Programmation Système et Réseau
Communication Inter-Processus (IPC) – Les Tubes (Pipes)

Tubes

Tubes anonymes et nommés

Tubes

LES TUBES OU PIPE

- Types :
  - tube anonyme
  - tube nommé
- Moyen de communication entre deux processus s'exécutant sur une même machine
- Fichiers particuliers
- Gérés par le noyau
- File de données en mémoire (FIFO)
- Lectures destructrices

Communication

Par communication inter-processus on entend :

- Existence de **plusieurs processus** sur la **même machine** travaillant “**simultanément**”.
- Échange d’information entre ces processus.

Problèmes classiques

- Quels moyens de communications sont disponibles ?
- Comment choisir le moyen de communication approprié ?
- Comment être sûr que le bon processus accède à la donnée qu'il attend ?
- Comment hiérarchiser l'exécution des processus pour que l'information demandée soit disponible ?

Un air de déjà vu...

- **Échange d'information par fichier :** un conteneur (fichier) stocke l'information la rendant disponible pour tout autre pro- cessus ayant accès à ce fichier.
- **En script :** l'utilisation de pipe permet d'enchaîner des commandes (donc processus différents) en leur transmettant des informations.

Pipe “$commande shell$”

- La commnande “**ps -a | wc -l**” entraîne la création de deux processus concurrents (allocation du processeur).
- Un tube est créé dans lequel les résultats du premier processus ("**ps-a**") sont écrits.
- Le second processus lit dans le tube.
- Un tube de communication (|) permet de mémoriser des informations.

Pipe “commande shell”

- Il se comporte comme une file **FIFO** (première donnée écrite, première donnée lue), d’où son aspect **unidirectionnel**. Par ailleurs, **les lectures sont destructives**. C’est-à-dire que les données lues par un processus disparaissent du tube.

- Lorsque le **processus écrivain** se termine et que le **processus lecteur** dans le tube a fini d’y lire (le tube est donc vide et sans lecteur), ce processus détecte une fin de fichier sur son entrée standard et se termine.

Synchronisation

- Le système assure la synchronisation de l'ensemble dans le sens où :
  - il bloque le processus lecteur du tube lorsque le **tube est vide** (ne contient aucun caractère) en attendant qu'il se remplisse (s'il y a encore des processus écrivains)
  - il bloque (éventuellement) le processus écrivain lorsque le **tube est plein** (si le lecteur est plus lent que l'écrivain et que le volume des résultats à écrire dans le tube est important).

- Le système assure l'implémentation des tubes. Il est chargé de leur création et de leur destruction.

Tubes

Tubes anonymes

Tubes

TUBE ANONYME

- Structure sans nom
- Communication entre deux processus

  Deux descripteurs : lecture et écriture
- Deux pointeurs automatiques : lecture et écriture
  - pointeur de lecture sur le 1er caractère non lu
  - pointeur d'écriture sur le 1er emplacement vide
- Processus de même filiation ou ancêtre commun ayant créé le tube

Tubes

TUBE ANONYME

- **tube** = canal half-duplex (permet une communication dans les deux sens, mais une seule direction à la fois)
- signifie PA peut parler à PB et PB peut aussi parler à PA mais pas en même temps.

Les Tubes

- Un tube est presque identique à un fichier ordinaire. Il est caractérisé par :
  - **Taille limitée** (définie par la constante PIPE_BUF dans le fichier **<limits.h>.**)
  - **Deux extrémités**, permettant chacune soit de lire dans le tube, soit d'y écrire.
  - Au plus **deux entrées** dans la **table des fichiers** ouverts (une pour la lecture et une pour l'écriture)
  - L'opération de lecture dans un tube est **destructive** : une information ne peut être lue qu'une **seule fois** dans un tube.

Tubes anonymes (non nommés)

- Fichier logique
- Deux descripteurs (lecture/écriture)
- Aucune référence dans le système de fichier (fichier anonyme)
- Norme :
  - Unidirectionnel
  - Un descripteur pour la lecture, un descripteur pour l'écriture
- Lien de parenté obligatoire (processus créateur et ses descendants créés après le tube)
- Destruction automatique à la fin de l'utilisation

Tubes anonymes : comportement par défaut

- Tube vide :
  - Lecture bloquante.
- Tube non vide :
  - On lit uniquement les caractères disponibles, même si nous n'en attendons plus
- Tube plein :
  - Écriture bloquante

Tubes anonymes : comportement sans lecteur ou écrivain

- **Ecriture sans lecteur :**
  - Nombres de lecteurs = nombre de descripteurs associés à la lecture depuis le tube.
  - S'il n'y a pas de lecteurs (nombre de lecteurs = 0), le tube est inutilisable : les écrivains sont prévenus par le signal **SIGPIPE** envoyé par le système.
  - Comportement par défaut : fin du processus
- **Lecture sans écrivain :**
  - S'il n'a pas d'écrivains, le tube est inutilisable : les lecteurs sont prévenus : notion de fin de fichier

Tube anonymes : Principe

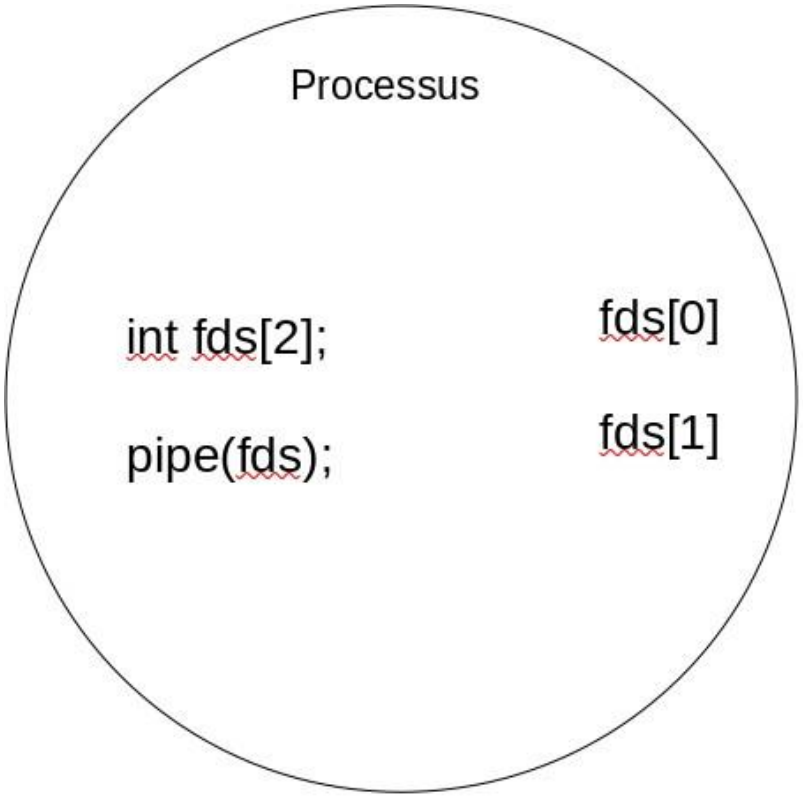
- **pipe()** : création du tube par le père
- **fork()** : création du processus fils
- héritage de l'ouverture du tube (fichier)
- **exec()** : passage des descripteurs en paramètres

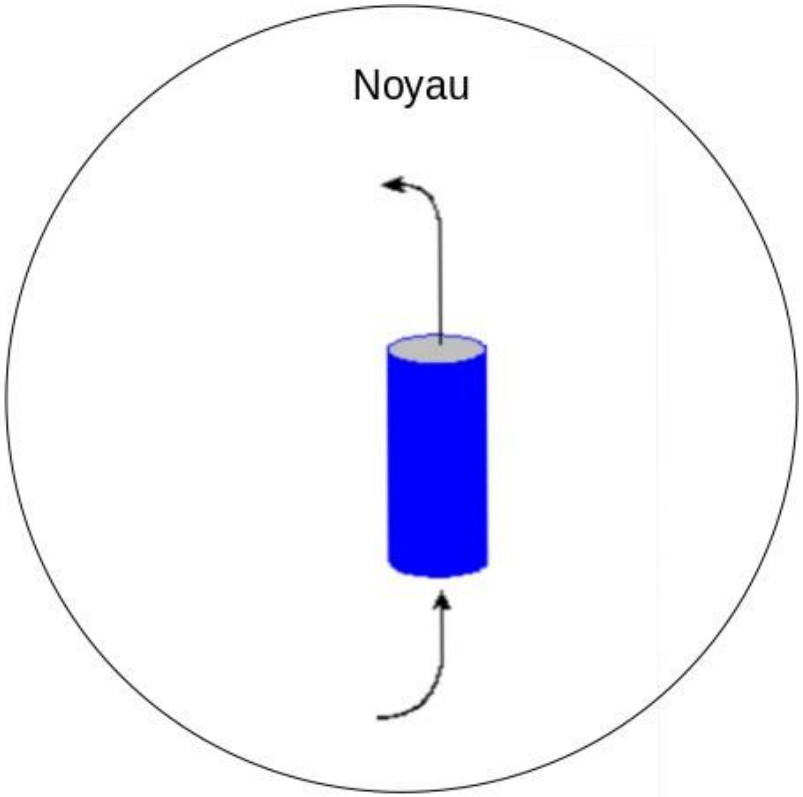
Tubes anonymes : primitives

```
#include<unistd.h>
int pipe(int p[2]);
```

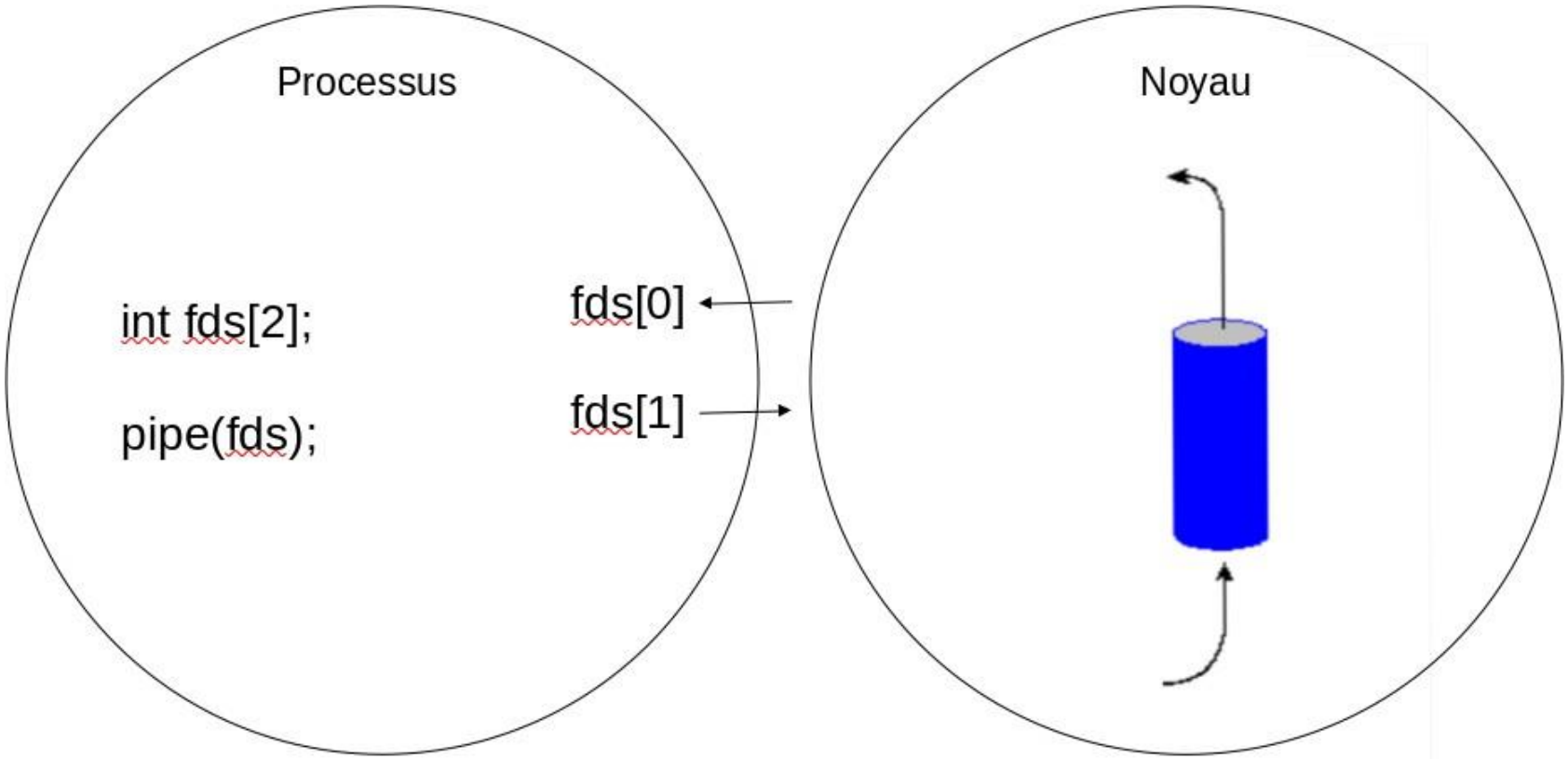
- La primitive `pipe()` permet de créer un tube anonyme. Elle retourne 2 descripteurs placés dans le tableau `p`.
  - **p[0]** : descripteur en lecture
  - **p[1]** : descripteur en écriture
- Les 2 descripteurs sont alloués dans la table des fichiers ouverts du processus et pointent respectivement vers un objet fichier en lecture et un objet fichier en écriture.
- Connaissance du tube
  - Avoir réalisé l'opération `pipe()`
  - Héritage des descripteurs lors de `fork()`
  - Perte d'un descripteur (fermeture) → accès impossible

