Cours n° C.2

Système de fichiers

Objectifs

- *Utilisateur* : sauvegarder des données, les organiser, y avoir accès facilement. Caractérisation par :
  - un nom
  - éventuellement une localisation
- *Système* :
  - gestion des ressources matérielles
  - stockage d'informations particulières (taille, date de création, droits, ...)
- *Système multi-utilisateurs* :
  - partage de ressources entre utilisateurs
  - protection des accès

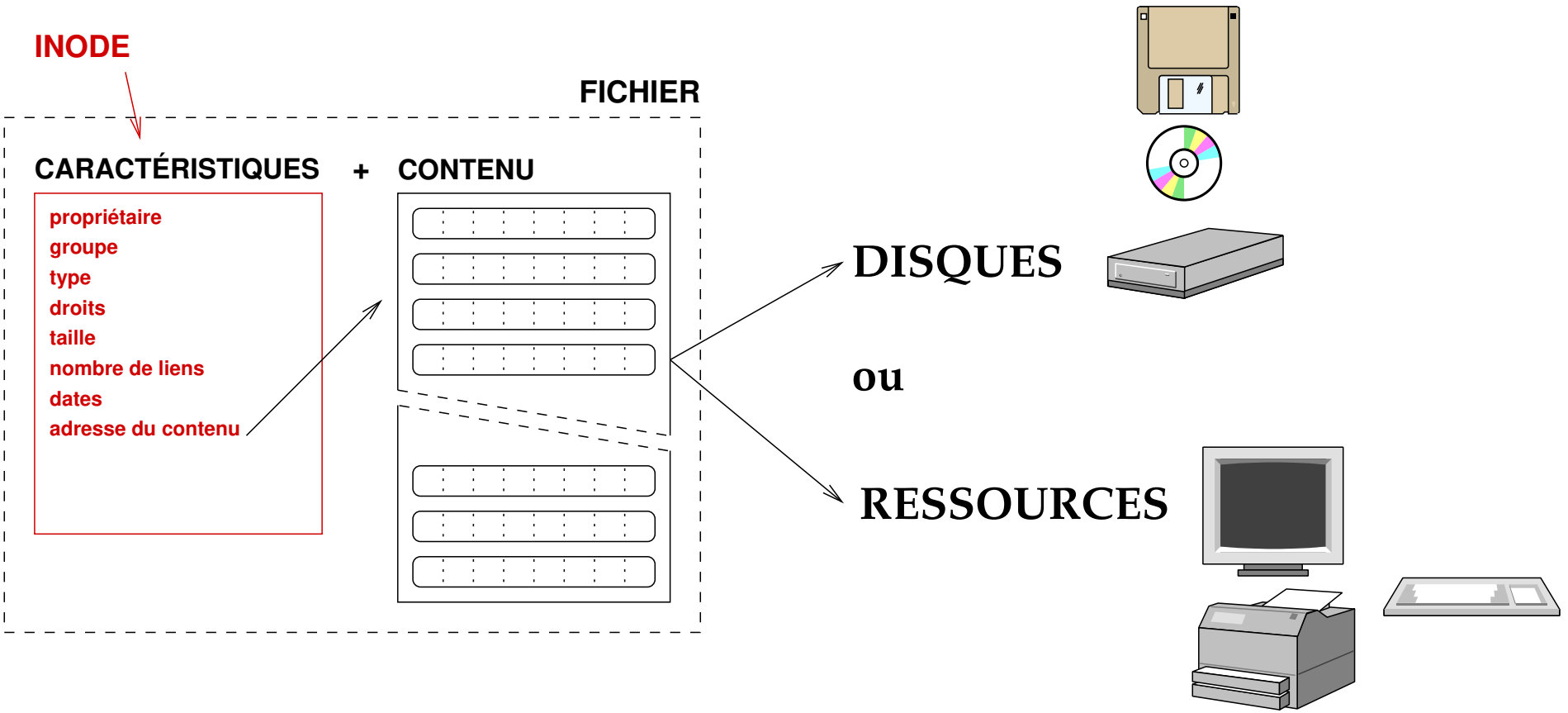
Philosophie

- ☞ Sous UNIX : **TOUT EST FICHIER**

- ☞ Le terme *fichier* désigne les ressources :
  - matérielles (disquette, disque dur, terminal, ...)
  - logicielles (image, son, texte, ...)

