Chapitre 11

Système de fichiers

UN fichier désigne un ensemble d'informations stockées sur le disque. Le **système de fichiers** est la partie du système d'exploitation qui se charge de gérer les fichiers. La gestion consiste en la création (identification, allocation d'espace sur disque), la suppression, les accès en lecture et en écriture, le partage de fichiers et leur protection en contrôlant les accès.

11.1 Introduction

11.1.1 Qu'est ce qu'un fichier ?

Pour le système d'exploitation, un fichier est une suite d'octets. Par contre, les utilisateurs peuvent donner des significations différentes au contenu d'un fichier (suites d'octets, suite d'enregistrements, arbre, etc.). Chaque fichier est identifié par un **nom** auquel on associe un emplacement sur le disque (une référence) et possède un ensemble de propriétés : ses **attributs**.

Le nom est en général, composé de deux parties séparées par un point. La partie qui suit le point est appelée **extension** (`prog.c, entete.h, fich.doc, archivo.pdf`, etc.). L'extension peut être de taille fixe, comme dans MS-DOS ou variable comme c'est le cas d'Unix/Linux ; obligatoire ou non. L'extension est nécessaire dans certains cas. Par exemple, le compilateur *C* rejettera le fichier `prog.txt` même si son contenu est un programme *C*. Le nom d'un fichier peut être sensible à la typographie : ainsi en Unix/Linux `Archivo` $\neq$ `archivo`.

11.1.2 Cycle de vie d'un fichier

Les fichiers — comme bien d'autres composants — ont un cycle de vie. Ils sont crées (ou ouverts), on les fait modifier (écrire), on lit à partir d'eux, et finalement, peut être, ils meurent (sont effacés). Ceci est illustré par la figure 11.1, avec des appels système qu'on étudiera dans le présent Chapitre.

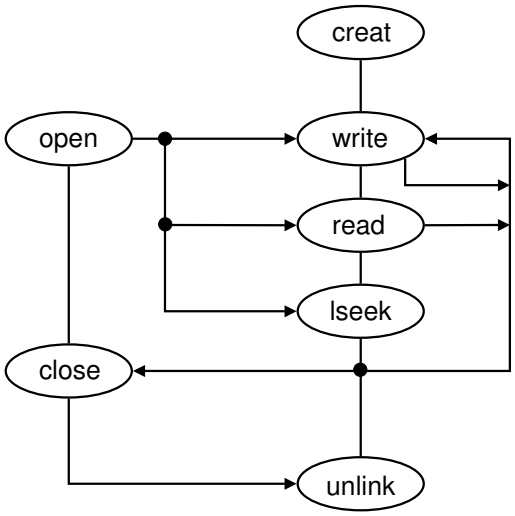


FIG. 11.1 – Le cycle de vie d'un fichier.

11.1.3 Types de fichiers

Dans un système, il existe plusieurs types de fichiers. Unix/Linux et MS-DOS ont des fichiers ordinaires, des répertoires, des fichiers spéciaux caractère et des fichiers spéciaux bloc.

Les **fichiers ordinaires** contiennent les informations des utilisateurs. Ils sont en général des fichiers ASCII ou binaires. Les **répertoires** sont des fichiers système qui maintiennent la structure du système de fichiers. Les **fichiers spéciaux caractère** sont liés aux Entrées/Sorties et permettent de modéliser les périphériques d'E/S série tels que les terminaux, les imprimantes et les réseaux. Les **fichiers spéciaux bloc** modélisent les disques.

Dans le système Unix/Linux, `"-"` désigne les fichiers ordinaires, `"d"` les répertoires, `"c"` les périphériques à caractères, `"b"` les périphériques à blocs, et `"p"` les tubes avec nom (*named pipe*). Les périphériques sont des fichiers désignés par des références.

Les **fichiers spéciaux bloc** et **caractère** vont identifier les dispositifs physiques du matériel : les disques, les rubans magnétiques, les terminaux, les réseaux, etc. Chaque type de dispositif a un contrôleur responsable de sa communication. Dans le système d'exploitation il y a une table qui pointe vers les différents contrôleurs de dispositifs. Tous les dispositifs seront alors traités comme de fichiers.

Dans Unix/Linux les périphériques comme les terminaux sont des fichiers spéciaux qui se trouvent sous le répertoire `/dev`. On copie par exemple le texte : "`abcd`" tapé au clavier, dans un fichier par la commande :

```
leibnitz> cat > fichier
 abcd
 ^D
```

et sur un terminal, par la commande :

```
leibnitz> cat > /dev/tty
abcd
^D
```

11.2 Accès aux fichiers

Pour accéder à un fichier il faut fournir au système de fichiers les informations nécessaires pour le localiser sur le disque, c'est-à-dire lui fournir un **chemin d'accès**. Les systèmes modernes permettent aux utilisateurs d'accéder directement à une donnée d'un fichier, sans le parcourir depuis le début du chemin.

11.2.1 Attributs des fichiers

Les attributs des fichiers diffèrent d'un système à un autre. Cependant ils peuvent être regroupés en deux catégories :

1. Les attributs qui servent à **contrôler les accès** comme le code de protection, le mot de passe, le propriétaire, etc.
2. Les attributs qui définissent le **type** et l'**état courant** du fichier : indicateur du type ASCII/binaire, taille courante, date de création, date de la dernière modification, etc.

11.2.2 i-nœuds

Dans le système Unix/Linux toutes les informations des fichiers sont rassemblées dans une structure associée au fichier, appelée **nœud d'information**, **i-nœud** ou **i-node**. L'i-nœud contient les informations suivantes : le type du fichier (régulier, répertoire, caractère, bloc ou tube) ; le code de protection sur 9 bits ; l'identificateur et groupe du propriétaire ; les dates de création, du dernier accès et de la dernière modification ; le compteur de références ; la taille et finalement la table d'index composée de 13 numéros de blocs et de pointeurs. La figure 11.2 montre la structure typique d'un i-nœud.

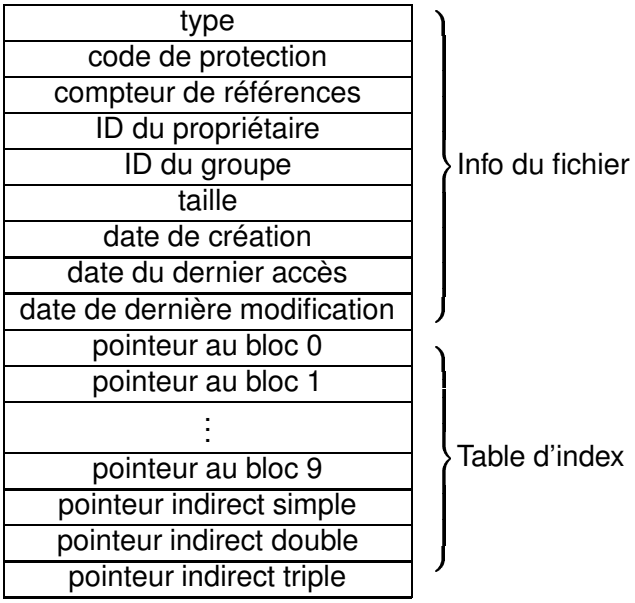


FIG. 11.2 – Structure d'un i-nœud.

11.3 Services Posix sur les fichiers

Les fichiers permettent de stocker des informations et de les rechercher ultérieurement. Les systèmes fournissent un ensemble d'appels système relatifs aux fichiers. Dans le cas d'Unix/Linux, les principaux appels système Posix relatifs aux fichiers sont :

- `open()` et `creat()` pour l'ouverture d'un fichier.
- `close()` pour la fermeture d'un fichier.

- `read()` pour la lecture d'un fichier.
- `write()` pour l'écriture dans un fichier.
- `lseek()` pour déplacer le pointeur de fichier.
- `stat()` pour récupérer des informations d'un fichier.
- `link()` pour créer un lien entre deux fichiers.
- `unlink()` pour supprimer un lien ou un fichier.

Ouverture d'un fichier

```
#include <unistd.h>
#include <fcntl.h>
int open(char * filename, int mode);
int open(char *filename, int mode, int permission);
int creat(char * filename, int permission);
```

En cas de succès, `open()` retourne un descripteur du fichier. En cas d'échec, il retourne `0`. Chaque fichier ouvert a un pointeur qui pointe sur un élément du fichier. Ce pointeur sert à parcourir le fichier. A l'ouverture du fichier, le pointeur de fichier pointe sur le premier élément du fichier, sauf si l'option `O_APPEND` a été spécifiée. Les lectures et les écritures se font à partir de la position courante du pointeur. Chaque lecture et chaque écriture entraînent la modification de la position du pointeur.

L'appel système `creat()`[1] est de moins en moins utilisée. Son usage est équivalent à celui de :
`open(char *filename, O_WRONLY|O_CREAT|O_TRUNC, int mode)`
Le `mode` est une combinaison de plusieurs élements assemblés par un `OU` logique. Il faut utiliser de façon obligée l'une de trois constantes :

- `O_RDONLY` : fichier ouvert en lecture exclusive.
- `O_WRONLY` : fichier ouvert en écriture exclusive.
- `O_RDWR` : fichier ouvert en lecture et en écriture.

On peut ensuite utiliser d'autres constantes qui permettent de mieux préciser l'usage :

- `O_CREAT` : créer le fichier ouvert même en lecture exclusive.
- `O_EXCL` : employé conjointement avec `O_CREAT`, cette constante garantit qu'on n'écrasera pas un fichier déjà existant. L'ouverture échoue si le fichier existe déjà.
- `O_TRUNC` : utilisée avec `O_WRONLY` ou avec `O_RDWR`, permet de ramener la taille d'un fichier existant à zéro.

[1] Bien qu'en anglais le mot correct soit *create*, Ken Thompson a décidé de la nommer `creat()`, à cause d'une raison inconnue.

Les **permissions** sont utilisées lors de la création d'un fichier. Elles servent à signaler les autorisations d'accès au nouveau fichier crée. On peut les fournir en représentation octale directement (précédées d'un `0`) ou bien on peut utiliser les constantes symboliques les plus fréquentes de la table 11.1.

Constante symbolique	Valeur octale	Signification de l'autorisation
`S_IRUSR`	`00400`	Lecture pour le propriétaire
`S_IWUSR`	`00040`	Écriture pour le propriétaire
`S_IXUSR`	`00100`	Exécution pour le propriétaire
`S_IRWXU`	`00700`	Lecture + écriture + exécution pour le propriétaire
`S_IRGRP`	`00020`	Lecture pour le groupe
`S_IROTH`	`00004`	Lecture pour tout le monde
`S_IWOTH`	`00002`	Écriture pour tout le monde
`S_IXOTH`	`00001`	Exécution pour tout le monde

TAB. 11.1 – Constantes symboliques de permission.

Ainsi si l'on utilise la combinaison :

`"S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH" = 0644 = rw-r-r-`

on donne les droits de lecture à tous et les droits de lecture et écriture seulement au propriétaire du fichier.

