# Systèmes d'exploitation

Gestion des entrées-sorties

**Pilot Systems** - Gaël Le Mignot

INSIA SRT - 2007

# Table des matières

**1 Gestion des périphériques** **3**
1.1 Principes de la gestion des périphériques . . . 3
1.1.1 Périphériques, contrôleurs et pilotes . . . 3
1.1.2 Types de périphériques . . . 3
1.1.3 PIO, MMIO et DMA . . . 4
1.1.4 Les interruptions . . . 5
1.2 Les couches logicielles . . . 6
1.2.1 Architecture générale d'un pilote . . . 6
1.2.2 La gestion des interruptions . . . 6
1.2.3 Les copies et les tampons . . . 6
1.2.4 La gestion des erreurs . . . 6
1.3 Quelques exemples . . . 7
1.3.1 Les disques . . . 7
1.3.2 Les horloges . . . 9
1.3.3 Les terminaux . . . 9

**2 Systèmes de fichiers** **10**
2.1 Les fichiers . . . 10
2.1.1 Principes des fichiers . . . 10
2.1.2 Types de fichiers . . . 10
2.1.3 Attributs des fichiers . . . 10
2.1.4 Primitives d'accès aux fichiers . . . 11
2.1.5 Implémentation des fichiers . . . 12
2.2 Les répertoires . . . 13
2.2.1 Principes des répertoires . . . 13
2.2.2 Répertoires récursifs et chemins . . . 13
2.2.3 Liens physiques et symboliques . . . 13
2.2.4 Principe des points de montage . . . 14
2.2.5 Implémentation des répertoires . . . 14
2.3 Implémentation des systèmes de fichiers . . . 14
2.3.1 Architecture générale . . . 14
2.3.2 Gestion des blocs libres . . . 14
2.3.3 Problématiques de performances . . . 15
2.3.4 Problématiques de fiabilité . . . 15
2.4 Quelques exemples . . . 15
2.4.1 ISO9660 . . . 15
2.4.2 FAT . . . 16
2.4.3 Ext-2 . . . 17
2.5 Les systèmes de fichiers journalisés . . . 17
2.5.1 Principes du journal . . . 17
2.5.2 Ext-3 . . . 17
2.5.3 Systèmes de fichiers organisés autour du journal . . . 18

# Table des figures

1.1 Fonctionnement du RAID . . . . . 8
1.2 Comparaison des RAIDs . . . . . 9

# Chapitre 1

# Gestion des périphériques

## 1.1 Principes de la gestion des périphériques

### 1.1.1 Périphériques, contrôleurs et pilotes

Comme nous l'avons rapidement vu en introduction, les périphériques sont souvent complexes à utiliser (vitesse de rotation, position des têtes, gestion de l'espacement entre secteurs, ...). Les détails exacts sont en plus souvent dépendant du modèle et du fabriquant.

Afin de pallier à ces problèmes, il y a en général trois niveaux pour la gestion d'un périphérique :

1. Le périphérique lui-même (par exemple, un disque dur), qui effectue son traitement et reconnaît un protocole d'assez bas niveau (par exemple, « mettre en marche le moteur », « déplacer la tête de trois pistes »).
2. Le contrôleur, souvent sous forme d'une carte d'extension, parfois intégré directement dans le périphérique, qui reconnaît un protocole de niveau légèrement plus élevé (par exemple, « lire le secteur numéro 4 de la piste numéro 3 »).
3. Le pilote de périphérique, dans le système d'exploitation, qui lui reconnaît un protocole de niveau encore plus élevé (par exemple, « lire le secteur numéro 124 », sans se soucier de la géométrie du disque, ni du fait qu'il s'agisse d'un disque dur ou d'une clé USB).

Le découpage exact peut varier, et il y a parfois deux contrôleurs, un intégré au périphérique et un externe (par exemple dans le cas des disques durs, il y a un contrôleur IDE/SATA/SCSI dans la carte mère ou en carte d'extension, et un contrôleur à l'intérieur même du disque, les deux étant de niveau différent).

### 1.1.2 Types de périphériques

L'interaction avec un périphérique (sauf exceptions) consiste à envoyer ou recevoir des données, souvent accompagnées de méta-données (position dans le disque, fréquence d'échantillonnage, ...).

Les périphériques sont en général classifiés en deux catégories, suivant le type d'accès à ces données.

#### Les périphériques de flux

La première classe de périphériques fonctionne en mode d'un flux de données. Par exemple, un clavier envoie des données à chaque touche enfoncée/relâchée. Une souris à chaque mouvement ou clic. Une carte son va recevoir des données en mode flux pour jouer du son, et émettre des données en mode flux lorsqu'on enregistre depuis un micro.

Les périphériques en mode flux sont souvent appelés des *character device*. Ils ne supportent pas d'opérations de positionnement (on ne peut pas demander à un clavier la touche enfoncée il y a 10 secondes).

Ces périphériques possèdent en général un tampon (interne, dans le contrôleur et/ou dans le système d'exploitation) permettant de conserver une quantité limitée d'informations avant qu'elle ne soit traitée.

Si le système d'exploitation n'arrive pas à traiter les données suffisamment rapidement, une fois le tampon plein, des données peuvent être perdues si le périphérique est un périphérique d'entrée, ou alors des conséquences néfastes pourront lieu (son saccadé, par exemple) si le périphérique est un périphérique de sortie.

#### Les périphériques de stockage

La deuxième classe de périphérique fonctionne sous la forme d'une zone de données, divisée en blocs, qui peuvent être accédés de manière arbitraire. Ce type de périphériques, comme les disques durs, sont appelés *block device*.

Ces périphériques reconnaissent eux des opérations de positionnement, soit relatives (par exemple pour une bande magnétique, avancer ou reculer d'un certain nombre de blocs), soit absolues (le plus fréquent).

L'existence d'un tampon n'est pas nécessaire pour ces périphériques là, mais reste fréquente pour des raisons de performances.

#### Les autres périphériques

Certains périphériques ne rentrent pas totalement dans ces deux catégories (qui, comme toute catégorisation, est imparfaite). Voici quelques exemples :

- Les horloges, dont le fonctionnement sera détaillé plus loin ;
- Les écrans, qui sont hybrides entre le mode bloc et le mode flux : une image est gérée en mode bloc, mais une succession d'images constitue un flux ;
- Un graveur de CD/DVD, qui, lorsqu'il écrit, s'apparente plus au fonctionnement en mode flux, alors qu'en lecture il fonctionne en mode bloc.

### 1.1.3 PIO, MMIO et DMA

La deuxième grande question sur la gestion des périphériques est la technique utilisée pour transférer les données entre le périphérique et le reste du système. Trois modes sont utilisés.

#### Le mode PIO

Le mode le plus simple et le plus ancien est le mode PIO, pour « *Programmed IO* ». Ce mode consiste à lire (ou écrire) les données mot par mot (un mot étant une quantité de données manipulable en une seule opération, en général de 8 à 64 bits, donc de 1 à 8 octets), avec une instruction assembleur spéciale.

Ce mode est le plus simple à gérer pour le matériel. Cependant, il est très peu flexible, pour plusieurs raisons :

- Une instruction assembleur spécifique doit être utilisée, ce qui interdit l'écriture du code dans un langage de plus haut niveau, et nuit à la portabilité ;
- Le transfert de données est intégralement géré par le CPU, ce qui pour des débits importants peut coûter très cher ;
- La latence peut être importante, puisque si le CPU est occupé par un autre processus, le transfert n'aura pas lieu.

#### Le mode MMIO

Le mode MMIO pour « *Memory mapped IO* » est une évolution du mode PIO. Il consiste à utiliser une zone de mémoire spéciale pour effectuer la communication entre les applications (ou le système d'exploitation) et les périphériques.

En général, seul le noyau accède à ces zones de mémoires spéciales, mais la présence de la mémoire virtuelle permet d'attribuer sélectivement des droits sur ces zones de mémoires à des applications si nécessaire..