- ☞ Les primitives génériques d'accès aux fichiers permettent de réaliser des opérations de lecture/écriture sur **toutes** les ressources du système

- ☞ Chaque fichier est associé à une structure décrivant ses caractéristiques (*inode*)


INODE
FICHIER
CARACTÉRISTIQUES + CONTENU
propriétaire
groupe
type
droits
taille
nombre de liens
dates
adresse du contenu
DISQUES
ou
RESSOURCES

Organisation logique

- ☞ Les *fichiers* UNIX peuvent être des fichiers disques **classiques** ou des fichiers **ressources**.
- ☞ Plusieurs disques et de nombreuses ressources peuvent être connectés à une machine.
- ☞ Chaque disque physique peut également être partitionné en plusieurs disques logiques.
- ☞ À chaque fichier correspond un inode qui contient, entre-autres :
  - l'identification du disque logique du fichier
  - son numéro d'identification dans ce disque logique

➠ Tous les fichiers apparaissent à l'utilisateur dans une arborescence unique

Les inodes

Un *inode* est une structure qui contient les informations suivantes :

① identification du propriétaire et du groupe propriétaire du fichier ;

② type et droits du fichier ;

③ taille du fichier en nombre de caractères (si possible) ;

④ nombre de liens physiques du fichier ;

⑤ trois dates (dernier accès au fichier, dernière modification du fichier, dernière modification de l'inode) ;

⑥ adresse des blocs utilisés sur le disque pour ce fichier (pour les fichiers disques) ;

⑦ identification de la ressource associée (pour les fichiers spéciaux).

➠ **Chaque fichier est identifié uniquement par son inode**

La structure stat

```
struct stat
{
  dev_t         st_dev;      /* device */
  ino_t         st_ino;      /* inode */
  mode_t        st_mode;     /* protection */
  nlink_t       st_nlink;    /* number of hard links */
  uid_t         st_uid;      /* user ID of owner */
  gid_t         st_gid;      /* group ID of owner */
  dev_t         st_rdev;     /* device type (if inode device) */
  off_t         st_size;     /* total size, in bytes */
  blksize_t     st_blksize;  /* blocksize for filesystem I/O */
  blkcnt_t      st_blocks;   /* number of blocks allocated */
  time_t        st_atime;    /* time of last access */
  time_t        st_mtime;    /* time of last modification */
  time_t        st_ctime;    /* time of last status change */
};
```

issu du manuel : `open(2)`

Modification du propriétaire et du groupe (1)

```
int chown(const char *path, uid_t owner, gid_t group);
```

- ☞ `path` → chemin (relatif ou absolu) du fichier à tester
- ☞ `owner` → l'identifiant du nouveau propriétaire
- ☞ `group` → l'identifiant du nouveau group
- ➠ retourne 0 si le changement a pu être effectué et -1 sinon

plus de détails dans le manuel : `chown(2)`

Il faut que le processus soit exécuté par `root`
pour que l'appel puisse réussir

Il existe des commandes associées : `chown` et `chgrp`

plus de détails dans le manuel : `chown(1)` *et* `chgrp(1)`

Vérifier les droits d'accès

```
int access(const char *pathname, int mode);
```

- ☞ `pathname` → chemin (relatif ou absolu) du fichier à tester
- ☞ `mode` → droits à tester sur le fichier.
  - `R_OK`, test d'accès en lecture
  - `W_OK`, test d'accès en écriture
  - `X_OK`, test d'accès en exécution
  - `F_OK`, test d'existence du fichier

  } combinable avec l'opérateur « | »

