UNSA
Université de Nice Sophia-Antipolis

UFR Sciences
Département Informatique

Licence d'Informatique – Module L3'

Programmation Système dans l'environnement Unix

Roger Rousseau
CNRS-UNSA/I3S

Version 3.1 *Mai 2003*

Chapitre 3

Système de fichiers Unix

1) Organisation générale d'un S.F.
2) Informations sur les i-noeuds : *stat...*
3) Gestion des dates :
   conversions, internationalisation, changements...
4) Changer de propriétaire : *chown...*

Sommaire suite(1)

5) **Effacements :** *truncate, unlink*

6) **Liens :** *link, symlink, readlink...*

7) **Répertoires :** *mkdir, rmdir, opendir, readdir chdir, getcwd...*

8) **Protections :** *access, umask, chmod...*

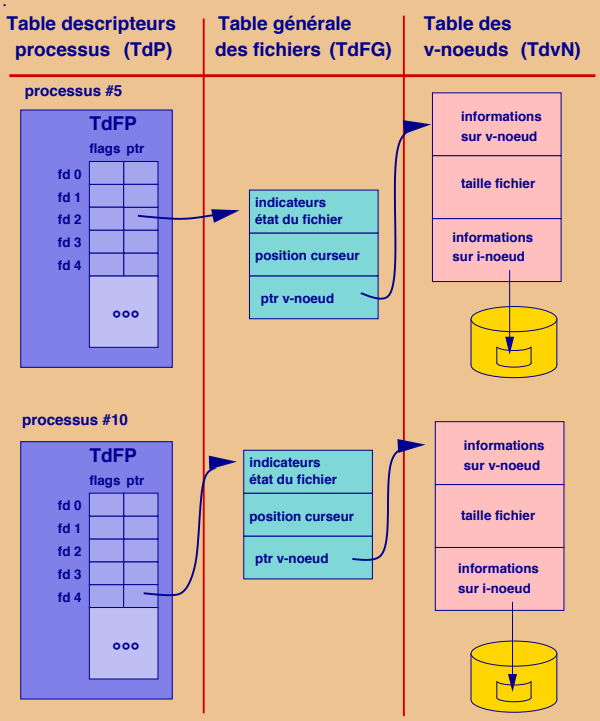
9) **Maj du superbloc :** *sync*

Organisation d'un système de fichiers

Rappels :

*trois niveaux de descripteurs en **mémoire volatile**, qui dépendent de l'usage par les processus...*

Au niveau du disque : l'information est persistante



Organisation d'un système de fichiers suite(1)

systeme de fichiers

Concept de « systeme de fichiers »

- **hiérarchie de noms** de fichiers disques, répertoires, fichiers spéciaux, liens, tubes nommés...
- partitionnement des sous-hiérarchies en **systeme de fichiers** *(file system)* ;
- chaque système de fichiers gère sa propre **allocation du média** (gestion des blocs libres) ;

Organisation d'un système de fichiers suite(2)

- chaque système de fichiers peut être de **technologie différente** des autres : *différents modèles de disques, disquette, CD-ROM..., être local ou distant à travers le réseau (montage NFS)*
- chaque système de fichiers peut avoir des **propriétés globales** spécifiques : *protections, exportations...*
- le **montage/démontage** d'un système de fichiers sur la hiérarchie est dynamique :

requêtes ***mount****(2) /* ***umount****(2)*
commandes mount*(8) /* umount*(8)*

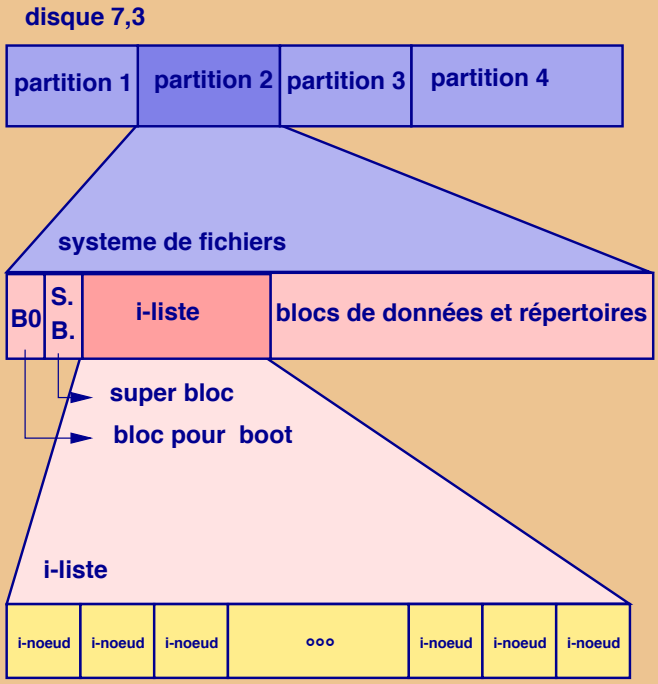
Organisation d'un système de fichiers suite(3)

