Le cours
Un peu de C6
C
Pointeurs
Types
Compilation
E/S C
Systèmes de fichiers
E/S Systèmes
Avancé
reste

gestion des erreurs

La manipulation des flots peut conduire à des «erreurs» il existe quelques fonctions permettant de tester le statut correspondant.

clearerr, feof et ferror

Definition (clearerr, feof,ferror)

```
#include <stdio.h>
void clearerr(FILE *stream);
int feof(FILE *stream)
int ferror(FILE *stream);
```

`clearerr` permet de remettre l'indicateur de fin de fichier associé au flot à la valeur par défaut (faux). Tant qu'aucune fonction ne replace les indicateurs à une autre valeur, ils ne sont pas modifiés.
`feof` renvoie la valeur de l'indicateur de fin de fichier.
`ferror` renvoie la valeur de l'indicateur d'erreur (impossibilité de réaliser l'entrée/sortie).

Note

Toutes les fonctions ne positionnent pas les indicateurs, comme par exemple la fonction de positionnement `fseek`.
Consulter la documentation de chacune des fonctions pour plus d'informations.
La règle générale est que les fonctions qui réalisent des lectures ou des écritures positionnent les indicateurs.

Attention

Le test d'un indicateur ne doit intervenir qu'après une opération de lecture ou d'écriture!

Exemple

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  if (argc!=2) exit(1);
  char c;
  FILE *f;

  f = fopen(argv[1], "rb");
  if (f==NULL) exit(2);

  while(fread(&c, sizeof c, 1, f)==1) {
    printf("%c", c);
  }

  if (ferror(f)) printf("Error\n");

  if (feof(f)) printf("End␣of␣file\n");

  fclose(f);
  if (ferror(f)) exit(4);
  exit(0);
}
```

```
$ ./err err.c
#include <stdio.h>
#include <stdlib.h>

...

  if (ferror(f)) exit(4);
  exit(0);
}
End of file
$
```

codes d'erreurs

Comme on a pu le constater la gestion des erreurs est plutôt primitive, on sait si une erreur a été produite mais on ne connait pas la cause essentielle de cette erreur. En effet, les fonctions de bibliothèques renvoient :

- soit des valeurs «correctes» en cas de succès,
- soit une valeur prédéterminée en cas d'échec (généralement 0 pour les fonctions qui renvoient un type intégral, `NULL` pour les fonctions qui renvoient un pointeur).

Definition (errno)

```
#include <errno.h>
```

permet d'obtenir la visibilité globale d'un symbole de nom `errno` et de type **`int`** qui contient un **code d'erreur lorsqu'une erreur a été produite** lors d'un appel de fonction de bibliothèque.
La valeur 0 n'est pas un code d'erreur.

Le cours

Un peu de C6

C

Pointeurs

Types

Compilation

E/S C

Systèmes de fichiers

E/S Systèmes

Avancé

reste

Attention

Les fonctions de bibliothèques ne positionnent **jamais** cette variable à 0. Cela signifie qu'il faut soi-même la remettre à zéro.

Attention (piège)

Les fonctions de bibliothèques peuvent positionner `errno` à une valeur non nulle y compris si la fonction n'échoue pas!

Comment faire ?

On teste d'abord si la fonction échoue ou non et **si** elle échoue on consulte le contenu d'`errno`.

Attention

Les valeurs possibles de la variable `errno` sont, pour la plupart, définies par l'implémentation. Il est donc nécessaire de consulter le manuel du système hôte.

```
$ man fgetpos
...
ERRORS
     [EBADF]              The stream argument is not a
          seekable stream.

     [EINVAL]             The whence argument is invalid or
          the resulting file-position indicator would be set
          to a negative value.

     [EOVERFLOW]          The resulting file offset would be a
           value which cannot be represented correctly in an
          object of type off_t for fseeko() and ftello() or
          long for
                          fseek() and ftell().

     [ESPIPE]             The file descriptor underlying
          stream is associated with a pipe or FIFO or file-
          position indicator value is unspecified (see ungetc
          (3)).
...
$
```

Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main(int argc,char *argv[]) {
  if (fseek(stdin,10,SEEK_CUR)!=0) {
    switch (errno) {
    case EBADF:
      fprintf(stderr,"pas␣de␣seek␣la-dessus!\n")
          ;
      break;
    case ESPIPE:
      fprintf(stderr,"c'est␣pas␣un␣pipe␣ca␣?\n")
          ;
      break;
    }
    exit(1);
  }
  printf("ok\n");
  exit(0);
}
```

```
$ ./errno1
ok
$ ./errno1 < errno1.c
ok
$ cat errno1.c | ./errno1
c'est pas un pipe ca ?
$
```

Le cours

Un peu de C6

C

Pointeurs

Types

Compilation

E/S C

Systèmes de fichiers

E/S Systèmes

Avancé

reste

de l'erreur au message

Il existe des fonctions permettant d'associer un message standardisé à un code d'erreur donné :

Definition (strerror)

```
#include <string.h>
char *strerror(int errnum);
```

permet d'obtenir un pointeur vers une chaîne de caractères représentant un message d'erreur standardisé correspondant au code d'erreur indiqué.