Exemple : Cet appel système crée, puis ouvre un fichier nommé `archivo` pour un accès en lecture/écriture. Le descripteur de fichier est `fd`. Le code de protection du fichier est `0600` ce qui correspond à `(wr-----)`.

```
int fd;
fd = open("archivo", O_CREAT | O_RDWR, 0600);
```

Exemple : Cet appel système ouvre le fichier `archivo` en mode écriture et positionne le pointeur à la fin du fichier. Les écritures dans le fichier ajouteront donc des données à la fin de celui-ci.

```
int fd;
fd = open("archivo", O_WRONLY | O_APPEND);
```

Exemple : Les appels suivants sont équivalents :

```
fd = creat("datos.txt", 0751);
fd = open("datos.txt", O_WRONLY | O_CREAT | O_TRUNC, 0751);
```

Fermeture des fichiers

L'appel système `close()` permet de libérer un descripteur de fichier `fd`. Le fichier pointé par le descripteur libéré est fermé s'il n'a plus aucun descripteur associé au fichier.

```
int close (int fd);
```

Lecture d'un fichier

```
#include <unistd.h>
int read(int fd, char* buf, int count);
```

L'appel système `read()` lit, à partir de la position courante du pointeur, `count` octets au maximum et les copie dans le tampon `buf`. En cas de succès, il retourne le nombre d'octets réellement lus. L'appel système `read()` retourne `0` si la fin de fichier est atteinte. Il retourne `-1`, en cas d'erreur.

Écriture dans un fichier

```
#include <unistd.h>
int write(int fd, void* buf, int count);
```

L'appel système `write()` copie `count` octets du tampon `buf` vers le fichier `fd`, à partir de la position pointée par le pointeur de fichier. En cas de succès, il retourne le nombre d'octets réellement écrits. Sinon, il renvoie `-1`.

Il faut retenir que `read()` et `write()` manipulent des octets, et que le programmeur a la responsabilité de les interpréter d'une ou autre forme. Par exemple, regardez les lectures de types différents de données à partir d'un descripteur du fichier `fd` (les écritures seraient tout à fait équivalentes) :

```
char c, s[N];
int i, v[N];

// 1 char :
   read(fd, &c, sizeof(char));
// N char's:
   read(fd, s, N*sizeof(char));
// 1 int :
   read(fd, &i, sizeof(int));
// N int's :
   read(fd, v, N*sizeof(int));
```

▶ **Exemple 1.** Le programme `copie-std.c` crée un fichier appelé `"fichier"` dans lequel il copie les données lues à partir du clavier. Les Entrées/Sorties sont effectuées avec des appels système (de bas niveau) sans utiliser la bibliothèque standard de *C* `"stdio.h"`.

Listing 11.1 – copie-std.c

```
#include <unistd.h>
#include <fcntl.h>

#define taille 80

int main( )
{
    int fd , nbcar;
    char buf[taille] ;

    // écrer un fichier
    fd = open("fichier", O_CREAT| O_WRONLY);
    if(fd==-1)
    {
        write(2,"Erreur d'ouverture\n", 25) ;
        return -1 ;
    }
    write(1, "Ouverture avec èsuccs\n" , 30) ;
    // copier les édonnes introduitesà  partir
    // du clavier dans le fichier
    while ((nbcar = read(0, buf, taille)) > 0)
        if( write(fd, buf,nbcar) == -1) return -1;
    return 0;
}
```

▶ **Exemple 2.** Le programme `copie.c`, copie un fichier dans un autre.

Listing 11.2 – copie.c

```
#include <sys/wait.h>  // wait
#include <fcntl.h>     // les modes
#include <unistd.h>    // les appels èsystme

#define TAILLE 4096
int main (int argc, char*argv[])
{
   int status, src, dst, in, out;
   char buf[TAILLE];
   if ( fork () == 0)
```

```
    {
        if (argc !=3) exit(1);
        src = open(argv[1], O_RDONLY);
        if (src < 0) exit(2);
        dst = open( argv[2], O_RDWR | O_CREAT, 0666);
        if(dst<0) exit(3);
        while (1)
        {
            in = read(src, buf, TAILLE);
            if (in<0) exit(4);
            if(in==0) break;
            out = write(dst, buf, in);
            if( out<=0) exit(5);
        }
        close (src);
        close(dst);
        exit(0);
    }
    else
    {
        if (wait(&status)>0 && (status>>8)==0)
            write(1,"done\n",5);
        else write(2, "error\n",6);
        return 0;
    }
}
```

L'exécution de `copie.c` montre qu'on peut copier n'importe quel type de fichiers, même des fichiers exécutables. Cependant cette version du programme a écrasé le mode (`rwxr-xr-x` a été transformé en `rw-r-r-`) :

```
leibnitz> ls -l stat*
-rwxr-xr-x    1 jmtorres prof    15926 Nov 15 14:53 stat*
leibnitz> copie stat stat2
done
leibnitz> ls -l stat*
-rwxr-xr-x    1 jmtorres prof    15926 Nov 15 14:53 stat*
-rw-r--r--    1 jmtorres prof    15926 Nov 15 15:12 stat2
leibnitz>
```

Accès direct à une position dans le fichier

```
int lseek(int fd, int offset, int origine);
```

L'appel système `lseek()` permet de modifier la position courante du pointeur de parcours du fichier. Il retourne la nouvelle position du pointeur ou

`-1` en cas d'erreur. Le positionnement dans un descripteur est mémorisé dans la Table de fichiers et non dans la Table de descripteurs. Le deuxième argument `offset` fournit la nouvelle valeur de la position. Le troisième argument `origine` peut prendre comme valeurs :

- `SEEK_SET` : Position = `offset`.
- `SEEK_CUR` : Position courante + `offset`.
- `SEEK_END` : Taille du fichier + `offset`.

Observez le diagramme de la figure 11.3.

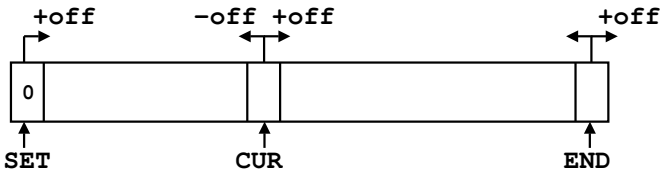


FIG. 11.3 – Pointeur de fichier : `lseek()`.

Ainsi, pour connaître la position courante on peut utiliser simplement : `lseek(fd,0,SEEK_CUR)`.

► **Exemple 3.** Soit le fichier `archivo` contenant la chaîne : "`le printemps`". A l'ouverture de ce fichier, le pointeur de fichier est positionné sur le 1er caractère "`l`". Sa valeur est `0`.

```
char buf ;
 //déplacer le pointeur du fichier à la position 3 du fichier
 lseek(fd, 3,SEEK_SET);

 // la valeur du pointeur est 3.
 // lire un caractère dans buf (buf reçoit p)
 read(fd, buf, 1);

 // positionner le pointeur deux caractères plus loin
 lseek(fd,2,SEEK_CUR);

 // la valeur du pointeur est 4+2.
 // lire un caractère dans buf (buf reçoit n)
 read(fd,buf,1);

 //positionner le pointeur à la fin du fichier
 lseek(fd,0,SEEK_END);
 read(fd,buf,1) ; // retournera 0
```

▶ **Exemple 4.** Accès directs aux éléments d'un fichier avec `seek.c` :

Listing 11.3 – seek.c

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

int main (int argc, char* argv [])
{
   int PLine[200];
   int fd, ncar, i;
   int  nline=0, pfich=0;
   char buffer[4096];

   // ouverture du fichier en lecture seulement
   fd = open (argv[1], O_RDONLY);
   if( fd==-1) exit(1); // erreur d'ouverture
   PLine[0]=0; //position de la ligne 0
   while (1)
   {   //lecture du fichier
       ncar = read(fd, buffer, 4096);
       if(ncar == 0) break; //fin de fichier
       if(ncar ==-1) exit(1); // erreur de lecture
       for( i=0; i<ncar; i++)
       {
           pfich++;
           // fin de ligne érencontre
           if(buffer[i]=='\n') PLine[++nline]=pfich;
         }
   }
   PLine[nline+1]=pfich;
   for(i=0; i<=nline; i++)
       printf(" PLine[%d] = %d \n",i,PLine[i]) ;
   // èaccsà  la èpremire ligne en utilisant lseek.
   lseek(fd, PLine[0], SEEK_SET);
   ncar = read (fd, buffer, PLine[1]-PLine[0]);
   //afficherà  lé'cran le contenu du buffer
   write(1, buffer, ncar);
   return 0;
}
```

Exécution de `seek.c` :

```
leibnitz> cat >> exemple
nom
prenom
```

```
code
leibnitz> cat exemple
nom
prenom
code
leibnitz> seek exemple
PLine[0] = 0
PLine[1] = 4
PLine[2] = 11
PLine[3] = 16
nom
leibnitz>
```

Duplication des descripteurs de fichier

Chaque processus a une table de descripteurs de fichier. Cette table contient les descripteurs de fichier du processus. Chaque descripteur pointe vers un fichier ouvert pour le processus. Lorsqu'un processus est créé, le système lui ouvre automatiquement trois fichiers :

- L'entrée standard auquel il associe le descripteur `0`.
- La sortie standard auquel il associe le descripteur `1`.
- La sortie standard auquel il associe le descripteur `2`.

Si le processus ouvre explicitement un fichier, le descripteur associé au fichier serait `3` et ainsi de suite.

```
int dup (int oldfd);
int dup2(int oldfd, int newfd);
```

L'appel système `dup()` permet d'associer à un même fichier plusieurs descripteurs. `dup()` associe le plus petit descripteur libre au fichier pointé par le descripteur `oldfd`. L'appel système `dup2()` associe le descripteur `newfd` au fichier pointé par `oldfd`.

▶ **Exemple 5.** Duplication de descripteurs de fichier :

Listing 11.4 – dup.c

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

int main(void)
{
```

```
    int fd1,
        fd2,
        fd3;

    fd1 = open ("test", O_CREAT | O_RDWR , 0644);
    if(fd1==-1)
      {
        perror("open");
        exit(1) ;
      }
    printf("fd1 = %d\n", fd1);
    write(fd1, "hello ",6);
    fd2 = dup(fd1); // fd2 et fd1 sont éassocis au fichier test
    printf("fd2 = %d\n", fd2);
    write(fd2, "wor",3);
    close(0);
    fd3 = dup(fd1); //fd3 et fd1 sont éassocis au fichier test
    printf("fd3 = %d\n", fd3);
    write(0, "ld",2);
    dup2 (3,2);
    write(2, "!\n", 2);
    return 0;
}
```

Exécution de `dup.c` :

```
leibnitz> gcc -o dup dup.c
leibnitz> dup
fd1 = 3
fd2 = 4
fd3 = 0
leibnitz> ls -l test
-rw-r--r--    1 jmtorres prof   13   Nov 15 16:03 test
leibnitz> more test
hello world!
leibnitz>
```

Il faut retenir que si deux descripteurs de fichiers sont associés à la même entrée de la table, le déplacement du pointeur avec `lseek()` (ou bien avec `read/write`) affectera l'autre.

