# Fichiers

## LIFASR5 - Systèmes d'exploitation

L. Gonnord, N. Louvet

4 février 2022

# Introduction

Outre le stockage des données et des programmes, le système doit fournir plusieurs propriétés aux fichiers :

- indépendance vis-à-vis des médias de stockage,
- gestion automatique de l'espace disponible,
- permettre un accès le plus rapide possible aux données,
- protection des données contre l'accès concurrent (verrous),
- notion d'ouverture et fermeture de fichiers pour gérer les ressources (ex : démontage interdit d'une clé usb),
- protection des données privées (droits),
- en plus, archivage, sauvegarde, journalisation...

# Plan

# Plan

# Types de fichiers

Sous Unix, de façon à uniformiser les traitements, « tout est fichier ».

On a quand même quatre *types* :

- *Fichiers « réguliers » :*
  données ou programmes des utilisateurs.
- *Répertoires :*
  fichiers contenant une liste d'autres fichiers et permettant d'organiser l'ensemble des fichiers du système sur un mode hiérarchique.
- *Fichiers spéciaux de type caractères :*
  - périphériques d'entré/sortie de type caractère (terminaux),
  - fichiers de communication (pipes ou sockets).
- *Fichiers spéciaux de type bloc :*
  périphériques d'entrée/sortie accessibles par blocs (disques durs).

Différentes conventions de *typage* existent pour présicer le type (exécutable, texte, image, . . . ) du contenu des fichiers réguliers.

- *Typage fort* : le fichier a un type défini par son nom (*extension*). C'est le cas par exemple sous DOS ou Windows.
  - Un fichier exécutable doit se terminer par `.exe`, `.com` ou `.bin`.
  - Le système reconnaît le logiciel à utiliser en fonction de l'extension.
- *Typage déduit* : le type du fichier dépend de son contenu ou de ses propriétés. C'est le cas sous Unix/Linux.
  - Un fichier est présumé exécutable s'il a le droit d'exécution.
  - On utilise souvent un code ou des directives placées en début de fichier.
  - Voir la commande `file` sous Unix.
- *Typage MIME* ou *Content-Type* : typage des données sur internet.
  - Les pages web ou les emails (pièces jointes) utilisent le type MIME.
  - Le navigateur ou le logiciel de lecture choisit le logiciel à appeler.

Chaque type de fichier présente une organisation interne qui lui est propre, pour pouvoir représenter un certain genre de données : on parle de *format*.

Quelques exemples :

- *Fichiers texte* :
  - choix d'un encodage pour les caractères accentués ou spéciaux (`UTF8` ou `iso8859-1` ; commande `iconv`).
  - Lignes terminées par des caractères spéciaux : Carriage Return (CR, `'\r'`), Line Feed (LF, `'\n'`), ou CRLF (les deux à la suite).
- *Fichier exécutable ELF* (sous Unix/Linux)
  - Une entête décrit la position des différentes parties du fichier, le sens du codage (little/big-endian), l'architecture du processeur...
  - Des sections (données `.data`, constantes `.rodata`, code `.text`, la table des symboles `.symtab`...)
- Chaque *Système de Gestion de Base de Données* (SGBD) utilise des formats qui lui sont propre pour stocker efficacement les données.

# Répertoires

## Définition

Un répertoire (ou catalogue) est un fichier dont le rôle est d'organiser l'ensemble des fichiers :

- c'est une liste de fichiers.
- un répertoire peut faire partie d'un autre, d'où une structure arborescente.
- deux répertoires particuliers : . et ..

- Un même répertoire ne peut faire partie que d'*un seul autre*.
- Un fichier peut faire partie de plusieurs répertoires : *lien en dur*.
- Possibilité de fichiers qui sont des pointeurs, les *liens symboliques*

# Appels systèmes (POSIX)

On se concentre sur les opérations de base sur les fichiers réguliers mais :

- il existe d'autres appels pour manipuler les fichiers réguliers,
- il existe d'autres fichiers que les fichiers réguliers !