Disque, partition et systeme de fichiers

Deux niveaux d'organisation :

- *physique :* secteurs, pistes, cylindres ;
- *logique :* suite de **blocs**, unités d'allocation (1Ko) ;

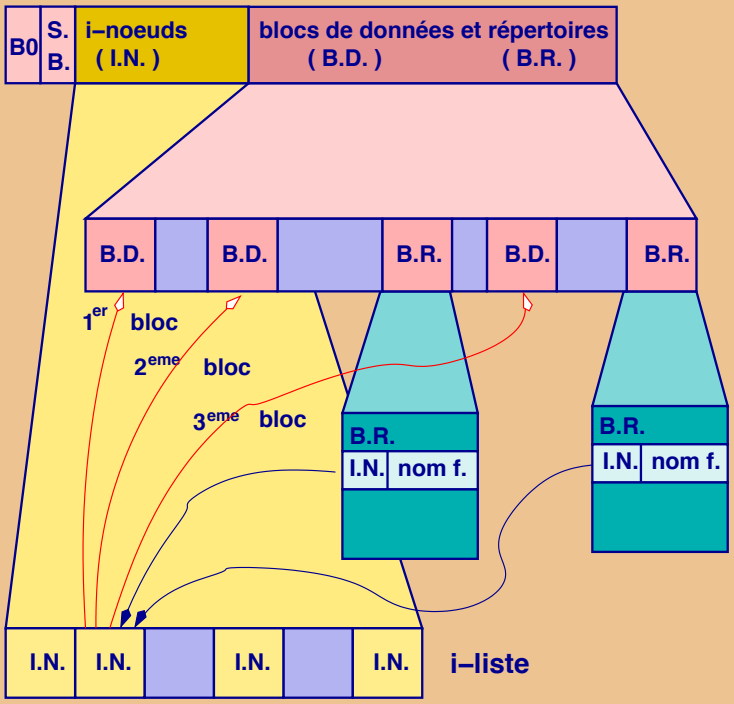
Organisation d'un système de fichiers suite(4)



Organisation d'un système de fichiers suite(5)

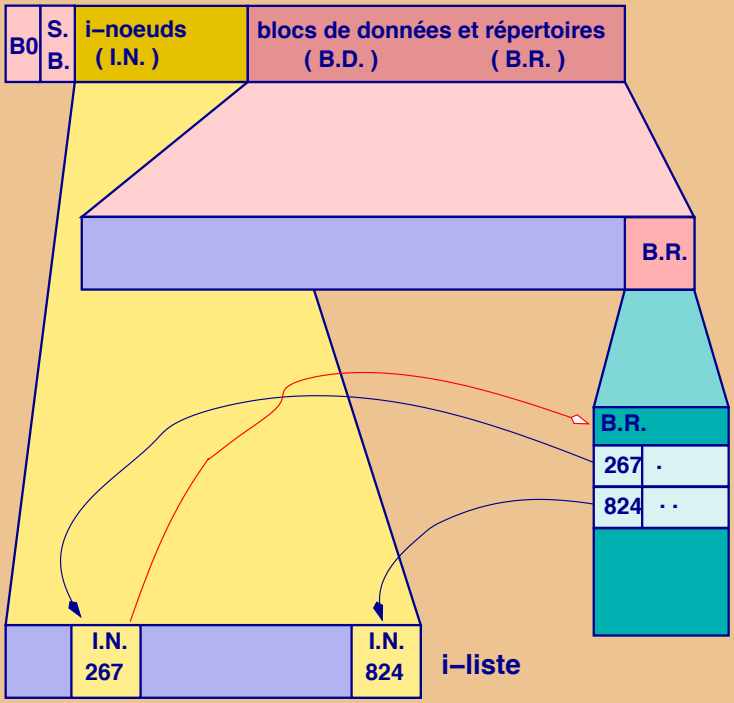
Si nécessaire, le *i-noeud* d'un répertoire peut repérer plusieurs blocs de répertoire.