▶ **Exemple 6.** Observez le petit programme `lseek.c` :

Listing 11.5 – lseek.c

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

```
int main(void)
{
   int fd;

   if ((fd = open("file", O_RDWR)) < 0)
       exit(1);
   if (lseek(fd, 10, SEEK_SET) < 0)
       exit(1);
   if (write(fd, "123", 3) < 0)
       exit(1);
   return 0;
}
```

Sans surprise, l'exécution sur un fichier `file` :

```
pascal> cat file
ABCDEFGHIJKLMNOPQRSTUVWXYZ
pascal> lseek
pascal> cat file
ABCDEFGHIJ123NOPQRSTUVWXYZ
pascal>
```

▶ **Exemple 7.** Mais dans cet autre programme `dup-lseek.c`, deux descripteurs différents avec la même entrée de la table de fichiers, vont modifier le pointeur du parcours du fichier :

Listing 11.6 – dup-lseek.c

```
#include <unistd.h>

int main(void)
{
  int fd;

  if ((fd = open("file", O_RDWR)) < 0)
       exit(1);
  dup2(fd, 1);
  if (lseek(fd, 10,SEEK_SET) < 0)
       exit(1);
  if (lseek(1, -5, SEEK_CUR) < 0)
       exit(1);
  if (write(fd,"123", 3) < 0)
       exit(1);
  return 0;
}
```

Exécution de `dup-lseek.c` sur un fichier `file` :

```
pascal> cat file
ABCDEFGHIJKLMNOPQRSTUVWXYZ
pascal> dup-lseek
pascal> cat file
ABCDE123IJKLMNOPQRSTUVWXYZ
pascal>
```

Le premier `lseek` met le pointeur à la position absolue `10`, mais le deuxième `lseek` (malgré l'utilisation d'autre `fd`, utilise la même entrée de la table) le fait reculer `5` octets.

▶ **Exemple 8.** Finalement, regardez l'exemple `dup3.c` suivant, qui ouvre le même fichier `file` deux fois :

Listing 11.7 – dup3.c

```
#include <unistd.h>

void main(void)
{
 int fd1, fd2;

 if((fd1 = open("file", O_RDWR)) < 0)
       exit(1);
 if((fd2 = open("file", O_WRONLY)) < 0)
       exit(1);
 if(lseek(fd1, 10, SEEK_SET) < 0)
       exit(1);
 if(write(fd1, "456", 3) < 0)
       exit(1);
 if(lseek(fd2, 5, SEEK_CUR) < 0)
       exit(1);
 if(write(fd2, "123", 3) < 0)
       exit(1);
 return 0;
}
```

et sa sortie :

```
pascal> cat file
ABCDEFGHIJKLMNOPQRSTUVWXYZ
pascal> dup3
pascal> cat file
ABCDE123IJ456NOPQRSTUVWXYZ
pascal>
```

Pourquoi ce comportement ? Souvenez vous que deux ouvertures *du même fichier* fournissent *deux descripteurs de fichiers*, avec des entrées indépendantes dans la Table de fichiers.

Obtention des informations d'un fichier

```
int stat(char* filename, struct stat * buf);
```

L'appel système `stat()` permet de récupérer des informations sur le fichier `filename`. En cas de succès, l'appel système `stat()` retourne `0` et les informations sur le fichier sont récupérées dans `buf`. En cas d'erreur, il retourne `-1`. `stat()` stocke l'information recuperée dans une structure du type `stat` :

```
struct stat {
   mode_t  st_mode;  /* le code de protection */
   ino_t   st_ino;   /* numéro du i-noeud du fichier */
   dev_t   st_dev;   /* dispositif */
   nlink_t st_nlink; /* nombre de liens physiques */
   uid_t   st_uid;   /* UID du propriétaire */
   gid_t   st_gid;   /* GID du groupe */
   off_t   st_size;  /* la taille du fichier (octets) */
   time_t  st_atime; /* dernier accès */
   time_t  st_mtime; /* la date de la dernière modification */
   time_t  st_ctime; /* dernière modification de données */
};
```

Ainsi `buf.st_ino` indique par exemple le numéro du i-nœud du fichier stocké dans `buf`. Il y a des macros prédéfinis qui prennent `buf.st_mode` comme argument et retournent `1` ou `0` :

- `S_ISDIR(buf.st_mode)` : vrai si le fichier est un répertoire.
- `S_ISCHAR(buf.st_mode)` : vrai si c'est un fichier spécial caractère.
- `S_ISBLK(buf.st_mode)` : vrai si c'est un fichier spécial bloc.
- `S_ISREG(buf.st_mode)` : vrai si c'est un fichier régulier.
- `S_ISFIFO(buf.st_mode)` : vrai si c'est un `pipe` ou `FIFO`.

▶ **Exemple 9.** Récupérer des informations à partir d'un fichier.

Listing 11.8 – stat.c

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

```
int main( int argc, char* argv[])
{
   struct stat buf ;
   mode_t mode ;
   int result ;

   result = stat(argv[1], &buf);
   if (result == -1)
       printf("Infos sur %s non disponibles\n", argv[1]);
   else
   {
       mode = buf.st_mode;
       if (S_ISDIR(mode))
           printf("%s est un érpertoire", argv[1]);
       else
       if (S_ISREG(mode))
           printf("%s est un fichier ordinaire", argv[1]);
       else
       if (S_ISFIFO(mode))
           printf("%s est un pipe", argv[1]);
       else
           printf("%s est un fichier special ") ;
       printf("\nTaille du fichier : %d\n", buf.st_size);
   }
   return 0;
}
```

Exécution de `stat.c` :

```
leibnitz> stat
Infos sur (null) non disponibles
leibnitz> stat stat.c
stat.c est un fichier ordinaire
Taille du fichier : 775
leibnitz>
```

Le programme suivant `estado.c` montre toutes les caractéristiques d'une liste de fichiers. Observez l'utilisation des fonctions `getpwuid()` et `getgrgid()` pour l'obtention de l'`UID` et du groupe de l'utilisateur.

▶ **Exemple 10.** Caractéristiques d'une liste de fichiers :

Listing 11.9 – estado.c

```
#include<stdio.h>
#include<sys/types.h>
```

```
#include<sys/stat.h>
#include<pwd.h>
#include<grp.h>

char permisos[]={'x','w','r'};
void estado (char *archivo);

int main (int argc, char **argv)
{
  int i;
  if(argc != 2)
  {
     printf("Usage : %s nombre_archivo\n", argv[0]);
     exit(1);
  }
  for (i=1; i<argc; i++)
     estado(argv[i]);
  return 0;
}

void estado (char *archivo)
{
  struct stat buf;
  struct passwd *pw;
  struct group *gr;
  int i;

  stat(archivo,&buf);
  printf("Fichier            : %s\n",archivo);
  printf("Dispositivo : %d, %d\n",
        (buf.st_dev & 0xff00) >> 8, buf.st_dev & 0x00ff);
  printf("i-noeud            : %d\n", buf.st_ino);
  printf("Type               : ");
  switch(buf.st_mode & S_IFMT)
  {
     case S_IFREG : printf("érgulier\n");
        break;
     case S_IFDIR : printf("érpertoire\n");
        break;
     case S_IFCHR : printf("éspcial caractere\n");
        break;
     case S_IFBLK : printf("éspcial bloc\n");
        break;
     case S_IFIFO : printf("FIFO\n");
        break;
  }
  printf("Permissions        : 0%o, ",buf.st_mode & 0777);
  for(i=0; i<9; i++)
     if(buf.st_mode & (0400 >> i))
```

```
            printf("%c", permisos[(8-i)%3]);
        else printf("-");
    printf("\n");
    printf("Liens              : %d\n",buf.st_nlink);
    printf("UID                : %d\n", buf.st_uid);
    printf("Nom                : ");
    pw = getpwuid(buf.st_uid);
    printf("%s\n", pw->pw_name);
    printf("GID                : %d\n", buf.st_gid);
    printf("Nom                : ");
    gr = getgrgid(buf.st_gid);
    printf("%s\n", gr->gr_name);
    switch(buf.st_mode & S_IFMT)
    {
        case S_IFCHR :
        case S_IFBLK : printf("Dispositif   : %d, %d\n",
                (buf.st_rdev & 0xff00) >> 8, buf.st_rdev & 0x00ff);
    }
    printf("Longueur           : %d bytes.\n",buf.st_size);
    printf("Dernier èaccs      : %s",
        asctime (localtime(&buf.st_atime)));
    printf("èDernire modif.    : %s",
        asctime (localtime(&buf.st_mtime)));
    printf("Dernier ch. dé'tat: %s",
        asctime (localtime(&buf.st_atime)));
}
```

Exécution de `estado.c`:

```
leibnitz> gcc -o estado estado.c
leibnitz> estado estado
Fichier           : estado
Dispositivo       : 0, 0
i-noeud           : 1854250
Type              : régulier
Permissions       : 0755, rwxr-xr-x
Liens             : 1
UID               : 11047
Nom               : jmtorres
GID               : 100
Nom               : prof
Longueur          : 17555 bytes.
Dernier accFs     : Tue Dec  3 15:03:24 2002
Dernière modif.   : Tue Dec  3 15:03:24 2002
Dernier ch. d'état: Tue Dec  3 15:03:24 2002
leibnitz>
```

Effacer un fichier

```
#include <unistd.h>
int unlink(char* nom_fichier);
```

L'appel système `unlink()` permet d'effacer un fichier. Il supprime le lien entre `nom_fichier` et l'i-nœud correspondant. Il retourne `1` si le fichier a été bien effacé ou `-1` s'il y a eu une erreur.

L'exemple suivant permet de constater que même si l'on efface un fichier, son contenu est encore accessible tant qu'il ne soit pas fermé.

▶ **Exemple 11.** Effacer (et encore re-utiliser) un fichier.

Listing 11.10 – unlink-cs.c

```
#include <unistd.h>
#include <fcntl.h>
#define N 16

int main (void)
 {
  char chaine[N];
  int fp;

  write(1, "éCration fichier\n",17);
  fp = open("test", O_RDWR | O_CREAT);
  if (fp<0)
       exit(0);
  write (fp, "0123456789abcdef", N);
  system ("ls -l test");
  write (1, "Effacement fichier\n",19);
  if (unlink("test") == -1)
  {
      perror ("unlink");
      exit (1);
  }
  system ("ls -l test");
  write (1, "Relecture du contenu du fichier\n",32);
  lseek(fp,0,SEEK_SET);
  read(fp, chaine, N);
  write(1, chaine, N);
  write(1, "Fermeture fichier\n",18);
  close (fp);
  return 0;
}
```

Exécution de `unlink-cs.c`:

```
leibnitz> gcc -o unlink unlink-cs.c
leibnitz> unlink
Création fichier
-rw-r--r--    1 jmtorres prof 26 Nov  1 17:17 test
Effacement fichier
ls: test: No such file or directory
Relecture du contenu du fichier
Lu : ABCDEFGHIJKLMNOPQRSTUVWXYZ
Fermeture fichier
leibnitz>
```

11.3.1 Transfert rapide de données

La forme classique de copier un fichier dans un autre, consiste en allouer un *buffer* de taille fixe, copier ensuite quelques données dans le *buffer*, puis écrire le *buffer* dans un autre descripteur et répéter jusqu'à ce que toutes les données soient copiées. Ceci n'est pas efficace ni en temps ni en espace car on utilise une mémoire additionnelle (le *buffer*) et on exécute une copie additionnelle des données. L'appel système `sendfile()` fournit un mécanisme efficient pour copier fichiers où le *buffer* est éliminé.

```
#include <sys/sendfile.h>
sendfile(write_fd, read_fd, &offset, stat_buf.st_size);
```

Les descripteurs peuvent être des fichiers, des **sockets**[2] ou d'autres dispositifs. Les paramètres sont `write_fd` le descripteur du fichier à écrire, `read_fd` le descripteur de lecture, `offset` un pointeur vers un offset dynamique et le nombre d'octets à copier `st_size`. On peut utiliser `fstat()` pour déterminer cette taille. `offset` contient le décalage (`0` indique le début du fichier). La valeur retournée est le nombre d'octets copiés.

