FICHIERS ET RÉPERTOIRES

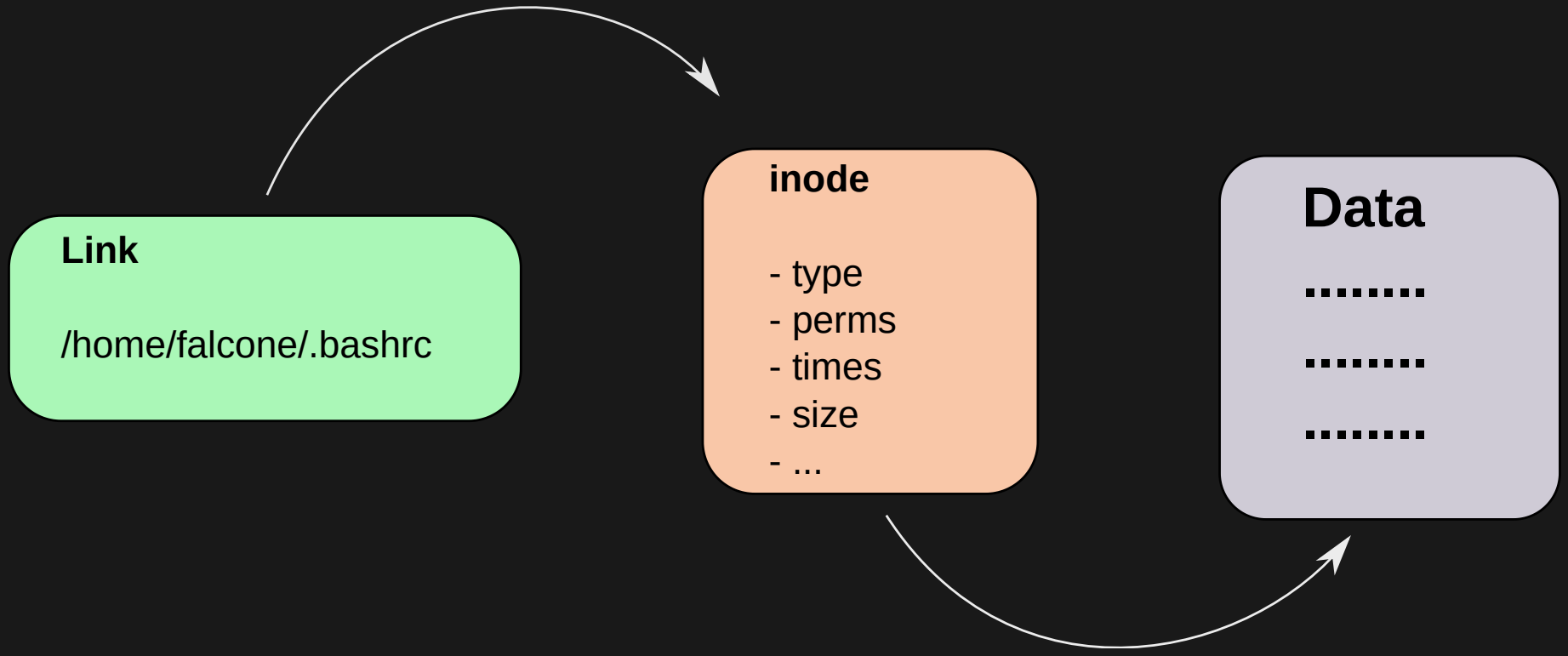
Guillaume Chanel

Remerciements à Jean-Luc Falcone

Septembre 2019

INODES

SCHÉMA GLOBAL



INODE

- Les inodes sont des structures de données contenant des informations sur un "fichiers" (fichier, directory, socket, device, pipe, etc.).
- Ils **ne contiennent pas le nom du fichier**.
- Ils contiennent généralement (POSIX) des informations sur:
  - Numéro d'inode
  - Périphérique contenant le fichier (`device ID`)
  - Propriétaire et groupe
  - Permissions
  - Taille du fichier
  - Temps d'accès et de modification
  - Nombre de liens pointant vers l'inode
  - **Pointeurs vers les données**

