Sytèmes 1 — Fichiers

November 19, 2012

Contents

0.1 Histoire

Le premier *s5fs* (System V file system). 4.2BSD introduit *FSF* (Fast File System) connu aussi comme *ufs* (Unix file system).

1 Panic

```
/***************** panic.h ********************/
#ifndef PANIC_H
#define PANIC_H
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#define PANIC( fin )  do{                              \
    fprintf(stderr,\
      "\n error in file %s in line %d : %s\n",\
      __FILE__,__LINE__,strerror(errno)); \
    if (fin > 0) exit(fin);                    \
   } while(0)
#endif
```

`EXIT_FAILURE` et `EXIT_SUCCESS` dans `<stdlib.h>` (c'est du C standard).

2 Gestion d'erreurs sous UNIX

Si une erreur survient pendant l'exécution d'une fonction de système UNIX souvent la fonction retourne une valeur négative et `errno` donne le code d'erreur. Les codes d'erreur et `errno` sont définis dans le fichier `<errno.h>`. La définition historique de `errno` est

```
extern int errno;
```

mais avec les threads chaque thread peut avoir ses propre erreur donc `errno` n'est peut pas être une variable (sinon `errno` serait partagé entre les threads).

Deux règles d'utilisation de `errno`.

(1) La valeur n'est jamais mis automatiquement à 0 s'il n'y a pas d'erreur. Pour cette raison on examine `errno` uniquement si la valeur de retour signale une erreur.

(2) Il n'y a pas de code d'erreur 0. Donc nous pouvons toujours mettre 0 dans `errno` sans que cela soit confondu avec un code d'erreur.

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

transforme le numéro d'erreur en message.

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

affiche `msg` concaténé au message d'erreur.

3 La structure de système de fichiers sous Linux.

Le système de fichiers UNIX est hiérarchisé et composé de fichiers et de répertoires. La hiérarchie commence avec le répertoire racine (root) dont le nom est `/` . Deux noms sont créés dans chaque répertoire à sa création : `.` (dot) et `..` (dot-dot), le premier fait référence au répertoire courant le deuxième au répertoire parent. (Dans le répertoire racine et `..` (dot-dot) et `.` (dot) pointent les deux vers le répertoire courant.)

Les chemins absolus se sont les chemin qui commencent par `/` comme `/usr/bin/ash`. Le chemin relatif ne commence pas par `/`. Le chemin relatif est résolu par rapport au répertoit courant du processus[1]

Par exemple si le processus utilise le chemin relatif `../toto/bin/cfd.c` et le répertoire courant est `/home/kowalski` alors le chemin absolu correspondant est `/home/kowalski/../toto/bin/cfd.c`.

Différents types de fichiers que nous pouvons rencontrer : fichiers reguliers, répertoire, liens symboliques (soft links), fichiers spéciaux comme les fichiers dans le répertoire `/dev`, tubes nommés, les sockets etc.

Le type de fichier est visible si on liste un répertoire avec `ls -l`. Le type de fichier apparaît avec les droits d'accès avec les codes suivants:

code	type de fichier
`-`	fichier régulier
`d`	répertoire
`s`	socket
`p`	tube nommée (named pipe)
`c`	fichier spécial type caractère
`b`	fichier spécial type block
`l`	lien symbolique

Linux peut gérer plusieurs systèmes de fichiers en même temps: Ext2, Ext3, NTSF, etc. Chaque système réside sur un disque logique (disque physique peut être divisé en plusieurs disques logiques). Il y a un système qui est monté à la racine `/` mais on peut monter d'autres avec la commande `mount`. Dans mon Linux d'autres systèmes de fichiers sont montés dans le répertoire `/media` et constituent les sous-arbres de l'arbre principal. Mais avec la commande `mount` nous pouvons monter les systèmes de fichiers dans n'importe quel répertoire.

Sur mon portable le système NTFS où réside MSWindows est monté sur `/media/OS`, c'est-à-dire `/media/OS` est la racine de ce système de fichiers.

[1] Chaque processus a un répertoire courant. La commande bash `pwd` parmet d'afficher ce répertoire et `cd` permet de changer le répertoire courant. Voir Section 4.2 pour le fonctions POSIX correspondant.

Les droits de la lecture, écriture et exécution. `ls -l` permet d'afficher les droits d'accès à un fichier sous forme `rwxrwxrwx` donnant les droits respectivement pour le propriétaire (user), le groupe propriétaire (group) et les autres (other) (dans cet ordre de gauche à droite).

rwx pour un fichier régulier. La signification de `rwx` pour un fichier régulier est évidente: droit de lecture, écriture et exécution.

- Par exemple pour ouvrir le fichier avec `O_RDONLY` et `O_RDWR` il faut avoir le droit de lecture.
- Pour ouvrir le fichier avec `O_TRUNC` ou `O_RDWR` il faut avoir le droit d'écriture.

rwx pour un répertoire. La signification de `rwx` pour un répertoire est moins évidente. Le répertoire peut être vu comme une table ou une liste composée de couples

`(nom_de_fichier, pointeur)`

où pointeur pointe vers un fichier[2].

Dans le cas d'un répertoire le droit de lecture `r` c'est le droit de lire la liste des entrées de ce répertoire, par exemple il suffit d'avoir le droit de lecture sur le répetoire pour faire `ls` simple (sans options) sur ce répertoire.

Pour le répertoire écrire signifie modifier la liste des entrées dans cette répertoire (modifier c'est-à-dire ajouter une entrée, supprimer une entrée, modifier une entrée exsitante). Donc il faut avoir le droit d'écriture `w` sur le répertoire pour ajouter ou supprimer une entrée dans un répertoire, i.e. créer ou supprimer un fichier dans un répertoire).

Par contre `x` pour un répertoire signifie que nous avons le droit de passage par le répertoire. Donc, par exemple, pour pouvoir lire un fichier[3] qui se trouve dans un répertoire il faut avoir le droit `x` sur ce répertoire (et sur tous les répertoires qui mènent vers ce fichier).

Par exemple pour ouvrir le fichier `/usr/include/stdio.h` il faut avoir les droits de passage `x` sur les répertoires `/`, `/usr` et `/usr/include`.

En conclusion :

- nous pouvons créer un nouveau fichier dans un répertoitre si nous avons les droits `wx` sur ce répertoire, `w` parce que cette opération modifie la liste des entrées du répertoire et `x` pour pouvoir passer dans le répertoire.
- pour supprimer le fichier d'un répertoire il faut aussi les droits `wx` sur le répertoire, `w` parce que l'opération supprime une entrée du répertoire et `x` pour pouvoir passer dans le répertoire. Par contre nous n'avons besoin ni droit de lecture ni d'écriture sur le fichier lui-même pour le supprimer.

3.1 Descripteurs

Un descripteur est un entier non négatif que le système associe avec un fichier ouvert. Trois descripteurs sont définis par des constantes symboliques dans `unistd.h`:

`STDIN_FILENO` `STDOUT_FILENO` `STDERR_FILENO`

pour l'entrée standard, sortie standard et sortie d'erreurs standard. Dans tous les systèmes UNIX les valeurs de ces trois constantes sont toujours respectivement $0, 1, 2$ mais on préfère utiliser les constantes symboliques.

