EPU - Informatique ROB4
Informatique Système
Les tubes

Miranda Coninx
Presented by
Ludovic Saint-Bauzel
ludovic.saint-bauzel@sorbonne-universite.fr

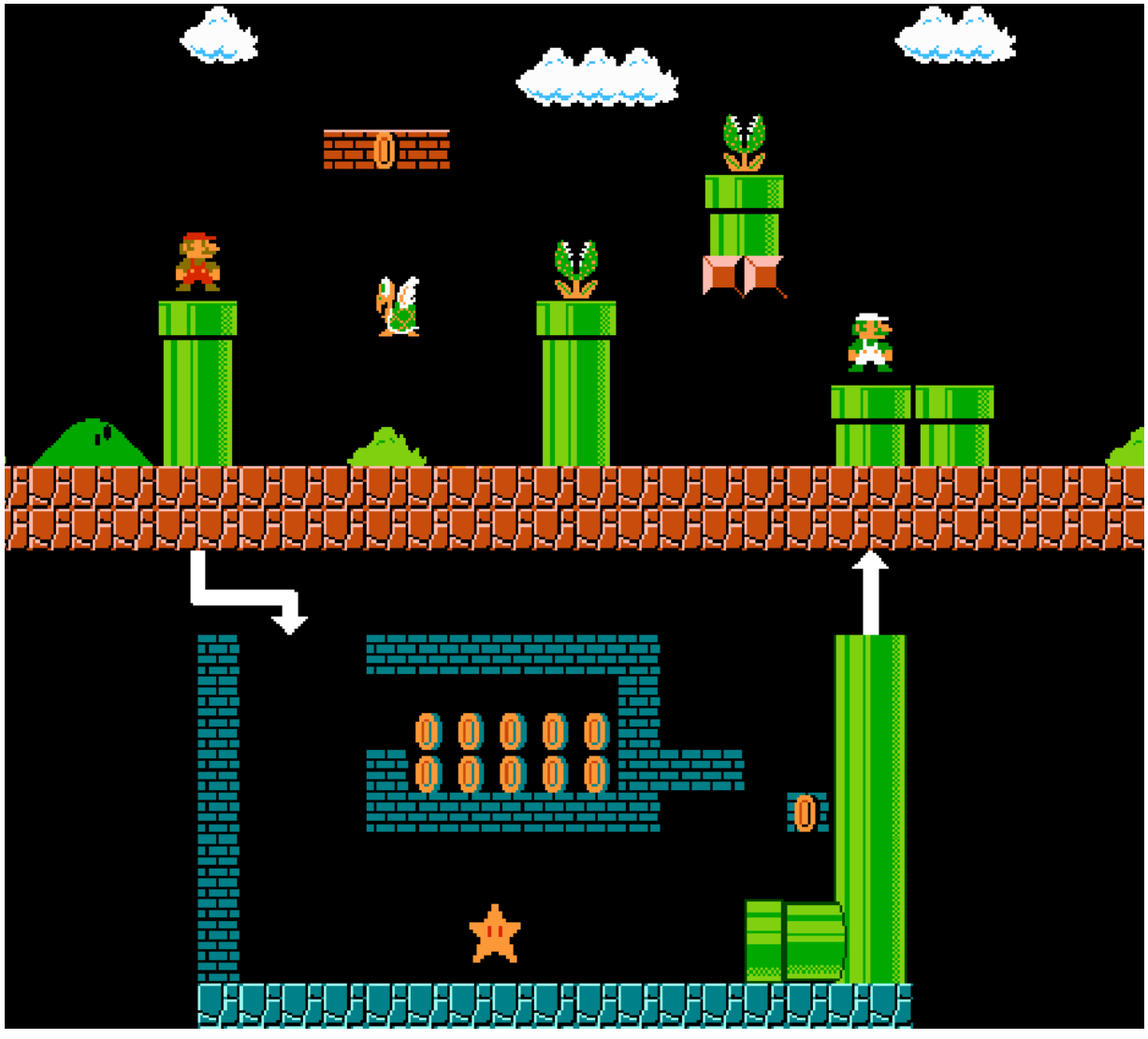
Sorbonne Université
Institut des Systèmes Intelligents et de Robotique (CNRS UMR 7222)