TEMPS DE L'INODE

Un inode contient trois temps différents:

atime	date du dernier accès à l'inode (ou aux données)
mtime	date de la dernière modification des données
ctime	date de la dernière modifications des méta-données

Information

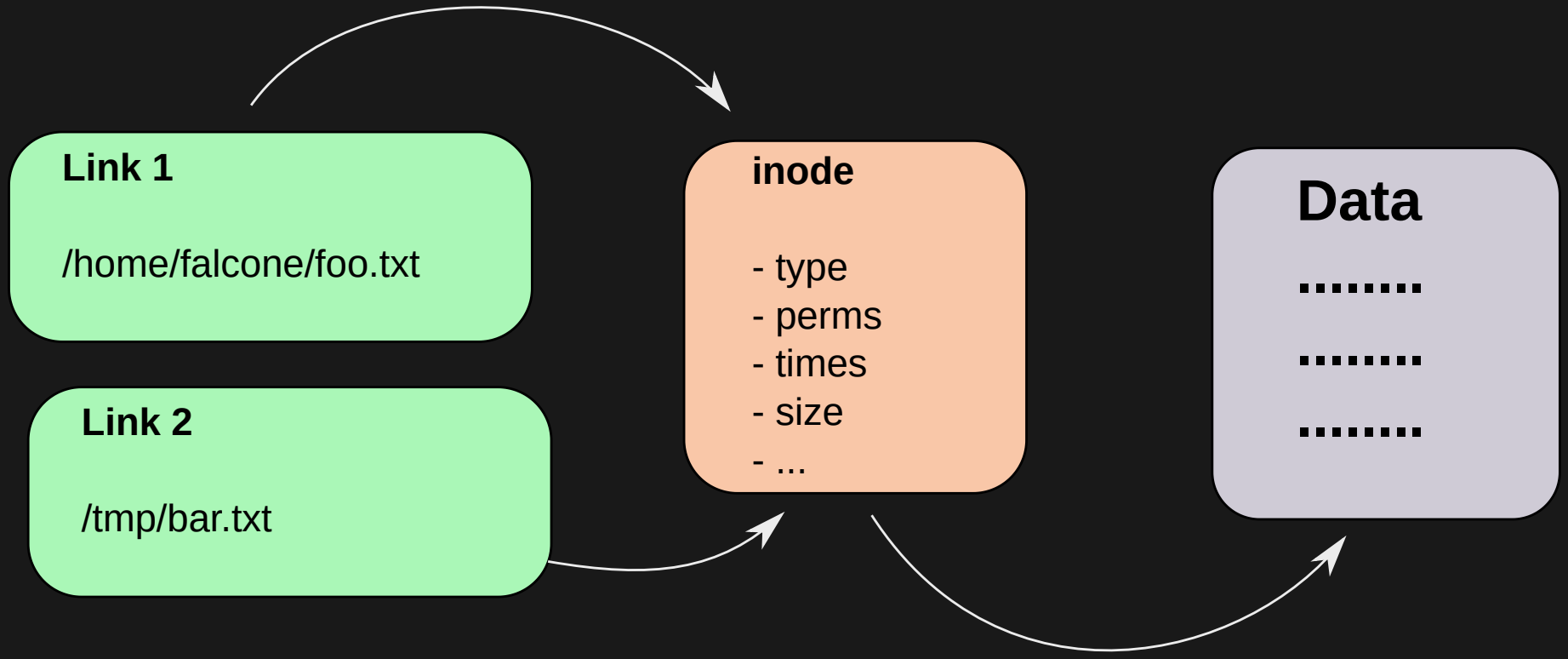
Pour gagner en performance, il est possible de désactiver la mise à jour de `atime` lorsque la partition est montée.