Tube anonymes : Principe

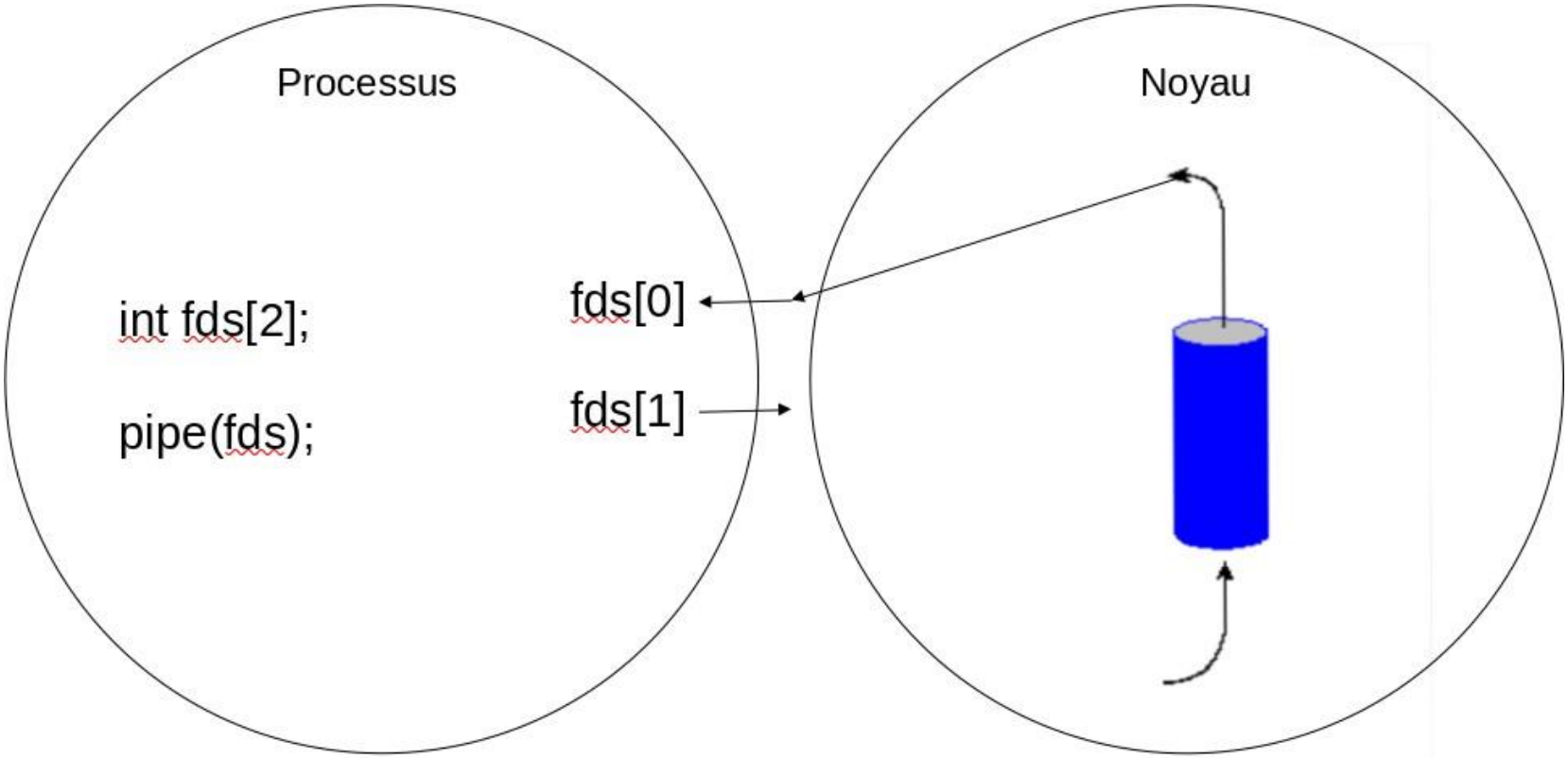




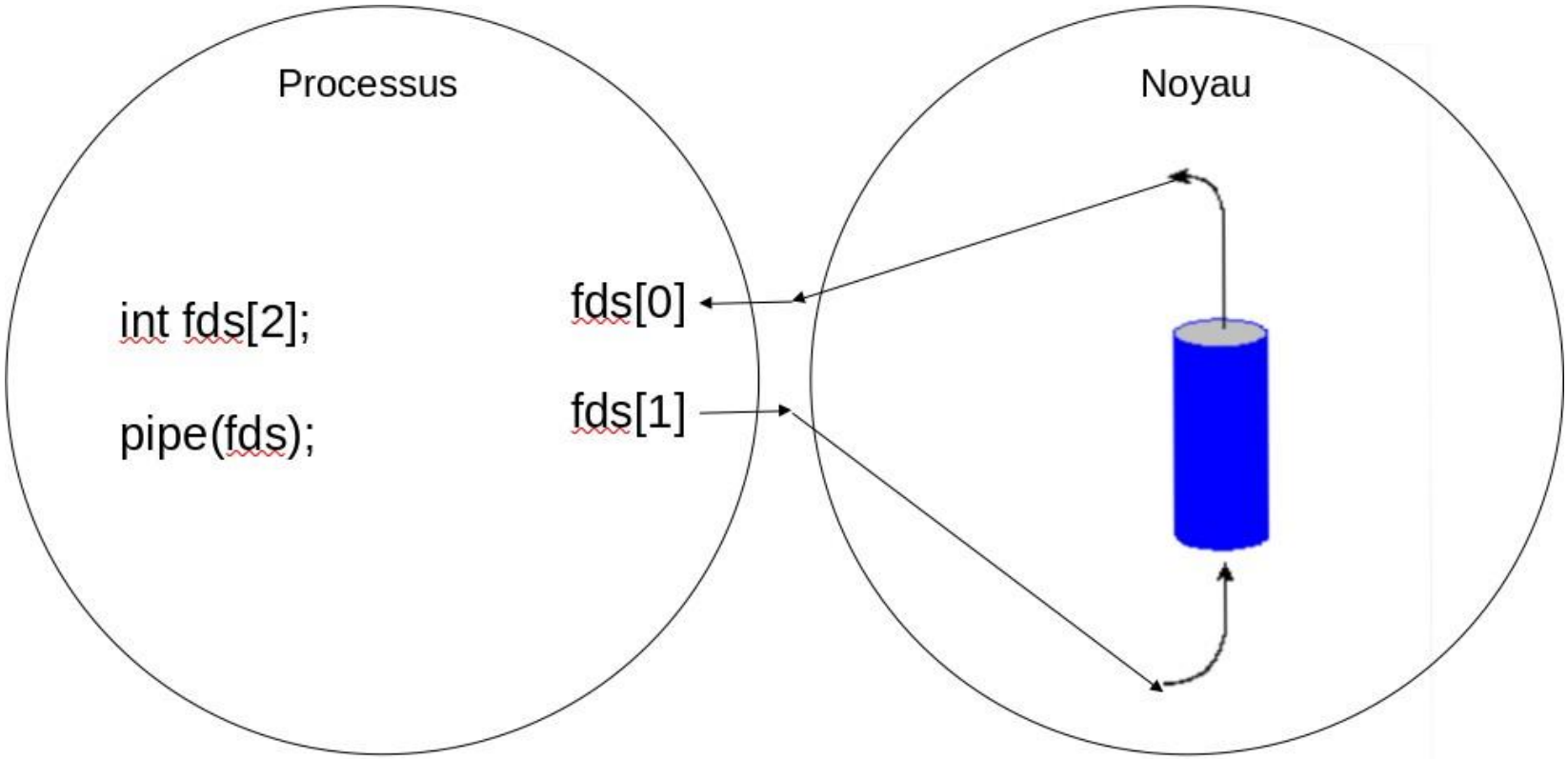
Tube anonymes : Principe



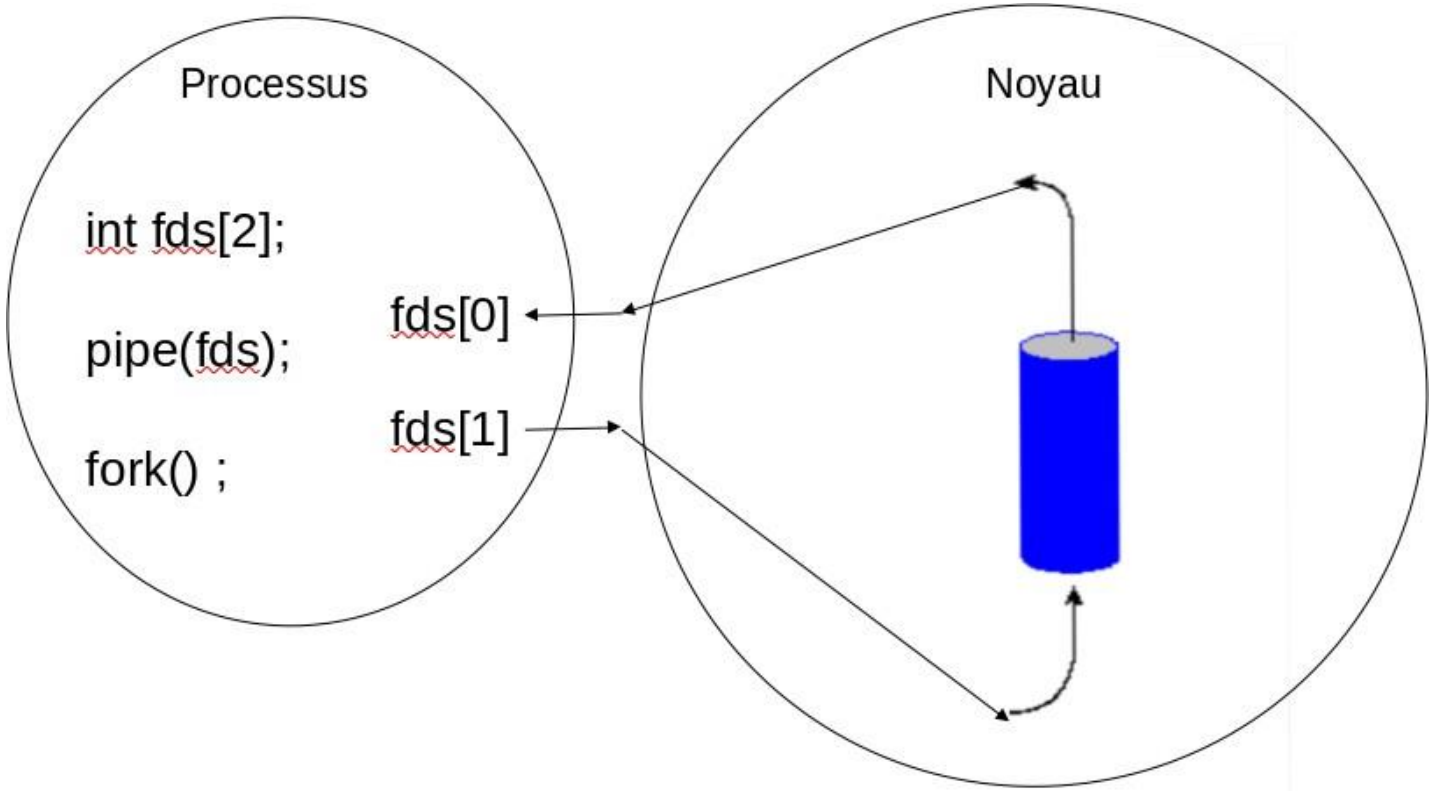
Tube anonymes : Principe



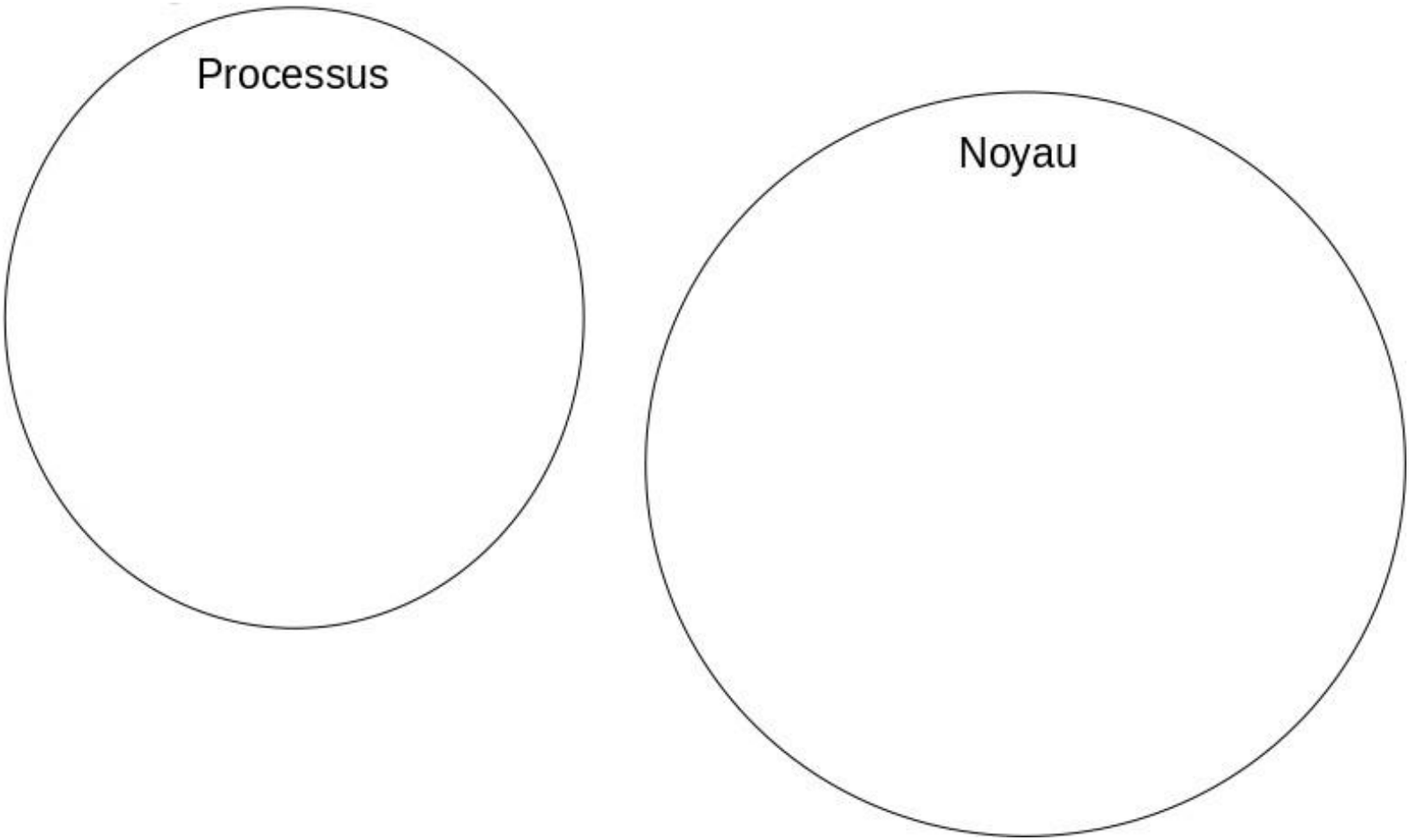
Tube anonymes : Principe



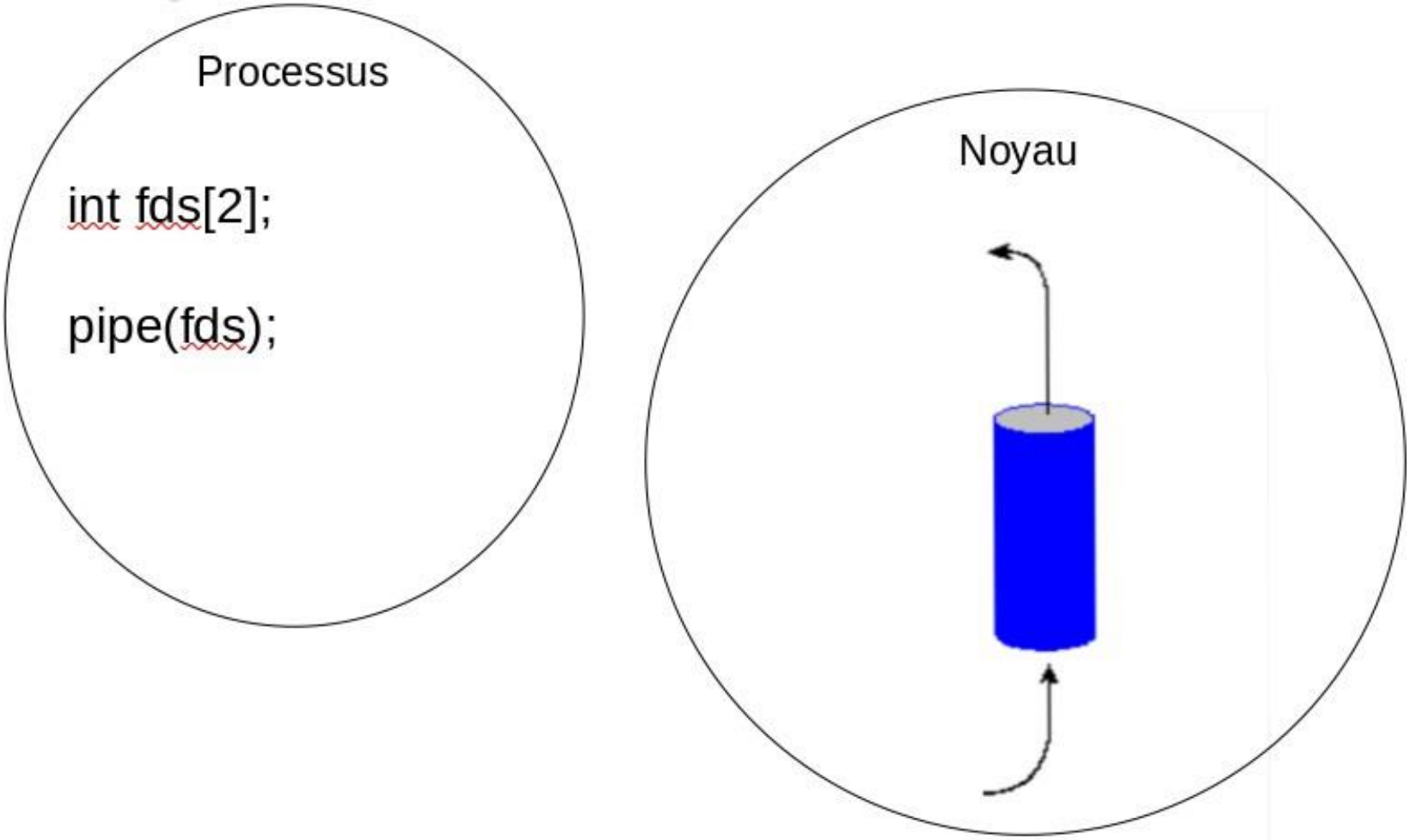
Tube anonymes : Principe



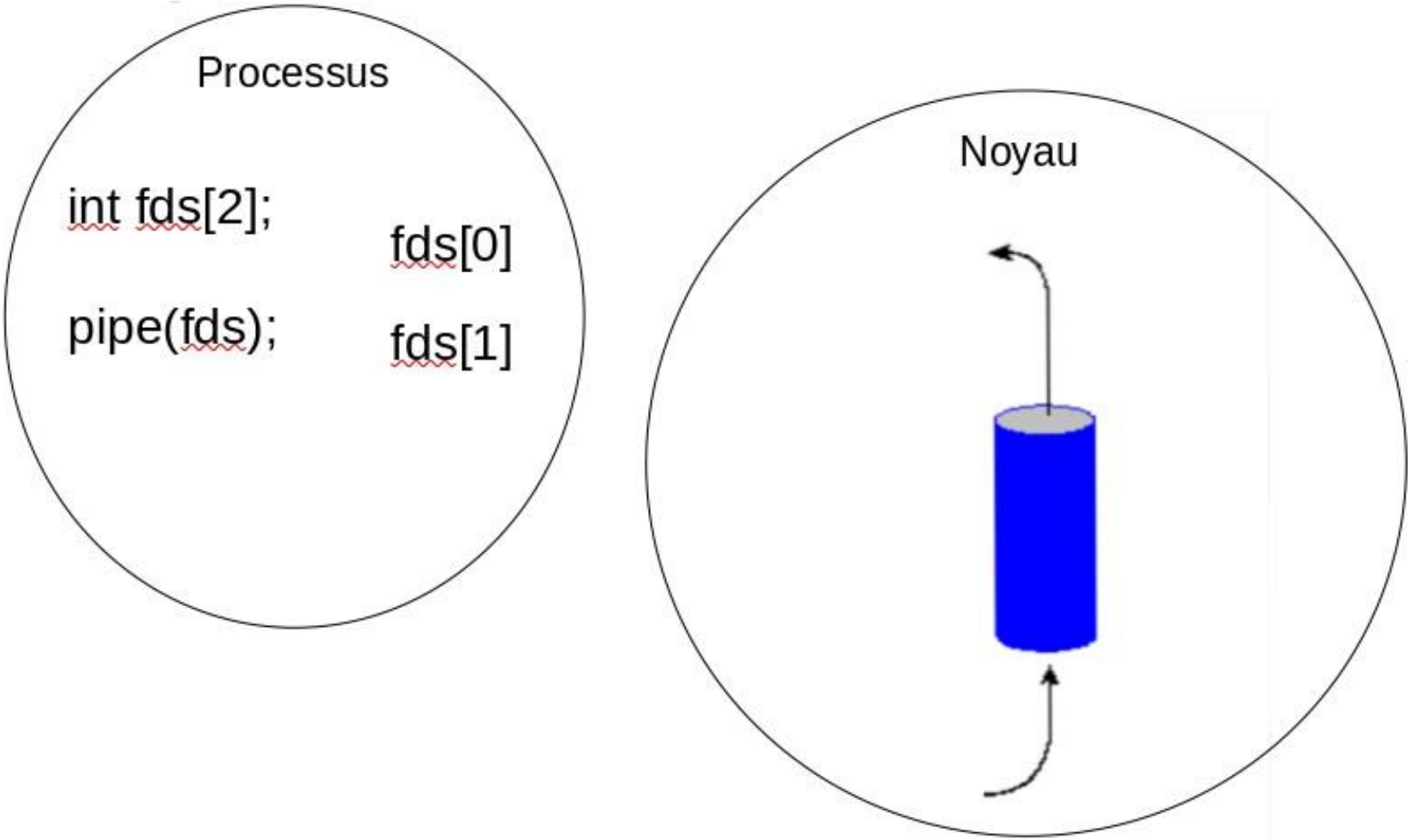
Tube anonymes : Principe



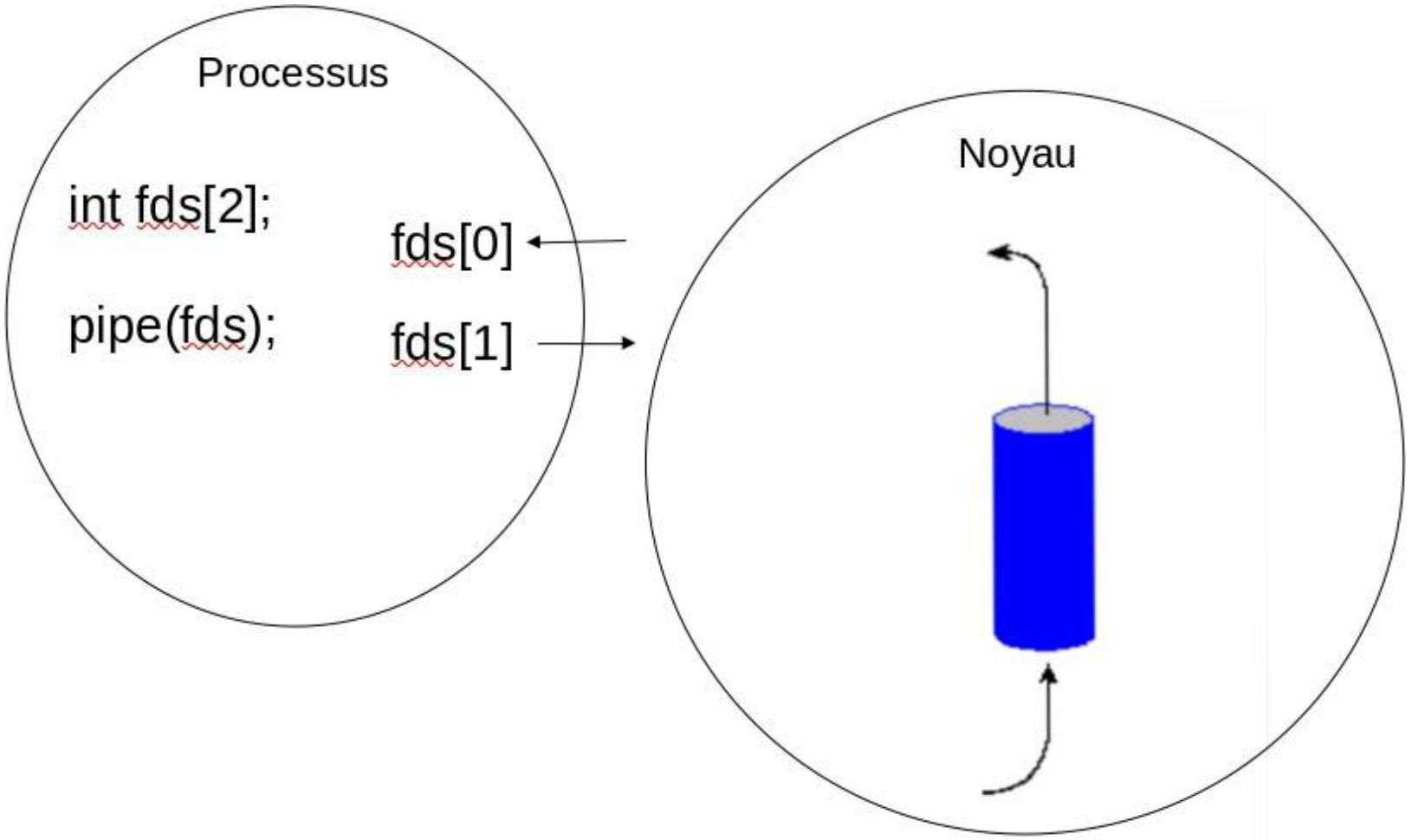
Tube anonymes : Principe



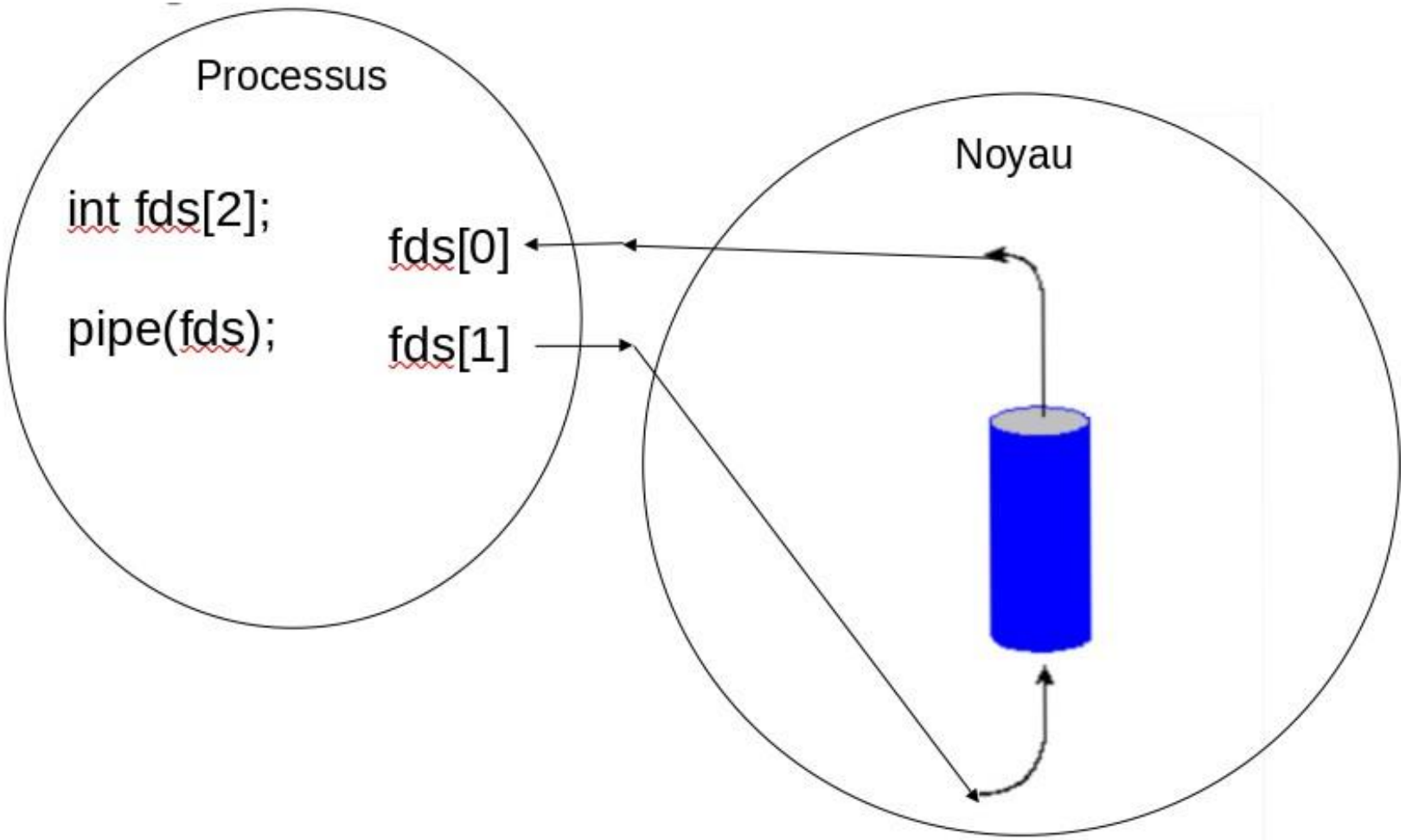
