EPU - Informatique ROB4
Informatique Système
Les tubes

Sovannara Hak, Jean-Baptiste Mouret
hak@isir.upmc.fr

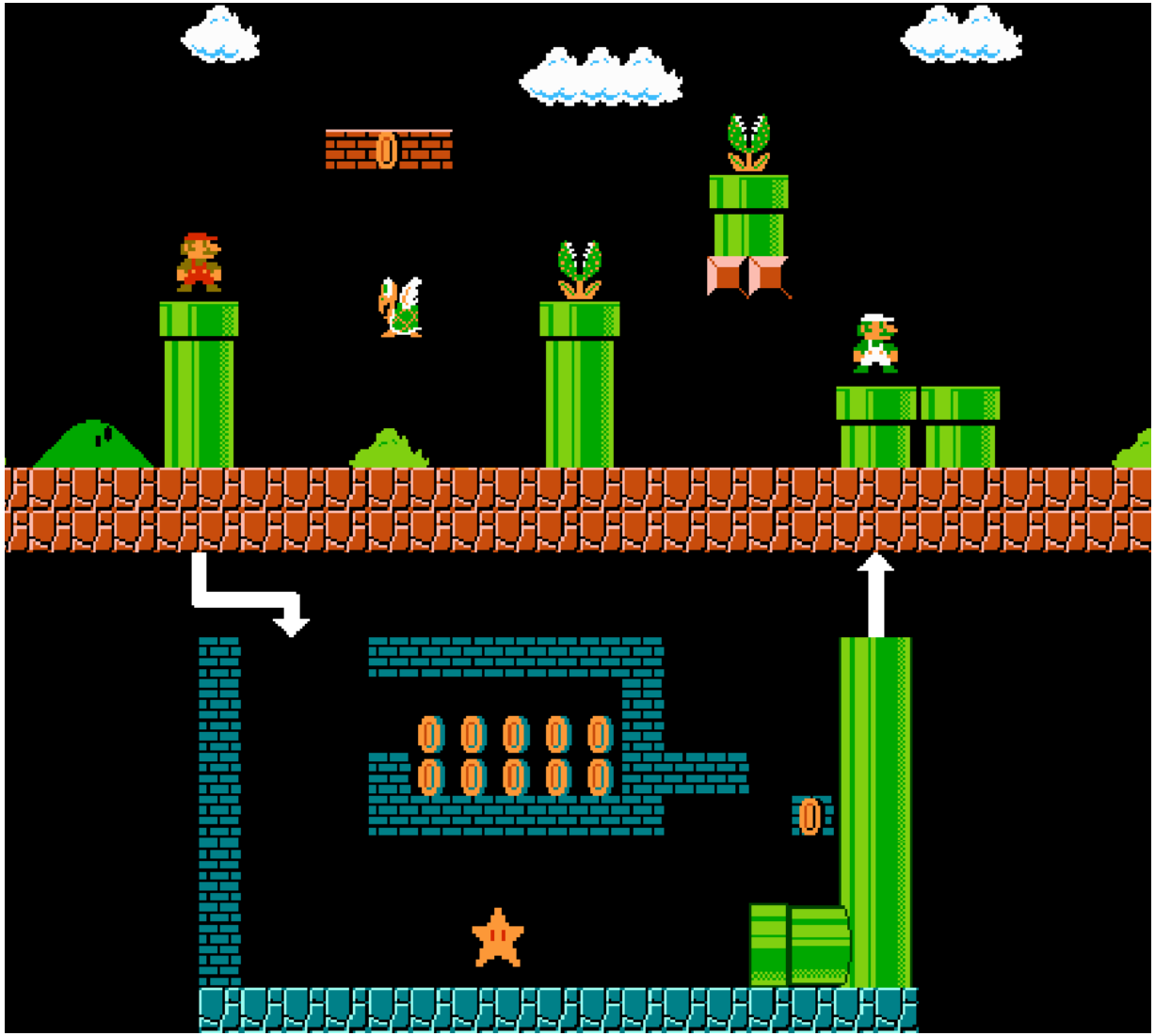
Université Pierre et Marie Curie
Institut des Systèmes Intelligents et de Robotique (CNRS UMR 7222)

2014-2015

Plan de ce cours

Rappel du plan

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 7)
- Fork (cf. cours 2)
- Semaphores, mutex, mémoire partagée (cf. cours 3)
- Pipes & Fifos
- Sockets, messages