Les blocs de données et de répertoire sont de même taille.



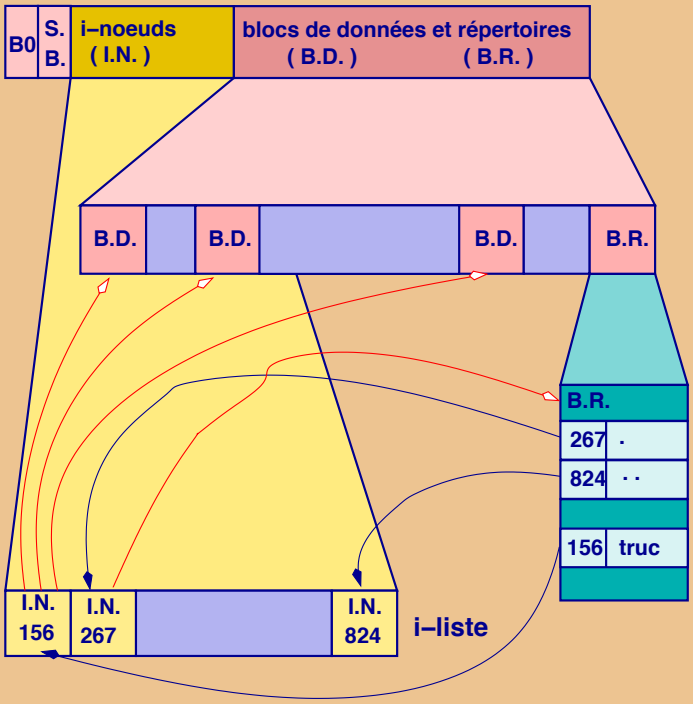
Création d'un fichier ou d'un répertoire

Situation avant la création du répertoire dir1 et du fichier truc.



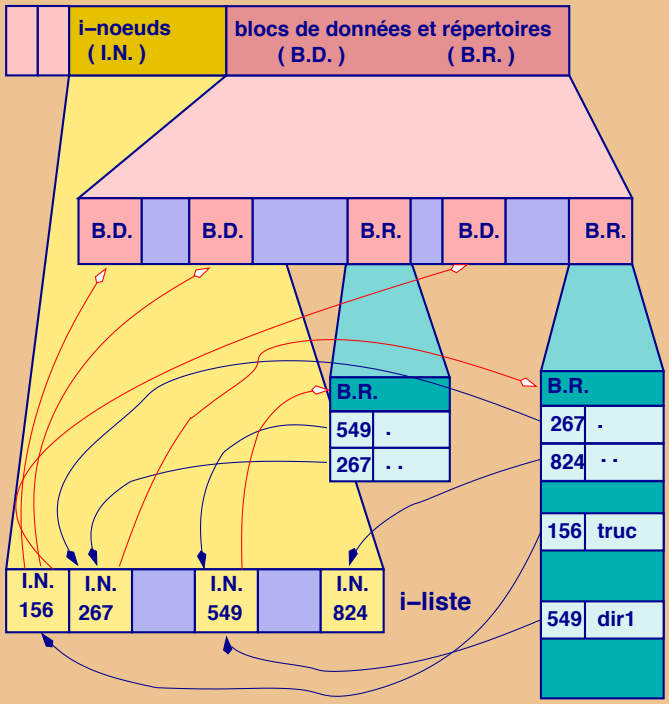
Création d'un fichier suite(1)

Situation après la création du fichier truc.



Création d'un répertoire suite(2)

Après la création du répertoire dir1.