► **Exemple 12.** Le programme `fcopie.c` montre cette technique efficace :

Listing 11.11 – fcopie.c

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

int main (int argc, char* argv[])
```

[2] Voir Chapitre **??** *Introduction aux systèmes distribués*, Section **??** *Sockets*.

```
{
 int fd_r;
 int fd_w;
 struct stat buf;
 off_t offset = 0;

 // fd source
 fd_r = open(argv[1], O_RDONLY);
 // obtenir sa taille
 fstat(fd_r, &buf);
 // fd de sortie avec les êmmes permissions du fd source
 fd_w = open(argv[2], O_WRONLY|O_CREAT, buf.st_mode);
 // Copier
 sendfile(fd_w, read_fd, &offset, buf.st_size);
 close(fd_r);
 close(fd_w);
 return 0;
}
```

11.4 Répertoires

Les systèmes d'exploitation modernes adoptent une **structure arborescente** pour représenter le système de fichiers. Les nœuds de l'arbre sont des répertoires et les feuilles sont des fichiers. Un répertoire est composé de fichiers et de sous répertoires. Pour accéder à un fichier, il suffit de spécifier les répertoires du chemin qui mène de la racine de l'arborescence au fichier (chemin d'accès). Dans le système Unix/Linux, chaque répertoire contient aussi sa propre référence `"."` et celle du répertoire immédiatement supérieur `".."`. Dans la figure 11.2 on montre un répertoire d'Unix.

i-nœud	Nom du fichier
20	.
3	..
100	chap1
2378	chap2
125	scheduler
⋮	⋮

TAB. 11.2 – Répertoire Unix.

Un répertoire est aussi considéré comme un fichier particulier composé

de fichiers[3].

11.5 Services Posix sur les répertoires

Les répertoires d'Unix/Linux possèdent une entrée par fichier. Chaque entrée possède le nom du fichier et le numéro de l'i-nœud. Le système Unix/Linux offre des appels système Posix pour manipuler les répertoires :

- `mkdir()` créer un répertoire.
- `rmdir()` supprimer un répertoire vide.
- `opendir()` ouvrir un répertoire.
- `closedir()` fermer un répertoire.
- `readdir()` lire les entrées d'un répertoire.
- `rewindir()` placer le pointeur d'un répertoire.
- `link()` créer une entrée dans un répertoire.
- `unlink()` effacer une entrée d'un répertoire.
- `chdir()` changer de répertoire de travail.
- `rename()` renommer un répertoire.
- `getcwd()` obtenir le nom du répertoire actuel.

Création d'un répertoire

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

L'appel système `mkdir()` crée un répertoire avec le `UID` du propriétaire = `UID` effectif et le `GID` du propriétaire = `GID` effectif. Comme arguments il reçoit `name` qui est le nom du répertoire, et `mode` qui sont les bits de protection. Il retourne `0` ou `-1` en cas d'erreur.

[3] Les répertoires de MS-DOS possèdent une entrée par fichier. Chaque entrée a la structure suivante :

Nom du fichier
Extension
Attributs
Réservé
Heure Der. Mod.
Date Der. Mod.
N° du 1er bloc
Taille

Effacer un répertoire

```
#include <sys/types.h>
int rmdir(const char *nom);
```

`rmdir()` efface le répertoire s'il est vide. Si le répertoire n'est pas vide, on ne l'efface pas. Arguments : `nom` correspond au nom du répertoire. Il retourne `0` ou `-1` en cas d'erreur.

Ouverture d'un répertoire

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

`opendir()` ouvre un répertoire comme une séquence d'entrées. Il met le pointeur au premier élément. `opendir()` reçoit comme argument `nom` qui est le nom du répertoire. Il retourne un pointeur du type `DIR` pour l'utiliser en `readdir()` ou `closedir()` ou bien `NULL` s'il a eu une erreur.

Fermer un répertoire

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

L'appel système `closedir()` ferme le lien entre `dirp` et la séquence d'entrées du répertoire. Arguments : `dirp` pointeur retourné par `opendir()`. Il retourne `0` ou `-1` en cas d'erreur.

Lecture des entrées du répertoire

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

L'appel système `readdir()` reçoit comme arguments `dirp` qui est le pointeur retourné par `opendir()`. Il retourne un pointeur à un objet du type `struct dirent` qui représente une entrée de répertoire ou `NULL` s'il y a eu erreur. Il retourne l'entrée suivante du répertoire associé à `dirp` et avance le pointeur à l'entrée suivante. La structure est dépendante de l'implantation. Pour simplification, on peut assumer qu'on obtient un membre `char *d_name`.

Changer du répertoire

```
int chdir(char *name);
```

`chdir()` modifie le répertoire actuel, à partir duquel se forment les noms relatifs. Arguments : `name` est le nom d'un répertoire. Il retourne `0` ou `-1` en cas d'erreur.

Renommer un répertoire

```
#include <unistd.h>
int rename(char *old, char *new);
```

`rename()` change le nom du répertoire `old`. Le nom nouveau est `new`. Arguments : `old` nom d'un répertoire existant, et `new` le nom nouveau du répertoire. Il retourne `0` ou `-1` en cas d'erreur.

▶ **Exemple 13.** Parcours d'un répertoire avec `liste-dir.c` :

Listing 11.12 – liste-dir.c

```
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>

#define MAX_BUF    256

int main(int argc, char **argv)
{
   DIR *dirp;
   struct dirent *dp;
   char buf[MAX_BUF];

   // montrer le érpertoire actuel
   getcwd(buf, MAX_BUF);
   printf("éRpertoire actuel : %s\n", buf);
   // ouvre le érpertoire épass comme argument
   dirp = opendir(argv[1]);
   if (dirp == NULL)
   {
      printf("Ne peut pas l'ouvrir %s\n", argv[1]);
   }
   else
   {
      // lit éentreà é entre
      while ( (dp = readdir(dirp)) != NULL)
            printf("%s\n", dp->d_name);
```

```
        closedir(dirp);
    }
    exit(0);
}
```

Exécution de `liste-dir.c`:

```
leibnitz> pwd
/home/ada/users/jmtorres/inf3600/logiciel
leibnitz> ls
bin/    chap2/  chap4/  chap7/  examen/    scheduler/  tmp/
chap1/  chap3/  chap5/  chap9/  parallel/  sockets/
leibnitz> chap9/liste-dir ../logiciel
Répertoire actuel : /home/ada/users/jmtorres/inf3600/logiciel
.
..
chap1
chap2
chap3
scheduler
bin
tmp
parallel
chap4
sockets
chap5
chap7
examen
chap9
leibnitz>
```

Le programme précèdant ne montre pas le contenu des sous-répertoires qu'il trouve. Pour cela il nous faut des appels récursifs du même programme. Ceci, nous le laissons comme exercice.

11.6 Répertoires sur Unix System V

En Unix System V l'appel système `getdents()` permet de lire une entrée du répertoire courant et insère dans `buf` les informations concernant cette entrée.

```
#include <sys/dirent.h>
int getdents (int fd, struct dirent* buf, int size)
```

La structure `dirent` est composée des champs suivants :
- `buf.d_ino` c'est le numéro du i-nœud.
- `buf.d_off` c'est la position relative de la prochaine entrée du répertoire.
- `buf.d_reclen` c'est la longueur de la structure d'une entrée du répertoire.
- `d_name` c'est le nom du fichier.

▶ **Exemple 14.** Parcours d'un répertoire avec `getd.c` :

Listing 11.13 – getd.c

```
#include <stdio.h>      //printf
#include <string.h>     // strcmp
#include <fcntl.h>      //O_RDONLY
#include <dirent.h>     //getdents
#include <sys/types.h> // mode_t
#include <sys/stat.h>  // les macros S_ISDIR
#include <unistd.h>     //open ...

int main (int argc, char* argv[])
{
int fd , nbcar, result;
struct stat buf;
mode_t mode;
struct dirent entree ;
result = stat (argv[1], &buf);
if (result == -1)
{
   printf("les infos sur %s non disponibles \n", argv[1]) ;
   exit(1);
}
mode = buf.st_mode ;
if ( S_ISDIR(mode))
   printf(" %s est un érpertoire\n ", argv[1]) ;
else
{
   printf(" %s n'est pas un érpertoire\n ", argv[1]) ;
   exit(1) ;
}
fd = open(argv[1], O_RDONLY) ;
if (fd == -1)
{
   printf("erreur d'ouverture du érp. %s\n", argv[1]) ;
   exit(1) ;
}
lseek(fd,0,SEEK_SET);
while (1)
```

```
{
   nbcar = getdents(fd, &entree, 40) ;
   if (nbcar == -1)
   {
       printf("Erreur de lecture du érp. %s\n",argv[1]);
       exit(1) ;
   }
   if(nbcar==0)
   {
       printf("Fin de lecture du érp. %s\n ",argv[1]) ;
       break ;
   }
   if (strcmp(entree.d_name, ".")!=0 &&
       strcmp(entree.d_name, "..") !=0 )
       printf("%s\n ", entree.d_name) ;
   lseek(fd,entree.d_off, SEEK_SET) ;
} // fin du while
close(fd) ;
printf("\nChangement de érpertoire de travail\n");
chdir(argv[1]);
system("pwd");
printf("Contenu du nouveau érpertoire de travail : \n");
system("ls");
exit(0);
}
```

Exécution de `getd.c` :

