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...*

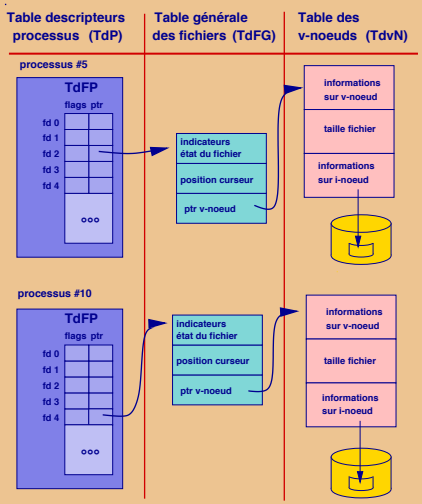
Sommaire suite(1)

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)*

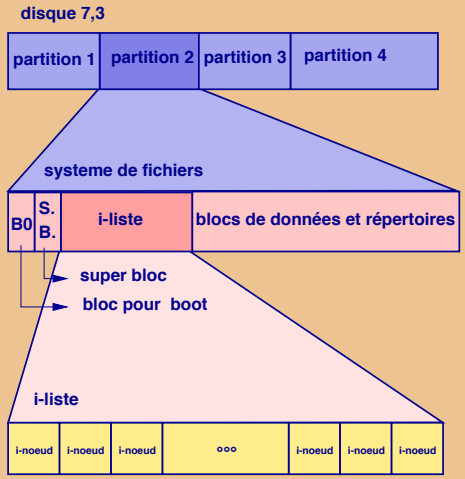
Organisation d'un système de fichiers suite(3)

Disque, partition et systeme de fichiers

Deux niveaux d'organisation :

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

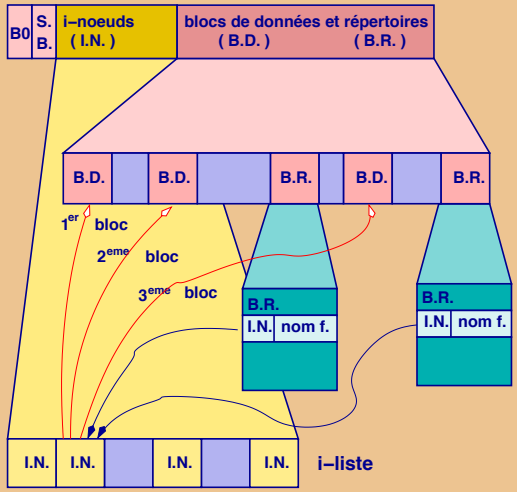
Organisation d'un système de fichiers suite(4)



Organisation d'un système de fichiers suite(5)

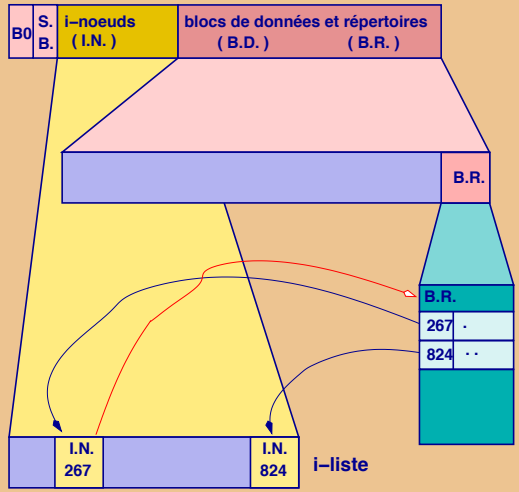
Si nécessaire, le *i-noeud* d'un répertoire peut repérer plusieurs blocs de répertoire.