Lorsqu'une lecture ou écriture est faite sur ces zones de mémoire spéciale, les instructions de lecture/écriture ne sont pas envoyées sur le bus mémoire, mais sur le bus du périphérique. Ce sont les *bridge* de la carte mère qui gèrent cette bascule.

L'avantage principal de ce mode de transfert est qu'il ne nécessite aucune instruction spéciale, et permet d'utiliser toutes les fonctions des langages de plus haut niveau (comme `memcpy` en C).

Par contre, c'est toujours le CPU qui effectue le transfert, mot par mot, avec un coût qui peut être important sur un transfert avec un débit élevé ; et le problème de latence c'est pas résolu.

#### Le DMA

Enfin de palier au problème de coût CPU et de latence, une technique nommée DMA pour « *Direct Memory Acess* » a été inventée.

Lorsque l'on communique avec un périphérique via le DMA, on réserve une zone de mémoire vive pour le transfert. On signale alors à une puce spéciale, le contrôleur DMA, que l'on souhaite initier un transfert entre cette zone mémoire et un périphérique, ainsi que le sens du transfert. C'est le contrôleur DMA qui va alors transférer les données, à la vitesse requise pour le périphérique, et ceci sans intervention du CPU.

Le contrôleur DMA est capable de gérer plusieurs transferts en parallèle, un par canal DMA (16 par exemple). Par contre, un seul mot peut circuler sur le bus au même moment, bien sûr.

Afin d'éviter les problèmes de données non synchronisées, les cache L1 et L2 doivent être désactivés sur les zones mémoires concernées (en général possible avec une granularité d'une page, sur les MMUs modernes).

Une variante du DMA, utilisée sur le bus PCI, se nomme le *bus master*, et permet d'effectuer des transferts de type DMA sans contrôleur DMA. La carte PCI demande le contrôle du bus, et peut alors envoyer des requêtes au *south bridge* qui va les transférer à la mémoire centrale.

### 1.1.4 Les interruptions

#### Les boucles d'attente

Maintenant que nous savons transférer des données vers ou depuis un périphérique, il reste un certain nombre de questions : comment savoir qu'un périphérique est près à recevoir des données ? Qu'il y a des données disponibles pour la lecture ? Qu'un transfert DMA est terminé ?

Pour répondre à ces questions, la première solution est de faire une boucle d'attente : on teste régulièrement si des données sont disponibles, si le périphérique est près, si le transfert DMA est terminé. Si la condition n'est pas remplie, on réessaie plus tard, jusqu'à ce qu'elle le soit. La fréquence des tests va bien sûr dépendre du type de périphérique (élevée pour une carte réseau, faible pour un clavier, par exemple).

Mais ces tests réguliers peuvent finir par coûter cher en terme de CPU utilisé, surtout quand le nombre de périphériques à surveiller est élevé (cartes réseaux, disques, périphériques multimédias, périphériques d'entrée, ...) et la fréquence de vérification élevée.

De plus, dans le cadre d'un système multiprogrammé, ce test n'est de plus possible que quand le CPU est actuellement attribué à l'application qui utilise le périphérique, ou au noyau pour un pilote présent dans le noyau, ce qui peut introduire des latences élevées.

#### Le principe des interruptions

Pour pallier à ces deux problèmes, le mécanisme d'interruptions a été inventé. Il permet au matériel d'interrompre la tâche en cours d'exécution pour donner le CPU au pilote de périphérique correspondant, lorsque l'une des ces conditions est remplie.

Le principe théorique a déjà été étudié dans le cadre des processus et des threads : il s'agit de sauvegarde la valeur des registres spéciaux du processeur (comme le pointeur de l'instruction en cours), puis d'effectuer un saut sur un point d'entrée du système d'exploitation (le noyau en général). Celui-ci se charge de sauvegarder le reste des registres, et peut alors effectuer son traitement, puis rendre la main au processus interrompu.

La configuration des points d'entrée (adresses à l'intérieur du système d'exploitation) des différentes interruptions possibles se fait dans une table spécifique, présente en mémoire, nommée l'IDT (*Interrupt descriptors table*), de manière analogue aux tables de segments ou de pages (mais en général de petite taille).

#### IRQ, interruptions logicielles et NMI

Le premier type d'interruptions, dont nous avons déjà parlé rapidement, sont les interruptions logicielles. Elles sont émises volontairement par le programme, par exemple pour faire un appel système. Elles ne nous intéressent pas dans notre cas.

Le deuxième type d'interruptions, qui est le plus répandu dans la gestion des périphériques, est l'IRQ. Un contrôleur spécifique est présent sur la carte mère, et peut recevoir de la part des périphériques une demande d'interruption (*Interrupt Request*, IRQ en anglais), avec un numéro (de 0 à 15 par exemple). Le contrôleur va alors mémoriser la demande, et la signaler au CPU, via une patte spécialement dédiée.

Si le CPU est disponible, il va passer le contrôle au gestionnaire d'interruption préalablement enregistré pour cette IRQ. S'il ne l'est pas, l'interruption est mémorisée par le contrôleur d'IRQ, qui réessaiera régulièrement jusqu'à ce que le CPU soit disponible. Si plusieurs IRQ arrivent alors que le CPU est occupé, le contrôleur les transmettra dans un ordre fixe, en général les IRQs de numéro le plus bas sont prioritaires. En effet, le système d'exploitation peut désactiver (temporairement) les IRQ, pour éviter qu'un traitement critique ne soit interrompu.

Le troisième type d'interruption est la NMI, pour *Non masquable Interrupt*. Ce type d'interruption, en général émis par le CPU lui-même ou des composants très proches (la MMU par exemple) ne peut pas être désactivé. Il correspond le plus souvent à des erreurs, qui doivent impérativement être traitées (erreurs de protection dans la mémoire virtuelle, division par zéro, ...).

Une erreur survenant pendant le traitement d'une interruption est nommée un *double fault*, et une erreur survenant durant le traitement de cette erreur là, le *triple fault* provoque en général un arrêt ou un redémarrage de la machine. Ils sont cependant extrêmement rares.

## 1.2 Les couches logicielles

### 1.2.1 Architecture générale d'un pilote

Un pilote de périphérique est en général un module qu'il est possible de charger dynamiquement dans le système d'exploitation. Il contient une fonction d'initialisation, et doit ensuite implémenter un certain nombre de fonctions, dépendant du type de périphérique supporté.

En général, les pilotes sont découpés avec une partie générique et une partie spécifique, par exemple pour une carte réseau, il y a en général un pilote générique gérant les opérations de haut niveau, qui appelle des fonctions de plus bas niveau qui elles sont spécifiques à la carte.

### 1.2.2 La gestion des interruptions

L'un des grosses parties complexes d'un pilote va être de gérer les interruptions. En effet, le traitement d'une interruption doit se faire le plus rapidement possible : le contrôleur d'IRQ attend en général un acquittement pour pouvoir émettre de nouveau des interruptions, et le matériel, lui, a souvent besoin de données dans un intervalle court.

En général, le gestionnaire d'interruptions est découpé en deux parties : une partie basse qui s'exécute directement et effectue les opérations nécessaires au niveau du matériel pour revenir à une situation normale, et une partie haute, qui effectue un traitement potentiellement plus long et moins urgent, qui sera exécutée plus tard. Par exemple, dans le cas d'un paquet sur une carte réseau, la partie basse va recopier les données pour libérer le tampon de la carte réseau elle-même, mais la partie haute va gérer les protocoles de plus haut niveau (ip, tcp) et délivrer le paquet à l'application concernée.

Vu le nombre limité d'interruptions et le grand nombre possible de périphériques, il est possible qu'une interruption soit partagée entre plusieurs périphériques. Dans ce cas, chacun des pilotes est informé et doit vérifier si c'est bien son périphérique qui est concerné ou non.

