# UE Programmation Unix

# TP4-b Gestion des Fichiers

## Les appels système : communication par tubes

**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`



- 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

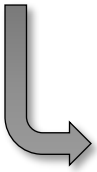
# Exemple de cours

```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
```

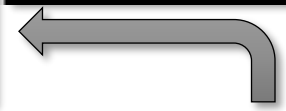


```
[ij04115@saphyr:~/unix_tpl]:ven. sept. 11$ nano affich_param.c
```



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
```

# interprocess communication (IPC)

**we described the process control primitives and saw how to work with multiple processes. But the only way for these processes to exchange information is by passing open files across a `fork` or an `exec` or through the file system. We'll now describe other techniques for processes to communicate with one another: interprocess communication(IPC)**

# Communication par tubes *Pipes*

**Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems.**

**Pipes have two limitations.**

1. **Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.**

2. **Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls `fork`, and the pipe is used between the parent and the child.**

**We'll see that `FIFO`s get around the second limitation, and that UNIX domain `sockets` get around both limitations.**

## Communication par tubes

- Tube : mécanisme de communication appartenant au système de fichiers.
  → nœud (type : `S_IFIFO`), descripteur, `read`, `write`, ...
- Canal unidirectionnel (une entrée, une sortie)
  → deux entrées dans la table des fichiers ouverts
- Lecture destructrice
- Communication d'un flot continu de caractères
- Gestion en mode FIFO
- Capacité finie (nombre d'adresses directes) → tube plein
- Nombres de lecteurs/écrivains

# 1 Communication par tubes ordinaires pipe( )

**Les tubes ordinaires permettent la communication entre processus de même filiation.**

→ **Despite these limitations, half-duplex pipes are still the most commonly used form of IPC.**

→ **Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe.**

→ **A pipe is created by calling the `pipe` function.**

# Tubes ordinaires

- Compteur de liens nul (aucune référence à ce nœud)
- Supprimé lorsque aucun processus ne l'utilise
- Impossibilité d'ouvrir un tube (pas de `open()`)
- Connaissance de l'existence → possession d'un descripteur (création ou héritage)
- Communication entre processus ayant un ancêtre commun
- Perte d'accès à un tube irréversible

# Création d'un tube ordinaire

- La primitive de création `pipe()`

```
#include <unistd.h>
   int pipe(int filedes[2]);
```

  - alloue un nœud, deux entrées dans la table des fichiers ouverts et deux descripteurs dans la table du processus appelant,
  - retourne 0 en cas de succès et -1 sinon.

- Manipulation de tubes ordinaires
  - `read, write, close, fstat, fcntl`

  Opération interdite : `lseek`