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



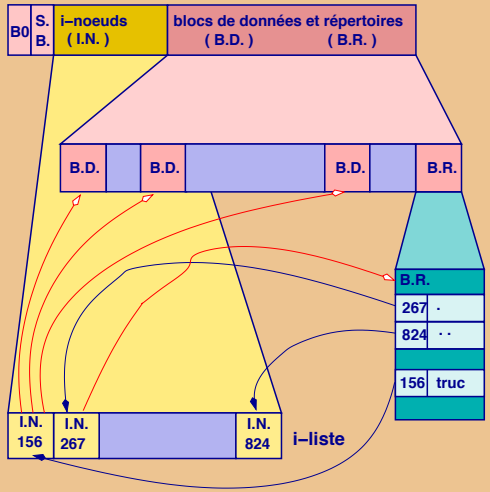
Création d'un fichier ou d'un répertoire

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



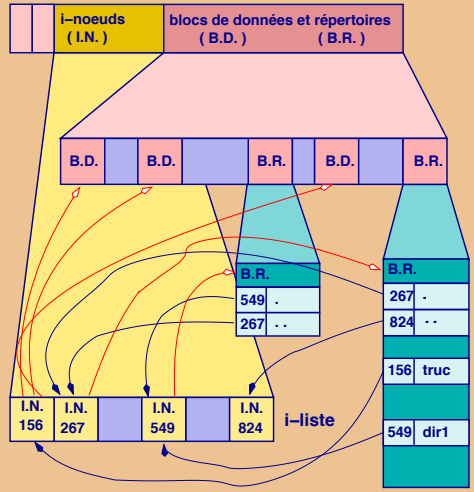
Création d'un fichier suite(1)

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)

- **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*

```
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 :

champ *st_mode* suite(1)

- S_ISREG(mode_t t) /* *fichier régulier*/
  S_ISDIR(mode_t t) /* *répertoire*/
  S_ISCHR(mode_t t) /* *fichier spécial, (mode caract.)*/
  S_ISBLK(mode_t t) /* *fichier spécial, (mode blocs)*/
  S_ISFIFO(mode_t t) /* *tube nommé (FIFO)*/
  S_ISLNK(mode_t t) /* *lien symbolique*/
  S_ISSOCK(mode_t t) socket
- Ces macros sont définies ainsi :

```
#define S_ISDIR(mode) \
        (((mode) & S_IFMT) == S_IFDIR)
...
```


champ *st_mode* suite(2)

- **mode_t :** (*cf. open, p. 80*)

S_ISUID	set-user-ID
S_ISGID	set-group-ID
S_ISVTX	saved-text (*sticky bit*)
S_IRWXU	*read*, *write*, execute by user
S_IRUSR	*read* by user
S_IWUSR	*write* by user
S_IXUSR	execute/cross by user

champ *st_mode* suite(3)

S_IRWXG	*read*, *write*, execute by group
S_IRGRP	*read* by group
S_IWGRP	*write* by group
S_IXGRP	execute/cross by group
S_IRWXO	*read*, *write*, execute by other
S_IROTH	*read* by other
S_IWOTH	*write* by other
S_IXOTH	execute/cross by other

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)

- 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)

- **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*

- 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)

- **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)
```


Manipulations de dates

Il y a trois dates de fichiers dans les *i-noeud* s :
champs st_atime, st_mtime, st_ctime

Le système Unix enregistre trois dates GMT dans chaque *i-noeud* , exprimées par le nombre de secondes depuis le 01/01/1970 à minuit GMT.

Manipulations de dates suite(1)

Champ du *i-noeud*	Signification	primitives pouvant affecter la date	Option de ls montrant cette date
st_atime	date de dernière lecture des donnée du fichier	*read*	-u
st_mtime	date de dernière écriture des données du fichier	*write*	par défaut
st_ctime	date de la dernière modif des attributs du *i-noeud*	*creat*, *chmod*, *chown*...	-c

N.B. : La lecture d'un *i-noeud* ne modifie pas sa date st_atime :
elle n'est modifiée que par la lecture des données du fichier.