Information sur les i-noeuds : *stat, fstat, lstat*(2)

`Python:help(stat)`
`Ex: os.stat('/')`

- **Syntaxe :**

```
#include <sys/types.h>
#include <sys/stat.h>
int stat ( const char* pathname, struct stat* buf)
int fstat ( int fd, struct stat* buf)
int lstat ( const char* pathname, struct stat* buf)
```

- **Retour :**
  - 0 si OK,
  - -1, si *erreur*

stat, *fstat*, *lstat* suite(1)

- **Sémantique :**
  - Ces 3 primitives rendent l'information d'un *i-noeud* dans la structure *buf* de type *stat*.
  - Comme nombre d'autres primitives de nom **xxx**, **fxxx**, *stat* spécifie le fichier par un nom de chemin alors que *fstat* fournit le *fd* d'un fichier préalablement ouvert.
  - Comme quelques autres primitives, *lstat* spécifie le lien symbolique lui-même, si c'en est un, au lieu du fichier répéré par le lien symbolique.

Définition de *struct stat*

`Python:help(stat_result)`

```
struct stat {
  mode_t  st_mode;    type du fichier et permissions
  uid_t   st_uid;     n° du propriétaire
  gid_t   st_gid;     n° du groupe
  off_t   st_size;    taille en octets du fichier, si régulier
  time_t  st_ctime;   date de la dernière modif du i-noeud
  time_t  st_atime;   date de la dernière lecture du fichier
  time_t  st_mtime;   date de la dernière écriture du fichier
  ino_t   st_ino;     n° du i-noeud
  nlink_t st_nlink;   nombre de liens en dur
...
}
```


struct stat suite(1)

```
...
  dev_t   st_dev;     n° logique (majeur/mineur) du syst.
                      de fichier qui contient le fichier
  dev_t   st_rdev;    n° logique (majeur/mineur)
                      de l'appareil si le fichier est "spécial"
                      (situé dans /dev)
  long    st_blksize; taille des blocs d'allocation
  long    st_blocks;  nombre de blocs de 512 octets alloués
}
```

Chaque modification de ces champs affecte st_ctime.

st_blocks indique l'espace occupé sur le disque, car il ne peut se déduire de st_size, à cause des « trous » éventuels.

Type et permissions : champ *st_mode*

Le champ st_mode est un mot mémoire dont les bits combinent deux informations : le type de fichier et les permissions.

Des macros et des masques définis dans <sys/stat.h> permettent d'interpréter ces bits.

Le masque S_IFMT extrait les bits des différents types de fichiers et des noms symboliques pour chacuns d'eux : S_IFDIR, S_IFREG...

Mais il vaut mieux utiliser les macros POSIX qui suivent :

Numéro de propriétaire : *st_uid*

- c'est un entier naturel, qui identifie chaque utilisateur sur une machine ou un domaine *(si la table des utilisateurs est accessible sur un serveur de yellow pages).*
- toutes les informations sur un utilisateur sont définies dans deux fichiers de texte :
  - /etc/passwd, lisible par tous
  - /etc/shadow, lisible seulement par l'administrateur.

Table des utilisateurs :

/etc/passwd(5), *getpwent*(3)

```
import pwd
pwd.getpwnam('odalle')
```

- c'est un simple fichier de texte, champs séparés par un ":", lisible par tous :

```
root:x:0:0:administrateur:/root:/bin/zsh
...
rr:x:20004:2000:Roger Rousseau:/home/rr:/bin/zsh
```

- On peut obtenir les différents champs par **getpwent** (3) ;
- Autrefois, /etc/passwd contenait aussi les mots de passe encryptés : *aujourd'hui danger de dictionnaires géants.*

Table des mots de passe : /etc/shadow(5)

- c'est un simple fichier de texte, champs séparés par un ":", lisible seulement par l'administrateur :

```
root:1%bU.0dFoh...otUBGYEa1:11638:0:99999:7:::
...
```

- Deux premiers champs : nom de connexion et mot de passe encrypté (*cf.* login)
- Autres champs : durées en jours *(date du dernier changement du mot de passe, durée de validité, limite de validité...)*
- L'administrateur peut obtenir les différents champs par **shadow**(3).

