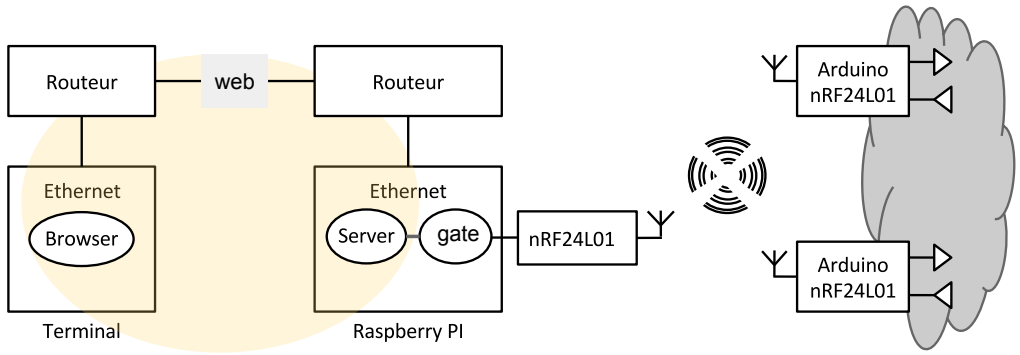
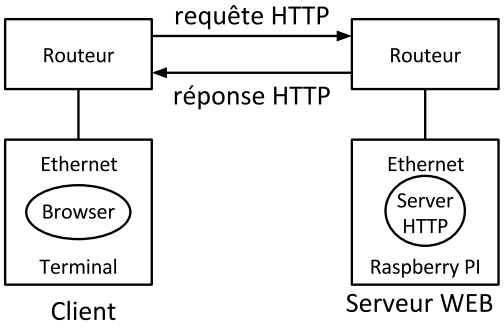
Serveur Web

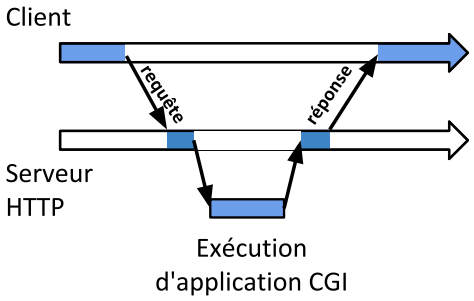
minimaliste mais suffisant

Objectif Final



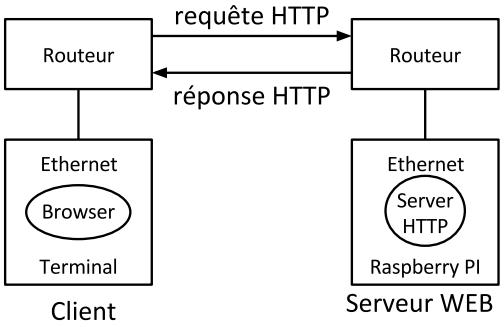
Serveur WEB

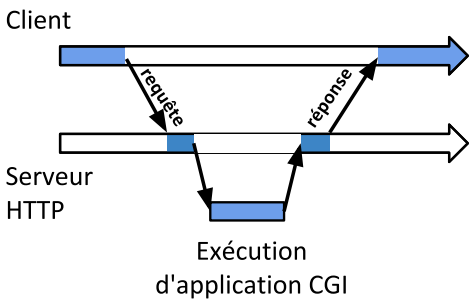




- Un serveur HTTP est une application d'un serveur WEB interprétant des requêtes HTTP (Hyper Text Transport Protocol) venant de clients (Browser web) rendant des pages web HTML par réponse HTTP.
- Le protocole HTTP définit des méthodes comme [GET URI HTTP/ver] ou [POST URI HTTP/ver data].
- URI signifie Universal Ressource Identification qui identifie une ressources sur les serveurs.
- Le format HTML (Hyper Text Markup Language) est un format de représentation des pages permettant de décrire des liens entre les pages et d'inclure des contenus multimédia (images, sons, video).
- Le serveur peut renvoyer de simples pages au format HTML statique ou exécuter des applications CGI (Common Gateway Interface) pour générer des pages dynamiques. Les applications sont écrites en n'importe quel langage mais sont souvent interprétés (Python, PHP, Javascript, etc.)