### 1.2.3 Les copies et les tampons

#### Transfert direct

Dans la conception d'un système et des modes de communication avec les périphériques, il est nécessaire de se demander à quel niveau mettre des tampons (et donc, combien de copies il y aura).

Dans le mode le plus simple, le programme envoie directement les données sur le périphérique (avec un appel système `write` par exemple) ou les reçoit directement du périphérique (avec un appel système `read` par exemple). Le pilote transfère les données directement du programme au périphérique, en mode PIO ou DMA (ou alors, le programme communique directement avec le périphérique en mode MMIO).

Mais ce mode de fonctionnement comporte de nombreux problèmes : il perturbe la pagination (les pages concernées ne doivent pas être déplacées ou évincées), et nécessite un accès exclusif du programme sur le périphérique (par exemple, quand on reçoit un paquet d'une carte réseau, on ne sait pas quel programme est concerné avant de connaître le port TCP). En cas d'accès non bloquant, ou de threads, il est aussi possible que les données soient modifiées en cours de transfert, avec un comportement parfois dangereux.

#### Tampon dans l'espace noyau

Pour pallier à ces problèmes, en général une copie est faite dans l'espace noyau. Lorsqu'un programme souhaite émettre des données, elles sont copiées dans un tampon spécial du pilote de périphérique.

Ce mode de fonctionnement ajoute une copie des données (ce qui peut faire jusqu'à deux copies, si on doit ensuite copier les données vers le périphérique en mode PIO), mais apporte une plus grande souplesse et limite les risques de problèmes.

D'autres copies des données sont parfois nécessaires, par exemple, dans le cas d'un transfert réseau, il peut nécessaire de recopier les données pour pouvoir y ajouter les en-têtes des différents protocoles, gérer la fragmentation, ...

Un nombre élevé de copies peut dégrader les performances, mais il est souvent très difficile de les limiter sans compromettre la fiabilité ou la sécurité du système.

### 1.2.4 La gestion des erreurs

Le dernier grand point dans un pilote de périphérique concerne la gestion des erreurs. En effet, dans le monde du matériel, les erreurs se produisent hélas régulièrement.

La première solution quand une erreur arrive consiste tout simplement à réessayer, dans le cas d'une lecture sur un disque par exemple, parfois l'erreur a été transitoire, causée par une micro-poussière, ou par un champ magnétique externe, et réessayer permet d'avoir la bonne information.

La solution suivante consiste à reporter l'erreur au programme, qui devra alors la gérer comme il le peut (afficher un message d'erreur, se quitter proprement, ...).

Si vraiment l'erreur est critique, il peut être nécessaire de prendre des mesures draconiennes. Par exemple, s'il y a des erreurs sur les méta-données d'un système de fichiers, il peut être plus prudent d'interdire tout accès en écriture dessus, pour éviter une corruption en chaîne des données. Une erreur de lecture sur un disque système peut parfois forcer l'arrêt du système complet.

## 1.3 Quelques exemples

### 1.3.1 Les disques

#### Architecture générale des disques

Un disque est constitué d'un ou plusieurs plateaux, chacun équipé d'une tête de lecture/écriture. Ces plateaux sont découpés en cylindres concentriques, eux-mêmes découpés en secteurs. Un secteur est composé d'un préambule rappelant son identité (pour pallier à d'éventuels problèmes mécaniques), ainsi que des informations permettant le contrôle d'erreurs.

Pour lire (ou écrire) des données, il faut commencer par déplacer la tête pour qu'elle se trouve sur le bon cylindre, et ensuite attendre que le bon secteur se trouve sous la tête. L'opération la plus lente est le déplacement de la tête sur le bon cylindre, ce qui explique que les systèmes tentent de minimiser les déplacements de la tête.

Lorsque plusieurs requêtes sont en attente de traitement, il y a pour ça deux algorithmes : le premier consiste à toujours répondre à la demande la plus proche de la position courante. C'est le plus efficace en espérance, mais il est injuste, puisqu'il favorise fortement les données présentes au milieu du disque. Le second, l'algorithme d'ascenseur, consiste à choisir un sens de déplacement de la tête (vers l'intérieur ou vers l'extérieur), et de le respecter jusqu'à ce qu'il n'y ait plus de demandes dans ce sens. Alors, on inverse le sens, et ainsi de suite.

Par exemple, si la tête est en position 5, que les requêtes sont pour les positions 8, 3, 2, 6 et que la direction actuelle est vers l'extérieur, on ira d'abord traiter la requête pour la position 6, puis pour la position 8, ensuite pour la position 3 et enfin pour la position 2.

Il faut aussi savoir que les disques comportent en général des complexités supplémentaires, comme un nombre de secteur dépendant du cylindre (les cylindres de l'intérieur du disque étant de surface inférieure, ils ont moins de secteurs), ou des secteurs de remplacement pour pallier aux secteurs défectueux.

#### Le RAID

Le RAID est une technique permettant de grouper les disques entre eux, afin de donner l'illusion d'un seul disque dur de capacité supérieur, avec en général de la tolérance de faille.

Il peut être géré soit par le matériel (carte contrôleur RAID) soit par le système d'exploitation (Linux le gère par exemple).

Voici un exposé rapide des modes RAID les plus courant :

**RAID 0** Le RAID 0 consiste à répartir les données sur les différents disques. Les données sont découpées en *strip* d'une taille allant de 1 à quelques secteurs. Par exemple, avec un RAID de 2 disques, des secteurs de 512 octets et un strip de 2 secteurs, le premier Ko sera écrit sur le premier Ko du premier disque, le deuxième Ko sur le premier Ko du deuxième disque, le troisième Ko sur le deuxième Ko du premier disque, et ainsi de suite. Le RAID 0 donne les meilleurs performances, et permet d'utiliser toute la capacité, mais n'apporte aucune tolérance de faille.

**RAID 1** Le RAID 1 consiste à dupliquer les données entre deux (ou plus) disques durs. La lecture est accélérée parce qu'il est possible de répartir les requêtes de lecture sur tous les disques, mais l'écriture est légèrement plus coûteuse. La RAID 1 coûte cher en capacité (la moitié de la capacité, pour une copie) mais apporte de la tolérance de faille.

**RAID 1+0** Le RAID 1+0 ou RAID 10 consiste à utiliser du RAID 0 et du RAID 1 simultanément, par exemple, avec 6 disques, on groupera les disques par 2 pour faire du RAID 1, puis on fera du RAID 0 entre ces trois disques virtuels (ou l'inverse).

**RAID 4** Le RAID 4 permet d'avoir de la tolérance de faille pour un coût en capacité réduit. Il consiste à effectuer du RAID 0 sur tous les disques sauf un, et d'utiliser le disque restant pour écrire la parité de chacun des strips. Si un disque est détruit ou retiré, les données peuvent être reconstruites à partir de la parité.

**RAID 5** Le RAID 5 est une légère optimisation du RAID 4, où la parité est répartie sur tous les disques. Le RAID 5, comme le RAID 4, a de bonnes performances en lectures, mais des performances plus médiocres en écriture, puisqu'il faut reconstruire la parité.

