Systèmes d'exploitation : le système de gestion de fichiers (File System) (point de vue utilisateur)

Licence MIAGE — Université Lille 1
Pour toutes remarques : Alexandre.Sedoglavic@univ-lille1.fr

Semestre 6 — 2012-2013

Qu'est ce qu'un système de gestion de fichiers ?

La mémoire persistante répond à plusieurs besoins :

- conserver des données au delà de la durée de vie d'un processus ;
- mémoriser une quantité de données supérieure aux capacités de la mémoire vive ;
- partager des données.

Le système de gestion de fichiers est un ensemble de structures de données et de procédures les manipulant qui doit assurer :

- la définition et la manipulation de l'abstraction *fichier* ;
- l'organisation logique de ces abstractions (hiérarchie arborescence basée sur la notion de *répertoire*) ;
- le liens entre cette abstraction et son implantation matériel ;
- la pérénité des informations stockées (confidentialité, tolérance aux pannes, robustesse, etc).

Notion de fichier et informations utilisateur relatives

Un fichier est considéré comme un flux linéaire d'octets.

Aucune information sur l'organisation de l'espace du support à ce niveau d'abstraction. Pour manipuler les fichiers, il faut juste pouvoir les identifier par leurs caractéristiques :

- nom, type, taille du fichier ;
- propriétaire du fichier ;
- date de création, date de dernière modification ;
- protection : qui a droit de le lire et de le manipuler ;

sans s'occuper de l'implantation de ces dernières.

Dans un shell de type UNIX la commande `ls -al nom_de_fichier` permet d'obtenir ces informations :

```
% ls -al Juin2005.tex
-rw-r--r--    1 sedoglav calforme  0 Aug 19 05:09 Juin2005.tex
```

Ces informations correspondent à :

droits, nombre de liens symbolique sur ce fichier, propriétaire, son groupe, taille, date de création, identificateur.

Dans les OS dérivés d'UNIX, le codage des droits se fait sur 10 lettres qui sont dans l'ordre :

- le type du fichier (`d` pour répertoire, `l` pour un lien, `c` et `b` pour un périphérique, `p` pour un tube, `-` pour un fichier classique) ;
- **r** le fichier est lisible par le propriétaire (- dans le cas contraire) ;
- **w** le fichier est modifiable par le propriétaire (- sinon) ;
- **x** le fichier est exécutable par le propriétaire (- sinon).

Le groupe suivant de 3 lettres reprend le même principe mais définit les droits pour les membres du groupe auquel appartient le propriétaire.
Le dernier groupe reprend le même principe mais concernant les autres utilisateurs.

Ainsi le fichier `Juin2005.tex` n'est pas un répertoire, il n'est exécutable par personne, il est lisible par tout le monde et n'est modifiable que par son propriétaire.

Format de fichiers et fichiers spéciaux

Le format d'un fichier est la signification que l'utilisateur donne à la suite d'octets le constituant (pour le FS, tout fichier n'est qu'une suite d'octets, seul le traitement diffère). Les types de fichiers sont :

- les fichiers ordinaires non exécutables :
  - fichiers textes dont les octets codent des caractères ASCII, ISO, unicode ou tout autre standard ;
  - fichiers binaires qui ne sont pas censés être décodé par un format du type ci-dessus mais par une application utilisateur.
- les fichiers ordinaires exécutables que l'OS peut interpréter (qui commencent par `#!/acces/interpreteur`) ou exécuter directement au niveau du microprocesseur (format ELF, etc.) ;
- les fichiers spéciaux associés aux périphériques ou aux processus ;
- les *répertoires* : ces fichiers définissent les chemins d'accès aux fichiers. Les *liens* permettent le partage de fichiers sans duplication.

Le format est souvent définit par un postfixe accolé au nom du fichier après un point (les exécutables nécessitent un droit d'exécution).

Structure d'un fichier exécutable : Exécutable Linkable Format

Un fichier de ce type contient :

- une entête permettant de localiser les éléments du fichier :
  - *e_ident* le fichier commence par les trois caractères ELF ;
  - *e_entry* l'adresse du début du code du programme (qui est généralement noyée dans la masse d'instruction) ;
  - *e_phoff* l'offset ou commence le code ;
  - *e_phentsize*
  - *e_phnum* le nombre d'entête physique (une pour le code, une pour les données, etc.) ; etc.
- pour chaque segment (code, données), ELF fournit une entête contenant :
  - *p_filesz* la taille du segment en octets ;
  - *p_vaddr* l'adresse virtuelle ou le code doit être chargé ;
  - *p_flag* les droits de ce segment (exécution, lecture, écriture). etc.