Serveur WEB





- Les pages envoyées au client sont interprétées par un navigateur (browser)
  - HTML pour décrire la structure (titre, liste, tables, etc.)
  - CSS pour décrire le style (forme, couleur, position, etc.)
  - Javascript pour décrire des comportements sur le poste du client
- Les pages sont préparées par le serveur
  - Statiques : juste du HTML
  - Dynamiques : du HTLM (+ javascript)
    - fabriquées par des scripts écrits en PHP, Python, Perl, Java, etc...
    - en consultant des bases de données : MySQL, SQLite3, PostgreSQL, OracleBD, etc..
    - en communiquant avec des applications

CMS / Framework

- CMS : Content Management System (système de gestion de contenus) permettent d'éditer des pages sans avoir besoin d'écrire du HTML, il y en a beaucoup (libre ou payant) avec chacun une spécialité (blog, wiki, etc.) :
  → Wordpress, Drupal, Typo3
- Framework : regroupe l'ensemble des langages et outils nécessaires à la création d'un site, ils intègrent
  - des langages du front end,
  - des modèles de pages pour ne pas avoir à écrire du HTML,
  - des styles pour les boutons, polices, couleurs, etc.
  - des langages du back end,
  - une base de données,
  - un système de gestion de droits d'accès,
  - un gestionnaire de version,
  - etc.

  → Dango (Python), Symphony (PHP)

Serveur WEB minimaliste

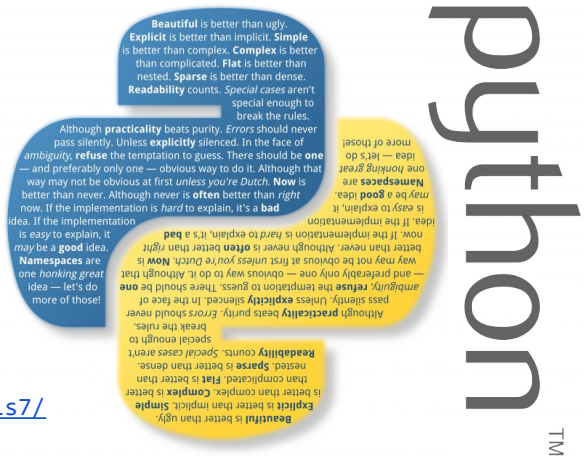
```
server
├── server.py                serveur http
└── www                      répertoire du serveur
      ├── index.html         page d'accueil
      ├── img                répertoire d'image
      │     └── logo.png     logo du site
      └── cgi                répertoire des scripts CGI
            ├── command.py   scripts CGI en python
            ├── charts.py
            └── stat.py
```

```
$ chmod u+x server.py
$ ls -F
$ img/ www/ server.py*
$ cd www
$ ../server.py
```

Python

Langage de programmation objet, multi-paradigme et multiplateformes [...] Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions.

- http://openclassrooms.com/courses/apprenez-a-programmer-en-python
- http://python.developpez.com/tutoriels/cours-python-uni-paris7/
- http://en.wikibooks.org/wiki/A_Beginner%27s_Python_Tutorial



http://westmarch.sjsoft.com/2012/11/zen-of-python-poster/

Le serveur est programmé en python pour sa simplicité, nous allons expliquer le code mais vous allez devoir vous former grâce aux tutoriaux si vous voulez aller plus loin...