Tube anonymes : Principe



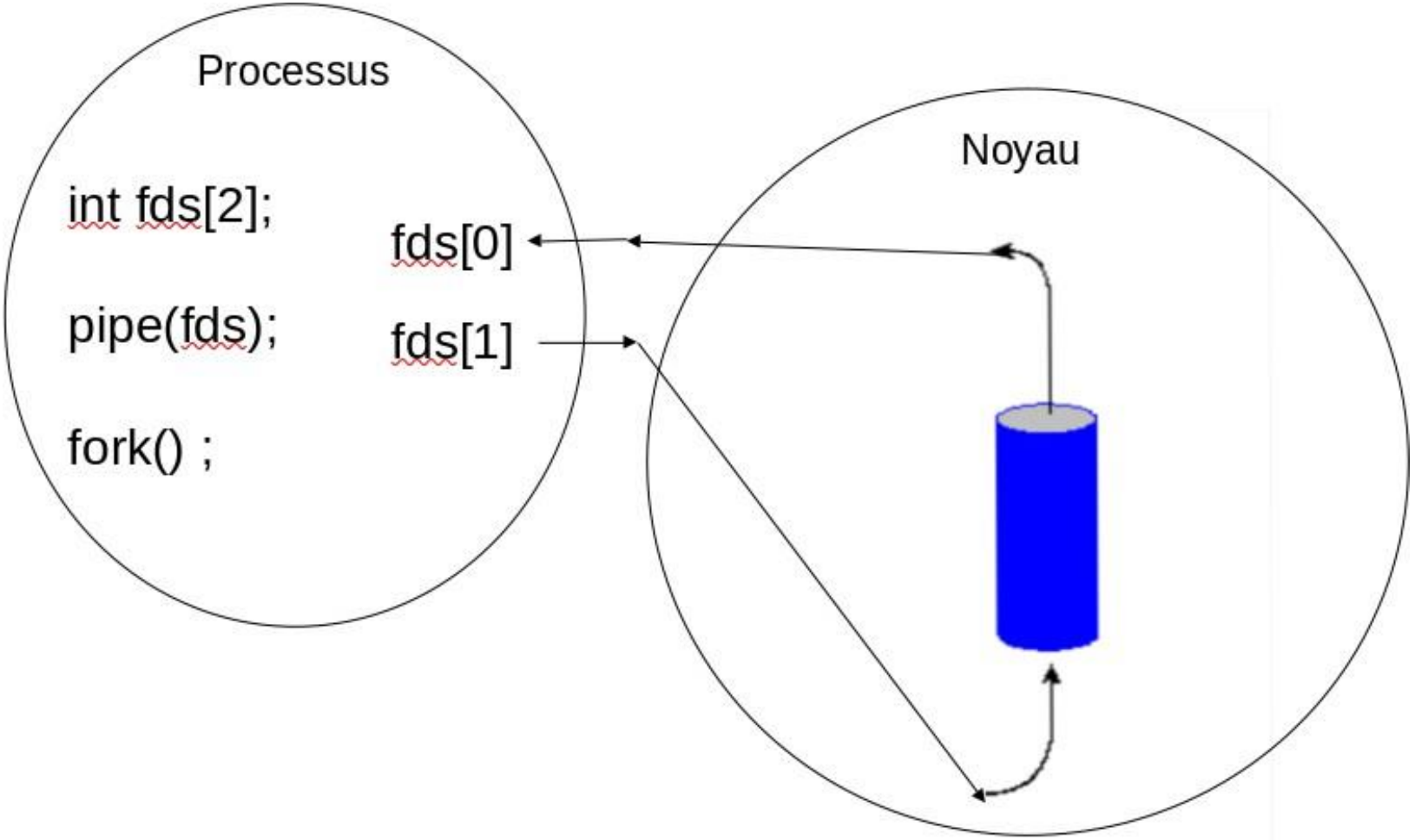
Tube anonymes : Principe



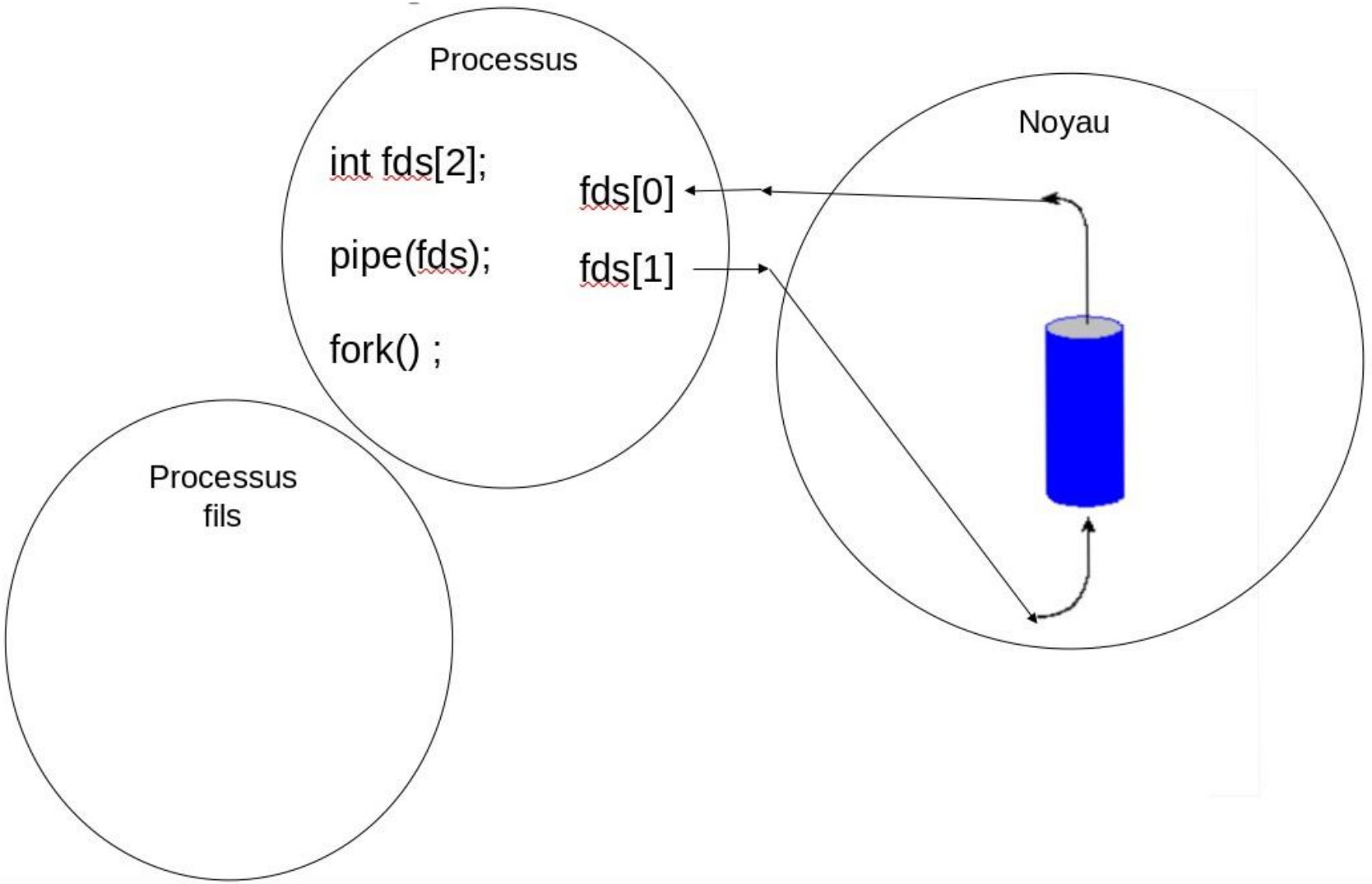
Tube anonymes : Principe



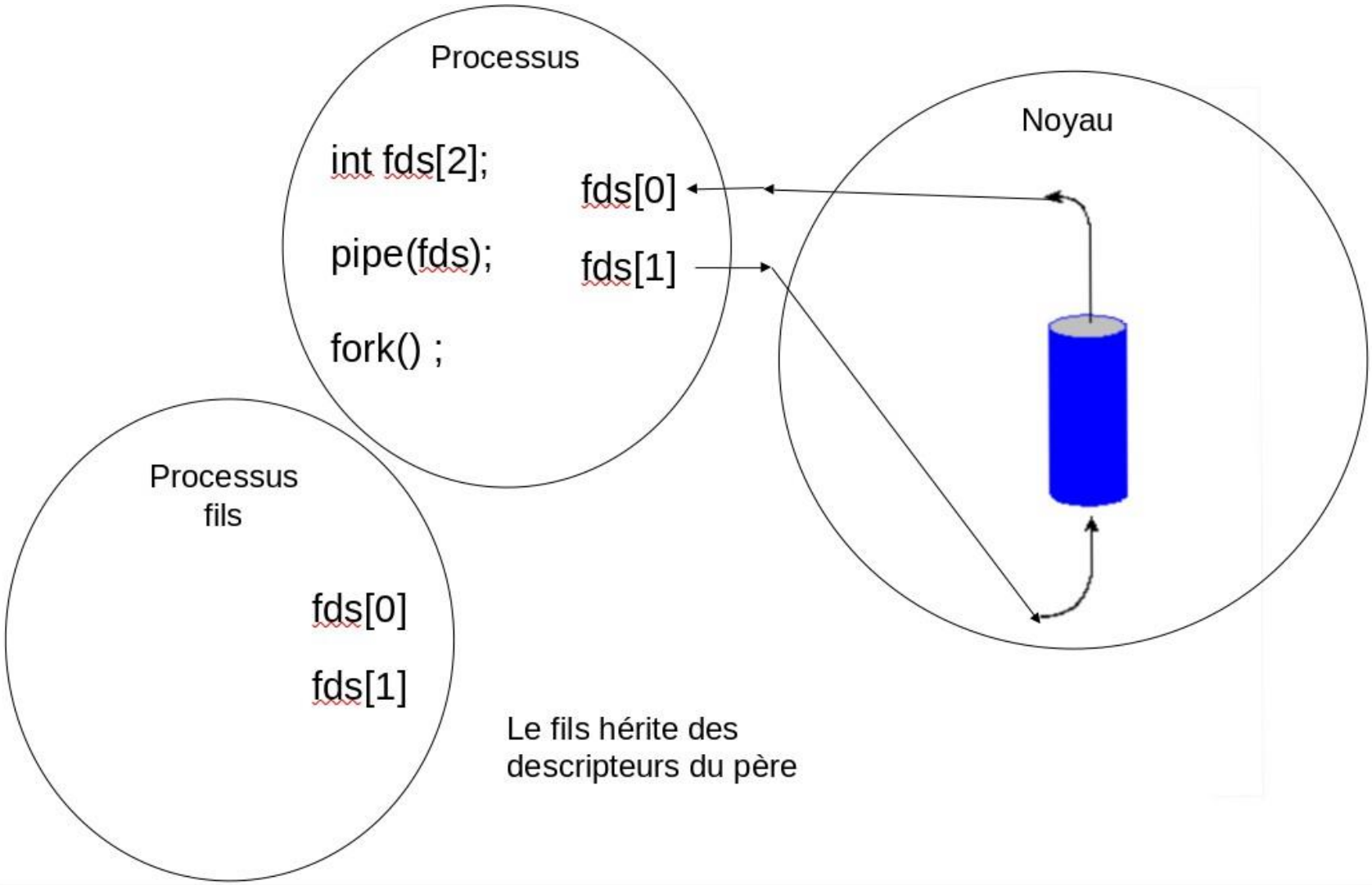
Tube anonymes : Principe



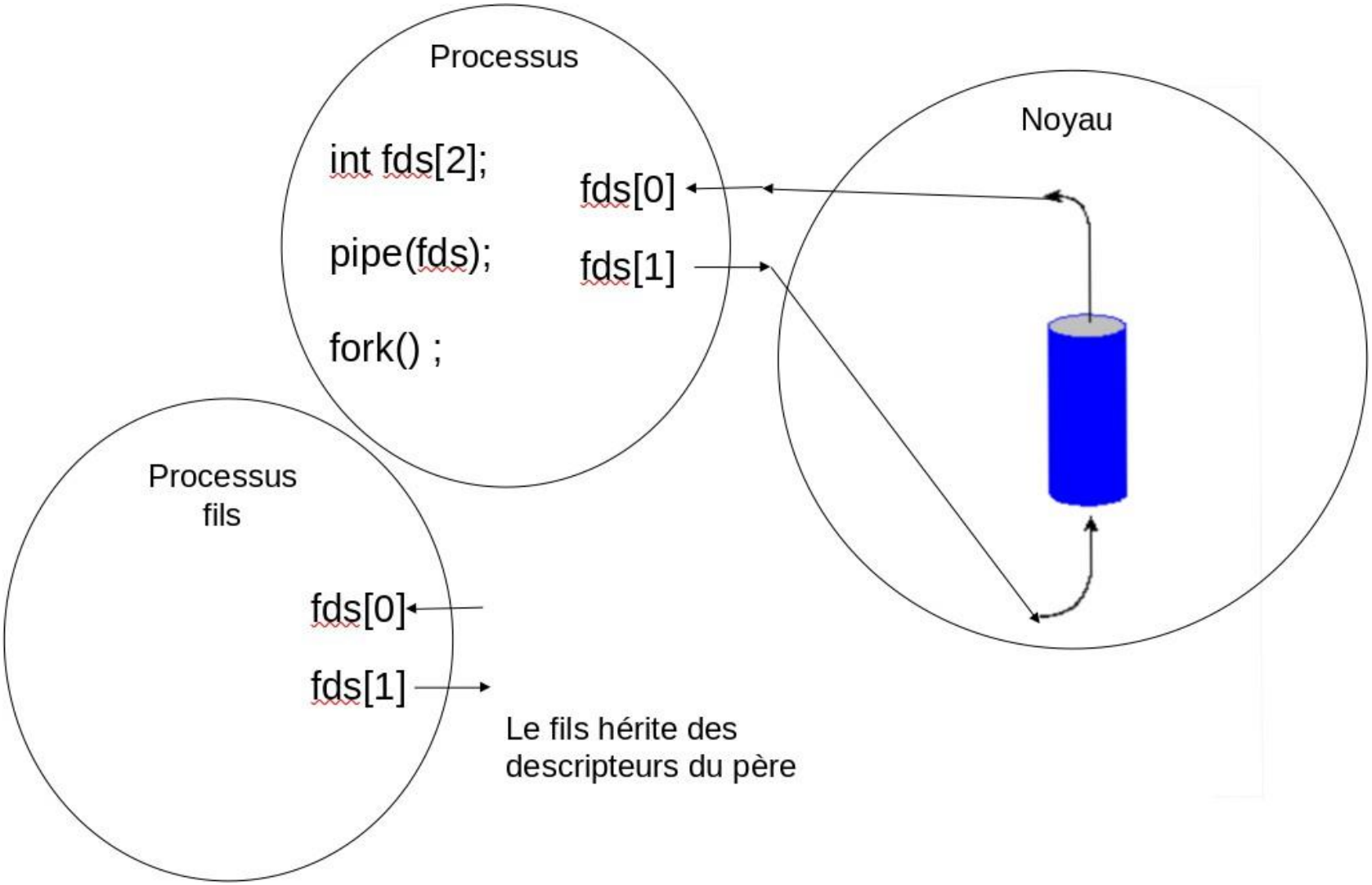
Tube anonymes : Principe



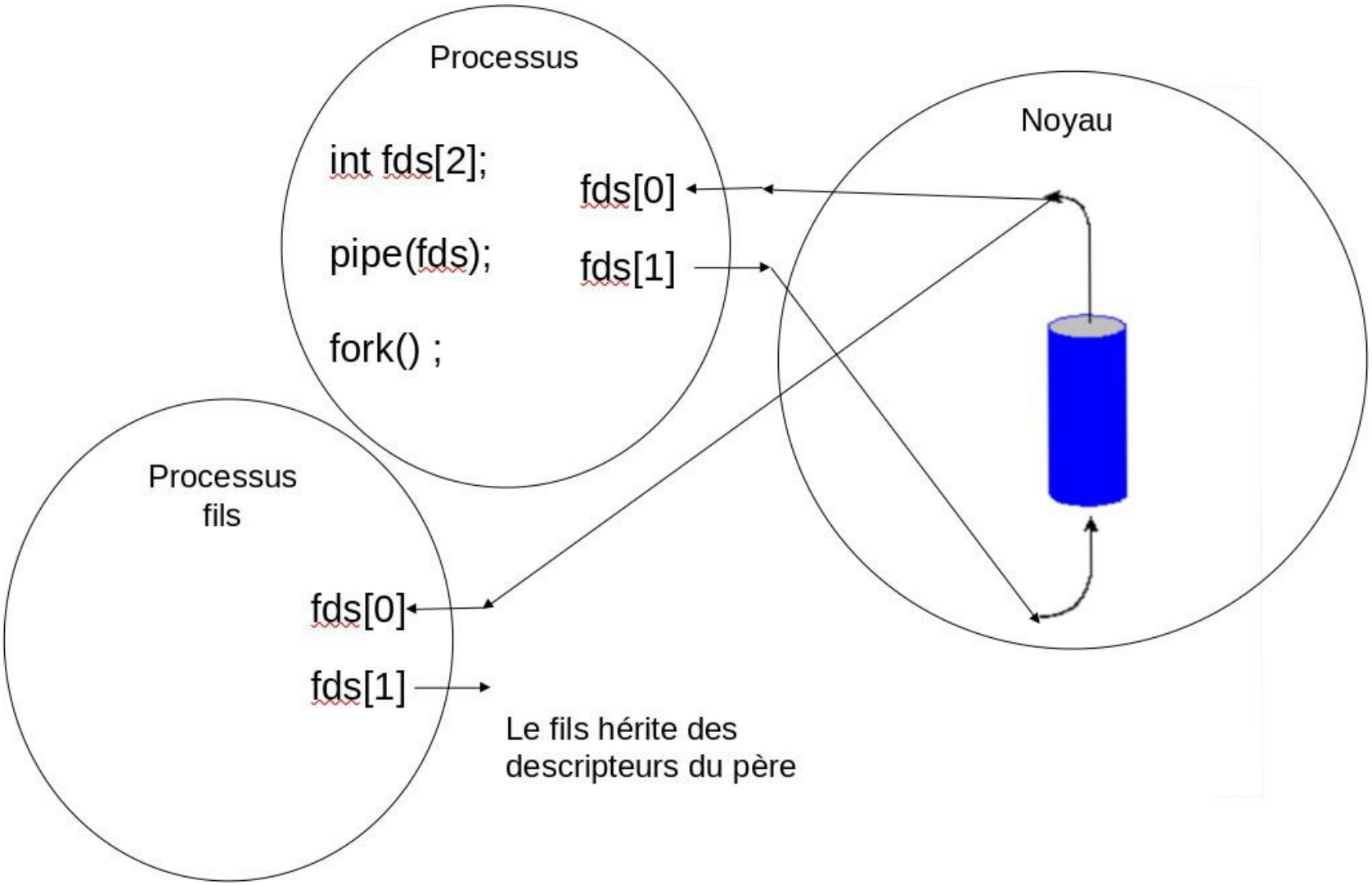
Tube anonymes : Principe



Tube anonymes : Principe

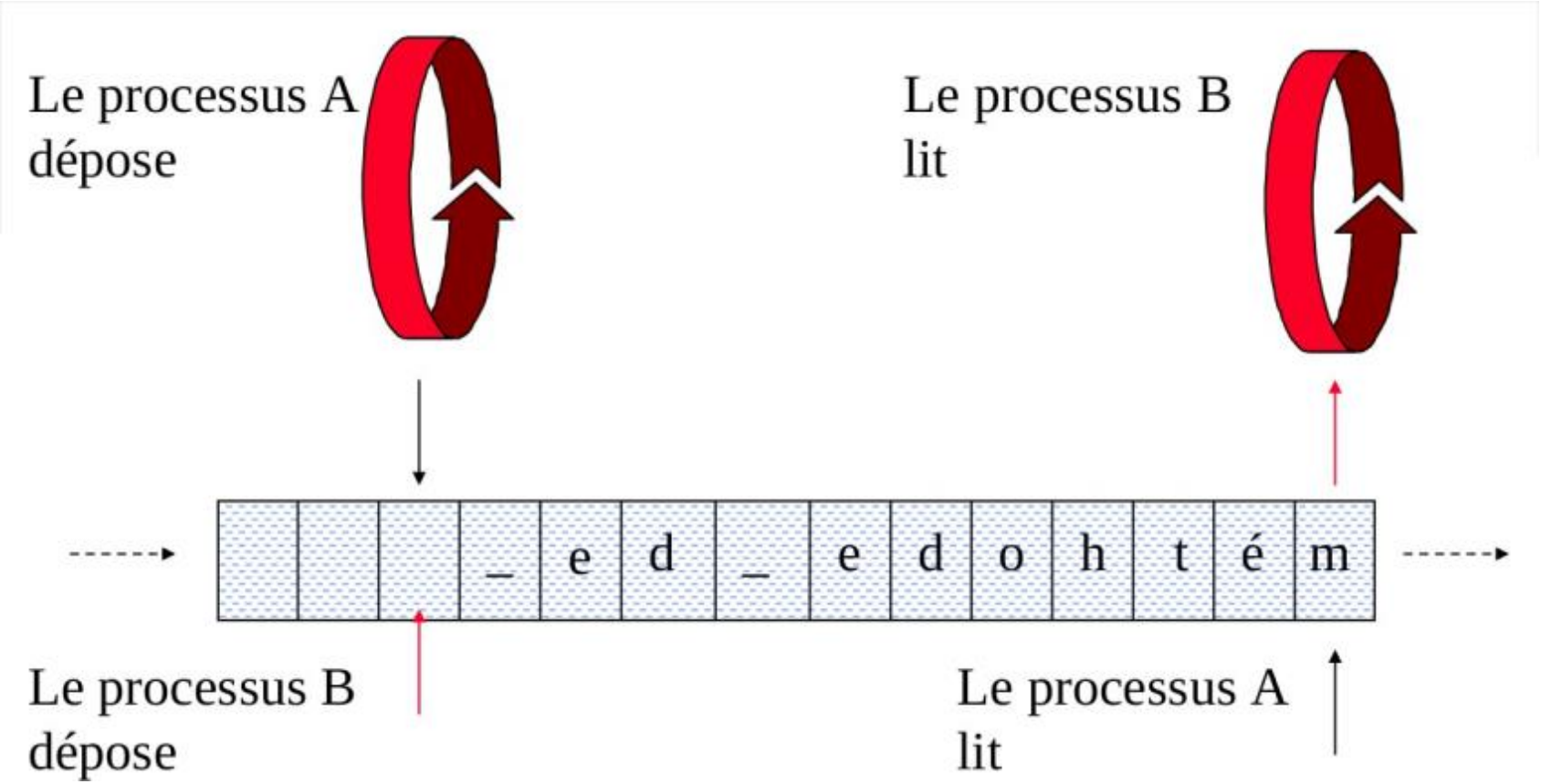


Tube anonymes : Principe



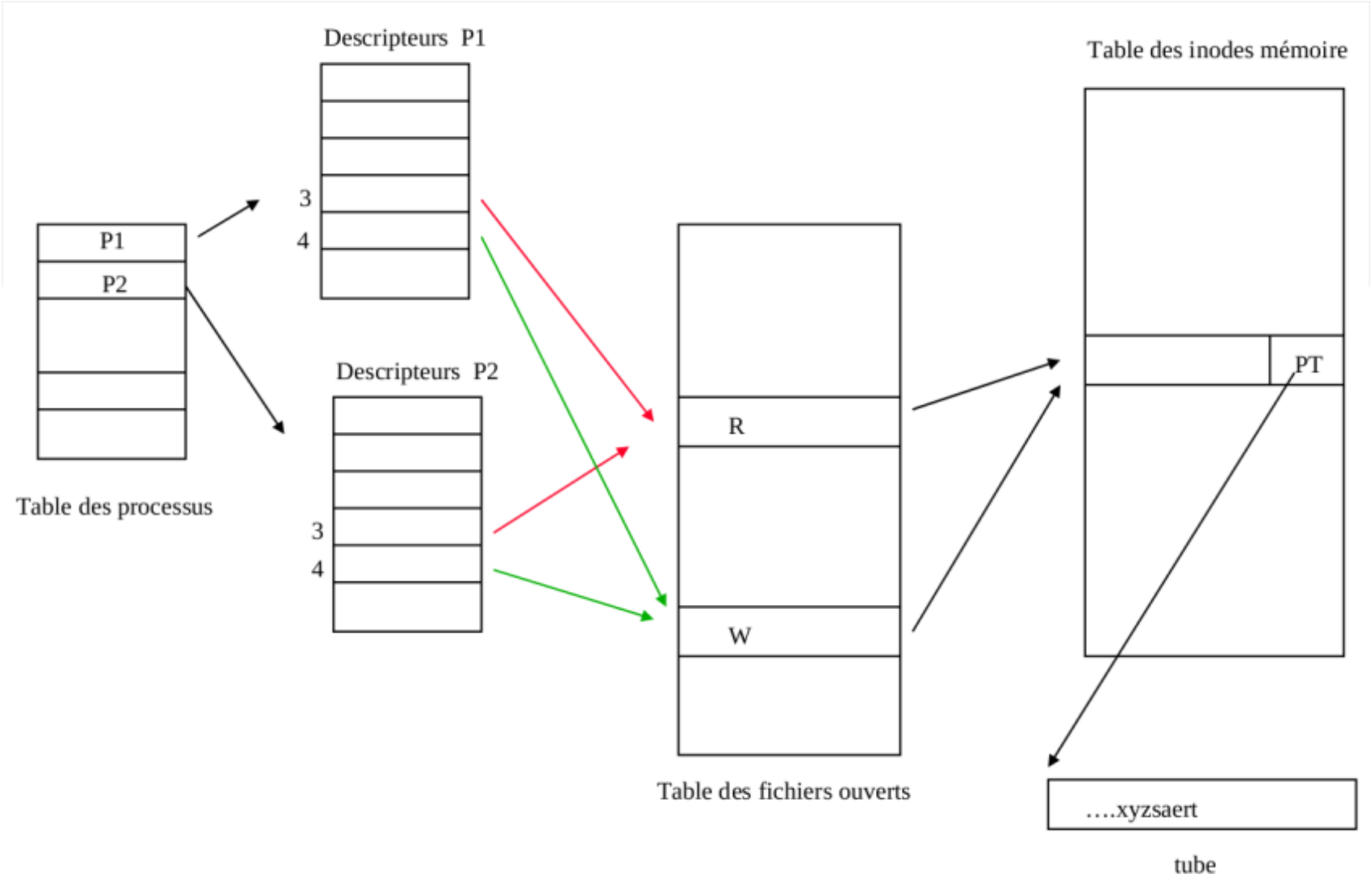