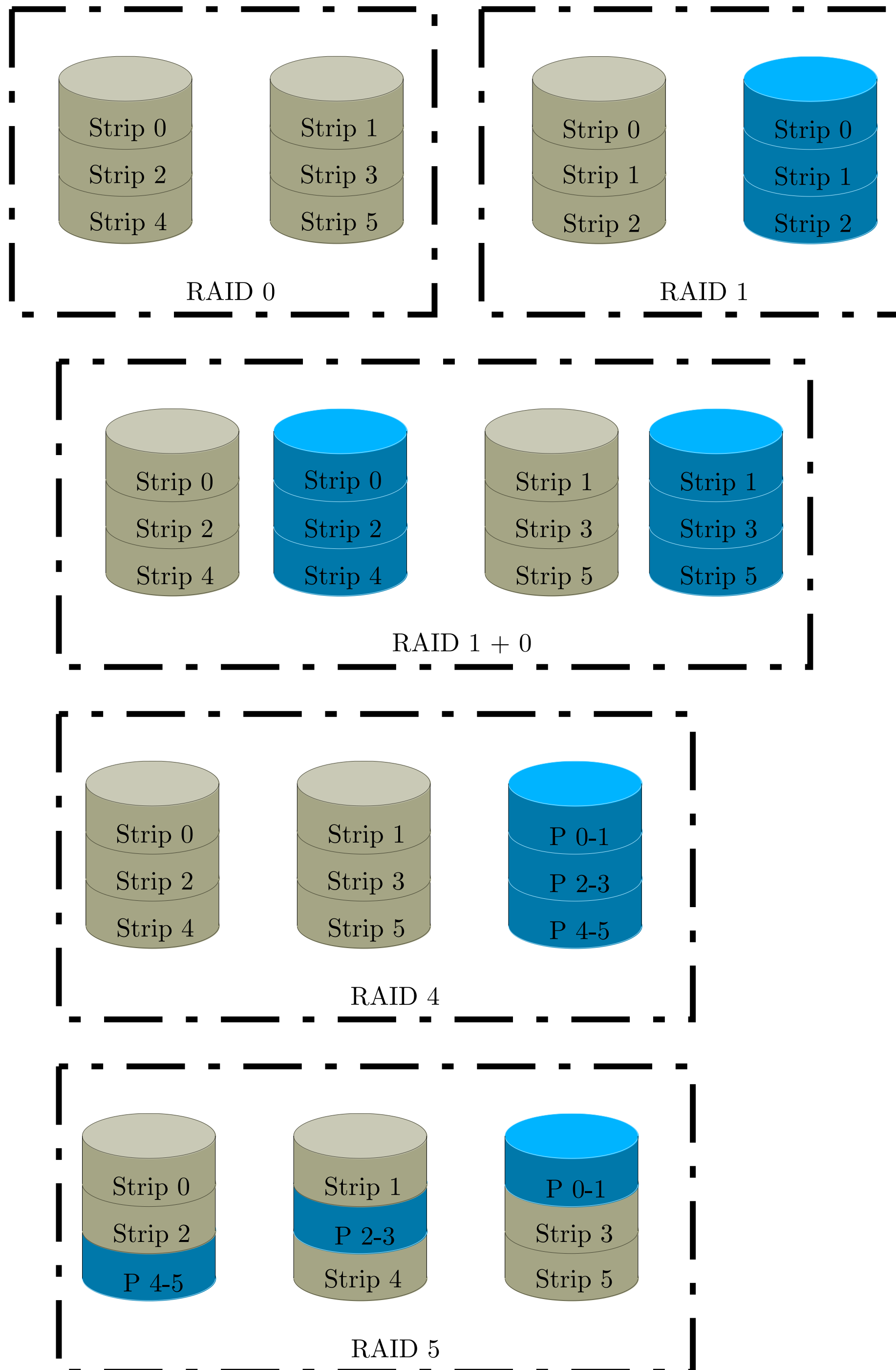


FIG. 1.1 – Fonctionnement du RAID

La figure 1.2 compare les caractéristiques des différents formats de RAID entre $n$ disques de capacité $C$ et de vitesse unitaire $V$. Les vitesses données sont les maximales théoriques, dans la pratique elles sont souvent légèrement inférieures. La tolérance de faille est donnée en pire cas.

| **Niveau RAID** | **Capacité** | **Tolérance de failles** | **Lecture** | **Écriture** |
|---|---|---|---|---|
| RAID 0 | $n \times C$ | Aucune | $n \times V$ | $n \times V$ |
| RAID 1 | $C$ | $n - 1$ disques | $n \times V$ | $V$ |
| RAID 1+0 | $\frac{n}{2} \times C$ | 1 disque | $n \times V$ | $\frac{n}{2} \times V$ |
| RAID 4 ou 5 | $(n - 1) \times C$ | 1 disque | $n \times V$ | Mauvaise (variable) |

Fig. 1.2 – Comparaison des RAIDs

### 1.3.2 Les horloges

Les horloges sont une classe de périphérique à part, puisqu'elles ne permettent pas de lire ou d'écrire des données au sens usuel du terme. Une horloge reçoit une impulsion électrique à une fréquence très élevée (en général celle de la carte mère, soit plusieurs centaines de MHz).

Elle est programmée pour émettre une interruption toutes les $n$ impulsions (par exemple, à une fréquence de 1000Hz). Le système d'exploitation utilise ensuite cette interruption pour maintenir l'heure courante, et pour effectuer les opérations comme le scheduling.

Lorsque les processus nécessitent des alarmes pour leur utilisation propre, le système d'exploitation ne fait que gérer en interne une file d'attente des alarmes, classées (la plus proche en premier). À chaque interruption, il va comparer l'heure prévue de la prochaine alarme avec l'heure courante, et éventuellement déclencher l'alarme. L'implémentation exacte dépend du système, bien sûr.

### 1.3.3 Les terminaux

Une autre catégorie de périphériques intéressante est un terminal en mode texte. Ces terminaux, encore utilisés via des émulateurs de terminaux (type `xterm`) dans le monde Unix, reçoivent des flux de caractères. Ils comprennent aussi des séquences spéciales, nommées séquences d'échappement, afin de gérer des comportements comme l'affichage en couleur, le déplacement du curseur, l'effacement de caractères, ...

Ces terminaux, n'utilisant qu'un flux de caractère comme moyen de communication, peuvent être encapsulés facilement dans des protocoles diverses, comme un port série, un modem, une connexion TCP ou même une connexion de plus haut niveau, comme un `ssh`.

# Chapitre 2

# Systèmes de fichiers

## 2.1 Les fichiers

### 2.1.1 Principes des fichiers

Les programmes ont besoin de stocker des données de manière persistante, tout en pouvant les retrouver par la suite. Il est aussi nécessaire de pouvoir transférer ces données d'un ordinateur à un autre, en utilisant des supports amovibles.

Le concept retenu dans tous les systèmes pour répondre à ce besoin est le concept de "fichier". Un fichier est un conteneur de données, identifié par un nom (ou un numéro) et sur lequel on peut effectuer des opérations de lecture et d'écriture. Un fichier possède aussi des méta-données, dépendant du système, comme une date de création et des permissions.

Un fichier a une taille qui n'est, en général, pas fixe (contrairement aux périphériques de stockage), et de nombreux fichiers peuvent exister sur le même périphérique.

### 2.1.2 Types de fichiers

Les fichiers peuvent être de plusieurs types, et ceci à plusieurs niveaux.

#### Fichiers normaux et spéciaux

La première distinction possible est entre des fichiers réguliers ("normaux") et des fichiers spéciaux. Les fichiers réguliers sont ceux qui correspondent à la définition précédente, les fichiers spéciaux sont des ressources présentées sous forme de fichiers.

Par exemple, un périphérique peut être présenté sous forme d'un fichier, ou alors un canal de communication entre deux processus, comme nous l'avons vu dans la partie sur les tubes nommés.

#### Contenu des fichiers

La deuxième distinction possible est sur le contenu des fichiers. Les fichiers peuvent être structurés de différentes manières :

1. Sous forme de flux d'octets, sans aucune signification particulière pour le système ;
2. Sous forme de texte en ASCII, composé de lignes ;
3. Sous forme d'enregistrements de taille fixe.

Les système modernes ne reconnaissent eux que le premier type de fichier, et ce sont les applications qui doivent savoir comment les données sont représentées.

Cependant, les systèmes des main-frames, encore utilisés dans certaines industries, ont souvent des types de fichiers différents, et des appels systèmes spécifiques aux fichiers du 2ème ou 3ème type.

