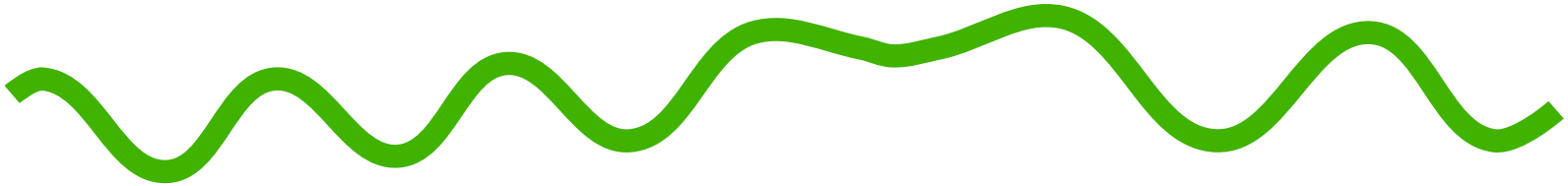


Programmation des systèmes

Communication par tubes

Philippe MARQUET

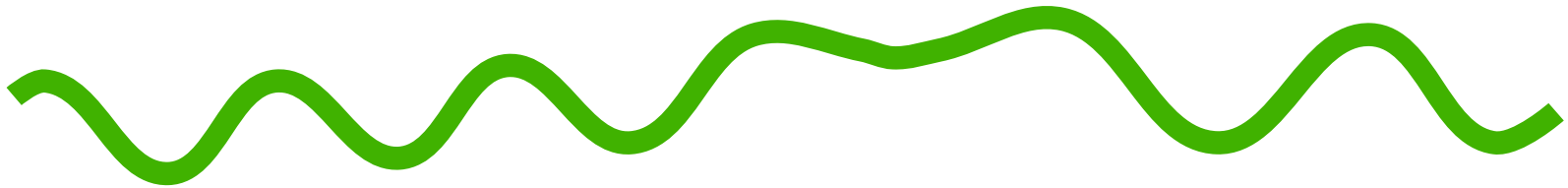
Philippe.Marquet@lifl.fr

Laboratoire d'informatique fondamentale de Lille

Université des sciences et technologies de Lille

Licence d'informatique de Lille

mars 2005



- La dernière version de ce cours est accessible à
  `www.lifl.fr/~marquet/cnl/pds/`
- `$Id: pipe.tex,v 1.11 2012/11/27 23:17:54 marquet Exp $`

Références & remerciements

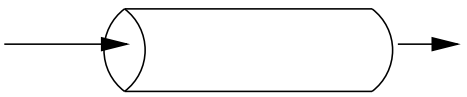
- *Unix, programmation et communication*
  Jean-Marie Rifflet et Jean-Baptiste Yunès
  Dunod, 2003
- *The Single Unix Specification*
  The Open Group
  `www.unix.org/single_unix_specification/`
- `man` section 2

Communications inter-processus

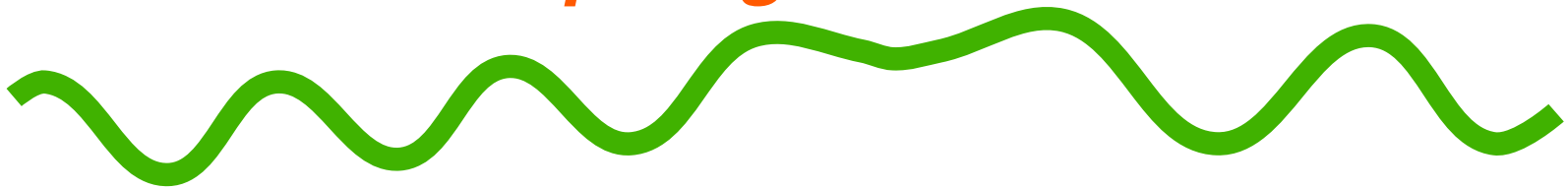
- Nécessité de communications inter-processus
  - coopération entre processus
  - combinaison de résultats de deux, plusieurs processus
- Réalisation de communications possibles
  - `exit()` & `wait()` : valeur de retour du fils vers son père
  - `kill()` : signalisation d'un événement
  - `write()` & `read()` via un fichier
- Inconvénients d'un fichier intermédiaire
  - pas de signalisation de la fin d'écriture du rédacteur à l'écrivain
  - suppression manuelle du fichier après la communication

- Tubes (*pipes*)
  - moyen de communication entre lecteur(s) et écrivain(s)
  - vont permettre de palier ces inconvénients

```
cat file | more
```