Un fichier est manipulé à l'aide d'un entier appelé *descripteur de fichier* : il s'agit de son indice dans la table des fichiers ouverts du processus.

Tout processus dispose des descripteurs de fichiers suivants :

- 0, `STDIN_FILENO` : son entrée standard ;
- 1, `STDOUT_FILENO` : sa sortie standard ;
- 2, `STDERR_FILENO` : sa sortie d'erreur standard.

On va voir comment en ouvrir d'autres.

- **Ouverture ou création d'un fichier régulier :**

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

- `pathname` est le chemin du fichier à ouvrir ou à créer.
- `flag` détermine le mode d'ouverture du fichier ; il s'agit d'un « ou-bit-à-bit » entre différentes constantes :
  - `flag` doit inclure soit `O_RDONLY`, soit `O_WRONLY` soit `O_RDWR` ;
  - peut inclure `O_CREATE`, `O_APPEND`, `O_TRUNC`, ...
- En cas d'échec, -1 est retourné, et `errno` contient un code d'erreur.
- En cas de succès, l'appel retourne un *descripteur de fichier* qui est un entier que l'on va pouvoir utiliser par la suite pour manipuler le fichier.

- **Ecriture via un descripteur de fichier :**

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

  - `size_t` est un type entier non-signé, et `ssize_t` son homologue signé.
  - L'appel tente d'écrire *au plus* `count` octets à partir de l'adresse `buf` sur le descripteur de fichier `fd`, et retourne `nbwr`.
  - En cas de succès, `nbwr` $> 0$ (mais on peut avoir `nbwr` $<$ `count`).
  - En cas d'échec, `nbwr` $= -1$, et `errno` contient un code d'erreur.
  - Si `nbwr` $= 0$, la situation dépend du code présent dans `errno` : dans certains cas on peut reprendre, dans d'autres il faut abandonner.

- **Lecture via un descripteur de fichier :**

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

  - L'appel tente de lire *au plus* `count` octets sur `fd` en les les rangeant à partir de l'adresse `buf`, et retourne `nbrd`.
  - En cas de succès, $\texttt{nbrd} > 0$ (mais on peut avoir $\texttt{nbrd} < \texttt{count}$).
  - En cas d'échec, $\texttt{nbrd} = -1$, et `errno` contient un code d'erreur.
  - Si $\texttt{nbrd} = 0$, à nouveau, ça dépend de `errno` ; mais pour un fichier régulier, cela indique souvent que l'on a atteint la fin du fichier.

- **Fermeture d'un descripteur de fichier :**

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

  - `fd` est le descripteur de fichier à fermer.
  - En cas de succès, 0 est retourné, et les ressources associées avec le descripteur de fichier ouvert sont libérées.
  - En cas d'échec, -1 est retourné, et `errno` contient un code d'erreur. Un échec peut indiquer que des erreurs se sont produites précédemment.

- **Remarque importante :** personne ne sait tout ça par cœur ( ?)

*Il faut être capable de retrouver ces infos dans les pages du manuel.*

**Exemple :** une commande cat, sans gérer les erreurs :c

```
#define LEN 16
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_RDONLY); // descripteur de fichier
  char buf[LEN];                    // servira ici de buffer

  int nbrd = read(fd, buf, LEN);
  while(nbrd > 0) {
    int nrem = nbrd; // nb. d'octets restant à écrire
    int nbwr = 0     // nb. d'octets déjà  écrits
    while(nrem > 0) {
      int t = write(STDOUT_FILENO, buf+nbwr, nrem);
      nrem -= t;
      nbwr += t;
    }
    nbrd = read(fd, buf, LEN);
  }
  close(fd);
  return 0;
}
```