- ➠ retourne 0 si l'accès est autorisé et -1 sinon

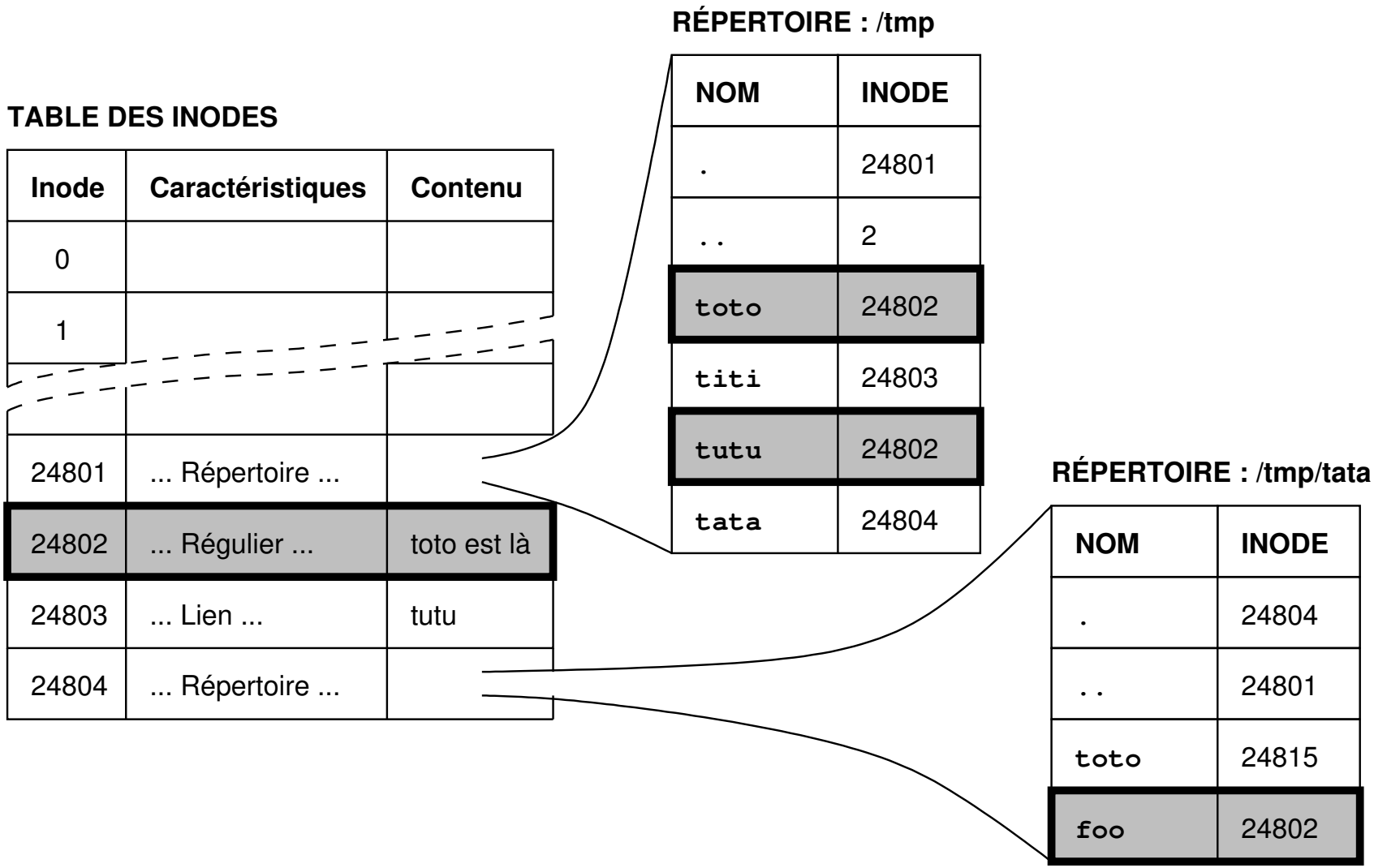
plus de détails dans le manuel : `access(2)`

Types de fichier

Ils permettent un niveau d'abstraction de plus :

- ☞ **fichier réguliers**
  Le contenu est une suite d'octets non structurée classique. La taille est connue et elle permet de trouver la fin du fichier sur le disque.
- ☞ **répertoires**
  Le contenu est structuré comme une liste d'entrées
- ☞ **fichiers spéciaux**
  Le contenu correspond à une ressources du système. Ils permettent des accès par blocs (disques, etc.) ou par octets (par *caractères*) (terminaux, imprimantes, etc.)
- ☞ **liens symboliques**
  Le contenu est interprété comme le chemin vers un autre fichier
- ☞ **tubes**
  Ils permettent la communication entre processus
- ☞ **sockets**
  Ils permettent la communication au sens général

Exemple



nombre de liens physique $\equiv$ nombre de noms

Exemple

```
bash$ ls -l /dev
brw-rw----   1 root     disk       3,   1 May  5  1998 hda1
brw-rw----   1 root     disk       3,  10 May  5  1998 hda10
brw-rw----   1 root     disk       3,  11 May  5  1998 hda11
brw-rw----   1 root     disk       3,  12 May  5  1998 hda12
brw-------   1 root     floppy     2,   0 May  5  1998 fd0
drwxr-xr-x   2 root     root           0 Jan  9 11:23 pts/
```

```
bash$ ls -l /dev/pts
crw--w----   1 root     tty      136,   0 Jan  9 13:19 0
crw--w----   1 root     tty      136,   1 Jan  9 13:16 1
```

Accès aux caractéristiques d'un fichier

```
int stat(const char *file_name, struct stat *buf);
```

- ☞ `file_name` → chemin (relatif ou absolu) du fichier à caractériser
- ☞ `buf` → adresse d'une zone de type `struct stat` qui sera remplie avec les caractéristiques du fichier
- ➠ retourne 0 en cas de succès et -1 en cas d'erreur

- ☞ le processus doit posséder les droits d'accès en recherche sur tous les répertoires du chemin spécifié
- ☞ le pointeur `buf` doit pointer sur une structure `stat`

plus de détails dans le manuel : `stat(2)`

Déterminer le type

Pour déterminer le type d'un fichier il suffit d'observer le contenu du champ `st_mode` d'une structure `stat`

Des macros, applicables à ce champ, permettent de tester facilement le type d'un fichier (technique des masques) :

- `S_ISLNK(m)` est-ce un lien symbolique ?
- `S_ISREG(m)` est-ce un fichier régulier ?
- `S_ISDIR(m)` est-ce un répertoire ?
- `S_ISCHR(m)` est-ce un fichier caractères ?
- `S_ISBLK(m)` est-ce un fichier blocs ?
- `S_ISSOCK(m)` est-ce une socket ?
- ...