```
jupiter% gcc getd.c -o getd
jupiter% getd nouvrep
nouvrep est un répertoire
exemple
test
Fin de lecture du rép. nouvrep
Changement de répertoire de travail
/jupiter/home/bouchene/gestfichier/nouvrep
Contenu du nouveau répertoire de travail :
exemple test
```

11.7 Exercises

1. Expliquez la raison principale pour laquelle le système alloue une table de descripteur de fichier à chaque processus mais maintient qu'une seule table d'état de fichier pour tous les processus. Quelle information contient la table d'état de fichier ?
2. Expliquez la raison pour laquelle tout nouveau fichier créé est affecté par un numéro de descripteur à partir de `3` et non `0`.
3. Expliquez la raison pour laquelle les descripteurs de fichier physique (sur disque) et logique (en mémoire) ne contiennent pas nécessairement la même information.
4. Expliquez le type d'information que contient le champ de compte de références pour un descripteur de fichier physique et aussi pour un descripteur de fichier logique.
5. Expliquez les étapes suivies et les structures de données utilisées lors d'une opération d'ouverture de fichier sous Unix/Linux.
6. Expliquez un avantage et un désavantage pour chacune des techniques d'allocation de blocs suivante :
   (a) Allocation contiguë de blocs.
   (b) Allocation de blocs linéairement chaînés.
   (c) Allocation de blocs par indexage.
7. Considérez un système d'allocation de blocs sous Unix/Linux ayant les caractéristiques suivantes.
   (a) Taille du bloc de 16 Ko.
   (b) Taille de pointeurs de blocs de 32 bits.

   Calculez la taille maximale d'un fichier. En pratique, il n'existe pas de fichiers d'application qui nécessiteraient la capacité calculée ci-dessus. Alors, donnez une raison pour laquelle le système maintient telle capacité maximale.
8. On considère un système disposant d'un système de fichiers similaire à celui d'Unix avec une taille de blocs de données de 4K (4096 octets) et des pointeurs (numéros de blocs) définis sur 4 octets. On supposera que le i-nœud de chaque fichier compte 12 pointeurs directs, 1 pointeur indirect simple, 1 pointeur indirect double et 1 pointeur indirect triple. On désire créer un fichier contenant un total de 20.000.000 (vingt millions) de caractères (caractères de fin de ligne et

de fin de fichier compris). Quelle est la fragmentation interne totale sur le disque résultant de la création de ce fichier ?

9. Réécrire le code du programme `unlink-cs.c` en utilisant un processus fils et `exec()` en lieu de la fonction `system()`.
10. Écrire un programme `rec-dir.c` afin de parcourir l'arbre des sous-répertoires de façon récursive.

Chapitre 12

Stockage et partage de fichiers

Les fichiers de données sur les disques se répartissent dans des blocs de taille fixe. La lecture ou l'écriture d'un élément d'un fichier impliquera donc le transfert du bloc entier qui contient cet élément. Pour un accès rapide, on aura donc intérêt à prendre des blocs de grandes tailles. Cependant, les fichiers, y compris les fichiers de 1 octet, ont une taille minimale de 1 bloc. Si un disque comprend beaucoup de fichiers de petites tailles et si les blocs sont de grandes dimensions, l'espace gaspillé sera alors considérable. Il faut donc implanter des techniques adéquates de stockage de fichiers. Il y a trois manières de faire l'allocation des fichiers : l'**allocation contiguë**, la **liste chaînée de blocs** et l'**indexation par i-nœuds**.

12.1 Stockage des fichiers

Des études sur de nombreux systèmes ont montré que la taille moyenne d'un fichier est de 1 Ko. En général, les tailles de blocs fréquemment utilisées sont de 512, 1024, ou 2048 octets. Chaque disque conserve, dans un ou plusieurs blocs spécifiques, un certain nombre d'informations telles que le nombre de ses blocs, leur taille, leurs états, entre autres. À chaque fichier correspond une liste de blocs contenant ses données. L'allocation est en général, non contiguë et les blocs sont donc répartis quasi aléatoirement sur le disque. Les fichiers conservent l'ensemble de leurs blocs suivant deux méthodes : la **liste chaînée** et la **table d'index** par i-nœuds.

12.1.1 Allocation contiguë

La table d'allocation de fichiers contient seulement une entrée par fichier, avec le bloc de début et la taille du fichier. L'**allocation contiguë** est simple et performante, mais a le grave défaut de remplir rapidement l'espace avec des zones inutilisables (à cause de la fragmentation externe), et dans ce cas la compactation n'est pas envisageable avec des temps raisonnables. En plus il faudrait déclarer la taille du fichier au moment de sa création.

12.1.2 Liste chaînée

Les blocs d'un même fichier sont chaînés sous une représentation de **liste chaînée** montrée à la figure 12.1. Chaque bloc contiendra des données ainsi que l'adresse du bloc suivant. Le fichier doit mémoriser le numéro du 1er bloc. Par exemple, si un bloc comporte 1024 octets et si le numéro d'un bloc se code sur 2 octets, 1022 octets seront réservés aux données et 2 octets au chaînage du bloc suivant.

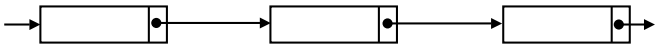


FIG. 12.1 – Liste chaînée.

Cette méthode rend l'accès aléatoire aux éléments d'un fichier particulièrement inefficace lorsqu'elle est utilisée telle quelle. En effet, pour atteindre un élément sur le bloc n d'un fichier, le système devra parcourir les n-1 blocs précédents.

Le système MS-DOS utilise une version améliorée de listes chaînées. Il conserve le premier bloc de chacun des fichiers dans son répertoire. Il optimise ensuite l'accès des blocs suivants en gardant leurs références dans une **Table d'Allocation de Fichiers** (**FAT**)[1]. Chaque disque dispose d'une table **FAT** et cette dernière possède autant d'entrées qu'il y a de blocs sur le disque. Chaque entrée de la **FAT** contient le numéro du bloc suivant. Par exemple, dans la table suivante :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x	x	EOF	13	2	9	8	L	4	12	3	E	EOF	EOF	L	...

"**x**" indique la taille du disque, "**L**" désigne un bloc libre et "**E**" un bloc endommagé. Le fichier commençant au bloc 6, sera constitué des blocs :

[1] Il existe de **FAT**s de 16 et de 32 bits.

$6 \rightarrow 8 \rightarrow 4 \rightarrow 2$. Le parcours de la FAT est nettement plus rapide que la chaîne de blocs. Cependant elle doit constamment être tout entière en mémoire, afin d'éviter les accès au disque pour localiser un bloc.

12.1.3 Table d'i-nœuds

Le système Unix associe à chaque fichier un numéro unique d'identification. A chaque numéro est associé un ensemble d'informations appelé **nœud d'information** ou **i-nœud**. Parmi les champs de l'i-nœud, la **table d'index** indique l'emplacement physique du fichier. Elle est composée de 13 entrées. Les 10 premières contiennent les numéros des 10 premiers blocs composant le fichier. Pour les fichiers de plus de 10 blocs, on a recours à des indirections. Le bloc n° 11 contient le numéro d'un bloc composé lui-même d'adresses de blocs de données. Si les blocs ont une taille de 1024 octets et s'ils sont numérotés sur 4 octets, le bloc n° 11 pourra désigner jusqu'à 256 blocs. Au total, le fichier utilisant la simple indirection aura alors une taille maximale de 266 blocs. De la même manière, le bloc n° 12 contient un pointeur à double indirection, et le bloc n° 13 un pointeur à triple indirection. Sur la figure 12.2 nous montrons l'usage des indirections simples, doubles et triples d'un i-nœud.

Un fichier peut avoir une taille maximale de 16 Go quand il utilise des pointeurs à triple indirection. La table d'index comporte jusqu'à 4 niveaux d'indexation, mais seul le premier niveau est présent en mémoire lorsque le fichier correspondant est ouvert.

Dans Linux, la table d'i-nœuds diffère un peu de celle utilisée dans Unix **System V**. Sous Linux la table est composée de 12 entrées pour les blocs et de 3 entrées pour les indirections.

12.2 Le système de fichiers `/proc`

Linux comporte le pseudo-système de fichiers `/proc`. Ceci est en réalité un répertoire qui contient des entrées numérotées par les processus en cours. Les entrées possèdent de l'information sous un format lisible par les personnes.

Statistiques du système

L'entrée de `/proc/uptime` contient de l'information en secondes du temps de démarrage du système et du temps mort :

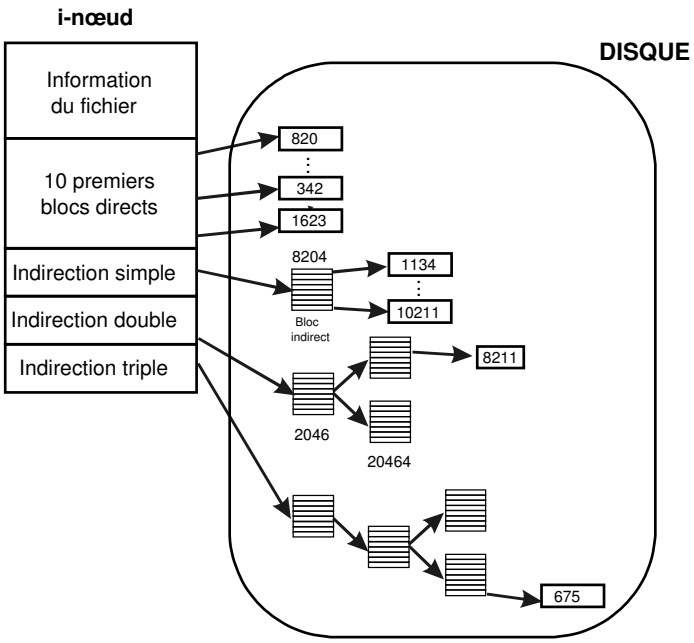


FIG. 12.2 – Indirections d'un i-nœud.

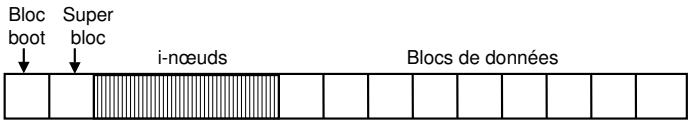


FIG. 12.3 – i-nœud et super-bloc.

```
leibnitz> cat /proc/uptime
547217.94 288458.71
```

Variable d'environnement

L'entrée de `/proc/environ` contient de l'information de l'environnement. Le programme suivant montre la variable d'environnement obtenue à partir de l'entrée `/proc`

Listing 12.1 – env.c

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

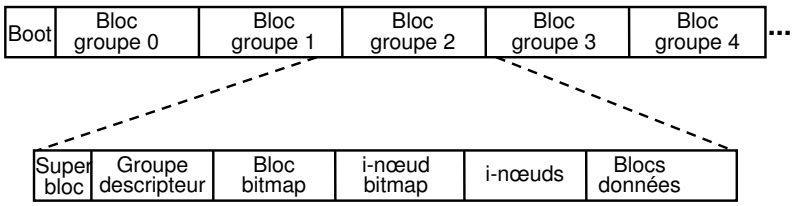


FIG. 12.4 – Partition EXT2 (en haut) et un groupe de blocs EXT2 (en bas).

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

int main(int argc, char* argv[])
{
  int pid = atoi(argv[1]);
  int fd;
  char filename[24];
  char environment[8192];
  size_t length;
  char* next_var;

  // éégnrer le nom d'environnement pour le processus
  snprintf(filename, sizeof (filename), "/proc/%d/environ",
         (int) pid);
  // lire le fichier
  fd = open(filename, O_RDONLY);
  length = read(fd, environment, sizeof(environment));
  close(fd);
  environment[length] = '\0';
  // cycle sur les variables
  next_var = environment;
  while (next_var < environment + length)
  {
    printf ("%s\n", next_var);
    next_var += strlen (next_var) + 1;
  }
  return 0;
}
```

Par exemple, dans une fenêtre on lance la commande `cat` qui reste bloqué en attente d'un argument de la part de l'utilisateur :

```
leibnitz> cat
...
```

et dans un autre terminal, on vérifie les processus et on lance le programme `env.c` avec le numéro de processus de `cat` :

```
leibnitz> ps -u jmtorres
 PID TTY          TIME CMD
1956 pts/4    00:00:00 tcsh
5764 pts/4    00:00:00 cat
5768 pts/1    00:00:00 tcsh
5819 pts/1    00:00:00 ps
leibnitz> gcc -o env env.c
leibnitz> env 5764
REMOTEHOST=torres.dgi.polymtl.ca
TERM=ansi
HZ=100
HOME=/home/ada/users/jmtorres
SHELL=/bin/tcsh
PATH=/home/ada/users/jmtorres/bin:/home/ada/users/jmtorres/
home/ada/users/jmtorres/office52:/home/ada/users/jmtorres/
bin/X11:/usr/local/bin:/usr/local/sbin:/usr/local/qt/bin:
/usr/openwin/bin:/usr/X11/bin:/usr/share/texmf/bin:/usr/bin:/
usr/local/java/j2sdk1.4.0_01/bin:/bin:/usr/bin:/sbin:/usr/games:/
usr/local/Simscript/bin:/usr/local/:/usr/local:/usr/local/Eclips
e/bin/i386_linux
...
HOSTNAME=leibnitz
...
EDITOR=emacs
...
CONSOLE=OFF
leibnitz>
```

Fichier de descripteurs de processus

L'entrée `fd` est un sous-répertoire qui contient des entrées pour les fichiers descripteurs ouverts par un processus. Chaque entrée est un lien symbolique au fichier ou au dispositif ouvert dans ce fichier descripteur. On peut écrire ou lire de ces liens symboliques ; écrire vers ou lire du fichier correspondant ou du dispositif ouvert par le processus. Les entrées dans le sous-répertoire `fd` sont nommées par les numéros du fichier descripteur.