[2] cela ne veut pas dire que le répertoire est vraiment implémenté comme une table ou une liste

[3] et plus en général pour ouvrir le fichier

3.2 Ouverture

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

int open(const *char chemin, int cmd, ... /* mode_t droits */)
int creat(const *char chemin, mode_t droits)
```

Les deux fonctions retournent le descripteur de fichier en cas de succès et -1 en cas d'échec.

Le paramètre `cmd` est une de trois constantes (`fcntl.h`)

- `O_RDONLY` – lecture uniquement
- `O_WRONLY` – écriture seule
- `O_RDWR` – en lecture et écriture

L'information sur le mode d'ouverture figure dans la table de fichiers ouverts et ne pourra pas être modifiée après l'ouverture.

Les constantes qui suivent sont utilisées à l'ouverture de fichier et ne sont pas mémorisées. Elles sont spécifiées en utilisant ou bit à bit `|` avec le mode d'ouverture:

`O_TRUNC` – le fichier ouvert en `O_RDWR` ou `O_WRONLY` sera tronqué à l'ouverture (cela concerne le fichiers réguliers).

`O_CREAT` si la référence n'existe pas alors un i-noeud régulier est créé. Le troisième paramètre donne les droits des utilisateurs. Le masque de `umask` est appliqué. Le fichier créé a comme propriétaire et groupe de propriétaire le propriétaire et groupe de propriétaire effectifs du processus qui crée le fichier.

`O_EXCL` – si `O_CREAT` mais le fichier existe une erreur est envoyée.

`O_NOCTTY` si la référence désigne un terminal ce terminal ne devient pas terminal de contrôle du processus.

Les constantes suivantes sont mémorisées mais peuvent être modifiées avec un appel à `fcntl`.

`O_APPEND` – toute écriture se fera à la fin de fichier.

`O_NONBLOCK` – le processus bloqué à l'ouverture ne le sera pas. `open` renvoi -1 et `errno` est positionné à `EAGAIN`. Pour ouvertures non bloquantes indicateur est sans effet.

`O_SYNC` – écriture en mode bloc (dans le cache système) est bloquée jusqu'à ce que l'écriture sur le disque est réalisé.

Droits d'accès

- `S_IRUSR S_IWUSR S_IXUSR` (respectivement read, write, exec pour le propriétaire), la macro-constante `S_IRWXU` est équivalent à `S_IRUSR|S_IWUSR|S_IXUSR`,
- `S_IRGRP S_IWGRP S_IXGRP` (read, write, exec pour le groupe propriétaire), `S_IRWXG`
- `S_IROTH S_IWOTH S_IXOTH` (read, wrire, exec pour les autres), `S_IRWXO`

`creat(chemin, droits)` est équivalent à
`open(chemin, O_WRONLY | O_CREAT | O_TRUNC, droits)`

Fermeture de fichier

```
#include <unistd.h>
int close(int descriptor)
```

La fonction retourne 0 en cas de succès et -1 en cas d'erreur.

3.3 umask - masque de création de fichier

```
#include <unistd.h>
mode_t umask(mode_t mask)
```

Chaque processus possède un masque qui est appliqué au moment de la création d'un fichier (ou d'un répertoire). La fonction `umask` permet de changer le masque et retourne la valeur précédente du masque.

La commande `umask` de l'UNIX joue exactement le même rôle. La command `umask` sans argument retourne le masque actuel, sur mon portable

```
umask
0022
```

indique que le masque est `0022` en octal, ce qui donne `000010010` en binaire. En comparant ceci avec les droits d'accès `rwxrwxrwx` nous pouvons vois que les bits `1` correspondent aux droits `w` pour le groupe propriétaire et les autres.

Les droits réellement appliqués pendant la création de fichier sont obtenus par

```
demande & ~umask
```

où `demande` les droits demandés.

Par exemple soit

```
umask
0022
```

et je crée dans un programme C un fichier `toto` avec

```
int desc = open("toto", O_WRONLY|O_CREAT,
           S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH);
```

en demandant les droits `rw-rw-rw-` (lecture et écriture pour user,group,other) où `110110110` en binaire. Mais le fichier `toto` aura les droits `rw-r--r--`, le masque empêche d'accorder les droits `w` pour group et other.

Pour que les droits d'accès soient positionnés selon notre demande il faut temporairement modifier le masque :

```
mode_t m;
int desc;
/* changer le masque et mémoriser l'ancien masque*/
m=umask(0000);
desc = open("toto", O_WRONLY|O_CREAT,
           S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH);
/*revenir à l'ancien masque*/
umask(m);
```

Maintenant le nouveau fichier `toto` aura les droits `rw-rw-rw-`

3.4 Lecture de fichiers

```
#include <unistd.h>
ssize_t read(int descriptor, vois *tampon, size_t nombre)
```

Le primitif envoie le nombre de caractères lus si la lecture réussit, 0 si c'est la fin de fichier, -1 en cas d'erreur.

Les types `size_t` et `ssize_t` définis dans `<sys/types.h>`, `size_t` c'est un type entier non signé, `ssize_t` est un type entier signé.

S'il y a un verrou exclusif impératif sur le fichier dans la portée de la lecture alors `open` bloque en mode bloquant ou on retourne -1 et `errno==EAGAIN` en mode non bloquant (voir le section sur les verrous).

En POSIX.1.2001

```
#include <sys/uio.h>
ssize_t readv(int descriptor, const struct iovec *vecteur, int iovcnt)
```

```
#include <sys/uio.h>
struct iovec {
  char *iov_base;    /*addresse en memoire */
  size_t iov_len;    /*nombre de caractères à lire */
}
```

La fonction `readv` lit dans les tampons spécifiés dans `vecteur`.
`vecteur[i].iov_base` donne l'adresse de $i$ème tampon, `vecteur[i].iov_len` donne sa longueur.

```
#include <unistd.h>
ssize_t pread(int descripteur, void *tampon, size_t nombre, off_t offset)
```

Cette fonction lit à partir de la position indiquée, mais cela n'affecte pas offset du fichier, c'est-à-dire la position courante ne changera pas.

Le type `off_t` est un type signé entier utlisé pour la taille de fichier.

3.5 Écriture dans un fichier

```
#include <unistd.h>
ssize_t write(int descripteur, void *tampon, size_t nombre)
```

`write` retourne le nombre d'octets écrits dans le fichier, -1 en cas d'erreur.

S'il n'y a pas de verrou (exclusif, impératif, partagé) alors : l'écriture soit à la position courante soit à la fin si `O_APPEND`. Si le nombre renvoyé ¡ nombre alors une erreur. La position courante est augmentée.

S'il y a un verrou est écriture bloquante il bloque. Si verrou et non bloquant alors retourne -1 et `errno==EAGAIN`

```
#include <sys/uio.h>
ssize_t writev(int descripteur, const struct iovec *vecteur, int n)
```

`vecteur` c'est un vecteur de structures `iovec`, chaque structure décrit un tampon, `int n` donne le nombre d'éléments dans le vecteur.

```
#include <unistd.h>
ssize_t pwrite(int descripteur, void *tampon, size_t n, off_t position)
```

3.6 Copier un fichier

Le programme suivant copie un fichier en utilisant le descripteurs de fichiers. La taille de tampon est passée comme le paramètre de `main` ou, à défaut, elle est égale à 1024.

Si le tampon est de taille d'un octet alors sur mon portable le temps d'exécution affiché avec `time` est

```
real 0m6.388s
user 0m0.768s
sys 0m5.592s
```

pour un fichier de taille 2594272 octets. Avec le tampon de 1024 octets le temps est

```
real 0m0.026s
user 0m0.000s
sys 0m0.024s
```

pour le même fichier.

