UE Programmation Unix

TP4 Gestion des Fichiers

Les appels système : gestion des fichiers

Un appel système permet à un processus utilisateur d'accéder à une ou plusieurs fonctions internes au noyau du system d'exploitation et de les exécuter.

Ainsi, un programme peut invoquer un ou plusieurs services du système d'exploitation.

Tous les programmes devront être développés avec passage de leurs éventuels paramètres à la fonction

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

- **Les valeurs de retour des appels aux primitives devront être testées et les messages d'erreurs affiches avec `perror`.**
- **Les messages d'erreurs à destination de l'utilisateur se feront sur le fichier standard des erreurs `stderr`.**

La fonction main()

Arguments de la ligne de commandes

- Langage C offre des mécanismes qui permettent d'intégrer parfaitement un programme C dans l'environnement hôte
  - environnement orienté ligne de commande (Unix, Linux)
- Programme C peut recevoir de la part de l'interpréteur de commandes qui a lancé son exécution, une liste d'arguments

⇔ ligne de commande qui a servi à lancer l'exécution du programme

- Liste composée
  - du nom du fichier binaire contenant le code exécutable du programme
  - des paramètres de la commande

Entête à inclure

```
#include <stdlib.h>  // <cstdlib> en C++
```

Fonction *atoi*

```
int atoi( const char * theString );
```

Cette fonction permet de transformer une chaîne de caractères, représentant une valeur entière, en une valeur numérique de type `int`. Le terme d'`atoi` est un acronyme signifiant : ASCII to integer.

⚠ **ATTENTION :** la fonction `atoi` retourne la valeur 0 si la chaîne de caractères ne contient pas une représentation de valeur numérique. Du coup, il n'est pas possible de distinguer la chaîne `"0"` d'une chaîne ne contenant pas un nombre entier. Si vous avez cette difficulté, veuillez préférer l'utilisation de la fonction `strtol` qui permet bien de distinguer les deux cas.

La fonction `main ()`

Un processus débute par l'exécution de la fonction `main()` du programme correspondant

Definition

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

- `argc`: nombre d'arguments de la ligne de commande y compris le nom du programme
- `argv[]`: tableau de pointeurs vers les arguments (paramètres) passés au programme lors de son lancement

A NOTER

- `argv[0]` pointe vers le nom du programme
- `argv[argc]` vaut `NULL`

MICHEL SOTO

- argc (argument count)
  - nombre de mots qui compose la ligne de commande (y compris le nom de la commande qui a servi à lancer l'exécution du programme)
- argv (argument vector)
  - tableau de chaînes de caractères contenant chacune un mot de la ligne de commande
  - argv[0] est le nom du programme exécutable

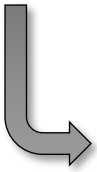
Exemple de cours

```c
#include <stdio.h>

int main (int argc, char *argv[]){
int i;
for (i=0;i<argc;i++){
    printf ("argv[%d]: %s\n",i, argv[i]);
}
return (0);
}// main
```

```
>./affich_param Bonjour à tous
argv[0]: ./affich_param
argv[1]: Bonjour
argv[2]: à
argv[3]: tous
```

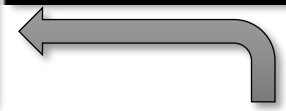
MICHEL SOTO

```
[ij04115@saphyr:~/unix_tpl]:ven. sept. 11$ nano affich_param.c
```



saphyr.ens.math-info.univ-paris5.fr - PuTTY

```
 GNU nano 2.5.3            Fichier : affich param.c
#include <stdio.h>

int main(int argc, char *argv[]){
        int i;
        for (i = 0 ; i < argc ; i++){
                printf("argv[%d]: %s\n", i, argv[i]);
        }
        return (0);
}//main
```



Spécifier le nom de l'exécutable avec l'option `-o`

```
ProgC > gcc -o toto premierProg.c
ProgC > ls
toto    premierProg.c
ProgC > ./toto
```

```
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$ gcc -o affich_param affich_param.c
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$ ls
affich_param  affich_param.c
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$
```

```
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$ ./affich_param Bonjour à tous
argv[0]: ./affich_param
argv[1]: Bonjour
argv[2]: à
argv[3]: tous
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$
```

Le fichier standard des erreurs
stderr

Structure *FILE ** et variables *stdin*, *stdout* et *stderr*

Entête à inclure

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

Structure *FILE ** et variables *stdin*, *stdout* et *stderr*

```
FILE * stdin;
FILE * stdout;
FILE * stderr;
```

La structure FILE permet de stocker les informations relatives à la gestion d'un flux de données. Néanmoins, il est très rare que vous ayez besoin d'accéder directement à ses attributs.

Effectivement, il existe un grand nombre de fonctions qui acceptent un paramètre basé sur cette structure pour déterminer ou contrôler divers aspects.

- **stdin** (Standard input) : ce flot correspond au flux standard d'entrée de l'application. Par défaut, ce flux est associé au clavier : vous pouvez donc acquérir facilement des données en provenance du clavier. Quelques fonctions utilisent implicitement ce flux (**scanf**, par exemple).
- **stdout** (Standard output): c'est le flux standard de sortie de votre application.
  Par défaut, ce flux est associé à la console d'où l'application à été lancée. Quelques fonctions utilisent implicitement ce flux (**printf**, par exemple).
- **stderr** (Standard error) : **ce dernier flux est associé à la sortie standard d'erreur de votre application.** Tout comme stdout, ce flux est normalement redirigé sur la console de l'application.

fprintf it is the same as printf,
except now you are also specifying the place to print to :

```
printf("%s", "Hello world\n"); // "Hello world" on stdout (using printf)
fprintf(stdout, "%s", "Hello world\n"); // "Hello world" on stdout (using fprintf)
fprintf(stderr, "%s", "Stack overflow!\n");//Error message on stderr (using fprintf)
```

Messages d'erreurs affiches avec perror

C Library - <stdio.h>

C library function - perror()

Description

The C library function **void perror(const char *str)** prints a descriptive error message to stderr. First the string **str** is printed, followed by a colon then a space.

Declaration

Following is the declaration for perror() function.

```
void perror(const char *str)
```

Parameters

- **str** – This is the C string containing a custom message to be printed before the error message itself.

Return Value

This function does not return any value.

```c
#include <stdio.h>

int main () {
   FILE *fp;

   /* first rename if there is any file */
   rename("file.txt", "newfile.txt");

   /* now let's try to open same file */
   fp = fopen("file.txt", "r");
   if( fp == NULL ) {
      perror("Error: ");
      return(-1);
   }
   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will produce the following result because we are trying to open a file which does not exist –

```
Error: : No such file or directory
```

1 Création de fichiers
open( )

Les appels système de base

- Ouverture

```
#include <unistd.h>
   int open(const char *pathname, int flags [, mode_t mode]);
```

  - Retourne un descripteur local de fichier
  - `flags` : `O_RDONLY, O_WRONLY, O_RDWR | O_CREAT, O_EXCL, O_APPEND, O_TRUNC, O_NONBLOCK, ...`
  - `mode` : permissions si un nouveau fichier est créé (modifiées par le `umask`).

```
#include <fcntl.h>

/* Not technically required, but needed on some UNIX distributions */
#include <sys/types.h>
#include <sys/stat.h>
```

Function Definition

```
int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

Field	Description
const char *path	The relative or absolute path to the file that is to be opened.
int oflags	A bitwise 'or' separated list of values that determine the method in which the file is to be opened (whether it should be read only, read/write, whether it should be cleared when opened, etc). See a list of legal values for this field at the end.
mode_t mode	A bitwise 'or' separated list of values that determine the permissions of the file if it is created. See a list of legal values at the end.
return value	Returns the file descriptor for the new file. The file descriptor returned is always the smallest integer greater than zero that is still available. If a negative value is returned, then there was an error opening the file.

Available Values for `oflag`

Value	Meaning
O_RDONLY	Open the file so that it is read only.
O_WRONLY	Open the file so that it is write only.
O_RDWR	Open the file so that it can be read from and written to.
O_APPEND	Append new information to the end of the file.
O_TRUNC	Initially clear all data from the file.
O_CREAT	If the file does not exist, create it. If the O_CREAT option is used, then you must include the third parameter.
O_EXCL	Combined with the O_CREAT option, it ensures that the caller *must* create the file. If the file already exists, the call will fail.

Available Values for `mode`

Value	Meaning
S_IRUSR	Set read rights for the owner to true.
S_IWUSR	Set write rights for the owner to true.
S_IXUSR	Set execution rights for the owner to true.
S_IRGRP	Set read rights for the group to true.
S_IWGRP	Set write rights for the group to true.
S_IXGRP	Set execution rights for the group to true.
S_IROTH	Set read rights for other users to true.
S_IWOTH	Set write rights for other users to true.
S_IXOTH	Set execution rights for other users to true.

Source : http://codewiki.wikidot.com/c:system-calls:open

a Ecrire un programme qui crée un fichier en lecture/écriture au travers de l'appel

`int open(const char *pathname, int flags, mode_t mode);`

Si le fichier existe déjà, une erreur doit être retournée

```c
#include <fcntl.h>

/* Not technically required, but needed on some UNIX distributions */
#include <sys/types.h>
#include <sys/stat.h>
```

Available Values for `oflag`

O_CREAT	If the file does not exist, create it. If the O_CREAT option is used, then you must include the third parameter.
O_EXCL	Combined with the O_CREAT option, it ensures that the caller *must* create the file. If the file already exists, the call will fail.