Avec une gestion minimaliste des erreurs pour `read` et `write` :

```
int nbrd = read(fd, buf, LEN);
while(nbrd > 0) {
  int nrem = nbrd;
  int nbwr = 0;
  while(nrem > 0) {
    int t = write(STDOUT_FILENO, buf+nbwr, nrem);
    if(t < 0) {
      cerr << strerror(errno) << endl;
      return 1;
    }
    nrem -= t;
    nbwr += t;
  }
  nbrd = read(fd, buf, LEN);
}
if(nbrd < 0) {
  cerr << strerror(errno) << endl;
  return 1;
}
```

Il existe beaucoup d'autres appels POSIX pour l'accès aux fichiers :

- `fcntl()`, `lseek()`, `stat()`,
- pour des fichiers de communication entre processus : `dup()`, `socket()`,
- pour accéder aux répertoires : `opendir()`, `readdir()`, `scandir()`, ...

Nous reparlerons de certains en temps voulu.

Il existe des fonctions d'accès de plus haut niveau aux fichiers :

- Les types et fonctions de la bibliothèque C définis dans `stdio.h` : `FILE*`, `fopen()`, `fclose()`, `fread()`, `fwrite()`, `fprintf()`, `fscanf()`, ...
- Les objets de biliothèque C++ standard définies dans `iostream` : `cin`, `cout`, `cerr`, `operator<<`, `operator>>`, ...

Dans les deux cas, les fichiers sont manipulés comme des flux pour faciliter les opérations de lecture et d'écritures. Mais attention : *vous ne pouvez pas utiliser directement ces bibliothèques sur des descripteurs de fichier POSIX*.

# Plan

On parle de *système de fichiers* pour désigner la façon dont le stockage des fichiers est organisé sur un périphérique de mémoire secondaire.

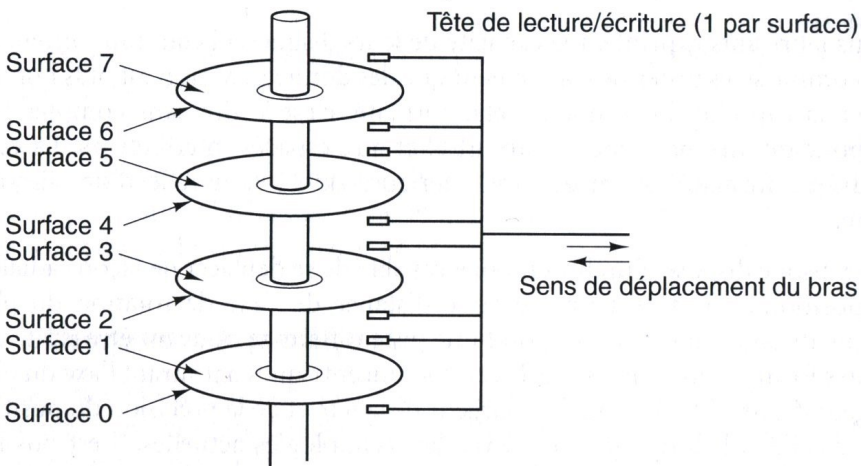
- Il existe différents types de mémoires secondaires :
  disque dur, mémoire flash (clé USB, carte SD, Solid Stat Drive),...
- De nombreux systèmes de fichiers ont été développés :
  NTFS, FAT, FAT16, FAT32, ext, ext2/3/4, zfs...
- Il existe aussi des systèmes de fichiers réseaux :
  SMB (Server Message Block), NFS (Network File System)...