```
#define _POSIX_C_SOURCE 200112L
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include "panic.h"

#define TAILLE 1024
int main(int argc, char *argv[]){

  int fd1, fd2, rc, wc;
  char *tamp;
  int t;

  if( argc == 4 ){
    t=atoi(argv[3]);
  }
  else if( argc == 3 )
    t = TAILLE;
  else{
    fprintf(stderr,"usage:\n %s fichier_in fichier_out [taille_de_tampon]\n",
            argv[0]);
```

```
    exit(1);
  }
  if( ( tamp = malloc(t) ) == NULL)
    PANIC(1);

  if ( ( fd1 = open(argv[1],O_RDONLY) ) < 0 )
    PANIC(2);

  if( ( fd2 = open(argv[2],O_WRONLY | O_CREAT | O_TRUNC,
                   S_IRWXU | S_IRWXG | S_IRWXO ) ) < 0 )
    PANIC(3);


  for( ;; ){
    rc = read(fd1, tamp, t);
    if( rc < 0 )
      PANIC(4);
    if( rc == 0 )
      break;

    wc=write(fd2, tamp, rc);
    if( wc < 0 )
      PANIC(5);
  }
  close(fd1);
  close(fd2);
  free(tamp);
  return 0;
}
```

3.7 access

Pour déterminer si un processus possède un accès à un fichier on utilise la fonction

```
#include <unistd.h>
int access(const char *chemin, int mode)
```

La fonction retourne 0 si le test d'accès est positif et -1 sinon. Le paramètre mode peut prendre les valeurs suivantes :

- `F_OK` pour tester l'existence,
- `R_OK`, `W_OK`, `X_OK` pour tester le droits de lecture, écriture, exécution.

3.8 Manipulation d'offset

```
#include <unistd.h>
off_t lseek(int descripteur, off_t position, int origine)
```

`origine` une des trois constantes :

`SEEK_SET`	par rapport au début de fichier
`SEEK_CUR`	par rapport à la position courante
`SEEK_END`	par rapport à la fin de fichier

En cas d'erreur la valeur `(off_t) -1` est envoyée, sinon la fonction envoie la position courante après le deplacement.

Exemple 1. `off_t pos=lseek(desc, (off_t) -20, SEEK_CUR);`

déplace la position courante de 20 octet vers le début de fichier.

Par contre `off_t pos=lseek(desc, (off_t) 20, SEEK_SET);` place la position courante 20 octets après le début de fichier.

Il est impossible de se placer à une position < 0, par contre il est bien possible de passer à une position au-delà de la taille de fichier. Si on écrit dans le fichier est la position courante est supérieures à la taille de fichier alors le "trou" sera rempli par les caractères nul `'\0'`.

3.9 Quelques remarques sur les types

D'après Single UNIX Specification:

- `size_t` – utilisé pour la taille d'objet, type entier non signé.
- `ssize_t` — utilisé pour compter les octets ou pour indiquer erreur, type signé entier.

C'est difficile de voir comment `size_t` et d'autres de ces types sont définis en regardant les fichiers en-tête, trop de conditions à suivre. Une solution c'est voir comment les macros se développent:

```
cpp -dD -std=c99 copy.c > toto.txt
```

et dans `toto.txt` j'ai trouvé les lignes

```
typedef unsigned int size_t;

typedef int __ssize_t;
typedef __ssize_t ssize_t;

typedef long int __off_t;
typedef __off_t off_t;
```

4 Les répertoires

4.1 Suppression, création, parcours d'un répertoire

La création:

```
#include <sys/stat.h>
int mkdir(const char *pathname, mode_t mode);
```

Retourne 0 si OK et -1 sinon. `mode_t` permet de spécifier les droits d'accès au répetoire, on utilise les mêmes constantes que pour les fichiers.

La suppression:

```
#include <unistd.h>
int rmdir(const char *pathname);
```

Retourne 0 si OK et -1 sinon. Le répertoire doit être vide.

La lecture d'un répertoire:

```
#include <dirent.h>
DIR *opendir(const char *pathname);
struct dirent *readdir(DIR *dp);
void rewinddir(DIR *dp);
int closedir(DIR *dp);
```

`opendir` retourne un pointer si OK, NULL si error.
`readdir` retourne un pointer si OK, NULL si la fin du répertoire ou une erreur.

Les fonctions suivantes ne sont POSIX mais une extension XSI.

```
long telldir(DIR *dp);
void seekdir(DIR *dp, long loc);
```

`telldir` retourne la position courante dans le répertoire associé à `dp`.

```
struc dirent {
        ino_t d_ino;                    /* i-node number */
        char  d_name[NAME_MAX + 1];     /* null-terminated filename */
}
```

dépend de l'implementation. `d_name` existe toujours mais pas `d_ino`.

La valeur de `NAME_MAX` n'est pas importante mais elle peut-être trouvée avec `fpathconf`.

```
#define _POSIX_C_SOURCE 1
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "panic.h"


static int parcours(const char *nom_rep);
```

```
int  main(int argc, char *argv[]){
  int i;
  for(i=1;i < argc;i++){
    parcours(argv[i]);
  }
  return EXIT_SUCCESS;
}

static int parcours(const char *nom_rep){
  DIR *flot;

  if((flot = opendir(nom_rep))==NULL){
    PANIC(1);
  }
  for (;;){
    struct dirent *entree;
    errno = 0;
    entree = readdir(flot);
    if (entree == NULL){
      if (errno){ /* erreur de parcours*/
        PANIC(2);
      }
      closedir(flot);
      return 0;
    }
    if (strcmp(entree->d_name, ".") == 0
        || strcmp(entree->d_name, "..") == 0)
      continue;
    printf("%s/%s\n", nom_rep, entree->d_name);
  }
}
```

4.2 Répertoire courant

Chaque processus possède le répertoire courant. Ce répertoire est utilisé pour évaluer les chemins relatifs comme par exemple `../../toto`. Le processus hérite son répertoire courant de son parent. Les fonctions

```
#include <unistd.h>
char *getcwd(char *tampon, size_t taille)
int chdir(const char *chamin)
```

permettent respectivement de récupérer le répertoire courant et changer le répertoire courant. `getcwd` retourne `NULL` (et `errno=ERANGE`) si taille du tampon n'est pas suffisante pour stocker le chemin d'accès vers le répertoire courant (dans ce cas il convient d'augmenter la taille de `tampon`).

Exemple 2. `chdir(getenv(``HOME''))` place le processus dans le répertoire principal de l'utilisateur.

5 La structure de système de fichier

5.1 i-noueds

Chaque fichier possède un i-noeud (i-node) qui contient plusieurs informations sur le fichier:

- le type de fichier (fichier regulier, répertoire, lien symbolique, tube, fichier spécial, etc.),
- le nombre de liens physiques (liens durs) vers le fichier,
- la taille en octets (si cela a un sens pour le type donné de fichiers),
- device ID – identifie le volume (c'est-à-dire le disque logique) où réside le fichier,
- le numéro i-node – c'est un numéro que le système attribue au fichier. Pour le même disque logique les numéros i-node sont tous différents, il n'y a jamais deux fichiers avec le même numéro de i-node pour le même disque logique. Le numéro de i-node peut être vu comme l'identifiant de fichier pour un disque logique donné. Ceci implique que chaque fichier est identifié par le couple (device ID, le numéro d'i-node).
- ID de propriétaire et ID du groupe propriétaire,
- trois dates :
  - le date du dernier accès au fichier,
  - la date de la dernière modification de données,
  - la date de la dernière modification d'attributs, c'est-à-dire la dernière modification du noeud lui-même (création/suppression d'un lien, changement de droits etc.)
- les drapeaux de droits lecture/écriture/exécution pour propriétaire, groupe propriétaire et les autres,
- les drapeaux setuid, setgid et sticky bit.

Toutes les dates depuis EPOCH (le 1 janvier 1970).

Notez que le nom de fichier n'est pas stocké dans i-node, la seule information permettant d'identifier le fichier c'est le numéro de i-node.

Exemple 3. Supposons que l'arborescence de fichiers contient un répertoire nommé `Document` et le dessin 3 montre le fragment de l'arborescence avec la racine dans ce répertoire. Les dessin 3 montre la même arborescence plus en détail, avec le contenu des répertoires et le compteur de liens physiques dans chaque i-node.

Notez que pour un répertoire vide le compteur de liens physiques a valeur 2: il y a le lien depuis son père et un autre lien depuis le répertoire lui même (l'entrée "point" dans le répertoire). Pour le seul fichier régulier `loyer.pdf` le compteur de liens est 1, seul le répertoire `Autres` qui le contient pointe vers ce fichier.

Les liens physiques sont créés avec la création d'un fichier ou d'un répertoire.

On peut aussi créer un nouveau lien physique vers un fichier déjà existant avec la commande UNIX `ln`