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

/* open() */
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>


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

    if(argc < 2) {
        fprintf(stderr, "Usage: %s file_path\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const char* file_path = argv[1];

    /* int open(const char* path, int oflags, mode_t mode); */
    int open_result = open(file_path, O_CREAT | O_EXCL, 0666);


    /* If a negative value is
    returned, then there was an
    error opening the file. */
    if(open_result < 0) {
        perror("Open function: ");
        exit(EXIT_FAILURE);
    }

    printf("Created successfully");

}
```

0666 is an octal number,
i.e. every one of the 6's corresponds to three permission bits

first three bits for owner permission,
next three bits for group permission
and next is for the world

the first digit - represents that is file or directory.
(0 - file, d - directory) here we used o means file

valeur octale	valeur binaire	droits
0	0 0 0	- - -
1	0 0 1	- - x
2	0 1 0	- w -
3	0 1 1	- w x
4	1 0 0	r - -
5	1 0 1	r - x
6	1 1 0	r w -
7	1 1 1	r w x

```
[ij04115@saphyr:~/unix_tp4]:sam. oct. 10$ nano question1.c
[ij04115@saphyr:~/unix_tp4]:sam. oct. 10$ gcc question1.c -o question1
[ij04115@saphyr:~/unix_tp4]:sam. oct. 10$ ls
question1  question1.c
[ij04115@saphyr:~/unix_tp4]:sam. oct. 10$ ./question1 newFile.txt
Created successfully[ij04115@saphyr:~/unix_tp4]:sam. oct. 10$ ls
newFile.txt  question1  question1.c
```

```
[ij04115@saphyr:~/unix_tp4]:sam. oct. 10$ ./question1 newFile.txt
Open function: : File exists
[ij04115@saphyr:~/unix_tp4]:sam. oct. 10$
```

```
[ij04115@saphyr:~/unix_tp4]:sam. oct. 10$ ls -l
total 12
-rw-r--r-- 1 ij04115 licence3in    0 oct.  10 05:48 newFile.txt
-rwxr-xr-x 1 ij04115 licence3in 7520 oct.  10 05:47 question1
-rw-r--r-- 1 ij04115 licence3in  598 oct.  10 05:47 question1.c
[ij04115@saphyr:~/unix_tp4]:sam. oct. 10$
```

```
mode_t umask(mode_t newmask);
```

General description

Changes the file creation mask of the process. *newmask* specifies new file-permission bits for the file creation mask of the process.

This mask restricts the setting of (or turns off) file-permission bits specified in the 'mode' argument used with all open(), creat(), mkdir(), and mkfifo() functions issued by the current process. File-permission bits set to 1 in the file creation mask are set to 0 in the file-permission bits of files that are created by the process.

For example, if a call to open() specifies a *mode* argument with file-permission bits, the file creation mask of the process affects the *mode* argument; bits that are 1 in the mask are set to 0 in the *mode* argument, and therefore in the mode of the created file.

Only the file-permission bits of the new mask are used.

La valeur typique par défaut pour le umask du processus est *S_IWGRP* | *S_IWOTH* (022 en octal). Dans le cas général où l'argument *mode* de **open**(2) vaut :

S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH

(0666 en octal) lors de la création d'un nouveau fichier, les permissions sur le fichier créé seront :

S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH

(car 0666 & ~022 = 0644 ; c'est-à-dire, rw-r--r--).

b Quel est le code d'erreur retourné lorsque le fichier existe déjà ?

return value	Returns the file descriptor for the new file. The file descriptor returned is always the smallest integer greater than zero that is still available. If a negative value is returned, then there was an error opening the file.

RETURN VALUE

open(), openat(), and creat() return the new file descriptor (a nonnegative integer), or -1 if an error occurred

ERRORS

open(), openat(), and creat() can fail with the following errors:

EEXIST *pathname* already exists and **O_CREAT** and **O_EXCL** were used.

Si l'argument mode spécifié est 755 (rwxr-xr-x), est-ce que le fichier est créé avec exactement ces droits ? Expliquer !

755 is an octal number,
i.e. every one of the numbers corresponds to three permission bits
first three bits for owner permission,
next three bits for group permission
and next is for the world

valeur octale	valeur binaire	droits
0	0 0 0	- - -
1	0 0 1	- - x
2	0 1 0	- w -
3	0 1 1	- w x
4	1 0 0	r - -
5	1 0 1	r - x
6	1 1 0	r w -
7	1 1 1	r w x

755 **&**~022 = 733 **c'est-à-dire** rwx-wx-wx

2 Caractéristique d'un fichier
start( ) , fstat( ) , lstat( )

Les fichiers

- Décrits par des attributs (contenus dans l'inode)
- Obtention des attributs d'un fichier

```
#include <sys/stat.h>
#include <unistd.h>
    int stat(const char *file_name, struct stat *buf);
    int fstat(int filedes, struct stat *buf);
    int lstat(const char *file_name, struct stat *buf);
```

struct stat

```
dev_t     st_dev;      /* device file resides on */
ino_t     st_ino;      /* the file serial number */
mode_t    st_mode;     /* file mode */
nlink_t   st_nlink;    /* number of hard links to the file */
uid_t     st_uid;      /* user ID of owner */
gid_t     st_gid;      /* group ID of owner */
dev_t     st_rdev;     /* the device identifier (special files only)*/
off_t     st_size;     /* total size of file, in bytes */
time_t    st_atime;    /* file last access time */
time_t    st_mtime;    /* file last modify time */
time_t    st_ctime;    /* file last status change time */
long      st_blksize;  /* preferred blocksize for file system I/O*/
long      st_blocks;   /* actual number of blocks allocated */
```

Michel Bonet

Description

These functions return information about a file. No permissions are required on the file itself, but-in the case of **stat**() and **lstat**() - execute (search) permission is required on all of the directories in *path* that lead to the file.

stat() stats the file pointed to by *path* and fills in *buf*.

lstat() is identical to **stat**(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fstat() is identical to **stat**(), except that the file to be stat-ed is specified by the file descriptor *fd*.

Return Value

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

Ecrire un programme qui récupère les caractéristiques de fichiers donnés, au travers des appels des fonctions

`int stat(); int fstat();` **et** `int lstat;`

Pour un fichier donné, afficher les caractéristiques suivantes :

- **Le numéro d'inode,**
- **La taille du fichier,**
- **La protection,**
- **La taille de bloc,**
- **Le nombre de liens physiques,**
- **Le nombre de blocs,**
- **L'ID du propriétaire,**
- **L'heure du dernier accès.**
- **L'ID du groupe,**

L'affichage d'une heure dans un format lisible peut être accompli en utilisant la fonction `ctime()`

C Library - <time.h>

Description

The C library function **char *ctime(const time_t *timer)** returns a string representing the localtime based on the argument **timer**.

The returned string has the following format: **Www Mmm dd hh:mm:ss yyyy**, where *Www* is the weekday, *Mmm* the month in letters, *dd* the day of the month, *hh*:*mm*:*ss* the time, and *yyyy* the year.

Declaration

Following is the declaration for ctime() function.

```
char *ctime(const time_t *timer)
```

Parameters

- **timer** – This is the pointer to a time_t object that contains a calendar time.

Return Value

This function returns a C string containing the date and time information in a human-readable format.

```c
#include <stdio.h> // fprintf(), perror(), printf()
#include <stdlib.h> // exit() , EXIT_FAILURE
#include <time.h> // ctime()

/* stat(), fstat(), lstat()  */
#include <sys/stat.h>
#include <unistd.h>

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

      if(argc < 2) {
         fprintf(stderr, "Usage: %s file \n", argv[0]);
         exit(EXIT_FAILURE);
      }

      struct stat buf;
      /* int start(const char* file_name, struct stat* buf); */
      int stat_result = stat(argv[1], &buf);

      if(stat_result < 0) {
            perror("Function stat : ");
            exit(EXIT_FAILURE);
      }


      printf("**** Caractéristiques du fichier **** \n");

      // ino_t st_ino;  (the file serial number) le numéro d inode
      printf("Numéro d'inode : %d \n", buf.st_ino);

      // mode_t st_mode; (file mode)
      printf("Protection : %lo (octal) \n", (unsigned long) buf.st_mode);

      // nlink_t st_nlink;  (number of hard links to the file)
      printf("Nombre de liens physiques : %d \n", buf.st_nlink);

      // uid_t  st_uid;  (user ID of owner)
      printf("ID du propriétaire : %d \n", buf.st_uid);

      // gid_t st_gid;  (group ID of owner)
      printf("ID du groupe : %d \n", buf.st_gid);

      // off_t st_size;  (total size of file, in bytes)
      printf("Taille du fichier : %d \n", buf.st_size);

      // long st_blksize;  (preferred blocksize for file system I/O)
      printf("Taille de bloc : %d \n", buf.st_blksize);

      // long  st_blocks;   ( actual number of blocks allocated )
      printf("Nombre de blocs : %d \n", buf.st_blocks);

      // time_t st_atime; (file last access time)
      // char* ctime(const time_t* timer)
      printf("Heure du dernier accès : %s \n", ctime(&buf.st_atime) );

}
```

```
[ij04115@saphyr:~/unix_tp4]:sam. oct. 10$ ls
newFile.txt  question1  question1.c  question2  question2.c
[ij04115@saphyr:~/unix_tp4]:sam. oct. 10$ ./question2 newFile.txt
**** Caractéristiques du fichier ****
Numéro d'inode : 25576814
Protection : 100644 (octal)
Nombre de liens physiques : 1
ID du propriétaire : 11978
ID du groupe : 3015
Taille du fichier : 0
Taille de bloc : 1048576
Nombre de blocs : 0
Heure du dernier accès : Sat Oct 10 05:48:25 2020

[ij04115@saphyr:~/unix_tp4]:sam. oct. 10$
```

3 Utilisation d'un fichier
open( ) , read( ) , write( )

Les appels système de base

- Ouverture

```
#include <unistd.h>
   int open(const char *pathname, int flags [, mode_t mode]);
```

  - Retourne un descripteur local de fichier
  - `flags` : `O_RDONLY, O_WRONLY, O_RDWR`
    `| O_CREAT, O_EXCL, O_APPEND, O_TRUNC, O_NONBLOCK, ...`
  - `mode` : permissions si un nouveau fichier est créé (modifiées par le `umask`).

- Lecture/Ecriture

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const char *buf, size_t count);
```

  - Retournent le nombre d'octets effectivement lus/écrits

- Fermeture

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

Plan Michel

a Ecrire un programme qui recopie un fichier source, `fichier_source`, dans un fichier destinataire, `fichier_destinataire`.