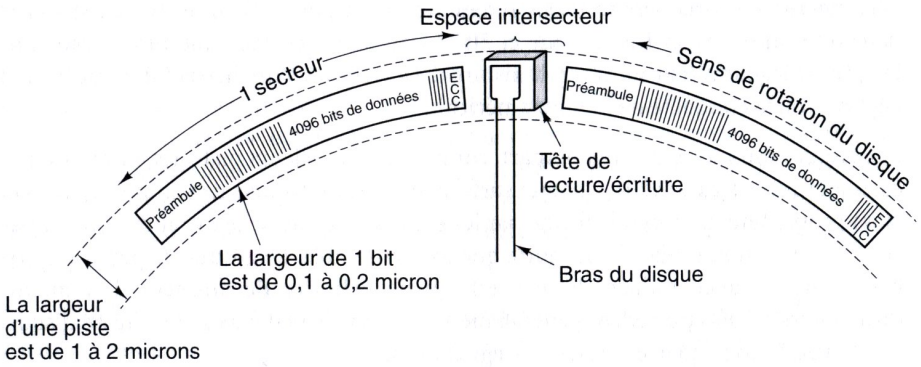
Dans la suite, on va

- surtout prendre le disque dur comme exemple de support,
- donner quelques idées sur les systèmes FAT et ext.

Pour les détails spécifiques à un système de fichiers, il n'y a pas de mystère : il faut aller lire les spécifications !

Les disques durs sont constitués de disques magnétiques rigides, équipés de têtes de lecture/écriture. La plus petite zone de mémoire accessible en une opération sur un disque dur est appelé secteur.




En général, le disque dur présente au système une interface :

- chaque secteur est adressable par un unique entier,
- il peut y a avoir une distinction entre secteurs physiques et logiques.

Mais on n'a pas besoin d'entrer dans ces détails.

En tout cas, cela correspond bien au fait que les disques durs sont des *périphériques accédés par blocs*.

Le *bloc* est l'abstraction, au niveau du système, du secteur (un bloc peut être composé de un ou plusieurs secteurs logiques) :

- les données sont lues ou écrites *par bloc* (`:D`),
- l'on ne peut pas lire ou écrire moins d'un bloc,
- la lecture de toutes les données d'un bloc est « rapide »,
- entre deux blocs accédés, le déplacement du bras de lecture et l'attente du passage du bon secteur est plus lent.

Comme on ne peut pas lire ou écrire moins d'un bloc, ce découpage a tendance à provoquer de la fragmentation interne : seulement une partie de bloc est utilisée par un petit fichier, d'où une perte de place.

## Exemple (Un disque dur)

```
Disque /dev/sda: 931,5GiB, 1000204886016 octets, 1953525168 secteurs
Unités: secteur de 1 * 512 = 512 octets
Taille de secteur (logique / physique) : 512 octets / 4096 octets
taille d'E/S (minimale / optimale) : 4095 octets / 4096 octets
```

$1000204886016 = 1953525168 \times 512$ : ça tombe bien !

Par contre, une clé USB ne contient aucune partie mécanique, pas de tête, pas de cylindre, pas de piste... Mais le système fait l'interface :

## Exemple (Une clef USB)

```
Disque /dev/sdb : 7,5GiB, 8011120640 octets, 15646720 secteurs
Unités : secteur de 1 * 512 = 512 octets
Taille de secteur (logique / physique) : 512 octets / 512 octets
taille d'E/S (minimale / optimale) : 512 octets / 512 octets
```

$8011120640 = 15646720 \times 512$ : ça colle toujours !

# Structure au niveau du fichier

**Objectifs :**

- allouer les blocs aux fichiers ;
- pouvoir retrouver les blocs dans le bon ordre ;
- la plupart des fichiers sont petits (quelques blocs) ;
- certains sont très gros.

**1er exemple, allocation contiguë.**

- les fichiers sont stockés en un seul morceau ;
- le répertoire ne contient que le numéro du premier bloc et la taille ;
- les accès sont très rapides ;
- cause une grosse fragmentation ;
- problème pour augmenter un fichier.