Les segments commencent juste après les entêtes.

Fichiers spéciaux : les fichiers d'entrée-sortie

Un fichier n'est pas seulement un paquet d'octets stocké sur le disque. Certains fichiers servent d'abstraction aux accès des périphériques d'entrée-sortie et seul l'OS devraient les manipuler. Ainsi, puisque les périphériques sont spécifiques à chaque matériel, on se sert de la notion de fichier pour standardiser leurs accès.
Un des avantages est de disposer des mesures de protections implantées par le FS (droits d'accès, etc).
Il existe deux type de fichiers périphériques :

- **bloc** dont l'unité d'échange est le bloc (b dans les droits) et
- **caractères** dont l'atome est l'octet (c dans les droits).

Dans les systèmes de type UNIX, ces fichiers se trouvent dans le répertoire `/dev` :

- `/dev/null` est utilisé pour supprimer des flux ;
- `/dev/random` est un générateur physique d'octets aléatoires ;
- `/dev/mem` donne accès à la mémoire vive physique, etc.

Appels système associés aux droits d'accès

La commande externe `kpseaccess` d'un shell permet la vérification des droits associés à un fichier :

```
%  ls -l Cours.tex
-rw-r--r--  1 sedoglav users 38145 Jan 15 20:14 Cours.tex
% kpseaccess -rw Cours.tex ; echo $?
0
% kpseaccess -x Cours.tex ; echo $?
1
```

De même, la commande externe `chmod` d'un shell permet de changer les droits d'un fichier (cf. `man -S2 chmod`).
Ces commandes externes du shell (i.e. fichiers sur le disque exécutables par cet interpréteur) utilisent des appels système :

```
#include <unistd.h>
int access(const char *pathname, int mode);
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

Un *appel système* est une fonction fournie par le noyau d'un OS et utilisé par les processus dans l'espace utilisateur (distincts du noyau). Un appel système repose sur une *interruption* matériel du microprocesseur.

Un début d'implantation d'une commande shell équivalente

En séance de travaux pratiques, nous écrirons ce genre de code :

```
#include<stdio.h>
#include<unistd.h>
#include <errno.h> /* pour d\'efinir EINVAL */
int main(int argc, char *argv[]){
  int mode ;
  switch(argv[1][1]){
    case 'r' : mode = R_OK ; break ;
    case 'w' : mode = W_OK ; break ;
    case 'x' : mode = X_OK ; break ;
    default : { printf("access: Invalid MODE") ;
                return EINVAL ;                  }
  }
  return access(argv[2],mode) ; /* implique un appel au syst\'eme */
}
```

afin de comprendre le fonctionnement de l'OS. (Ce code est incomplet car un seul appel à `access` permet plusieurs tests simultanés (rwx) à la fois. Les `?_OK` sont des bits que l'on peut associés.)

De l'intérêt de lire les fichiers d'entête

On trouve dans grâce au fichier `unistd.h` la plupart des informations nécessaires à l'utilisation de cet appel :

```
/* Values for the second argument to access.
   These may be OR'd together.  */
#define R_OK    4               /* Test for read permission.  */
#define W_OK    2               /* Test for write permission.  */
#define X_OK    1               /* Test for execute permission.  */
#define F_OK    0               /* Test for existence.  */

/* Test for access to NAME using the real UID and real GID.  */
extern int access(__const char*__name,int __type) __THROW __nonnull (
```

Plus généralement, les fichiers d'entête implantent la famille de standards *Portable Operating System Interface* (POSIX IEEE 1003— avec un X pour UNIX :-). Ces standards forment une interface de programmation (ensemble de structure de données et de fonctions les manipulants mise à disposition par l'OS).

Prérequis

Afin de pouvoir gérer les fichiers, plusieurs tables sont maintenues en relation avec l'OS.

Gestion dans le noyau : il existe une table des fichiers ouverts par l'ensemble des processus et contenant :

- le déplacement (*offset*) courant dans le fichier ;
- un mode d'ouverture (lecture, lecture/écriture, etc).

Gestion d'un fichier dans un processus : il existe une table — dite des descripteurs — associées à chaque processus :

- un descripteur est un entier identifiant unique d'une *ouverture* de fichier dans le processus ;
- un même fichier peut être ouvert plusieurs fois par un seul processus et/ou par des processus différents ;
- le *descripteur* d'un fichier est son index dans la table des descripteurs du processus ;
- pointe dans la table des fichiers ouverts du noyau.

Descripteurs de fichier

Lorsqu'un processus doit manipuler un fichier, il le désigne par un entier appelé descripteur de fichier.
Il s'agit d'une clef dans une table dont l'entrée est une structure contenant notamment l'*inœud*[1] d'un fichier.
L'association de ce descripteur avec l'*inombre*[2] désignant le fichier se fait par lors de l'appel `open()`.
Chaque processus UNIX dispose de 20 descripteurs de fichiers,
Par convention, les trois premiers[3] sont toujours ouverts au début de vie du processus :

- 0 est l'entrée standard (clavier) ;
- 1 est la sortie standard (écran) ;
- 2 est la sortie erreur standard (écran aussi).

1. cf. seconde partie du cours sur les FS.
2. ibidem.
3. Bien qu'ils s'agissent de périphérique, ce sont bien des fichiers (cf. la suite).

Exemple d'entrées-sorties par appels système

Les appels système de manipulation de fichier les plus utilisés sont :

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

Leurs déclaration se trouvent dans `<fcntl.h>`

L'appel système