INSPECTER UN INODE EN SHELL (`stat`)

La commande `stat` permet d'afficher des données sur un inode.

```
$ touch /tmp/myfile  # met à jour les dates (crée un fichier si inexistant)
$ stat /tmp/myfile
  File: /tmp/myfile
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: 2fh/47d Inode: 422766      Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/  chanel)   Gid: ( 1000/  chanel)
Access: 2019-07-19 16:56:35.540113838 +0200
Modify: 2019-07-19 16:56:35.540113838 +0200
Change: 2019-07-19 16:56:35.540113838 +0200
 Birth: -
```

LIEN DUR (*HARD LINK*)



LIEN DUR (2)

- Les entrées des répertoires sont des liens pointant vers des `inodes`.
- On peut créer plusiers liens vers un fichier
- Un fichier est "effacé" lorsqu'il n'y a plus de liens pointant sur son inode (cf un `link`).

On peut créer un lien dur avec la commande `ln`:

```
$ ln /tmp/myfile /tmp/newlink
$ stat /tmp/newlink
  File: /tmp/myfile
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: 2fh/47d Inode: 422766      Links: 2
Access: (0644/-rw-r--r--)  Uid: ( 1000/  chanel)   Gid: ( 1000/  chanel)
Access: 2019-07-19 16:56:35.540113838 +0200
Modify: 2019-07-19 16:56:35.540113838 +0200
Change: 2019-07-19 16:56:35.540113838 +0200
 Birth: -
```

LIEN DUR (3)

Limitations

Un répertoire ne peux posséder qu'un seul lien dur.

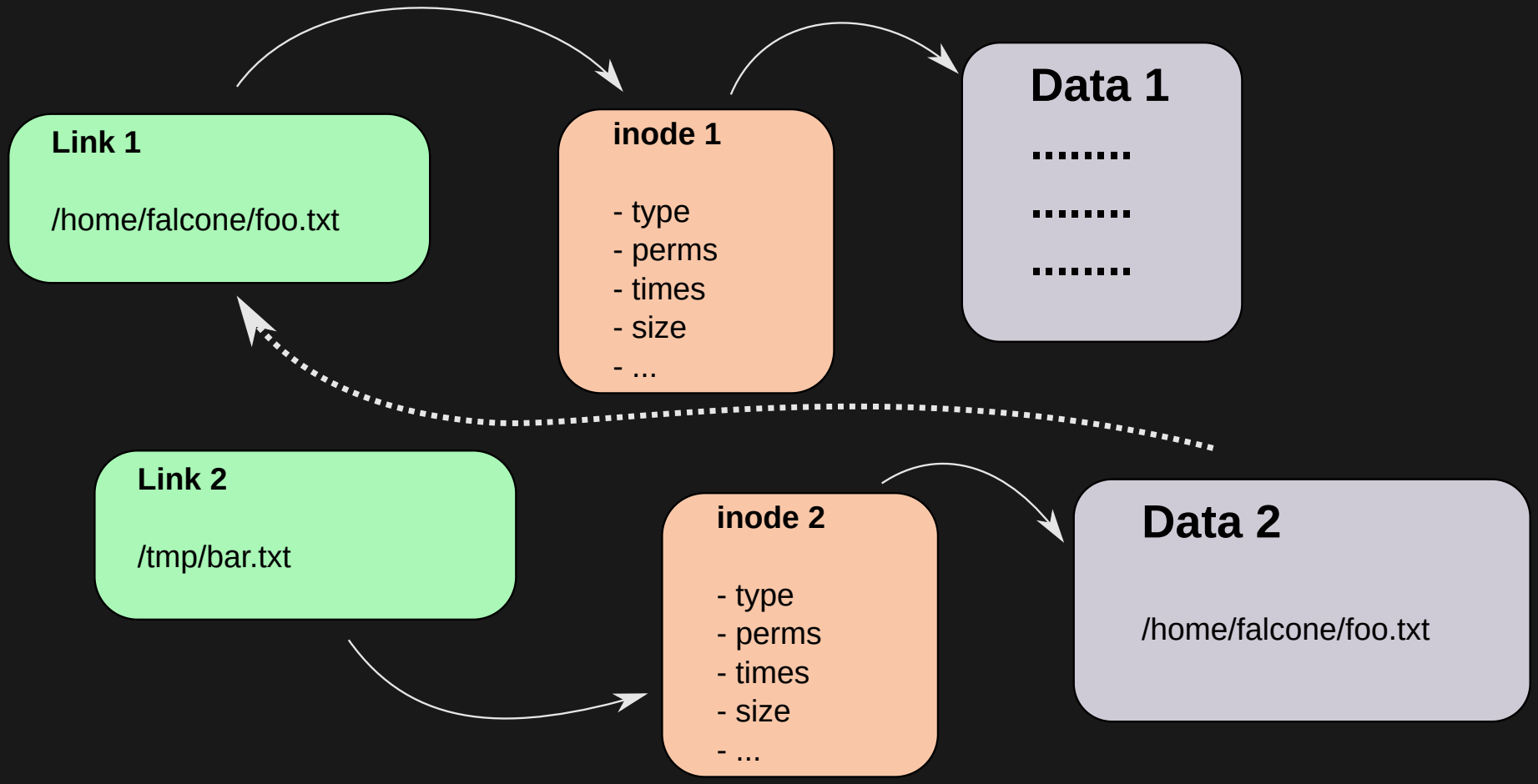
Tous les liens durs pointant sur un inode doivent se trouver sur le même système de fichier que cet inode.