Le cours
Un peu de C6
C
Pointeurs
Types
Compilation
E/S C
Systèmes de fichiers
E/S Systèmes
Avancé
reste

de l'erreur au message

Definition (perror)

```
#include <stdio.h>
void perror(const char *s);
```

Cette fonction permet d'obtenir l'affichage sur la sortie erreur standard d'un message d'erreur correspondant à la valeur courante d'`errno`. Si le paramètre *s* est différent de `NULL`, la chaîne pointée par *s* est d'abord affichée, suivie de `": "` puis du message standardisé le tout terminé par un retour à la ligne. Si le paramètre est `NULL` seul le message standardisé et le retour à la ligne sont envoyés. Cette fonction utilise un appel à `strerror(errno);` pour déterminer le message standardisé.

Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main(int argc,char *argv[]) {
  if (fseek(stdin,10,SEEK_CUR)!=0) {
    perror(argv[0]);
    exit(1);
  }
  printf("ok\n");
  exit(0);
}
```

```
$ ./errno2 < errno2.c
ok
$ ./errno2 < /dev/null
ok
$ cat errno2.c | ./errno2
./errno2: Illegal seek
$
```

tampons (*buffers*)

Les entrées/sorties de la bibliothèques peuvent utiliser une mémoire tampon.

Definition (mémoire tampon)

Une **mémoire tampon** (*data buffer*) est une zone mémoire utilisée pour stocker temporairement des données lors de leur échange entre deux entités dont le fonctionnement n'est pas synchrone ou de vitesses très différentes.
On utilise cette mémoire afin d'accélérer les entrées/sorties en évitant que l'entité la plus lente n'impose sa loi à l'autre. Par exemple, une écriture, au lieu de se réaliser effectivement dans le fichier et de bloquer le processus qui écrit, est réalisée dans le tampon de sorte que l'écrivain n'est pas bloqué et continuer sa tâche, tandis que de l'autre côté l'opération peut être réalisée lorsque nécessaire et indépendamment.

La bibliothèque du C offre trois modes :

1. **complet** (*fully buffered*). Dans ce mode une écriture s'effectue dans le tampon et lorsque celui-ci est plein, il est alors vidé vers la destination; une lecture s'effectue depuis le tampon et lorsque celui-ci est vide, il est alors rempli depuis la source.
2. **par ligne** (*line buffered*)). Dans ce mode, le critère de vidage ou remplissage est la ligne, c'est-à-dire une suite de données délimitée par un retour à la ligne; lorsque la ligne est trop grande, c'est la taille du tampon qui emporte la mise.
3. **sans tampon** (*unbuffered*). Dans ce mode il n'y a aucune mise en tampon.

Par défaut, les flots obtenus sont en mode complet, sauf dans le cas de flots interactifs (type terminaux) pour lesquels le mode est par ligne.

Le cours
Un peu de C6
C
Pointeurs
Types
Compilation
E/S C
Systèmes de fichiers
E/S Systèmes
Avancé
reste

contrôle du tampon : `setvbuf`, `setbuf`

Definition

```
#include <stdio.h>
int setvbuf(FILE *flot, char *tampon, int
    mode, size_t taille);
void setbuf(FILE *flot, char *tampon);
```

Ces fonctions permettent de positionner un *tampon* de *taille* indiquée pour le *flot* dans le *mode* spécifié. Un appel à `setbuf(flot,tampon);` est équivalent à `setvbuf(flot,tampon,_IOFBF,BUFSIZ);` si $tampon \neq NULL$ et à `flot,tampon,_IONBF,0);` sinon. Si $tampon == NULL$ et que $taille > 0$, `setvbuf` alloue un tampon adéquat.
Ces fonctions renvoient 0 en cas de succès et une valeur non nulle en cas d'échec.

Les valeurs pour le mode sont :

- `_IOFBF` pour le mode complet;
- `_IOLBF` pour le mode par ligne;
- `_IONBF` pour le mode sans tampon.

Important

Les fonctions de contrôle de la temporisation ne peuvent être employées qu'immédiatement après l'obtention du flot. Aucune lecture ou écriture ne doit avoir été effectuée.

Attention

L'emploi, la taille et le mode d'un tampon ont un impact direct sur les performances des opérations...

Exemple

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  if (argc!=3) exit(1);
  FILE *f = fopen(argv[1],"r");
  if (f==NULL) { perror(argv[0]); exit(1); }
  int s;
  s = atoi(argv[2]+1);
  int mode = _IONBF;
  switch (argv[2][0]) {
  case 'F':
    mode = _IOFBF;
    break;
  case 'L':
    mode = _IOLBF;
    break;
  }
  if (setvbuf(f,NULL,mode,s)) fclose(f), exit(8);
  size_t count = 0;
  int c;
  while ( (c=fgetc(f)) != EOF ) count++;
  printf("Read %zd chars\n",count);
  fclose(f);
}
```