2023-2024

SORBONNE UNIVERSITÉ

ISIR
INSTITUT
DES SYSTÈMES
INTELLIGENTS
ET DE ROBOTIQUE

La communication par tubes - Pipes & FIFOs



Introduction

- Chaque processus dispose d'un espace d'adressage propre et indépendant, protégé des autres processus.
- Les processus peuvent avoir besoin de communiquer entre eux pour échanger des données.
- Linux offre différents outils de communication aux processus utilisateurs :
  - les tubes anonymes (pipes) ou nommés ; les files de messages (message queues) ; la mémoire partagée.
- **Les tubes sont gérés par le système de gestion de fichiers** (cf. cours 4).
- Les files de message et la mémoire partagée appartiennent à la famille des IPC (Inter Processus Communication).

IPC - Inter process communication

- Signals (cf. cours 6)
- Semaphores, mutexes (cf. cours 7-8), shared memory (cf. cours 5)
- Pipes & Fifos
- Sockets, messages (cf. cours 7-8)

Principe

- Un tube est une “tuyau” dans lequel un processus peut écrire des données qu'un autre processus peut lire.
- La communication dans un tube est unidirectionnelle et une fois choisie ne peut être changée.
- Un processus ne peut donc être à la fois écrivain et lecteur d'un même tube.
- Les tubes sont gérés au niveau du système de gestion de fichiers et correspondent à des fichiers au sein de ce dernier.
- Lors de la création d'un tube, deux descripteurs sont créés, permettant respectivement de lire et d'écrire dans le tube.
- Descripteurs $\rightarrow$ accès séquentiel.

Accès séquentiel : rappel du cours 4

- les enregistrements sont traités dans l'ordre où ils se trouvent dans le fichier (octet par octet) ;
- une opération de lecture délivre l'enregistrement courant et se positionne sur le suivant ;
- une opération d'écriture place le nouvel enregistrement en fin de fichier ;
- mode d'accès simple, pas forcément pratique, fichier accessible en lecture seule ou en écriture seule.

Gestion des données dans le tube

- La gestion des données dans le tube est liée au mode d'accès séquentiel.
- Gestion en flots d'octets : pas de préservation de la structure du message d'origine.
- Le lecteur reçoit d'abord les données les plus anciennes (FIFO).
- Les lectures sont destructives : les données lues disparaissent du tube.
- Un tube a une capacité finie qui est celle du tampon qui lui est alloué.
- Cette capacité varie selon les systèmes ; sur les OS Linux récents ($\geq$ 2.6.35) elle est ajustable.
- Un tube peut donc être plein et de fait amener les processus écrivains à s'endormir en attendant de pouvoir réaliser leur écriture : opération bloquante.

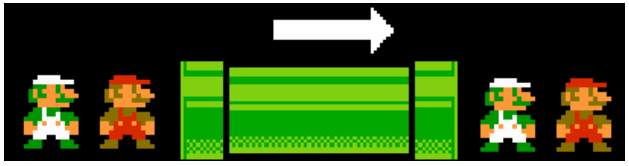
Les tubes et le shell : principe

- Les tubes de communication peuvent être utilisés au niveau de l'interpréteur de commande shell pour transmettre le résultat d'une commande à une autre qui interprète alors les données correspondantes comme données d'entrée.
- Un tube shell est symbolisé par le caractère | (alt Gr + 6).
- Par exemple, il n'existe pas de fonction shell permettant d'obtenir directement le nombre de fichiers dans un répertoire.
- Cette information peut être obtenue en comptant le nombre de lignes retourné par ls.
- La fonction shell wc −l permet de retourner le nombre de lignes d'un fichier.
- Il suffit donc d'envoyer la sortie de ls vers l'entrée de wc −l. Ceci peut être fait avec un pipe : ls | wc −l.