```
int open(char *name, int mode <optionel>, int perm</optionel>)
```

permet d'ouvrir un fichier dont le chemin d'accès est `name` suivant le mode et les permissions spécifiés.
Cette fonction retourne le descripteur correspondant et -1 en cas d'erreur.

L'appel `int close(int fd)` ferme le fichier associé au descripteur `fd`.
Cet appel retourne 0 si l'opération est un succès et -1 sinon.

Paramètres de la fonction open

- le chemin d'accès `name` peut être relatif ou absolu ;
- `perm` est un entier représentant les permissions du fichier (en octal à la UNIX) et n'est utilisé qu'en création ;
- `mode` est un entier formant un *drapeaux* — bit à bit — de lecture/écriture :

`O_RDONLY` : ouverture en lecture seule ;
`O_WRONLY` : ouverture en écriture seule ;
`O_RDWR` : ouverture en lecture/écriture ;
`O_APPEND` : positionne l'offset à la fin du fichier avant *chaque* écriture ;
`O_CREAT` : crée le fichier s'il n'existe pas ;
`O_EXCL` : en combinaison avec `O_CREAT`, provoque une erreur si le fichier existait ;
`O_TRUNC` : si le fichier existe à l'ouverture, il est tronqué à 0 caractères ;
`O_NONBLOCK` : ouverture non-bloquante (pour pipes et fichiers spéciaux).

Ces drapeaux se combinent par un *et* bit à bit, par exemple :

```
O_WRONLY | O_CREAT | O_TRUNC
```

Le fichier d'entête `unistd.h` fournit les prototypes des fonctions suivantes :

- `ssize_t read(int fd, void *buf, size_t nbyte)` essaie de lire `nbyte` octets, à partir de l'offset courant, dans le fichier associé au descripteur `fd` et stocke les octets lus dans `buf`. La valeur retournée est le nombre d'octets lus : 0 en fin de fichier, -1 en cas d'erreur. Le nombre d'octets lus peut être inférieur à `nbyte`, si la fin du fichier est atteinte en cours de lecture.
- `ssize_t write(int fd,const void *buf,size_t nbyte)` essaie d'écrire `nbyte` octets provenant de `buf` dans le fichier associé au descripteur `fd` à partir de l'offset courant. La valeur retournée est le nombre d'octets écrits, et -1 en cas d'erreur. Le nombre d'octets effectivement écrits peut être inférieur à `nbyte` (si le disque est plein par exemple).

L'appel `off_t lseek(int fd, off_t offset, int whence)` déplace l'offset courant du fichier associé au descripteur `fd` sans lire ni écrire. `offset` (entier long) donne le nombre d'octets à sauter.

Le paramètre `whence` permet de donner une origine :

- `SEEK_SET` : par rapport au début du fichier ;
- `SEEK_CUR` : par rapport à l'offset courant ;
- `SEEK_END` : par rapport à la fin du fichier.

Il est possible de dépasser la fin du fichier (fichier creux).

Librairie standard : une couche de plus

En en se basant sur les appels système, il est possible d'ajouter une couche supplémentaire de stockage dans la gestion des entrée – sortie (ce niveau est géré au niveau du processus).
Une *librairie standard* est une collection normalisée de structure de données et de routines les manipulants qui permettent d'implanter des opérations courantes (hors du noyau). En conséquence :

- on peut avoir une lecture/écriture par bloc dans un tampon (zone mémoire intermédiaire) ;
- il y a moins d'appels système pour des accès sur de petites zones (sachant qu'un tel appels est *coûteux* ;
- et de vidage des tampons s'il le faut.

Pour ce faire, on utilise un identificateur d'ouverture de fichier (flot) : de type `FILE *` (pointeur sur une structure de ce nom). Cette structure est décrite dans les fichiers d'entête.

On décrit dans la suite quelques fonctions de la librairie C correspondantes (ce ne sont pas des appels au système mais des fonctions qui nécessitent une édition de liens et utilisent de tels appels). Ainsi la fonction