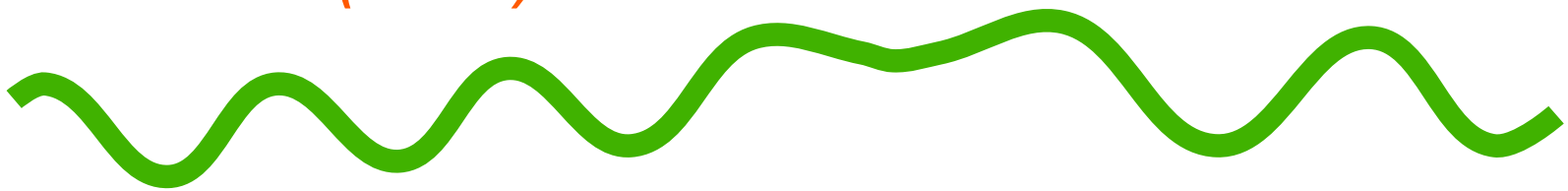


Caractéristiques générales des tubes



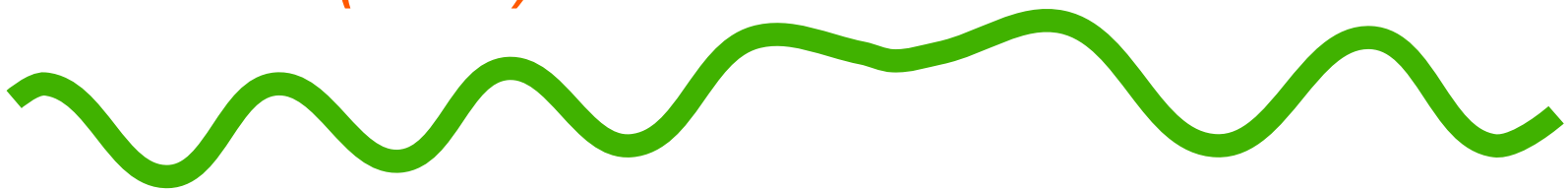
- Deux types de tubes
  - tubes anonymes, *pipe*
  - tubes nommés, *fifo*
  - caractéristiques communes
- Objets du système de fichiers
  - accès via un descripteur de fichier
  - opérations `read()` et `write()`
  - redirection des entrées/sorties standard depuis/vers un tube
  - etc.
- Communication unidirectionnelle
  - un descripteur pour écrire (mode `O_WRONLY`)
  - un descripteur pour lire à l'autre bout (mode `O_RDONLY`)

Caractéristiques générales des tubes *(cont'd)*



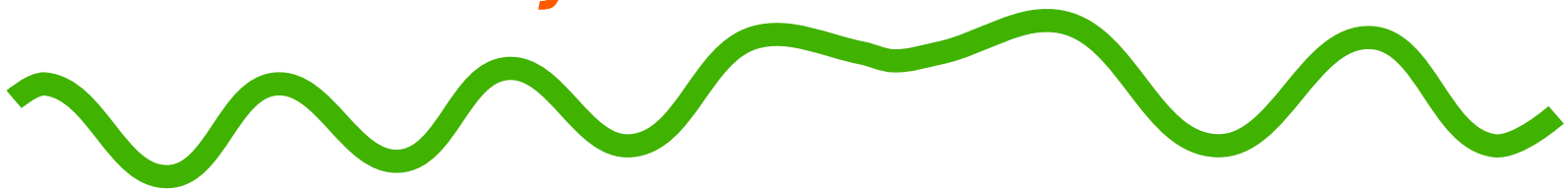
- Communication d'un flot continu de caractères
  - un caractère lu depuis un tube est extrait du tube
  - communication en mode flot (opposé à datagramme)
    - opérations de lecture indépendantes des opérations d'écriture
    - ex : écrire 5 caractères, en lire 2, en écrire 3, en lire 6, etc.
- Communication en mode *fifo*
  - *first-in, first-out*
  - premier caractère écrit, premier caractère lu
  - pas de possibilité de positionnement dans le tube (`lseek()`)
- Taille bornée
  - un tube peut donc être plein
  - une écriture peut donc être bloquante

Caractéristiques générales des tubes (cont'd)



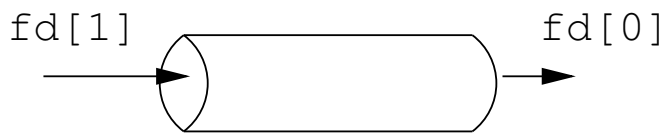
- Nombre de lecteurs
  - nombre de descripteurs associés à la lecture depuis le tube
  - si nul, le tube inutilisable, les écrivains pourront être prévenus
- Nombre d'écrivains
  - nombre de descripteurs associés à l'écriture dans le tube
  - si nul, le tube vide est inutilisable, les lecteurs pourront être prévenus

Tubes anonymes



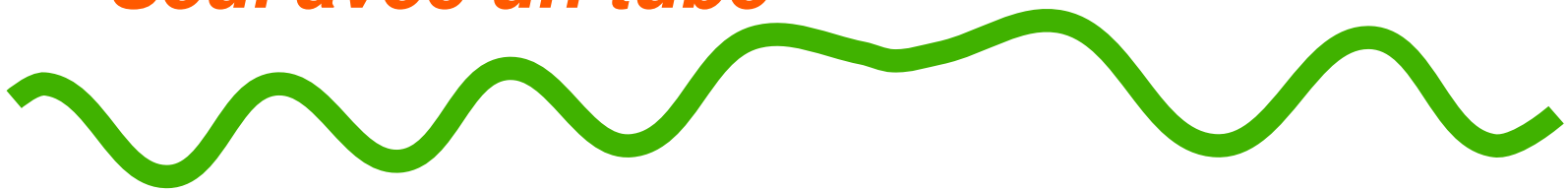
- Un fichier sans nom
  - pas d'entrée dans le système de fichier
  - on ne peut utiliser `open()`
  - création par une opération ad hoc
  - destruction automatique à la fin de l'utilisation
- Primitive de création d'un tube
  - `#include <unistd.h>`

    `int pipe(int *fd)`
  - retourne deux descripteurs
    - `fd[1]` pour l'écriture
    - `fd[0]` pour la lecture



- Connaissance du tube
  - avoir réalisé l'opération `pipe()`
  - perte d'un descripteur (fermeture) $\Rightarrow$ accès impossible
  - héritage des descripteurs lors de `fork()`

Seul avec un tube



- Un processus unique écrit et lit dans un tube
  - intérêt limité...