Usager :

getpwuid(3), *struct passwd*

- **Syntaxe :**

```
#include <pwd.h>
struct passwd* getpwuid ( uid_t uid )
struct passwd {
   char*  pw_name;      /* nom de login */
   char*  pw_passwd;    /* mot de passe encrypté */
   uid_t pw_uid;        /* UID */
   gid_t pw_gid;        /* GID */
   char*  pw_gecos;     /* nom en clair (gcos historique)*/
   char*  pw_dir;       /* homedir de l'utilisateur */
   char*  pw_shell;     /* nom d chemin du shell choisi */
}
```


getpwuid suite(1)

```
>>> pwd.getpwnam('odalle')
pwd.struct_passwd(pw_name='odalle',pw_passwd='********',
pw_uid=501,pw_gid=20,pw_gecos='Olivier Dalle',pw_dir='/
Users/odalle', pw_shell='/bin/bash')
```

- **Retour :**
  - Adresse de la structure résultat, si OK,
  - NULL, si *uid* n'existe pas
- **Sémantique :** : décode les informations associées à l'*uid* donné.
- **Remarque :** dans les nouvelles versions de UNIX, le champ pw_passwd n'est accessible qu'à l'administrateur.

Numéro du groupe : *st_gid*

```
import grp
grp.getgrpnam('users')
```

- st_gid :
  - c'est un entier naturel, qui identifie chaque groupe d'utilisateurs sur une machine ou un domaine *(si la table des utilisateurs est accessible sur un serveur de yellow pages).*
  - toutes les infos sur un groupe sont définies dans un fichier de texte /etc/group :
    (*cf. getpwent, p. 147*)
  - pour décoder les informations associées à un *GID* :
    (*cf. getgrgid, p. 152*).

Groupe :

getgrgid(3), *struct group*

- **Syntaxe :**

```
#include <grp.h>
struct group* getgrgid ( gid_t gid )
struct group {
   char*  gr_name;     /* nom du groupe */
   char*  gr_passwd;   /* mot de passe encrypté du groupe */
   gid_t  gr_gid;      /* GID */
   char*  gr_mem;      /* tableau de chaînes terminé par 0 */
                       /* noms de login des membres du groupe*/
};
```

getgrgid suite(1)

```
>>> grp.getgrnam('staff')
grp.struct_group(gr_name='staff',gr_passwd='*',gr_gid=20
,gr_mem=['root'])
```

- **Retour :**
  - Adresse de la structure résultat, si OK,
  - NULL, si *gid* n'existe pas
- **Remarque :** dans les nouvelles versions de UNIX, gr_passwd n'est accessible qu'à l'administrateur.

Numéros logiques d'appareils : *st_dev, st_rdev*

- st_dev :
  majeur/mineur du système de fichiers qui contient le fichier considéré.
- st_rdev :
  majeur/mineur de l'appareil, si le fichier est spécial (normalement placé dans /dev/).

Numéros majeur et mineur

- un n° d'appareil (type dev_t) combine :
  - **majeur**, correspond à la classe de l'appareil (pilote) et donne accès aux primitives de manipulation.
  - **mineur**, correspond à un numéro d'instance de la classe, construit dans le répertoire /dev/ par la commande mknod(8) ou mieux par le script shell /dev/MAKEDEV(8).
  - On peut séparer les deux valeurs par des macros :

```
#define <sys/types.h>
int major(dev_t d)
int minor(dev_t d)
```


Changer le propriétaire et le groupe : *chown, fchown, lchown*(2) `os.chown`

- **Syntaxe :**

```
#include <sys/types.h> #include <unistd.h>
int chown ( const char* pathname,
            uid_t owner, gid_t group )
int fchown ( int fd, uid_t owner, gid_t group)
int lchown (const char* pathname,
            uid_t owner, gid_t group )
```

- **Retour :**
  - 0 si OK,
  - -1, si *erreur*

chown, *fchown*, *lchown* suite(1)

- **Sémantique :**
  - Changent les *UID* (propriétaire) et *GID* (groupe) d'un fichier créé (avec restrictions).
  - Mise à jour de st_ctime,
  - *chown* spécifie le fichier par un nom de chemin et *fchown* par un *fd* de fichier préalablement ouvert.
  - *lchown* considère le lien symbolique lui-même, si c'en est un, au lieu du fichier repéré par celui-ci, comme le fait *chown*.