```
ln chemin nom_de_lien
```

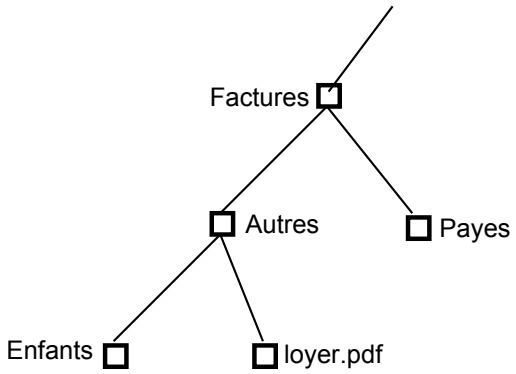

Figure 1: Le seul fichier régulier c'est le fichier `loyer.pdf`, `Enfants`, `Payes` sont des répertoires vides.

où `chemin` est un chemin (absolu ou relatif) vers un fichier régulier[4] et `nom_de_lien` donne le nouveau nom de lien.

[4]Seul le superuser peut créer de liens physiques vers les répertoires.

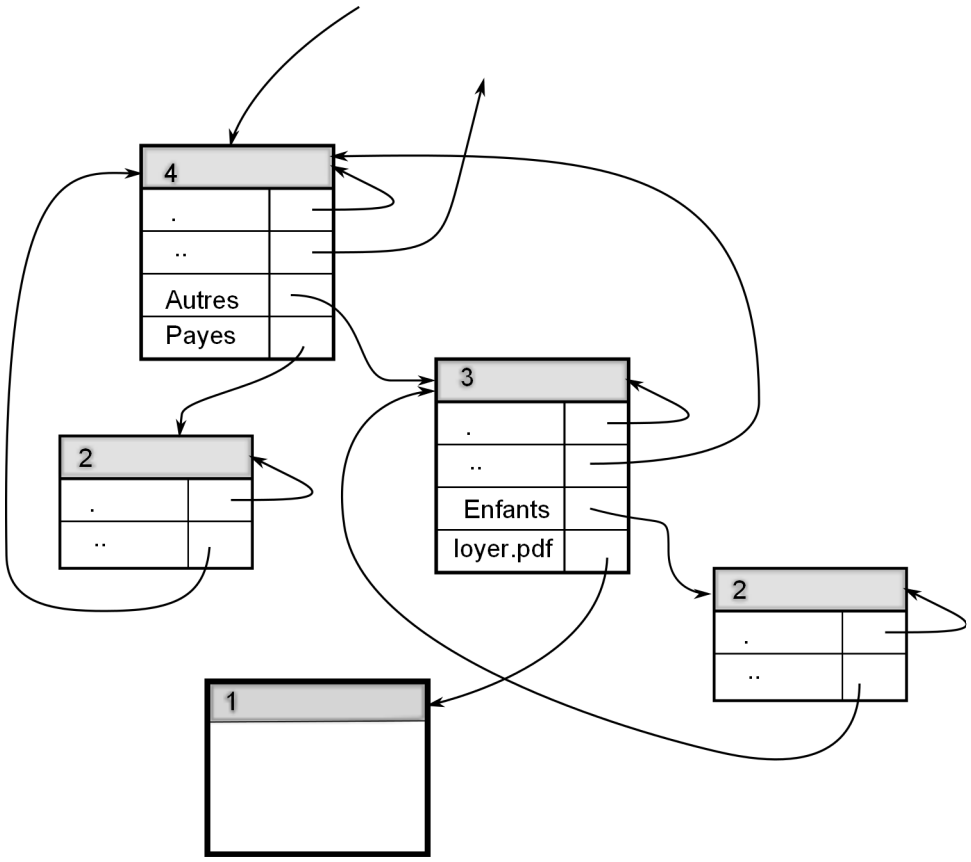

Figure 2: Les fichiers et les répertoires sont représentés par des rectangles. La partie grise de chaque rectangle représente le i-node correspondant, le nombre à l'intérieur c'est le nombre de liens physiques vers le fichier ou répertoire. Nous pouvons voir que, par exemple, le nom de répertoire `Enfants`, n'est pas du tout stocké dans ce répertoire mais dans le répertoire père.

Exemple 4. Supposons par exemple que pour la configuration de dessin 3 notre répertoire courant est `Payes`. Dans ce cas la commande

```
ln ../Autres/loyer.pdf toto.pdf
```

exécutée depuis le répertoire `Payes` ajoute un nouveau lien physique dans le répertoire `Payes` vers le fichier `loyer.pdf`. Ce lien (la nouvelle entrée dans le répertoire `Payes`) porte le nom `toto.pdf`.

Le résultat de l'exécution de cette commande est presenté sur le dessin 4.

Le même résultat on peut obtenir en exécutant

```
link("../Autre/loyer.pdf", "toto.pdf");
```

dans un programme C (à condition que `Payer` soit le répertoire courant de processus qui exécute `link`., voir Section 5.2.

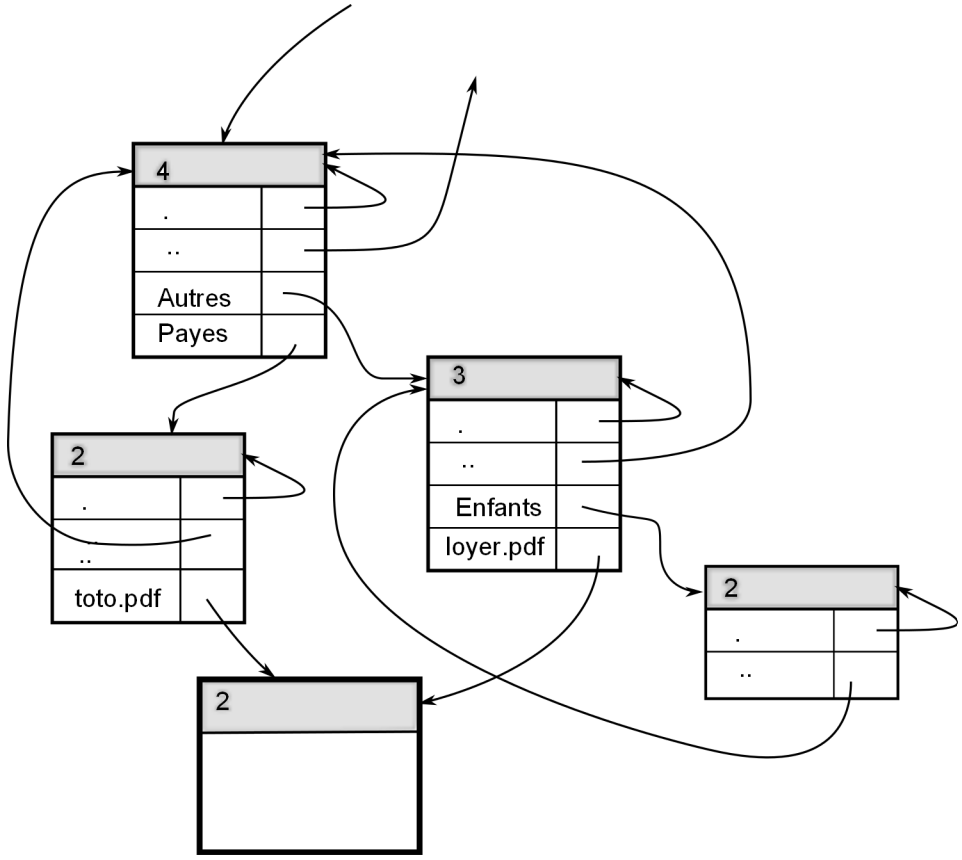

Figure 3: La commande `ln ../Autre/loyer.pdf toto.pdf` crée un nouveau lien physique vers le fichier `loyer.pdf`. Noter qu'il n'y a pas de crétion d'un nouveau fichier, juste une nouvelle entrée dans le répertoire `Payes`. Le compteur de lien physiques du fichier passe de 1 à 2.

5.2 Création/suppression/changement de nom de lien physique