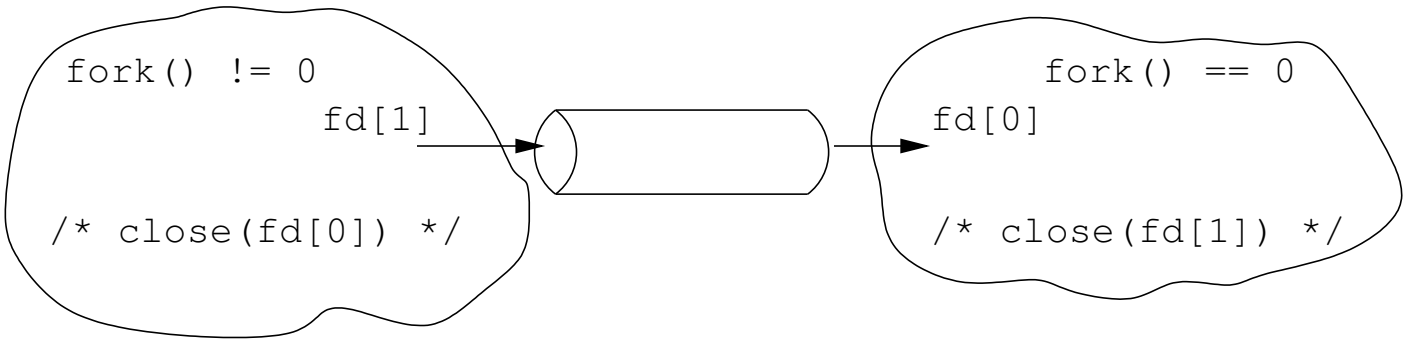
pp-solo.c

```
#define BSIZE   1024

int
main (int argc, char *argv[])
{
    int fds[2];
    char buf[BSIZE];
    int n;

    n = atol(argv[1]);
    assert(n<=1024);

    pipe(fds);

    for(;;) {
        putchar('*'); fflush(stdout);

        write(fds[1], buf, n);
        read(fds[0], buf, 256);
    }

    exit(EXIT_SUCCESS);
}
```

```
% ./pp-solo 256
****************************************************************
****************************************************************
****************************************************************
****************************^C

% ./pp-solo 0
*

% ./pp-solo 1024
**********

% ./pp-solo 2
****************************************************************
****************************************************************
****************************************************************
****************************^C
```

Partage d'un tube

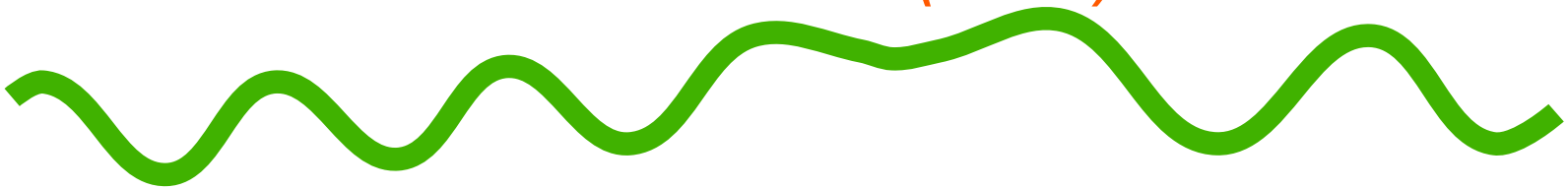
- Utilisation typique d'un tube
  - un processus écrivain
  - un processus lecteur
- Mécanisme de partage
  - un tube est créé par un processus père
  - processus père crée un processus fils
  - le père et le fils partagent les descripteurs d'accès au tube
  - chacun va utiliser un « bout » du tube
  - chacun détruit le descripteur inutile



Lecture dans un tube

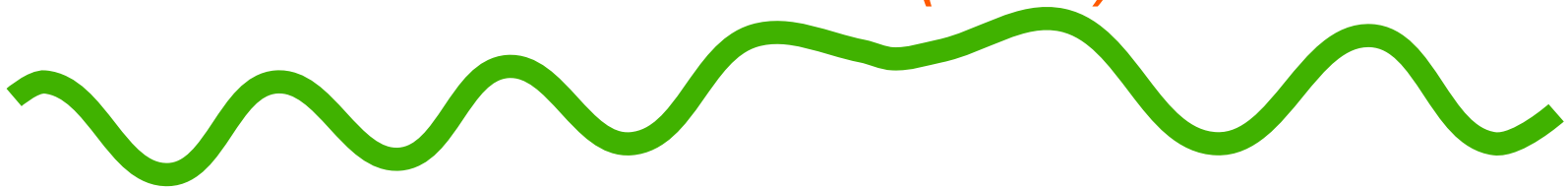
- Primitive `read()`
  - `ssize_t read(int fd, void *buf, size_t nbyte);`
- Demande de lecture
  - d'au plus `nbyte` caractères
- Si le tube n'est pas vide
  - contient `tbyte` caractères
  - extrait `min(tbyte,nbyte)` caractères
  - écrit à l'adresse `buf`
  - retourne cette valeur `min(tbyte,nbyte)`
- Si le tube est vide
  - si le nombre d'écrivains est nul
    - il n'y aura jamais plus rien à lire
    - c'est la « fin de fichier »
    - retourne 0
  - si le nombre d'écrivains est non nul
    - processus bloqué jusqu'à écriture par un écrivain