```
#include <unistd.h>
int pipe(int fd[2]);
```

Returns: 0 if OK, –1 on error

**Two file descriptors are returned through the `fd` argument: `fd[0]` is open for reading, and `fd[1]` is open for writing. The output of `fd[1]` is the input for `fd[0]`.**

**Two ways to picture a half-duplex pipe are shown in Figure15.2. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.**

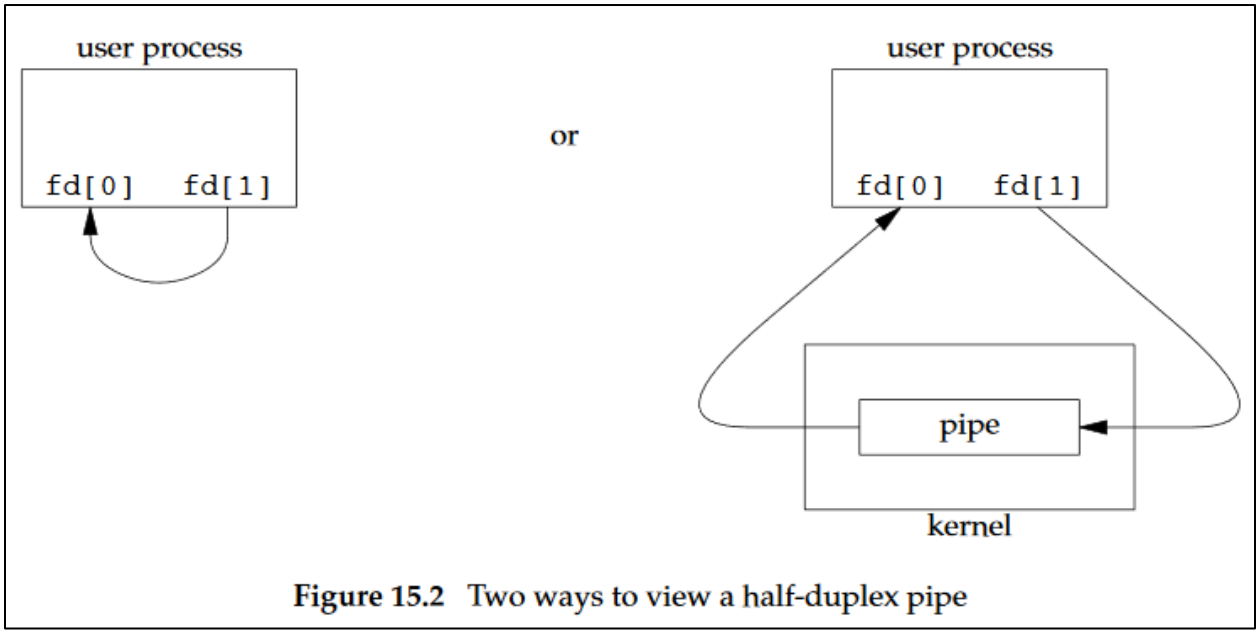



Figure 15.2 Two ways to view a half-duplex pipe

**The `fstat` function returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.**

**A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child, or vice versa. Figure15.3 shows this scenario.**

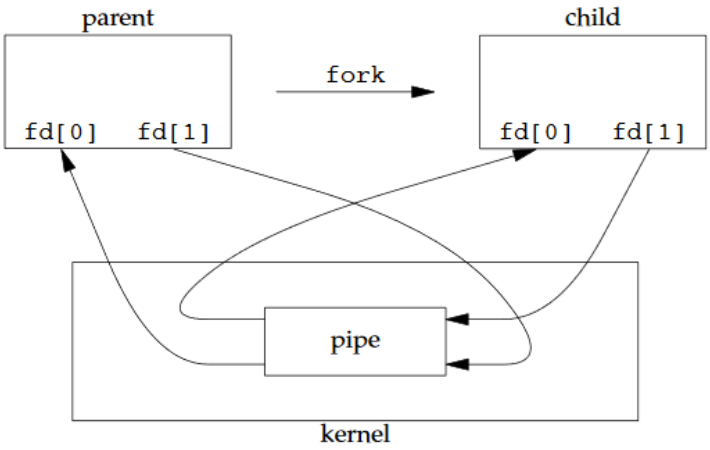



Figure 15.3 Half-duplex pipe after a `fork`

**What happens after the `fork` depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe(`fd[0]`), and the child closes the write end (`fd[1]`). Figure15.4 shows the resulting arrangement of descriptors.**

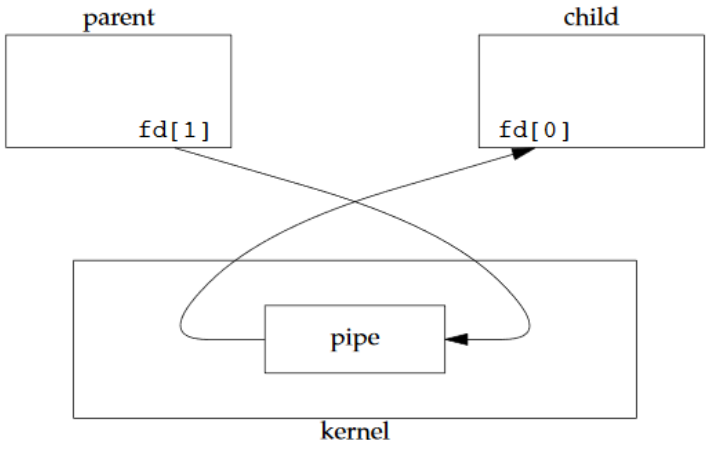



Figure 15.4 Pipe from parent to child

**For a pipe from the child to the parent, the parent closes `fd[1]`, and the child closes `fd[0]`.**

## When one end of a pipe is closed, two rules apply.

1. If we `read` from a pipe whose write end has been closed, `read` returns 0 to indicate an end of file after all the data has been read. (Technically, we should say that this end of file is not generated until there are no more writers for the pipe. It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing. Normally, however, there is a single reader and a single writer for a pipe. When we get to FIFOs in the next section, we'll see that often there are multiple writers for a single FIFO.)
2. If we `write` to a pipe whose read end has been closed, the signal `SIGPIPE` is generated.

```c
#include <stdio.h> // perror(), printf(), fprintf()
#include <stdlib.h> // exit()
#include <unistd.h> // pipe(), fork(), close(), write(), read()
#include <string.h> // strlen()

#define MAX_DATA 10
int main(int argc, char* argv[]) {
   if(argc < 2) {
      fprintf(stderr,"Usage : %s data(max : %d)\n", argv[0], MAX_DATA);
      exit(1);
   } // if

   /* int pipe(int fd[2]); returns 0 if OK, -1 on error
    * fd : Two file descriptors
    * fd[0] is open for reading, fd[1] open for writing
    * The output of fd[1] is the input for fd[0]
    */
    int fd[2];
    int result_pipe = pipe(fd);
    if(result_pipe < 0){
       perror("Function pipe() : ");
       exit(1);
   }

   // pid_t fork(void); Returns 0 in child, pid in parent, -1=error
   int fork_result = fork();
   if(fork_result < 0)
   {
      perror("Function fork() : ");
      exit(1);
   }
   else
   {

     if(fork_result > 0) // CODE DU PERE
     {
        /* fd[0] is open for reading, fd[1] open for writing
         * The output of fd[1] is the input for fd[0]
         */
         close( fd[0] ); // int close(int fd);

         /* ssize_t write(int fd, const void *buf, size_t nbytes);
          * Returns number of bytes wrriten if OK, -1 on error
          */
         int write_result = write(fd[1], argv[1], strlen(argv[1]));
         if(write_result < 0) {
            perror("Function write() : ");
            exit(1);
         } // if(write_result < 0)

         close( fd[1] );
     } // CODE DU PERE
```

```
    else // CODE DU FILS (fork_result == 0)
    {
      close( fd[1] );
      char buffer[MAX_DATA];