Fichiers catalogues (répertoires)

Les données dans un fichier catalogue correspondent grossièrement à une liste de couples (entrées) :

- ☞ nom ..... une chaîne de caractères
- ☞ numéro d'inode ..... un entier

Les structures exactes de la liste et des entrées dépend du système de fichiers. Pour éviter les appels bas-niveaux (dépendant du système de fichiers) on utilise les fonctions de la librairie standard C.

➠ Interface indépendante du système

Les données des répertoires sont manipulées via des pointeurs vers une structure spécifique : `DIR *`

Ouverture

```
DIR * opendir(const char *name);
```

- ☞ `name` → chemin (relatif ou absolu) du répertoire à ouvrir
- ➠ retourne un pointeur vers un flux de répertoire ou `NULL` si une erreur s'est produite

1. ouvre le répertoire
2. alloue un espace de type `DIR` pour ce répertoire
3. positionne la structure `DIR` pour qu'elle représente la première entrée du répertoire
4. renvoie l'adresse de cet espace (ou `NULL` si impossible)

plus de détails dans le manuel : `opendir(3)`

Fermeture

```
int closedir(DIR *dir);
```

- ☞ `name` → chemin (relatif ou absolu) du répertoire à ouvrir
- ➠ retourne 0 si la fermeture s'est bien déroulée et -1 sinon

① ferme le répertoire
② libère l'espace utilisé par la structure `DIR` pour ce répertoire
③ renvoie 0 si tout s'est bien passé ou -1 sinon

plus de détails dans le manuel : `closedir(3)`

Lecture de répertoire

```
struct dirent *readdir(DIR *dir)
```

- ☞ `dir` $\rightarrow$ pointeur vers le flux de répertoire à parcourir
- ➠ retourne un pointeur vers une structure de description d'entrée de répertoire ou `NULL` si une erreur s'est produite

1. lit l'entrée courante
2. modifie le flot pour pointer vers l'entrée suivante du répertoire
3. retourne l'adresse de cette entrée ou NULL si la fin est atteinte

plus de détails dans le manuel : `readdir(3)`

Entrée de catalogue

```
struct dirent
{
  ino_t                  d_ino;          /* numero d'inode */
  off_t                 d_off;         /* offset to the next dirent */
  unsigned short int  d_reclen;         /* length of this record */
  unsigned char          d_type;         /* type of file */
  char                   d_name[256];    /* filename */
}
```

Autres manipulations de répertoires

- ☞ Création de répertoire vide

```
int mkdir(const char *pathname, mode_t mode);
```

- ☞ Suppression de répertoire vide

```
int rmdir(const char *pathname);
```

plus de détails dans le manuel : `mkdir(2)`, `rmdir(2)`

Il existe des commandes associées : `mkdir`, `rmdir`

plus de détails dans le manuel : `mkdir(1)`, `rmdir(1)`

➠ Attention aux effets de bord d'autres fonctions
(la création d'un fichier modifie un répertoire par exemple)

Fichiers liens

- Les données dans un fichier lien correspondent grossièrement à un chemin vers un autre fichier

- Les fonctions «*usuelles*» suivents les liens symboliques :
  Les liens sont transparents pour l'utilisateur

  Création d'un lien symbolique :

```
int symlink(const char *oldpath, const char *newpath);
```

  plus de détails dans le manuel : `symlink(2)`

- L'option «`-s`» de la commande «`ln`» permet également de créer un lien symbolique
- Consultation des caractéristiques du fichier lien :

```
int lstat(const char *file_name, struct stat *buf);
```

  ➠ La fonction `stat` appliquée à un lien ne donne pas l'information sur le fichier lien mais sur le fichier vers lequel pointe le lien

Lecture d'un lien

```
int readlink(const char *path, char *buf, size_t bufsiz);
```

- ☞ `path` → chemin du fichier lien à lire
- ☞ `buf` → l'adresse d'une zone mémoire où les octets lus seront stockés par la fonction
- ☞ `busiz` → taille de l'espace réservé à l'adresse `buf`
- ➠ retourne le nombre de caractères lus ou -1 en cas de problème

Place la valeur du lien dans le buffer `buf` qui a la taille `bufsiz`, si `buf` est trop petit, la chaîne est tronquée.

Attention la zone remplie ne comporte pas de «\0» à la fin.

plus de détails dans le manuel : `readlink(2)`