```
#include <stdio.h>
 FILE *fopen(const char *name, const char *mode);
```

ouvre le fichier dont le chemin d'accès est donné par `name`. Le mode d'ouverture est spécifié par `mode` :

- `"r"` : ouverture en lecture seule ;
- `"w"` : ouverture en écriture seule. Création éventuelle du fichier. Efface le contenu si le fichier existe ;
- `"a"` : ouverture en mode ajout. Création éventuelle du fichier. Positionnement en fin de fichier si il existe ;
- `"r+"` : ouverture en lecture/écriture. Positionnement en début de fichier ;
- `"w+"` : ouverture en lecture/écriture avec création éventuelle. Efface le contenu si le fichier existe ;
- `"a+"` : ouverture en mode mise à jour avec création éventuelle. Positionnement en fin de fichier. Renvoie un pointeur sur le flot, ou `NULL` si échec.

- `int fflush(FILE *stream)` procède au vidage des buffers associés au flot de sortie `stream`. Son comportement est indéterminé si `stream` est un flot d'entrée. Elle retourne 0 en cas de succès, `EOF` sinon.
- `size_t fread(void *ptr, size_t size,size_t nitems, FILE *stream)` place dans le tableau pointé par `ptr` jusqu'à `nitems` éléments lus sur le flot pointé par `stream`. La taille d'un item est spécifiée par `size`. Retourne le nombre d'éléments lus.
- `size_t fwrite(void *ptr, size_t size, size_t nitems, FILE *stream)` écrit à partir du tableau pointé par `ptr` jusqu'à `nitems` éléments sur le flot pointé par `stream`. La taille d'un item est spécifiée par `size`. Retourne le nombre d'éléments écrits.

- `int fclose(FILE *stream)` ferme le fichier associé au flot `stream` et vide les tampons. Cette fonction renvoie 0 en cas de succès, `EOF` sinon.
- `FILE *freopen(const char *name, const char *mode, FILE *stream)` ; ouvre le fichier de chemin d'accès `name` dans le mode spécifié par `mode`, et lui associe le flot pointé par `stream`. Le fichier associé à `stream` est préalablement fermé. Cette fonction retourne `stream` en cas de succès, `NULL` sinon.
- `int remove(const char *name)` détruit le fichier de chemin d'accès `name`. Renvoie 0 en cas de succés, une valeur non nulle sinon.
- `int rename(const char *old, const char *new)` renomme le fichier de nom `old` en `new`. Renvoie 0 en cas de succés, une valeur non nulle sinon.

Écriture avec format dans un fichier

`int fprintf(FILE *stream, const char *format, ...)` écrit sur le flot pointé par `stream` au format spécifié par la chaîne `format`. `format` peut contenir des caractères ordinaires, recopiés tels quels, et des spécifications de conversion.

L'instruction `printf` est dérivée de `fprintf` en passant le flot prédéfini `stdout` associé à la sortie standard en paramètre.

Ces spécifications utilisent un ou plusieurs des arguments passés à la suite de `format`. Une spécification débute par un % suivi de :

- drapeaux de remplissage/justification :
  - `-` : justification à gauche ;
  - `+` : impression systématique du signe ;
  - `0` : remplit le début du champ avec des zéros ;
- un nombre donnant la largeur minimum du champ ;
- un caractère `.` séparateur ;
- un nombre donnant la précision ;
- une lettre : `h` pour un `short`, `l` pour un `long`, `L` pour un `long double` ;
- un caractère indiquant le type de conversion.

La précision ou la largeur minimum peuvent être remplacées par un astérisque (*) : leur valeur sera alors prise dans la liste des arguments. Seul le dernier caractère de conversion est obligatoire :

- `d`, `i` : `int` en notation décimale signée ;
- `x`, `X` (`o`) : `int` en notation hexadécimale (octale) non signée ;
- `u` : `int` en notation décimale non signée ;
- `c` : `int` converti en caractère non signé ;
- `f` : `double` en notation décimale signée (*dd.ddd*) ;
- `e`, `E` : `double` en notation scientifique signée ($d.ddd\text{e}\pm dd$) ;
- `p` : `void *` en format pointeur (hexa. en général).

Lecture formatée depuis un fichier

`int fscanf(FILE *stream, const char * format, ...)` lit sur le flot pointé par `stream` au format spécifié par la chaîne `format`. `format` peut contenir des caractères ordinaires, lus comme tels dans `stream`,ou des spécifications de conversion. Les résultats des conversions sont stockés dans les variables pointées par les arguments suivant `format`. `fscanf` reconnaît toujours la plus longue chaîne correspondant à `format`. Une spécification débute par un % suivi de :

- `*` : supprime l'affectation ;
- un nombre donnant la largeur maximum du champ ;
- une lettre : `h`, `l` ou `L` (idem `fprintf`) ;
- un caractère indiquant le type de la conversion.

Seul le dernier caractère de conversion est obligatoire :

- `d` (`i`) : entier sous forme décimale (ou octale ou hexa.) — `int *` ;
- `o` : entier sous forme octale — `int *` ;
- `x` : entier sous forme hexadécimale — `int *` ;
- `u` : entier non signé sous forme décimale — `unsigned int *` ;
- `c` : caractère (espacement compris) — `char` ;
- `s` : chaîne de caractères — `char *` — (espacement supprimé au début) qui doit être assez grand pour contenir le résultat ;
- `f`, `e` :] nombre en virgule flottante – `float *` ;
- `p` : pointeur–`void *` ;
- `[..]` : plus longue chaîne composée de caractères placés entre `[]`–`char *` ;
- `[..]` : plus longue chaîne composée de caractères ne faisant pas partie de l'ensemble entre `[]`–`char *`.

Exemple d'utilisation