## 2ème exemple, organisation par listes chainées.

- Le répertoire ne contient que le premier bloc du fichier.
- Ce bloc renvoie au suivant . . .
- C'est le principe des système des fichiers utilisant une table FAT (*File Allocation Table*) :
  - il y a exactement une entrée dans la table par bloc du disque,
  - chaque entrée contient l'indice de l'entrée suivante, ou EOF.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | EOF | 13 | 2 | 9 | 8 | | 4 | 12 | 3 | | EOF | EOF | |

| Répertoire | | | | | |
|---|---|---|---|---|---|
| A | 6 | $\rightarrow$ 8 | $\rightarrow$ 4 | $\rightarrow$ 2 | $\rightarrow$ EOF |
| B | 10 | $\rightarrow$ 3 | $\rightarrow$ 13 | $\rightarrow$ EOF | |
| C | 5 | $\rightarrow$ 9 | $\rightarrow$ 12 | $\rightarrow$ EOF | |

**3ème exemple, organisation par inœud.**

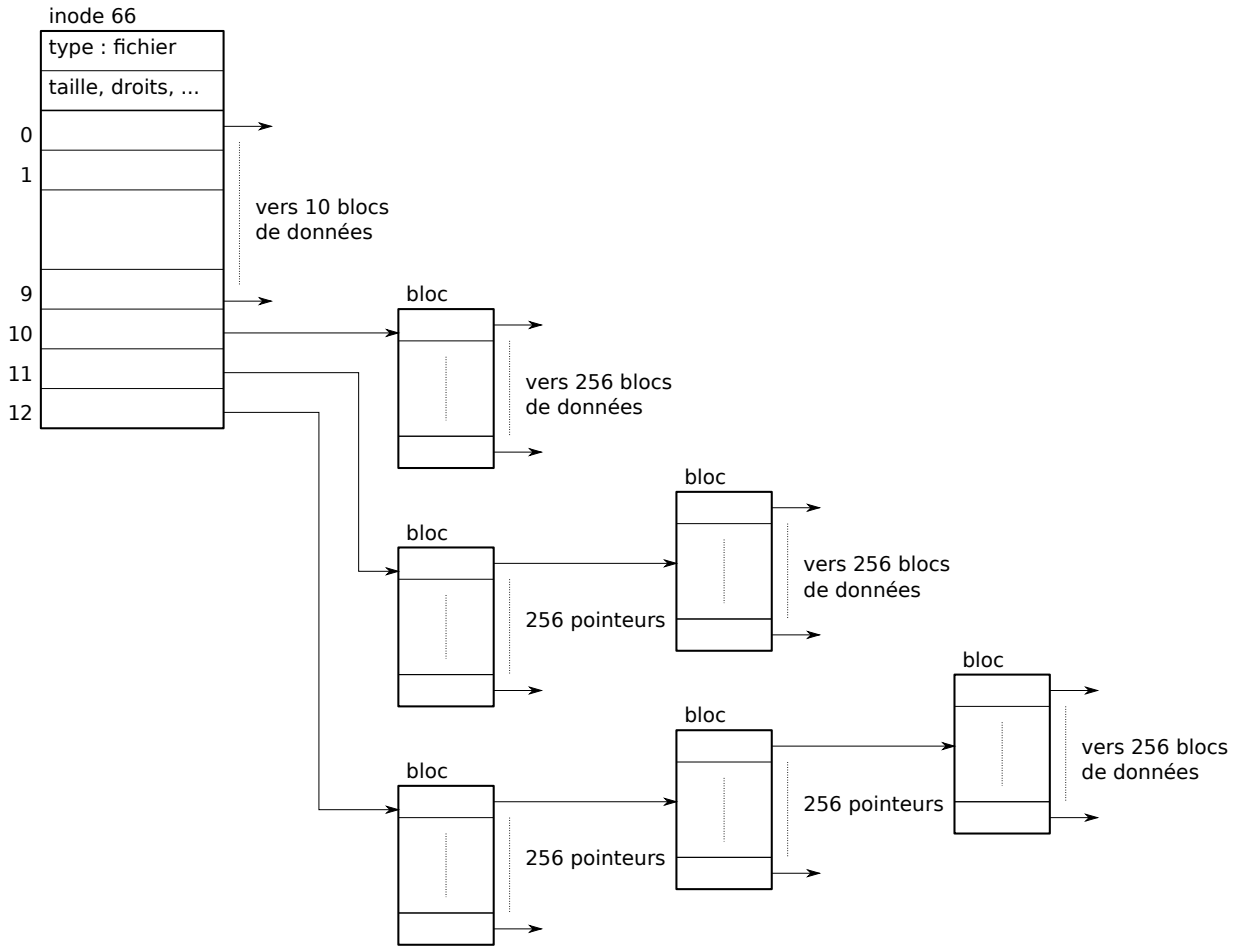
On utilise plusieurs niveaux d'indirections :