Le cours

Un peu de C6

C

Pointeurs

Types

Compilation

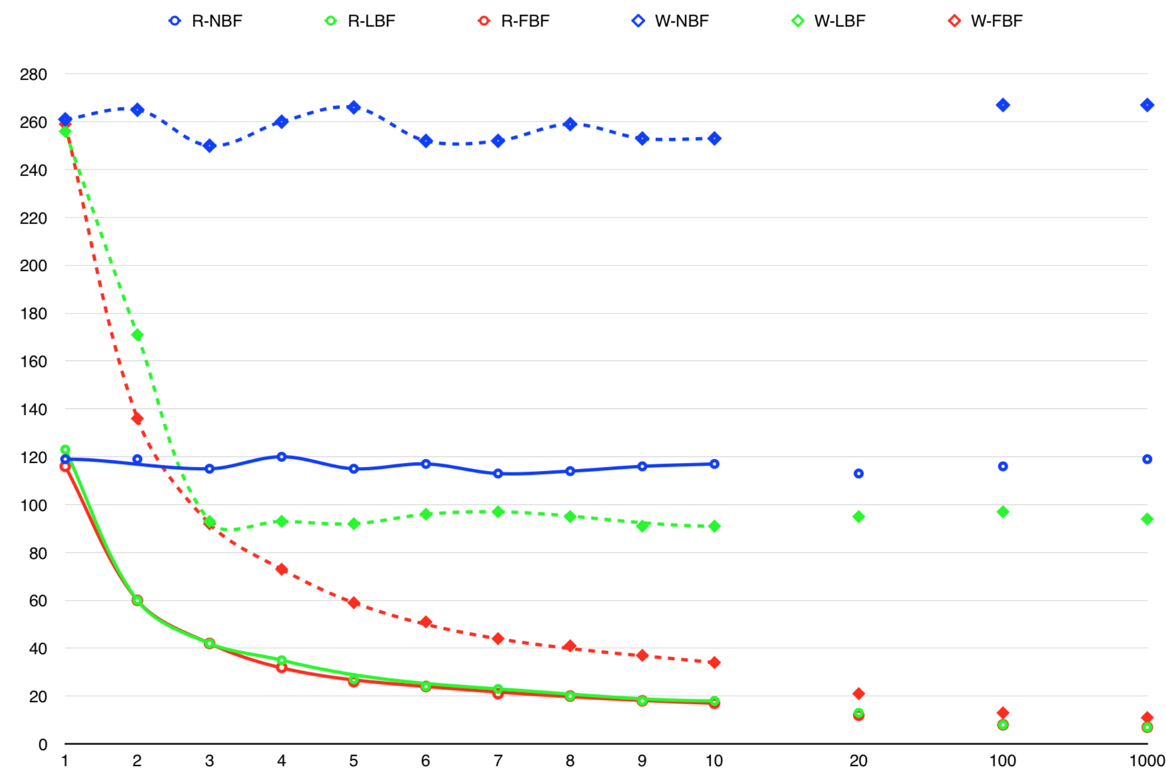
E/S C

Systèmes de fichiers

E/S Systèmes

Avancé

reste



Le cours
Un peu de C6
C
Pointeurs
Types
Compilation
E/S C
Systèmes de fichiers
E/S Systèmes
Avancé
reste

vidage du tampon : `fflush`

Definition (`fflush`)

```c
#include <stdio.h>
int fflush(FILE *flot);
```

permet d'obtenir le vidage du tampon (c'est-à-dire son écriture vers la destination) associé au *flot* si celui-ci correspond à une ouverture avec écriture et que la dernière opération est une écriture, sinon le comportement est indéfini.
Si le *flot* est le pointeur nul, tous les flots existants et correspondant à la description précédente sont vidés.
En cas de succès la fonction renvoie 0 et `EOF` en cas d'échec.