### 2.1.3 Attributs des fichiers

#### Nommage des fichiers

Les fichiers doivent pouvoir être retrouvés par leur nom. Ce nom, dans les modèles les plus simples, est un numéro choisi par le système à la création du fichier.

Dans les systèmes modernes, pour le confort de l'utilisateur, les fichiers peuvent avoir de véritables noms. Les règles de nommage (caractères autorisés, taille maximale, ...) dépendent des systèmes. Le DOS ne supportait

que des fichiers de 8+3 caractères, mais les systèmes récents supportent tous en général 255 caractères au minimum. Certains systèmes sont sensibles à la casse (les Unix) et d'autres non (les Windows).

Les noms des fichiers comportent en général une extension, qui indique le type de données contenues dans le fichier. Par exemple, un fichier LaTeX sera nommé `.tex`, un fichier source C `.c` et ainsi de suite.

Dans la plupart des systèmes, l'extension est une simple convention pour que les utilisateurs reconnaissent les fichiers, mais n'a aucune signification pour le système d'exploitation. Dans d'autres, comme Windows, l'extension a une réelle signification.

Outre l'extension, certains programmes, comme Nautilus (et plus généralement les applications du bureau Gnome) reconnaissent les fichiers en lisant le début (ou la fin) à la recherche d'une signature, par exemple une image au format PNG commence par les caractères `PNG\r\n`.

#### Sécurité des fichiers

La gestion de la sécurité sur les fichiers est un point important, puisque des données confidentielles peuvent souvent y être stockées.

Certains systèmes d'exploitation permettent d'associer un mot de passe à un fichier, et demandent à l'utilisateur de saisir le mot de passe à chaque fois qu'une application tente d'accéder au fichier. Ce mode de sécurité n'est cependant plus utilisé de nos jours au niveau du système (il peut l'être encore au niveau applicatif).

Les modes de sécurité les plus utilisés sont le mode Unix et les ACLs.

**Le mode de sécurité Unix** Dans ce mode de sécurité, les fichiers appartiennent à un utilisateur et à un groupe. Trois séries de droits (lecture, écriture et exécution) sont positionnés : une pour l'utilisateur à qui le fichier appartient, une pour les autres membres du groupe auquel le fichier appartient, et une dernière pour tous les autres utilisateurs. Seul le propriétaire du fichier (ou l'administrateur système) peut modifier ces droits.

**Les ACLs** Les ACLs (« *access control lists* ») sont un mode de sécurité sur les fichiers plus flexible, mais qui rapidement devenir complexe à gérer, et plus coûteux en performances, sur des systèmes avec beaucoup de fichiers et beaucoup d'utilisateurs.

Ils consistent à associer aux fichiers une liste d'accès autorisés, par exemple une liste d'utilisateurs ayant le droit de lecture, et une autre liste d'utilisateurs ayant le droit en écriture.

Les ACLs sont disponibles sous forme d'extensions sur la plupart des Unix, et en standard sous Windows 2000 et ultérieurs.

#### Autres attributs des fichiers

Les autres attributs sont nombreux, et dépendent des systèmes. Il y a en général une date de création et/ou de dernière modification, la taille du fichier, parfois une date de dernier accès, et éventuellement des attributs spécifiques, comme un fichier caché (n'apparaît pas lorsqu'on demande la liste des fichiers).

### 2.1.4 Primitives d'accès aux fichiers

Le mode d'accès à un fichier est général découper en trois phases.

#### Ouverture de fichiers

Tout d'abord, le fichier est créé (pour un nouveau fichier), ou ouvert (pour un fichier existant), via un appel système spécifique (`open`), et à partir du nom du fichier. Cet appel va effectuer une vérification de droits, ainsi qu'un certain nombre de traitements spécifiques au système de fichiers utilisé.

Cet appel système prend souvent en paramètres des informations sur le mode d'ouverture du fichier (lecture seule, ajout à la fin, écriture, ...).

Il renvoie une ressource (en général un entier), nommé un *file descriptor* permettant de manipuler le fichier ensuite.

Dans la plupart des systèmes, la création d'un fichier inexistant effectue au même moment une opération d'ouverture sur le fichier. Parfois il est nécessaire de procéder en deux étapes : d'abord le créer, puis l'ouvrir.

#### Manipulation de fichiers

Une fois le *file descriptor* obtenu, il est possible d'effectuer des opérations de lecture et d'écriture sur le fichier, via deux appels systèmes, `read` et `write`.

Soit ces appels systèmes prennent en paramètre l'endroit du fichier où les données doivent être lu ou écrites, soit, dans la plupart des systèmes, les données sont automatiquement lues ou écrites à un endroit nommé "position courante" dans le fichier.

Cette position courante est initialement le début ou la fin du fichier (suivant les paramètres donnés à l'appel système d'ouverture), et est modifiée par les opérations de lecture et d'écriture (pour toujours pointer à la fin des dernières données lues/écrites).

Un appel spécifique permet de modifier cette position, soit de manière relative, soit de manière absolue.

#### Fermeture et suppression de fichiers

Une fois que le fichier n'est plus nécessaire pour l'application en cours, il doit être fermé, pour permettre au système d'exploitation de libérer les ressources allouées en interne pour gérer les fichiers ouverts.

Si un fichier n'est plus du tout nécessaire pour l'utilisateur, il doit être supprimé pour libérer l'espace disque utilisé par le fichier, et éviter de polluer les listes de fichiers.

Lorsqu'un programme tente de supprimer un fichier utilisé par un fichier utilisé par un autre programme, trois solutions sont possibles :

1. Le système refuse l'opération de suppression (Windows) ;
2. Le fichier est marqué comme "à supprimer" mais reste présent sur le disque le temps qu'il est encore ouvert par un processus (Unix) ;
3. Le système ignore le problème, et le comportement est indéfini (Dos).

#### Autres appels sur les fichiers

Il existe d'autres appels sur les fichiers, en particuliers des appels pour récupérer et modifier les méta-données. Certains systèmes fournissent des appels pour vérouiller un fichier (s'assurer qu'un seul processus y accède), ou les projeter en mémoire, par exemple.

### 2.1.5 Implémentation des fichiers

#### Secteurs et blocs

Le premier concept important pour l'implémentation des fichiers est la division du fichier en blocs. Un disque est lui-même divisé en secteur. Une opération de lecture ou d'écriture s'effectue (au minimum) sur un secteur entier. Un secteur a en général une taille de 512 octets sur un disque dur (souvent plus sur les supports optiques).

Pour des raisons diverses (que nous verrons par la suite), les secteurs sont souvent regroupés par le système d'exploitation en blocs d'une taille supérieure, appelés *cluster* en anglais (blocs en français, en général). Ces blocs ont une taille variable, souvent égale à 4Ko (qui a la bonne idée d'être aussi la taille usuelle d'une page mémoire, ce qui simplifie les interactions entre la VM et le système de fichier, même si ce n'est en rien obligatoire).

#### Fichiers simples

La manière la plus simple d'implémenter un fichier est de le stocker de manière continue sur le disque, et de ne conserver que le secteur où le fichier commence. On peut alors retrouver n'importe quelles données par une simple addition.

Deux problèmes se posent avec ce mode de fonctionnement : tout d'abord, il est difficile d'agrandir un fichier après sa création, car il est fort possible que le secteur d'après soit utilisé par un autre fichier. Une solution pourrait être d'indiquer, à la création, la taille maximale que le fichier va faire, mais c'est une approche très peu pratique.

Le deuxième problème survient lorsqu'on efface des fichiers : on va créer des trous de différente taille, et comme pour la segmentation, on risque d'avoir deux trous de 128Ko et ne pas avoir la place pour mettre un fichier de 192Ko.

Bien que limité, ce mode de fonctionnement reste la solution la plus simple dans certains cas : lorsqu'on ne peut plus modifier les données une fois écrites, c'est à dire, sur les supports en lecture seule comme les CD-Rom ou les DVD.