- « une table, qui pointe sur une table, . . ., qui pointe sur un bloc »
- comme beaucoup de fichiers sont petits, la capacité doit être variable.

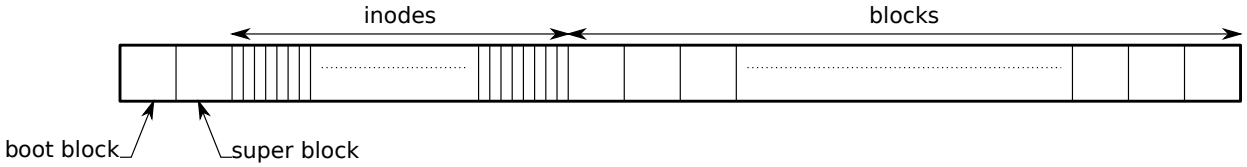
Supposons des blocs d'1 kio (soit 256 entiers de 32 bits).

- L'*inœud* contient 13 adresses de blocs.
- Les blocs d'indices 0 à 9 contiennent des données (au plus 10 kio).
- Le bloc d'indice 10 contient une table d'indirections simples :
  - 1 table qui pointe vers 256 blocs de données, soit 256 kio max.
- Le bloc d'indice 11 contient des indirections doubles :
  - 1 table, qui pointe vers 256 tables, qui pointent vers 256 blocs, soit $2^8 \times 2^8 \times 2^{10}$ o $= 64$ Mio max.
- le bloc d'indice 12 contient des indirections triples :
  - 1 table, qui pointe vers 256 tables, qui pointent vers 256 tables, qui pointent vers 256 blocs, soit 16 Gio max.

# Un bloc contient donc soit un fichier, soit une table d'indirections :




# Une table des inœuds est stockée sur le disque :

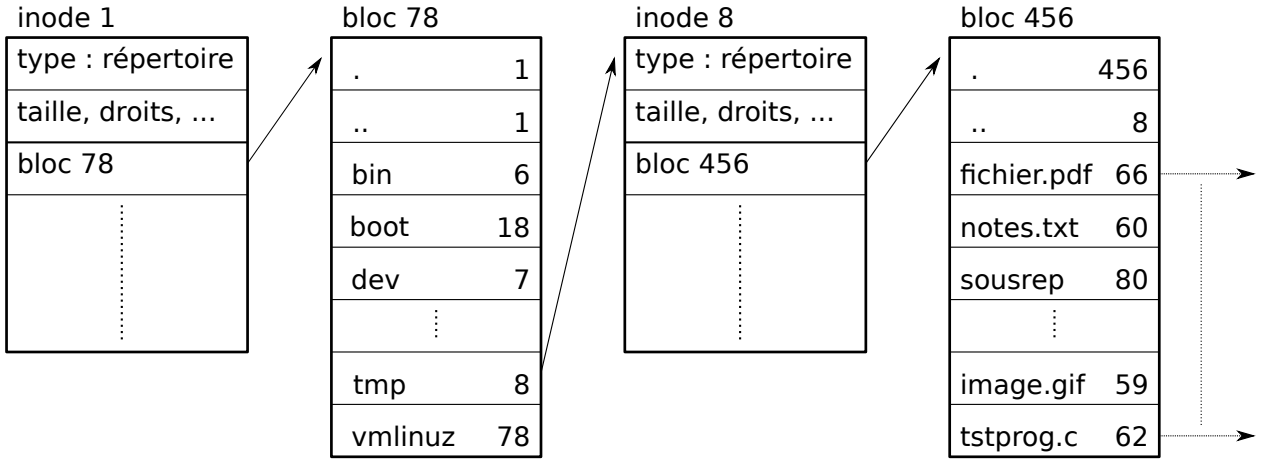


Outre les adresses des blocs, l'inœud contient :