Attention

On ne doit vider que les flots compatibles avec l'écriture!
Le (tristement) célèbre `fflush(stdin);` ne fait pas partie du standard!

Exemple

```
#include <stdio.h>
#include <stdlib.h>

static char *name=NULL;
FILE *w=NULL, *r=NULL;

void clean() {
  if (w!=NULL) fclose(w);
  if (r!=NULL) fclose(r);
  if (name!=NULL) remove(name);
}

void tryReading() {
  FILE *r = fopen(name,"r");
  if (r==NULL) exit(10);
  char data[100];
  if (fgets(data,100,r)==NULL) fprintf(stderr,"empty?");
  else puts(data);
  fclose(r);
}

int main(int argc,char *argv[]) {
  atexit(clean);
  name = tmpnam(NULL);
  if (name==NULL) exit(11);
  FILE *f = fopen(name,"w");
  if (f==NULL) exit(1);
  if (setvbuf(f,NULL,_IOFBF,1000)) exit(8);
  if (fputs("abracadabra",f)==EOF) exit(9);
  tryReading();
  fflush(w);
  tryReading();
}
```

```
$ ./flush
empty?abracadabra
$
```

Attention

Seules les terminaisons par appel implicite ou explicite à `exit` vident les tampons des flots ouverts...
Les terminaisons prématurées et violentes amènent à la perte de données que l'on pensait avoir écrites!

Exemple

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  printf("Salut!");
  int *p=NULL;
  *p = 10;
}
```