#### FAT

Une autre solution consiste à utiliser une table, nommée FAT (« *File Allocation Table* ») qui va contenir, pour chaque secteur, le numéro du secteur suivant. Le fichier lui-même est alors identifié par son premier secteur. Si le premier secteur est le numéro 37, et qu'on souhaite le second secteur, on va aller lire l'entrée numéro 37 de la FAT, qui contiendra le numéro du secteur suivant, par exemple le numéro 42. Si on souhaite le troisième secteur, on ira lire l'entrée numéro 42 de la FAT, et ainsi de suite.

Cette FAT est destinée à rester en mémoire, pour permettre de suivre rapidement la chaîne des secteurs jusqu'à trouver le secteur qui nous intéresse. Mais si le disque fait 40 Go et contient des secteurs de 4 Ko, la FAT aura 10 millions d'entrées, ce qui pose quelques problèmes.

#### I-nodes

Une autre solution consiste utiliser des secteurs pour stocker les informations sur les blocs utilisés par les fichiers. Le fichier est identifié par un *i-node*, un numéro de bloc contenant une liste de blocs. Lorsque le fichier est ouvert, cet i-node est chargé en mémoire. Ensuite, un bloc est retrouvé rapidement en regardant dans cet i-node (le bloc numéro 10 est la dixième entrée de l'i-node).

Cette approche évite d'avoir à suivre de longues listes chaînées, et évite de devoir maintenir une grosse table en mémoire, seuls les fichiers ouverts sont concernés. Elle impose cependant une limite sur la taille maximale des fichiers, si un i-node fait 512 octets et un numéro de secteur 4 octets, un fichier est limité à 128 secteurs, par exemple.

Afin de contourner cette approche, on utilise donc aussi un chaînage, mais entre les i-nodes : la dernière (ou les quelques dernières) entrée d'un i-node correspond non pas à un nouveau secteur de données, mais à une nouvelle liste de secteurs, et ainsi de suite.

## 2.2 Les répertoires

### 2.2.1 Principes des répertoires

Jusqu'à présent, nous n'avons pas parler du moyen de retrouver les fichiers. Nous avons supposé qu'ils étaient accessibles via un nom ou un numéro, sans précision.

La solution à ce problème est le concept de répertoire. Qu'est qu'un répertoire (*directory* en anglais) ? Dans la vrai vie, un répertoire c'est une liste de noms, avec pour chacune un numéro de téléphone ou une adresse.

En informatique, c'est à peu près la même chose : une liste de noms de fichiers, avec à chaque fois leur adresse (c'est à dire, leur premier bloc pour une FAT, ou le numéro d'i-node).

### 2.2.2 Répertoires récursifs et chemins

Il existe un répertoire principal sur chaque système de fichiers, nommé le répertoire principal, ou répertoire racine.

Sauf sur les plus primitifs de systèmes, ce répertoire racine peut lui-même contenir des répertoires, et ainsi de suite, créant en réalité un arbre (au sens de la théorie des graphes) de répertoires et de fichiers.

Comment retrouver un fichier alors ? C'est là qu'entre en jeu la notion de chemin : un séparateur est choisi (/ sous Unix, \ sous Windows, > sous Multics, par exemple). Un chemin est constitué d'une liste de répertoires à traverser pour atteindre un fichier, séparés par ce fameux caractère.

Les chemins peuvent être absolus (partant du répertoire racine) ou relatifs (partant d'un autre chemin, déjà connu, nommé le répertoire courant).

### 2.2.3 Liens physiques et symboliques

Parfois, il peut arriver qu'on souhaite avoir le même fichier à deux endroits différents de l'arborescence, pour différentes raisons, sans pour autant en avoir une copie (pour ne pas avoir deux fois plus de place disque utilisée, et pour éviter que les deux versions ne divergent par la suite). Il y a deux moyens de traiter cette demande.

#### Les liens physiques

Les liens physiques consistent à avoir le même fichier (même secteur de départ ou même i-node) référencé dans deux répertoires différents. Un peu comme la même personne peut être dans plusieurs carnets d'adresse, et sous des noms différents "papa" dans l'un, "Mr Dupont" dans l'autre, et "Jean" dans le troisième.

Le fichier, s'il est modifié en accédant par un nom, sera vu comme modifié aussi en y accédant par d'autres noms. S'il est supprimé sous un nom, il restera sur le disque le temps qu'il y a encore un seul nom qui permet de le retrouver.

La question du stockage des méta-données est alors importante, nous y reviendrons.

#### Les liens symboliques

La deuxième solution consiste à avoir un type de fichier spécial, nommé lien symbolique, qui ne contient que le chemin du fichier destination. Lors d'une tentative d'ouverture du lien symbolique, le système va alors suivre le lien, et ouvrir le fichier dont le nom était contenu dans le lien.

Ce mode de fonctionnement permet de faire des liens entre systèmes de fichiers, mais nécessite de faire très attention aux éventuelles boucles.

### 2.2.4 Principe des points de montage

Grâce au chemin, nous savons comment accéder à un fichier dans un système de fichiers. Mais il reste un problème à résoudre : le même système d'exploitation possède en général plusieurs systèmes de fichiers (plusieurs périphériques, par exemple).

Il y a, là encore, deux approches possibles : la première, celle utilisée sous DOS et sous Windows, consiste à utiliser des lettres (de A à Z) pour identifier les différents systèmes de fichier, et de préfixer les chemins par cette lettre et un séparateur spécial ( :).

L'autre approche, celle des Unix, consiste à avoir un système de fichier racine (en général celui contient le système d'exploitation lui-même), et d'accrocher dessus, via une opération nommée "montage" les autres systèmes de fichier, à la place de répertoires. Ainsi, si un CD est monté dans `/mnt/cdrom`, alors le fichier `/data/music.ogg` sera accessible sur `/mnt/cdrom/data/music.ogg`.

Ce mode de fonctionnement est beaucoup plus souple, vu qu'il permet à l'utilisateur de découper lui-même ses données entre plusieurs périphériques, sans que les programmes n'en soient impactés. Combiné aux liens symboliques, il accorde une très grand flexibilité.

### 2.2.5 Implémentation des répertoires

L'implémentation la plus simple d'un répertoire consiste à stocker, de manière linéaire, le nom d'un fichier, son premier bloc (ou son i-node) puis le fichier suivant, et ainsi de suite. Il faut alors gérer la fragmentation, mais ce n'est qu'une difficulté technique.

Ce mode de fonctionnement nécessite de parcourir tous les fichiers pour en trouver un, et peut donc s'avérer inefficace avec beaucoup de fichiers dans le même répertoire. Il possible d'utiliser des structures plus évoluées, comme des tables de hash ou des arbres binaires afin de retrouver l'information rapidement.

Une autre question qui se pose consiste à savoir où stocker les méta-données (taille, date de modification, permissions, ...). Si on utilise un système sans i-node, la solution généralement choisie est de les stocker dans le répertoire lui-même, à côté du nom et du numéro de bloc.

Si on utilise un système d'i-nodes, l'i-node est lui-même utilisé pour stocker les méta-données. Ceci à l'avantage de réduire la taille des répertoires, mais surtout, permet la mutualisation des méta-données dans le cas des liens physiques.

## 2.3 Implémentation des systèmes de fichiers

### 2.3.1 Architecture générale

Maintenant que nous avons vu les deux composants des systèmes de fichiers (les fichiers et les répertoires), regardons quelques considérations annexes sur la manière dont un système de fichier est architecturé.

Un système de fichier commence en général par un secteur spécial, nommé secteur d'amorçage, dont la seule utilité est de permettre, éventuellement, de démarrer un système d'exploitation depuis cette partition.

Ensuite, un bloc spécial (le « *superbloc* ») contient des méta-données générale du système de fichiers, comme sa taille globale, la taille des blocs, et par exemple la date où le système a été vérifié pour la dernière fois.