Tube anonymes : Principe

Fonctionnement

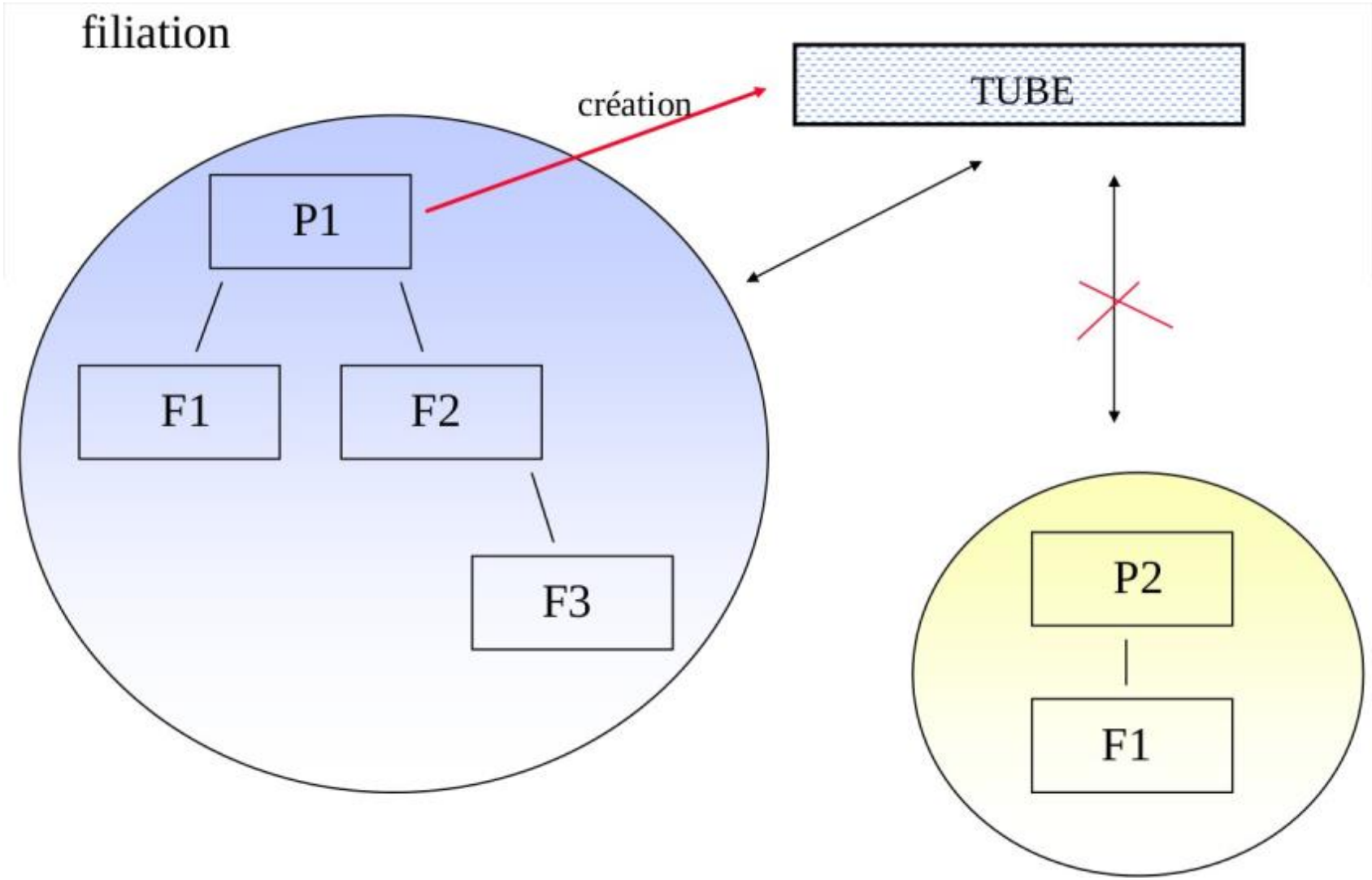


Tube anonymes : Principe

Détails :



Tube anonymes : Principe



Tubes anonymes : primitives

- Un tube anonyme est considéré comme étant fermé lorsque tous les descripteurs en lecture et en écriture existants sur ce tube sont fermés.

- Fermeture d'un descripteur :

```
int close(int desc);
```

Tubes anonymes : lecture

- La lecture dans un tube s'effectue avec la primitive **read()** :

```
int read(int desc[0],char * buf,int nb);
```

- Elle permet la lecture de **nb** bytes depuis le tube **desc**, qui sont placés dans le tampon **buf**, et retourne le nombre de bytes réellement lus. Elle répond à la sémantique suivante :
  - si le tube n'est pas vide et contient **taille** caractères, la primitive extrait du tube **min(taille, nb)** 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. Le processus 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 de fichier est atteinte. Le nombre de caractères rendu est nul.