Obtenir la date et l'heure : *time*(2)

- **Syntaxe :**

```
#include <time.h>
time_t time ( const time_t* time)
```

- **Retour :**
  - l'heure et la date du moment, si OK,
  - (time_t) -1, si *erreur*
- **Sémantique :**
  - si *time* est NULL, rend la date et l'heure actuelle en temps GMT universel
  - de plus, si *time* est non NULL, remet à l'heure et à la date indiquée dans *time* (*cf.* exemple p.).

Décodage numérique d'une date : *localtime*(3), *gmtime*(3)

- **Syntaxe :**

```
#include <time.h>
struct tm* localtime ( const time_t* time)
struct tm* gmtime ( const time_t* time)
```

- **Retour :**
  - un pointeur sur struct tm si OK,
  - NULL sinon

localtime, *gmtime* suite(1)

- **Sémantique :**
  - *gmtime* rend la date en temps GMT universel
  - *localtime* rend la date en temps local, en tenant compte :
    - du fuseau horaire (défini à la configuration du système),
    - de l'heure d'été ou d'hiver (si cela existe).

Définition de *struct tm*

- **Syntaxe :**

```
struct tm {
  int tm_sec;     nb de secondes: [0 .. 59]
  int tm_min;     nb de minutes: [0 .. 59]
  int tm_hour;    nb de heures: [0 .. 23]
  int tm_mday;    n° du jour dans le mois: [1 .. 31]
  int tm_mon;     n° du mois dans l'année: [0 .. 11]
  int tm_year;    n° de l'année depuis 1900
  int tm_wday;    n° du jour depuis dim.: [0 .. 6]
  int tm_yday;    n° du jour depuis 1er jan: [0 .. 365]
  int tm_isdst;   heure d'été (>0) d'hiver (=0) sans objet (<0)
};
```

Décodage numérique d'une date : *mktime*(3)

- **Syntaxe :**

  #include <time.h>

  time_t mktime (struct tm* *tmptr*)

- **Retour :**
  - time_t : nb de secondes si OK,
  - -1 sinon
- **Sémantique :**
  - *mktime* convertit une date struct tm en time_t.

Décodage d'une date en une chaîne : *ctime, asctime, ctime_r, asctime_r*(3)

- **Syntaxe :**

  #include <time.h>

  char* ctime (const time_t* *t*)

  char* asctime (const struct tm* *tptr*)

  char* ctime_r (const time_t* *t*, char* *buf*)

  char* asctime_r (const struct tm* *tptr*, char* *buf*)

- **Retour :**
  - chaîne résultat, si OK,
  - NULL, si *erreur*

ctime, asctime, ctime_r, asctime_r suite(1)

- **Sémantique :**
  - décode la date fournie en une chaîne standard.
  - *ctime* et *asctime* allouent dynamiquement la place pour le résultat.
  - *ctime_r* et *asctime*lr fournissent le tampon pour le résultat.
  - Le résultat est internationalisé (*cf. setlocale, p. 173*).
  - date(1) peut s'écrire : *printf*("%s\n", *ctime*(*time*()))

Décodage d'une date en une chaîne formatée : *strftime*(3)

- **Syntaxe :**

  #include <time.h>

  size_t strftime (char* *buf*, size_t *maxbuf*,
  const char* *format*, const struct tm* *timeptr*);

- **Retour :**
  - nb de bytes du résultat écrits dans *buf*, si OK,
  - 0, si *erreur*.
- **Sémantique :**
  - décode la date *timeptr* en une chaîne dans *buf*, de capacité *maxbuf*, selon un format à la *printf*, dans la langue choisie par le programmeur ou l'utilisateur.

strftime suite(1)

- spécifieurs de *strftime* :
  - `%A` ou `%a` : nom du jour dans la semaine ou en abrégé,
  - `%B` ou `%b` : nom du mois ou en abrégé,
  - `%d` : numéro du jour dans le mois,
  - `%y` : numéro de l'année,
  - `%m` : numéro du mois dans l'année,
  - `%H`, `%M`, `%S` : heures, minutes, secondes
  - et bien d'autres formats... *cf.* man 3 *strftime*

Principes de l'internationalisation : locale(7), locale(5), *setlocale*(3), strcoll(3)...