```
$ ./noflushinerror
[1]    65115 segmentation fault  ./
   noflushinerror
$
```

Le cours

Un peu de C6

C

Pointeurs

Types

Compilation

E/S C

Systèmes de fichiers

E/S Systèmes

Avancé

reste

Systèmes de fichiers

Le cours
Un peu de C6
C
Pointeurs
Types
Compilation
E/S C
Systèmes de fichiers
E/S Systèmes
Avancé
reste

Fichiers *NIX

Definition (Fichier)

Un fichier est une suite ordonnée d'octets.

Attention

Si en général un fichier est stocké sur un dispositif de stockage permanent, ce n'est en aucun cas requis.
On notera aussi qu'il n'est en aucun cas requis non plus que les octets soient indexés (un flux peut être considéré comme un fichier).
Le contenu des fichiers est fréquemment structuré, c'est-à-dire que le placement des données en son sein a une certaine importance.

Pour le système *NIX il n'y a pas de «fichier» au sens commun. Les concepteurs du système ont souhaité faire apparaître comme «fichier» tout ce qui peut être vu comme suite ordonnée d'octets. Ce qui les différencie c'est éventuellement la structure des données qu'ils contiennent ou la façon d'accéder à leurs données. Le système *NIX distingue différents types (liste non exhaustive):

- les fichiers ordinaires (*regular file*)
- les répertoires (*directory*)
- les périphériques à accès par bloc ou caractère (*block device* ou *character device*)
- les tubes nommés ou anonymes (*FIFO* special files).

On laissera de côté (pour l'instant) la description des deux derniers types pour se concentrer sur les deux premiers.

Ce qu'il est important de noter c'est que les fichiers *NIX sont typés. Leur type (comme tous les types) permet de donner sens à leur représentation ainsi que de définir les manipulations autorisés. Ces objets typés sont appelés i-nœuds (*inodes*).

L'ensemble des i-noeuds est plat, et ils sont identifiés par leur numéro. À toute création de fichier (quelque soit son type) correspond une allocation d'i-nœud. On pourrait donc techniquement accéder à un i-nœud par son numéro mais il faut avouer que ce serait pénible (il faut noter que c'est bien ce que fait le système en interne). C'est pourquoi afin de rendre plus simple la vie de l'utilisateur, le système différencie les fichiers ordinaires des répertoires.

Le cours
Un peu de C6
C
Pointeurs
Types
Compilation
E/S C
Systèmes de fichiers
E/S Systèmes
Avancé
reste

Répertoires et noms de fichiers

Definition (Répertoire et noms de fichiers)

Pour *NIX un **répertoire** est un i-nœud structuré de façon particulière dont l'accès s'effectue à l'aide d'opérations spéciales. Leur structure logique est un ensemble de noms chacun associé à un numéro d'i-nœud. Cette association est appelée **lien physique** (*physical link* ou *link*). Un répertoire est donc un moyen d'associer un nom (**filename**) à un i-nœud. Pour un répertoire donné, deux entrées ne peuvent utiliser le même nom.

⚠ Remarque

D'un point de vue logique, un répertoire ne contient pas de fichier, un répertoire n'est qu'un système d'indexation permettant de retrouver un ensemble de fichiers.

La propriété des systèmes de fichiers *NIX est que tous les i-nœuds alloués sont nécessairement connectés les uns aux autres et que *via* ceux de type répertoire ils forment un arbre enraciné.
D'autre part, tout répertoire contient nécessairement deux liens pour les noms `.` et `..` de sorte que `.` désigne toujours le répertoire lui-même et `..` le répertoire parent.
Par définition la racine de cet arbre est l'i-nœud de numéro 2 (0 et 1 sont réservés à un usage interne).

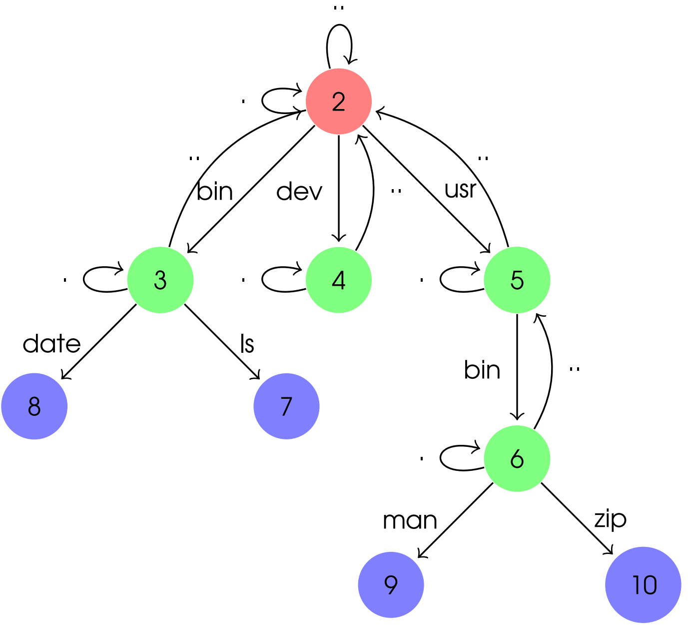
La cohérence de cette structure est essentielle au bon fonctionnement du système, c'est en partie pourquoi l'écriture d'un système est délicate : il faut entre autres s'assurer que quoiqu'il arrive la cohérence est préservée...
Malgré toutes les précautions prises, il est si difficile de garantir cette cohérence pour tout évènement qu'au démarrage d'un système celui-ci prend la précaution de vérifier cette cohérence et éventuellement de la reconstruire ou d'en reconstruire une satisfaisante. Un outil est dédié pour cela : `fsck`.

Note

Pour qu'un i-nœud soit accessible il est nécessaire qu'il possède dans un répertoire un lien physique associé, il faut de plus que ce même répertoire possède la même propriété. L'exception est la racine dont le numéro est toujours 2.

Le cours
Un peu de C6
C
Pointeurs
Types
Compilation
E/S C
Systèmes de fichiers
E/S Systèmes
Avancé
reste

Un arbre de fichiers *NIX pourrait être comme :



Une fois un tel arbre construit, il est aisé de retrouver un i-nœud à partir d'un chemin absolu.
Par exemple, si l'on souhaite accéder au «fichier» `/usr/bin/man`, il faut utiliser l'algorithme de résolution des noms :

- le chemin commence par `/`, il faut donc accéder à l'i-nœud numéro 2,
- on extrait le composant suivant, soit `usr` et on recherche dans la liste d'association de l'i-nœud 2 une association pour le nom `usr`, on trouve (`usr`,5), on accède donc à l'i-nœud numéro 5,
- on extrait le composant suivant, soit `bin` et on recherche dans la liste d'association de l'i-nœud 5 une association pour le nom `bin`, on trouve (`bin`,6), on accède donc à l'i-nœud numéro 6,
- on extrait le composant suivant, soit `man` et on recherche dans la liste d'association de l'i-nœud 6 une association pour le nom `man`, on trouve (`man`,9), on accède donc à l'i-nœud numéro 9.

Cet algorithme de résolution des noms est même documenté dans le manuel de Linux : `man path_resolution` en section 7.
Sa lecture fera apparaître l'omission (volontaire) de certains détails (lesquels apparaîtront plus tard).
Rappelons qu'à tout processus est attaché un répertoire de travail, essentiellement donc le numéro d'i-nœud d'un répertoire qui servira à initier l'algorithme en lieu et place de la racine lorsque qu'un chemin à résoudre est relatif. Il n'y a donc aucune différence d'importance entre les chemins absolus et relatifs pour le système.
On peut aussi observer que les liens `.` et `..` sont résolus comme n'importe quels autres.

Une observation attentive fera apparaître qu'il n'existe pas de chemin unique pour accéder à un i-nœud donné, même si on se restreint aux chemins absolus. Par exemple `/usr/bin/man` et `/bin/../usr/bin/man` désignent le même i-nœud.

Le cours

Un peu de C6

C

Pointeurs

Types

Compilation

E/S C

Systèmes de fichiers

E/S Systèmes

Avancé

reste

La nature des i-nœud

À chaque i-nœud est associé un certain nombre d'informations :

- de quoi retrouver les données du contenu s'il est stocké sur le disque correspondant au système de fichiers,
- des données descriptives de l'i-nœud lui-même (par exemple son type).

Ce qui nous intéresse ici sont ces dernières. L'exploration du système de fichiers avec la commande `ls` va nous y aider.

Exemple