Lecture dans un tube (cont'd)



- Réveil d'un processus bloqué en lecture sur un tube vide
  - si un écrivain réalise une écriture
    - `read()` retourne une valeur non nulle
  - si le dernier écrivain ferme son descripteur d'accès au tube
    - explicitement par un `close()`
    - automatiquement à sa terminaison
    - `read()` retourne 0
- Bonne pratique
  - *systématiquement fermer, au plus tôt, tous les descripteurs non utilisés*
  - garder un descripteur en écriture sur un tube
    - potentiellement bloquer un processus

Lecture dans un tube *(cont'd)*



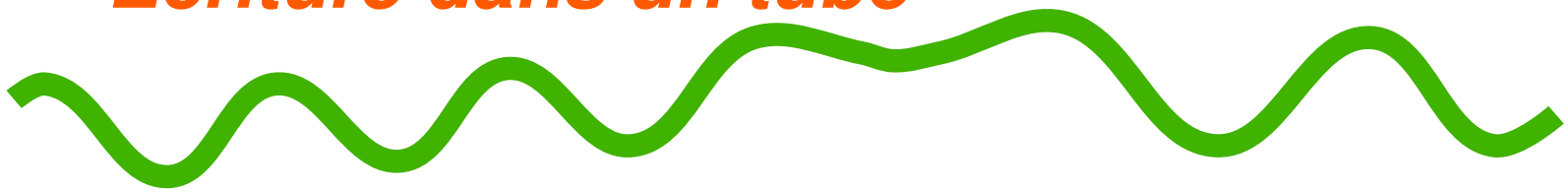
- Attention aux interblocages, étreintes fatales (*deadlocks*)

pp-dlck.c

```
int
main ()
{
    int fdts[2],                /* to son  */
        fdfs[2];                /* from son */
    char bufr[BSIZE], bufw[BSIZE];

    pipe(fdts);
    pipe(fdfs);

    if(fork()) {                /* pere */
        read(fdfs[0], bufr, 1);
        write(fdts[1], bufw, 1);
    } else {                    /* fils */
        read(fdts[0], bufr, 1);
        write(fdfs[1], bufw, 1);
    }

    prinf("bye\n");
    exit(EXIT_SUCCESS);
}
```

```
% ./pp-dlck
```

Écriture dans un tube



- Primitive `write()`
  - `ssize_t write(int fd, const void *buf, size_t nbyte);`
- Demande d'écriture
  - de `nbyte` caractères
- Écriture atomique
  - si `nbyte` inférieur à `PIPE_BUF`
  - sinon découpage par le système en plusieurs écritures... non portable.
- Si le nombre de lecteurs est non nul
  - écriture atomique, donc
  - bloquant tant qu'il n'y a pas la place pour écrire les `nbyte` caractères

Écriture dans un tube *(cont'd)*

pp-size.c

```
int
main ()
{
    int fd[2];
    int pipe_size;
    struct sigaction sa;

    /* install sigalarm handler  */
    sa.sa_handler = alarm_hndlr;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGALRM, &sa, NULL);

    /* pipe charcateristics */
    pipe(fd);

    pipe_size = fpathconf(fd[0],
                          _PC_PIPE_BUF);
    printf("PIPE_BUF: %d\n", pipe_size);

    /* write to pipe */
    for(;;) {
        alarm(3);
        write(fd[1], "a", 1);
        alarm(0);           /* reset alarm */

        if ((++count % 1024) == 0)
            printf("%d bytes in pipe\n", count);
    }
}
```

```
static int count = 0;

static void
alarm_hndlr(int signum)
{
    printf("write() blocks after %d bytes\n",
           count);
    exit(EXIT_SUCCESS);
}
```

```
% ./pp-size
PIPE_BUF: 4096
1024 bytes in pipe
2048 bytes in pipe
3072 bytes in pipe
4096 bytes in pipe
write() blocks after 4096 bytes
```

Écriture dans un tube (cont'd)

- Si le nombre de lecteurs est nul
  - les caractères écrits dans le tube ne pourront plus être lus
  - information de l'écrivain
  - signal `SIGPIPE`
  - comportement par défaut = terminaison

pp-sig.c

```
static void
sigpipe_hndlr(int signum)
{
    printf("signal SIGPIPE recu\n");
}

int
main ()
{
    int fd[2];
    int nbytes;
    struct sigaction sa;

    sa.sa_handler = sigpipe_hndlr;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGPIPE, &sa, NULL);

    pipe(fd);
    close(fd[0]);

    nbytes = write(fd[1], "a", 1);
    if (nbytes == -1)
        perror("write fd[1]");
    else
        printf("%d bytes written\n", nbytes);

    exit(EXIT_SUCCESS);
}
```

```
% ./pp-sig
signal SIGPIPE recu
write fd[1]: Broken pipe
```