- **Le programme doit vérifier que le fichier source est un fichier régulier (utiliser la macro `S_ISREG(m)`, cf. `int stat()`).**
- **Le programme doit également vérifier qu'il n'existe pas déjà de fichier de même nom que le fichier destinataire**

Macros de détermination du type d'un fichier

- Macros définies dans `types.h`

`S_ISREG()`: fichier régulier
`S_ISDIR()`: fichier répertoire
`S_ISCHR()`: fichier spécial caractère
`S_ISBLK()`: fichier spécial bloc
`S_ISFIFO()`:FIFO
`S_ISLNK()`: lien symbolique
`S_ISSOCK()`:socket

L'argument de chacune des macros est le champs `st_mode` de la structure `stat`.

```
#include <sys/types.h>
…
main (int argc, char *argv[]) {
   struct stat buf; char *ptr;
   …
   if (stat(argv[1], &buf) != 0) { perror("Echec stat : "); exit(0); }
   …
   if (S_ISREG(buf.st_mode)) ptr = "regular";
   else if (S_ISDIR(buf.st_mode)) ptr = "directory";
   else if (S_ISCHR(buf.st_mode)) ptr = "character special";
   else if (S_ISBLK(buf.st_mode)) ptr = "bloc special";
   else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";
   else if (S_ISLNK(buf.st_mode)) ptr = "symbolic link";
   else if (S_ISSOCK(buf.st_mode)) ptr = "socket";
   else ptr = "** unknown mode **";

   printf("%s\n", ptr);
   exit(0);
}
```

Michel Buffo

```c
#include <stdio.h> // fprintf(), perror()
#include <stdlib.h> // exit(), atoi(), malloc()
/* open */
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h> // open() & stat() & S_ISREG

#include <unistd.h> // stat(), read(), write()

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

       if(argc < 4) {
              fprintf(stderr, "Usage : %s ", argv[0]);
              fprintf(stderr, "source_file destination_file buffer_size(octets) \n");
              exit (1);
       }

     struct stat buf;
     // int stat(const char* file_name, struct stat *buf);
     // On success, zero is returned. On error, -1 is returned.
       int stat_result = stat(argv[1], &buf);
       if(stat_result < 0) {
              perror("Function stat() : ");
              exit (1);
       }

      /* Vérifier que le fichier source est un fichier régulier */
       // int S_ISREG(mode_t mode)
       // This macro returns non-zero if the file is a regular file.
     // L'argument de cette macro = le champs st_mode de la structure buf
     // mode_t  st_mode; ( file mode )
       int isreg_result = S_ISREG(buf.st_mode);
       if(isreg_result == 0) {
              fprintf(stderr, "%s is not a regular file \n", argv[1]);
              exit (1);
       }

       const char* source_file = argv[1];
       // int open(const* pathname, int flags [, mode_t mode] );
     // O_RDONLY Open the file so that it is read only.
     // Negative value is returned = was an error opening the file.
       int open_source_result = open(source_file,O_RDONLY);
       if(open_source_result < 0) {
              perror("Function open() - source file : ");
              exit (1);
       }

       const char* destination_file = argv[2];
       // int open(const* pathname, int flags [, mode_t mode] );
       // Vérifier qu'il n'existe pas déjà de fichier de même nom
       int open_destination_result = open(destination_file, O_WRONLY|O_CREAT|O_EXCL, 0644);

        // O_WRONLY - Open the file so that it is write only.
        // O_CREAT - If the file does not exist, create it.
       // O_EXCL - If the file already exists, the call will fail.
       //  O_CREAT option is used, then you must include the third parameter.
       // 644 - 022 = 622   ->   rw--w--w-



       if(open_destination_result < 0) {
              perror("Function open() - destination file : ");
              exit (1);
       }
```

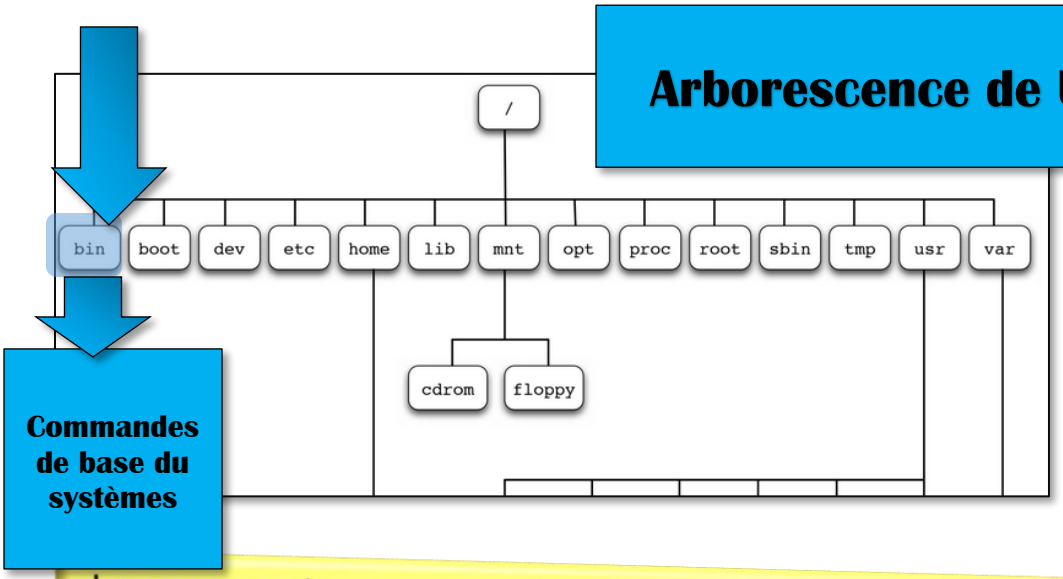
```
   /*  ssize_t read(int fd, void *buf, size_t count);     */
   // int fd - The file descriptor of where to read the input.
  // const void *buf - A character array where the read content will be stored.
   // size_t count - The number of bytes to read before truncating the data.
  // If the data to be read is smaller than count, all data is saved in the buffer.
  // Returns the number of bytes that were read. value is negative = error.

  int buffer_size = atoi(argv[3]);
  char* buffer = malloc(buffer_size);
  if(!buffer) {
     perror("Function malloc() : ");
     exit (1);
  }

  int numberBytesRead = read(open_source_result, buffer, buffer_size);
  int i = 0;
  int totalBytesWrite = 0;

  while( numberBytesRead > 0 ) {

      // ssize_t write(int fd, const char *buf, size_t count);
      // int fd = The file descriptor of where to write the output
      // const char *buf = pointer to a buffer of at least count bytes
      // size_t count = The number of bytes to write.
      // Returns the number of bytes that were written
     //  If value is negative, then the system call returned an error.
     int numberBytesWrite = write(open_destination_result, buffer, numberBytesRead);

     if(numberBytesWrite < 0) {
        perror("Function write() : ");
        exit (1);
     }

     totalBytesWrite += numberBytesWrite;
     i++;
     numberBytesRead = read(open_source_result, buffer, buffer_size);
  } // while

  if(numberBytesRead < 0) {
     perror("Function read() : ");
     exit (1);
  }

  printf("totalBytesWrite (octets) = %d \n", totalBytesWrite);
  printf("Number of iterations = %d \n", i);

} // main
```

```
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$ nano question3.c
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$ gcc question3.c -o question3
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$ ls
newFile.txt  question1  question1.c  question2  question2.c  question3  question3.c
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$ ./question3
Usage : ./question3 source_file destination_file buffer_size(octets)
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$ ./question3 newFile.txt destinationFile.txt 17
totalBytesWrite (octets) = 17
Number of iterations = 1
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$
```

```
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$ ls
destinationFile.txt  question1.c  question3
newFile.txt          question2    question3.c
question1            question2.c
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$ cat destinationFile.txt
11111111100000000
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$
```

b **Quels sont les temps d'exécution (utiliser `/bin/time`) respectifs si la taille du buffer utilisé dans la fonction `read()` est de 1024 octets puis un octet ? Expliquer !**

Arborescence de UNIX



Commandes de base du systèmes

La commande `time` permet de lancer une commande quelconque avec affichage, à la fin du processus l'exécutant, d'informations relatives à son temps d'exécution.

```
[ev00000@saphyr ~]$ time ps > fps

real    0m0.017s
user    0m0.004s
sys     0m0.007s
```

save a record of the time command information in a file called ' fps '

Use the cat command to display output on screen

```
$ cat fps
```

Purpose Run command/programs or script and summarize system resource usage

The time command show the following information on screen (on the standard error output, by default) about resources used by command :

1. real time - correspond au temps réel de la tâche.
2. user time - correspond au temps utilisateur, c'est à dire le temps CPU utilisé par le programme utilisateur
3. sys time - définit le temps système, cela correspond au temps utilisé par le système pour gérer l'exécution de la tâche.

Le temps CPU de la tâche = temps user + temps sys

A **central processing unit (CPU)**, *also called a central processor, main processor or just processor, is the electronic circuitry within a computer that executes instructions that make up a computer program.*

Bit (abréviation de BInary digiT)	Plus petite quantité d'information **Prend que deux valeurs 0 ou 1** ("faux" ou "vrai")
Octet byte en anglais Prononcer baït	**Composé de 8 bits** (Exemple : 0 1 1 0 0 0 1 0) **Les octets sont utiles pour exprimer des quantités de données.** **Un octet peut représenter** $2^8 = 256$ **informations différentes**
1 Kilooctet (Ko)	$2^{10} = 1024$ **octets**

```
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$ time ./question3 question2.c  destinationFile.txt 1
totalBytesWrite (octets) = 1543
Number of iterations = 1543

real    0m0.019s
user    0m0.000s
sys     0m0.004s
```

```
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$ time ./question3 question2.c  destinationFile.txt 1024
Function open() - destination file : : File exists

real    0m0.005s
user    0m0.000s
sys     0m0.000s
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$ rm destinationFile.txt
rm : supprimer 'destinationFile.txt' du type fichier ? o
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$ time ./question3 question2.c  destinationFile.txt 1024
totalBytesWrite (octets) = 1543
Number of iterations = 2

real    0m0.009s
user    0m0.000s
sys     0m0.000s
[ij04115@saphyr:~/unix_tp4]:dim. oct. 11$
```

La taille du buffer augmente -> Le temps d'exécution diminue.