      /* ssize_t read(int fd, void* buf, size_t nbytes) ;
       * Returns number of bytes read, 0 if end of file, -1 = error
       */
      int read_result = read(fd[0], buffer, MAX_DATA);
      if(read_result < 0) {
         perror("Function read() : ");
         exit(1);
      } // if(read_result < 0)

      buffer[read_result] = '\0';
      printf("DATA : %s \n", buffer);

      close( fd[0] );
    } // CODE DU FILS
  }
  exit(0);
} // main
```

```
[ij04115@saphyr:~/unix_tp4b]:mer. nov. 11$ nano questionla.c
[ij04115@saphyr:~/unix_tp4b]:mer. nov. 11$ gcc questionla.c -o questionla
[ij04115@saphyr:~/unix_tp4b]:mer. nov. 11$ ./questionla
Usage : ./questionla data(max : 10)
[ij04115@saphyr:~/unix_tp4b]:mer. nov. 11$ ./questionla test
DATA : test
[ij04115@saphyr:~/unix_tp4b]:mer. nov. 11$ ./questionla test
DATA : test
[ij04115@saphyr:~/unix_tp4b]:mer. nov. 11$ ./questionla hello
DATA : hello
```

**The pipe direction here matches the orientation shown in Figure 15.4**

**For a pipe from the parent to the child, the parent closes the read end of the pipe(`fd[0]`), and the child closes the write end (`fd[1]`). Figure15.4 shows the resulting arrangement of descriptors.**

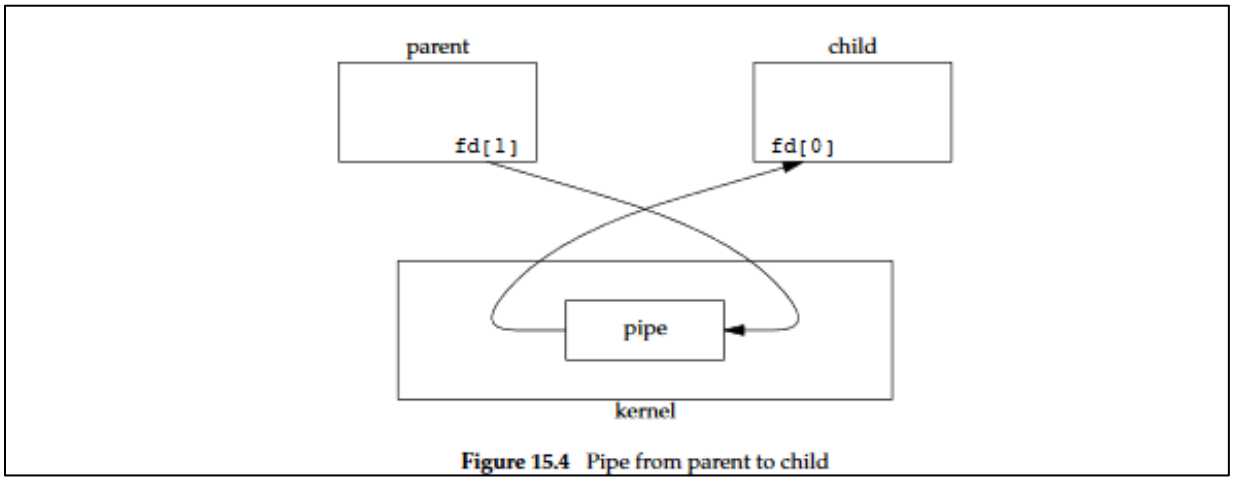



**Figure 15.4** Pipe from parent to child

## b Modifier le programme précédent de façon que les deux processus communiquent dans les deux sens.

**Utiliser deux tubes et chaque processus doit garder ouvert l'ensemble des descripteurs de tubes.**

```
#include <stdio.h> // perror(), printf(), fprintf()
#include <stdlib.h> // exit()
#include <unistd.h> // pipe(), fork(), close(), write(), read()
#include <string.h> // strlen()

#define MAX_DATA 10
int main(int argc, char* argv[]) {
   if(argc < 3) {
      fprintf(stderr,"Usage : %s data1(pere -> fils), data2(fils->pere) \n", argv[0]);
      exit(1);
   } // if

   /* int pipe(int fd[2]); returns 0 if OK, -1 on error
    * fd : Two file descriptors
    * fd[0] is open for reading, fd[1] open for writing
    * The output of fd[1] is the input for fd[0]
    */
    int fd[2], fd2[2];
    int result_pipe = pipe(fd);
    int result_pipe2 = pipe(fd2);
    if(result_pipe < 0 || result_pipe2 < 0){
       perror("Function pipe() : ");
       exit(1);
   }

   // pid_t fork(void); Returns 0 in child, pid in parent, -1=error
   int fork_result = fork();
   if(fork_result < 0)
   {
      perror("Function fork() : ");
      exit(1);
   }
   else
   {

     if(fork_result > 0) // CODE DU PERE
     {
        /* fd[0] is open for reading, fd[1] open for writing
         * The output of fd[1] is the input for fd[0]
         */
         close( fd[0] ); // int close(int fd);
         close( fd2[1]);
         /* ssize_t write(int fd, const void *buf, size_t nbytes);
          * Returns number of bytes wrriten if OK, -1 on error
          */
         int write_result = write(fd[1], argv[1], strlen(argv[1]));
         if(write_result < 0) {
            perror("Function write() : ");
            exit(1);
         } // if(write_result < 0)

         char buffer2[MAX_DATA];
         int read_result2 = read(fd2[0], buffer2, MAX_DATA);
         if(read_result2 < 0) {
            perror("Function read() : ");
            exit(1);
         } // if(read_result2 < 0)

         buffer2[read_result2] = '\0';
         printf("DATA fils -> père : %s\n", buffer2);

         close( fd[1] );
         close( fd2[0]);
     } // CODE DU PERE
```

```
    else // CODE DU FILS (fork_result == 0)
    {
      close( fd[1] );
      close( fd2[0]);
      char buffer[MAX_DATA];