```
#include <stdio.h>
#include <errno.h>
int
main
(void)
{
   FILE *fd = fopen("fichierquinexistepas","r") ;
   if (fd==NULL){
     perror("L'erreur suivante est survenue") ;
     return -1 ;   }

   for(i=0; i<Max; i++)
      fprintf(fd,"%d\n",tab[i]) ;

   fclose(fd) ; /* fclose ferme le flot */
   return 0 ;
}
```

Organisation des fichiers en arbre

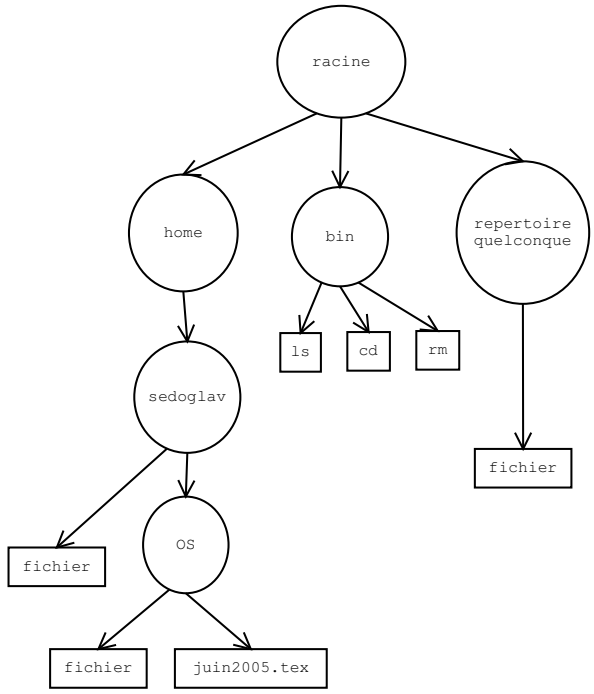
La communauté des fichiers est organisée en arbre i.e. en un ensemble de nœuds reliés par des arêtes orientées (sans cycle) : chaque nœud a exactement une arête pointant vers lui (à l'exception de la racine qui est un nœud sans prédécesseur). Les feuilles sont les noeuds sans successeur.

Les feuilles correspondent aux fichiers et les autres nœuds sont des *répertoires*. On peut ainsi définir un chemin d'accès à un fichier :

- absolu : depuis la racine ;
- relatif : notion de répertoire courant.

Le fichier `juin2005.tex` est localisé par le chemin d'accès