Serveur HTTP/CGI en Python

server.py

```
#!/usr/bin/env python
```

Placer au début du fichier, cela permet au shell de savoir quel programme appeler pour ce fichier s'il exécutable.

```
import BaseHTTPServer
import CGIHTTPServer
import cgitb;
cgitb.enable()
```

Importation des blibliothèques python, respectivement :
BaseHTTPServer serveur sur un port défini à la création
CGIHTTPServer gestionnaire pour l'exécution des cgi
cgitb pour avoir les erreurs des scripts cgi

```
server = BaseHTTPServer.HTTPServer
handler = CGIHTTPServer.CGIHTTPRequestHandler
```

Création du serveur HTTP
Création du Gestionnaire de CGI

```
server_address = ("", 8000)
handler.cgi_directories = ["/cgi"]
```

Spécification de l'adresse ethernet et du port d'écoute
Spécification du répertoire contenant les scripts CGI

```
httpd = server(server_address, handler)
httpd.serve_forever()
```

création et
Lancement du serveur

index.html

Documentation : aide-mémoire / doc html / editeur de test

- http://www.html.su/
- http://www.w3schools.com/html/default.asp
- http://www.w3schools.com/html/tryit.asp?filename=tryhtml_basic_document

```
index.html
<html>
<body>
<h1>My First Heading</h1>
<p>My first paragraph</p>
</body>
</html>
```

```
index.html
<html>
  <head><title>Peri Web Server</title></head>
  <frameset rows="100,*" frameborder=0>
    <frame src="img/peri.png">
    <frame src="peri.html" />
  </frameset>
</html>
```

frameset définit, ici, deux zones dans la page

1. pour un logo
2. pour le contenu proprement dit

Appel de script CGI

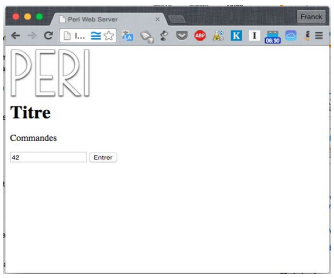
Pour exécuter un script python avec des valeurs recueillies par un formulaire

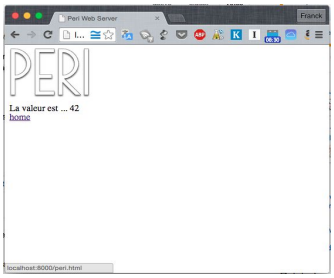
peri.html

```
<html>
<head><title>Peri Web Server</title></head>
<body>
<h1>Titre</h1>
<p>Commandes</p>
<form method="post" action="cgi/command.py">
   <input name="val1" cols="20"></input>
   <input type="submit" value="Entrer">
</form>
</body>
</html>
```

cgi/command.py

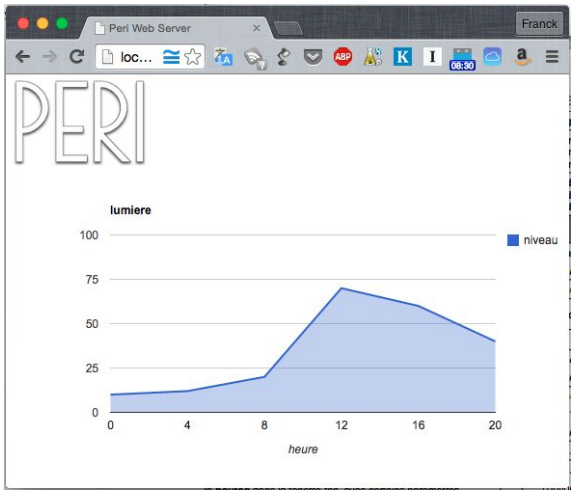
```
#!/usr/bin/env python

import cgi
form = cgi.FieldStorage()
val1 = form.getvalue('val1')

print """
<html>
<body>
La valeur entrée est ... %s
<a href="/peri.html">home</a>
</body>
</html>
""" % (val1,)
```

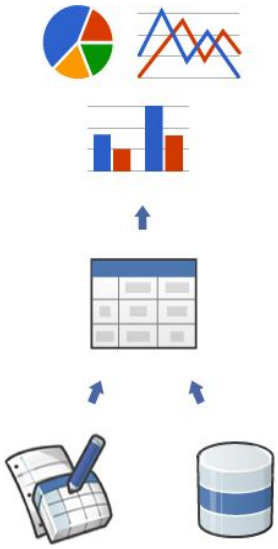




charts.py

Pour l'affichage formaté des résultats, google propose des programmes javascript :
https://developers.google.com/chart/interactive/docs/index