Cela permet de produire des logiciels pour différents pays, avec des messages dans différentes langues :
messages à l'utilisateur, manuels et documentations, dates, signes monétaires, représentation des nombres....

On peut ainsi, sans rien changer à un logiciel, l'adapter automatiquement aux usages locaux d'un pays utilisateur.

Principes de l'internationalisation suite(1)

Il suffit, généralement, de placer dans des variables d'environnement les choix de l'utilisateur pour chaque catégorie :

- LC_TIME : convention utilisée pour les dates,
- LC_MONETARY, LC_COLLATE, LC_CTYPE, LC_NUMERIC, LANG : signe monétaire, jeu de caractères, ordre alphabétique, signe décimal, langage...
- LC_ALL : le même choix pour toutes les catégories.

Principes de l'internationalisation suite(2)

On met dans ces variables des identificateurs prédéfinis :

- en, pour l'anglais,
- fr_FR, pour le français de France,
- de, pour l'allemand...

Principes de l'internationalisation suite(3)

Exemple en anglais :

```
% export LC_ALL=en
% date
Thu Mar 23 16:03:59 CET 2000
% cat /bin/zzz
cat: /bin/zzz: No such file or directory
```


Principes de l'internationalisation suite(4)

Exemple en français :

```
% export LC_ALL=fr_FR # choix du français de France
% date
jeu mar 23 16:04:12 CET 2000
% cat /bin/zzz
cat: /bin/zzz: Aucun fichier ou répertoire de ce type.
```


Principes de l'internationalisation suite(5)

Exemple en allemand :

```
% export LC_ALL=de # choix de l'allemand
% date
Die März 23 16:04:16 CET 2000
% cat /bin/zzz
cat: /bin/zzz: Datei oder Verzeichnis nicht gefunden
```


Exemple d'internationalisation

```
#include <time.h> #include <locale.h>
#define MAX 80
main() {
    char datestr[MAX];
    time_t datebin;
    const struct tm* datetm;
    /* Choix de la langue: "" indique que c'est l'utilisateur
     * qui la choisit dans sa variable d'environnement LC_ALL */
(void) setlocale (LC_ALL, "");
    /* setlocale(LC_ALL, "en") permettrait au programmeur
     * d'imposer l'anglais, ou "fr" pour le français... */
```

Exemple d'internationalisation suite(1)

```
    /* Demander la date actuelle */
    if (time (&datebin) == (time_t) -1) ERRNO_EXIT;
    /* Conversion de la date en format numérique struct tm */
    datetm= localtime (&datebin);
    /* Conversion de la date en chaîne, dans la langue locale */
    if (strftime (datestr, MAX, "%A %x", datetm)
            == (size_t)0) ERRNO_EXIT;
    /* Imprimer la date */
    printf("Date du jour: %s\n", datestr);
}
```


Fin de séance n° 3
(année : 2004)

Fin de séance n° 4
(année : 2004)

Changer les dates des fichiers :
utime(2), struct utimbuf

- **Syntaxe :**

```
#include <sys/types.h> #include <utime.h>
int utime ( const char* pathname,
                const utimbuf* times )
struct utimbuf {
    time_t actime ;    /* date de dernière lecture*/
    time_t modtime ;/* date de dernière écriture*/
}
```

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

utime suite(1)

- **Sémantique :**
  - Corrige les dates st_atime et st_mtime du *i-noeud* avec celles fournies dans *times* (struct utimbuf).
  - Met à jour st_ctime.
  - Si *times* est NULL, les dates choisies sont celles de l'instant de la primitive.
  - Il faut être *SU* , propriétaire ou ayant droit d'écriture du fichier pour modifier ces dates.
  - Les primitives *ctime*(3) permettent d'interpréter en clair ou de coder ces dates.