Utilité

En général, assez peu utiles (préférer les liens symboliques, voir slides suivantes).

Permet de faire un *snapshot*, par exemple pour archiver un répertoire: http://www.mikerubel.org/computers/rsync_snapshots/ .

LIEN SYMBOLIQUE (*SYMLINK*)



LIEN SYMBOLIQUE (2)

- Un lien symbolique possède son propre inode qui pointe vers un **nom**
- Contrairement aux liens durs, on peut créer des symlinks:
  - Vers un répertoire
  - Vers un fichier/répertoire sur un autre système de fichier.

On peut créer un lien dur avec la commande la command `ln` en utilisant l'option `-s`:

```
$ ln -s /tmp/myfile /tmp/newlink  # (le lien dure précédent à été supprimé)
$ stat /tmp/newlink  # comme indiqué ci-dessous on a bien à faire à un autre inode
  File: /tmp/newlink -> /tmp/myfile
    Size: 11              Blocks: 0          IO Block: 4096   symbolic link
  Device: 2fh/47d Inode: 563876      Links: 1
  Access: (0777/lrwxrwxrwx)  Uid: ( 1000/  chanel)   Gid: ( 1000/  chanel)
  Access: 2019-07-19 18:16:26.343945684 +0200
  Modify: 2019-07-19 18:16:13.240259297 +0200
  Change: 2019-07-19 18:16:13.240259297 +0200
   Birth: -
```

CRÉER DES LIENS: APPELS SYSTÈMES (`link`, `symlink`)

Créer un lien dur

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

- `oldpath` est le nom existant et `newpath` le nouveau nom.
- Retourne 0 si succès et -1 si erreur (cf. `errno`).
- Si `newpath` existe, code d'erreur `EEXIST`.

Créer un lien symbolique

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

- Utilisation identique à `link()`.

EFFACER UN LIEN / UN NOM DE FICHIER: APPEL SYSTÈME (`unlink`)

- On efface un nom (lien) d'un fichier avec:

```c
int unlink(const char *pathname);
```

- où `pathname` est le nom à supprimer
- Retourne 0 en cas de succès et -1 en cas d'erreur
- voir `man 2 unlink` pour les codes d'erreurs

Remarques

- Fonctionne avec les liens durs ou symboliques
- Si le nom est le dernier à pointer sur un inode:
  - Si aucun processus n'a ouvert le fichier: inode supprimé immédiatement
  - Sinon: inode supprimé lorsque le dernier processus le ferme.