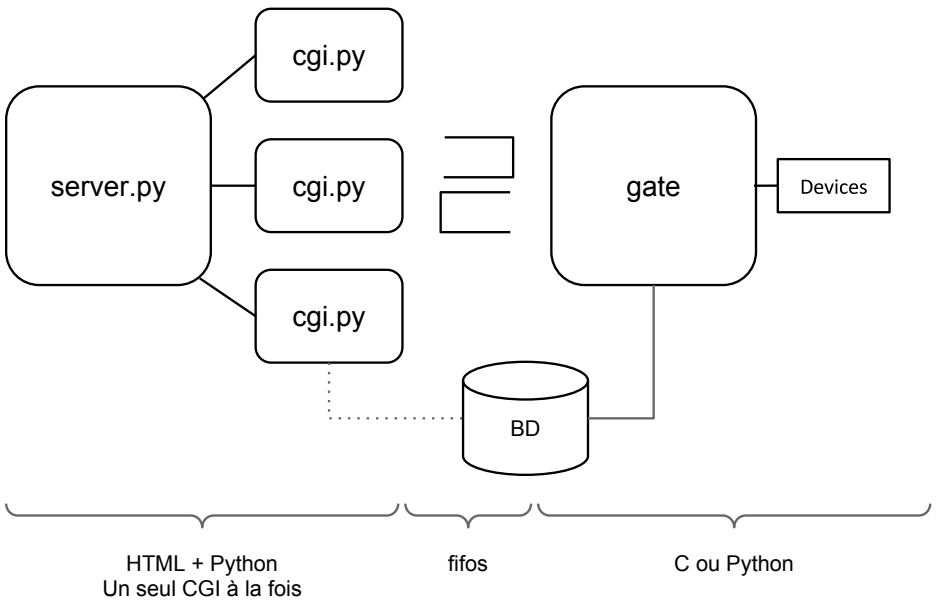
charts.py

```
<html>
  <head>
    <script type="text/javascript" src="https://www.google.com/jsapi"></script>
    <script type="text/javascript">
      google.load("visualization", "1", {packages:["corechart"]});
      google.setOnLoadCallback(drawChart);
      function drawChart() {
        var data = google.visualization.arrayToDataTable([
          ['heure', 'niveau']
         ,['0', 10]
         ,['4', 12]
         ,['8', 20]
         ,['12', 70]
         ,['16', 60]
         ,['20', 40]
        ]);

        var options = {
          title: 'lumiere',
          hAxis: {title: 'heure',  titleTextStyle: {color: '#333'}},
          vAxis: {minValue: 0, maxValue: 100}
        };
        var chart = new google.visualization.AreaChart(document.getElementById('chart_div'));
        chart.draw(data, options);
      }
    </script>
  </head>
  <body>
    <div id="chart_div" style="width: 600; height: 300px;"></div>
  </body>
</html>
```