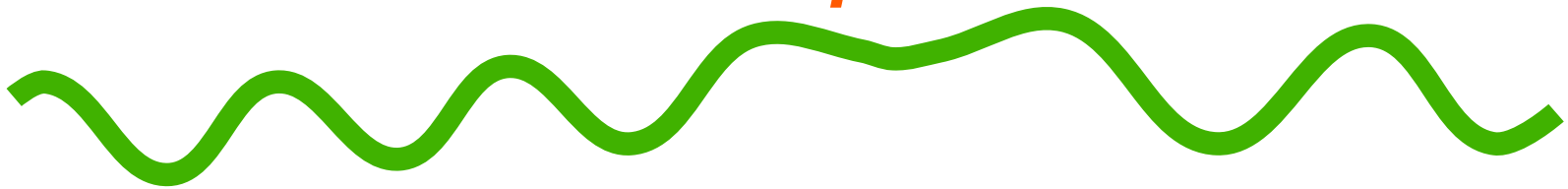
Fermeture d'un tube

- Primitive `close()`

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

- Fermer tous les descripteurs non utilisés
  - évite les situations de blocage
- Fermeture d'un tube
  - libère le descripteur
  - pour le descripteur d'écriture, agit comme une fin de fichier

Redirection vers/depuis un tube



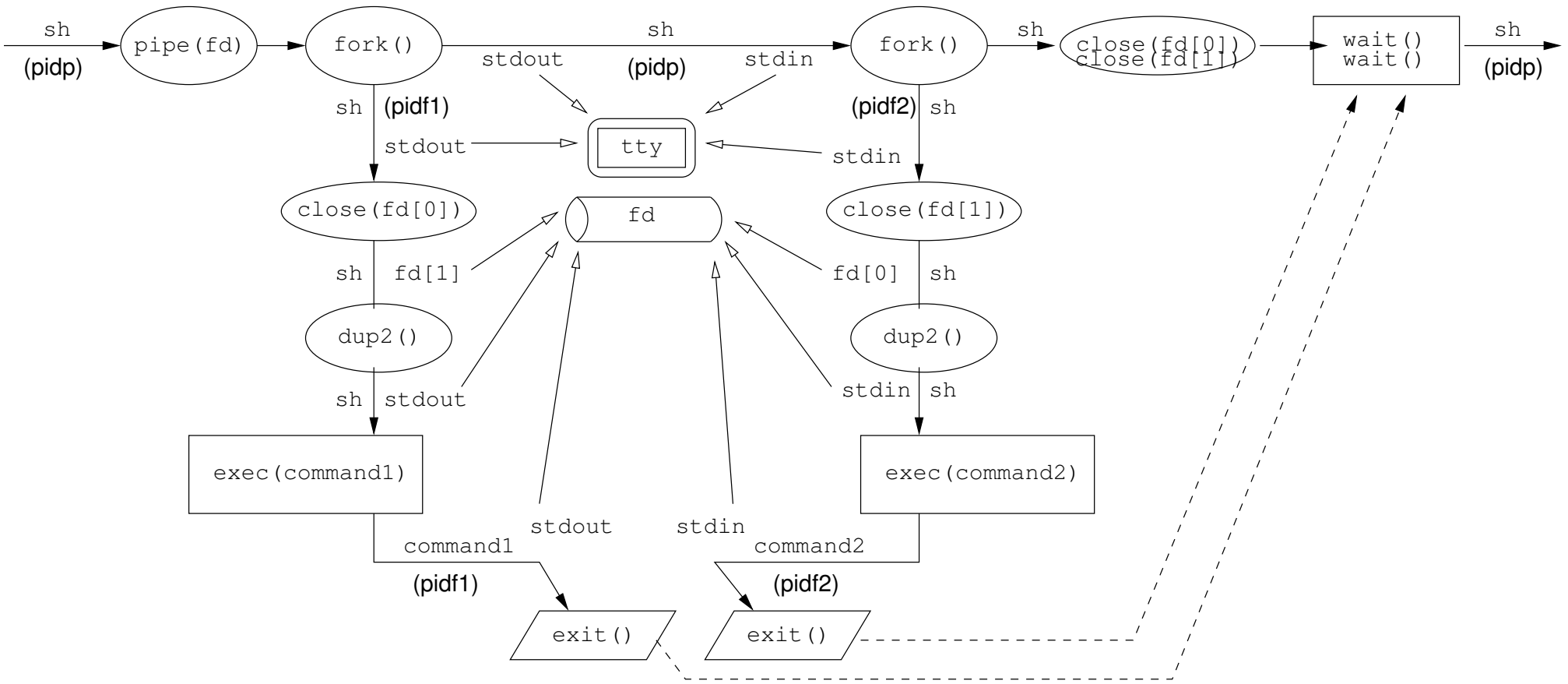
- Motivation première des tubes, combiner
  - connexion de la sortie standard d'un processus vers un tube
  - connexion de l'entrée standard d'un processus depuis un tube
- Tube liant deux commandes
  - issues d'un même père, le shell