- **Ex : redirection for better reading large outputs**
  - `dmesg`
  - `dmesg | less`
  - `dmesg | grep disabled | grep ACPI`
- **Ex : redirection to isolate items of interest**

```
$ ls -l | grep Makefile
-rw-rw-r-- 1 icub icub  306 Sep 19 18:22 Makefile
```

Pipes & FIFOs



Pipe

- The basic mechanism of **IPC**
- **Read/write** : data written to the pipe by one process can be read by another process
- Data handled in FIFO order (first-in first-out)
- A pipe has **no name** : to use it, both ends (processes) must inherit it from the single process that created the pipe
- Anonymous, temporary connection

Fifo

- A `fifo` is a "**named pipe**"
- Similar to a pipe in its purpose, but substantially different, since it is a special file
- Its name is in fact a **path** within the file system
- Any process can open a `fifo` for reading/writing in the same way as an ordinary file (assuming file permissions allow it)

Pipes & FIFOs

Similarities

- Both are used to read/write datas (**one-way flow** of data)
- Once they are opened (different way !) the semantics fro read/write is the same
- Both have to be open at both ends simultaneously
- Reading from a pipe/fifo where nobody writes returns EOF
- Writing to a pipe/fifo without reader fails (SIGPIPE signal and error EPIPE are generated)
- The communication channel is a **byte stream**
- File positioning (`lseek`) is not allowed !
- Read/write operations are sequential and in *fifo* (**first-in-first-out**) order

Differences

	pipe	**fifo**
Name	no (anonymous)	yes (path)
Creation	`pipe`	`mkfifo`
Access	through file descriptors	through path in file system
Reading	`file descriptor [0]`	`O_RDONLY`
Writing	`file descriptor [1]`	`O_WRONLY`

Pipes (tubes anonymes)

Spécificités de tubes anonymes

- Les tubes anonymes sont gérés au niveau du système de gestion de fichiers et correspondent à des fichiers au sein de ce dernier.
- Ce fichiers ont la particularité d'être **sans nom**.
- Le lecteur reçoit d'abord les données les plus anciennes (FIFO).
- Cette absence de nom induit que ce type de tube ne peut être manipulé que par des processus ayant connaissance des deux descripteurs (lecture/écriture) associés au tube.
- Ce sont donc le processus créateur du tube et ses descendants qui prennent connaissance des descripteurs du tube par héritage des données de leur parent.

Création

- Un tube anonyme est créé par appel à la fonction `pipe()` dont les prototype est `int pipe (int desc[2]);` (défini dans `#include <unistd.h>`).
- `pipe()` retourne deux descripteurs placés dans `desc`.
- `desc[0]` : descripteur en lecture.
- `desc[1]` : descripteur en écriture.
- Les deux descripteurs sont respectivement associés à un fichier ouvert en lecture et à un fichier ouvert en écriture dans la table des fichiers ouverts.
- Un une entrée dans la table des fichiers est associée au tube mais aucun bloc de données ne lui correspond.
- Les données transitant dans un tube sont placées dans un tampon alloué dans la mémoire centrale ; sa taille peut être lue et ajustée par `fcntl()` avec les opérations `F_GETPIPE_SZ` et `F_SETPIPE_SZ`.
- Tout processus ayant connaissance du descripteur en lecture `desc[0]` d'un tube peut lire dans ce dernier (on peut donc avoir plusieurs processus lecteurs).
- Tout processus ayant connaissance du descripteur en écriture `desc[1]` d'un tube peut écrire dans ce dernier (on peut donc avoir plusieurs processus écrivains).
- En cas d'échec, `pipe()` renvoie -1 (0 sinon) et `errno` contient le code d'erreur et le message associé peut être récupéré via `perror()`.
- 3 erreurs possibles :
  - `EFAULT` : le tableau `desc` passé en paramètre n'est pas valide.
  - `EMFILE` : le nombre maximal de fichiers ouverts par le processus a été atteint.
  - `ENFILE` : le nombre maximal de fichiers ouverts par le système a été atteint.

Fermeture

- Un tube anonyme est considéré comme fermé lorsque tous les descripteurs en lecture et en écriture existants sur ce tube sont fermés.
- Un processus ferme un descripteur de tube `fd` en utilisant la fonction `int close(int fd);` .
- A un instant donné, le nombre de descripteurs ouverts en lecture détermine le nombre de lecteurs existants du tube. idem pour le nombre d'écrivains.
- Un descripteur fermé ne permet plus d'accéder au tube et ne peut pas être régénéré.