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

- **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)

- **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)

- **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)

- **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)

- **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)

- **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)

- **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)

- **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.

opendir(3)

- **Syntaxe :**

```
#include <sys/types.h>
#include <dirent.h>
DIR * opendir ( const char* pathname)
```

- **Retour :**
  - pointeur si OK,
  - NULL, si *erreur*
- **Sémantique :**
  - Ouvre le répertoire en lecture et rend un pointeur sur une structure abstraite struct DIR.

readdir(3), *struct dirent*

- **Syntaxe :**

```
struct dirent* readdir (DIR* dp)
struct dirent {
    ino_t d_ino{dirent} ; /* i-noeud */
    char d_name [NAME_MAX+1]; /* filename*/
    /* autres champs (non POSIX) ~: cf. <dirent.h>*/
}
```

- **Retour :**
  - ptr si OK,
  - NULL si fin de parcours,
  - -1, si *erreur*

readdir suite(1)

- **Sémantique :**
  - Rend une adresse de struct dirent qui contient toutes les informations sur l'entrée courante du répertoire, repérée par le curseur *dp* (struct DIR).
  - Avance le curseur sur l'entrée suivante du répertoire.

rewinddir(3)

- **Syntaxe :**
  void rewinddir (struct DIR* *dp*)
- **Retour :** pas d'erreur détectée,
- **Sémantique :**
  - Place le curseur de parcours de la structure abstraite struct DIR repérée par *dp* (initialisé par un *opendir* précédent) sur la 1ère entrée du répertoire.

closedir(3)

- **Syntaxe :**
  int closedir (DIR* *dp*)
- **Retour :**
  - 0 si OK,
  - -1, si *erreur*
- **Sémantique :**
  - Ferme le répertoire et libère la structure repérée par *dp* (struct DIR).

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

- **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)

- **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)

- **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)

- **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`.

Gestion des protections

Le droit d'un processus à manipuler un fichier dépend :

- **des identités** des responsables du processus : *propriétaire ou son groupe, réel ou effectif... ;*
- **de la manipulation** envisagée : *lecture, écriture, exécution, traversée, effacement, création, changement d'un droit...*
- **du type de fichier :** *régulier (ordinaire) ou répertoire ;*
- **des permissions** placées dans le *i-noeud* du fichier : S_IRGRP, S_IWOTH, S_IXUSR, S_ISUID...

Identités associées au processus

Identités réelles :

- obtenues lors de login(1) (jamais modifiées, sauf si *SU*) :

RUID	real user ID	identité **user**
RGID	real group ID	identité **group**

Identités associées au processus suite(1)

Identités effectives :

- utilisées pour les protections, (peuvent varier)

EUID	effective user ID	identité **user**
EGID	effective group ID	identité **group**
ESGID	supplementary group IDs	groupes auxquels l'utilisateur appartient

Identités associées au processus suite(2)

Identités effectives sauvegardées :

- sauvegardées lors des changements d'identité :

SUID	saved set-user-ID	identité **user**
SGID	saved set-group-ID	identité **group**

Changements des identités effectives

***EUID* et *EGID* peuvent changer :**

- avec exec : si le programme à charger a une permission S_ISUID / S_ISGID (set-user-ID / set-group-ID).
- avec les primitives *setuid* ((*cf. setuid, p. 271*)), *setreuid* (resp. *setgid*, *setregid*) :
  - si *EUID* = *SU* ⇒ aquisition nouvelle identité,
  - sinon, si l'identité demandée est acceptable : *RUID* ou *SUID* (resp. *RGID* ou *SGID*) *(retour à 1 identité précédente)*

Changements des identités réelles

***RUID* et *RGID* peuvent changer avec :**

- *setuid* ou *setgid* avec *EUID* == *SU* :
  - ils prennent la valeur demandée, ainsi que *EUID* et *RUID* (resp. *EGID* et *RGID*).
  - login(1) utilise cette possibilité lors de la connexion pour fixer les identités réelles.
- *setreuid* ou *setregid* avec *EUID* $\neq$ *SU* :
  - ils peuvent échanger leur valeur avec *EUID* (resp. *EGID*).
  - Cela permet de retrouver les identités antérieures.