Des tables systèmes, comme la FAT, sont alors stockées. Ensuite, se trouve le répertoire racine, et enfin, le reste des données (les fichiers eux-mêmes, les autres répertoires, ...).

### 2.3.2 Gestion des blocs libres

Un point, essentiel, que nous n'avons pas encore abordé est la gestion des blocs libres. Deux méthodes sont utilisées pour garder une trace des blocs libres, et permettre la création ou l'agrandissement des fichiers.

La première méthode consiste à utiliser une liste de blocs libres. Des blocs spéciaux (en général, des blocs libres particuliers) contiennent une liste de numéros de blocs libres, avec comme dernier élément le numéro du prochain bloc listant des blocs libres. Une partie de cette liste est gardée en mémoire, afin de satisfaire aux requêtes rapidement. La place prise est de quelques octets (4 en général) par bloc, mais vu que la liste est stockée dans les blocs libres, le gaspillage est en réalité très faible.

Le deuxième méthode consiste à avoir un bitmap du disque (un bit par bloc), à 1 si le bloc est libre et 0 sinon (ou inversement). Cette méthode nécessite une taille de constante, de 1 bit par bloc, soit 5 Mo pour un disque de 160 Go avec des blocs de 4Ko. Le bitmap possède deux avantages : il prend moins de places (un seul bit par bloc) et peut donc plus facilement être gardé en mémoire, et il permet plus facilement de trouver des blocs libres proches les uns des autres. Son inconvénient est qu'il prend de la place (certes faible) même quand le disque est presque plein.

### 2.3.3 Problématiques de performances

L'un des principaux objectifs d'un bon système de fichiers est de donner de performances, et ceci pour toutes les opérations : lecture et écriture, séquentielle ou non, création ou effacement de fichiers, modification de méta-données, ...

Pour les systèmes de fichiers orientés disque dur, le principal facteur de lenteur est le temps de déplacement des têtes, et donc une grande partie de l'optimisation sur ces systèmes et de lutter contre la fragmentation des fichiers, et de grouper les écritures. Sur les cartes mémoires, la problématique est différente.

Il est à noter qu'il est difficile d'avoir de bonnes performances dans tous les domaines (lecture et écriture, petits fichiers et gros fichiers, ...), certains systèmes ayant des performances meilleures dans certains cas, et d'autres d'en d'autres cas. Choisir un système de fichier approprié, parmi ceux proposés par le système d'exploitation, suivant l'utilisation qui sera faite de la partition est la tâche de l'administrateur système.

### 2.3.4 Problématiques de fiabilité

Une autre problématique fondamentale pour un système de fichier est sa fiabilité, sa tolérance aux pannes. Les problèmes peuvent venir de multiples raisons :

- Un bloc (ou une série) corrompu sur le disque ;
- L'extinction brutale de la machine, pendant qu'une opération d'écriture était en cours ;
- Un bug dans une partie du noyau ayant entraîné une corruption de données lors d'une écriture.

Pour les données des fichiers même, les systèmes de fichier ne font en général aucun effort particulier - si on souhaite accroître la fiabilité de ces données, il faut faire du RAID et des sauvegardes.

Par contre, les systèmes accordent souvent une importance élevée à la corruption de leurs propres méta-données, qui peuvent amener, dans le pire cas, une perte de l'ensemble du système de fichiers. Certaines données, comme le *superbloc*, la bitmap des blocs libres ou la FAT peuvent être recopiées plusieurs fois sur le disque.

Le point qui a donné lieu au plus grand nombre d'efforts est l'arrêt du système pendant l'écriture. En effet, lorsqu'on ajoute des données à la fin d'un fichier, par exemple, les informations suivantes doivent être mises à jour :

- La date de dernière modification du fichier ;
- La liste des blocs utilisés par le fichier (i-node ou FAT) ;
- La liste des blocs libres (liste ou bitmap) ;
- Les données elles-mêmes.

Si le système est arrêté alors que certaines données ont été mises à jour et pas d'autres, les conséquences peuvent être graves. Par exemple, si le bloc est déjà ajouté au fichier, mais pas enlevé de la liste des blocs libres, il pourra être assigné à un autre fichier.

Deux solutions sont possibles : la première consiste à lancer automatiquement un outil de vérification si le système n'a pas été arrêté proprement, outil qui corrigera les erreurs. L'autre solution est le système de fichier journalisé, que nous verrons plus loin.

## 2.4 Quelques exemples

### 2.4.1 ISO9660

Le système ISO9660, du nom du standard dans lequel il est décrit, est celui utilisé sur les CD-Rom et sur certains DVD-Rom. Un CD-Rom est constitué d'une longue spirale divisée en blocs de 2048 octets utiles.

Après une zone de 16 secteurs non spécifiée par la norme (utilisée pour rendre les CD-Roms amorçables, par exemple), se trouve le superbloc nommé ici *primary volume descriptor*, qui contient des informations textuelles courtes sur le CD, et le numéro du premier secteur du répertoire racine.

Chaque entrée d'un répertoire contient les informations suivantes, dans l'ordre :

| **Taille** | **Contenu** |
|---|---|
| 1 | La taille totale de l'entrée |
| 1 | La taille des attributs étendus |
| 8 | La position du fichier (deux fois : en little et en big endian) |
| 8 | La taille du fichier (toujours deux fois) |
| 7 | La date et l'heure |
| 7 | Des attributs particuliers |
| 5-16 | Le nom du fichier sous une forme 8.3 avec un numéro de version additionnel |
| Variable | Des extensions, non définies par la norme |

Les extensions sont utilisées par différents systèmes (Unix, Windows, MacOS) pour stocker des attributs supplémentaires (noms plus longs, permissions, liens symboliques, ...). L'extension Unix se nomme "rock ridge" et l'extension Windows "Joliet".

Des nouvelles versions de la norme (ISO9660 :1999 par exemple) ont aussi été créées, pour contourner certaines limites (comme la taille des noms de fichiers).

Dans le cas des CD-R, un support multisession a été ajouté. Lorsque de nouvelles données doivent être ajoutées, un nouveau superbloc est créé, et le répertoire racine, ainsi que tous les répertoires nécessaires, sont recopiés. Ils peuvent cependant toujours référencer les données des sessions précédentes (le contenu des fichiers n'a pas à être dupliqué, lui).

### 2.4.2 FAT

FAT, le système de fichiers du DOS et de certains Windows, comporte plusieurs variantes.

Toutes ces variantes supportent une hiérarchie de répertoire arbitrairement profonde, avec des entrées d'une taille fixe de 32 octets. Dans toutes ces variantes aussi, l'allocation des fichiers est gérée par une FAT redondée (deux copies sont maintenues).

#### FAT-12 et FAT-16

Dans le système initial, les entrées sont :

| **Taille** | **Contenu** |
|---|---|
| 8 | Nom du fichier |
| 3 | Extension du fichier |
| 1 | Attributs |
| 10 | Réservé |
| 4 | Date et heure |
| 2 | Numéro du premier bloc |
| 4 | Taille |

À noter que la date est encodée de manière assez complexe, avec 5 bits pour les secondes, 6 bits pour les minutes, 5 pour les jours, 4 pour les mois et 7 pour les années. Ce découpage permet une précision de 2 secondes, et de stocker les années de 1980 à 2107. En utilisant un décompte des secondes depuis le 1er janvier 1980, sur les mêmes 32-bits, il aurait été possible de stocker des dates avec une précision d'une seconde, et jusqu'en 2116.

Seul le numéro du premier bloc est contenu, ensuite, les blocs suivant sont retrouvés par une FAT classique, comme vue précédemment.