```
racine -> home -> sedoglav -> OS ->
```



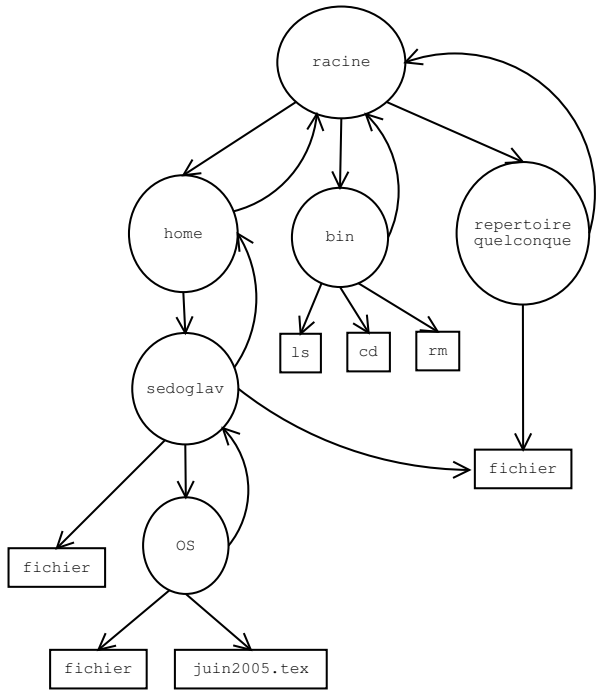
Les répertoires sont des fichiers (flux linéaire d'octets) contenant l'information liée aux arêtes les quittant.

Organisation en graphe

Ce type de représentation de la communauté des fichiers est assoupli en autorisant plusieurs arêtes à pointer sur un même élément et en faisant pointer chaque répertoire sur son prédécesseur.

On obtient ainsi un graphe qui permet :

- de remonter l'arborescence depuis n'importe quel répertoire sans repartir systématiquement de la racine ;
- de permettre l'accès depuis le répertoire `sedoglav` à un fichier référencé dans un autre répertoire (lien symbolique codé par un fichier).



Cette organisation des fichiers est basée sur un type de fichier — les répertoires — codant les arêtes constituant le graphe. Les répertoires étant des fichiers, ils ont les mêmes attributs (droits, etc).

Quelques répertoires classiques de l'arborescence de type unix

`/boot` contient le noyau et le gestionnaire de démarrage ;

`/bin` contient les exécutables des programmes basiques ;

`/dev` contient les fichiers périphériques ;

`/etc` contient les fichiers de configurations ;

`/home` contient les fichiers utilisateurs (vos données) ;

`/lib` contient les librairies partagées (du langage C par exemple) ;

`/swap` est l'espace utilisé pour décharger la mémoire ;

`/proc` est l'image de l'exécution du noyau (voir la suite) ;

`/root` contient les fichiers du super-utilisateur ;

`/sbin` contient les exécutables des fichiers d'administration ;

`/tmp` est de l'espace réservé pour les données temporaires ;

`/var` contient les données fréquemment modifiées (journaux, etc.) ;

`/local` contient ce que les utilisateurs partagent et qui n'est pas standard au système. Il convient de séparer ce qui propre à l'OS de ce qui l'est aux applications.

Remarques sur ce type d'abstraction

L'abstraction *arborescence des répertoires* hérite des propriétés de l'abstraction fichier sans travail supplémentaire (droits, etc). Il s'agit de la première occurrence d'un principe général.
On présente souvent les répertoires suivant la métaphore d'un dossier contenant les fichiers dans les interfaces graphiques. Il est important de distinguer la métaphore de l'abstraction.
Remarquez que :

- la taille d'un répertoire n'est pas celle des fichiers qu'il contient mais celle nécessaire pour coder l'ensemble des liens. Par exemple :