```
#include <unistd.h>
int link(const char *origine, const char *cible)
int unlink(const char *reference)
int rename(const char *ancien, const char *nouveau)
```

link `link` crée un nouveau lien physique vers un fichier ordinaire. `origine` est la référence vers un fichier existant, `cible` est la nouvelle référence qui sera créée par `link`, voir l'exemple de la section précédente.

Cible ne doit pas exister. Origine ne peut être un répertoire (sauf si c'est le super-utilisateur qui exécute link). Origine et cible dans le même système de fichiers.

`link` crée un nouveau lien physique, il n'y a pas de création de i-noeud, voir la section précédente.

ulink `unlink` supprime le lien physique, c'est-à-dire `ulink` supprime l'entrée dans le répertoire. Le fichier correspondant est supprimé aussi seulement si deux conditions sont réunies:

(1) le nombre de liens physiques vers le fichier devient nul,

(2) le nombre d'ouvertures du fichier est nul (pas de descripteur ouvert sur le fichier).

rename `rename` change le nom de lien physique. On ne peut pas renommer ni `.` (dot) ni `..` (dot dot) . Si `nouveau` existe déjà il sera supprimé avant l'opération.

```
Pour détruire un fichier (ou plus exactement pour supprimer le lien
dur vers le fichier avec unlink)  il n'est pas nécessaire d'en
être propriétaire ni d'avoir une quelconque permission sur ce
fichier.
Par contre il est nécessaire d'avoir la permission d'écrire
dans le répertoire dont on veut supprimer la référence au fichier.
La même remarque s'applique au renommage.
```

5.3 Les fichiers ouverts

Chaque processus a une entrée dans la table de processus. Chaque entrée de la table de processus contient la table de descripteurs ouverts. Pour chaque descripteur il y a le drapeaux close-on-exec[5] et le pointeur vers une entrée de la table de fichiers ouverts.

Le noyau maintient une table de fichiers ouverts. Chaque entrée de cette table contient

- différents drapeaux (read, write, append, sync, nonblocking,
- un pointeur vers une entrée de la table de v-noeuds.

Chaque fichier ouvert a un v-noeud qui contient des informations sur le type de fichier et pointeurs vers les fonctions qui opèrent sur ce fichier, voir la figure 5.3.

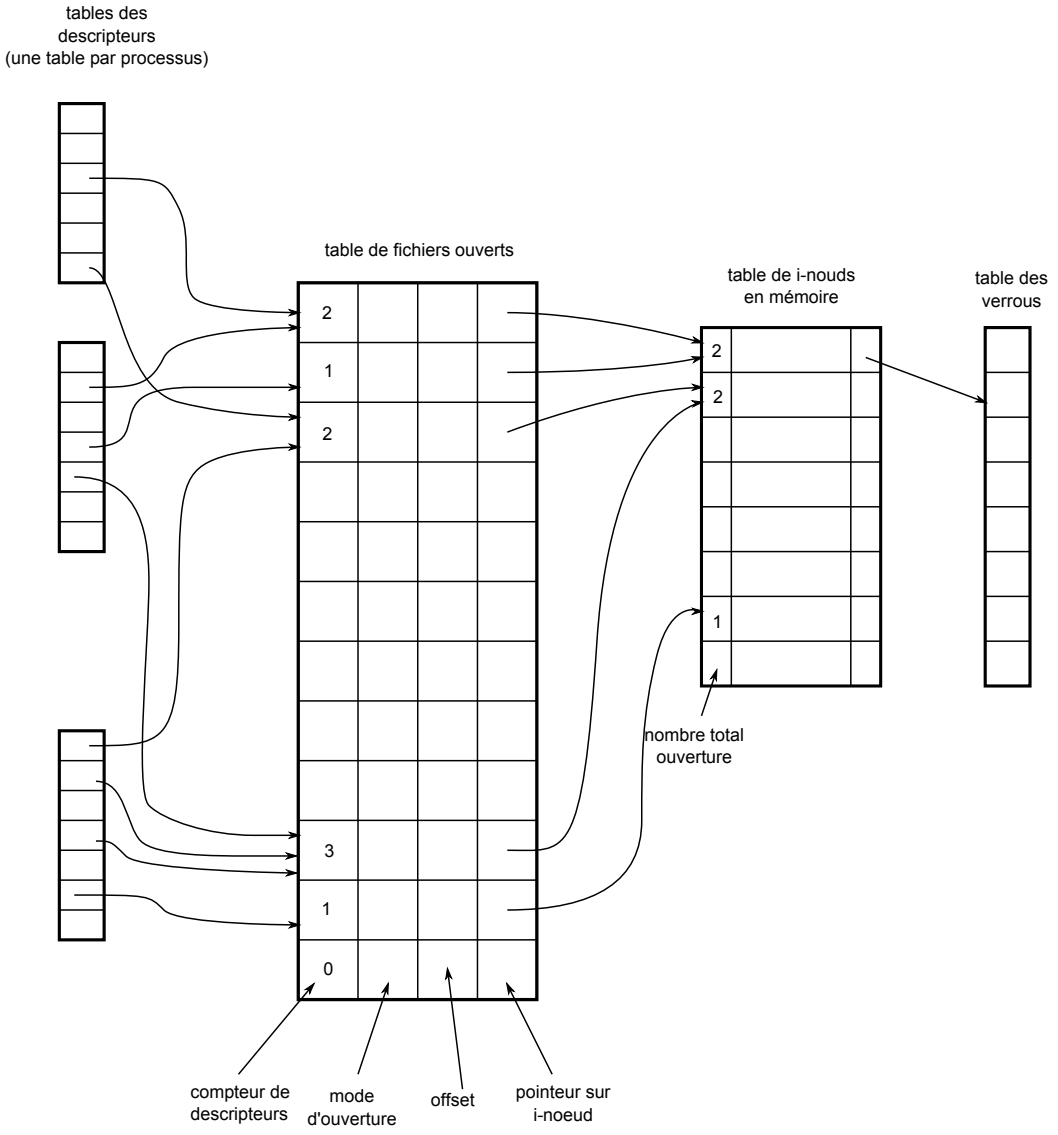

Figure 4: Chaque processus possède sa propre table de descripteurs. Par contre la table de fichiers ouverts et la table de i-noeuds sont unique dans tout le système.

[5]Ce drapeaux sera expliquer quand nous abordons les processus.

5.4 Dernières remarques sur les liens physiques

Les premiers systèmes UNIX (SVR3 et 4.1BSD) implémentaient seulement les liens physiques. Les problèmes: les liens physiques ne peuvent pas traverser d'un système de fichiers à autre. Faire les liens durs vers des répertoires peut former de cycles dans le système de fichiers, et certains fonctions comme `find` et `du` sont récursives et les cycles provoquent des problèmes pour les fonctions de ces fonctions.

Pour cette raison seulement super-utilisateur peut faire les liens durs vers des répertoires.

Les liens durs provoquent des problèmes de contrôles. Supposons que l'utilisateur X possède un fichier `/usr/X/file1` et l'utilisateur Y fait un lien dur `/usr/Y/link1` vers ce ce fichier. Pour cela Y a besoin seulement les permissions d'exécution (de passage) sur les répertoires qui mènent vers le fichier. Maintenant l'utilisateur X peut supprimer (unlink) `/usr/X/file1` et il croit que le fichier est effectivement supprimé (d'habitude on ne regarde pas le compteur de lien sur nos propre fichier).

Bien sûr X est le propriétaire de `/usr/Y/link1` mais il ne sait rien que le lien existe et si X protège le lecture du répertoire `/usr/Y` X n'a aucun moyen de trouver ce lien.

6 Structure stat et les fonctions stat fstat lstat – consultation des attributs stockés dans un i-node

6.1 struct stat

Les informations stockées dans un i-node sont disponible dans la structure `stat` :