Tubes anonymes : écriture

L'écriture dans un tube s'effectue avec la primitive **write()** :

```
int write(int desc[1], char * buf, int nb);
```

Elle permet d'écrire **nb** caractères, placés dans le tampon **buf**, dans le tube desc. La function retourne le nombre de caractères réellement écrit. Elle répond à la sémantique suivante :

- si le nombre de lecteurs dans le tube est nul, alors une erreur est générée et le signal **SIGPIPE** est délivré au processus écrivain, et le processus se termine. L'interpréteur de commandes shell affiche par défaut le message «***Broken pipe*** » ;

- si le nombre de lecteurs dans le tube est non nul, l'opération d'écriture est bloquante jusqu'à ce que les nb caractères aient effectivement été écrit dans le tube.

Exemple de tube anonyme

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int  main() {
const int Nbuff=1000;
char buff[Nbuff];
int fds[2], pid, n, status;
pipe(fds);
if ((pid==fork()) > 0) { // pere
close(fds[0]); write(fds[1],"Salut",5); wait(&status);}
else { // fils
close(fds[1]);
n = read(fds[0],buff,Nbuff-1); buff[n] = '\0';
printf("%s\n",buff); exit(0); }
return  0; }
}
```

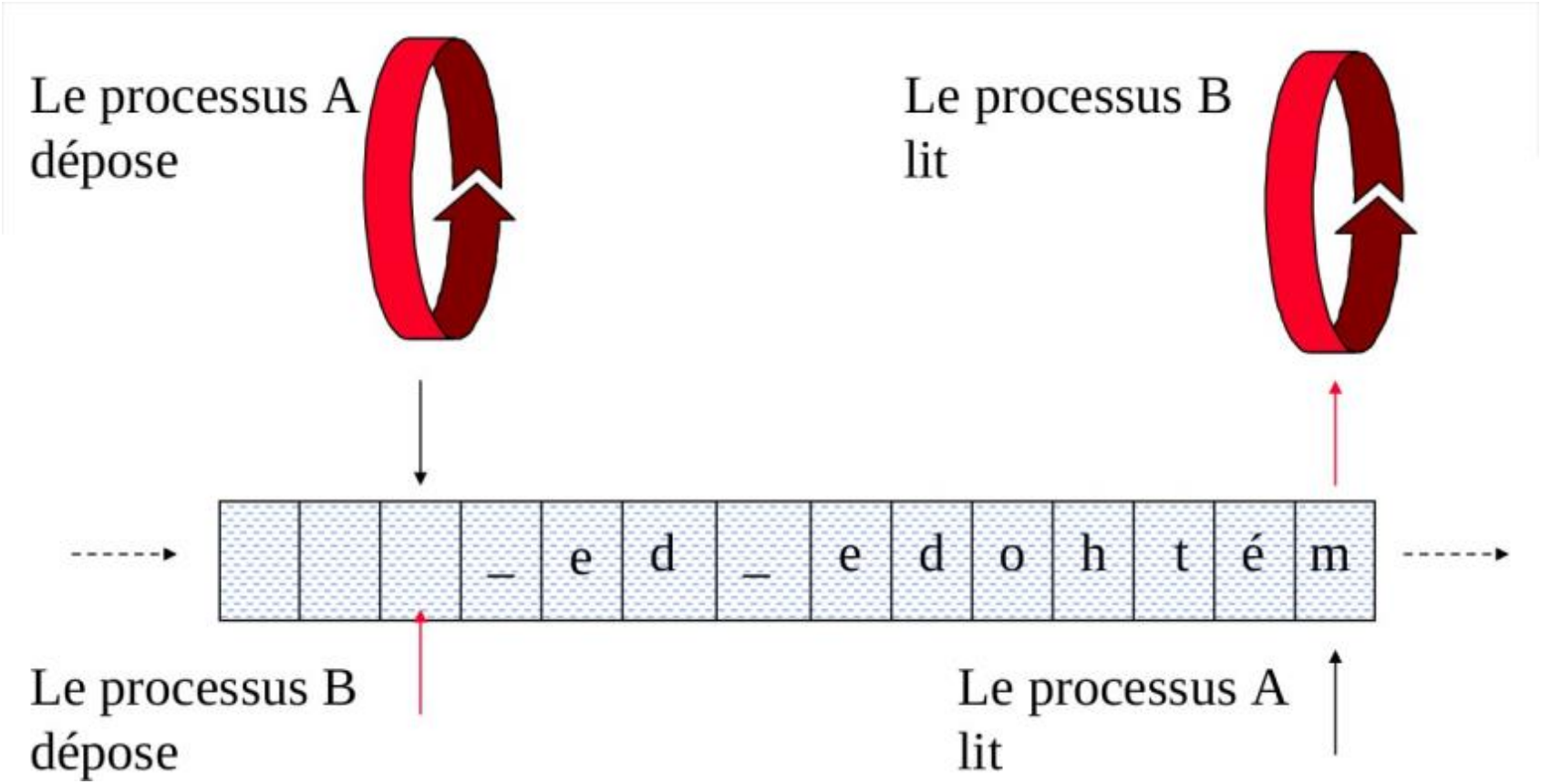
Tube avec deux processus : communication unidirectionnelle

Supposons que le père écrit dans le tube alors que le fils lit dans le tube.

1. Le processus crée un tube.
2. Le processus fait appel à `fork()` pour créer un fils.
3. Les deux processus père et fils possèdent alors chacun un descripteur en lecture et en écriture sur le tube.
4. Le processus père ferme son descripteur en lecture. Le processus fils ferme son descripteur en écriture sur le tube.
5. Le processus père peut écrire sur le tube ; les valeurs écrites pourront être lues par le fils.