```
$ ls -ail
493477 drwxr-xr-x+ 147 yunes  staff  4998 30 aoû 14:06 .
847427 drwx------+ 178 yunes  staff  6052 30 aoû 17:10 ..
261269 -rw-r--r--+   1 yunes  staff    33  4 jul 15:42 fic
...
$
```

On notera que la commande pourrait être `ls -i -a -l` mais que pour beaucoup de commandes et d'options celles-ci peuvent être combinées. D'autre part leur ordre n'est en général pas significatif.

Quelques options de `ls`

L'option `-a` permet d'obtenir tous les liens d'un répertoire donné lorsqu'on souhaite la liste de ceux contenus dans un répertoire. Par défaut la commande «cache» ceux dont le nom commence par `.`. C'est par pure convention, car sont concernés `.` et `..` dont chacun connaît normalement l'existence mais aussi les fichiers de configurations divers dont la convention veut qu'ils commencent par `.` comme `.emacsrc` ou `.vim` (qui contiennent des configurations pour `emacs` ou `vim`). Comme ils ne sont pas d'usage courant, ils sont donc cachés.
L'option `-i` permet d'obtenir le numéro i-nœud des «fichiers», c'est purement informationnel puisque qu'il n'existe aucun moyen disponible à l'utilisateur d'en faire quoi que ce soit d'intéressant.
L'option `-l` permet de faire apparaître de nombreuses informations descriptives associées aux «fichiers».

Par ordre et par exemple pour :

```
261269 -rw-r--r--+   1 yunes  staff    33  4
    jul 15:42 fic
```

261269 est le numéro d'i-nœud associé au nom `fic` dans le répertoire exploré.
On trouve ensuite 11 symboles `-rw-r-r-+` (c'est particulier à certains systèmes, certains n'en utilisent que 10). Le premier de ceux-ci indique le type de l'i-nœud. Un simple tiret `-` indique un fichier ordinaire. Un `d` désigne un répertoire, un `p` un tube nommé, un `s` une socket, `c` un périphérique en mode caractère, `b` un périphérique en mode bloc, un `l` un lien symbolique, etc.
Les neufs autres qui suivent concernent les droits d'accès `rw-r-r-`, en effet les systèmes *NIX sont multi-utilisateurs et il est par conséquent nécessaire de fournir un mécanisme de protection de données. On reviendra sur ces droits.

Par ordre et par exemple pour :

```
261269 -rw-r--r--+   1 yunes  staff    33  4
    jul 15:42 fic
```

On trouve ensuite un nombre qui est le nombre de liens physiques associés à cet i-nœud, cette information est importante. Elle indique combien d'entrées de répertoires correspondent à cet i-nœud et ceci dans tout le système de fichiers. Ici `1`, ce qui signifie qu'il n'y a qu'une seule façon d'atteindre cet i-nœud (venir dans ce répertoire et retrouver cette entrée).
Ensuite sont les identifiants du propriétaire `yunes` et du groupe `staff` auquel ce fichier appartient (c'est lié aux droits d'accès).

Par ordre et par exemple pour :