Lecture

- La lecture dans un tube anonyme s'effectue en mode binaire par le biais de la fonction `read()` dont le prototype est rappelé ici :
  `ssize_t read(int fd, void *buf, size_t count);`
- Cette fonction permet la lecture de `count` caractères (octets) depuis le tube dont le descripteur en lecture est `fd` qui sont placés à l'adresse `buf` .
- Elle retourne en résultat le nombre de caractères réellement lus :
  - si le tube n'est pas vide et contient `taille` caractères, `read` extrait du tube min(*taille*, *count*) caractères qui sont lus et placés à l'adresse `buf` ;
  - si le tube est vide et que le nombre d'écrivains est non nul, la lecture est bloquante et le processus lecteur est mis en sommeil jusqu'à ce que le tube ne soit plus vide ;
  - si le tube est vide et que le nombre d'écrivains est nul, la fin du fichier est atteinte et le nombre de caractères rendu est nul.
- L'opération de lecture peut être rendue non bloquante par un appel à la fonction de manipulation des descripteurs de fichier `fcntl ()` :
  `fcntl (desc [0], F_SETFL, O_NONBLOCK); .`
- Dans ce cas, le retour est immédiat si le tube est vide.

Ecriture

- L'écriture dans un tube anonyme s'effectue en mode binaire par le biais de la fonction `write()` dont le prototype est rappelé ici :
  `ssize_t write(int fd, const void *buf, size_t count);`
- Cette fonction permet l'écriture de `count` caractères (octets) placés à l'adresse `buf` dans le tube dont le descripteur en écriture est `fd` .
- Elle retourne en résultat le nombre de caractères réellement écrits :
  - si le nombre de lecteurs dans le tube est nul alors une erreur est générée et le signal SIGPIPE est délivré au processus écrivain qui se termine.
  - si le nombre de lecteurs dans le tube est non nul, l'opération d'écriture est bloquante jusqu'à ce que `count` caractères aient effectivement été écrits dans le tube, en attendant que des caractères aient été lus si le tube est plein ;
  - dans le cas où le nombre de caractères `count` à écrire dans le tube est $<$ à la constante symbolique `PIPE_BUF` définie dans `<limits.h>` (4096 octets), l'écriture des `count` caractères est *atomique* : les `count` caractères sont tous écrits les uns à la suite des autres dans le tube. Dans le cas contraire l'écriture des `count` caractères peut être arbitrairement découpée par le système.
- L'opération d'écriture peut elle aussi être rendue non bloquante.

Duplication of file descriptors

- We can duplicate file descriptors using the functions dup(), dup2() :

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

- dup uses the lowest-numbered unused descriptor for the new descriptor
- dup2 makes *newfd* be the copy of *oldfd*, closing *newfd* first if necessary,
  - if oldfd is not a valid file descriptor, then the call fails, and newfd is not closed
  - if oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does nothing, and returns newfd
  - mostly useful to redirect the standard input (STDIN_FILENO) or the standard output (STDOUT_FILENO)
- After a successful return from one of these system calls, the old and new file descriptors may be used interchangeably. They refer to the same open file description and thus share file offset and file status flags ; for example, if the file offset is modified by using lseek() on one of the descriptors, the offset is also changed for the other.
- The two descriptors do not share file descriptor flags (the close-on-exec flag).