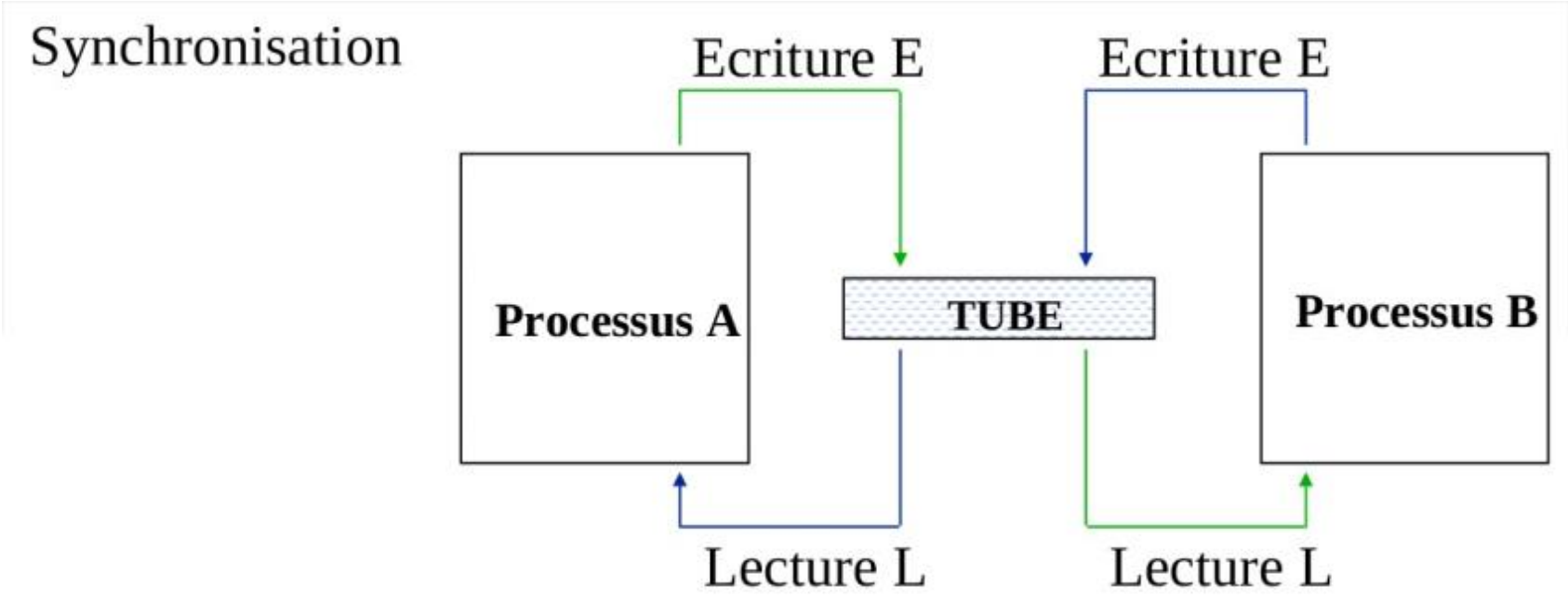
Tube avec deux processus : communication unidirectionnelle

Fonctionnement



Tube avec deux processus : communication unidirectionnelle

Fonctionnement sans synchronisation

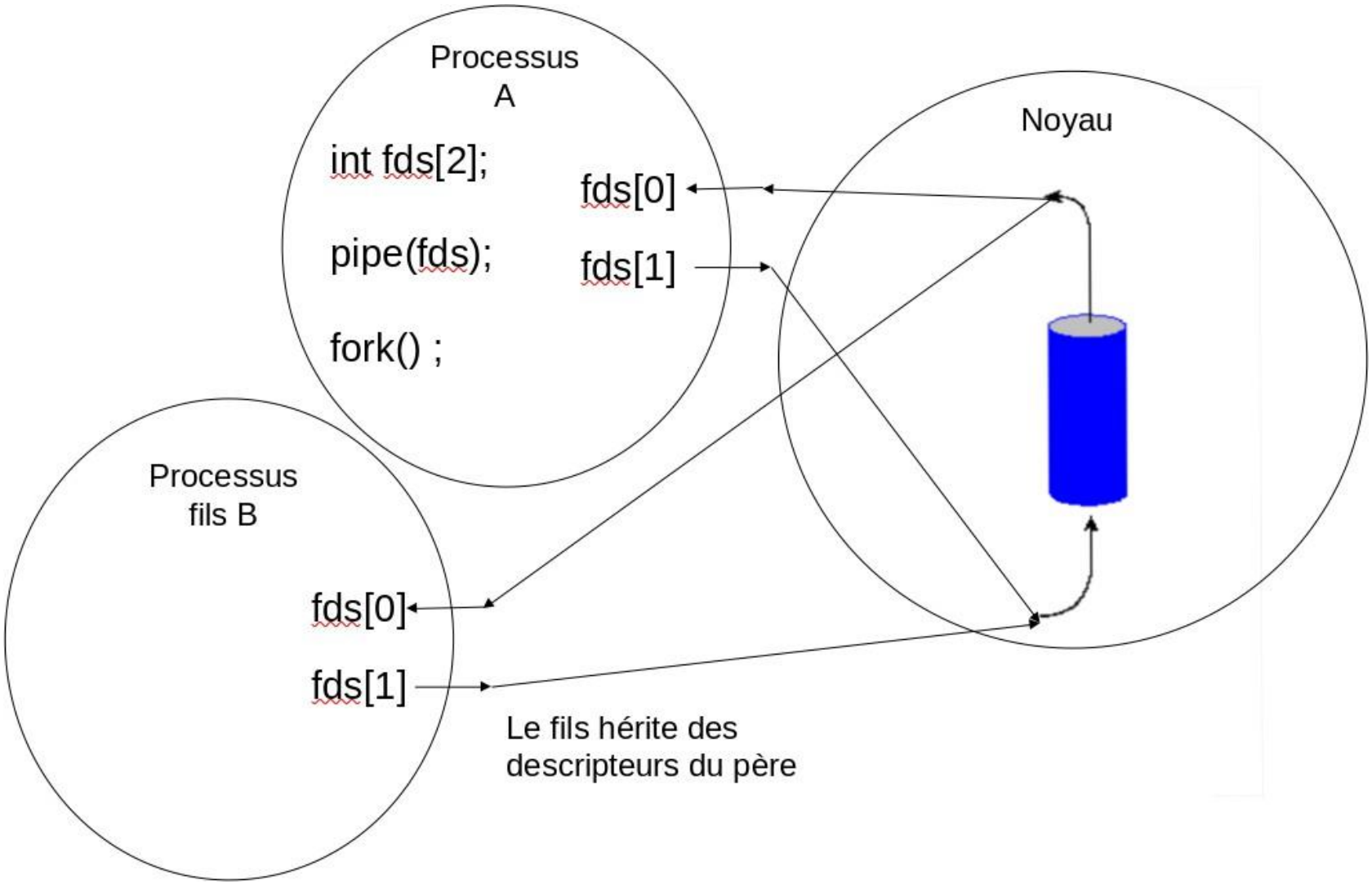


- Soit: PA transmet à PB ou PB transmet à PA

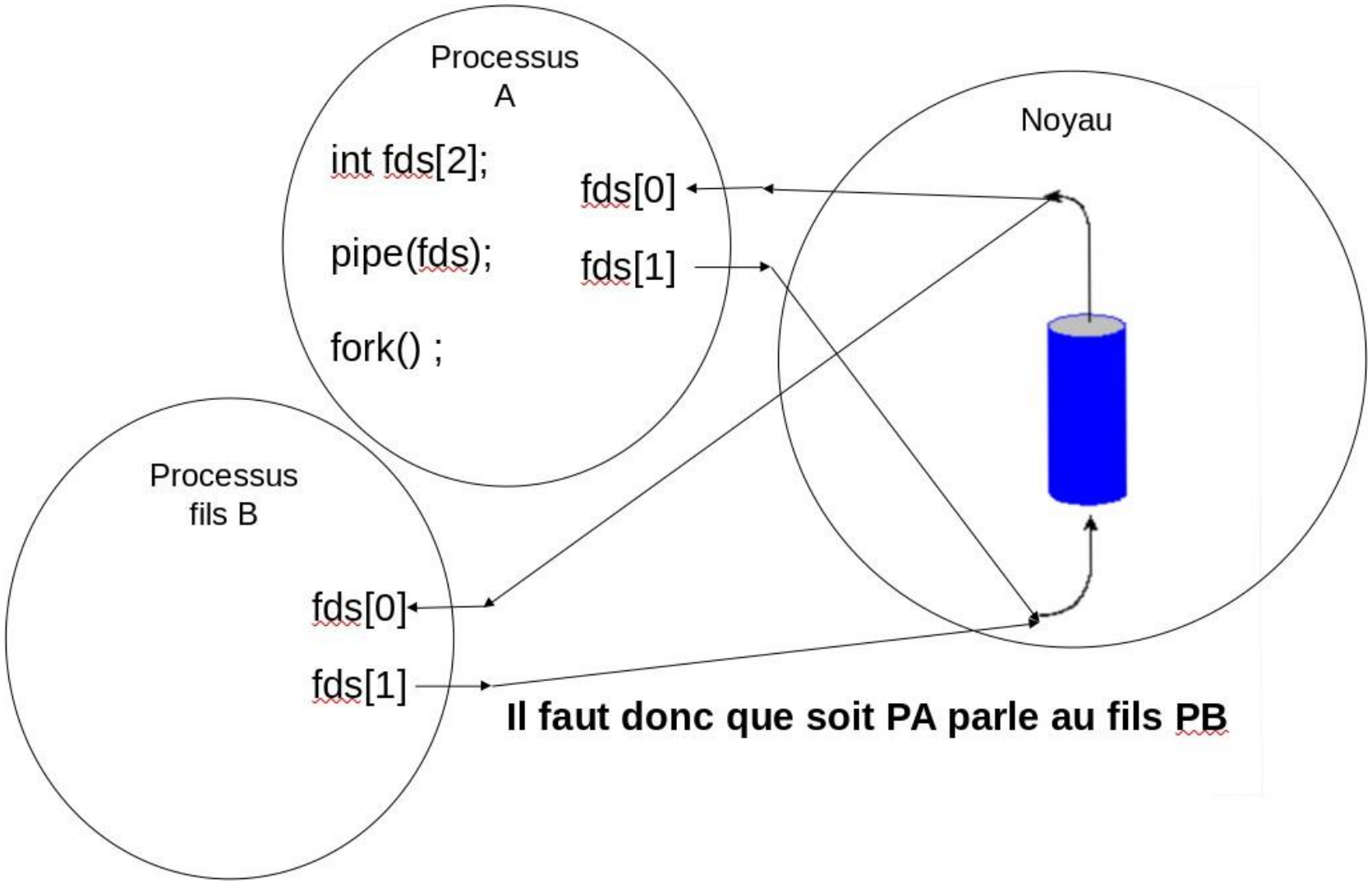
SI

- PA dépose et PA lit => PB bloqué
- PA et PB déposent et PB lit => risque que PB lise sa propre donnée

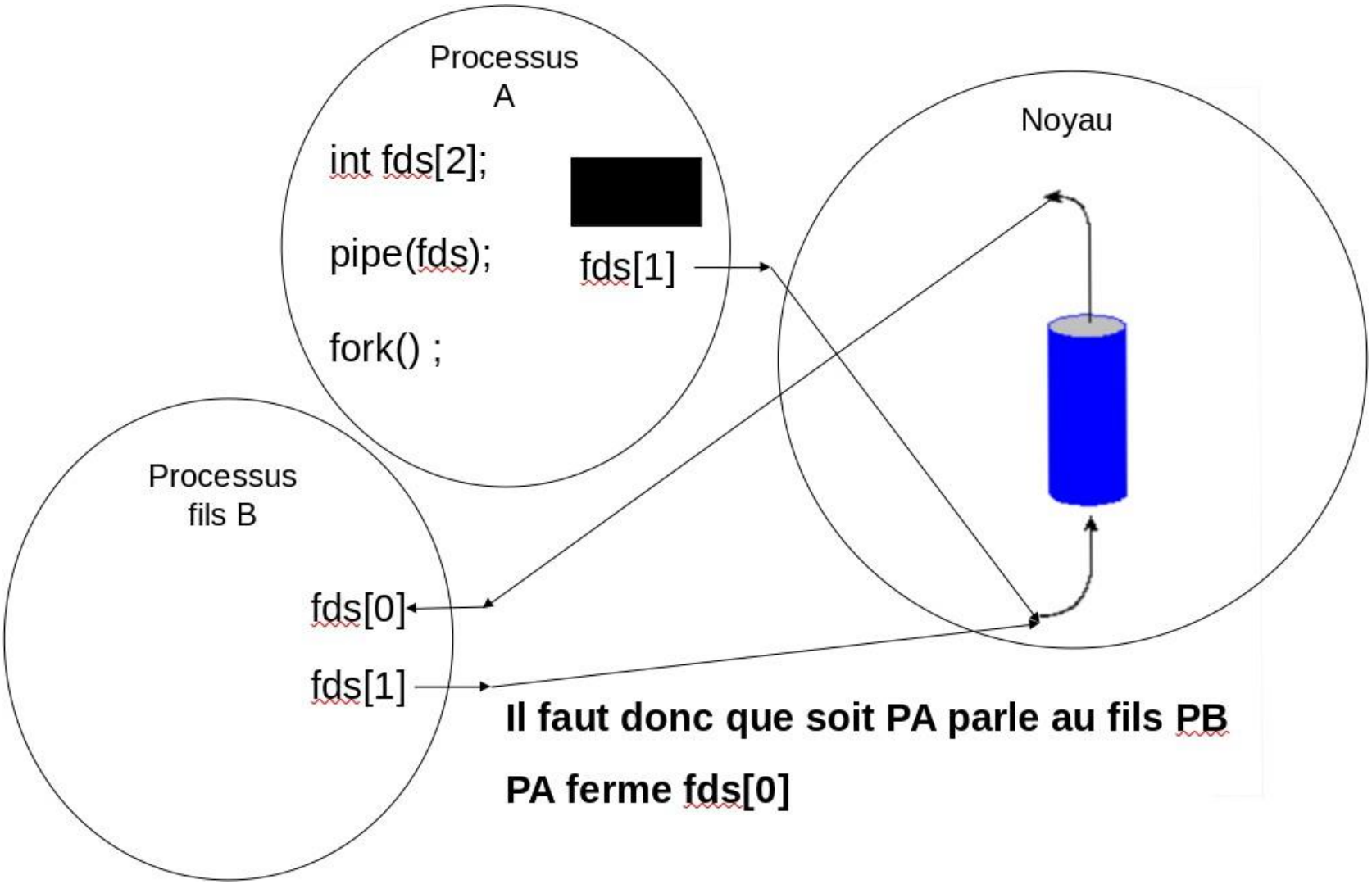
Tube avec deux processus : communication unidirectionnelle



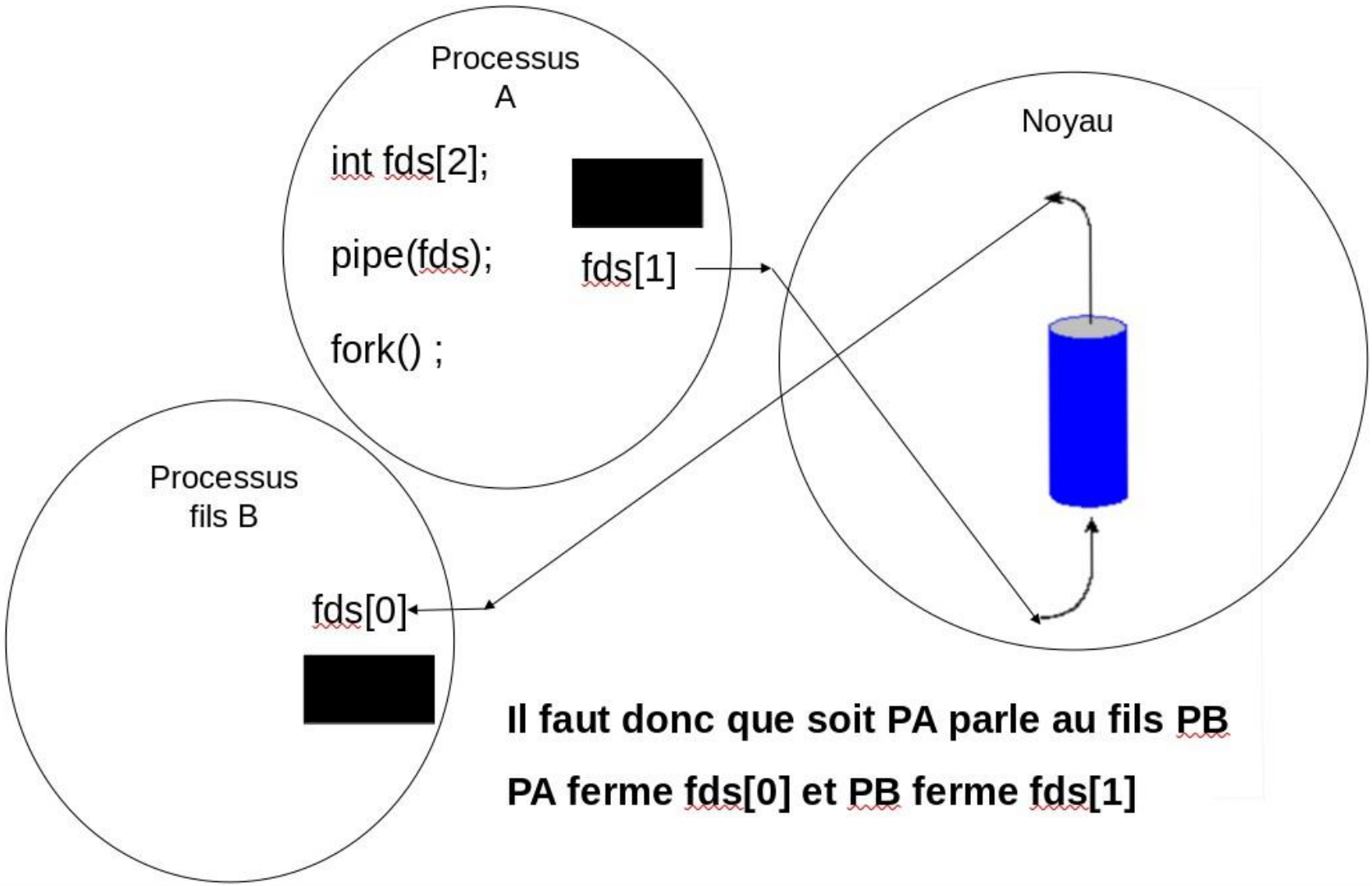
Tube avec deux processus : communication unidirectionnelle



Tube avec deux processus : communication unidirectionnelle



Tube avec deux processus : communication unidirectionnelle



Tube avec deux processus : communication unidirectionnelle

Code :

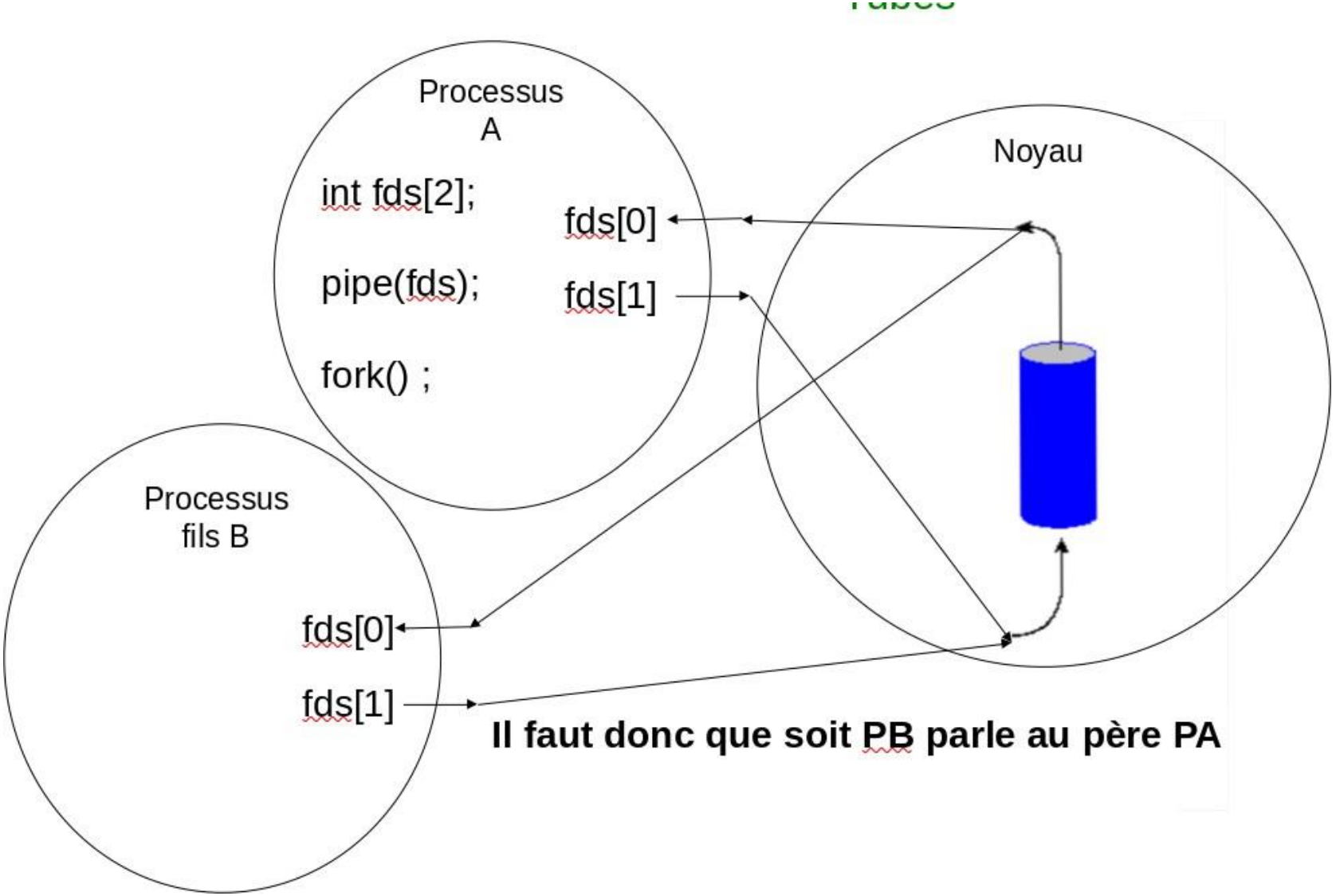
```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int   main() {
const int Nbuff=1000;
char buff[Nbuff];
int fds[2], pid, n, status;
pipe(fds);
if ((pid==fork()) > 0) { \\pere
    close(fds[0]); write(fds[1],"Salut",5);
    wait(&status);
}
else { //  fils
    close(fds[1]);
    n = read(fds[0],buff,Nbuff-1); buff[n] = '\0';
    printf("%s\n",buff); exit(0);
}
return  0;
}
```

Tube avec deux processus : communication unidirectionnelle

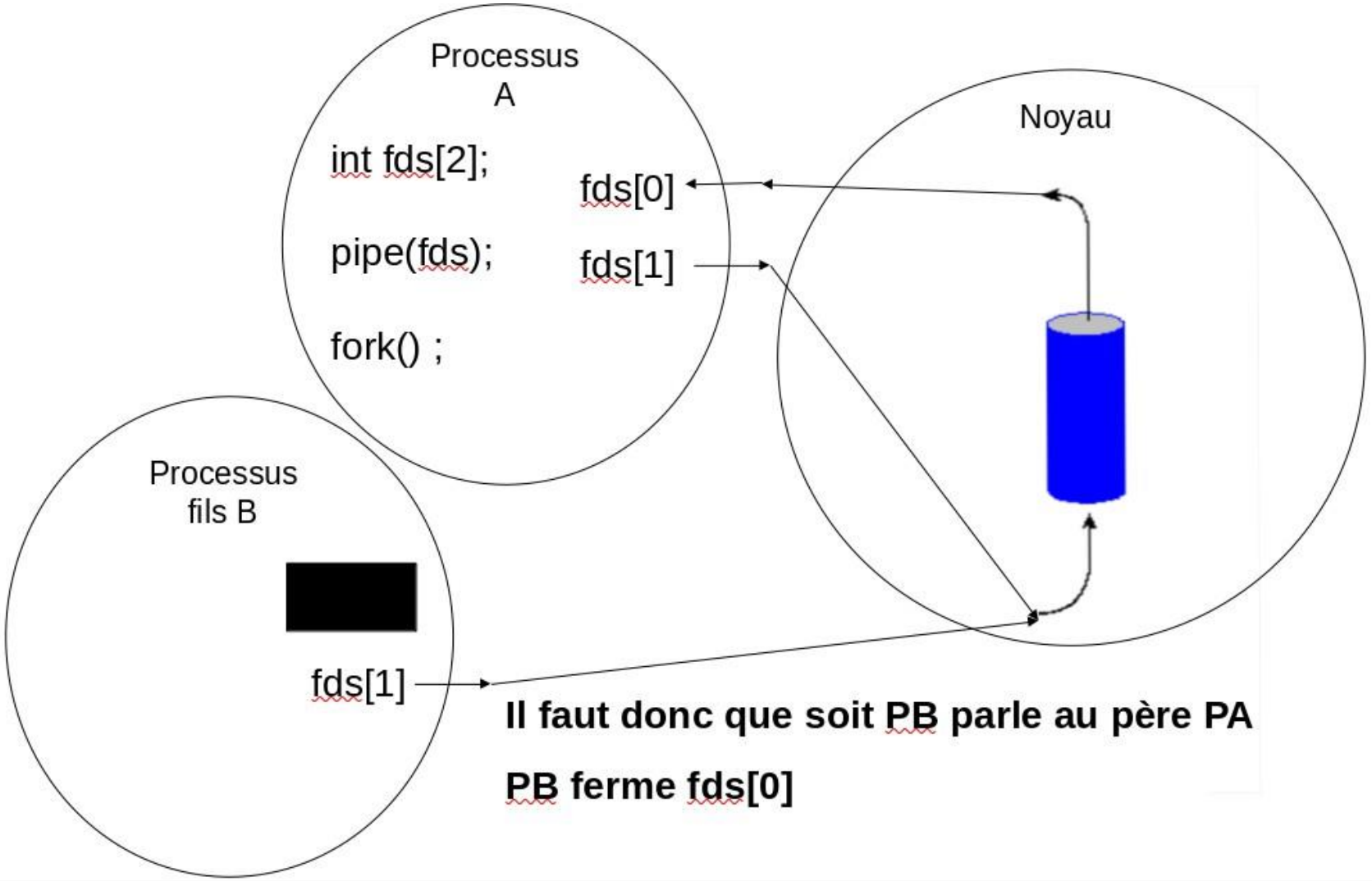
Code :

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int   main() {
const int Nbuff=1000;
char buff[Nbuff];
int fds[2], pid, n, status;
pipe(fds);
if ((pid==fork()) > 0) { //  pere
     close(fds[1]);
     n = read(fds[0],buff,Nbuff-1); buff[n] = '\0';
     printf("%s\n",buff);
     wait(&status);
}
else { //  fils
     close(fds[0]); write(fds[1],"Salut",5);
     exit(0);
}
return  0;
}
```