charts.py

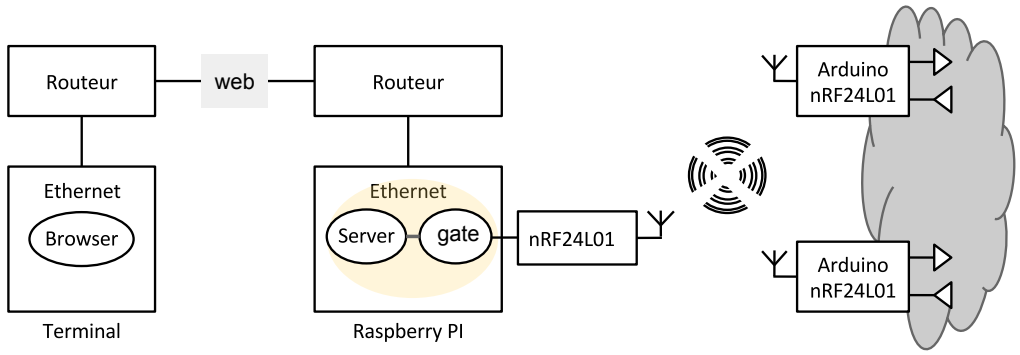
```
#!/usr/bin/env python
import cgi
form = cgi.FieldStorage()
val1 = (int)(form.getvalue('val1'))
tab = [10,12,20,70,60,40]
print """
<html>
  <head>
    <script type="text/javascript" src="https://www.google.com/jsapi"></script>
    <script type="text/javascript">
      google.load("visualization", "1", {packages:["corechart"]});
      google.setOnLoadCallback(drawChart);
      function drawChart() {
        var data = google.visualization.arrayToDataTable([
          ['heure', 'niveau']"""
i=0
while i <= val1:
  print "         ,['%d', %d]" % (i,tab[i/4])
  i += 4
print """\
        ]);
        var options = {
          title: 'lumiere',
          hAxis: {title: 'heure',  titleTextStyle: {color: '#333'}},
          vAxis: {minValue: 0, maxValue: 100}
        };
        var chart = new google.visualization.AreaChart(document.getElementById('chart_div'));
        chart.draw(data, options);
      }
    </script>
  </head>
  <body>
    <div id="chart_div" style="width: 600; height: 300px;"></div>
  </body>
</html>
"""
```

Architecture possible pour l'accès aux capteurs

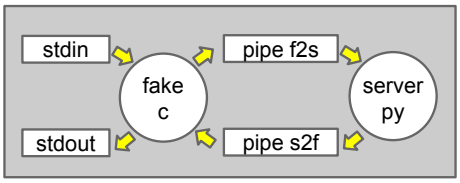


Communication inter-processus par tube

Objectif Final



communication par tube



fake lit une valeur sur stdin, ajoute un numéro id et écrit le message sur f2s, le server lit le message, calcule la parité et renvoie l'id et la parité sur s2f, fake lit le message et affiche l'id, la valeur et la parité.

- Communication par tubes en C et en python
- Attente sur plusieurs fichiers en C

Liste des fonctions à utiliser

La liste n'est pas exhaustive,
il y a seulement les fonctions spécifiques

en C	en python
open	open
read	readline
write	write
close	close
mkfifo	mkfifo
select	

Communication par tube anonyme (pipe)

Objectif communication dans un sens fifo entre deux processus.

- Une fifo à deux extrémités
- Chaque extrémité est référencée par un descripteur de fichiers (entier)

Les pipes peuvent être anonymes

- création : **`int pipe(int fd[2]);`**
  - **`pipe()`** renvoie des numéro de descripteurs de fichiers dans **`fd[]`**
  - `lit 0, écrit 1 :` **`fd[0]`** ← **`pipe`** ← **`fd[1]`**
  - la taille d'un pipe est limitée à 64ko (depuis la 2.6.11)
- l'écriture : **`ssize_t write(int fd, const void *buf, size_t count);`**
  - écrit **`count`** octets de **`buf`** dans **`fd`**
  - rend le nombre d'octet écrits
- la lecture : **`ssize_t read(int fd, void *buf, size_t count);`**
  - lit **`count`** octets dans **`fd`** et les écrit dans **`buf`**
  - rend -1 si echec (+ **`errno`**) sinon le nombre d'octets lus, 0 si **`fd`** vide
- la fermeture : **`int close(int fd);`**
  - il faut fermer les deux extrémités

exemple un seul processus

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

int main( int argc, char ** argv )
{
  char buffer[BUFSIZ+1];

  /* create the pipe */
  int fd[2];
  pipe(fd);

  /* write into the pipe */
  write(fd[1], "Hello World\n", strlen("Hello World\n"));