Effacement d'un fichier : *unlink*(2)

```
os.unlink('tmp/toto')
```

- **Syntaxe :**

  #include <unistd.h>

  int unlink (const char* *pathname*)

- **Retour :**
  - 0 si OK,
  - -1, si *erreur*

unlink suite(1)

- **Sémantique :**
  - Efface le lien en dur de nom *pathname*, si les droits du processus sur le fichier le permettent.
  - Décrémente le nombre de liens du *i-noeud* associé à *pathname*.
  - Si le nombre de liens du *i-noeud* devient nul, efface le *i-noeud* et les blocs de données associés sur le système de fichier.

Troncature un fichier existant : *truncate*, *ftruncate*(2)

`os.truncate()/os.ftruncate()`

- **Syntaxe :**

  #include <sys/types.h> #include <unistd.h>

  int truncate (const char* *pathname*, off_t *length*)
  int ftruncate (int *fd*, off_t *length*)

- **Retour :**
  - 0 si OK,
  - -1, si *erreur*

truncate, *ftruncate* suite(1)

- **Sémantique :**
  - Tronque le fichier à la longueur indiquée.
  - Si la longueur *length* est plus grande que la taille du fichier, cela crée un trou.
  - Comme d'habitude, *truncate* spécifie le fichier par un nom de chemin et *ftruncate* par un *fd* de fichier préalablement ouvert.
  - Le *i-noeud* est inchangé par cette primitive : seuls des blocs de données sont éventuellement libérés.

Établissement d'un lien en dur : *link*(2)

`os.link()`

- **Syntaxe :**

  #include <unistd.h>

  int link (const char* *pathname*, const char* *newpath*)

- **Retour :**
  - 0 si OK,
  - -1, si *erreur*

link suite(1)

- **Sémantique :**
  - Crée un lien en dur de nom *newpath* sur le *i-noeud* atteint par le nom *pathname*.
  - Le nombre de liens du *i-noeud* est incrémenté.
  - *newpath* doit être un nom du **même systeme de fichiers** que le *i-noeud* atteint par *pathname*. *(sinon, il faut utiliser un nom symbolique)*

link suite(2)

Avantages des liens en dur/liens symboliques :

- efficacité : pas besoin d'ouvrir un fichier de lien;
- le changement ou l'effacement du nom *pathname* est sans incidence sur le lien : pas de *"lien en l'air".*
- le compteur de liens permet de savoir s'il y a des aliasings (mais il faut faire un parcours récursif pour les trouver).

Établissement d'un lien symbolique : *symlink*(2)

`os.symlink()`

- **Syntaxe :**

  #include <unistd.h>

  int symlink (const char* *path*,
  const char* *newpath*)

- **Retour :**
  - 0 si OK,
  - -1, si *erreur*

symlink suite(1)

- **Sémantique :**
  - Crée un lien symbolique de nom *newpath* sur le nom *path*.
  - *newpath* est un fichier à part entière, avec son *i-noeud* et son entrée dans un répertoire; les données du *i-noeud* contiennent le nom *path*.

symlink suite(2)

Avantages des liens en dur / liens symboliques :

- lisibilité : on voit l'aliasing de manière symb.
- généralité : on peut poser des liens symboliques sur des fichiers situés sur d'autres systèmes de fichiers, même distants.

Inconvénients des liens symboliques :

- inefficacité, mais très acceptable aujourd'hui (**caches**).
- possibilité d'avoir des "liens en l'air" si le nom d'origine *path* est modifié ou effacé.

Lecture d'un lien symbolique : *readlink*(2)

`os.readlink()`

- **Syntaxe :**

```
#include <unistd.h>
int readlink ( const char* pathname,
               char* buf, int bufsize )
```

- **Retour :**
  - nb d'octets lus si OK,
  - -1, si *erreur*

readlink suite(1)

- **Sémantique :**
  - Lit la chaîne de caractères représentant le lien symbolique *pathname* .
  - Cette primitive est nécessaire car la plupart des primitives réalisent un *“dérepérage”* automatique, en particulier *open*.
  - *readlink* combine un *open* sans dérepérage avec un *read*, comme *symlink* combine un *creat* avec un *write*.