La taille du buffer augmente -> le nombre d'appels à read et write diminue

-> diminuer le temps d'exécution

* **Le nombre de lecture/écriture physiques impact également le temp d'exécution.**

Le nombre de lecture/écriture physiques dépendant de la taille du bloc sur le disque.

4 Duplication des descripteurs de fichier dup( ) , dup2( ) , write( )

a Ecrire un programme qui redirige la sortie d'erreur standard vers un fichier, `fichier_erreur`, préalablement créé.

à l'avance

C'est à dire, toute écriture de la forme `write(2, ...)` doit se faire dans le fichier `fichier_erreur`.

Le descripteur de valeur 2 étant au départ celui de la sortie d'erreur standard.

Tables en mémoire exploitées par le noyau

u_ofile

- Descripteurs standards dans la u_ofile
  - 0 (stdin) : flux correspondant à l'entrée standard
  - 1 (stdout) : flux correspondant à la sortie standard
  - 2 (stderr) : flux correspondant à la sortie d'erreur standard

ssize_t write(int fd, const char *buf, size_t count);

```
int fd = The file descriptor of where to write the output
const char *buf = pointer to a buffer of at least count bytes
size_t count = The number of bytes to write.
Returns the number of bytes that were written
If value is negative, then the system call returned an error.
```

- Duplication des descripteurs de fichiers

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

  - Utilisés pour rediriger les entrées/sorties standards vers des fichiers ou vers des tubes.

```c
#include <stdio.h> // fprintf(), perror()
#include <unistd.h> // close(), dup()
#include <stdlib.h> // exit(), EXIT_FAILURE
#include <fcntl.h> // open()
#include <sys/types.h> // open()
#include <sys/stat.h> // open()

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

       int result_open, result_close, result_dup;

       if(argc < 2){
         fprintf(stderr, "Utilisation : %s nom_fichier \n", argv[0]);
           exit(EXIT_FAILURE);
     }

       const char* f_erreur = argv[1];

    // int open(const char* pathname, int flags [, mode_t mode] );
    result_open = open(f_erreur, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    /* O_WRONLY - Open the file so that it is write only.
     * O_CREAT - If the file does not exist, create it.
     * O_TRUNC - Initially clear all data from the file.
     * O_CREAT option is used -> must include the third parameter.
       * 644 - 022 = 622  ->  rw--w--w-
       */

       if(result_open < 0){ // negative value is returned ->  error
              perror("Erreur fonction open() : ");
              exit(EXIT_FAILURE);
     }

     // int close(int fildes); fildes = The file descriptor to be closed.
       result_close = close( 2 ); // 2 = stderr file
       if(result_close != 0) {// Retuns a 0 upon success, and a -1 upon failure
              perror("Error Closing the standard error file stderr");
              exit(EXIT_FAILURE);
     }

      // int dup(int fildes);
      // fildes = The file descriptor that you are attempting to create an alias for.
       result_dup = dup( result_open ); // open() returns the file descriptor
       if(result_dup < 0){ // negative value is returned ->  error
              perror("Erreur fonction dup() : ");
              exit(EXIT_FAILURE);
     }

     fprintf(stderr,"This message will appear in the new error file \n");
     return 0;
}
```

dup (C System Call)

oodup is a system call similar to dup2 in that it creates an alias for the provided file descriptor. dup always uses the smallest available file descriptor. Thus, if we called dup first thing in our program, then you could write to standard output by using file descriptor 3 (dup uses 3 because 0, 1, and 2 are already taken by default). You can determine the value of the new file descriptor by saving the return value from dup

dup always use the 1st free entry of the u_ofile. **Dans cette exo,**

Here it will be the ex-entry of the standard error file thanks to the close (2) which preceded it

Tables en mémoire exploitées par le noyau
u_ofile

- **u_ofile:** table des descripteurs de fichiers d'un processus
  - Chaque processus possède la sienne
    - Vision PAR PROCESSUS des fichiers ouverts
- Lors de l'ouverture d'un fichier, le noyau lui associe une entrée dans cette table
- Se trouve dans le structure U associée à chaque processus

```
struct user{
        struct inode *u_cdir; <-- pointe sur le répertoire courant
        struct inode *u_rdir; <-- pointe sur le répertoire racine
                              du volume
        short u_cmask <-- protection
        struct *u_ofile[NOFILE] <-- NOFILE: nombre de fichiers
                                    maximun ouverts par un processus
        ....
       }
```

7

Tables en mémoire exploitées par le noyau
u_ofile

- Descripteurs standards dans la u_ofile
  - 0 (stdin) : flux correspondant à l'entrée standard
  - 1 (stdout) : flux correspondant à la sortie standard
  - 2 (stderr) : flux correspondant à la sortie d'erreur standard
- Exemple

```
...
fprintf(stderr,"erreur numero %d", errno);
…
```

- Chaque entrée associée à un fichier ouvert pointe vers une entrée de la table des ouvertures de fichiers: file table

8

```
[ij04115@saphyr:~/unix_tp4]:dim. oct. 25$ gcc question4.c -o question4
[ij04115@saphyr:~/unix_tp4]:dim. oct. 25$ ls
destinationFile.txt  question1    question2    question3    question4
newFile.txt          question1.c  question2.c  question3.c  question4.c
[ij04115@saphyr:~/unix_tp4]:dim. oct. 25$ nano fichier_erreur.txt
[ij04115@saphyr:~/unix_tp4]:dim. oct. 25$ ls
destinationFile.txt  question1    question2.c  question4
fichier_erreur.txt   question1.c  question3    question4.c
newFile.txt          question2    question3.c
[ij04115@saphyr:~/unix_tp4]:dim. oct. 25$ ./question4
Utilisation : ./question4 nom_fichier
[ij04115@saphyr:~/unix_tp4]:dim. oct. 25$ ./question4 fichier_erreur.txt
[ij04115@saphyr:~/unix_tp4]:dim. oct. 25$ cat fichier_erreur.txt
This message will appear in the new error file
[ij04115@saphyr:~/unix_tp4]:dim. oct. 25$
```

dup2()

The dup2() system call is similar to dup() but the basic difference between them is that instead of using the lowest-numbered unused file descriptor, it uses the descriptor number specified by the user.

Syntax:

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

oldfd: old file descriptor

newfd new file descriptor which is used by dup2() to create a copy.

b **Quelle est la propriété de la fonction `dup()` qui est exploitée pour ainsi rediriger les E/S standards ?**

dup always use the 1st free entry of the u_ofile.

Dans cette exo,

Here it will be the ex-entry of the standard error file thanks to the close (2) which preceded it

dup2()

The steps of closing and reusing the file descriptor *newfd* are performed *atomically*. This is important, because trying to implement equivalent functionality using close(2) and dup() would be subject to race conditions, whereby *newfd* might be reused between the two steps. Such reuse could happen because the main program is interrupted by a signal handler that allocates a file descriptor, or because a parallel thread allocates a file descriptor.

Donc c mieux d'utiliser dup2()

C Modifier le code du mini shell afin qu'il prenne en charge la redirection des entrées (<) et des sorties (>).

Redirection des sorties vers un fichier

Cette redirection permet d'écrire dans un fichier le résultat d'une commande au lieu de l'afficher à l'écran.

- Sortie standard (descripteur de fichier 1)

Simple redirection

```
$ commande > fichier
ou
$ commande 1> fichier
```

Si le fichier n'existe pas, il est automatiquement créé. S'il existe déjà, il est tout simplement écrasé.

Récupérer le résultat de la commande ls dans un fichier "liste"

```
$ ls > liste
$ cat liste
1coucou
cOucou
Coucou
```

Redirection de l'entrée standard

La redirection de l'entrée standard concerne toutes les commandes attendant une saisie de la part de l'utilisateur sur le descripteur 0 (saisie écran).

```
$ mail toto
>Coucou
>Comment vas tu ?
>^d (équivalent de CTRL+d)
$
```

La commande **mail** lit sur l'entrée standard toutes les données saisies à l'écran. La saisie se termine par la fonction CTRL+d.
Ensuite, la commande mail envoie ces données dans un message à l'utilisateur indiqué (toto).

Il est tout à fait possible d'écrire le contenu du message dans un fichier et de "l'injecter" à la commande **mail** sur son descripteur 0.

```
$ commande 0< fichier
ou
$ commande < fichier
```