Nous présentons une stratégie élégante qu'on peut essayer avec les entrées `fd` en `/proc`. Ouvrir une fenêtre shell, et trouver le processus ID du processus `shell` en exécutant `ps` :

```
bash-2.05a$ ps
```

```
 PID TTY          TIME CMD
1956 pts/4    00:00:00 tcsh
2413 pts/4    00:00:00 bash
2414 pts/4    00:00:00 ps
bash-2.05a$
```

Dans ce cas, le shell (`bash`) s'exécute dans le processus `2413`. Maintenant, ouvrir une deuxième fenêtre et voir le contenu du sous-répertoire `fd` pour ce processus :

```
bash-2.05a$ ls -l /proc/2413/fd
total 0
lrwx------ 1 jmtorres prof  64 Nov 11 15:53 0 -> /dev/pts/4
lrwx------ 1 jmtorres prof  64 Nov 11 15:53 1 -> /dev/pts/4
lrwx------ 1 jmtorres prof  64 Nov 11 15:53 2 -> /dev/pts/4
lrwx------ 1 jmtorres prof  64 Nov 11 15:53 255 -> /dev/pts/4
bash-2.05a$
```

Les descripteurs de fichier `0`, `1`, et `2` sont initialisés à l'entrée, la sortie et l'erreur standard respectivement. Ainsi, en écrivant à `/proc/2413/fd/1`, on écrit au dispositif attaché à `stdout` par le processus shell — dans ce cas là, le pseudo `TTY` de la première fenêtre — et dans la deuxième fenêtre, on écrit un message sur ce fichier :

```
bash-2.05a$ echo "Hola..." >> /proc/2413/fd/1
bash-2.05a$
```

Le texte `"Hola..."` apparaît alors sur la première fenêtre.

12.3 Partage de fichiers

Lorsque plusieurs utilisateurs travaillent sur un même projet, ils doivent souvent **partager des fichiers**. Pour permettre le partage d'un fichier existant, le système Unix/Linux offre la possibilité d'associer plusieurs références à un fichier, en créant des **liens physiques** ou des **liens symboliques**. Par exemple, dans la figure 12.5, le fichier **X** est partagé entre les répertoires `/B/B/X` et `/C/C/C/X`

12.3.1 Liens physiques

Pour réaliser le partage d'un fichier se trouvant sur un disque, il suffit d'associer à son numéro d'i-nœud plusieurs chemins d'accès ou références.

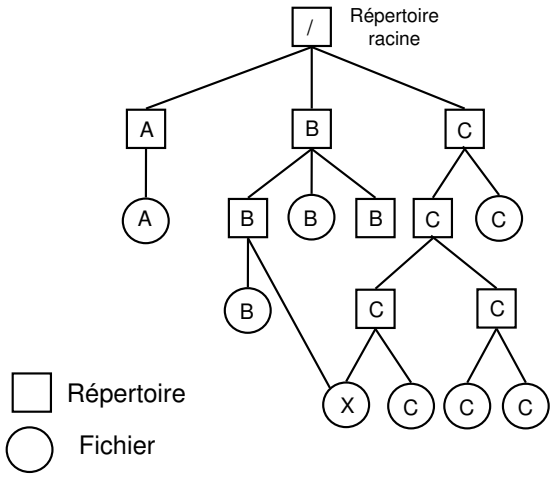


FIG. 12.5 – Fichier partagé.

Ce type de lien est **physique**. Les lien physiques sont possibles seulement si les chemins d'accès font références uniquement aux répertoires gérés par un *même système de fichiers*. En d'autres mots, on ne peut pas créer de lien à partir d'un répertoire d'une machine vers un fichier se trouvant sur une autre machine.

L'appel système qui permet de créer un lien physique à un fichier existant est `link()` :

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

La commande shell équivalente à `link()` est `ln` :

```
ln oldpath newpath
```

L'appel système `link()` associe un autre chemin d'accès `newpath` à un fichier ayant comme chemin d'accès `oldpath`. Les deux chemins désignent le même fichier (`newpath` et `oldpath` référencent le même fichier). Une fois les liens établis, le fichier pourra être désigné par l'un des deux noms (chemins d'accès).

Par exemple, l'appel système `link("origref", "../autreref")` opère comme suit :

– Il recherche dans le répertoire courant l'entrée telle que le nom de fichier soit `"origref"`, puis récupère le numéro de l'i-nœud. Il incrémente le nombre de références de l'i-nœud.

– Il ajoute dans le répertoire père du répertoire courant une entrée ayant comme nom de fichier "autreref" et le numéro d'i-nœud du fichier "origref".

Il existe un appel système qui permet de supprimer un lien physique à un fichier. Il s'agit de unlink :

```
int unlink(char * path);
```

unlink() supprime la référence path à un fichier. Si le fichier n'a plus de référence, il est supprimé du disque. Par exemple, unlink("origref") opère comme suit :

– Il localise, dans le répertoire courant, l'entrée telle que le nom de fichier soit "origref". Il récupère le numéro de l'i-nœud puis supprime cette entrée du répertoire.
– Il décrémente le nombre de références à l'i-nœud. Si le nombre de références devient égal à 0, le fichier est supprimé du disque (blocs des données et i-nœud).

La commande shell qui permet de supprimer un lien est rm. Regardez l'exemple suivant :

```
pascal> cat original
exemple d'un fichier avec plusieurs références.
pascal> ln original autreref
pascal> ls -l original autreref
-rw-r--r--    2 jmtorres prof      48 Nov 18 15:06 autreref
-rw-r--r--    2 jmtorres prof      48 Nov 18 15:06 original
pascal> cat autreref
exemple d'un fichier avec plusieurs références.
pascal> rm original
pascal> ls -l original autreref
/bin/ls: original: No such file or directory
-rw-r--r--    1 jmtorres prof      48 Nov 18 15:06 autreref
pascal> cat autreref
exemple d'un fichier avec plusieurs références.
pascal>
```

12.3.2 Liens symboliques

Les **liens symboliques** permettent de créer des liens vers des fichiers qui ne sont pas forcément gérés par le même système de fichiers. Ils présentent donc l'avantage de pouvoir constituer des liens vers des fichiers situés sur n'importe quel ordinateur à distance. Outre le chemin d'accès

sur la machine même, il faut placer l'adresse réseau de la machine dans le chemin d'accès.

Un lien symbolique est un pointeur indirect vers un fichier existant. La destruction d'un lien symbolique vers un fichier n'affecte pas le fichier. La commande `shell` d'Unix qui crée des liens symboliques est `ln` avec l'option `s` pour symbolique. Par exemple, la commande :

```
ln -s /usr/include/stdio.h stdio.h
```

crée une nouvelle entrée dans le répertoire courant pour un nouveau fichier dont le nom est `stdio.h`. Le contenu de ce nouveau fichier est le chemin d'accès :

```
/usr/include/stdio.h.
```

Le type du fichier créé est un link `(l)`.

Par exemple :

```
pascal> ln -s /usr/include/stdio.h stdio.h
pascal> ls -l stdio.h
lrwxrwxrwx  1 jmtorres prof   20 Nov 18 15:02 stdio.h ->
              /usr/include/stdio.h
pascal> cat stdio.h
...
```

cette commande affiche le contenu du fichier `/usr/include/stdio.h`

Problèmes Les liens permettent aux fichiers d'avoir plusieurs chemins d'accès. Les programmes qui parcourent tous les catalogues pour en traiter les fichiers peuvent rencontrer les fichiers partagés plusieurs fois. Un même fichier peut donc subir le même traitement plusieurs fois (par exemple, ils peuvent être recopiés plusieurs fois). Une fois les liens établis, plusieurs utilisateurs peuvent accéder à un même fichier. Les accès peuvent être concurrents. Le système Unix fournit des appels système qui permettent de contrôler les accès aux données d'un même fichier.

Lecture de liens symboliques

L'appel système `readlink()` récupère la source d'un lien symbolique. Il prend comme arguments le chemin au lien symbolique, un `buffer` pour recevoir le nom de la source et la longueur de ce `buffer`. `readlink` n'ajoute pas caractère `NUL` à la fin du nom du chemin. Cependant, il retourne le nombre de caractères du chemin. Si le premier argument à `readlink` pointe vers un fichier qui n'est pas un lien symbolique, `readlink` donne `errno = EINVAL` et retourne `-1`.

▶ **Exemple 1.** Le programme suivant imprime le chemin d'un lien symbolique spécifié sur la ligne de commande :

Listing 12.2 – sym.c

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

int main (int argc, char* argv[])
{
   char target_path[256];
   char* link_path = argv[1];

   // Essai de lire le chemin du lien symbolique
   int len = readlink(link_path, target_path, sizeof(target_path));
   if(len == -1)
   {   // L'appelé échou
       if (errno == EINVAL)
           printf("%s n'est pas un lien symbolique\n",
                  link_path);
       else
           // Un autre èproblme a eu lieu
           perror("readlink");
       return 1;
   }
   else
   {
       // NUL-terminaison du chemin objectif
       target_path[len] = '\0';
       // Imprimer
       printf ("%\n", target_path);
       return 0;
   }
}
```

Regardez l'utilisation de `sym.c` :

```
pascal> ln -s /usr/bin/wc my_link
pascal> sym my_link
/usr/bin/wc
pascal>
```

▶ **Exemple 2.** Voici un autre exemple sur les différences entre liens symboliques et physiques. L'utilisation de l'option `"-li"` (listage des i-nœuds) de `ls` sera utilisée. Montrons d'abord les liens symboliques :

```
pascal> cat > liga.txt
exemple d'un fichier avec plusieurs références. ^D
pascal> ls -l liga*
-rw-r--r--   1 jmtorres prof    48 Nov 18 15:06 liga.txt
pascal> ln liga.txt liga.ln
pascal> ls -li liga*
3052239 -rw-r--r--   2 jmtorres prof    48 Nov 18 15:06 liga.ln
3052239 -rw-r--r--   2 jmtorres prof    48 Nov 18 15:06 liga.txt
pascal> rm liga.txt
pascal> cat liga.ln
exemple d'un fichier avec plusieurs références.
pascal> ls -li liga*
3052239 -rw-r--r--   1 jmtorres prof    48 Nov 18 15:06 liga.ln
pascal>
```

► **Exemple 3.** Observez maintenant ce qui se passe avec les liens physiques :

