420 Tubes

INF3173
Principes des systèmes d'exploitation

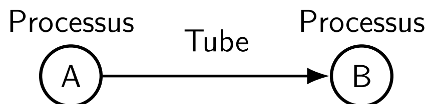
Jean Privat

Université du Québec à Montréal

Hiver 2021

Tubes

- Canal de communication **unidirectionnel** avec deux bouts
- Les octets écrits au bout en écriture (`write(2)`)
- Sont lisibles **dans l'ordre** au bout en lecture (`read(2)`)
- Flot d'octets (*stream*) : pas de concept de messages
- Les octets lus sont consommés
- `pipe(7)` pour les détails



Tube et processus

Descripteurs de fichiers

- Pour un processus, un bout de tube est un descripteur
- Chaque extrémité se manipule comme fichier ouvert
  `read(2)`, `write(2)`, `close(2)`, `dup2(2)`, `poll(2)`, etc.
- Mais pas `lseek(2)` (erreur `ESPIPE`)

Niveau noyau

- Espace mémoire du système d'exploitation
- L'espace et son accès sont gérés par le SE
- Capacité limitée (64ko défaut actuel sous Linux)
- Mais c'est pas un problème (on y reviendra)
- Libéré automatiquement quand plus utilisé

Deux sortes de tubes

Tubes simples (majoritairement utilisés)

- Création : appel système `pipe(2)`
- « Retourne » deux descripteurs de fichiers
- `int fds[2]; pipe(fds);`
- `fds[0]` le bout en lecture
- `fds[1]` le bout en écriture
- Astuce mnémotechnique: 0=stdin 1=stdout

Tubes nommés

- Création : `mkfifo(1)` et `mkfifo(3)`
- On y reviendra...

Exemple tube simple

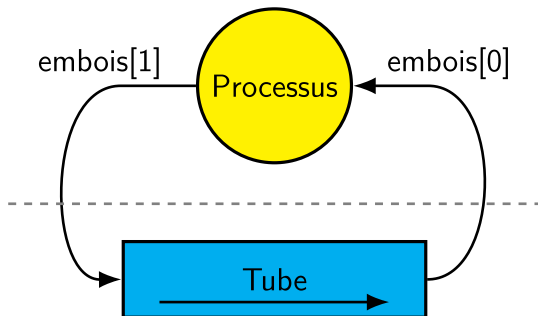
```
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include<string.h>
int main(void) {
  char *msg = "Bonjour, le monde!", buf[32];

  int embois[2];
  pipe(embois);

  write(embois[1], msg, strlen(msg)+1);

  read(embois[0], buf, sizeof(buf));
  printf("lu: « %s »\n", buf);
  return 0;
}
```

Tubes simples



Communication par tube

- Un tube est créé par un processus
- Mais est global au système

Partage de tube par fork

- Les descripteurs de fichiers sont copiés
- Les bouts de tubes sont partagés

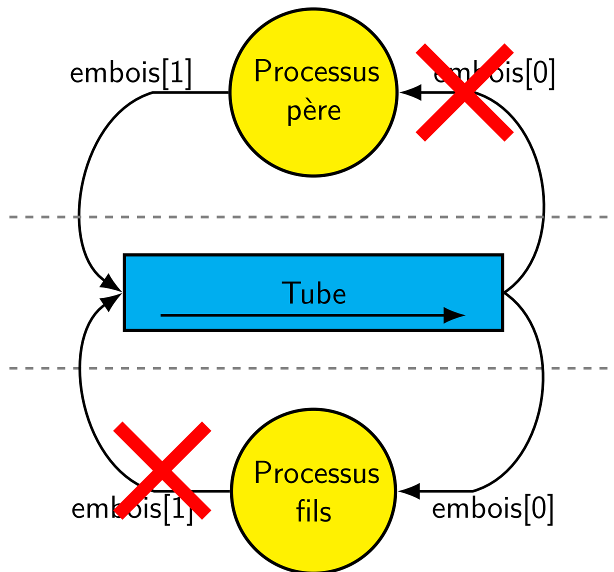
Communications

- Entre parent et enfant
  - Le parent crée le tube
  - L'enfant hérite les descripteurs
- Entre deux enfants
  - Le parent crée le tube
  - Les enfants héritent les descripteurs

pipe-fork.c

```c
#include "machins.h"
int main(void) {
  int embois[2];
  pipe(embois);
  pid_t pid = fork();
  if (pid==0) { // enfant
    char buf[32];
    close(embois[1]);
    read(embois[0], buf, sizeof(buf));
    printf("lu: « %s »\n", buf);
  } else { // parent
    char *msg = "Bonjour, le monde!";
    close(embois[0]);
    write(embois[1], msg, strlen(msg)+1);
    waitpid(pid, NULL, 0);
  }
  return 0;
}
```