Le shell

Avant la modification

```c
#include <stdio.h> // printf(), fgets(), perror()
#include <string.h> // strcpy(), strlen(), strrchr(), strtok()
#include <stdlib.h> // malloc()
#include <sys/types.h> // wait(), fork()
#include <unistd.h> // fork(), exevp()
#include <sys/wait.h> // wait()

#define MAX_NB_ARGS 40
#define MAX_LONG_CMD 250
int main(int argc, char* argv[]) {

    char cmd[MAX_LONG_CMD]; // La commande
    char* argv_execvp[MAX_NB_ARGS]; // Les mot de la ligne de commande
    int bg, i, status, fork_result;
    char* ptrBg;
    char* token;

       while(1) { // boucle infinie
           printf("----->");
           fgets(cmd, MAX_LONG_CMD - 1 , stdin); // la commande + \n
           i = 0;
           while(cmd[i] != '\n') //on remplace \n par \0
               i++;
           cmd[i] = '\0';

         //on remplace & par \0 et on retient la présence sa dans variable bg
         ptrBg = strrchr(cmd, '&');
         bg = 0;
         if(ptrBg!= NULL){
            *ptrBg = '\0'; // on remplace & par \0
            bg = 1;
         }

         token = strtok(cmd, " "); //get the first token
         for( i = 0 ; token != NULL ; i++) {  // walk through other tokens

           argv_execvp[i] = (char*) malloc( strlen(token) + 1 );
           strcpy(argv_execvp[i], token);
           token = strtok(NULL, " ");

         }// for
         argv_execvp[i] = NULL; // Le tableau se termine par NULL

         if( i > 0) { // ligne de commande PAS vide
            fork_result = fork();
            if(fork_result == 0) { // CODE DU FILS
               printf("\n");
               execvp(argv_execvp[0], argv_execvp);
               perror("Erreur function execvp() : ");
            }
            else // CODE DU PERE
            {
               if(bg == 0) // si la commande est PAS executee en arrière plan
                 wait(&status); //On attend le processus
            }
         }
    } // while
} // main
```

```c
#include <stdio.h> // printf(), fgets(), perror()
#include <string.h> // strcpy(), strlen(), strrchr(), strtok()
#include <stdlib.h> // malloc()
#include <sys/types.h> // wait(), fork(), open()
#include <unistd.h> // fork(), exevp(), dup2(), close()
#include <sys/wait.h> // wait()
#include <fcntl.h> // open()
#include <sys/stat.h> // open()

#define MAX_NB_ARGS 40
#define MAX_LONG_CMD 250
int main(int argc, char* argv[]) {

    char cmd[MAX_LONG_CMD]; // La commande
    char* argv_execvp[MAX_NB_ARGS]; // Les mot de la ligne de commande
    int bg, i, status, fork_result, new_file_descriptor, result_open;
    char *ptrBg, *token, *ptr_new_file_name;

        while(1) { // boucle infinie
            printf("----->");
           fgets(cmd, MAX_LONG_CMD - 1 , stdin); // la commande + \n
            i = 0;
            while(cmd[i] != '\n') //on remplace \n par \0
                i++;
            cmd[i] = '\0';

        //on remplace & par \0 et on retient sa présence sa dans variable bg
        ptrBg = strrchr(cmd, '&');
        bg = 0;
        if(ptrBg!= NULL){
           *ptrBg = '\0'; // on remplace & par \0
           bg = 1;
        }

        new_file_descriptor = -1;
        //on remplace > par \0 et on retient sa présence
        ptr_new_file_name = strrchr(cmd, '>');
        if(ptr_new_file_name != NULL){
           *ptr_new_file_name = '\0'; // on remplace > par \0
           ptr_new_file_name++; // pointer to the new stdout file name
           new_file_descriptor = 1;
           bg = 1;
        }
        else { //on remplace < par \0 et on retient sa présence
                ptr_new_file_name = strrchr(cmd, '<');
                if(ptr_new_file_name != NULL){
                        *ptr_new_file_name = '\0'; // on remplace < par \0
                        ptr_new_file_name++; // pointer to the new stdin file name
                       new_file_descriptor = 0;
                }
        }

        if(ptr_new_file_name != NULL){
                result_open = open(ptr_new_file_name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
                /* O_WRONLY - Open the file so that it is write only.
                * O_CREAT - If the file does not exist, create it.
                * O_TRUNC - Initially clear all data from the file.
                * O_CREAT option is used -> must include the third parameter.
                * 644 - 022 = 622  ->  rw--w--w-
                */
            if(result_open < 0){ // negative value is returned ->  error
                    perror("Erreur fonction open() : ");
                    exit(EXIT_FAILURE);
            }

            int result_dup2 = dup2( result_open,new_file_descriptor);
            if(result_dup2 < 0){ // negative value is returned ->  error
                    perror("Erreur fonction dup2() : ");
                    exit(EXIT_FAILURE);
            }
        }
```

```
        token = strtok(cmd, " "); //get the first token
        for( i = 0 ; token != NULL ; i++) {  // walk through other tokens
          argv_execvp[i] = (char*) malloc( strlen(token) + 1 );
          strcpy(argv_execvp[i], token);
          token = strtok(NULL, " ");

        }// for
        argv_execvp[i] = NULL; // Le tableau se termine par NULL

        if( i > 0) { // ligne de commande PAS vide
             fork_result = fork();
           if(fork_result == 0) { // CODE DU FILS
              printf("\n");
              execvp(argv_execvp[0], argv_execvp);
              perror("Erreur function execvp() : ");
           }
           else // CODE DU PERE
           {
              if(bg == 0) // si la commande est PAS executee en arrière plan
                wait(&status); //On attend le processus

              if(new_file_descriptor != -1) {
                int result_close = close( result_open );
                if(result_close < 0){ // -1 is returned ->  error
                        perror("Erreur fonction close() : ");
                         exit(EXIT_FAILURE);
                }
              }
                }// else

        }// if(i>0)
    } // while
} // main
```

```
[ij04115@saphyr:~/unix_tp4]:lun. oct. 26$ nano mini_shell.c
[ij04115@saphyr:~/unix_tp4]:lun. oct. 26$ gcc mini_shell.c -o mini_shell
[ij04115@saphyr:~/unix_tp4]:lun. oct. 26$ ./mini_shell
----->ls>ls_result.txt



Erreur de segmentation (core dumped)
[ij04115@saphyr:~/unix_tp4]:lun. oct. 26$ cat ls_result.txt
----->
 a.txt
destinationFile.txt
fichier_erreur.txt
file
ls
 ls
 ls_result.txt
ls_result.txt
mini_shell
mini_shell.c
newFile.txt
question1
question1.c
question2
question2.c
question3
question3.c
question4
question4.c
t
 t
 testFileName
 test.txt
 t.txt
t.txt
----->----->----->[ij04115@saphyr:~/unix_tp4]:lun. oct. 26$
```

PBM !

5 Verrouillage des fichiers
fcntl( )

Modification des caractéristiques d'un fichier

La primitive `fcntl`

The `fcntl` function can change the properties of a file that is already open.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* int arg */ );

                        Returns: depends on cmd if OK (see following), –1 on error
```

In the example, the third argument is always an integer, corresponding to the comment in the function prototype just shown.

When we describe record locking the third argument becomes a pointer to a structure.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* struct flock *flockptr */ );

                        Returns: depends on cmd if OK (see following), –1 on error
```

La primitive fcntl permet, en fonction de la valeur du paramètre cmd, de réaliser un certain nombre d'opérations sur le descripteur de fichier fd.

Valeur de cmd	opération	Valeur retour
F_DUPFD	duplication de descripteur	nouveau descripteur
F_GETFD, F_SETFD	modification des attributs du descripteur	état de l'attribut
F_GETFL, F_SETFL	consultation/modification du mode d'ouverture	état des attributs
F_GETLK, F_SETLK, F_SETLKW	**manipulation des verrous**	
F_GETOWN, F_SETOWN (BSD)	modification du propriétaire d'une socket	propriétaire du fichier

Record Locking

What happens when two people edit the same file at the same time? In most UNIX systems, the final state of the file corresponds to the last process that wrote the file. In some applications, however, such as a database system, a process needs to be certain that it alone is writing to a file. To provide this capability for processes that need it, commercial UNIX systems provide record locking. (In Chapter 20, we develop a database library that uses record locking.)

Record locking is the term normally used to describe the ability of a process to prevent other processes from modifying a region of a file while the first process is reading or modifying that portion of the file. Under the UNIX System, "record" is a misnomer; the UNIX kernel does not have a notion of records in a file. A better term is *byte-range locking*, given that it is a range of a file (possibly the entire file) that is locked.

Le verrouillage - Caractéristiques

- Le verrouillage s'applique au fichier (et non au descripteur)
- Un verrou est associé à un processus et à un fichier
  - seul le propriétaire du verrou peut le modifier ou le supprimer
  - lorsqu'un processus se termine, tous les verrous qu'il détient sont supprimés
  - chaque fois qu'un descripteur est fermé par un processus, tous les verrous sur le fichier référencé par le descripteur pour le processus donné sont supprimés

Verrouillage par la primitive `fcntl`

- Gérer les accès concurrents (sur des portions de fichier)

  Le troisième paramètre de la primitive est un pointeur sur une `struct flock`.

```
struct flock {  /* défini dans fcntl.h */
   short l_type; /* type de verrou : F_RDLCK partagé
                                       F_WRLCK exclusif
                                       F_UNLCK déverrouillage
                 */
   short l_whence; /* position (idem lseek) */
   short l_start;  /* position relative de début : l_whence */
   short l_len;    /* nombre d'octets verrouillés, si 0 →
                      jusqu'à fin fichier */
   int l_pid;      /* PID du processus auquel appartient le
                      verrou. Retourné par F_GETLK */
}
```

- Les situations d'interblocage sont détectées.

```c
#include <stdio.h> // fprintf(), perror()
#include <stdlib.h> // exit(), EXIT_FAILURE, atoi
#include <fcntl.h> // open(), fcntl(), struct flock, fcntl()
#include <sys/types.h> // open()
#include <sys/stat.h> // open()
#include <string.h> // strcmp()

int main(int argc, char *argv[]) {
       int result_open, result_fcntl;
       struct flock the_flock;

       if(argc < 5) {

fprintf(stderr, "Utilisation: %s nom_fichier type_verrou début_zone longueur_zone\n", argv[0]);
           exit(EXIT_FAILURE);
      }

       // the_flock.l_type = The type of lock desired

      // int strcmp (const char* str1, const char* str2);
      // compares two strings and returns 0 if both strings are identical.
      if( strcmp(argv[2], "exclusif") == 0) {
             the_flock.l_type = F_WRLCK; // an exclusive write lock
      } else {
             if( strcmp(argv[2], "partage") == 0) {
                    the_flock.l_type = F_RDLCK; // a shared read lock
             } else {
                    fprintf(stderr,"type_verrou doit etre exclusif ou partage\n");
                    exit(EXIT_FAILURE);
             }
      }

       // the_flock.l_whence = The starting byte offset of the region being locked or unlocked
       the_flock.l_whence = SEEK_SET; // specify the  beginning  of  the  file
       the_flock.l_start = atoi( argv[3] ); // position relative de début : l_whence
       the_flock.l_len =   atoi(argv[4]);//nombre d'octets verrouillés,0=jusque fin file


    // int open(const char* pathname, int flags [, mode_t mode] );
    // O_RDWR - Open the file so that it can be read from and written to..
    result_open = open(argv[1], O_RDWR);
    if(result_open < 0){ // negative value is returned ->  error
             perror("Erreur fonction open() : ");
              exit(EXIT_FAILURE);
    }


     // int fcntl(int fd, int cmd, struct flock flockptr);
     result_fcntl = fcntl(result_open, F_SETLK, &the_flock);
     if(result_fcntl == -1){ // -1 is returned ->  error
             perror("Erreur fonction fcntl() : ");
              exit(EXIT_FAILURE);
    }

    printf("Succès verrou \n");
    while(1){ }

    return 0;

    } // main()
```

`fcntl` Record Locking

Let's repeat the prototype for the `fcntl` function from Section 3.14.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* struct flock *flockptr */ );
```

Returns: depends on *cmd* if OK (see following), –1 on error

For record locking, *cmd* is `F_GETLK`, `F_SETLK`, or `F_SETLKW`. The third argument (which we'll call *flockptr*) is a pointer to an `flock` structure.