      /* ssize_t read(int fd, void* buf, size_t nbytes) ;
       * Returns number of bytes read, 0 if end of file, -1 = error
       */
      int read_result = read(fd[0], buffer, MAX_DATA);
      if(read_result < 0) {
         perror("Function read() : ");
         exit(1);
      } // if(read_result < 0)

      buffer[read_result] = '\0';
      printf("DATA père -> fils : %s \n", buffer);

      int write_result2 = write(fd2[1], argv[2], strlen(argv[2]));
      if(write_result2 < 0) {
        perror("Function write() : ");
        exit(1);
      }

      close( fd[0] );
      close( fd2[1]);
    } // CODE DU FILS
  }
  exit(0);
} // main
```

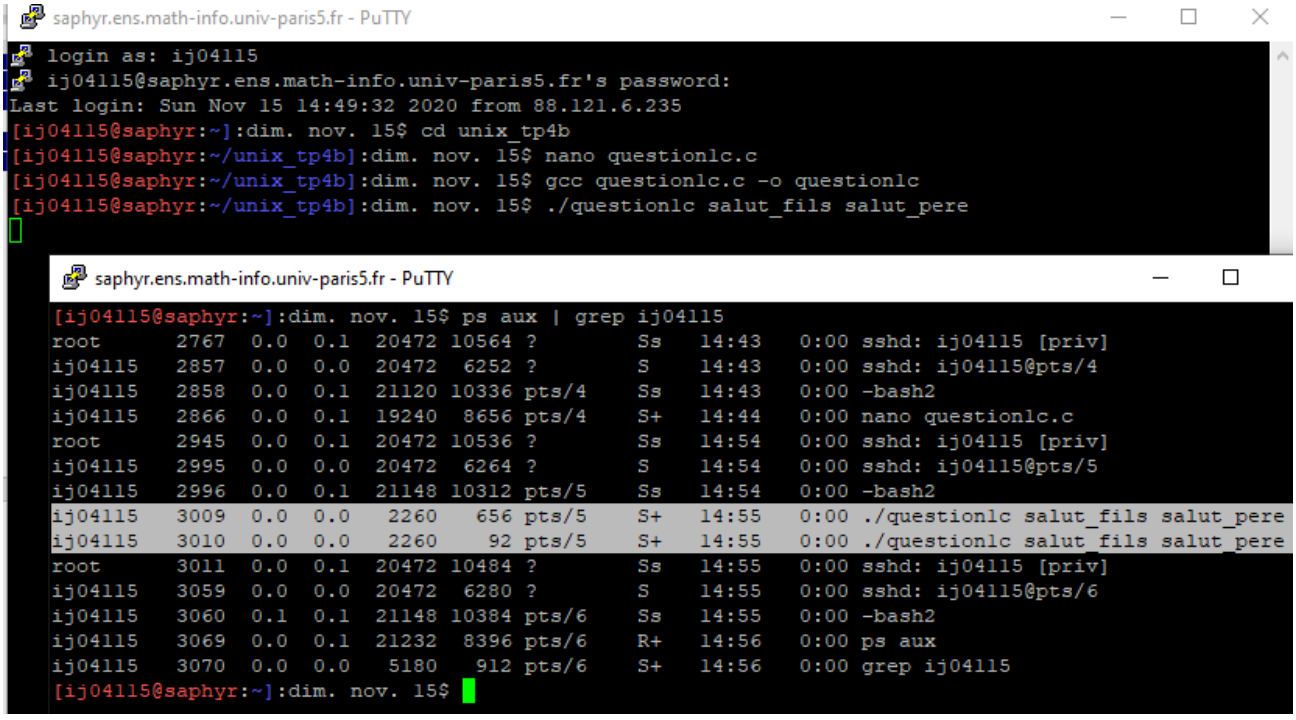
```
[ij04115@saphyr:~/unix_tp4b]:mer. nov. 11$ nano questionlb.c
[ij04115@saphyr:~/unix_tp4b]:mer. nov. 11$ gcc questionlb.c -o questionlb
[ij04115@saphyr:~/unix_tp4b]:mer. nov. 11$ ./questionlb
Usage : ./questionlb data1(pere -> fils), data2(fils->pere)
[ij04115@saphyr:~/unix_tp4b]:mer. nov. 11$ ./questionlb f p
DATA père -> fils : f
DATA fils -> père : p
[ij04115@saphyr:~/unix_tp4b]:mer. nov. 11$ ./questionlb salut_fils salut_pere
DATA père -> fils : salut_fils
DATA fils -> père : salut_pere
[ij04115@saphyr:~/unix_tp4b]:mer. nov. 11$
```

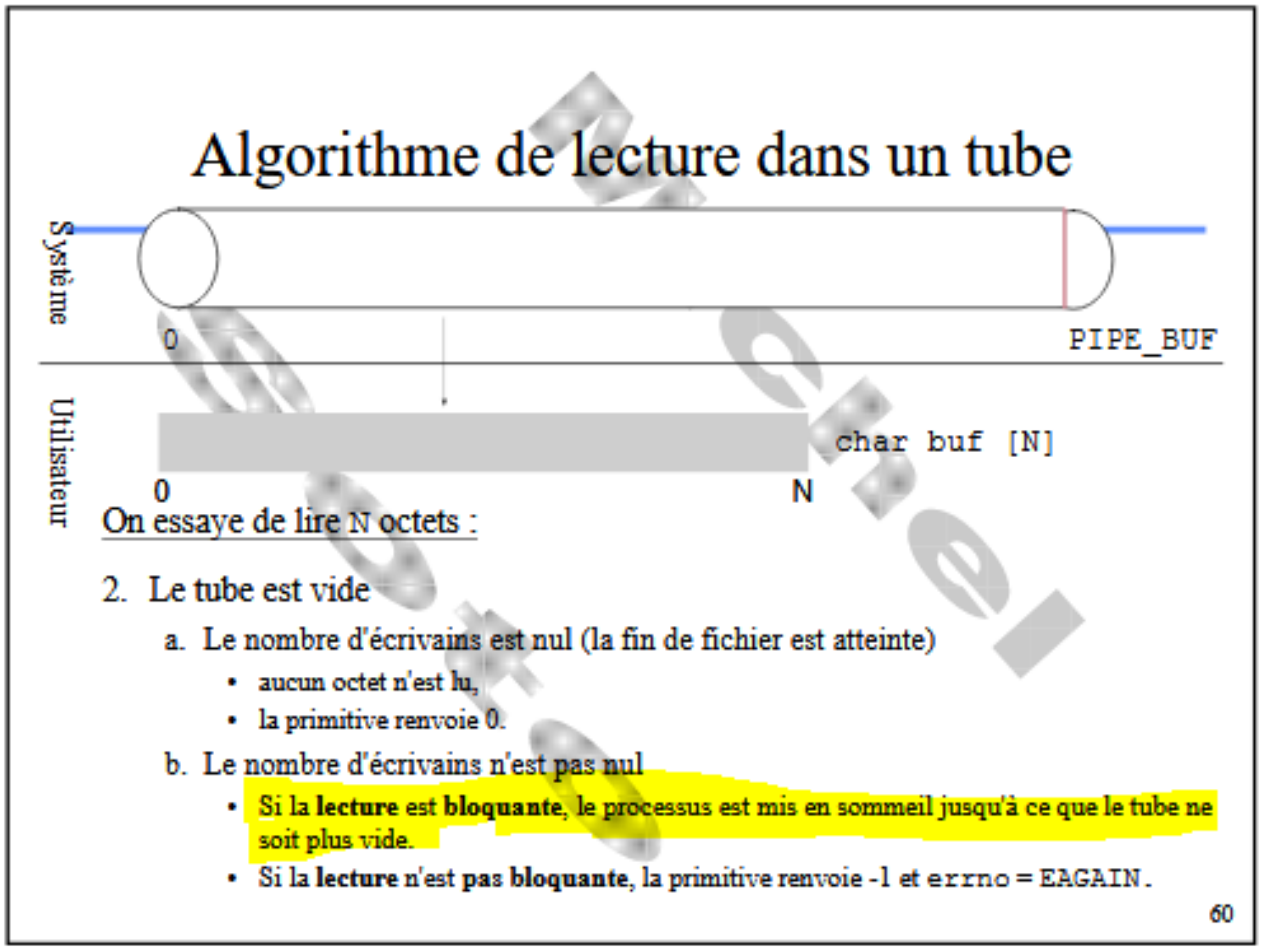
**Quel comportement obtenez-vous de l'exécution du programme précédent si les deux processus commencent chacun par une lecture ?**

**Quelles solutions à ce problème ?**