Communication par tube



Synchronisation

Lecture

- S'il y a des données dans le tube
  - `read` **lit** le **maximum** d'octets
- Si le tube est vide
  - Si un écrivain existe : `read` **bloque**
  - Si aucun écrivain : `read` **retourne 0** (fin de tube)

Écriture

- S'il y a aucun lecteur
  - Signal `SIGPIPE` **envoyé** (par défaut, **termine** le processus)
- S'il y a un lecteur (ou plus)
  - Si assez de place: `write` **écrit tous** les octets
  - Si le tube est plein (ou presque): `write` **bloque**

Contrôle de flux

Lecteur qui va trop vite

- Bloqué jusqu'à ce qu'un écrivain écrive
- Ou plus de données ni d'écrivain (`read` retourne 0)

Écrivain qui va trop vite

- Bloqué jusqu'à ce qu'un lecteur consomme
- Ou que plus de lecteurs (`SIGPIPE`)

Questions

- Pourquoi c'est pas symétrique ? (0 vs. `SIGPIPE`)
- Comment gérer `SIGPIPE` ?

Extra

Opérations atomiques

- `PIPE_BUF` (512 minimum, 4096 chez Linux)
- `write` écrits `PIPE_BUF` octets (ou moins) atomiquement
- Atomiquement = écrit d'un coup sans que d'autres écritures concurrentes s'entrelacent

Entrées-sorties non bloquantes

- Flag `O_NONBLOCK` possible (via `fnctl(2)`)
- Les règles de synchronisation et d'atomicité changent
- RTFM

Danger : interblocage

Comment se bloquer tout seul ?

Danger : interblocage

Comment se bloquer tout seul ?

```
#include<unistd.h>
int main(void) {
  int embois[2];
  char buf;
  pipe(embois);
  read(embois[0], &buf, 1);
}
```

Question

- Comment se bloquer à deux ?

Bonnes pratiques

Un seul lecteur et un seul écrivain

- « Toujours par deux ils vont, ni plus, ni moins » — Yoda

Fermer les bouts inutiles

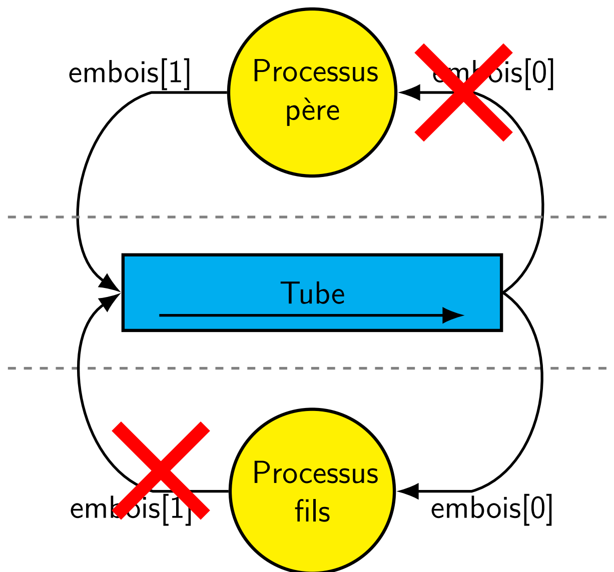
- Laisser des bouts trainer cause des problèmes de synchronisation
- Souvent: lecteur bloqué, car un bout d'écrivain reste quelque part

Plusieurs écrivains et/ou lecteurs ?