```
struct flock {
  short  l_type;    /* F_RDLCK, F_WRLCK, or F_UNLCK */
  short  l_whence;  /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_start;   /* offset in bytes, relative to l_whence */
  off_t  l_len;     /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;     /* returned with F_GETLK */
};
```

This structure describes

- The type of lock desired: `F_RDLCK` (a shared read lock), `F_WRLCK` (an exclusive write lock), or `F_UNLCK` (unlocking a region)
- The starting byte offset of the region being locked or unlocked (`l_start` and `l_whence`)
- The size of the region in bytes (`l_len`)
- The ID (`l_pid`) of the process holding the lock that can block the current process (returned by `F_GETLK` only)

Numerous rules apply to the specification of the region to be locked or unlocked.

- The two elements that specify the starting offset of the region are similar to the last two arguments of the `lseek` function (Section 3.6). Indeed, the `l_whence` member is specified as `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`.
- Locks can start and extend beyond the current end of file, but cannot start or extend before the beginning of the file.
- If `l_len` is 0, it means that the lock extends to the largest possible offset of the file. This allows us to lock a region starting anywhere in the file, up through and including any data that is appended to the file. (We don't have to try to guess how many bytes might be appended to the file.)
- To lock the entire file, we set `l_start` and `l_whence` to point to the beginning of the file and specify a length (`l_len`) of 0. (There are several ways to specify the beginning of the file, but most applications specify `l_start` as 0 and `l_whence` as `SEEK_SET`.)

We previously mentioned two types of locks: a shared read lock (`l_type` of `F_RDLCK`) and an exclusive write lock (`F_WRLCK`). The basic rule is that any number of processes can have a shared read lock on a given byte, but only one process can have an exclusive write lock on a given byte. Furthermore, if there are one or more read locks on a byte, there can't be any write locks on that byte; if there is an exclusive write lock on a byte, there can't be any read locks on that byte. We show this compatibility rule in Figure 14.3.

		Request for	
		read lock	write lock
Region currently has	no locks	OK	OK
	one or more read locks	OK	denied
	one write lock	denied	denied

Figure 14.3 Compatibility between different lock types

The compatibility rule applies to lock requests made from different processes, not to multiple lock requests made by a single process. If a process has an existing lock on a range of a file, a subsequent attempt to place a lock on the same range by the same process will replace the existing lock with the new one. Thus, if a process has a write lock on bytes 16–32 of a file and then tries to place a read lock on bytes 16–32, the request will succeed, and the write lock will be replaced by a read lock.

To obtain a read lock, the descriptor must be open for reading; to obtain a write lock, the descriptor must be open for writing.

We can now describe the three commands for the `fcntl` function.

`F_GETLK` Determine whether the lock described by *flockptr* is blocked by some other lock. If a lock exists that would prevent ours from being created, the information on that existing lock overwrites the information pointed to by *flockptr*. If no lock exists that would prevent ours from being created, the structure pointed to by *flockptr* is left unchanged except for the `l_type` member, which is set to `F_UNLCK`.

`F_SETLK` Set the lock described by *flockptr*. If we are trying to obtain a read lock (`l_type` of `F_RDLCK`) or a write lock (`l_type` of `F_WRLCK`) and the compatibility rule prevents the system from giving us the lock (Figure 14.3), `fcntl` returns immediately with `errno` set to either `EACCES` or `EAGAIN`.

> Although POSIX allows an implementation to return either error code, all four implementations described in this text return `EAGAIN` if the locking request cannot be satisfied.

This command is also used to clear the lock described by *flockptr* (`l_type` of `F_UNLCK`).

`F_SETLKW` This command is a blocking version of `F_SETLK`. (The `W` in the command name means *wait*.) If the requested read lock or write lock cannot be granted because another process currently has some part of the requested region locked, the calling process is put to sleep. The process wakes up either when the lock becomes available or when interrupted by a signal.

Be aware that testing for a lock with `F_GETLK` and then trying to obtain that lock with `F_SETLK` or `F_SETLKW` is not an atomic operation. We have no guarantee that, between the two `fcntl` calls, some other process won't come in and obtain the same lock. If we don't want to block while waiting for a lock to become available to us, we must handle the possible error returns from `F_SETLK`.

```
login as: ij04115
ij04115@saphyr.ens.math-info.univ-paris5.fr's password:
Last login: Thu Nov 12 11:50:13 2020 from 37.172.46.219
[ij04115@saphyr:~]:jeu. nov. 12$ cd unix_tp4
[ij04115@saphyr:~/unix_tp4]:jeu. nov. 12$ ./question5 fichier_erreur.txt exclusif 0 5&
[1] 26623
[ij04115@saphyr:~/unix_tp4]:jeu. nov. 12$ Succès verrou
ps
  PID TTY          TIME CMD
26612 pts/12   00:00:00 bash2
26623 pts/12   00:00:11 question5
26630 pts/12   00:00:00 ps
[ij04115@saphyr:~/unix_tp4]:jeu. nov. 12$ ./question5 fichier_erreur.txt exclusif 0 5
Erreur fonction fcntl() : : Resource temporarily unavailable
[ij04115@saphyr:~/unix_tp4]:jeu. nov. 12$
```

6 Déplacement de la tête de lecture/écriture des fichiers

lseek( )

Bit
(abréviation de BInary digiT)

Octet
byte en anglais
Prononcer baït

`lseek` Function

- → **Every open file has an associated "current file offset,"**
- → **Normally a non-negative integer that measures the number of bytes from the beginning of the file.**
- → **We describe some exceptions to the "non-negative" qualifier later in this section.**
- → **Read and write operations normally start at the current file offset and cause the offset to be incremented by the number of bytes read or written.**
- → **By default, this offset is initialized to 0 when a file is opened, unless the O_APPEND option is specified**

O_APPEND	Append to the end of file on each write.

An open file's offset can be set explicitly by calling `lseek`

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
                                    Returns: new file offset if OK, –1 on error
```

The interpretation of the offset depends on the value of the whence argument.

value of the whence argument	interpretation of the offset
SEEK_SET	the file's offset is set to *offset* bytes from the beginning of the file
SEEK_CUR	the file's offset is set to its current value plus the *offset*. The *offset* can be positive or negative.
SEEK_END	the file's offset is set to the size of the file plus the *offset*. The *offset* can be positive or negative.

Manipulation de l'*offset*

- Déplacement du pointeur courant d'un fichier régulier

```
#include <unistd.h>
   off_t lseek(int fildes, off_t offset, int whence);
```

- whence : SEEK_SET (0) par rapport au début du fichier
  SEEK_CUR (1) par rapport à la position courante
  SEEK_END (2) par rapport à la fin du fichier
- offset : position par rapport à whence

a Ecrire un programme qui crée un fichier vide.

Positionner la tête de lecture/écriture sur le 10 000ème octet à partir du début du fichier. Ecrire un caractère à cette position

```c
#include <stdio.h> // fprintf(), perror()
#include <stdlib.h> // exit(), EXIT_FAILURE
#include <unistd.h> // write()

/* open() */
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define POSITION 10000
int main(int argc, char* argv[]) {
    if(argc < 2){
      fprintf(stderr, "Usage : %s file_name \n", argv[0]);
      exit(EXIT_FAILURE);
    }

     const char* file_name = argv[1];
     // int open(const char* pathname, int flags [, mode_t mode]);
     int open_result = open(file_name, O_WRONLY|O_CREAT | O_EXCL, 0666);
     // O_CREAT = If the file does not exist, create it.
     // O_EXCL = If the file already exists, the call will fail.
     // O_CREAT option is used, then you must include the third parameter
     // 644 - 022 = 622 -> rw- -w- -w-

     if(open_result < 0) {
        perror("Function open() : ");
        exit(EXIT_FAILURE);
     }

     if( lseek(open_result, POSITION, SEEK_SET) < 0) {
         perror("Function lseek() : ");
         exit(EXIT_FAILURE);
     }

     char buf[] = "A";
     if( write(open_result, buf, 1) < 0){
         perror("Function write() : ");
         exit(EXIT_FAILURE);
     }

     printf("Succefully \n");

} // main
```

```
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ nano question6a.c
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ nano question6a.c
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ gcc question6a.c -o question6a
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ./question6a
Usage : ./question6a file_name
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ./question6a file_name_for_test
Succefully
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ cat file_name_for_test
A[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$
```

b Quelle doit être la taille du fichier ?

Est-ce cette taille qui est retournée par la commande `ls -l` ?

Est-ce que les blocs correspondant au trou de 9 999 caractères ont été alloués (utiliser df) ?

```
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ls -l | grep file_name_for_test
-rw-r--r-- 1 ij04115 licence3in 10001 nov.   9 21:04 file_name_for_test
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$
```

La commande df est utile pour voir l'état d'occupations des disks, si je crée un fichier, je fait un df et je voie la taille occuper sur le disk. Ca a du sens sur des machines perso, mais pas sur une machine comme saphyr. Sinon, on aura fait df avant l'exécution du pgm, et df après : on vérifie le delta.

On peut réutiliser le pgm de la question2 qui renvoie tout l'information sur un fichier donner. L'info intéressante : 8 blocks de 512 octets = > pas cohérent avec le résultat du ls -l

C Ecrire un programme qui, après un certain nombre de lecture/écriture sur un fichier, retourne la valeur de l'`offset` courant.