Recap

```
#include <unistd.h>
int pipe(int filedes[2]);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- filedes[0] is for reading
- filedes[1] is for writing
- Creation : on success, 0 is returned. On error, -1 is returned (see errno for more)
- We typically fork after creating a pipe, because
  1. there is no reason to use a IPC primitive such as a pipe in a single process
  2. there is no way for any other process to access the pipe other than through the file descriptors copied in a fork

Mise en oeuvre

Cas typique

- Le processus parent ouvre un tube en utilisant la fonction `pipe()` ;
- Le processus parent `fork()` (création d'un enfant qui connaît donc les descripteurs du tube) ;
- Les descripteurs en lecture et écriture sont utilisables par les deux processus. Chacun ferme le descripteur qui lui est inutile : par exemple le parent ferme le descripteur en écriture et le enfant le descripteur en lecture.
- L'enfant envoie un message à son parent.
- le parent lit le message.

- Lorsqu'un tube est créé, le processus associé à cette création est potentiellement écrivain ou lecteur.
- La logique veut donc qu'on ne se déclare pas lecteur ou écrivain mais plutôt qu'on ferme avec `close()` le descripteur correspondant au mode de fonctionnement qui ne nous intéresse pas.
- La création d'une communication bi-directionnelle entre deux processus nécessite l'utilisation de deux tubes.

**** pipefork.c ****

```c
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/wait.h>
#include <time.h>

int main(int argc, char **argv)
{
  int fd[2];


  if (pipe(fd))                                /* create pipe              */
    fprintf(stderr, "pipe error\n");
  else if (fork())
  {                                                    /* parent: the reader  */
      int reception;
      close(fd[1]);                            /* close the write pipe */
      read(fd[0], &reception, sizeof (int));       /* read an integer         */
      printf("Parent: reading the value %d\n", reception);
  }
  else
  {
        int envoi;                                                 /* child: the
             writer      */
      close(fd[0]);
      srand(time(NULL));
      envoi = rand();                                /* generates integer */
      printf("Child: writing the value %d\n", envoi);
      write(fd[1], &envoi, sizeof (int));      /* writes the integer */
   }

  return 0;
}
```

**** pipefork.c ****

Output :

```
icub@eva:~/progsys/cours/c5/code$ ./pipe1
Child: writing the value 1786065064
Father: reading the value 1786065064
```

TEST

Write a program that creates a pipe, then forks to create a child. After the fork each process closes the descriptors that it does not need.

- The father process writes the string received as command-line parameter in the pipe (`argv[1]`)
- The child reads the string (note ! byte per byte !) from the pipe and print it on `stdout`

Example :

```
$ ./test1 "luke i am your father"
luke i am your father
```

TEST 2

- On souhaite écrire un programme affichant le nombre de fichiers dans un répertoire en utilisant deux processus (parent et enfant) dont l'un met en oeuvre wc −l (parent) et l'autre ls −l (enfant).
- Ces deux processus échangent les informations nécessaires au travers d'un tube.
- La destination naturelle des informations de sortie de ls −l est la sortie standard stdout tandis que wc −l prend naturellement ses données d'entrée depuis l'entrée standard stdin .
- Le problème est donc celui de la redirection de la sortie de ls −l vers l'entrée du tube et de l'entrée de wc −l vers la sortie du tube.
- Cette redirection peut être fait au moyen de la fonction dup2() dont le prototype est le suivant :
  `int dup2(int oldfd, int newfd);`
- dup2 transforme le descripteur newfd en une copie du descripteur oldfd en fermant auparavant newfd si besoin est.
- Ainsi dup2(desc[0], STDIN_FILENO); redirige l'entrée standard vers le tube en lecture (et dont le descripteur associé est desc[0]). Une fois cette opération effectuée, wc −l lira son entrée sur le tube.
- De même, dup2(desc[1], STDOUT_FILENO); redirige la sortie standard vers le tube en écriture (et dont le descripteur associé est desc[1]). Une fois cette opération effectuée, le résultat de l'appel à ls −l sera donc envoyé sur le tube.

**** fifodup.c ****

FIFOs (tubes nommés)

- Les tubes nommés sont également **gérés par le système de gestion de fichiers**.
- Le fichier associé possède un nom et le tube est donc accessible par tout processus connaissant ce nom et disposant des droits d'accès au tube.
- Les tubes nommés permettent donc à des processus sans lien de parenté de communiquer selon un mode séquentiel.
- De manière similaire au tube anonyme, un fichier est associé au tube nommé mais aucun bloc de données ne lui correspond.
- Les données transitant dans un tube nommé sont donc placées dans un tampon alloué dans la mémoire centrale.

Note : FIFO special files are indicated by `ls -l` with the file type `'p'`