- la taille du fichier en octets, et en nombre de blocs,
- le propriétaire, le groupe et les droits du fichier,
- les dates ctime, mtime, atime,
- le nombre de liens durs sur cet inœud.

*Il ne contient pas le nom du fichier*

Un fichier peut être : un fichier régulier, un répertoire, un fichier spécial . . .
Exemple avec des répertoires :




C'est le principe des systèmes de fichiers utilisés avec Linux : ext, ext2/3/4.

# Quelques détails

Les répertoires permettent de retrouver les fichiers en les référençant par un nom. Dans les systèmes

- à blocs contigus ou chainés (FAT) : nom $\rightarrow$ adresse du premier bloc.
- avec inœud : nom $\rightarrow$ numéro de l'inœud.

Le même inœud peut être référencé plusieurs fois : liens en dur.

Gestion des blocs : le système maintient une structure de données pour attribuer des blocs libres à un fichier, ou libérer les blocs d'un fichier supprimé. Une possibilité est l'utilisation d'une *bitmap* des blocs :

- chaque bit d'un vecteur correspond à un bloc,
- si un bloc est libre son bit vaut 1, 0 sinon.

# Gestion des droits

La plupart des systèmes doivent gérer plusieurs utilisateurs :
il faut donc de gérer leurs différents droits.

- Un utilisateur standard a le droit
  - d'utiliser les logiciels,
  - d'utiliser son espace de stockage (compte / répertoire personnel),
  - de lire les données partagées.
- Certains utilisateurs particuliers servent à
  - limiter les droits des serveurs (ex. apache),
  - gérer des accès distants (administration à distance),
  - avoir des configurations particulières (ex : oracle).
- Un utilisateur spécial a tous les droits :
  - l'*administrateur* sous Windows,
  - *root* sous Unix.

# Gestion des droits Unix

- Les droits sont les droits sur les fichiers (tout est fichier).
- Pour un fichier un utilisateur est dans l'une des classes :
  - `user, u` : propriétaire ;
  - `group, g` : groupe du propriétaire ;
  - `other, o` : tous les autres.
- Les droits de base sont :
  - `read` lecture du fichier, liste du contenu du répertoire ;
  - `write` écriture dans le fichier,
    ajout/suppression de fichier dans le répertoire ;
  - `execute` exécution du fichier,
    aller dans le répertoire *ou un sous répertoire*.
- `root` a toujours le droit
- Tout processus a un propriétaire égal à :
  - celui qui a lancé la commande (`SetUID bit = 0`) ;
  - celui à qui appartient la commande (`SetUID bit = 1`).

**Exemple.**

- Pour mettre en place sa page internet personnelle, il faut que l'utilisateur *apache* ou *html* ait le droit de lire le contenu du répertoire `~/public_html/` donc :
  - `~/` doit être autorisé en exécution pour les autres.
  - `~/public_html/` doit être autorisé en exécution et lecture pour les autres.
- Les mots de passe doivent être protégés . Mais la commande `password` doit permettre de lire et modifier son mot de passe :
  - `/etc/shadow` est en lecture uniquement pour son propriétaire `root`
  - `/usr/bin/passwd` appartient à `root`, est autorisé en exécution pour tous avec un `setUID bit` $= 1$.
- Les droits permettent de protéger le système tout en délégant des droits étendus via certaines commandes.

