenu (**mtime**), de dernier accès (**atime**) et de suppression (**dtime**) ;
- une taille (en octets) ;
- un nombre de « blocs » occupés (comptés en blocs de 512 octets) ;
- une table des numéros des blocs de données où est stocké le contenu du fichier.

Il est intéressant de noter que le nom par lequel on accède à un fichier n'est pas défini au niveau de son inode mais à celui du répertoire auquel il est lié.

Afin de ne pas trop limiter la taille des fichiers supportés par le FS (et de ne pas surdimensionner la taille des structures inodes), les trois dernières entrées de la table des numéros de blocs [21] ne pointent pas directement vers des blocs de données, mais respectivement vers :

- un bloc contenant la suite de la table (simple indentation) ;
- un bloc contenant une table de numéros de blocs où est répartie la suite de la table (double indentation) ;
- un bloc contenant une table de numéros de blocs contenant des tables où sont stockés les numéros de blocs où se poursuit la table (triple indentation).

Les accès aux fins de gros fichiers sont par conséquent légèrement plus lents que les accès aux débuts de fichiers. Des mécanismes de cache, internes à l'OS, effacent cette différence.

### 6.3 Arborescence et types de fichiers EXT

L'arborescence du FS est construite à partir d'une origine fixe (inode EXT_ROOT_INO (=2)), qui est de type répertoire [22]. Un fichier au sens large est composé de données (contenu) et de méta-données (inode), mais n'a pas de nom intrinsèque (le nom sera déterminé par le répertoire d'accroche). Le contenu d'un fichier de type répertoire est un tableau de références de type numéro d'inode, paramètres, nom , chaque référence étant relative aux fichiers et sous-répertoires (dont au moins . et ..) inclus dans ce répertoire.

19. détaillées.
20. parfois appelée, à tort, date de création.
21. La table des numéros de blocs peut également contenir des 0, qui représentent des "trous" dans le fichier. Un trou peut apparaître si le fichier est généré par **dd** avec un **seek** ou plus généralement en utilisant la fonction **[f]truncate** pour augmenter sa taille. La lecture dans un trou renvoie des octets nuls et l'écriture provoque l'allocation d'un bloc dans le FS qui vient boucher le trou.
22. Cette racine peut devenir un sous-répertoire en mémoire via le montage VFS.

Notion de lien dur : si deux répertoires[23] ont en entrée un même numéro d'inode, ils partagent un même fichier. Chronologiquement, le fichier a été d'abord créé dans un des deux répertoires, puis un lien dur a été établi au niveau du deuxième répertoire. Cependant, la situation est devenue symétrique, le fichier n'appartient pas « plus » à un répertoire qu'à l'autre, et il n'y a pas de différence objective entre un lien dur et un fichier. Comme l'inode n'est pas dupliquée, les attributs (permissions et autres) sont communs à tous les liens vers un même fichier.

*Important* : Le couple (n° de partition, n° d'inode) désigne un fichier de façon unique et non ambiguë.

Notion de lien symbolique : le système EXT gère un type de fichier particulier, le lien symbolique. Un tel fichier contient le nom d'un autre fichier, précédé d'un chemin absolu ou relatif[24] interprétable au niveau de VFS. Sauf indication contraire, l'opération d'accès à un fichier de type lien symbolique est automatiquement redirigée vers le fichier pointé par ce lien.
NB : Le fichier pointé peut ne pas exister.

Il existe, en plus des fichiers classiques, des répertoires et des liens symboliques, d'autres types de fichiers (voir le cours n°3).

**Remarque** : Il y a une limite dans le nombre de liens symboliques successifs que l'OS accepte de suivre pour trouver un fichier réel.

### 6.4. Manipulation en *Shell*

Plusieurs commandes shell permettent de manipuler les systèmes de fichiers EXT (voir les pages de manuel respectives pour les nombreuses options supportées). **Attention**, il est fortement déconseillé de tenter de modifier un système de fichiers monté ! Pour modifier le FS racine de VFS, démarrer sur une clé USB ou un CD-ROM (*live* CD).

- **ls -i** : lister les numéros d'inode ;
- **stat -f <fichier>** : voir les informations sur le FS auquel appartient un fichier ;
- **mke2fs** : créer un nouveau système de fichiers (formatage) ;
- **tune2fs** : changer les paramètres d'un système de fichiers existant ;
- **e2fsck** : réparer un système de fichiers endommagé (utiliser -f pour forcer la vérification si le FS est marqué "propre") ;
- **dumpe2fs** : sauver les informations critiques du FS ;
- **debugfs** : analyser un système de fichiers (et restaurer des fichiers supprimés, etc...).

### 6.5. Autres systèmes de fichiers UNIX

Les concepts explicités pour EXT sont applicables à la plupart des systèmes de fichiers UNIX (notions d'inodes, de liens, de blocs de données, etc...). En particulier, le système EXT3 est un système EXT2 augmenté d'une inode supplémentaire, représentant un fichier journal, dont le rôle est de faciliter la restauration du système de fichiers en cas de panne de type arrêt sur coupure de courant. Ces deux types de FS sont compatibles.

23. Un fichier peut également être lié plusieurs fois dans un même répertoire, sous des noms différents.
24. L'absence de chemin équivaut à "./".

*Ce document est inspiré du « Cours Système n°3 : les Fichiers (2/2) » délivré par monsieur Olivier Levillain dans les cours dispensés à l'Agence nationale de la sécurité des systèmes d'information (Octobre 2012).*