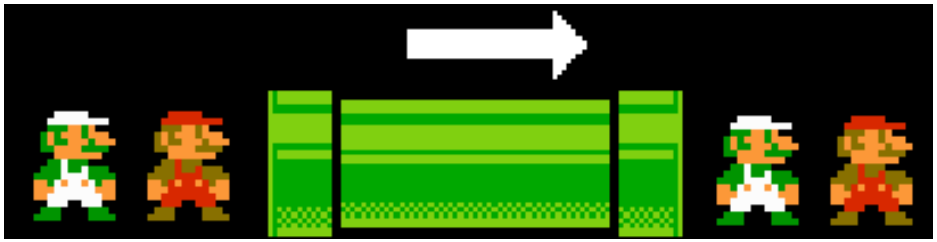
Principe

- 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 est définie par la constante symbolique `PIPE_BUF` définie dans `<limits.h>` .
- 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.

Rappel du plan

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 |
- 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 -l`.
- La fonction shell `wc -l` permet de retourner le nombre de lignes d'un fichier.
- Il suffit donc d'envoyer la sortie de `ls -l` vers l'entrée de `wc -l`. Ceci peut être fait avec un pipe : `ls -l | wc -l`.

Ex : autres exemples pour lire plus facilement les sorties longues ou filtrer

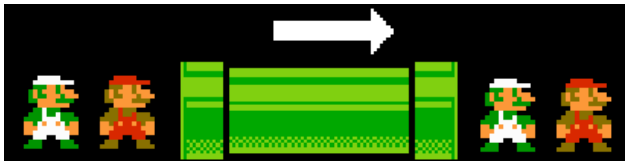
- dmesg : Affiche message du buffer de message du noyau
- less : lecture de fichier
- grep : recherche de chaine de caractères

- `dmesg`
- `dmesg | less`
- `dmesg | grep disabled | grep IO`

Pipes & FIFOs

Pipe (tube)

- **Lecture/écriture** : les données écrites dans un pipe par un processus peuvent être lues par un autre processus (ordre fifo)
- Un pipe n'est pas nommé
- 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 père.



Fifo

- Une `fifo` est un "**pipe nommé**"
- Similaire aux pipe (c'est un cas spécial)
- Son nom est en fait un **chemin** dans le système de fichier
- N'importe quel processus peut accéder au `fifo` pour lire/écrire comme un fichier ordinaire (avec les droits adéquats)

Pipes & FIFOs

Similitudes :

- Les deux servent à lire/écrire des données (**sens unique**)
- Une fois ouvert, les opérations de lecture/écriture sont les même
- Les deux bouts doivent être ouvert en même temps
- La lecture d'un pipe/fifo où rien n'a été écrit renvoie EOF
- Écrire dans un pipe/fifo sans lecteur échoue (signel SIGPIPE et erreur EPIPE)
- La lecture/écriture se fait par octet
- Changement de position (`lseek`) non autorisé
- Lecture/écriture séquentielle dans l'ordre *fifo* (**first-in-first-out**)

Différences :

	pipe	**fifo**
Nom	non (anonyme)	oui (chemin)
Création	`pipe`	`mkfifo`
Accès	via file descriptors	via chemin dans le sytème de fichiers
Lecture	`file descriptor [0]`	`O_RDONLY`
Écriture	`file descriptor [1]`	`O_WRONLY`

Rappel du plan

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()` crée 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 fichier physique est associé 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.
- 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 dans un même tube).
- 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 dans un même tube).
- 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.

Écriture

- 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 ;
  - Dans le cas où le nombre de caractères `count` à écrire dans le tube est $<$ à la constante symbolique `PIPE_BUF` (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 où le nombre de caractères `count` à écrire dans le tube est $>$ à la constante symbolique `PIPE_BUF` , l'écriture des `count` caractères peut être arbitrairement découpée par le système.
- L'opération d'écriture peut elle aussi être rendue non bloquante.

Duplication de file descriptors

- On peut dupliquer des file descriptors en utilisant les fonctions dup(), dup2() :

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

- dup utilise le plus petit numéro non utilisé d'un descripteur pour le nouveau
- dup2 transforme *newfd* en une copie de *oldfd*, en fermant d'abord *newfd* si besoin
  - si oldfd n'est pas un file descriptor valide, échec, et newfd n'est pas fermé
  - si oldfd est valide, et newfd a la même valeur que oldfd, alors dup2() ne fait rien et renvoie newfd
- Après un appel réussi, le nouvel et l'ancien file descriptors peuvent être utilisés de manière interchangeable. Ils font référence à la même description de fichier et donc partagent offset et status ; par exemple, si l'offset est modifié en utilisant lseek() sur l'un, l'offset sera modifié pour l'autre.
- Les attributs des file descriptors ne sont cependant pas partagé (man fcntl)