    ```
    % command1 | command2
    ```
- Création du tube
  - par le shell
- Réalisation des redirections
  - `command1` écrit sur sa sortie standard
  - que le shell a préalablement redirigé vers le tube
  - `command2` lit sur son entrée standard
  - que le shell a préalablement redirigé depuis le tube
- Destruction du tube
  - automatique, par le système
  - à la clôture des descripteurs

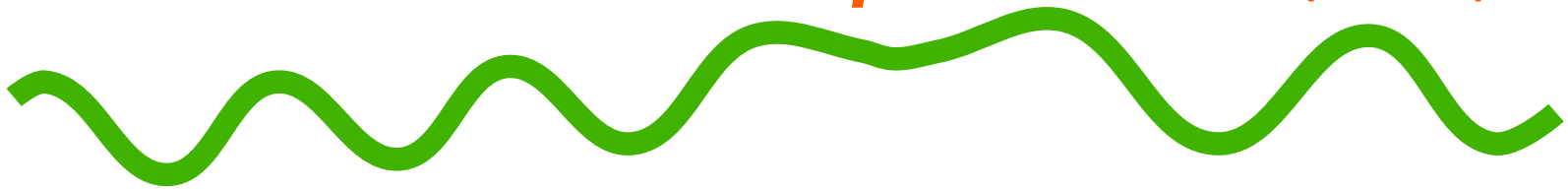
Redirection : l'exemple du shell

- Exécution d'une commande `command1` dont la sortie est redirigée vers l'entrée de la commande `command2`

```
% command1 | command2
```

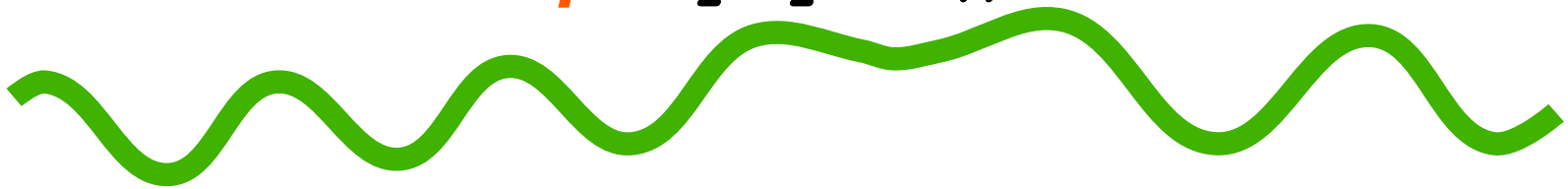


Redirection : l'exemple du shell *(cont'd)*



- Terminaison
  - en l'absence des `close(fd[1]);`
  - quand `command1` termine
    - ferme sa sortie standard
    - `command2` va lire `EOF` et terminer
  - mais il reste des écrivains sur le tube !
  - `command2` attend en lecture depuis le tube...

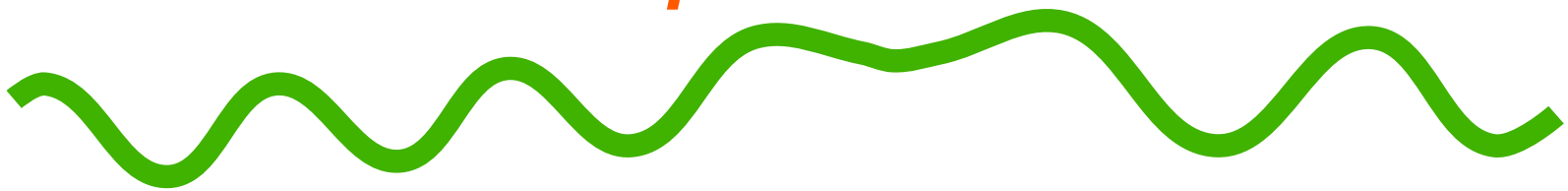
Redirection par `popen()`



- Primitive de la bibliothèque standard
  - `#include <stdio.h>`

    ```
    FILE *popen(const char *command, const char *mode);
    int pclose(FILE *stream);
    ```
- Création d'un tube, puis d'un processus fils
  - processus fils exécute la commande `command` (via un appel `system()`)
  - sortie ou entrée standard de ce processus est redirigée vers/depuis le tube
  - retourne le descripteur du tube permettant d'accéder à cette sortie ou entrée standard
- Option `mode`
  - `"r"` : le descripteur est un accès en lecture au tube depuis lequel le fils écrit
  - `"w"` : le descripteur est un accès en écriture au tube dans lequel le fils lit

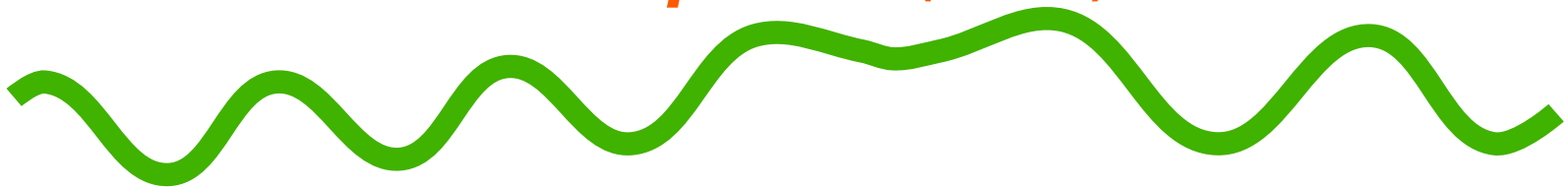
Accès non bloquants



- Lectures et écritures dans un tube bloquantes
  - comportement par défaut
- Possibles accès non bloquants
  - un autre traitement si le tube n'est pas prêt
  - par exemple erreur
  - par exemple traitement de données disponibles sur d'autres tubes
- Positionnement du drapeau `O_NONBLOCK`
  - primitive `fcntl()`
  - exemple pour les écritures