```c
#include <stdio.h> // perror(), printf(), fprintf()
#include <stdlib.h> // exit()
#include <unistd.h> // pipe(), fork(), close(), write(), read()
#include <string.h> // strlen()

#define MAX_DATA 10
int main(int argc, char* argv[]) {
   if(argc < 3) {
      fprintf(stderr,"Usage : %s data1(pere -> fils), data2(fils->pere) \n", argv[0]);
      exit(1);
   } // if

   /* int pipe(int fd[2]); returns 0 if OK, -1 on error
    * fd : Two file descriptors
    * fd[0] is open for reading, fd[1] open for writing
    * The output of fd[1] is the input for fd[0]
    */
    int fd[2], fd2[2];
    int result_pipe = pipe(fd);
    int result_pipe2 = pipe(fd2);
    if(result_pipe < 0 || result_pipe2 < 0){
       perror("Function pipe() : ");
       exit(1);
   }

   // pid_t fork(void); Returns 0 in child, pid in parent, -1=error
   int fork_result = fork();
   if(fork_result < 0)
   {
      perror("Function fork() : ");
      exit(1);
   }
   else
   {

     if(fork_result > 0) // CODE DU PERE
     {
        /* fd[0] is open for reading, fd[1] open for writing
         * The output of fd[1] is the input for fd[0]
         */
         close( fd[0] ); // int close(int fd);
         close( fd2[1]);

         char buffer2[MAX_DATA];
         int read_result2 = read(fd2[0], buffer2, MAX_DATA);
         if(read_result2 < 0) {
            perror("Function read() : ");
            exit(1);
         } // if(read_result2 < 0)

         buffer2[read_result2] = '\0';
         printf("DATA fils -> père : %s\n", buffer2);

         /* ssize_t write(int fd, const void *buf, size_t nbytes);
          * Returns number of bytes wrriten if OK, -1 on error
          */
         int write_result = write(fd[1], argv[1], strlen(argv[1]));
         if(write_result < 0) {
            perror("Function write() : ");
            exit(1);
         } // if(write_result < 0)


         close( fd[1] );
         close( fd2[0]);
     } // CODE DU PERE
```

```
    else // CODE DU FILS (fork_result == 0)
    {
      close( fd[1] );
      close( fd2[0]);
      char buffer[MAX_DATA];