```
261269 -rw-r--r--+    1 yunes   staff     33   4
    jul 15:42 fic
```

Le nombre qui suit est la taille `33`, exprimée en octets, du contenu du fichier, si cette information est sensée.
Puis une date `4 jul 15:42`, c'est celle de dernière modification du fichier (il existe d'autres dates associées mais c'est celle-ci qui est généralement la plus utile).
Puis enfin le nom du lien `fic`.

Le cours
Un peu de C6
C
Pointeurs
Types
Compilation
E/S C
Systèmes de fichiers
E/S Systèmes
Avancé
reste

création d'un lien (physique)

Puisqu'un nom de fichier n'est rien d'autre qu'un lien apparaissant (associé à un i-nœud) dans un répertoire, il est possible d'associer plusieurs noms à un i-nœud. Nous insistons sur le fait que les i-nœuds n'ont pas de nom, ce n'est pas une de leurs propriétés; il y a un moyen de les nommer mais qu'il leur est extérieur.

Definition (`ln`)

La commande `ln old new` permet d'obtenir la création d'un lien physique, c'est-à-dire une nouvelle association entre un nom et un i-nœud.

⚠ Attention

Aucun lien n'a de rôle particulier, ils sont tous équivalents...

```
$ ls -ail toto
50833364 -rw-r--r--+ 1 yunes  staff  0 30 ao
   û 11:14 toto
$ ln toto tutu
$ ls -ail toto tutu
50833364 -rw-r--r--+ 2 yunes  staff  0 30 ao
   û 11:14 toto
50833364 -rw-r--r--+ 2 yunes  staff  0 30 ao
   û 11:14 tutu
```

Comme on peut le voir, une fois le lien créé le nombre de lien de l'i-nœud associé est égal à deux. Ces deux liens (noms) désignent le même i-nœud (fichier/contenu).

Il n'y a pas non plus de restrictions sur le nombre de liens (si mais le nombre est très grand) ni l'emplacement de ces liens (si voir plus loin les liens symboliques) :

```
$ ln toto ../tutu
$ ls -ail toto tutu ../tutu
50833364 -rw-r--r--+ 3 yunes  staff  0 30 ao
   û 11:14 ../tutu
50833364 -rw-r--r--+ 3 yunes  staff  0 30 ao
   û 11:14 toto
50833364 -rw-r--r--+ 3 yunes  staff  0 30 ao
   û 11:14 tutu
```

suppression de lien (physique)

Insistons sur le fait qu'aucun lien n'a de rôle particulier.

Definition (`rm`)

Cette commande permet de supprimer un lien.

Attention

`rm` permet de supprimer un lien, pas un fichier!
Si la suppression d'un lien peut entraîner la libération de l'i-nœud (on dit abusivement suppression du fichier), ce n'est que l'effet secondaire de certaines suppressions.

Note

Un i-nœud ne peut être libéré (abusivement «un fichier ne peut être supprimé») qu'à la condition nécessaire qu'il n'existe plus de lien le désignant (on verra plus tard, qu'il existe une seconde condition).

```
$ rm toto
$ ls -ail ../tutu tutu
50833364 -rw-r--r--+ 2 yunes  staff  0 30 ao
   û 11:14 ../tutu
50833364 -rw-r--r--+ 2 yunes  staff  0 30 ao
   û 11:14 tutu
$ rm tutu
$ ls -ail ../tutu
50833364 -rw-r--r--+ 1 yunes  staff  0 30 ao
   û 11:14 ../tutu
$ rm ../tutu
```

À cet instant, l'i-nœud ne possède plus de lien, il pourrait donc être éligible à sa libération.

Les commandes de manipulation des fichiers et répertoires ne manipulent en réalité que les liens, il en va (liste non exhaustive) ainsi de la commande : `mv` qui ne fait que modifier un lien existant ou supprimer un lien et en créer un nouveau (quand ?).

montage

L'arborescence des fichiers accessibles n'est pas toujours réduite qu'à l'arborescence d'un seul système de fichiers.
En effet, on peut vouloir agréger différents systèmes de fichiers en un seul. Ce mécanisme est appelé **montage**. L'idée est d'associer la racine d'un système de fichier à un i-nœud désigné d'un autre système de fichier de sorte que l'on obtient alors une greffe du premier sur le second.
faire une démo...

Le cours
Un peu de C6
C
Pointeurs
Types
Compilation
E/S C
Systèmes de fichiers
E/S Systèmes
Avancé
reste

lien symbolique

Definition (lien symbolique)

Un **lien symbolique** (*symbolic link*) est un i-nœud de type particulier dont le **contenu** est un chemin; un tel «lien» permet donc de s'affranchir des montages (mais cela a des effets pernicieux). On peut en créer *via* la commande `ln -s src dst`.
Dans ce cas *dst* sera un lien symbolique qui désignera *src*.

```
$ ls -ail toto
53269103 -rw-r--r--+ 1 yunes  staff  0 18
   oct 12:45 toto
$ ln -s toto tutu
$ ls -ail toto tutu
53269103 -rw-r--r--+ 1 yunes  staff  0 18
   oct 12:45 toto
53269127 lrwxr-xr-x  1 yunes  staff  4 18
   oct 12:46 tutu -> toto
$
```

On aperçoit donc qu'il s'agit bien de deux i-nœuds différents.