Traitement des liens symboliques

Selon les primitives, certaines manipulent le fichier repéré par un lien symbolique et d’autres le lien lui-même :

Primitive	traite le lien symbolique	traite le fichier repéré par le lien
access		*
chdir		*
chmod		*
chown	*	*
creat		*
exec		*
lchown	*	
link		*

Traitement des liens symboliques suite(1)

Primitive	traite le lien symbolique	traite le fichier repéré par le lien
lstat	*	
mkdir		*
mkfifo		*
open		*
opendir		*
pathconf		*
readlink	*	
remove(3)	*	
rename	*	*
stat		*
truncate		*
unlink	*	

Création d'un répertoire : *mkdir*(2)

`os.mkdir()`

- **Syntaxe :**

  #include <sys/types.h> #include <sys/stat.h>

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

- **Retour :**
  - 0 si OK,
  - -1, si *erreur*

- **Sémantique :**
  - *mkdir* crée un répertoire avec les deux entrées prédéfinies "." et ".."
    Les permissions d'utilisation du répertoire sont précisées par *mode*.

Permissions d'un répertoire suite(1)

Le sens des indicateurs de permission diffère de ceux des fichiers ordinaires :

- S_ISGID indique que les fichiers créés dans ce répertoire hériteront de son GID.
  Exemple : /var/spool/, répertoire d'un projet : ~rr/l3/tps/...
- S_ISVTX indique une restriction pour effacer ou renommer un fichier du répertoire, même si les droits d'écriture sont mis.
  Exemple : /tmp/
- S_IX... indique le droit de traversée du répertoire et non de son exécution.

Lecture d'un répertoire :

opendir, *readdir*, *rewinddir*, *closedir*(3)

`os.listdir()/os.walk()`

- **Sémantique générale :**
  - ces primitives ne sont pas des appels système, mais des primitives de la bibliothèque.
  - un répertoire est d'abord un fichier, avec son *i-noeud* , donc lisible par *read*.

opendir, *readdir*, *rewinddir*, *closedir* suite(1)

- *opendir* ouvre le répertoire, crée une structure abstraite placée dans la mémoire du processus qui gère un curseur secret pour le parcours, puis appelle *rewinddir*.
- *readdir* lit l'entrée courante associée au curseur (s'il en reste à lire), puis avance le curseur sur l'entrée suivante.
- *rewinddir* démarre un nouveau parcours : place le curseur en tête. *(nom mal choisi !!).*
- *closedir* ferme le répertoire et libère la structure abstraite utilisée pour le parcours.

Destruction d'un répertoire : *rmdir*(2)

`os.rmdir()`

- **Syntaxe :**

  #include <sys/types.h>
  #include <sys/stat.h>

  int rmdir (const char* *pathname*)

- **Retour :**
  - 0 si OK,
  - -1, si *erreur*

rmdir suite(1)

- **Sémantique :**
  - *rmdir* décrémente le nombre de liens en dur du répertoire s'il est vide;
  - Comme *unlink*, l'espace disque ne sera libéré que s'il n'y a plus de liens en dur.

Changer le répertoire courant : *chdir, fchdir*(2)

`os.chdir()/os.fchdir()`

- **Syntaxe :**

  #include <unistd.h>
  int chdir (const char* *pathname*)
  int fchdir (int *fd*)

- **Retour :** 0, si *OK*, -1, si *erreur*
- **Sémantique :**
  - *chdir* change le répertoire de travail du processus courant.
  - *fchdir* agit comme *chdir*, mais à partir d'un *fd*.

Donner le répertoire courant : *getcwd*(2)

`os.getcwd()`

- **Syntaxe :**

  char* getcwd (char* *buf*, size_t *size*)

- **Retour :**
  - adresse de buf si OK,
  - -1, si *erreur*

- **Sémantique :**
  - *getcwd* donne le nom de chemin absolu du répertoire de travail.
  - L'appel doit fournir le tampon pour récupérer le résultat.

Changer le nom ou la position d'un fichier : *rename*(2)

`os.rename()`

- **Syntaxe :**

  int rename (const char* *oldpath*, const char* *newpath*);

- **Retour :**
  - 0, si *OK*,
  - -1, si *erreur* + *errno*

- **Sémantique :**
  - renomme un fichier (ou un lien symbolique), en le déplaçant éventuellement vers un autre répertoire.
  - écrase le lien *newpath* si nécessaire.
  - c'est la primitive utilisée par la commande `mv`.