      /* ssize_t read(int fd, void* buf, size_t nbytes) ;
       * Returns number of bytes read, 0 if end of file, -1 = error
       */
      int read_result = read(fd[0], buffer, MAX_DATA);
      if(read_result < 0) {
         perror("Function read() : ");
         exit(1);
      } // if(read_result < 0)

      buffer[read_result] = '\0';
      printf("DATA père -> fils : %s \n", buffer);

      int write_result2 = write(fd2[1], argv[2], strlen(argv[2]));
      if(write_result2 < 0) {
        perror("Function write() : ");
        exit(1);
      }

      close( fd[0] );
      close( fd2[1]);
    } // CODE DU FILS
  }
  exit(0);
} // main
```



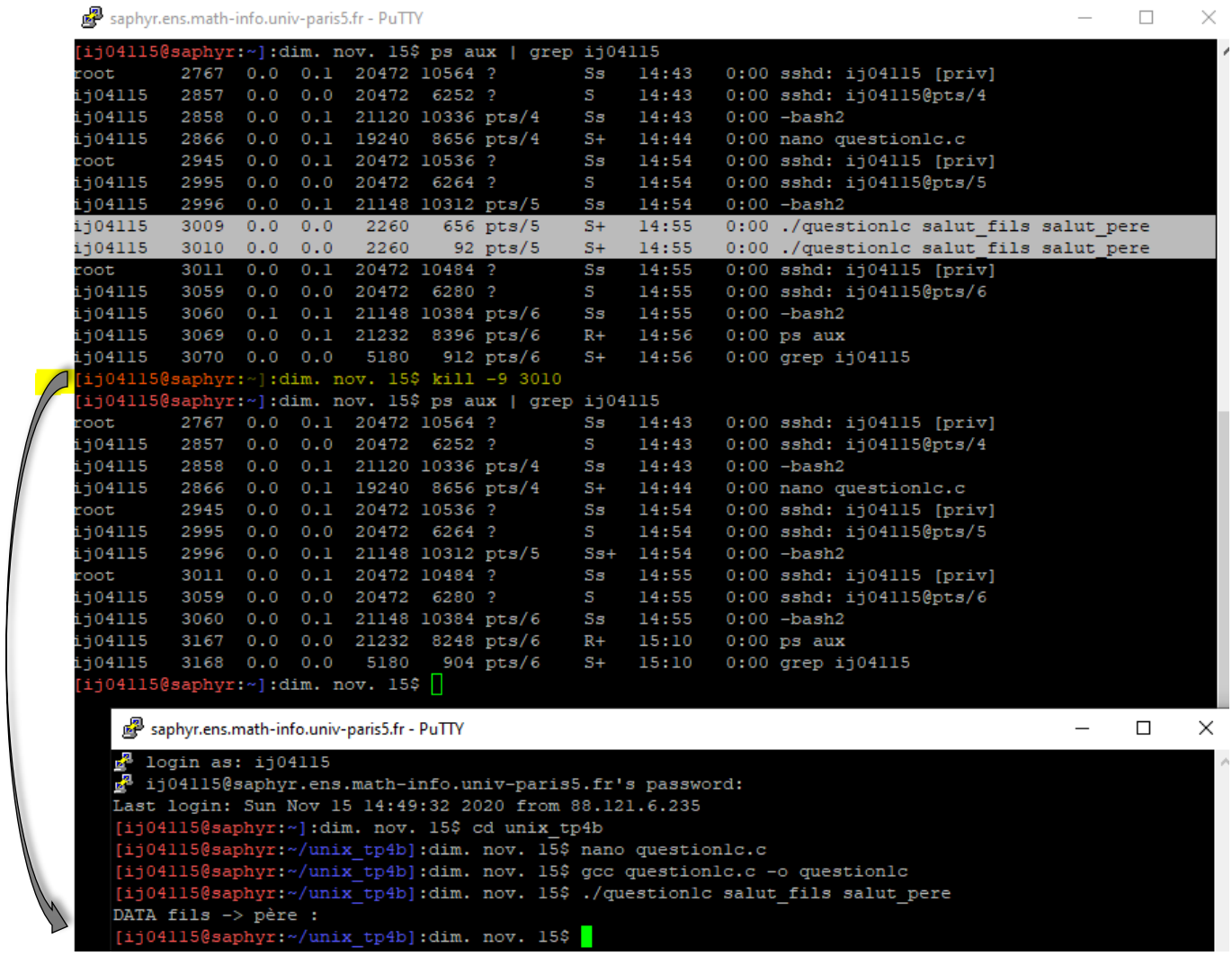



**Quelles solutions à ce problème ?**

# Les deux processus ne doivent pas commencer par une lecture

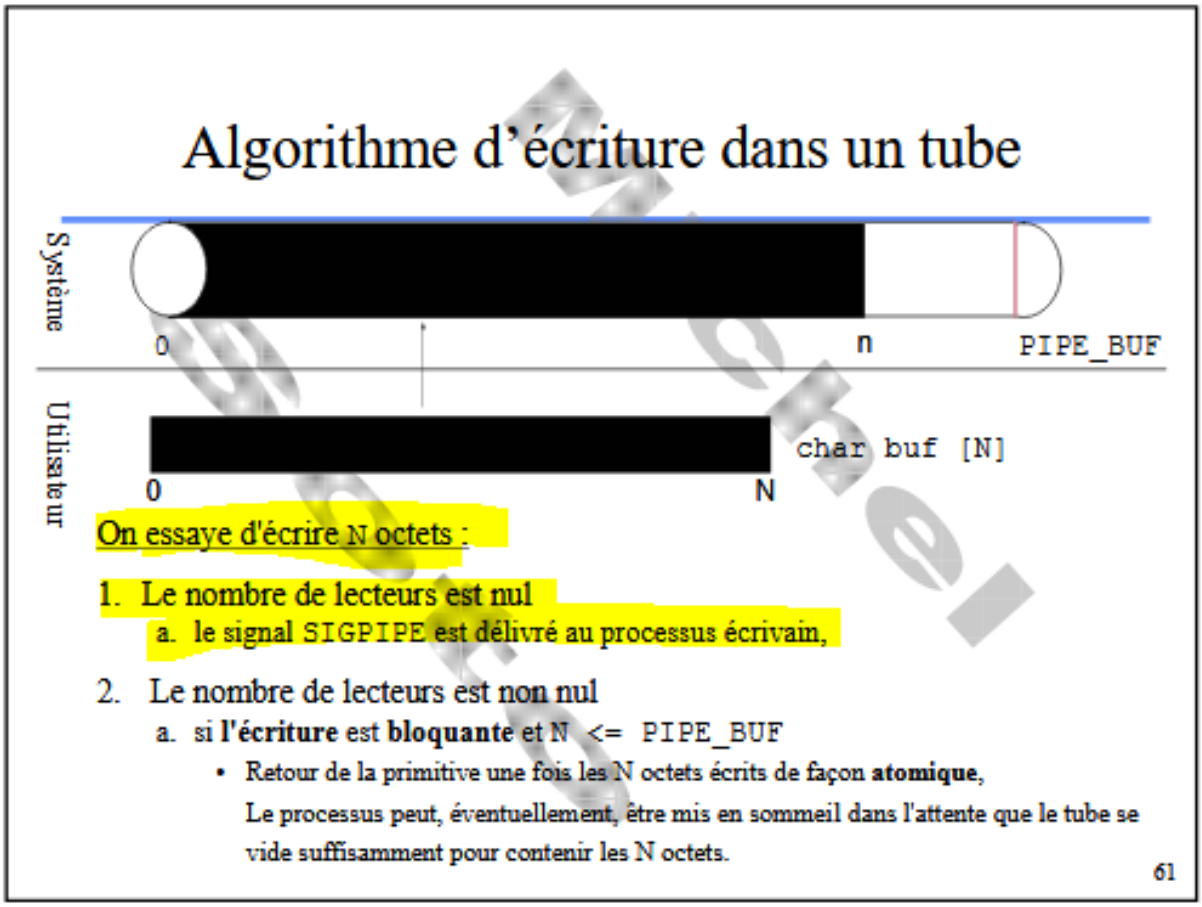
**d** **Envoyer un signal au processus fils.**

**Que constatez-vous ? Expliquer ce qui se passe.**




**Le processus père se termine.**

**La raison: lorsque le père écrit dans un tube destiné à transférer des données du père au fils, le nombre de lecteurs est nul donc le signal SIGPIPE est délivré au processus écrivain,**




# 2 Communication par tubes nommés mkfifo( ), mknod( )

## Les tubes nommés permettent la communication entre processus, même sans lien de parenté.

### FIFOs

→ **FIFOs are sometimes called named pipes.**
→ **Unnamed pipes can be used only between related processes when a common ancestor has created the pipe.**
→ **With FIFOs, however, unrelated processes can exchange data.**
→ **FIFO is a type of file.**
→ **Creating a FIFO is similar to creating a file.**

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

int mkfifo(const char *path, mode_t mode);        return: 0 if OK, –1 on error
```

