Plan de la séance

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

Notion de fichier et informations utilisateur relatives

Appels système d'entrées-sorties

Librairie d'entrées-sorties : une couche de plus

Organisation hiérarchique des fichiers en arbre

La notion de montage

Outils de manipulation des fichiers au niveau de l'interpréteur de commandes

Format de fichiers et fichiers spéciaux

Du coté de Microsoft : New Technologie File System

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).

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 :

```
[/quelquepart/dans/le/grand/univers/OS/Exam] ls -al Juin2005.tex
-rw-r--r--    1 sedoglav calforme        0 Aug 19 05:09 Juin2005.tex
```

Ces informations correspondent à

```
droits | nb_liens | proprietaire | son_groupe | taille | date_creation | nom
```

Dans les OS dérivés d'UNIX, le codage des droits se fait sur 10 bits 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 bits 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.

L'appel `access` permet la vérification de ces droits :

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

L'appel `chmod` permet de changer ces droits (`man -S2 chmod`).

Ces appels système sont disponibles sous la forme de fonctions C :

```
#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);
```

mais aussi depuis le shell (`man chmod`).

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.)

On trouve dans le 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 ((1));
```

Le code source de cette fonction est disponible dans les sources du noyau :

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 ;
- descripteur : index dans la table des descripteurs du processus ;
- pointe dans la table des fichiers ouverts du noyau.

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*[a] d'un fichier.

L'association de ce descripteur avec l'*inombre*[b] 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[c] 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).

[a] cf. seconde partie du cours sur les FS.

[b] ibidem.

[c] Bien qu'ils s'agissent de périphérique, ce sont bien des fichiers (cf. la suite).

Les appels système les plus utilisés sont :

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

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

`int open(char *name, int mode [, int perm])` ouvre le fichier :

`name` suivant le mode et les permissions spécifiés, et retourne le descripteur correspondant (-1 en cas d'erreur). `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 combine par un ET bit à bit, par exemple :
`O_WRONLY | O_CREAT | O_TRUNC`

L'appel `int close(int fd)` ferme le fichier associé au descripteur `fd` ses fichiers ouverts sont fermés.

Cet appel retourne 0 si l'opération est un succès et -1 si le fichier est déjà fermé.

Le fichier d'entête fournissant les prototypes des fonctions suivantes est `unistd.h`.

`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.

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).

Il est possible d'ajouter une couche supplémentaire de stockage dans la gestion des entrée – sortie (ce niveau est géré au niveau du processus). En conséquence :
– on peut avoir une lecture/écriture par bloc dans le buffer ;
– il y a moins d'appels système pour des accès sur de petites zones ;
– et de vidage des buffers si il survient une interruption du processus.

Pour ce faire, on utilise un identificateur d'ouverture de fichier (flot) : de type `FILE *` (pointeur sur une structure de ce nom).

Les déclarations sont dans `<libio.h>` et `<stdio.h>` pour les curieux.

On décrit dans la suite les 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 des appels).

`FILE *fopen(const char *name, const char *mode)` ouvre le fichier dont le nom 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 fclose(FILE *stream)`
ferme le fichier associé au flot `stream`. Vidage des buffers.
Renvoie 0 en cas de succès, `EOF` si échec.

```
FILE *freopen(const char *name, const char *mode,
      FILE *stream)
```
ouvre le fichier dont le nom est donné par `name` dans le mode spécifié par `mode`, et lui associe le flot pointé par `stream`.
Le fichier associé à `stream` est préalablement fermé.
Retourne `stream` en cas de succès, `NULL` si échec.

`int fflush(FILE *stream)`
procède au vidage des buffers associés au flot de sortie `stream`.
Comportement indéterminé si `stream` est un flot d'entrée.
Retourne 0 en cas de succès, `EOF` si échec.

```
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 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 indiquant comme flot le flot prédéfini `stdout` associé à la sortie standard.

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 ($d.ddd$`e`$\pm dd$) ;

`p` : `void *` en format pointeur (hexa. en général).