# Les listes de contrôle d'accès (ACL)

Le système de droits n'est pas suffisant :

- Il n'y a pas de droits négatifs (« tout sauf ») :
  - par exemple avec Apache, les accès sont basés sur `allow` et `deny` et un ordre de lecture des droits
- Seulement 3 types d'utilisateurs. . .
  - fastidieux, pour gérer finement les droits :
    les utilisateurs doivent être dans de nombreux groupes
  - quand un utilisateur crèe un fichier, à quel groupe appartient-il ?
- Une solution est d'associer à chaque objet une liste de droits (ou déni de droits) accordés à des utilisateurs ou des groupes. Ce sont les *Access Control List* ou *ACL*.

- Une ACL est une liste d'*ACE* (*AC Entries*)
- Les droits sont positifs ou négatifs. Une ACE est formé :
  - d'un droit particulier (lecture, écriture, changer les droits...) ;
  - d'un utilisateur ou d'un groupe ;
  - d'un objet sujet ;
  - d'un booléen `Allow` ou `Deny`.
- Exemple :
  - Windows (droits de base, droit sur NTFS),
  - LDAP, firewall, Andrew File System (AFS), ...
- Commandes Unix ACL : voir par exemple l'excellente page `http://bjobard.perso.univ-pau.fr/Cours/ISE/TP5.html`
  - commande `setfacl` pour fixer les droits
    ex : `setfacl -m u:homer:rw devoir.tex`
  - commande `getfacl` pour avoir des infos

Par exemple, dans un serveur LDAP, la syntaxe des ACE ressemble à :

```
olcAccess: to [ressource]
  by [à  qui] [type d'accès autorisé]
  by [à  qui] [type d'accès autorisé]
  by [à  qui] [type d'accès autorisé]
```

Un exemple de base, qui limite l'accès à l'attribut `userPassword` :

```
olcAccess: to attrs=userPassword
 by self write
 by group.exact="cn=adm,ou=groups,dc=example,dc=com" write
 by anonymous auth -- (seulmt auth possible)
 by * none
```

Pour gérer les accès sur un objet de l'arborescence :

```
olcAccess: to dn.subtree="ou=applications,dc=example,dc=com"
 by group.exact="cn=adm,ou=groups,dc=example,dc=com" write
 by users read
 by * none
```

# Plan

Ce dont on a peu ou pas parlé...

- Besoin des utilisateurs :
  - *sécurité des données* (détection et correction d'erreurs) ;
  - *confidentialité* (chiffrement des disques, droits d'accès) ;
  - *sauvegardes* (incrémentales, clichés).
- Au niveau du matériel :
  - support à mémoire flash, avec *répartition des écritures* ;
  - *RAID* (*Redundant Array of Independent Disks*) ;
  - disques de plus en plus gros ($\gg$ To).
- *Systèmes de fichier en réseau* :
  - Un (ou des) serveurs partagent leurs fichiers.
  - Le système client présente les fichiers comme des fichiers locaux : les applications ne font pas la différence
    - ★ Exemples « historiques » : NFSv3/4 (Unix), SMB (Windows).
    - ★ Autre systèmes : Lustre, GFS (RedHat), GoogleFS, OCFS (Oracle).

- *Journalisation* : chaque modification est d'abord écrite dans un journal, puis oubliée quand elle est effectuée ; le système garde sa cohérence (ex : ext3, NTFS, HFS+).
- *Pré-allocation de zone continue* : lors d'une écriture, une zone plus grande est allouée ; si le fichier est agrandi, il utilise cette zone ; cela évite la fragmentation (ex : ext4, NTFS, HFS+, Btrfs).
- *Vérification et défragmentation en ligne* : ces opérations sont faites durant l'utilisation normale ; permet la remise en route rapide.
- *Partionnement ou Logical Volume management (LVM)* :
  - partitionnement : découpage prévu en matériel d'un disque en plusieurs partitions ;
  - LVM : technologie permettant d'assouplir les schémas de partionnement usuels.