OBJETS TROUVÉS (*LOST+FOUND*)

- A la suite d'une erreur du système de fichier (p.e. suite à une mise hors-tension brutale), un inode peut se retrouver sans lien.
- Lors d'un contrôle de fichier (`fsck`), il sera copié dans le répertoire `lost+found` à la racine du système de fichier.

STRUCTURE STAT (sys/stat.h)

```
struct stat{
  dev_t     st_dev;     //device ID
  ino_t     st_ino;     //i-node number
  mode_t    st_mode;    //protection and type
  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 special file)
  off_t     st_size;    //total size, in bytes
  blksize_t st_blksize; //blocksize for filesystem I/O
  blkcnt_t  st_blocks;  //number of 512B blocks
  time_t    st_atime;   //time of last access
  time_t    st_mtime;   //time of last modification
  time_t    st_ctime;   //time of last change
};
```

APPEL SYSTÈME `stat`

L'appel système `stat()` permet de garnir une structure stat:

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

La fonction retourne 0 si tout s'est bien passé ou -1 en cas d'erreur (cf. `errno`)

```
struct stat infos;
char *filename = "/tmp/foo.txt";
if( stat( filename, &infos ) < 0 )
    fprintf( stderr, "Cannot stat %s: %s\n", filename, strerror(errno) );
else
    printf( "Filesize: %d\n", infos.st_size );
```

DÉTERMINER LE TYPE D'UN INODE

- Le champ `st_mode` est un champs de bits contenant les permissions et le type d'un inode.
- Il existe plusieurs macro POSIX permettant de tester les types:

`S_ISREG(m)`	fichier de données ?
`S_ISDIR(m)`	répertoire ?
`S_ISCHR(m)`	character device ?
`S_ISBLK(m)`	block device ?
`S_ISFIFO(m)`	FIFO (named pipe) ?
`S_ISLNK(m)`	lien symbolique ?
`S_ISSOCK(m)`	socket?

```
if( S_ISDIR( info.st_mode ) ) {
    printf( "L'inode est un repertoire.\n" );
}
```

DÉTERMINER LES PERMISSIONS D'UN INODE

On peut utiliser plusieurs flags pour accéder aux valeurs du champs de bits:

S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IROTH	00004	others have read permission
S_IWOTH	00002	others have write permission
S_IXOTH	00001	others have execute permission

APPEL SYSTÈME `lstat`

- Si le *A* est un lien symbolique vers *B*, `stat("A",...)` retourne les informations sur l'inode de *B*.
- On peut éviter ce comportement et obtenir les informations sur le lien symbolique lui-même grâce à `lstat()`:

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

- Le reste du comportement est identique à `stat()`.

APPEL SYSTÈME `fstat`

- Parfois on veut connaîtres les informations sur un fichiers déjà ouvert (cf suite du cours).
- L'appel `fstat()` fonctionne comme `stat()` mais permet d'utiliser un descripteur de fichier à la place d'un nom:

```
int fstat(int fd, struct stat *buf);
```

APPEL SYSTÈME `access`

- On peut tester si le processus en cours à le droit de lire/écrire/exécuter un fichier grâce à l'appel système `access()`:

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

- Le paramètre `mode` est un champs de bits formés des flags:

`R_OK`	lecture possible
`W_OK`	écriture possible
`X_OK`	éxécution possible

- On peut aussi tester le flag `F_OK` (seulement) qui indique si le fichier existe.
- Le test se fait en fonction de l'utilisateur/groupe vrai.
- `access()` retourne 0 si le test réussit, -1 sinon (cf `errno`)