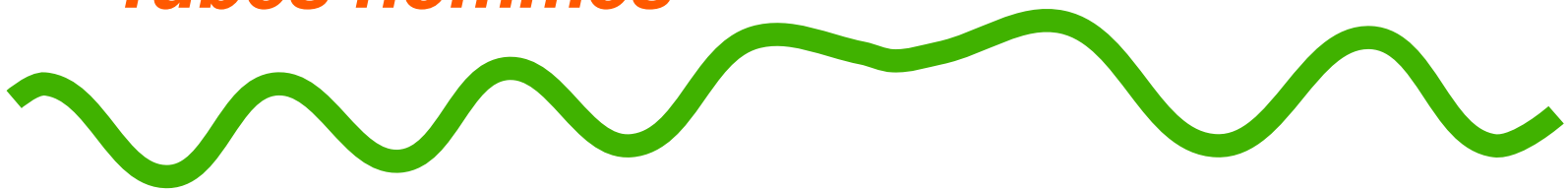
```
fcntl(fd[1], F_SETFL,
              fcntl(fd[1], F_GETFL) | O_NONBLOCK);
```

Accès non bloquants *(cont'd)*



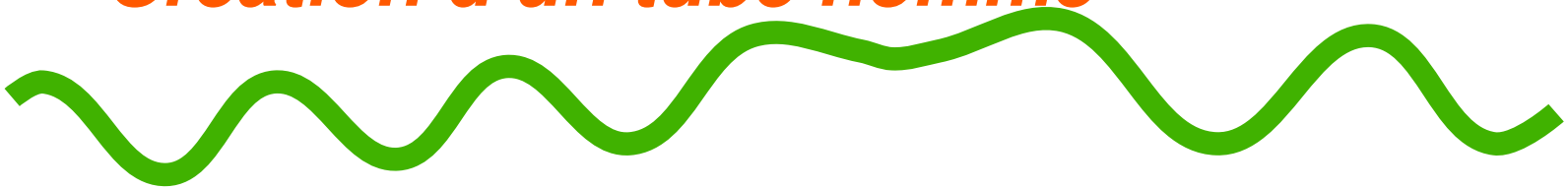
- Écriture non bloquante
  - `write()` retourne -1
  - si on ne peut écrire de manière atomique les `nbytes` caractères demandés
  - positionne `errno` à `EAGAIN`
- Lecture non bloquante
  - `read()` retourne -1
  - si le tube est vide et que le nombre d'écrivains n'est pas nul
  - positionne `errno` à `EAGAIN`

Tubes nommés



- Inconvénient des tubes anonymes
  - les processus communicants doivent avoir un processus parent commun... qui a créé le tube
  - pas de rémanence du tube : détruit, au plus tard, à la terminaison des processus
- Autres utilisations
  - clients se connectent à un serveur
  - rémanence du serveur et du tube
- Tube nommé ou fifo
  - même comportement en lecture/écriture que les tubes anonymes
  - rémanence du tube
  - liaison dans le système de fichier : nom du tube
- Exemple communication client/serveur
  - serveur : création d'un tube
  - serveur : écoute sur le tube
  - client : ouvrir le tube en écriture
  - client : écrire dans le tube

Création d'un tube nommé

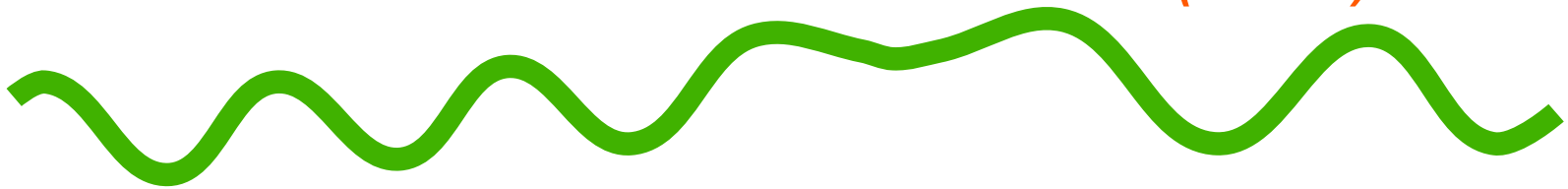


- Commande `mkfifo`
  - `mkfifo [-m mode] file...`
  - crée un entrée dans le système de fichier qui est un tube nommé
  - droits `mode`
  -
```
% mkfifo /tmp/tube
% ls -l /tmp/tube
prw-r--r--  1 phm  wheel  0 21 Mar 05:56 /tmp/tube
```
- Primitive `mkfifo()`
  -
```
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
```
  - création d'un inœud de type `S_IFIFO`