- C'est techniquement possible, mais :
- Bien comprendre les règles de synchronisation et d'atomicité
- Les clients doivent être coopératifs
- Messages de taille fixe aide beaucoup

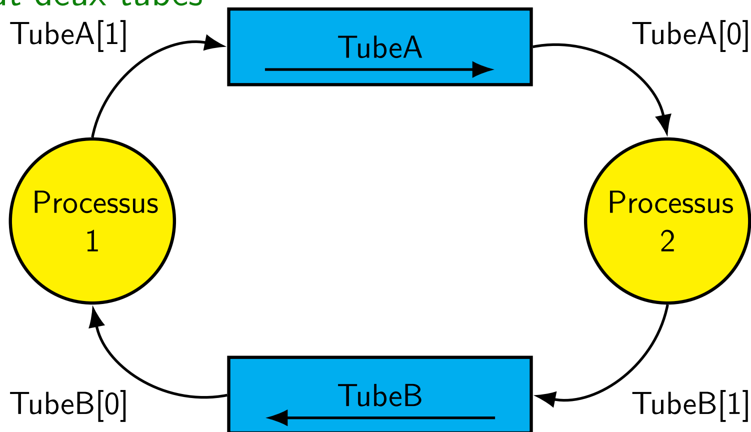
$\rightarrow$ Utiliser un autre IPC, c'est souvent moins risqué

Bonnes pratiques



Pour une communication bidirectionnelle

Il faut deux tubes



- Des systèmes offrent des tubes bidirectionnels (pas portable!)
- Utiliser un autre IPC,c'est souvent plus simple

Tubes shell

```
$ whoami | cowsay
```

```c
#include "machins.h"
int main(int argc, char **argv) {
  int p[2];
  pipe(p);
  pid_t whoami = fork();
  if (whoami == 0 ) {
    dup2(p[1], 1);
    close(p[0]); close(p[1]);
    execlp("whoami", "whoami", NULL);
    perror("whoami"); return 1;
  }
  pid_t cowsay = fork();
  if (cowsay == 0) {
    dup2(p[0], 0);
    close(p[0]); close(p[1]);
    execlp("cowsay", "cowsay", NULL);
    perror("cowsay"); return 1;
  }
  close(p[0]); close(p[1]);
  waitpid(whoami, NULL, 0); waitpid(cowsay, NULL, 0);
  return 0;
}
```

Linux et pseudo fichiers tubes

- Ouvrir un pseudo fichier tube de `/proc/PID/fd` est possible
- Le mode d'ouverture indique quel bout du tube on obtient

→ Permet d'ajouter des lecteurs et des écrivains

→ Même si c'est souvent pas une bonne idée

```
(echo marco; sleep 1; echo polo) | lolcat &
echo trololo > /proc/$!/fd/0
```

Tubes nommés

Limites des tubes simples

- Via héritage des processus
- En créant le tube d'avance

$\rightarrow$ Communication entre processus indépendants difficile

Principe des tubes nommés

- Les tubes nommés ne sont pas **hérités,** mais **désignés**
- Donc plus besoin d'hériter des descripteurs
- Ni de créer le tube d'avance

Caractéristiques

- Exactement comme un tube simple
- Mais: ouverture d'un tube par un nom
- Plus: gestion des droits
- Plus: mécanisme de rendez-vous entre processus

Tubes nommées

Fichier spécial « tube »

- Crée avec `mkfifo(1)` et `mkfifo(3)` (et `mknod(2)`)
- L'inode (via chemins) désigne le tube
- Les droits du fichier sont les droits d'accès au tube
- Ouvrir (`open(2)`) le fichier c'est accéder au tube
- Le fichier **est** et **reste vide**
  - Le tube est entièrement en mémoire
  - Le fichier n'est qu'une astuce pour désigner

Rendez-vous

- `open(2)` **bloque** jusqu'à avoir un lecteur **et** un écrivain
- Le tube se comporte ensuite comme un tube simple

→ Même synchronisation, même atomicité

Substitution de processus

Chez `bash(2)` : `<(CMD)`

Principe

- Exécute `CMD` dans un processus indépendant
- Où la sortie standard de `CMD` est redirigée dans un tube
- Substitue l'argument `<(CMD)` par le chemin du tube
- Quelqu'un qui ouvrira ce chemin sera connecté au tube

Deux implémentations

- Pseudo fichiers tubes (`proc(5)`) si disponible
- Tube nommé sinon

→ On verra ça en lab