```
#include <sys/types.h>
#include <sys/stat.h>
struct stat{
    dev_t st_dev;    /*identificateur de systeme de fichiers (volume *
                       * logique) contenant le fichier*/
    ino_t st_ino;     /*numero de fichier sur le disque   *
                       * l'identifient de fichier */
    mode_t st_mode ;  /*type de fichier et droits d'utilisateur*/
    nlink_t st_nlink; /*nombre de liens physiques*/
    uid_t st_uid;     /*proprietaire*/
    gid_t st_gid;     /*groupe  proprietaire*/
    off_t st_size:    /*taille*/
    time_t st_atime;   /*date de dernier acces (temps depuis 1.01.1970)*/
    time_t st_mtime;  /*date de derniere modif des donnees*/
    time_t st_ctime;  /* derniere modific de caracteristiques*/
}
```

Pour obtenir ces information on utilise les fonctions suivantes:

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

```
int stat(const char *reference,struct stat *p_stat)
int fstat(int descripteur, struct stat *p_stat)
int lstat(const char *reference, struct stat *p_stat)
```

Les fonctions `stat` et `lstat` prennent comme paramètre le chemin vers un fichier, la fonction `fstat` prend un descripteur de fichier ouvert. Le deuxième paramètre de chaque fonction c'est l'adresse de la structure `struct stat`, la structure sera mise à jour par l'appel.

Remarque sur les liens symboliques. La différence entre `stat` et `lstat` réside dans le traitement de liens symboliques. Pour `stat` les liens symboliques sont transparents, c'est-à-dire `stat` appliquée à un lien symbolique donne les attributs de fichier pointé par le lien et non les attribut de lien symbolique lui-même. Par contre `lstat` appliquée sur le lien symbolique donne les attributs de ce lien, voir section 7.

Exemple 5. Pour recuperer les informations sur le fichier `/home/dupont/toto` on peut faire:

```
struct stat bufstat;
stat("/home/dupont/toto", &bufstat);
```

et maintenant on peut afficher l'identifient du propriétaire avec de ce fichier:

```
printf("id proprio=%d\n", bufstat.st_uid);
```

Une fois la structure `struct stat` initialisée comme indiqué dans l'example ci-dessus il est possible de vérifier le type de fichier. On utilise les macro-fonctions suivantes qu'on applique au champ `st_mode` de la structure `struct stat`:

macro-fonction	type de fichier	lettre (type) affichée par `ls -l`
`S_ISREG(bufstat.st_mode)`	fichier régulier	`-`
`S_ISFIFO(bufstat.st_mode)`	fichier spéciale FIFO (tube)	`p`
`S_ISCHR(bufstat.st_mode)`	type spécial caractère	`c`
`S_ISBLK(bufstat.st_mode)`	type special bloc	`b`
`S_ISDIR(bufstat.st_mode)`	type spécial répertoire	`d`
`S_ISLNK(bufstat.st_mode)`	lien symbolique	`l`
`S_ISSOCK(bufstat.st_mode)`	socket	`s`

Les droits d'accès pour le répertoire:

- lecture — permission de consulter la liste des entrées de répertoire
- écriture — autorisation de modifier la liste des entrées de répertoire (supprimer un élément de répertoire, en ajouter un autre)
- exécution — permission de traverser un répertoire.

Les droits sont vérifiés vis-à-vis du propriétaire *effectif* du processus qui exécute la commande.

Les drapeaux de permission de lecture/écriture/exécution sont les mêmes que pour `open` et `creat`:

```
S_IRUSR S_IWUSR S_IXUSR    S_IRWXU
S_IRGRP S_IWGRP S_IXGRP    S_IRWXG
S_IROTH S_IWOTH S_IXOTH    S_IRWXO
```

Donc par exemple `bufstat.st_mode & S_IRUSR` est vrai si le propriétaire de fichier possède le droit de lecture.

D'autres drapeaux:
`S_ISUID` set-user-ID on execution
`S_ISGID` set-group-ID on execution
`S_ISVTX` on directories, restricted deletion flag .

6.2 Les bits : set-uid, set-gid — propriétaire réel et propriétaire effectif d'un fichier

Le champ `st_mode` de `struct stat` contient aussi trois drapeaux (bits): *set-uid set-gid* et *sticky*.

Pour comprendre à quoi servent ces trois bits il faut comprendre comment les systèmes d'exploitation determine si un processus peut accéder à un fichier.

Chaque processus possède deux propriétaires:

(1) propriétaire réel – c'est celui qui a créé le processus,
(2) propriétaire effectif – c'est celui que le système utilise pour déterminer si un processus possède les droits d'accès à un fichier.

Dans la plupart de cas le propriétaire effectif et le propriétaire réel sont les mêmes.

Supposons qu'un utilisateur `sophie` lance depuis le terminal la commande `cat` qui permet de lister le contenu d'un fichier. L'exécutable ls se trouve dans le répertoire `/bin` et `ls -l /bin/cat` affiche

```
-rwxr-xr-x 1 root root 46764 Oct  2 05:25 /bin/cat
```

Donc nous pouvons voir que le propriétaire de fichier exécutable `/bin/ls` est `root`. Supposant que `sophie` exécute la commande `cat toto.txt`. Dans ce cas le propriétaire réel et le propriétaire effectif du processus lancé par `sophie` et exécutant `cat` sera l'utilisateur `sophie` elle même, en particulier le fait que `root` est le propriétaire de fichier exécutable `/bin/cat` n'est pas pris en compte. Ce sont les droits de `sophie` qui déterminent si `sophie` peut ou ne peut pas lister le fichier `toto.txt`. C'est d'ailleurs tout à fait logique parce que `root` peut lister n'importe quel fichier et certainement nous ne voulons pas donner ce privilège à `sophie` même si nous permettons à `sophie` d'exécuter la commande `/bin/cat`.

Maintenant regardons la commande `passwd` qui permet de changer le mot de passe d'un utilisateur. `ls -l /usr/bin/passwd` affiche

```
-rwsr-xr-x 1 root root 37140 2011-02-14 23:11 /usr/bin/passwd
```

Nous constatons que `root` est le propriétaire de ce fichier exécutable mais à la place de `x` qui indique le droit de l'exécution pour `user` (propriétaire de fichier) nous avons

la lettre `s`. La lettre `s` indique que set-uid bit a été positionné pour ce fichier. Donc quand `toto` lance la commande `passwd` il sera le propriétaire réel de processus exécutant la commande mais le propriétaire effectif de ce processus sera `root` c'est-à-dire le propriétaire de fichier exécutable `/usr/bin/passwd`. Cela implique que ce sont les droits de `root` qui déterminent à quels fichiers peut accéder le processus exécutant `passwd`.

En effet `passwd` accède aux fichiers protégés qui contiennent les mots de passe donc il faut avoir les droits de `root` pour lire et modifier ces fichiers.

Le bit set-gid joue le même rôle que set-uid mais pour le groupe propriétaire.

set-uid et se-gid s'appliquent sur le fichiers exécutables.

La constante `S_ISUID` et la constante `S_ISGID` permettent de tester si set-uid, set-gid sont positionnés.

Donc, par exemple, `bufstat.st_mode & S_ISUID` est vrai ($\neq 0$) si le bits set-uid est positionné et `buffstat` est la structure `struct stat` qui contient les informations sur le i-node.

Depuis la console on peut voir si les bits set-uid ou set-gid sont positionné en regardant l'affichage produit par la commande `ls`.

Si on exécute `ls -l nomfichier` alors l'affichage `rwsr--r--` (c'est-à-dire `s` à la place de `x`) indique que le droit d'exécution est donné pour le propriétaire et set-uid bit est positionné. Par contre l'affichage `rwSr--r--` (c'est-à-dire `S` majuscule à la place de `x`) indique que set-uid bit est positionné mais le fichier n'est pas exécutable pour le propriétaire.

6.3 Sticky bit

Sticky bit est utilisé pour resteindre les droits de suppression des éléments d'un répertoire.

Le sticky bit est testé avec la constante `S_ISVTX`:

```
bufstat.st_mode & S_ISVTX
```

Si le sticky bit est positionné sur un répertoire alors le fichier dans le répertoire peut être supprimé ou renommé si l'utilisateur a la permission `write` sur le répertoire et une des conditions suivantes est satisfaite :

1. l'utilisateur[6] est le propriétaire du fichier,
2. l'utilisateur est le propriétaire du répertoire,
3. l'utilisateur est super-utilisateur.

Le répertoire `/tmp` est le candidat typique pour avoir le sticky bit positionné. Les permissions pour ce répertoire sont : read, write, execute pour tous (user, group, other). Mais l'utilisateur doit être capable de supprimer ou renommer seulement ses propres fichiers dans ce répertoire, nous ne voulons pas qu'il soit capable de supprimer ou renommer les fichiers présents dans `/tmp` mais qui ne lui appartiennent pas. `ls -l` pour `/tmp` affiche

[6] Plus precisement le propriétaire de processus qui essaie accéder au répertoire.

```
drwxrwxrwt  14 root root  4096 2011-11-27 20:26 tmp
```

Le `t` à la fin de droits d'accès (à la place de `x`) indique que le sticky bit est positionné pour le répertoire `tmp` et que le droit de passage `x` est accordé pour `other`.

Si à la place de `t` on trouve `T` cela signifie que le sticky bit est positionné mais `other` n'a pas de droit de passage sur ce répertoire.

6.4 Changement d'attributs d'un fichier : droits d'accès, propriétaire, dates d'accès

Les fonctions

```
#include <sys/stat.h>
int chmod(const char *reference, mode_t mode)
int fchmod(int descripteur, mode_t mode)
```

permettent de changer les droits d'accès, les bits set-uid, set-gid et sticky. Dans le cas de `chmod()` on suit les liens symboliques.

Pour changer le propriétaire de fichier le super-utilisateur root utilise

```
#include <sys/stat.h>
int chown(const char *reference, uid_t uid, gid_t gid)
int fchown(int descripteur, uid_t uid, gid_t gid)
```

Pour changer manuellement les dates:

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *reference, const struct utimbuf *temps)

struct utimbuf{
      time_t actime; /*pour st_atime*/
      time_t modtime; /*pour st_mtime*/
}
```

Temps mesuré depuis 1 janvier 1970.

6.5 Nom d'utilisateur, le répertoire initial, le shell

```
#include <pwd.h>
struct passwd *getpwuid(uid_t uid)
struct passwd *getpwnam(const char *nom)
```

La structure `struct passwd` contient les champs suivants:

```
char    *pw_name   User's login name.
uid_t    pw_uid    Numerical user ID.
gid_t    pw_gid    Numerical group ID.
char    *pw_dir    Initial working directory.
char    *pw_shell  Program to use as shell.
```

7 Liens symboliques

4.2BSD introduit les liens symboliques. C'est un fichier spéciale qui pointe vers un autre fichier. Le type du lien symbolique l'identifie en tant qu'un fichier spécial de type lien symbolique et les données contiennent le chemin vers le fichier pointé par le lien. Le chemin peut être sauvegardé dans le lien soit comme le chemin absolu soit comme le chemin relatif. Pour la plupart de programme le lien symbolique est transparent (en prenant le lien le programme arrive au fichier pointé par le lien). `lstat` appliqué au lien retourne les attributs du lien, `stat` appliqué au lien retourne les attribut du fichier pointé. Liens symboliques intégrés en POSIX.1:2001. Dans la commande `ls` les liens symboliques apparaissent avec le type `l`.

7.1 Création du lien symbolique

Création sous shell avec la commande

```
ln -s target link_name
```

qui crée un lien dont le nom est `link_name` et dont le contenu est `target`. Cette commande crée un nouveau i-noeud et un nouveau fichier de type lien symbolique (rappelons que ce n'est pas le cas pour les liens physiques où juste une nouvelle entrée de répertoire est créée).

Exemple 6. `ln -s /usr/bin monbin`

crée un lien symbolique `monbin` dans le répertoire courant. Le contenu de ce lien est `/usr/bin`.

```
ls -l | grep toto
```

affiche

```
lrwxrwxrwx  1 Wiesiek         None      8 Nov 12 16:37 monbin -> /usr/bin
```

On note que `monbin` est un nouveau fichier de type `l` (lien symbolique) et dont le contenu est affiché après `->`. Création d'un lien symbolique donne toujours lieu à la création d'un i-node correspondant à ce lien.

Une fois le lien symbolique construit si nous tapons sur le terminal

```
cd monbin
```

alors `/usr/bin` devient le répertoire courant. La commande `cd` suit le lien symbolique et interprète le contenu de liens comme le chemin.

Exemple 7. `ln -s ../bin autrebin`

crée un lien symbolique `autrebin` dans le répertoire courant. Le contenu de ce lien est `../bin`.

```
ls -l | grep toto
```

affiche

```
lrwxrwxrwx  1 Wiesiek         None      8 Nov 12 16:37 autrebin -> ../bin
```

Une fois le lien symbolique construit si nous tapons sur le terminal

```
cd autrebin
```

alors `../bin` devient le répertoire courant si la référence relative `../bin` est correcte, sinon nous aurons le message indiquant que `../bin` n'existe pas (no such file or directory).

Exemple 8. `ln -s ';,:titi t+'  ../toto`

crée un lien symbolique `toto` dans le répertoire père du répertoire courant. Le contenue de ce lien c'est une chaîne de caractères `;,:titi t+` Notons au moment de la création de lien il n'y a pas de vérification si le contenu du lien symbolique représente un chemin valable ou non, donc nous pouvons créer un lien symbolique dont le contenu est une chaîne de caractères quelconque. Bien sûr un tel lien n'est pas très utile.

Depuis un programme C on crée un lien symbolique avec

```
#include <unistd.h>
int symlink(const char *reference, const char *lien)
```

qui crée un lien symbolique dont le contenu est `reference`. La fonction retourne 0 si OK et -1 sinon.

Exemple 9. L'appel

```
symlink("toto/momo", "../exo")
```

- crée un lien symbolique `exo` dans le répertoire père du répertoire courant. Le contenu du lien est la chaîne de caractères `toto/momo`. Comme pour la commande `ln -s` Il n'y a aucune vérification si cette chaîne correspond à un fichier, à cette étape c'est juste une chaîne de caractères stockée dans le lien,
- une nouvelle entrée nommée `exo` est ajoutée dans le répertoire père du répertoire courant (dans le répertoire `..`). Le i-noeud associé à cette entrée c'est le i-noeud décrit ci-dessus.

Rappelons que le lien physique peut être créé uniquement vers un fichier, par contre nous pouvons créer un lien symbolique vers le répertoire ou vers un fichier.

Il est impossible de créer un lien physique qui réside dans un autre système de fichier, par contre nous pouvons créer un lien symbolique vers un fichier ou un répertoire résidant dans un autre disque logique.

7.2 Consultation des attributs d'un lien symbolique

Rappelons que la fonction `stat` suit le lien symbolique et récupère les attributs de la référence.