  /* read the pipe and print the read value */
  read(fd[0], buffer, BUFSIZ);
  printf("%s", buffer);
}
```

http://www.zeitoun.net/articles/communication-par-tuyau/start

Communication entre un père et son fils

- Le pipe est créé dans un processus
- Le processus est dupliqué par **`fork()`**
  - **`fork()`** rend **`0`** pour le fils et le **`pid`** créé chez le père
  - les deux nouveaux processus ont tous les deux accès au pipe
  - chacun va fermer l'une des extrémités avec **`close()`**
- Si on veut une communication dans les deux sens, il faut deux pipes

Communication entre un père et son fils

```
int main( int argc, char ** argv )
{
    int  pfd[2];                                    // file descriptors of pipe
    pipe(pfd)                                       // create anomymous pipe
    if (fork() == 0)                                // fork returns 0 to son
    {
        char buffer[BUFSIZ];
CHILD   close(pfd[1]);                              // close write side
        while (read(pfd[0], buffer, BUFSIZ) != 0)   // read and ...
            printf("child reads %s", buffer);       // ... print result on screen
        close(pfd[0]);                              // close the pipe
    }
    else {
        char buffer[BUFSIZ];
        close(pfd[0]);                              // close read side
PARENT  strcpy(buffer, "HelloWorld\n");
        write(pfd[1], buffer, strlen(buffer)+1);
        close(pfd[1]);                              // close the pipe
    }
    return 0;
}
```

Redirection d'un pipe

Pour rediriger les flux standards (**stdin**, **stdout** ou **stderr**) sur un pipe déjà ouvert, on utilise **dup2**

int dup2(int oldfd, int newfd);

oldfd et **newfd** doivent être des descripteurs de fichier ouverts.

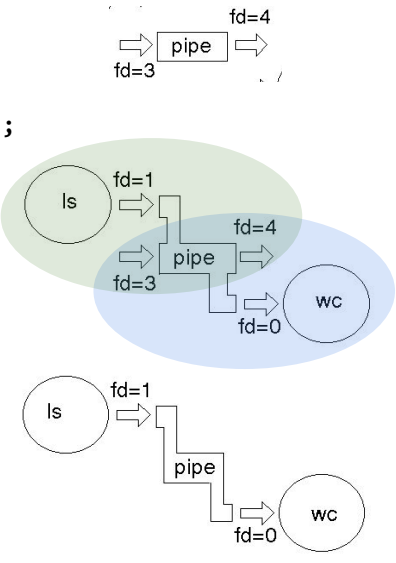
Si **oldfd** est un descripteur de pipe, il sera désormais accessible par **newfd**.

Les deux descripteurs sont des alias du même fichier.

On doit en principe fermer le descripteur **oldfd**.

exemple de redirection d'un pipe