Changements des identités de sauvegarde

***SUID* et *SGID* peuvent changer avec :**

- *setuid* ou *setgid* :
  - *EUID* = *SU* : ils prennent la valeur demandée ; (comme *EUID* et *RUID* (resp. *EGID* et *RGID*))
  - *EUID* $\neq$ *SU* : ils restent inchangés ;
- exec : si le programme à charger a S_ISUID / S_ISGID, ils prennent la valeur *EUID* /*EGID* d'avant l'exec.
- *setreuid* ou *setregid* avec *EUID* $\neq$ *SU* : chaque fois que *EUID* /*EGID* prend une nouvelle valeur, la valeur d'origine est copiée aussi dans *SUID* / *SGID* .

Contrôle de l'usage d'un fichier

Symbole	Nom	Action visée	Qui le peut ?
S_ISUID S_ISGID S_ISVTX	set-user-ID set-group-ID *sticky bit*	changer *EUID* changer *EGID* copier le code dans zone swap	ceux qui peuvent exécuter le fichier
S_ISRUSR S_ISWUSR S_ISXUSR	user-read user-write user-execute	lecture écriture exécution	*SU* , et le propriétaire
S_ISRGRP S_ISWGRP S_ISXGRP	group-read group-write group-execute	lecture écriture exécution	*SU* , et le groupe
S_ISROTH S_ISWOTH S_ISXOTH	other-read other-write other-execute	lecture écriture exécution	autres que groupe et propriétaire

Contrôle de l'usage d'un répertoire

Symbole	Nom	Action visée	Qui le peut ?
S_ISUID	set-user-ID	non utilisé	
S_ISGID	set-group-ID	force le GID des futurs fichiers créés au GID du rép.	ceux qui peuvent créer dans le répertoire
S_ISVTX	*sticky bit*	Restreint le droit effacer (/tmp/, /var/spool/)	propriétaire du fichier ou du répertoire

Contrôle de l'usage d'un répertoire suite(1)

Symbole	Nom	Action visée	Qui le peut ?
S_ISRUSR S_ISWUSR S_ISXUSR	user-read user-write user-execute	lecture (ls) créer, effacer traverser	*SU* , et le propriétaire
S_ISRGRP S_ISWGRP S_ISXGRP	group-read group-write group-execute	lecture (ls) créer, effacer traverser	*SU* , et le groupe
S_ISROTH S_ISWOTH S_ISXOTH	other-read other-write other-execute	lecture (ls) créer, effacer traverser	autres que groupe et propriétaire

Remarques

- Toute requête qui utilise un nom de chemin exige le droit de traverser de **tous les répertoires** cités dans ce nom.
- Pour effacer un fichier, il faut le droit de **traverser et d'écriture sur le répertoire**, pas sur le fichier !

Remarques suite(1)

- Les trois identités **effectives** du processus (*EUID* , *EGID* , *ESGID*) permettent de placer le processus dans **la premiere** des quatre classes suivantes :
  - *SU*
  - propriétaire du fichier (ou du répertoire),
  - groupe du propriétaire du fichier (ou du répertoire),
  - autres utilisateurs
- Seule **la classe** du processus est considérée, sinon rejet. Ainsi, un propriétaire qui s'est interdit un droit ne l'obtient pas, *même si son groupe ou les autres l'ont!*

Sticky-bit (S_ISVTX)

- Le ***sticky bit*** (S_ISVTX) permettait autrefois d'accélérer le chargement des gros programmes très utilisés : cc, emacs... Il n'est plus utile avec le « *fast file system* » de BSD.
- Pour un répertoire $\mathcal{R}$ avec droit d'écriture pour tous, le *sticky bit* interdit l'effacement des fichiers $\mathcal{F}$ sous $\mathcal{R}$, si l'on n'est propriétaire ni de $\mathcal{F}$, ni de $\mathcal{R}$ (e.g. /usr/tmp/, /usr/spool/lpd...)