```
icub@eva:/tmp/fifo$ ls
icub@eva:/tmp/fifo$ mkfifo myfifo
icub@eva:/tmp/fifo$ ls -l
total 0
prw-rw-r-- 1 icub icub 0 Sep 26 15:15 myfifo
```

Création

- Un tube nommé est créé par l'intermédiaire de la fonction `mkfifo()` dont le prototype est donné par :

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

- `pathname` correspond au chemin d'accès dans l'arborescence de fichiers pour le tube.
- `mode` permet de spécifier les droits d'accès associés au tube (cf. cours 4).
- `mkfifo` retourne 0 en cas de succès et -1 dans le cas contraire (EEXIST error if the FIFO already exists)

Ouverture

- L'ouverture d'un tube nommé se fait par l'intermédiaire de la fonction `open()` étudiée au cours 4.
- Le processus effectuant l'ouverture doit posséder les droits correspondants.
- `open()` renvoie un descripteur correspondant au mode d'ouverture spécifié.
- Par défaut, la fonction `open()` appliquée à un tube est bloquante. Ainsi, la demande d'ouverture en lecture est bloquante tant qu'il n'existe pas d'écrivain sur le tube.
- De manière similaire, la demande d'ouverture en écriture est bloquante tant qu'il n'existe pas de lecteur sur le tube.
- Ce mécanisme permet à deux processus de se synchroniser et d'établir un RDV en un point particulier de leur exécution.

Lecture / écriture

- La lecture et l'écriture dans un tube nommé s'effectue en utilisant les fonctions read et write (cf. "tubes anonymes").

Fermeture

- La fermeture se fait en utilisant close () .

Destruction

- La destruction se fait en utilisant unlink () dont le prototype est donné par :
  int unlink (**const char** *pathname);

Recap

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int mkfifo(const char *pathname, mode_t mode);
int open(const char *pathname, int flags);
int close(int fd);
int unlink(const char *pathname);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

- The `path` identifies the FIFO
- There is no need to fork : if the `path` is known, different programs can access the FIFO
  - $\mapsto$ see next example on client/server !

**** fifocs.h ****

```
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <sys/wait.h>
6  #include <sys/errno.h>
7  #include <stdio.h>
8  #include <time.h>
9
10 extern int errno;
11
12 #define FIFO1 "/tmp/my_fifo1"
13 #define FIFO2 "/tmp/my_fifo2"
14
15 #define PERMISSIONS 0666
```

**** fifo_server.c ****

```c
// use FIFO (not pipes) to implement a simple client-server model
// the server
#include "fifocs.h"

int main(int argc, char **argv)
{
    int readfd, writefd;

    if( (mkfifo(FIFO1,PERMISSIONS)<0) && (errno != EEXIST) )
    {
        printf("server: can't create FIFO1 to read: %s \n", FIFO1); return 1;
    }

    if( (mkfifo(FIFO2,PERMISSIONS)<0) && (errno != EEXIST) )
    {
        unlink(FIFO1);
        printf("server: can't create FIFO2 to write: %s \n", FIFO2); return 1;
    }

    if ( (readfd = open(FIFO1, 0)) < 0)
        { printf("server: can't open FIFO1 to read \n"); return 1; }
    if ( (writefd = open(FIFO2, 1)) < 0)
        { printf("server: can't open FIFO2 to write \n"); return 1; }

    /* server(readfd, writefd); */

    // just to give the time to the client to test the fifos
    sleep(1000);

    close(readfd);
    close(writefd);
    return 0;
}
```

**** fifo_client.c ****

```
1  // use FIFO (not pipes) to implement a simple client-server model
2  // the client
3  #include "fifocs.h"
4
5  int main(int argc, char **argv)
6  {
7      int readfd, writefd;
8
9      // open the fifos: assume the server already created them
10
11     if ( (writefd = open(FIFO1, 1)) < 0)
12       { printf("client: can't open FIFO1 to write\n"); return 1; }
13     if ( (readfd = open(FIFO2, 0)) < 0)
14       { printf("client: can't open FIFO2 to read\n"); return 1; }
15
16     /* client(readfd, writefd); */
17
18     close(readfd);
19     close(writefd);
20
21     // now delete the fifos since we're finished
22
23     if (unlink(FIFO1) < 0)
24       { printf("client: can't unlink FIFO1"); return 1; }
25     if (unlink(FIFO2) < 0)
26       { printf("client: can't unlink FIFO2"); return 1; }
27
28     return 0;
29 }
```

- Since the server creates the fifos, once we launch the server :

