TUBE NOMMÉ

- Pour concevoir une architecture client-serveur.
  - convention vers ref tubes
  - soit persistant, soit créés au démarrage du serveur.

- Problèmes communs :
  - entrelacement des requêtes (cf PIPE_BUF si on écrit ≤ PIPE_BUF)
  - désynchronisation des requêtes (on ne peut lire qu'un certain nbr d'octet ...
- Solutions :
  - Marqueur de fin (ex: $ à la fin du message) → peu satisfaisant car gourmand.
  - Imposer des requêtes de taille fixe.
  → gênant si on a des requêtes trop grandes ou de taille variables.
  - Préfixer les envois d'une entête de taille fixe.

On peut tout écrire en 1 seul write.
Côté lecture
read (entête, taille_entête_fixe)
read (reste, entête.taille_req)

⇒ Impossible de l'envoyer via un tube partagé par pls clients

⇒ Impossible d'utiliser un tube multi-usage

⇒ Pour architecture clt-serv :

- 1 tube requête
- 1 tube réponse PAR CLIENT.
  - ↳ · avoir une convention de nommage (et de création)
    · pas créer d'interblocage à l'ouverture.

⇝ voir fonction unlink.

Dans un serveur :

Si on ouvre puis while (1) on lit

↓

écriture en boucle car rien ne bloque dans le while

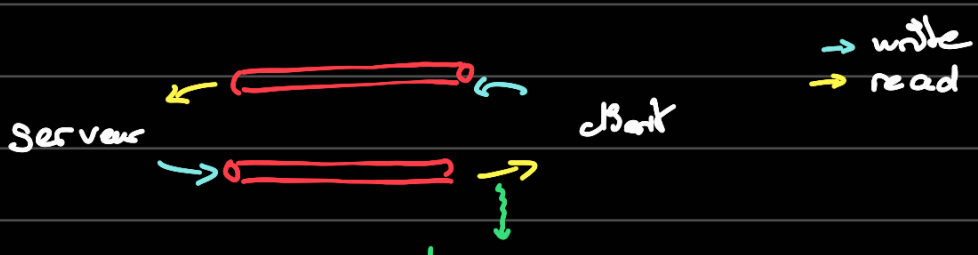
⇒ Solutions :

- while (1) :
  open (tube_req) ← ouverture bloquante.
  ⋮
  while (tube_rep)
- Seulement s/ linux)
  on peut open (Fd_req, O_RDWR)
  while (1)
  ⋮

↑ "leurer le système qu'il y a aussi une écriture"

Ici on bloquera sur read(fd_req) jsq ce que client soit lancé.

Imaginons maintenant qu'un serveur se retrouve à avoir tube de requête pour chaque client. ou qu'il doive lire sur std_in et sur un tube en même temps.



lecture qui bloque le serveur sur l'écriture et c'est problématique s'il doit relire dessus

→ Solutions :

1. Cloner le processus.
   → pas toujours possible (besoin d'unicité du processus) et de synchronisation.
2. Passer les descripteurs en mode O_NONBLOCK
   → attente active (while(1) s'effectue toujours ou while contient des read/write donc TRÈS (trop) gourmand)
3. (La bonne) Demander au système qu'il surveille les fd
   ("reveil moi quand fd_? sera prêt")

Developpons la 3.

cf poly pour les détails.

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

int poll(pollFd *fds, nfds_t nfds, int timeout);
```

un fd_set est un mot binaire

(01000111 = {1, 4, 5, 6})

= ens des fd_s à lire.

rmq : nFds = 7 = |fd_set| + 1

Pour savoir si fd est prêt :

```
int FD_ISSET(int fd, fd_set *set);
```

⚠ timeout renvoie une val différente (on peut envoyer un timeout mais il est modifié différemment si erreur) C'est une mauvaise idée de l'utiliser. ⚠

(cf code s/ selection.c)

→ vérifier en plus de s'il sont prêts, qu'ils peuvent être utilisés.

poll n'est pas vraiment plus simple (cf poly)
mais au moins il ne détruit pas les paramètres.
Astuce prendre -fd s'il on veut l'ignorer.

→ poll plus adapté par nous

Pour le démon: il doit (toutes les minutes voir s'il y a des tâches à lancer.
et surveille au moins un fd (tube de req)
On peut vouloir faire plusieurs écritures aussi.
et on doit donc surveiller tube de req et tube de réponse.

En gros:

```
while (1)
    :
    poll ?
    → est-ce que j'ai reçu une tache
    :
```