```
[espoir.lifl.fr-sedoglav-/home/calforme] ls -al
drwxr-xr-x   45 sedoglav calforme   126976 Aug 19 10:47 sedoglav/
```

- Certains systèmes de fichiers permettent l'accès à des fichiers stockés sur des supports distincts de l'ordinateur local (cf. la notion de montage) par le biais de répertoire. Il convient de garder à l'esprit qu'un répertoire ne contient pas les fichiers auxquels il donne accès.

La notion de montage

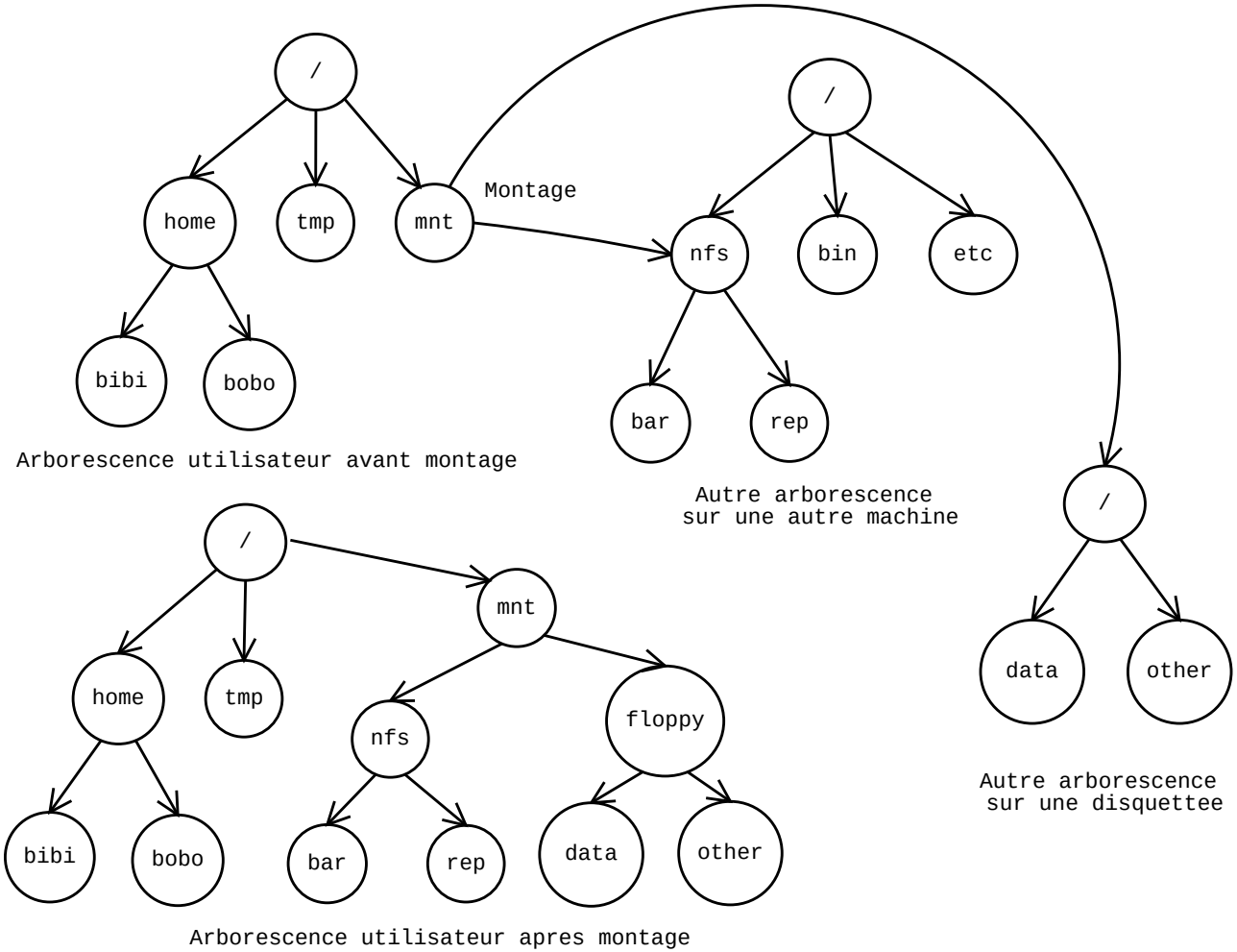
Dans les FS de type UNIX, le répertoire `/mnt` est utilisé pour les points de montage (cdrom, floppy, etc).

Certains FS — NTFS par exemple — distinguent les arborescences situées physiquement sur des supports différents (disquettes, cdrom, disques durs, clefs USB, réseaux, etc).

D'autres ne font pas cette distinction grâce à la notion de *montage*. Il s'agit d'associer à un répertoire une arborescence de fichiers codée par un FS pouvant être différent de celui auquel ce répertoire appartient.

Ainsi, une disquette formatée sur un ordinateur utilisant un OS de type Windows aura une arborescence de fichiers codées par le File System FAT (File Allocation Table cf. la suite). Il est possible de la 'monter 'sur une arborescence de type UNIX et d'accéder à ces fichiers.
Le même principe s'applique aux cdroms, aux FS accessibles par réseaux, etc.

Illustration du montage



On ne peut pas monter n'importe quel répertoire mais seulement une unité de base nommée *volume* (cf. seconde partie du cours sur les FS).

Montage automatique : stockage de l'information associée

Des fichiers contiennent les informations relatives aux montages :

- `/etc/fstab` décrit ce qui peut être automatiquement monté par le système en indiquant :
  - le périphérique utilisé dans le répertoire `/dev` (si besoin est) ;
  - le répertoire de montage dans l'arborescence ;
  - le type du système de fichier ainsi monté ;
  - des options concernant les droits.

  Ainsi sur ma machine de bureau, ce fichier ressemble à