Ouverture d'un tube nommé

- Primitive habituelle d'ouverture `open()`
  - `int open(const char *path, int oflag, ... );`
- C'est un tube !
  - impératif de choisir un mode lecture ou (exclusif) écriture : `O_RDONLY` ou `O_WRONLY`
  - pas de mode `O_RDWR`
- Ouverture bloquante par défaut, particularité des tubes
  - ouverture en lecture bloquante
    - si aucun écrivain, et
    - aucun processus bloqué en ouverture en écriture
  - ouverture en écriture bloquante
    - si aucun lecteur, et
    - aucun processus bloqué en ouverture en lecture
  - donc synchronisation des ouvertures en lecture et écriture
  - attention aux interblocages en cas d'ouverture de plusieurs tubes

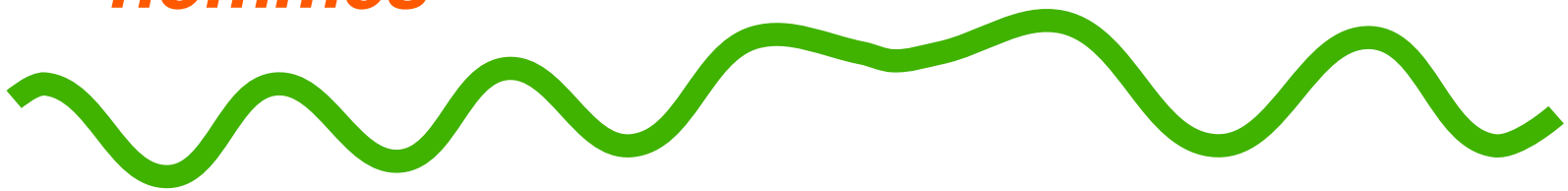
Ouverture d'un tube nommé (cont'd)



- Possibilité d'ouverture non bloquante
  - option `O_NONBLOCK` du mode d'ouverture

    ```
    fd = open("tube", O_WRONLY | O_NONBLOCK);
    ```
- Ouverture non bloquante en lecture
  - succès même s'il n'y a aucun écrivain
  - les lectures seront bloquantes
- Ouverture non bloquante en écriture
  - échec si aucun lecteur, `open()` retourne -1, `errno` à `ENXIO`
  - un descripteur sans lecteur provoquerait un `SIGPIPE` à la première écriture...
  - si l'ouverture réussit, les écritures sont non bloquantes

Exemple d'utilisation de tubes nommés



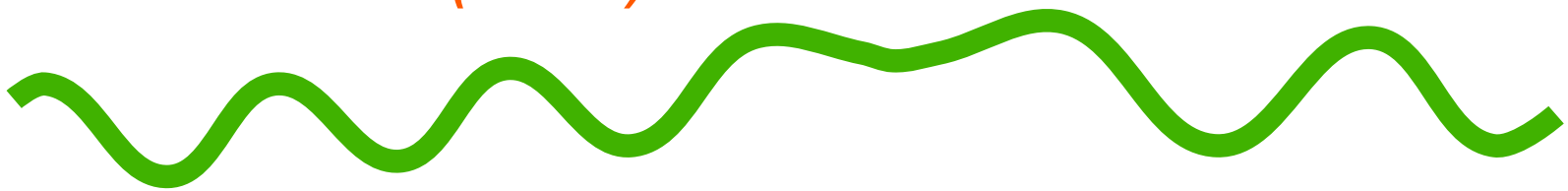
- Application client serveur
  - minimale !
  - un serveur lit sur un tube nommé
  - des messages de 64 caractères
  - et les affiche sur sa sortie standard
  - des clients écrivent des messages (de 64 caractères) dans ce tube nommé

ff-srv.c

```
#define MSIZE   64
#define FIFO    "/tmp/ff-fifo"

int
main ()
{
    int fd;
    char mbuf[MSIZE];
    int status;
    int nbytes;

    status = mkfifo(FIFO, S_666);
    assert(status != -1 || errno == EEXIST); }
```

```
fd = open(FIFO, O_RDONLY);
assert (fd >=0);

for(;;) {
    nbytes = read(fd, mbuf, MSIZE);
    assert(nbytes >= 0);
    if (nbytes > 0)
        printf("message: %s\n", mbuf);
}

exit(EXIT_SUCCESS);
```

Exemple d'utilisation de tubes nommés *(cont'd)*



ff-clt.c

```
#define MSIZE    64
#define FIFO     "/tmp/ff-fifo"

int
main (int argc, char *argv[])
{
     int fd;
     char mbuf[MSIZE];
     int i;

     fd = open(FIFO, O_WRONLY);
     assert (fd >=0);

     for(i=1; i< argc; i++) {
          strncpy(mbuf, argv[i], MSIZE-1);
          mbuf[MSIZE] = '\0';
          write(fd, mbuf, MSIZE);
     }

     exit(EXIT_SUCCESS);
}
```

```
% ./ff-clt "Before the big bang"
ff-clt.c:28: failed assertion `fd >=0'
Abort

% ls -l /tmp/ff-fifo
prw-r--r--  1 phm  wheel  0 21 Mar 06:08 /tmp/ff-fifo

% ./ff-clt "Hello world" "bye"

% ./ff-clt "Another client speaking..."

% ./ff-clt "Where are you "
```

```
% ./ff-srv
message: Hello world
message: bye
message: Another client speaking...
^C
%
```