The specification of the mode argument is the same as for the `open` function

**Applications can create FIFOs also with the mknod**

Using `mkfifo()` is standardized and portable. Using `mknod()` in general, is not portable — it is a part of POSIX (despite a statement to the contrary in an earlier version of this answer). The POSIX specification says that `mkfifo()` should be preferred. Otherwise, there is no difference between a FIFO created by `mkfifo()` and `mknod()`.

Note that `mknod()` can be used to create other device types than just FIFOs. It can create block special and character special devices. Once upon a very (very, very) long time ago, `mknod()` was used to create directories too — in the days before there was a `mkdir()` or `rmdir()` system call. After creating the directory, you had to use `link()` twice to create the `.` and `..` entries in the new directory. (And you had to have root privileges to use it, so the `mkdir` and `rmdir` commands were SUID root.) File systems are a lot more reliable nowadays because that's no longer part of the game.

Reference: Version 7 Unix — circa 1979.

**Source : https://stackoverflow.com/questions/43023329/difference-between-mkfifo-and-mknod**

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

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

- Les primitives de création d'un tube nommé

```
#include <sys/types.h>
#include <sys/stat.h>
    int mkfifo(const char *pathname, mode_t mode);
    int mknod(const char *pathname, mode_t mode |
              S_IFIFO, dev_t dev);
```

- Manipulation des tubes ordinaires
  - `open, close`
  - `read, write`
  - `unlink`

→ **Once we have used `mkfifo` to create a FIFO, we open it using `open`. Indeed, the normal file I/O functions (e.g., `close`, `read`, `write`, `unlink`) all work with FIFOs.**

→ **As with a pipe, if we `write` to a FIFO that no process has open for reading, the signal `SIGPIPE` is generated.**

# Ouverture d'un tube nommé

Demande d'ouverture d'un tube par un processus ayant les droits correspondants.