```
/dev/hda9 / ext3 rw 0 0
none /proc proc rw 0 0
/dev/hda8 /local ext3 rw 0 0
none /mnt/cdrom supermount ro,dev=/dev/hdc,fs=auto, <etc.>
/dev/hda2 /mnt/windows ntfs ro,iocharset=iso8859-1,umask=0 0 0
livinus:/vol/home/calforme /home/calforme nfs rw,addr=134...
```

- `/etc/mtab` est un fichier classique qui indique ce qui est effectivement monté (le fichier `/proc/mounts` présente le même type d'information mais il est géré par le noyau).

Quelques commandes externes des shells

L'outil fondamental est le manuel d'utilisation `man` et la première chose à faire est de lire l'aide sur le manuel en utilisant la commande `>man man` dans votre interpréteur de commandes (shell) favori.

- `man -a mount` affiche l'ensemble des pages d'aide contenant le mot mount. Entre autre :

```
mount                   (2)  - mount and unmount filesystems
mount                   (8)  - mount a file system
```

- `man -S8 mount` affiche l'aide sur `mount` issue de la section 8 du manuel.

On peut aussi utiliser l'utilitaire `info` mais, bien que plus évolué (liens hypertext) il n'est pas forcement complet.
Ceci fait les commandes shell n'auront plus de secrets pour vous :

- `ls` affichage des informations relatives au contenu d'un répertoire ;
- `cd` déplacement dans l'arborescence ;
- `mount` montage de système de fichier dans l'arborescence des fichiers.

Quelques commandes utiles

passwd	créer ou changer de mot de passe
ps	afficher la liste des processus de l'utilisateur
pwd	afficher le nom du répertoire courant
cd	changer de répertoire
chmod	changer les droits d'un fichier
cp	copie de fichier
date	afficher la date
find	rechercher un fichier
grep	afficher les lignes des fichiers contenant une chaîne donnée de caractères
kill	stopper un processus
less	afficher le contenu d'un fichier
mkdir	Créer un répertoire
mv	déplacement de fichier
rm	détruire un fichier
rmdir	supprimer un répertoire

Un système de fichier spécial : le répertoire /proc

Le contenu du répertoire /proc n'est jamais stocké sur un support physique : il est engendré par le noyau sur requête de l'utilisateur (less /proc/mounts par exemple).
Chaque sous-répertoire de /proc correspond à un processus actif et porte comme nom le numéro d'identification de ce dernier.

```
[espoir.lifl.fr-sedoglav-/proc] ps
  PID TTY          TIME CMD
22356 pts/1    00:00:00 csh
[espoir.lifl.fr-sedoglav-/proc] cd 22356 ; ls
binfmt cmdline cwd@ environ exe@ fd/ maps mem mounts root@ stat statm
```

Les fichiers ci-dessus donnent accès à des informations (environ), des statistiques sur le processus (status) ou à un périphérique (mem).
On peut modifier les arguments de l'OS en écrivant directement les valeurs ASCII correspondantes dans les fichiers adéquats dans le répertoire /proc (à conditions d'avoir les droits suffisants).

Du coté de Microsoft : New Technologie File System

En 1993 le FS NTFS remplace la FAT (1980). Dés lors, un volume contient :

- Partition Boot Sector ;
- Master File Table ;
- Fichiers système ;
- L'espace des fichiers.

Les modifications en résultant sont l'apparition de :

- droits propriétaires ;
- montage au sein d'une arborescence ;
- cryptage des fichiers ;
- base de données (pour la recherche et la tolérance aux pannes).

New Technologie File System & ext3

Le système de fichiers de Windows NT est basé sur une base de donnée.
Une des faiblesses des FS est la détérioration des structures de données représentant les fichiers lors de leurs manipulations.
Par exemple, on peut endommager les structures de données permettant de manipuler des fichiers [4].
(Dans ce cas on utilise la commande fsck qui examine l'ensemble des blocs d'un disque et essaye de recomposer le tout.)

Idée : mettre une couche supplémentaire et faire des transactions

- on utilise des copies des structures : les ombres ;
- on fait des transactions avec les ombres ;
- en cas de réussite, les ombres deviennent valides.

NTFS (Microsoft) et ext3 (Open Source Software, OSS) sont basés sur ce principe.

4. qui seront explicitées dans la seconde partie du cours sur les FS.