Set-group-ID

- Le S_ISGID d'un répertoire $\mathcal{R}$ permet de conserver le même *GID* pour tous les fichiers créés sous $\mathcal{R}$.
- Le S_ISGID d'un fichier sert :
  - soit à changer de *EGID* si le fichier est exécutable,
  - soit à armer le système de verrouillage renforcé (mandatory locking) si le fichier n'est pas exécutable.

    (cf. chapitre 9, 764)

    (un fichier est exécutable s'il a au moins une permission d'exécution)

Test des droits d'utilisation : *access*(2)

- **Syntaxe :**

  #include <unistd.h>

  int access (const char* *pathname*, int *mode*)
- **Retour :**
  - 0, si OK et vrai,
  - -1, si *erreur* ou faux

access suite(1)

- **Sémantique :**
  - Teste si le processus dispose du droit indiqué par le paramètre *mode* sur le fichier.
  - Les valeurs possibles pour *mode* sont les suivantes :
    - R_OK droit de lecture,
    - W_OK droit d'écriture,
    - X_OK droit d'exécution ou de traversée
    - F_OK test d'existence

Contrôle des droits de création de fichiers : *umask*(2)

- **Syntaxe :**

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

  mode_t umask (mode_t *new_mask*)
- **Retour :** mode de création de fichier, avant l'appel,
- **Sémantique :**
  - Change l'*umask* du processus et rend l'ancien.
  - Chacun des 9 booléens de l' *umask* indique une **interdiction** pour les fichiers créés par le processus : rwx pour user, group, other.

umask suite(1)

- **Sémantique (suite) :**
  - Ces indicateurs sont définis par les constantes :
    S_IRGRP | S_IWGRP | S_IROTH... (*cf.* p. 144)
  - L'*umask* fournit à **l'utilisateur** un moyen de **contrôle global** des permissions des fichiers créés par ses processus, quels que soient les droits proposés par le programmeur dans l'argument *mode* d'*open* ou *creat*.

Changer les protections d'un fichier : *chmod*, *fchmod*(2)

- **Syntaxe :**
  #include <sys/types.h> #include <sys/stat.h>
  int chmod (const char* *pathname*, mode_t *mode*)
  int fchmod (int *fd*, mode_t *mode*)
- **Retour :**
  - 0 si OK,
  - -1, si *erreur*

chmod, *fchmod* suite(1)

- **Sémantique :**
  - Change les 12 permissions d'un fichier créé, selon les valeurs de *mode* : 3 indicateurs spéciaux et 9 permissions rwx pour user, group, other.
  - Le mode est spécifié comme d'habitude par une expression de constantes symboliques :
    S_ISUID | S_IRGRP | S_IROTH... (*cf.* p. 144)
  - Comme d'habitude, *chmod* spécifie le fichier par un nom de chemin et *fchmod* par un *fd* de fichier préalablement ouvert.

Écriture des caches sur les disques : *sync*, *fsync*(2)

- **Syntaxe :**
  #include <unistd.h>
  void sync (void)
  int fsync (int *fd*)
- **Retour :**
  - 0 si OK,
  - -1, si *erreur*

sync, *fsync* suite(1)

- **Sémantique :**
  - *sync* synchronise les tampons d'écriture des caches avec les disques. La commande `sync`(8) fait appel à cette requête.
  - *fsync* n'écrit que les tampons du fichier ouvert *fd*.
  - L'intérêt de différer les écritures sur le disque est d'accélérer les échanges et d'optimiser les déplacements des têtes.
  - Mais il faut synchroniser le disque et les tampons avant chaque arrêt et régulièrement (30 secondes) pour se protéger d'une panne éventuelle.