Rappel du plan

Résumé

```
#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] pour lecture
- filedes[1] pour l'écriture
- Création : retourne 0 si succès, -1 si erreur (utiliser errno pour en savoir plus)
- Typiquement, un fork est exécuté après la création d'un pipe
  1. pas d'intêret d'utiliser des primitives IPC comme des pipes s'il n'y a qu'un processus
  2. pas d'autre moyen pour un autre processus d'accéder au pipe qu'à travers le file descriptor copié dans le fork

Mise en oeuvre

Cas typique

- Le processus père ouvre un tube en utilisant la fonction `pipe()` ;
- Le processus père `fork()` (création d'un fils qui connaît donc les descripteurs du tube) ;
- Le processus associé à cette création est potentiellement écrivain ou lecteur.
- Les descripteurs en lecture et écriture sont utilisables par les deux processus. Chacun ferme le descripteur qui lui est inutile : par exemple le père se déclare lecteur en fermant le descripteur en écriture et le fils se déclare écrivain en fermant le descripteur en lecture.
- Lecture/écriture
- 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];
  int i;

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

  return 0;
}
```

**** pipefork.c ****

Output :

```
shak@samaxe:~/cours/c5/code$ ./pipe1
Child: writing the value 1647264414
Father: reading the value 1647264414
```

Rappel du plan

TEST

Écrire un programme qui crée un pipe, puis fork pour créer un enfant. Après le fork, chaque processus ferme le descripteur qu'il n'utilise pas.

- Le père écrit la chaine de caractère passé en argument dans la ligne de commande (`argv[1]`)
- L'enfant lit la chaine (octet par octet) depuis le pipe et l'affiche sur `stdout`

Exemple :

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

Rappel du plan

TEST 2

- On souhaite écrire un programme affichant le nombre de fichiers dans un répertoire en utilisant deux processus (père et fils) dont l'un met en oeuvre wc −l (père) et l'autre ls −l (fils).
- 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.

Rappel du plan

- 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 nommé.
- 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 physique 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 : les fichiers spéciaux FIFO sont indiqué par `ls -l` avec le type `'p'`

```
shak@samaxe:/tmp/fifo$ ls -l
total 0
prw-rw-r-- 1 shak shak 0 Dec 16 17:49 mon_fifo
```

Rappel du plan

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)
- man 3 mkfifo

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 sur le tube.
- `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);`

Rappel du plan

Résumé

```
#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);
```

- path identifie le FIFO
- Pas besoin de fork : si path est connu, différent programmes peuvent accéder au FIFO
  - $\mapsto$ voir le prochain exemple sur des client/server !

**** fifocs.h ****

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

extern int errno;

#define FIFO1 "/tmp/fifo1.txt"
#define FIFO2 "/tmp/fifo2.txt"

#define PERMISSIONS 0666
```

**** fifo_server.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 ****

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

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

    // open the fifos: assume the server already created them

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

    /* client(readfd, writefd); */

    close(readfd);
    close(writefd);

    // now delete the fifos since we're finished

    if (unlink(FIFO1) < 0)
      { printf("client: can't unlink FIFO1"); return 1; }
    if (unlink(FIFO2) < 0)
      { printf("client: can't unlink FIFO2"); return 1; }

    return 0;
}
```

- Puisque le serveur crée les fifos, une fois le serveur lancé :

  ```
  $ ./fifo_server
  ```

  on peut voir les fifos en regardant leurs chemins :

  ```
  shak@samaxe:/tmp$ ls -l
  prw-rw-r-- 1 shak    shak        0 Dec 16 18:24 fifo1.txt
  prw-rw-r-- 1 shak    shak        0 Dec 16 18:24 fifo2.txt
  ```

- Dans cet exemple, le client suppose que les fifos sont déjà créés. Si on lance le client en premier :

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

- Attention ! Si on tue le serveur avec CTRL+C par exemple, les fifos ne sont pas détruites correctement...

Rappel du plan

TEST

Modifier l'exemple client/server pour que le client et le serveur fassent vraiment quelque chose, par exemple échanger des données :

- le client se connecte au serveur, envoie un entier et attend que le serveur lui réponde avec un autre entier
- le serveur attend une connexion client, reçoit l'entier, exécute un calcul avec cet entier (ajouter 665 par exemple), puis envoie le résultat au client

Le serveur :

```
$ ./fifo_server
server: wait for incoming connection
server: read 1, sent 666
```

The client :

```
$ ./fifo_client
client: sent 1
client: wait for server reply
client: read 666
```

Questions ?