APPEL SYSTÈME access (2)

```
char *fn = "/tmp/foo.txt";
if ( access( fn, R_OK|W_OK ) == 0 )
    printf( "On peut lire et ecrire sur %s\n", fn );
else if ( errno == EACCES )
    printf("Pas le droit de lire et/ou d'ecrire sur %s\n", fn);
else
    perror( fn );
```

APPEL SYSTÈME chmod

On peut changer les permissions d'un fichier grâce à l'appel système `chmod`, similaire à la commande shell du même nom:

```c
int chmod(const char *path, mode_t mode);

//Utilise un descripteur de fichier ouvert
int fchmod(int fd, mode_t mode);
```

Le paramètre `mode` est un champs de bits formés des mêmes flags que le champs `st_mode` de la structure `stat`.

RÉPERTOIRES / DIRECTORIES

LES RÉPERTOIRES (*DIRECTORIES*)

- permettent de lier un nom à un inode
- représentent l'organisation des fichiers sous forme d'**arborescence**
- contiennent une liste d'entrées (`dirent`)
- contiennent au moins `.` et `..`

STRUCTURE `dirent`

Les entrées d'un répertoire sont représentées par la structure:

```
struct dirent {                 /* dirent.h */
    ino_t   d_ino;              /* inode number */
    off_t   d_off;              /* opaque value used to get next dirent (do not use) */
    unsigned short  d_reclen;   /* length of this record */
    unsinged char   d_type;     /* type of file; not supported by all file systems */
    char            d_name[256]; /* filename (NULL terminated), sometimes d_name[0] */
};
```

Seulement deux champs sont décrit par POSIX: `d_ino` et `d_name`.

Ne jamais compter sur la taille du tableau `d_name`, uniquement sur la constante `MAX_NAME` qui indique la longueur maximale des noms d'entrées ou sur `strlen`

STRUCTURE `dir_ent` (3)

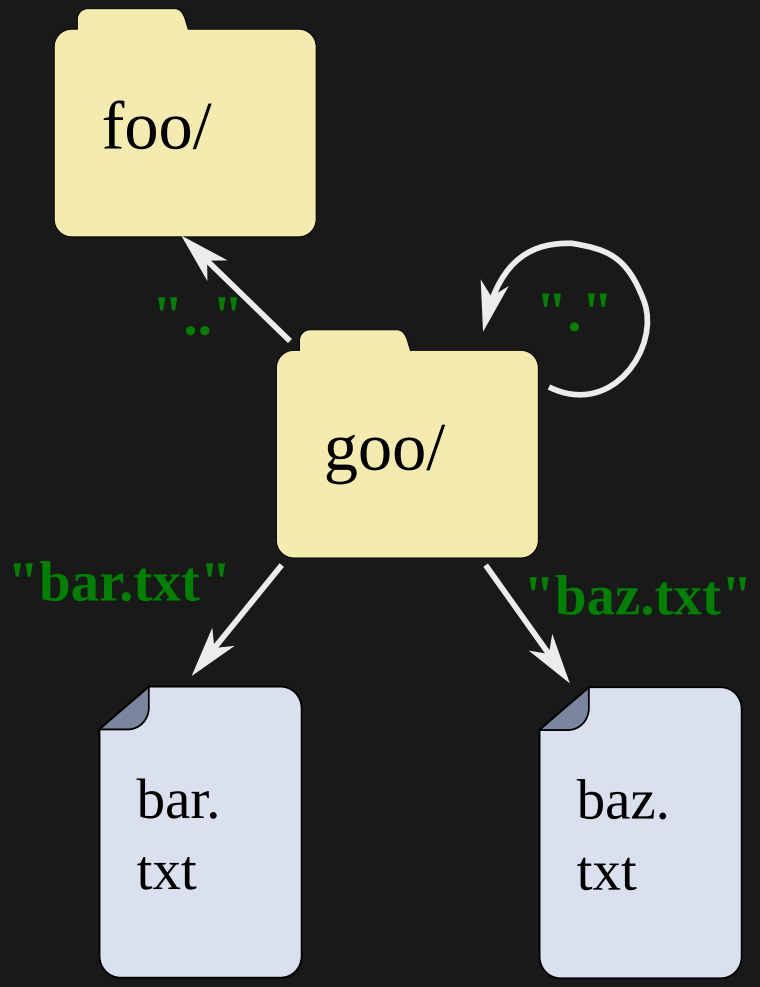
Le champs `d_type` est un champs de bits contenant des informations sur le type de l'inode associé:

`DT_DIR`	Répertoire
`DT_LNK`	Lien symbolique
`DT_REG`	Fichier de données
`DT_UNKNOWN`	Type inconnu
`DT_...`	Voir `man readdir` pour tous les types.

Attention

- Même sous GNU/Linux, tous les systèmes de fichiers ne donnent pas un accès au type par la structure `dir_ent`
- Dans ce cas, le `d_type` est toujours égal à `DT_UNKNOWN`.

ENTRÉES




```
foo/
foo/goo/
foo/goo/bar.txt
foo/goo/baz.txt
```

FLOT DE RÉPERTOIRES

Pour accéder aux entrées d'un répertoire, il faut:

1. "Ouvrir" le répertoire avec `opendir()`
2. "Lire" l'entrée suivante avec `readdir()`
3. Répéter 2, jusqu'à épuisement des entrées ou tout autre critère
4. "Fermer" le répertoire avec `closedir()`

OUVRIR UN RÉPERTOIRE (`opendir`)

- On peut ouvrir un répertoire grâce aux fonctions:

```c
DIR *opendir(const char *name);
DIR *fdopendir(int fd);
```

- `DIR` est un type opaque
- En cas d'erreur `DIR` sera `NULL`
- Exemples de codes d'erreurs (voir `man`):

`EACCESS`	opération interdite (permissions)
`ENOENT`	Le répertoire n'existe pas ou le nom est une chaîne vide.
`ENOTDIR`	Le nom existe mais n'est pas un répertoire.

LIRE L'ENTRÉE SUIVANTE (`readdir`)

- On peut lire l'entrée suivante d'un répertoire ouvert avec:

```
struct dirent *readdir(DIR *dirp);
```

- Retourne soit:
  - un pointeur sur une instance de la structure `dirent`
  - `NULL` s'il n'y a plus d'entrée **ou** en cas d'erreur.
- A chaque appel, une nouvelle entrée est retournée (s'il y en a encore).
- Un seul code d'erreur:

  `EBADF` Le descripteur `dirp` n'est pas valide.

LIRE L'ENTRÉE SUIVANTE (`readdir`)

Attention

- La structure retournée est susceptible d'être modifiée par chaque appel.
- Ne jamais appeler `free` sur le pointeur retourné.
- `readdir` n'est pas *thread-safe*.

FERMER UN RÉPERTOIRE (`closedir`)

- On peut fermer un répertoire ouvert avec:

```c
int closedir(DIR *dirp);
```

- Retourne 0 en cas de succès et -1 en cas d'erreur.
- Un seul code d'erreur:

  EBADF Le descripteur `dirp` n'est pas valide.

EXEMPLE (examples/listDir.c)

CRÉER UN RÉPERTOIRE (`mkdir`)

- On peut créer un répertoire avec:

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

- `pathname` est le nom du répertoire
- `mode` spécifie les permissions à utiliser, il est modifié par le `umask` du processus:

```
mode & ~umask & 0777
```

- Retourne 0 en cas de succès et -1 en cas d'erreur
- voir `man 2 mkdir` pour les codes d'erreurs

EFFACER UN RÉPERTOIRE (`rmdir`)

- On efface un répertoire **vide** avec:

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

- Retourne 0 en cas de succès et -1 en cas d'erreur
- voir `man 2 rmdir` pour les codes d'erreurs