```
pascal> cp liga.ln ligado.txt
pascal> ln -s ligado.txt liga2.ln
pascal> ls -li liga*
3052242 lrwxrwxrwx   1 jmtorres prof    10 Nov 18 15:51 liga2.ln ->
                       ligado.txt
3052240 -rw-r--r--   1 jmtorres prof    48 Nov 18 15:51 ligado.txt
3052239 -rw-r--r--   1 jmtorres prof    48 Nov 18 15:06 liga.ln
pascal> rm ligado.txt
pascal> cat liga2.ln
cat: liga2.ln: No such file or directory
pascal> ls -li liga*
3052242 lrwxrwxrwx   1 jmtorres prof    10 Nov 18 15:51 liga2.ln ->
                       ligado.txt
3052239 -rw-r--r--   1 jmtorres prof    48 Nov 18 15:06 liga.ln
pascal>
```

Est-ce que les liens symboliques marchent avec les répertoires ? Si vous en doutez, tapez les commandes suivantes :

```
pascal> ln -s /tmp temporal
pascal> cd temporal
pascal> ls -l
...
```

et maintenant :

```
pascal> cd /tmp
pascal> ls -l
...
```

Y-a-t-il des différences entre les deux sorties ?

L'exemple suivant, pris de [**?**], effectue la copie d'un fichier en utilisant des appels système **Posix** et la projection de fichiers en mémoire (Section **??** du Chapitre **??**) :

▶ **Exemple 4.** Copie avec projection en mémoire :

Listing 12.3 – copie-mem.c

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

void main(int argc, char **argv)
{
10   int i, fdo, fdd;
   char *org, *dst, *p, *q;
   struct stat bstat;

   if (argc!=3)
       {
           fprintf(stderr, "%s orig dest\n", argv[0]);
           exit(1);
       }
   // Ouvrir source en lecture
20   fdo=open(argv[1], O_RDONLY);
   // éCrer destination
   fdd=open(argv[2],O_CREAT|O_TRUNC|O_RDWR, 0640);
   // Longueur du fichier source
   if (fstat(fdo, &bstat)<0)
       {
           perror("fstat");
           close(fdo);
           close(fdd);
           unlink(argv[2]);
30           exit(1);
       }
   // Longueur de la destination = longueur de la source
   if (ftruncate(fdd, bstat.st_size)<0)
       {
           perror("ftruncate");
```

```
            close(fdo);
            close(fdd);
            unlink (argv[2]);
            exit(1);
        }
    // Projeter le fichier source
    if((org=mmap((caddr_t)0, bstat.st_size, PROT_READ,
                 MAP_SHARED,fdo,0)) == MAP_FAILED)
        {
            perror("mmap source");
            close(fdo);
            close(fdd);
            unlink (argv[2]);
            exit(1);
        }
    // Projeter le fichier destination
    if((dst=mmap((caddr_t)0,bstat.st_size, PROT_WRITE,
            MAP_SHARED,fdd,0)) == MAP_FAILED)
        {
            perror("mmap destination");
            close(fdo);
            close(fdd);
            unlink (argv[2]);
            exit(1);
        }
    close(fdo);
    close(fdd);
    // Copier
    p=org;
    q=dst;
    for(i=0; i<bstat.st_size; i++)
        *q++= *p++;
É    // liminer projections
    munmap(org, bstat.st_size);
    munmap(dst, bstat.st_size);
}
```

12.4 Verrouillage de fichiers

Les processus qui désirent effectuer des accès exclusifs à un fichier peuvent verrouiller, en une seule opération atomique, aussi bien un octet qu'un fichier entier. Deux sortes de **verrouillage de fichiers** sont disponibles : **partagé** et **exclusif**. Si une portion d'un fichier est verrouillée par un **verrou partagé**, une seconde tentative d'y superposer un verrou partagé est autorisée, mais toute tentative d'y poser un verrou exclusif sera rejetée tant que

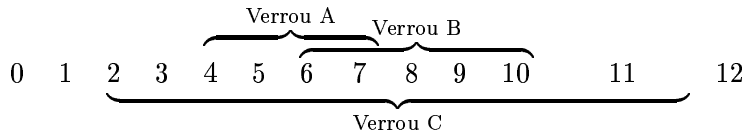
le verrou n'est pas relâché. Si une partie de fichier contient un **verrou exclusif**, toute tentative d'en verrouiller une quelconque portion sera rejetée tant que le verrou n'est pas relâché.

12.4.1 Verrouillage d'une partie de fichier

Lorsqu'un processus demande la pose d'un verrou, il doit spécifier s'il veut être bloqué si le verrou ne peut pas être posé. S'il choisit d'être bloqué, lorsque le verrou existant sera levé, le processus sera débloqué et son verrou pourra être posé. S'il ne veut pas être bloqué, l'appel système se termine immédiatement, avec un code qui indique si le verrouillage a pu être ou non effectué.

Par exemple, soient quatre processus concurrents **A**, **B**, **C** et **D** qui partagent un même fichier. Les processus font dans l'ordre les demandes de verrouillage suivantes :

- Le processus **A** demande la pose d'un verrou partagé sur les octets 4 à 7 du fichier.
- Le processus **B** demande de mettre un verrou partagé sur les octets 6 à 9.
- Le processus **C** demande la pose d'un verrou partagé sur les octets 2 à 11.
- Enfin, le processus **D** demande un verrouillage exclusif de l'octet 9 avec une requête bloquante en cas d'échec.



Les processus **A**, **B** et **C** arrivent à poser leurs verrous partagés. Par contre, le processus **D**, est bloqué jusqu'à ce que l'octet 9 devienne libre, c'est-à-dire jusqu'à ce que **B** et **C** libèrent leurs verrous.

12.4.2 Services Posix de verrouillage de fichiers

Posix a défini les appel système `lockf()` et `fcntl()` comme standard pour le verrouillage des parties d'un fichier.

`lockf()`

```
int lockf(int fd, int fonction, int size);
```

Le service `lockf()` retourne `0` en cas de succès et `-1` en cas d'échec. `fd` est un descripteur du fichier, le mode d'ouverture doit être `O_WRONLY` ou `O_RDWR` et `fonction` indique l'action à faire. Les valeurs possibles de `fonction` sont définies dans l'entête `<unistd.h>` comme suit :

`F_ULOCK` libère le verrou posé précédemment sur la partie composée de `size` octets contigus et pointée par le pointeur de fichier.

`F_LOCK` pose le verrou sur la partie libre composée de `size` octets contigus et pointée par le pointeur de fichier. Si la partie à verrouiller n'est pas libre, le processus demandeur est mis en attente jusqu'à ce que la partie devienne libre. Le processus n'est pas mis en attente si le système détecte un interblocage.

`F_TLOCK` pose le verrou sur la partie libre composée de `size` octets contigus et pointée par le pointeur de fichier. Si la partie à verrouiller n'est pas libre, l'appel système `lockf()` retourne -1 (échec de tentative de pose de verrou).

`F_TEST` teste si la partie composée de `size` octets contigus et pointée par le pointeur de fichier est libre ou non. Si la partie à verrouiller n'est pas libre, `lockf()` retourne `-1`.

▶ **Exemple 5.** Le programme `partage.c` montre comment effectuer le partage d'un fichier sans verrouillage :

Listing 12.4 – partage.c

```
#include <unistd.h>

int main(void)
{
   int i,
   fd=1;

   if(fork())
   {
10      for(i=0; i<5; i++)
        {
            write(fd,"pere ecris\n",11);
            sleep(1);
        }
   } else
   {
        for(i=0; i<4; i++)
        {
            write(fd,"fils ecris\n",11);
```

```
            sleep(1);
        }
    }
    return 0;
}
```

Exécution du programme `partage.c` :

```
pascal> gcc -o partage partage.c
pascal> partage
père écris
fils écris
père écris
fils écris
fils écris
père écris
père écris
fils écris
père écris
pascal>
```

▶ **Exemple 6.** Le programme `verrou.c` montre le partage d'un fichier avec verrouillage :

Listing 12.5 – verrou.c

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(void)
{
 int i, fd=1;

 if(fork()) //il s'agit du èpre
       {
           lseek(fd,0,0);
           if( lockf(fd, F_LOCK,1) <0)
           {
               write(fd,"pere lockf failed",18); return (-1);
           }
           for(i=0; i<5; i++)
           {
               write(fd,"pere ecris \n",13);
               sleep(1);
           }
```

```
            write(fd,"èpre va élibrer le verrou\n",26);
            lseek(fd,0,0);
            lockf(fd,F_ULOCK,0);
            wait(NULL);
        }
  else // il s'agit du fils
       {
            lseek(fd,0,0); //éverrouill l'octet 0
            if( lockf(fd, F_LOCK,1) <0)
            {
                write(fd,"fils lockf failed\n",18); return (-1);
            }
            for(i=0; i<4; i++)
                {
                    write(fd,"fils ecris \n",12);
                    sleep(1);
                }
            write(fd,"fils va élibrer le verrou\n",26);
            lseek(fd,0,0);
            lockf(fd,F_ULOCK,0);
       }
  close(fd);
  return 0;
}
```

Exécution de `verrou.c` :

```
pascal> gcc -o verrou verrou.c
pascal> verrou
père écris
père écris
père écris
père écris
père écris
père va libérer le verrou
fils écris
fils écris
fils écris
fils écris
fils va libérer le verrou
pascal>
```

`fcntl()`

L'appel système `fcntl()` est, en réalité beaucoup plus compliqué car le prototype est : `int fcntl(int fd, int commande, ...);` et sert à

plusieurs utilisations. Parmi elles on trouve la duplication de descripteurs, l'accès aux attributs d'un descripteur et le verrouillage des fichiers.

```
#include <fcntl.h>
int fcntl(int fd, int op, struct flock *verrou);
```

`fd` est le descripteur du fichier à verrouiller. `verrou` est une structure avec les champs :

- `l_type` : `int` Type de verrouillage.
- `l_whence` : `int` équivalent à l'appel système `lseek()` avec `SEEK_SET`, `SEEK_CUR` et `SEEK_END`.
- `l_start` : Début de la portion verrouillée du fichier.
- `l_len` : Longueur verrouillée en octets.
- `l_pid` : `PID` du processus. Donné par le système.

Le type de verrouillage peut être :

- `F_RDLCK` : Verrou partagé en lecture.
- `F_WRLCK` : Verrou exclusif en écriture.
- `F_UNLCK` : Absence de verrou.

`op` est l'une des opérations suivantes :

- `F_GETLK` : Accès aux caractéristiques d'un verrou existant.
- `F_SETLK` : Définition ou modification d'un verrou en mode non bloquant.
- `F_SETLKW` : Définition ou modification d'un verrou en mode bloquant.

▶ **Exemple 7.** Le programme **??** `lock.c` bloque un fichier jusqu'à ce qu'on tape `<Return>` pour le débloquer.

Listing 12.6 – lock.c

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char* file = argv[1];
  int fd;
  struct flock lock;

  printf("Ouverture de %s\n",file);
  fd = open(file, O_RDWR);
  printf("Verrouillage\n");
```

```
  memset(&lock, 0, sizeof(lock));
  lock.l_type = F_WRLCK;
  fcntl(fd,F_SETLKW,&lock);
  printf("éVerrouill. Presser ENTER pour debloquer...");
  getchar();
  printf("Deverrouillage\n");
  lock.l_type = F_UNLCK;
  fcntl(fd,F_SETLKW,&lock);
  close(fd);
  return 0;
}
```

Dans un terminal, créer un fichier de `TEST`, et le verrouiller avec `lock.c` :

```
leibnitz> cat > TEST
HOLA!
^D
leibnitz> lock TEST
Ouverture de TEST
Verrouillage
Verrouillé. Presser ENTER pour débloquer...
```

Dans un autre terminal, essayer de verrouiller le même fichier `TEST` :

```
leibnitz> lock TEST
Ouverture de TEST
Verrouillage
```

L'exécution reste gelée jusqu'à ce qu'on revienne à la première terminal et on presse `<Return>` :

```
...
Verrouillé. Presser ENTER pour débloquer...
Déverrouillage
leibnitz>
```

12.4.3 Verrouillage avec `flock()`

Le système Linux et Unix BSD offrent `flock()`, un autre appel système pour le verrouillage . Cet appel n'est pas standardisé Posix, mais il est encore utilisé.

```
#include <sys/file.h>
int flock(int fd, int operation);
```

Où `fd` est le descripteur de fichier, avec des `operation` valables :

- `LOCK_SH` **Verrou partagé**. Plus d'un processus peut verrouiller en même temps.
- `LOCK_EX` **Verrou exclusif**.
- `LOCK_UN` Déverrouiller le `fd`.
- `LOCK_NB` Non bloquer le verrouillage.