```
int main (int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    switch (fork()) {
    case -1: perror("fork"); exit(EXIT_FAILURE);
    case 0:
        close(fds[1]); // close 3
        dup2(fds[0], STDIN_FILENO);
        close(fds[0]);
        execlp("wc", "wc", NULL);
        perror("wc"); exit(EXIT_FAILURE);
    default:
        close(fds[0]);
        dup2(fds[1], STDOUT_FILENO);
        close(fds[1]);
        execlp("ls", "ls", NULL);
        perror("ls"); exit(EXIT_FAILURE);
    }
}
```



https://jan.newmarch.name/OS/I9_1.html

Communication par tube nommé (fifo)

Si on veut faire communiquer deux processus qui n'ont pas de liens de parenté direct, on utilise des tubes nommés qui seront placés sur le disque, nommé fifo.

- La création d'une fifo :

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

  la fifo est un fichier de type fifo créer avec le mode mode
  (par exemple Ox666 pour un droit rw pour l'utilisateur, le groupe et les autres)

  rend 0 en cas de succès

- La destruction d'une fifo

  `int unlink(const char * pathname);`

  efface le fichier mais reste accessible par les process qui l'ont encore ouvert.

Ouverture d'une fifo

Une fois créée une fifo doit être ouverte pour avec **open**.

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

`flags :`

- **`O_RDONLY`**
- **`O_WRONLY`**
- **`O_RDWR`**

Les deux extrémités doivent être ouvertes pour utiliser le tube.

- L'ouverture en lecture en bloquante tant qu'il n'y a pas d'ouverture en écriture afin de synchroniser les deux processus.
- On peut ajouter le mode **`O_NONBLOCK`** pour ne pas se bloquer à la condition que le lecteur ET l'écrivain fassent de même, sinon c'est une erreur.

lecture d'une fifo

nb_lu = read(fd, buffer, TAILLE_READ);

Si le tube n'est pas vide et contient **taille** caractères :

Lecture de **nb_lu = min (taille, TAILLE READ)** caractères.

Si le tube est vide

Si le nombre d'écrivains est nul

Alors c'est la fin de fichier et **nb_lu** est nul.

Si le nombre d'écrivains est non nul

Si lecture bloquante alors sommeil

Si lecture non bloquante alors en fonction de l'indicateur

O_NONBLOCK : nb lu= -1 et errno=EAGAIN.

O_NDELAY : nb lu = 0.

écriture d'une fifo

nb_ecrit = write(fd, buf, n);

L'écriture est atomique si le nombre de caractères à écrire est inférieur à **PIPE_BUF** (linux 4096), la taille du tube sur le système. (cf **<limits.h>**).

Si le nombre de lecteurs est nul

Envoi du signal **SIGPIPE** à l'écrivain et **errno = EPIPE.**

Sinon

Si l'écriture est bloquante, il n'y a retour que quand

Les n caractères ont été écrits dans le tube.

Si écriture non bloquante

Si **n > PIPE_BUF**, retour avec un nombre inférieur à n

Si **n <= PIPE BUF**

Et si n emplacements libres, écriture nb_ecrit = n

Sinon retour -1 ou 0.

écoute de plusieurs canaux

SI la fonction **read()** est bloquante, il ne faut la lire que si l'on est sûr qu'il y a des données, une fonction permet de se mettre en attente sur plusieurs flux.

```
int  select( int n, fd_set *readfds, fd_set *writefds,
             fd_set *exceptfds, struct timeval *timeout);
```

- **n** numéro du plus grand descripteur + 1
- **readfds** masque des descripteurs de fichiers attendus en lecture
- **writefds** masque des descripteurs de fichiers attendus en écriture
- **exceptfds** masque des descripteurs de fichiers attendus pour des conditions spéciales....

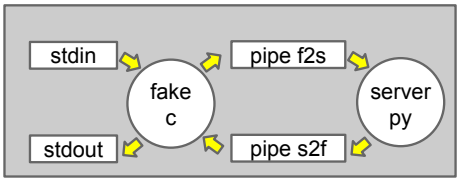
La fabrication des masques se fait avec les macros suivantes:

- **FD_ZERO(fd_set *set);** -> efface tout l'ensemble
- **FD_SET(int fd, fd_set *set);** -> met à **1** un descripteur
- **FD_CLR(int fd, fd_set *set);** -> met à **0** descripteur
- **FD_ISSET(int fd, fd_set *set);** -> rend l'état du descripteur **fd**

Exemple d'usage de select pour un timeout

```
#include <stdio.h>
#include <sys/time.h>

int main(void) {
     fd_set rfds;
     struct timeval tv;
     int retval;

     FD_ZERO(&rfds);    /* Surveiller stdin (fd 0) en attente d'entrées */
     FD_SET(0, &rfds);

     tv.tv_sec = 5;    /* Pendant 5 secondes maxi */
     tv.tv_usec = 0;

     retval = select(1, &rfds, NULL, NULL, &tv);

     if (retval) {
          char buffer[80];
          printf("Données disponibles\n"); /* FD_ISSET(0, &rfds) est vrai */
          fgets(buffer, sizeof(buffer),stdin);
          printf("got: %s\n", buffer);
     } else
          printf("Pas de données depuis 5 secondes\n");
     return 0;
}
```

TP



- Ecrire un site web qui permet d'interagir avec pour le contenu des pages

- Ecrire un site web qui permet de commander les LEDS et lire le BP