La primitive `lseek()` permet de déplacer l'offset courant d'un fichier et retourne le nouvel `offset`

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
                                    Returns: new file offset if OK, –1 on error
```

- If *whence* is `SEEK_CUR`, the file's offset is set to its current value plus the *offset*. The *offset* can be positive or negative.

Because a successful call to `lseek` returns the new file offset, we can seek zero bytes from the current position to determine the current offset:

```
off_t    currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```

```c
#include <stdio.h> // fprintf(), perror()
#include <stdlib.h> // exit(), EXIT_FAILURE, atoi()
#include <unistd.h> // write()

/* open() */
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define POSITION 10000
int main(int argc, char* argv[]) {
    if(argc < 3){
      fprintf(stderr, "Usage : %s file_name number_write \n", argv[0]);
      exit(EXIT_FAILURE);
    }

     const char* file_name = argv[1];
     // int open(const char* pathname, int flags [, mode_t mode]);
     int open_result = open(file_name, O_WRONLY|O_CREAT | O_EXCL, 0666);
     // O_CREAT = If the file does not exist, create it.
     // O_EXCL = If the file already exists, the call will fail.
     // O_CREAT option is used, then you must include the third parameter
     // 644 - 022 = 622 -> rw- -w- -w-

     if(open_result < 0) {
        perror("Function open() : ");
        exit(EXIT_FAILURE);
     }

     if(lseek(open_result, POSITION, SEEK_SET) < 0) {
         perror("Function lseek() : ");
         exit(EXIT_FAILURE);
     }


     int i;
     for( i = 0 ; i < atoi(argv[2]) ; i++) {
         char buf[] = "A";
         if( write(open_result, buf, 1) < 0){
            perror("Function write() : ");
            exit(EXIT_FAILURE);
         }
         printf("Succefully %d \n", i+1);
     }

     printf("Current offset : %d \n", lseek(open_result, 0, SEEK_CUR));

} // main
```

```
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ nano question6c.c
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ gcc question6c.c -o question6c
question6c.c: In function 'main':
question6c.c:45:13: warning: format '%d' expects argument of type 'int', but argument 2 has type '__off
_t {aka long int}' [-Wformat=]
      printf("Current offset : %d \n", lseek(open_result, 0, SEEK_CUR));
             ^
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ./question6c
Usage : ./question6c file_name number_write
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ./question6c file_question6c_test1 0
Current offset : 10000
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ./question6c file_question6c_test2 1
Succefully 1
Current offset : 10001
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ./question6c file_question6c_test3 2
Succefully 1
Succefully 2
Current offset : 10002
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ./question6c file_question6c_test4 10
Succefully 1
Succefully 2
Succefully 3
Succefully 4
Succefully 5
Succefully 6
Succefully 7
Succefully 8
Succefully 9
Succefully 10
Current offset : 10010
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ls -l | grep file_question6c_test
-rw-r--r-- 1 ij04115 licence3in     0 nov.   9 21:34 file_question6c_test1
-rw-r--r-- 1 ij04115 licence3in 10001 nov.   9 21:34 file_question6c_test2
-rw-r--r-- 1 ij04115 licence3in 10002 nov.   9 21:35 file_question6c_test3
-rw-r--r-- 1 ij04115 licence3in 10010 nov.   9 21:35 file_question6c_test4
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$
```

d Vérifier l'affirmation suivante.

L'`offset` est associé à un fichier et non pas à un descripteur. Si deux descripteurs référencent un même fichier, la modification (par lecture/écriture ou `lseek()`) de l'offset du fichier via un descripteur est "visible" via l'autre descripteur.

dup and dup2 Functions

An existing file descriptor is duplicated by either of the following functions:

```
#include <unistd.h>

int dup(int fd);

int dup2(int fd, int fd2);
```

Both return: new file descriptor if OK, –1 on error

- → **The new file descriptor returned by `dup` is guaranteed to be the lowest-numbered available file descriptor.**
- → **With `dup2`, we specify the value of the new descriptor with the fd2 argument.**
- → **If fd2 is already open, it is first closed.**
- → **If fd equals fd2, then `dup2` returns fd2 without closing it.**

The new file descriptor that is returned as the value of the functions shares the same file table entry as the *fd* argument. We show this in Figure 3.9.

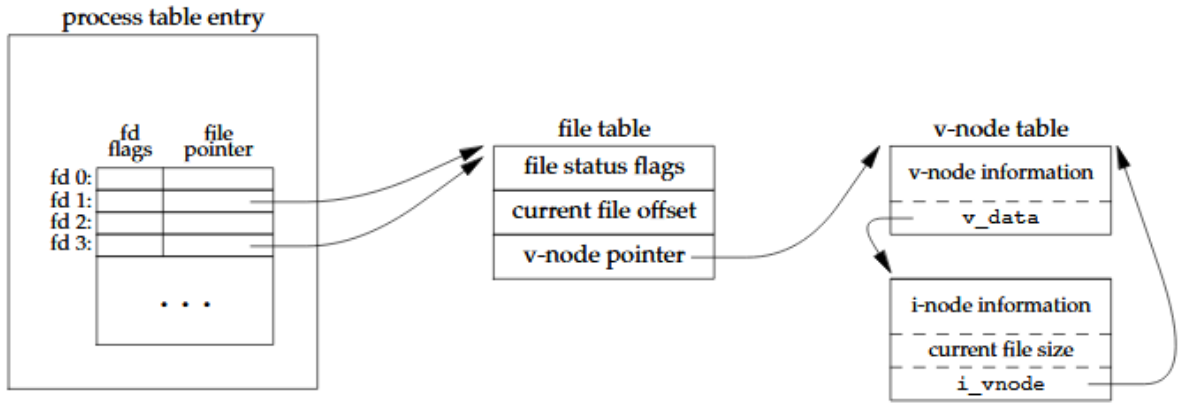


Figure 3.9 Kernel data structures after `dup(1)`

In this figure, we assume that when it's started, the process executes

```
newfd = dup(1);
```

We assume that the next available descriptor is 3 (which it probably is, since 0, 1, and 2 are opened by the shell).

Because both descriptors point to the same file table entry, they share the same file status flags—read, write, append, and so on—and the same current file offset.

```c
#include <stdio.h> // fprintf(), perror()
#include <stdlib.h> // exit(), EXIT_FAILURE, atoi()
#include <unistd.h> // write(), dup()

/* open() */
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define POSITION 10000
int main(int argc, char* argv[]) {
    if(argc < 3){
      fprintf(stderr, "Usage : %s file_name number_write \n", argv[0]);
      exit(EXIT_FAILURE);
    }

     const char* file_name = argv[1];
     // int open(const char* pathname, int flags [, mode_t mode]);
     int open_result = open(file_name, O_WRONLY|O_CREAT | O_EXCL, 0666);
     // O_CREAT = If the file does not exist, create it.
     // O_EXCL = If the file already exists, the call will fail.
     // O_CREAT option is used, then you must include the third parameter
     // 644 - 022 = 622 -> rw- -w- -w-

     if(open_result < 0) {
        perror("Function open() : ");
        exit(EXIT_FAILURE);
     }

     int dup_result = dup(open_result);
     if(lseek(dup_result, POSITION, SEEK_SET) < 0) {
         perror("Function lseek() : ");
         exit(EXIT_FAILURE);
     }


     int i;
     for( i = 0 ; i < atoi(argv[2]) ; i++) {
         char buf[] = "A";
         if( write(dup_result, buf, 1) < 0){
            perror("Function write() : ");
            exit(EXIT_FAILURE);
         }
         printf("Succefully %d \n", i+1);
     }

     printf("Current offset : %d \n", lseek(open_result, 0, SEEK_CUR));
} // main
```

```
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ nano question6d.c
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ gcc question6d.c -o question6d
question6d.c: In function 'main':
question6d.c:46:13: warning: format '%d' expects argument of type 'int', but argument 2 has t
ype '__off_t {aka long int}' [-Wformat=]
      printf("Current offset : %d \n", lseek(open_result, 0, SEEK_CUR));
             ^
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ./question6d
Usage : ./question6d file_name number_write
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ./question6d question6d_file_test1 0
Current offset : 10000
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ./question6d question6d_file_test2 1
Succefully 1
Current offset : 10001
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ./question6d question6d_file_test3 2
Succefully 1
Succefully 2
Current offset : 10002
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ./question6d question6d_file_test4 10
Succefully 1
Succefully 2
Succefully 3
Succefully 4
Succefully 5
Succefully 6
Succefully 7
Succefully 8
Succefully 9
Succefully 10
Current offset : 10010
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$ ls -l | grep question6d
-rwxr-xr-x 1 ij04115 licence3in  7712 nov.   9 23:49 question6d
-rw-r--r-- 1 ij04115 licence3in  1440 nov.   9 23:49 question6d.c
-rw-r--r-- 1 ij04115 licence3in     0 nov.   9 23:50 question6d_file_test1
-rw-r--r-- 1 ij04115 licence3in 10001 nov.   9 23:50 question6d_file_test2
-rw-r--r-- 1 ij04115 licence3in 10002 nov.   9 23:50 question6d_file_test3
-rw-r--r-- 1 ij04115 licence3in 10010 nov.   9 23:50 question6d_file_test4
[ij04115@saphyr:~/unix_tp4]:lun. nov. 09$
```

7 Manipulation de répertoires opendir( ) , readdir( ) , mkdir( )

PAS DANS LES CONTROLS
ANNEE 2020-2021

La primitive `DIR *opendir(const char *pathname)` permet d'ouvrir en lecture le répertoire référencé par `pathname`.

- Ouverture de répertoire

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

  - Ouvre le répertoire référencé par `pathname` en lecture et alloue un objet de type `DIR` dont l'adresse est renvoyée en retour.
  - Retourne un pointeur ou `NULL` en cas d'erreur.

La primitive `struct dirent *readdir(DIR *dp)` permet de lire l'entrée suivante du répertoire identifié par `dp`.