  ```
  $ ./fifo_server
  ```

  we can see the fifos by checking their path :

  ```
  icub@eva:/tmp$ ls -l
  prw-rw-r-- 1 icub    icub       0 Sep 26 16:00 fifo1.txt
  prw-rw-r-- 1 icub    icub       0 Sep 26 16:00 fifo2.txt
  ```

- Note that in this example, the client assumes the fifos have already been created. This is what happens if we launch the client first :

  ```
  $ ./fifo_client
  client: can't open FIFO1 to write
  ```

- Beware ! If we accidentally press CTRL+C on the server, fifos are not destroyed properly... check `/tmp/` to see if they were closed properly.

TEST

Modify the client-server example so that client and server do really something, e.g. exchange data :

- the client connects to the server, sends an integer, and waits for the server to reply with another integer
- the server wait for a client connection, receives an integer, perform some processing on this integer (for example, add $+10$), then send the result to the client

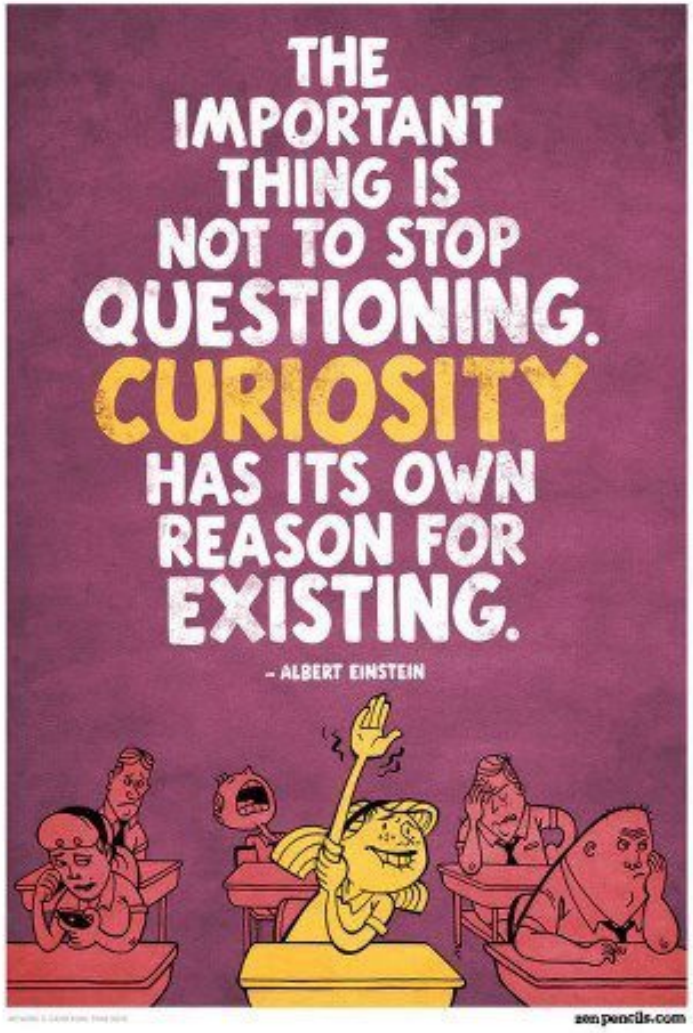
The server :

```
$ ./fifo_server
server: wait for incoming connection
server: read 42, sent 52
```

The client :

```
$ ./fifo_client
client: sent 42
client: wait for server reply
client: read 52
```

Questions ?


THE IMPORTANT THING IS NOT TO STOP QUESTIONING. CURIOSITY HAS ITS OWN REASON FOR EXISTING.
– ALBERT EINSTEIN
zenpencils.com