`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 *`.

```
#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 ;}
```

La communauté des fichiers est organisée en arbre i.e. en un ensemble de nœuds reliés par des arêtes orientées : 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.

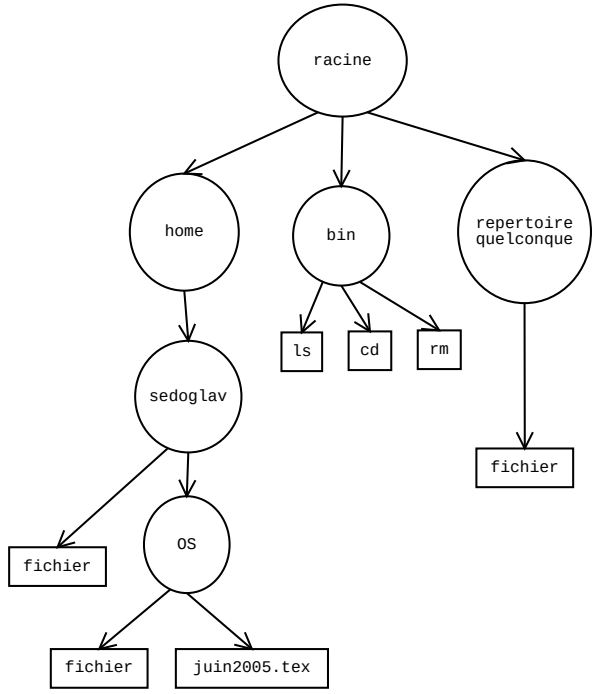
Les feuilles correspondent aux fichiers et les autres nœuds sont des *répertoires*. On peut ainsi définir un chemin d'accès à un fichier :

– absolu : depuis la racine ;

– relatif : notion de répertoire courant.

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

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

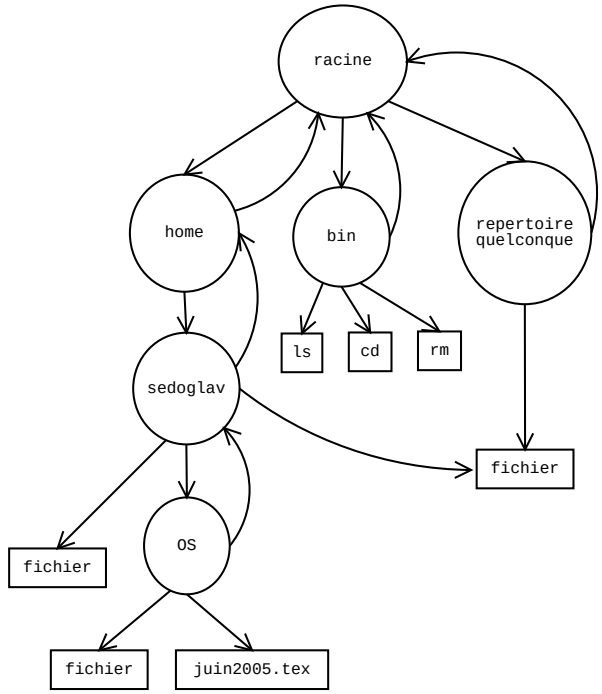


Les répertoires sont des fichiers (flux linéaire d'octets) contenant l'information liée aux arêtes les quittant.

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).

`/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.

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.