- Lecture d'une entrée de répertoire

```
#include <dirent.h>
   struct dirent *readdir(DIR *dp);
```

  - Lecture de l'entrée suivante dans le répertoire référencé par `dp`.
  - Retourne un pointeur ou `NULL` en fin de fichier et en cas d'erreur.

```
struct dirent {
        ino_t d_ino;                /* i-node number */
        off_t d_off;                /* offset to this dirent */
        unsigned short d_reclen;    /* length of this d_name */
        char d_name [NAME_MAX+1];   /* null-terminated filename */
        unsigned short d_type       /* type */
}
```

- Fermeture de répertoire

```
#include <dirent.h>
    int closedir (DIR *dp);
```

  - Libère les ressources allouées lors de l'appel à `opendir`.
  - Retourne 0 en cas de succès et -1 en cas d'erreur.

a **Ecrire un programme qui ouvre un répertoire (/tmp par exemple) et qui affiche tous les fichiers qui y sont contenus ainsi que le type de chacun (champ d_type de la strucure dirent)**

```c
#include <stdio.h> // perror(), fprintf()
#include <dirent.h> // opendir(), closedir(), readdir()
#include <stdlib.h> // exit(), EXIT_FAILURE

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

      if(argc < 2) {
       fprintf(stderr, "Usage : %s dir_patch \n", argv[0]);
       exit(EXIT_FAILURE);
      }

     DIR* dp;
     struct dirent* dirp;

     // DIR* opendir(const char* pathname)
     if( (dp = opendir(argv[1])) == NULL){
       perror("Can't open directory : ");
       exit(EXIT_FAILURE);
     }

     while( (dirp = readdir(dp)) != NULL)
        printf("%d %s \n", dirp->d_type, dirp->d_name);

    closedir(dp);
}
```

```
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ mkdir newDir
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ cd newDir
[ij04115@saphyr:~/unix_tp4/newDir]:mar. nov. 10$ nano fileTest
[ij04115@saphyr:~/unix_tp4/newDir]:mar. nov. 10$ mkdir testDir
[ij04115@saphyr:~/unix_tp4/newDir]:mar. nov. 10$ ls
fileTest  testDir
[ij04115@saphyr:~/unix_tp4/newDir]:mar. nov. 10$ cd ~/unix_tp4
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ nano question7a.c
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ gcc question7a.c -o question7a
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ ./question7a
Usage : ./question7a dir_patch
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ ./question7a newDir
0 .
0 ..
4 testDir
8 fileTest
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ ./question7a /tmp
4 .
4 ..
4 .ICE-unix
4 .X11-unix
4 .font-unix
4 .XIM-unix
4 systemd-private-1fd7eda1817141448e97d0f8942a44d2-systemd-timesyncd.service-KGrobx
4 .Test-unix
4 tmux-12794
4 vmware-root
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$
```

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

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

    if ( (dp = opendir (argv[1])) == NULL )
        printf("Erreur Ouverture\n");

    while ( (dirp = readdir(dp)) != NULL ) {
        if ( strcmp(dirp->d_name, ".") == 0 || strcmp(dirp->d_name, "..") == 0 )
        continue;

        printf("fichier trouvé : %s\t de type : %d\n", dirp->d_name, dirp->d_type);
    }

closedir(dp);
}
```

51

- **we include a system header,** `dirent.h`**, to pick up the function prototypes for** `opendir` **and** `readdir`**, in addition to the definition of the** `dirent` **structure.**
- •The `opendir` function returns a pointer to a `DIR` structure, and we pass this pointer to the `readdir` function. We don't care what's in the `DIR` structure. We then call `readdir` in a loop, to read each directory entry. The `readdir` function returns a pointer to a `dirent` structureor, when it's finished with the directory, a null pointer. All we examine in the `dirent` structure is the name of each directory entry (`d_name`). Using this name, we could then call the `stat` function to determine all the attributes of the file.

La primitive

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

permet de créer le répertoire de nom référencé par `pathname`.

- Création de répertoire

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

  - Crée un nouveau répertoire vide. Les entrées "." et ".." sont automatiquement créés.
  - Retourne 0 en cas de succès et -1 en cas d'erreur.

```
int        mkdir(const char *path, mode_t mode);
                                  <sys/stat.h>
                                  mode: S_IS[UG]ID, S_ISVTX,
                                        S_I[RWX](USR|GRP|OTH)
                                  Returns: 0 if OK, –1 on error
```

The specified file access permissions, `mode` ,are modified by the file mode creation mask of the process. A common mistake is to specify the same mode as for a file: read and write permissions only. But for a directory, we normally want at least one of the execute bits enabled, to allow access to filenames within the directory.

b Modifier le programme précédent en y ajoutant à la fin la création (dans le répertoire /tmp) d'un répertoire ayant pour nom votre propre nom.

C Vérifier que le répertoire a bien été créé.

```c
#include <stdio.h> // perror(), fprintf()
#include <dirent.h> // opendir(), closedir(), readdir()
#include <stdlib.h> // exit(), EXIT_FAILURE
#include <sys/stat.h> // mkdir()

#define FILE_MODE  (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define DIR_MODE   (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

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

     DIR* dp;
     struct dirent* dirp;

     // DIR* opendir(const char* pathname)
     if( (dp = opendir("/tmp")) == NULL){
       perror("Can't open directory : ");
       exit(EXIT_FAILURE);
     }

     while( (dirp = readdir(dp)) != NULL)
        printf("%d %s \n", dirp->d_type, dirp->d_name);

    if( mkdir("/tmp/NAMOLARU/", DIR_MODE) == 0)
        printf("Created !\n");

    closedir(dp);
}
```

```
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ nano question7b.c
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ gcc question7b.c -o question7b
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ ./question7b
4 .
4 ..
4 .ICE-unix
4 .X11-unix
4 .font-unix
4 test
4 .XIM-unix
4 systemd-private-1fd7eda1817141448e97d0f8942a44d2-systemd-timesyncd.service-KGrobx
4 ij04115
4 .Test-unix
4 tmux-12794
4 vmware-root
Created !
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ ls /tmp
ij04115   systemd-private-1fd7eda1817141448e97d0f8942a44d2-systemd-timesyncd.service-KGrobx  tmux-12794
NAMOLARU  test                                                                              vmware-root
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$
```

```
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ ls -l /tmp
total 24
drwxr-xr-x 2 ij04115 licence3in 4096 nov.  10 09:04 ij04115
drwxr-xr-x 2 ij04115 licence3in 4096 nov.  10 09:05 NAMOLARU
drwx------ 3 root    root       4096 nov.   4 22:14 systemd-private
syncd.service-KGrobx
drwxr-xr-x 2 ij04115 licence3in 4096 nov.  10 08:54 test
drwx------ 2 ik03158 licence3in 4096 nov.   5 10:02 tmux-12794
drwx------ 2 root    root       4096 nov.   4 22:14 vmware-root
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$
```

d Modifier le programme précédent en y ajoutant à la fin

1) le positionnement en début du pointeur de lecture du répertoire et

2) l'affichage à nouveau du contenu du répertoire (/tmp).

La primitive `void rewinddir(DIR *dp)` permet de positionner le pointeur de lecture dans un répertoire sur la première entrée de ce répertoire.

- Repositionnement du pointeur de lecture

```
#include <dirent.h>
    void rewinddir(DIR *dp);
```

- Suppression de répertoire (vide)

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

  - Retourne 0 en cas de succès et -1 en cas d'erreur.

```c
#include <stdio.h> // perror(), fprintf()
#include <dirent.h> // opendir(), closedir(), readdir(), rewinddir()
#include <stdlib.h> // exit(), EXIT_FAILURE
#include <sys/stat.h> // mkdir()
#include <unistd.h> //rmdir()

#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define DIR_MODE  (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

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

     DIR* dp;
     struct dirent* dirp;

     // DIR* opendir(const char* pathname)
     if( (dp = opendir("/tmp")) == NULL){
       perror("Can't open directory : ");
       exit(EXIT_FAILURE);
     }

     while( (dirp = readdir(dp)) != NULL)
        printf("%d %s \n", dirp->d_type, dirp->d_name);

    if( mkdir("/tmp/NAMOLARU/", DIR_MODE) == 0)
        printf("Created !\n");

    rewinddir(dp);
    while( (dirp = readdir(dp)) != NULL)
        printf("%d %s \n", dirp->d_type, dirp->d_name);

    closedir(dp);
    if( rmdir("/tmp/NAMOLARU") == 0)
      printf("Suppression de repertoire \n");

}
```

```
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ nano question7last.c
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ gcc question7last.c -o question7last
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ ./question7last
4 .
4 ..
4 .ICE-unix
4 .X11-unix
4 .font-unix
4 test
4 .XIM-unix
4 systemd-private-1fd7eda1817141448e97d0f8942a44d2-systemd-timesyncd.service-KG
4 ij04115
4 .Test-unix
4 tmux-12794
4 NAMOLARU
4 vmware-root
4 .
4 ..
4 .ICE-unix
4 .X11-unix
4 .font-unix
4 test
4 .XIM-unix
4 systemd-private-1fd7eda1817141448e97d0f8942a44d2-systemd-timesyncd.service-KG
4 ij04115
4 .Test-unix
4 tmux-12794
4 NAMOLARU
4 vmware-root
Suppression de repertoire
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$ ./question7last
4 .
4 ..
4 .ICE-unix
4 .X11-unix
4 .font-unix
4 test
4 .XIM-unix
4 systemd-private-1fd7eda1817141448e97d0f8942a44d2-systemd-timesyncd.service-KG
4 ij04115
4 .Test-unix
4 tmux-12794
4 vmware-root
Created !
4 .
4 ..
4 .ICE-unix
4 .X11-unix
4 .font-unix
4 test
4 .XIM-unix
4 systemd-private-1fd7eda1817141448e97d0f8942a44d2-systemd-timesyncd.service-KG
4 ij04115
4 .Test-unix
4 tmux-12794
4 NAMOLARU
4 vmware-root
Suppression de repertoire
[ij04115@saphyr:~/unix_tp4]:mar. nov. 10$
```