```
$ rm toto
$ ls -ail tutu
53269127 lrwxr-xr-x  1 yunes  staff  4 18
   oct 12:46 tutu -> toto
$
```

L'effet pernicieux est placé ici : on a un i-nœud qui représente un chemin qui ne désigne plus rien (puisque la référence `toto` a disparu).

fonctions POSIX

Le langage C n'est qu'une couche de programmation au-dessus d'un environnement (développement et exécution).
L'environnement d'exécution auquel nous nous intéressons est l'ensemble des systèmes de la famille *NIX. Bien que les variantes de ceux-ci soient nombreuses, ce qui constitue leur famille est le noyau commun de leurs fonctionnalités et de la façon d'y accéder. D'un point de vue «externe», les commandes sont similaires et c'est aussi le cas pour la cas «interne» avec leurs fonctions systèmes (on dit le plus souvent appels systèmes bien que toutes ne le sont pas forcément).

Definition (Appel système)

Un **appel système** (*system call* ou *syscall*) est la technique permettant à un programme d'accéder à un service du noyau. Un tel appel est habituellement strictement équivalent du point de vue syntaxique à un appel de fonction ordinaire, mais la réalisation de cet appel est techniquement assez différent. Il s'agit en effet de passer du monde «protégé» du programme qui s'exécute au monde «ultra-protégé» du code interne du système. Les sas d'accès sont particuliers, et leur existence influe généralement sur l'efficacité du programme (autrement dit il vaut mieux éviter d'en abuser lorsque c'est possible).

Un système conforme à la norme (ou au moins à l'esprit) POSIX (Portable Operating System Interface for Computer environmentS) doit offrir un certain nombre de fonctions (qui peuvent être réalisées autrement que par des appels systèmes).
Nous ne les étudierons pas toutes, mais l'essentiel d'entre elles.

répertoire courant

Toute exécution est à tout instant lié à un **répertoire courant** (*current directory*), c'est-à-dire un répertoire qui est le point de départ des résolutions des chemins relatifs.

Definition (getcwd)

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

permet d'obtenir le chemin absolu du répertoire courant dans l'arborescence des fichiers. *size* est la taille de la zone pointée par *buf* et dans laquelle sera rangée la référence obtenue.
En cas de succès le retour de la fonction est *buf* et sinon `NULL`.
La norme n'indique pas ce qui doit se passer si l'argument fourni par *buf* est le pointeur nul.

Exemple

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  char pwd[1024];
  if (getcwd(pwd,1024)==NULL) {
    perror(argv[0]);
    exit(1);
  }
  puts(pwd);
  exit(0);
}
```

```
$ pwd
/Users/yunes/TeX/Cours/Systemes/src
$ ./mypwd
/Users/yunes/ownCloud/TeX/Cours/Systemes/src
$
```

Ces exécutions ne fournissent pas le même résultat apparent, mais il s'agit en réalité du même répertoire (il y a quelque part un lien symbolique)...

Rappel

En cas d'échec, la variable `errno` contient la cause de l'erreur...

chdir

Tout processus peut changer son répertoire courant en faisant appel à :

Definition (chdir)

```
#include <unistd.h>
int chdir(const char *path);
```

En cas de succès le nouveau répertoire courant est celui spécifié par *path* et le retour de la fonction est 0, et -1 en cas d'échec (+ `errno` adéquat).

Note

Cette fonction ne modifie le répertoire courant que du processus qui appelle cette fonction...

Exemple

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void print_cwd(const char *cmd) {
  char pwd[1024];
  if (getcwd(pwd,1024)==NULL) {
    perror(cmd);
    exit(1);
  }
  puts(pwd);
}

int main(int argc,char *argv[]) {
  print_cwd(argv[0]);
  for (int i=0; i<3; i++) {
    if (chdir("..")!=0) {
      perror(argv[0]);
      exit(2);
    }
    print_cwd(argv[0]);
  }
  exit(0);
}
```

```
$ pwd
/Users/yunes/TeX/Cours/Systemes/src
$ ./mycd
/Users/yunes/ownCloud/TeX/Cours/Systemes/src
/Users/yunes/ownCloud/TeX/Cours/Systemes
/Users/yunes/ownCloud/TeX/Cours
/Users/yunes/ownCloud/TeX
$ pwd
/Users/yunes/TeX/Cours/Systemes/src
$
```