1. Si l'ouverture est bloquante → synchronisation (prise de rendez-vous)
   a. Une demande d'ouverture en lecture est bloquante s'il n'y a aucun écrivain.
   b. Une demande d'ouverture en écriture est bloquante s'il n'y a aucun lecteur.
2. Si l'ouverture est non bloquante
   a. Une demande d'ouverture en lecture réussit toujours.
      Les opérations de lecture ultérieures sont non bloquantes jusqu'à demande explicite du contraire.
   b. Une demande d'ouverture en écriture échoue s'il n'y a aucun lecteur.
   c. Une demande d'ouverture en écriture réussit s'il y a au moins un lecteur.
      Les opérations d'écriture ultérieures sont non bloquantes jusqu'à demande explicite du contraire.

```
mkfifo [p] [-m mode] pathname
```

**a** Ecrire un programme qui permet à deux processus sans lien de parenté d'échanger des données, entrées par l'utilisateur, via un tube nommé. Chaque processus affiche les données reçues.

```c
#include <stdio.h> // fprintf(), perror(), printf()
#include <stdlib.h> // exit()
#include <fcntl.h> // open()
#include <unistd.h> // write(), close()
#include <sys/stat.h> // mkfifo()
#include <string.h> // strlen()

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

	if(argc < 3){
	   fprintf(stderr, "Usage : %s FIFO_file_name Data_to_transfer \n", argv[0]);
	   exit(1);
	}

	/* int mkfifo(const char *path, mode_t mode);
	 * Return : 0 if OK, -1 on error
	 * 644 - 022 = 622 -> rw- -w- -w-
	 */
	int mkfifo_result = mkfifo(argv[1], 0666);
	if(mkfifo_result) {
	   perror("Function mkfifo() : ");
	   exit(1);
	}

	/* int open(const char *pathname, int flags [, mode_t mode]);
	 * Returns the file descriptor, error = a negative value is returned
	 * O_WRONLY open the file so that it is write only.
	 */
	int open_result = open(argv[1], O_WRONLY);
	if(open_result  < 0) {
	   perror("Function open() : ");
	   exit(1);
	}

	/* ssize_t write(int fd, const void *buf, size_t nbytes);
	 * Returns : number of bytes written if OK, -1 on error
	 */
	int write_result = write(open_result, argv[2], strlen(argv[2]));
	if(write_result < 0) {
	  perror("Function write() : ");
	  exit(1);
	}

	printf("Process was able to write to file \n");

	/* int close(int fd);
	 * Returns : 0 if OK, -1 on error
	 */
	int close_result = close(open_result);
	if(close_result < 0) {
	  perror("Function close() : ");
	  exit(1);
	}

	return 0;
}
```

```c
#include <stdio.h> // fprintf(), perror(), printf()
#include <stdlib.h> // exit()
#include <fcntl.h> // open()
#include <unistd.h> // read(), close()

#define MAX_BUFFER 25
int main(int argc, char *argv[]){
    if(argc < 2){
       fprintf(stderr,"Usage : %s FIFO_file_name \n", argv[0]);
       exit(1);
    }

    /* int open(const char *pathname, int flags [, mode_t mode]);
     * Returns the file descriptor, error = a negative value is returned
     * O_RDONLY Open the file so that it is read only.
     */
   int open_result = open(argv[1], O_RDONLY);
   if(open_result < 0) {
      perror("Function open() : ");
      exit(1);
   }

   /* ssize_t read(int fd, void *buf, size_t nbytes);
    * Returns : number of bytes read, 0 if end of file, -1 on error
    */
   char buffer[MAX_BUFFER];
   int read_result = read(open_result, &buffer, MAX_BUFFER);
   buffer[MAX_BUFFER] = '\0';

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

   printf("Process was able to read from file : %s\n", buffer );

   /* int close(int fd);
    * Returns : 0 if OK, -1 on error
    */
  int close_result = close(open_result);
  if(close_result < 0) {
     perror("Function close() : ");
     exit(1);
  }

  return 0;
}
```

```
[ij04115@saphyr:~/unix_tp4b]:dim. nov. 15$ nano question2_write.c
[ij04115@saphyr:~/unix_tp4b]:dim. nov. 15$ nano question2_read.c
[ij04115@saphyr:~/unix_tp4b]:dim. nov. 15$ gcc question2_write.c -o question2_write
[ij04115@saphyr:~/unix_tp4b]:dim. nov. 15$ gcc question2_read.c -o question2_read
```

```
[ij04115@saphyr:~/unix_tp4b]:lun. nov. 16$ ./question2_write
Usage : ./question2_write FIFO_file_name Data_to_transfer
[ij04115@saphyr:~/unix_tp4b]:lun. nov. 16$ ./question2_write fifo_name_test hello_world &
[1] 5617
[ij04115@saphyr:~/unix_tp4b]:lun. nov. 16$ ps aux | grep ij04115
root      5409  0.0  0.1  20472 10568 ?        Ss   nov.15   0:00 sshd: ij04115 [priv]
ij04115   5457  0.0  0.0  20472  6228 ?        S    nov.15   0:00 sshd: ij04115@pts/2
ij04115   5458  0.0  0.1  21140 10176 pts/2    Ss+  nov.15   0:00 -bash2
root      5489  0.0  0.1  20472 10452 ?        Ss   nov.15   0:00 sshd: ij04115 [priv]
ij04115   5537  0.0  0.0  20472  6272 ?        S    nov.15   0:00 sshd: ij04115@pts/3
ij04115   5538  0.0  0.1  21140 10388 pts/3    Ss   nov.15   0:00 -bash2
ij04115   5617  0.0  0.0   2260   692 pts/3    S    00:01   0:00 ./question2_write fifo_name_test hello_w
ij04115   5618  0.0  0.0  21232  8284 pts/3    R+   00:02   0:00 ps aux
ij04115   5619  0.0  0.0   5180   856 pts/3    S+   00:02   0:00 grep ij04115
[ij04115@saphyr:~/unix_tp4b]:lun. nov. 16$ ./question2_read
Usage : ./question2_read FIFO_file_name
[ij04115@saphyr:~/unix_tp4b]:lun. nov. 16$ ./question2_read fifo_name_test
Process was able to write to file
Process was able to read from file : hello_world
[1]+  Fini                    ./question2_write fifo_name_test hello_world
[ij04115@saphyr:~/unix_tp4b]:lun. nov. 16$
```

.