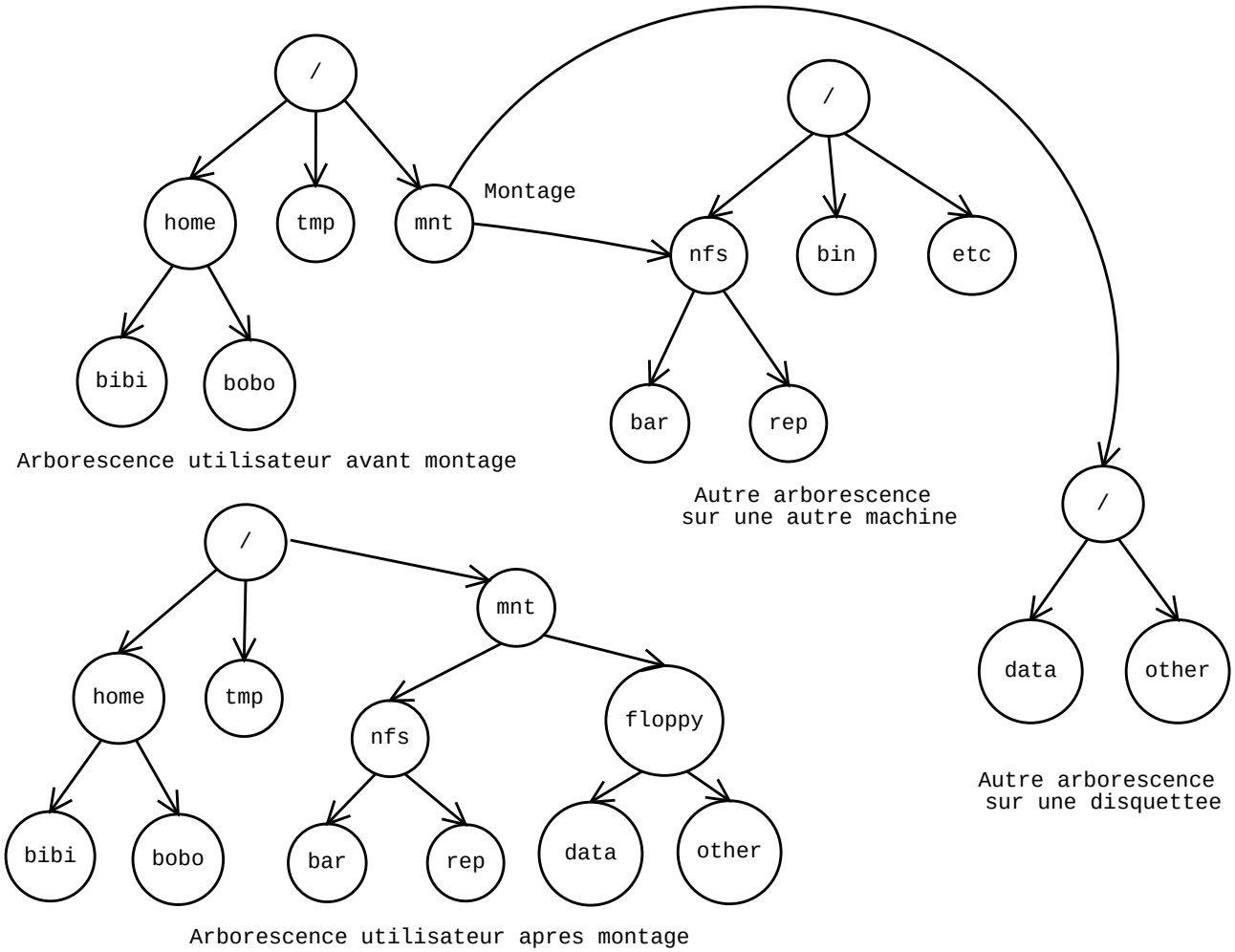
Dans les FS de type unix, le répertoire `/mnt` est utilisé pour les points de montage (cdrom, floppy, etc).

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.



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).

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,--,iocharset=iso8859-1, etc.
/dev/hda2 /mnt/windows ntfs ro,iocharset=iso8859-1,umask=0 0 0
livinus:/vol/home/calforme /home/calforme nfs rw,soft,addr=134.206.10.24 0 0
```

`/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).

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.

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

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 exécutable :
  - 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).

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.

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 status
```

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).

Un nouveau FS remplaçe l'ancien (basé sur la FAT). 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).

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[a].

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 (OSS) sont basés sur ce principe.

[a] qui seront explicitées dans la seconde partie du cours sur les FS.