Regardez l'exemple suivant :

▶ **Exemple 8.** Utilisation de `flock` dans le programme `flock.c` :

Listing 12.7 – flock.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/file.h>

void display(int fd)
{
  int donnee;

  flow(fd,LOCK_SH);

  while(read(fd, &donnee, sizeof(int)) > 0)
        printf("donnee: %d\n",donnee);
  close(fd);
}

void add(int fd)
{
  int donnee;

  flow(fd,LOCK_EX);
  do
  {
    printf("Donnee? ");
    scanf("%d",&donnee);
    write(fd, &donnee, sizeof(int));
  } while (donnee!=-1);
  close(fd);
}

int flow(int fd, int op)
{
  printf("lock %s sur fd=%d\n",
        (op & LOCK_SH)?"partage":"exclusif", fd);
  if(flock(fd, op | LOCK_NB ) != -1)
     return 0;
  printf("Autre processus a le lock. Atteindre\n");
```

```
  return flock(fd,op);
}

int main(int argc, char *argv[])
{
  int fd;
  int opt;

  if(argc==1)
    {
      printf("flock 1=Montrer 2=Ajouter\n");
      exit(1);
    }
  fd = open("file", O_CREAT|O_RDWR, 0640);
  opt = atoi(argv[1]);
  if(opt==1) display(fd);
  if(opt==2) add(fd);
  return 0;
}
```

Dans un écran shell exécuter `flock.c` :

```
pascal> flock
flock 1=Montrer 2=Ajouter
pascal> flock 2
lock exclusif sur fd=3
Donnee? 1
Donnee? 2
Donnee? 3
Donnee? 4
```

Sans terminer, dans un deuxième shell exécuter `flock 1` :

```
pascal> flock 1
lock partagé sur fd=3
Autre processus a le lock. Atteindre
```

Et dans un troisième shell exécuter `flock 2` :

```
pascal> flock 2
lock exclusif sur fd=3
Autre processus a le lock. Atteindre
```

Si l'on revient dans la première session et on termine l'ajout des données :

```
...
Donnee? 4
Donnee? -1
pascal>
```

probablement la troisième (ou deuxième) session va se débloquer, et on aura la demande des données. Si l'on poursuit et on finit la saisie :

```
...
Autre processus a le lock. Atteindre
Donnee? 5
Donnee? 6
Donnee? -1
pascal>
```

C'est alors que la deuxième (ou troisième) session se réveille et montre son exécution. On espère évidement qu'elle montrera la séquence `1, 2, 3, 4, 5, 6, -1`, mais ce qu'elle montre est totalement différent :

```
...
lock partage sur fd=3 Autre processus a le lock. Atteindre
donnee: 5
donnee: 6
donnee: -1
donnee: 4
donnee: -1
pascal>
```

Pourquoi ne montre-t-elle pas la liste attendue ? À vous de répondre.

12.5 Antémémoire

Les processus ne peuvent pas manipuler directement les données du disque. Ils doivent être déplacés en mémoire centrale. De nombreux systèmes de fichiers cherchent à réduire le nombre d'accès au disque car le temps d'accès moyen au disque est égal à quelques dizaines de millisecondes.

La technique la plus courante pour réduire les accès au disque, consiste à utiliser une **antémémoire** (**buffer cache** ou **block cache**). L'antémémoire est un espace situé en mémoire centrale dans laquelle on charge un ensemble de blocs du disque. A chaque demande d'accès à un élément d'un fichier, on examine d'abord si le bloc désiré — celui qui contient l'élément du fichier — se trouve dans l'antémémoire. Si c'est le cas, la demande est

satisfaite sans avoir à accéder au disque. Sinon, pour satisfaire la demande, le bloc désiré doit être d'abord chargé à partir du disque dans l'antémémoire. S'il faut charger un bloc et que l'antémémoire est pleine, il faut retirer un des blocs de l'antémémoire pour le remplacer par celui demandé. Le bloc retiré doit être recopié sur le disque s'il a été modifié depuis son chargement. Pour choisir le bloc à retirer, il est possible d'utiliser un des algorithmes de remplacement de pages (FIFO, LRU, etc.).

Problème : Si des blocs placés dans l'antémémoire ont été modifiés et une panne survient avant de les copier sur disque, le système de fichiers passe dans un état incohérent.

Pour minimiser les risques d'incohérence, le système Unix recopie les blocs contenant les i-nœuds et les répertoires immédiatement après leur modification. Les blocs de données ordinaires sont recopiés manuellement s'ils doivent être retirés de l'antémémoire ou automatiquement par un démon toutes les 30 secondes. On peut également déclencher ce démon par l'appel système `sync()`.

Dans MS-DOS, les blocs de l'antémémoire sont recopiés sur disque à chaque modification. L'anténénoire n'est pas la seule façon d'améliorer les performances d'un système de fichiers. Une autre technique consiste à réduire le temps de recherche du bloc dans le disque (voir Chapitre **??** *Périphériques d'entrées/sorties*, Section **??** *Gestion du bras du disque*).

12.6 Cohérence du système de fichier

La plupart des ordinateurs ont un programme utilitaire qui vérifie la cohérence du système de fichiers. Ce programme est peut être exécuté à chaque démarrage du système surtout à la suite d'un arrêt forcé. La vérification de la cohérence peut se faire à deux niveaux : blocs ou fichiers.

12.6.1 Au niveau des blocs

Au niveau des blocs, le vérificateur construit deux tables. La première indique pour chaque bloc occupé, le nombre de fois où le bloc est référencé dans les i-nœuds des fichiers. La seconde indique pour chaque bloc libre, le nombre de fois où il figure dans la liste des blocs libres (ou la table de bits des blocs libres).

Si le système de fichiers est cohérent, chaque bloc a un 1 soit dans la première table, soit dans la seconde. Si un bloc n'apparaît ni dans la première table, ni dans la deuxième, le bloc est dit manquant. Le vérificateur

ajoute les blocs manquants à la liste des blocs libres. Si le bloc apparaît deux fois dans la liste des blocs libres, le vérificateur reconstruit la liste des blocs libres.

Le pire qui puisse arriver est qu'un bloc appartienne à deux fichiers (ou plus). Si on détruit l'un des deux fichiers, le bloc sera placé dans la liste des blocs libres (il sera alors à la fois libre et utilisé). Si on détruit les deux fichiers, le bloc figurera deux fois dans la liste des blocs libres. Dans ce cas, la meilleure solution consiste à allouer un bloc libre, à y copier le bloc commun et à remplacer le bloc commun dans l'un des deux fichiers par le nouveau. La dernière possibilité d'incohérence dans les deux tables est qu'un bloc soit, à la fois, utilisé et libre. La solution dans ce cas consiste à retirer le bloc de la liste des blocs libres.

12.6.2 Au niveau des fichiers

Le vérificateur vérifie la cohérence du point de vue des liens. Chaque numéro de nœud d'index doit apparaître autant de fois dans la structure arborescente qu'il possède de liens. Il commence au catalogue racine et parcourt récursivement toute l'arborescence pour déterminer pour chaque numéro d'i-nœud le nombre de références. Il compare ensuite le nombre de références obtenu pour chaque numéro d'i-nœud avec celui contenu dans l'i-nœud correspondant. Si les deux nombres sont différents, le système est dans un état incohérent. La solution consiste à corriger dans l'i-nœud, la valeur du nombre de références. Dans la plupart des systèmes Unix, le programme `fsck` effectue cette tâche à chaque démarrage, si nécessaire. Voir `man fsck` pour plus de détails.

12.7 Protection

Les systèmes de fichiers contiennent parfois des informations très importantes. Ils doivent donc protéger ces informations contre les accès non autorisés et les pertes. Les causes des pertes sont les catastrophes naturelles, les erreurs matérielles ou logicielles, les erreurs humaines. La plupart de ces problèmes peuvent être résolus si l'on effectue des sauvegardes assez régulièrement.

Le problème des intrus est beaucoup plus complexe. Pour assurer la protection contre les intrus, plusieurs mécanismes ont été mis en oeuvre, parmi lesquels l'identification de l'utilisateur (mots de passe ou identification physique) et les codes de protection des objets.

12.8 Exercices

1. On considère un système de fichiers tel que l'information concernant les blocs de données de chaque fichier est donc accessible à partir de l'i-nœud de celui-ci (comme dans Unix). On supposera que :
   – Le système de fichiers utilise des blocs de données de taille fixe 1K (1024 octets).
   – L'i-nœud de chaque fichier (ou répertoire) contient 12 pointeurs directs sur des blocs de données, 1 pointeur indirect simple, 1 pointeur indirect double et 1 pointeur indirect triple.
   – Chaque pointeur (numéro de bloc) est représenté sur 4 octets.

   (a) Quelle est la plus grande taille de fichier que ce système de fichiers peut supporter ?

   (b) On considère un fichier contenant 100,000 octets. Combien de blocs de données sont-ils nécessaires au total pour représenter ce fichier sur disque ?

2. Dans le système Unix, tous les renseignements concernant un fichier sont conservés dans des i-nœuds. Un fichier est identifié par le numéro de son i-nœud à l'intérieur de la table des i-nœuds. Un répertoire est un ensemble de couples composés chacun d'un nom de fichier relatif à ce répertoire et du numéro de l'i-nœud identifiant le fichier. La commande `ln` permet d'associer un autre nom (lien) à un fichier existant. Cela est très utile lorsque l'on veut pouvoir accéder à un fichier depuis plusieurs répertoires. Il est indispensable de connaître à chaque instant le nombre de liens sur un fichier car on ne peut supprimer un fichier que si tous les liens ont été supprimés. Cette information se trouve dans l'i-nœud du fichier.

   (a) Indiquer les opérations à effectuer par le processus (s'exécutant en mode système) lors de l'exécution d'une commande `ln`.

   (b) Expliquez pourquoi est-il nécessaire d'empêcher la manipulation du fichier existant par d'autres processus pendant l'exécution de cette commande ?

   (c) Supposons que le verrouillage du fichier est réalisé à l'aide d'un bit de verrouillage placé dans l'i-nœud du fichier. Peut-on avoir des situations d'interblocage, si le fichier existant est verrouillé pendant l'exécution de `ln` ? Proposer une solution pour traiter ce problème.

3. Soient n processus identiques accédant en parallèle à un fichier contenant un numéro à incrémenter. Écrire le programme de ces processus et utiliser un verrou externe pour synchroniser l'accès. Observez le résultat sans le verrou.