Les numéros de blocs (dans la FAT et dans les répertoires) peuvent être sur 12-bits ou 16-bits. Les FAT en 12-bits sont utilisées sur les disquettes, les FAT en 16-bits sur les disques durs de petite capacité. Avec des blocs de 4Ko, la FAT-16 peut adresser jusqu'à 256Mo ; avec des blocs de 32Ko, on monte jusqu'à la taille immense de 2Go).

#### FAT-32

Pour contourner ces limites, une nouvelle version de la FAT fut proposée : la FAT-32 (en réalité FAT-28, puisque seule les 28 premiers bits sont réellement utilisables). La FAT-32 permet d'utiliser des disques d'1To, et ceci avec des blocs de 4Ko. La FAT est maintenant de grande taille, faisant jusqu'à 1Go pour $2^{28}$ entrées de 4 octets chacune. Afin d'éviter de gaspiller de la place, contrairement aux FAT 12 et 16 où la taille de la FAT était fixe (8Ko pour FAT-12, 128Ko pour FAT-16) la taille est maintenant dépendante de la taille du disque.

La taille d'un fichier reste elle par contre limitée à 4Go.

Les 10 octets réservés en FAT-16 ont été réassignés en FAT-32, comme suit :

| **Taille** | **Contenu** |
|---|---|
| 8 | Nom du fichier |
| 3 | Extension du fichier |
| 1 | Attributs |
| 1 | Compatibilité avec Windows NT |
| 5 | Date et heure de création, précision de 10ms |
| 2 | Date de dernier accès |
| 2 | Numéro du premier bloc (16-bits de poids fort) |
| 4 | Date et heure de modification |
| 2 | Numéro du premier bloc (16-bits de poids faible) |
| 4 | Taille |

Au même moment a été ajouté le support des noms de fichiers longs. Tous les fichiers doivent avoir un nom court (en 8+3), mais ils peuvent aussi avoir un nom long. Le nom court est généré automatiquement par le système si nécessaire. Le nom long, lui, est stocké découpé dans plusieurs entrées, se présentant comme des fichiers invalides (le champ Attribut étant à une valeur invalide) pour un DOS plus ancien.

### 2.4.3 Ext-2

#### Le système Unix original

Le système Unix original est constitué comme suit :
- Un superbloc qui contient des informations sur le disque ;
- Une série d'inodes, de 64 octets chacun ;
- Des blocs de données, contenant les fichiers et les répertoires.

Les inodes, numérotés, permettent de retrouver les données et métadonnées des fichiers même lors de l'utilisation de liens physiques. Ils contiennent les métadonnées (permissions, dates, taille), l'adresse des 10 premiers blocs du fichier, et 3 adresses spéciales : l'indirection simple, double et triple. L'indirection simple est l'adresse d'un bloc (dans la zone de données) qui contient une liste de blocs utilisés pour stocker le fichier (après les 10 premiers). L'indirection double contient une liste de blocs d'indirection simple, et l'indirection triple une nouvelle liste de blocs d'indirection double.

Ce principe permet d'avoir des fichiers de très grande taille (via le bloc d'indirection triple) tout en gardant un accès rapide, et avec très peu de surcharge, pour les petits fichiers.

Les répertoires, sur le système Unix classique, sont constitués d'entrées de taille fixe, de 16 octets. 14 sont utilisés pour le nom, et 2 pour le numéro de l'inode (donc, jusque 65536 fichiers peuvent exister).

#### Ext-2

Le système ext-2 utilisé nativement par Linux garde les grandes idées du système Unix original, en apportant un certain nombre d'améliorations.

Tout d'abord, les entrées des répertoires ont désormais des tailles variables, permettant d'avoir des noms de fichier long. Ces entrées de taille variable complexifient la gestion des répertoires, en créant des risques de fragmentation, mais le problème reste limité.

Une autre amélioration est la possibilité (non présente dans les toutes premières version d'ext-2, mais généralisée de nos jours) d'avoir, en plus de la liste des entrées, un arbre binaire par répertoire, afin d'accélérer la recherche dans les gros répertoires.

Une autre amélioration a été de découper le système de fichiers en groupes, au lieu d'avoir tous les inodes au départ puis toutes les données. Chaque groupe contient une copie du superbloc (redondance en cas de panne), une liste d'inodes et une liste de blocs de données. Autant que possible, les données d'un fichier sont créées dans le groupe contenant son inode, afin de limiter la fragmentation et de garder proches les données et l'inode (et donc, limiter les déplacement de la tête). Bien sûr, si un groupe est plein, les données seront créées ailleurs.

## 2.5 Les systèmes de fichiers journalisés

### 2.5.1 Principes du journal

Les systèmes de fichiers classiques nécessitent de mettre à jour plusieurs zones du disque pour une même opération. Par exemple, écrire à la fin d'un fichier nécessite d'écrire :
- Les données elles-mêmes ;
- La taille du fichier, dans le répertoire ou l'inode ;
- La date de dernière modification ;
- La FAT ou l'inode, pour indiquer le nouveau bloc alloué ;
- La bitmap ou liste des blocs libres.

Si l'opération est interrompue (par une coupure de courant, un plantage du système, ...) alors que certaines de ces opérations ont été effectuées mais pas d'autres, le système va être inconsistant, et les conséquences peuvent être graves.

L'idée d'un journal consiste à maintenir, sur le disque, une liste des opérations à effectuer, sous la forme d'un journal séquentiel. Ce journal est écrit avant de commencer à modifier les structures du système de fichier lui-même. Ainsi dans notre cas on aurait "écriture de 2346 octets à la fin du fichier 23, sur le bloc 456" dans le journal. Puis les opérations seront effectuées. Si jamais le système est interrompu, il peut alors savoir ce qu'il était entrain de faire, et corriger les incohérences. Une fois toutes les informations mises à jour, l'entrée du journal est marquée comme "commitée" et peut être effacée.

Le journal est en général de taille fixe et utilisé de manière circulaire.

### 2.5.2 Ext-3

Ext-3 est un exemple simple à comprendre, puisqu'il s'agit d'un journal ajouté à ext-2. Il peut fonctionner dans trois modes :

**Journal** Le premier mode, le plus fiable mais le plus lent, consiste à tout écrire, y compris les données, dans le journal avant de mettre à jour le système de fichiers lui-même.

**Ordered** Le mode "ordered" (mode par défaut) consiste à n'écrire que les modifications de métadonnées dans le journal (pas les données elles-mêmes). Cependant, l'entrée du journal n'est commitée (les métadonnées réelles ne sont modifiées) qu'une fois les données elles-mêmes écrites sur le disque.

**Writeback** Le mode "writeback" est le plus rapide, mais le moins fiable : seules les métadonnées sont journalisées, mais les données peuvent être écrites à n'importe quel moment, et il y a donc des risques de corruption.

### 2.5.3 Systèmes de fichiers organisés autour du journal

La plupart des systèmes de fichiers journalisés utilisent le journal comme sécurité en plus d'un système de fichier "classique", ce qui nécessite d'écrire les données deux fois.

Certains systèmes utilisent le journal lui-même pour stocker les données, en écrivant toutes les opérations de modification de manière linéaire.

Ces systèmes posent cependant deux problèmes : le premier est la gestion des entrées obsolètes du journal une fois le disque plein, et le deuxième est que les opérations de lecture sont souvent compliquées, puisqu'il faut reconstruire les transactions.

En pratique, ils ne sont utilisés que sur des supports particuliers, comme UDF sur les DVD et JFFS sur les cartes flash dans les Linux embarqués, où les opérations de réécriture sont dangereuses (sur les CD ou DVD ré-inscriptibles, comme sur les cartes flash, la durée de vie du support est principalement limitée par le nombre de réécriture que l'on peut faire de chaque secteur).

# Informations légales

Ce document est Copyright(c) Pilot Systems 2007. Il est disponible selon les termes de la GNU General Public License, version 3 ou supérieure.

La version PDF et le code source LaTeX sont disponibles sur `http://insia.pilotsystems.net`.