Pour récupérer les attributs d'un lien symbolique on utilise la fonction

```
#include <sys/types.h>
#include <sysy/stat.h>
int lstat(const char *reference, struct stat *pstat)
```

Pour tous les fichiers qui ne sont pas des liens symboliques `lstat()` donne le même résultat que `stat()`. Pour un lien symbolique le champ `st_size` de la structure `struct stat` donne la longueur du contenu du lien (sans caractères nul à la fin).

Exemple 10. Par exemple pour le lien symbolique crée dans l'exemple 9

```
struct stat s;
lstat("../exo",  &s);
```

`s.st_size` donne le nombre de caractères dans la chaîne `"toto/momo"`

En général certaines fonctions suivent les liens symboliques tandis que d'autres non.

Ces différents comportement sont répertoriés dans le tableau suivant:

fonction	ne suit pas le lien symbolique	suit le lien symbolique
access		x
chdir		x
chmod		x
chown		x
creat		x
exec		x
lchown	x	
link		x
lstat	x	
open		x
opendir		x
pathconf		x
readlink	x	
remove	x	
rename	x	
stat		x
truncate		x
unlink	x	

7.3 La lecture du lien symbolique

```
#include <unistd.h>
ssize_h readlink(const char *lien, char *tampon, size_t taille)
```

permet de récupérer le contenu d'un lien symbolique (sa valeur) qui est copié dans `tampon`. La suite de caractères copiés dans `tampon` *n'est pas suivie par le caractère nul*. Si la taille de tampon `taille` n'est pas suffisante alors le contenu du lien est tronqué. La fonction renvoie le nombre de caractères lus.

Exemple 11. Pour le lien crée dans l'exemple 9

```
int i;
char tampon[100];
i=readlink("../exo",&tampon,99);
```

va copier dans `tampon` la chaîne `"toto/momo"` (sans caractère nul à la fin). Pour obtenir une vraie chaîne de caractères avec nul à la fin il faut ajouter l'instruction

```
tampon[i]='\0';
```

Par contre

```
int i;
char tampon[3];
i=readlink("../exo",&tampon,3);
```

va copier dans `tampon` les trois premiers caractères de la chaîne `"toto/momo$"` (encore une fois sans caractère nul à la fin).

La question se pose comment savoir qu'elle est la taille de tampon qu'il faut préparer pour lire le contenu de lien symbolique. La solution passe par la lecture de caractèristiques de lien avec `lstat` et le champs `st_size` de la structure `struct stat` nous donnera la longueur (en octets) de contenu du lien:

```
char *tamp;
strcut stat b;
int i;

lstat("../exo", &b);
tamp = (char *)malloc(b->st_size+1);
i=readlink("../exo",tamp,b->st_size);
tamp[i]='\0';
```

7.4 Modifications des attributs de lien symbolique

```
int lchmod(const char *reference, mode_t mode)
int lchown(const char *reference, uid_t uid, gid_t gid)
```

permettent de modifier le droit d'accès et le propriétaire d'un lien symbolique.

8 Lecture/écriture non bloquantes

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int descripteur, int commande, ...)
```

```
int m=fcntl(desc, GET_FL)
```

Cet appel retourne l'état de la description de fichier associé ouvert, en particulier le drapeaux `O_APPEND` et `O_NONBLOCK`. La valeur retourné contient également le mode d'ouverture de fichier.

```
fcntl(desc, F_SETFL, mode)
```

modifie l'état de description en fonction du paramètre `mode`. Les valeurs possibles de `mode` `O_APPEND` `O_NONBLOCK`, `O_APPEND|O_NONBLOCK` et 0.

```
int mode = fcntl(desc,F_GETFL);
mode |= O_NONBLOCK ;
fcntl(desc, F_SETFL, mode);
```

permet de basculer vers le mode non bloquant.

L'inverse, passer au mode bloquant :

```
int mode = fcntl(desc, F_GETFL);
mode &= ~O_NONBLOCK;
fcntl(desc, F_SETFL, mode);
```

Si `O_NONBLOCK` est activé et il n'y a pas de donnés à lire alors `read()` retourne -1 et `errno` prend la valeur `EAGAIN`.

Si `O_NONBLOCK` n'est pas activé et il n'y a pas de donnés à lire alors `read()` et bloqué en attente de donnés.

9 Verrouillage de fichiers réguliers

Les verrous sont attachés aux i-noeuds, donc l'effet s'applique sur tous les descripteurs (et v-noeuds) attachées. Le verrou est la propriété de processus qui l'a posé, lui seul peut le modifier ou enlever.

La portée d'un verrou – l'intervalle d'application soit $[a, b]$ soit $[a, \infty]$ si le verrou jusqu'à la fin du fichier.

Types de verrous:

- **verrous partagés** (de lecture, shared lock, read lock) : plusieurs verrous de ce types peuvent être posé en même temps avec les portée non disjointes.
- **verrous exclusifs** (d'écriture, exclusive lock, write lock) : un verrou de ce type ne peut pas être posé sur la portée d'un autre verrou peu importe son type.

Il y deux **mode opératoires** des verrous:

- **mode consultatifs** (advisory mode) : le verrou n'empêche pas les opération read et write mais il empêche la pose d'un autre verrou,
- **mode impératif** (mandatory mode) : agit directement sur les opérations entrée-sortie provoquant le blocage.

```
#include <sys/types.h>
#include <fcntl.h>
struct flock{
   short l_type; /* F_RDLCK F_WRLCK F_UNLCK */
   short l_whence ; /* SEEK_SET SEEK_CUR SEEK_END */
   off_t l_start; /*debut de la portee par rapport au L_WHENCE*/
   off_t l_len; /*longueur, 0 si jusqu'a la fin de fichier*/
   pid_t l_pid; /*identite du proprietaire de verrou*/
}
```

`F_UNLCK` correspond au déverrouillage. La portée de verrou peut dépasser la fin de fichier.

```
#include <fcntl.h>
int fcntl(int descriptor, int commande, struct flock *pverrou)
```

La commande prend une de valeurs suivantes:

- `F_SETLK` : demande non bloquante, réussit s'il n'y a pas de verrou incompatible. Si demande impossible à satisfaire parce il y a déjà un verrou alors `errno` est `EACCES` ou `EAGAIN`
- `F_SETLKW` demande bloquante. Inter-blocage est détecté et `errno` positionné à `EDEADLK`
- `F_GETLK` test d'existence d'un verrou incompatible avec le verrou donné. S'il existe le verrou incompatible `pverrou` contient les information sur ce verrou.

Si un processus qui détient déjà un verrou essaie de poser un autre verrou avec attente (commande `F_SETLKW`) alors nous pouvons avoir un dead-lock. Le système détecte cette situation et `fcntl` échoue avec `errno=EDEADLK`.

S'il n'existe pas de verrou incompatible alors le champs `l_type` de `pverrou` contient `F_UNLCK` (les autres champs inchangés).

Le processus peut déverrouiller partiellement une zone qu'il a verrouillée ou changer le type de verrou.

La fermeture d'un descripteur libère tous les verrous posés par le processus sur le i-noed correspondant (même si le processus a d'autre descripteurs sur le même fichier).

Exemple 12. Pour poser le verrou exclusif non-bloquant à partir de la postion courante jusqu'à la fin de fichier :

```
struct flock fl;
fl.l_type = F_WRLCK;
fl.whence = SEEK_CUR;
fl.l_start=0;
fl.l_len=0;
/*demande non bloquante de poser un verrou
  sur tout le fichier */
if( fcntl(desc, F_SETLK, &fl) == -1 ){
   if( errno==EAGAIN){
    /* verrouillage impossible,
       deja un verrou pose par un autre processus
       essayez plus tard encore une fois  */
   }
   else{
     /* tentative echoue pour une autre raison
     * probablement terminer le programme te regarder quel probleme
     */
   }
}

/* enlever le verrou */
```

```
fl.l_type=F_UNLCK;
fcntl(desc, F_SETLK, &fl)
```