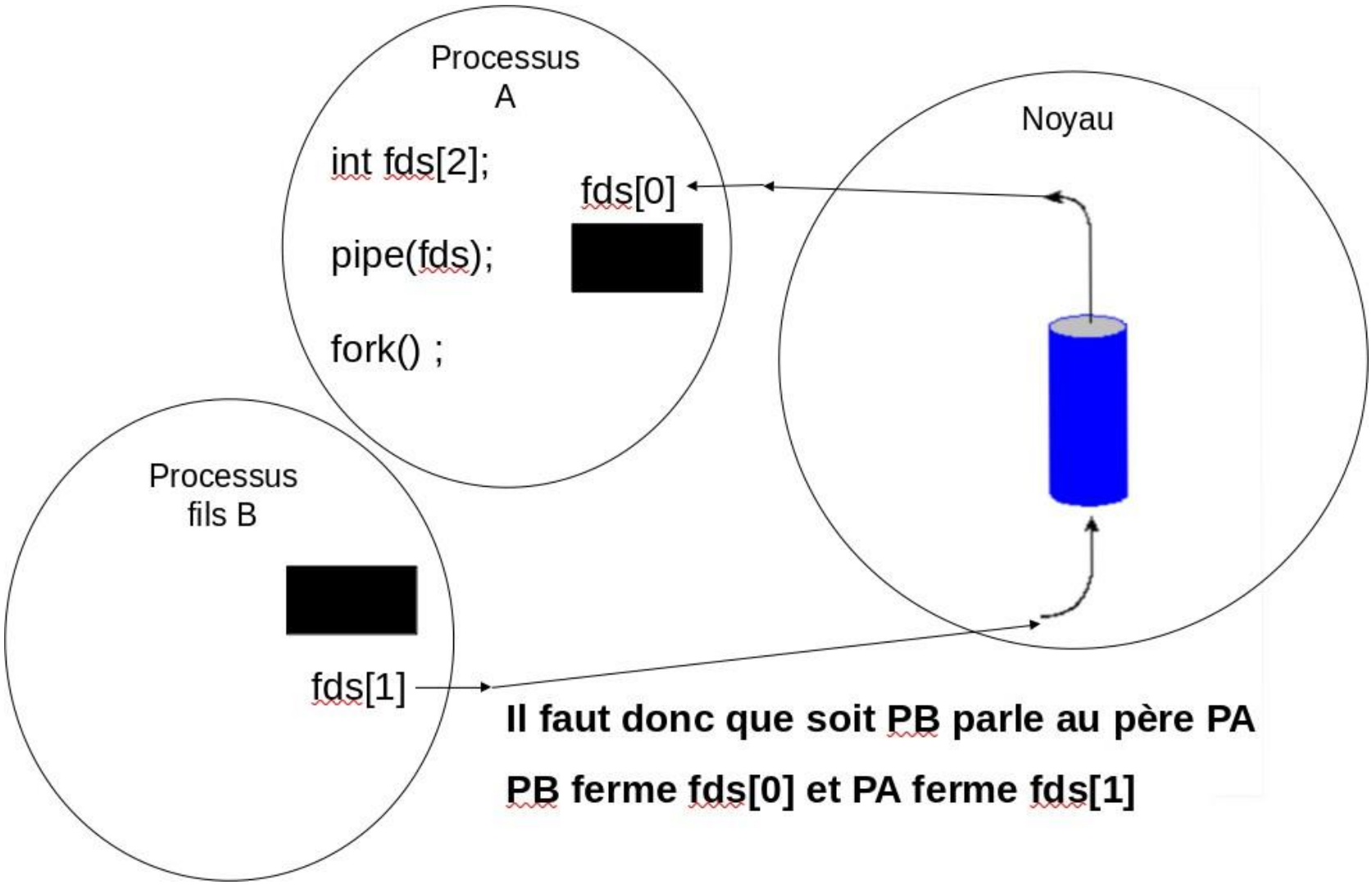
Tube avec deux processus : communication unidirectionnelle



Tube avec deux processus : communication unidirectionnelle



Tube avec deux processus : communication unidirectionnelle

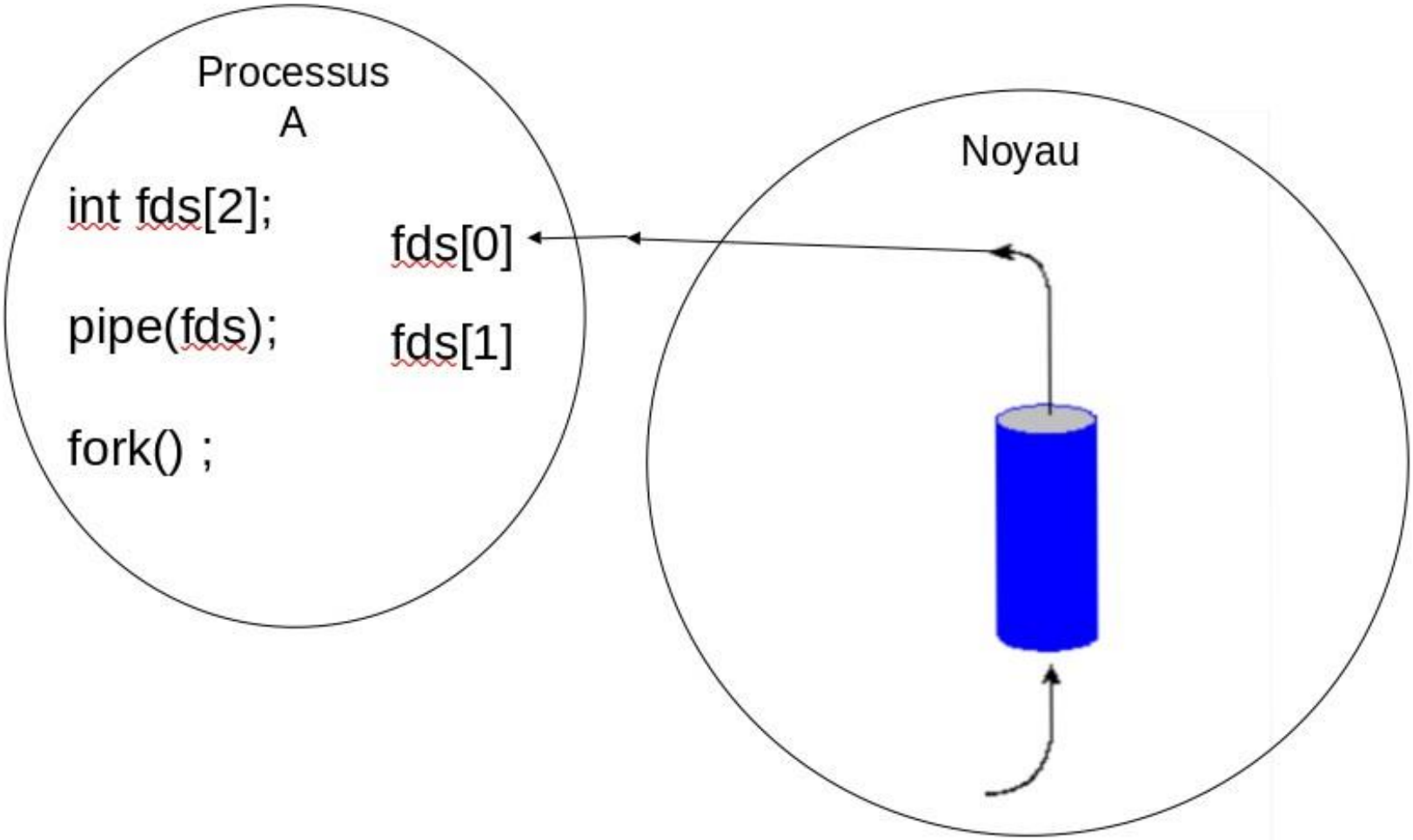


Tube avec deux processus : communication unidirectionnelle

Autre exemple :

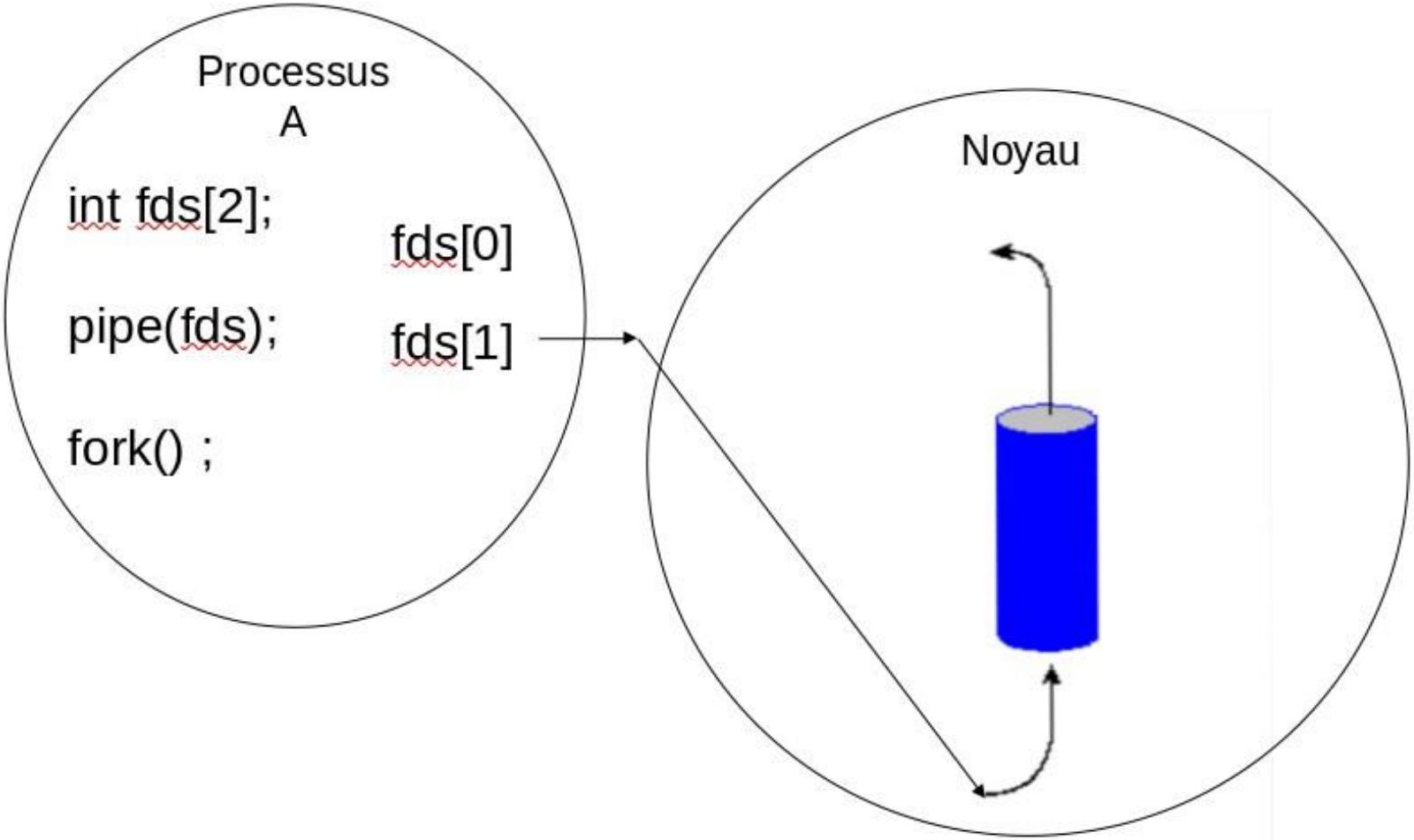
```c
#include <stdio.h>
#include<unistd.h>
int main()
{
  int p[2];
  char buf[20];

  pipe(p);
  switch (fork())
  {
    case -1: exit(1);
    case 0:
      close(p[1]);
      read(p[0], buf, sizeof(buf));
      printf("% s bien recu \n",buf);
      break;
    default:
      close(p[0]);
      write(p[1], "Bonjour", sizeof("Bonjour"));
  }
}
```

Tube avec deux processus : communication unidirectionnelle



Attention : si lecture sans quelqu'un qui écrit après avoir tout lu, read renvoie 0

Tube avec deux processus : communication unidirectionnelle



Attention : si écriture sans quelqu'un qui lit signal SIGPIPE, write renvoie errno = EPIPE

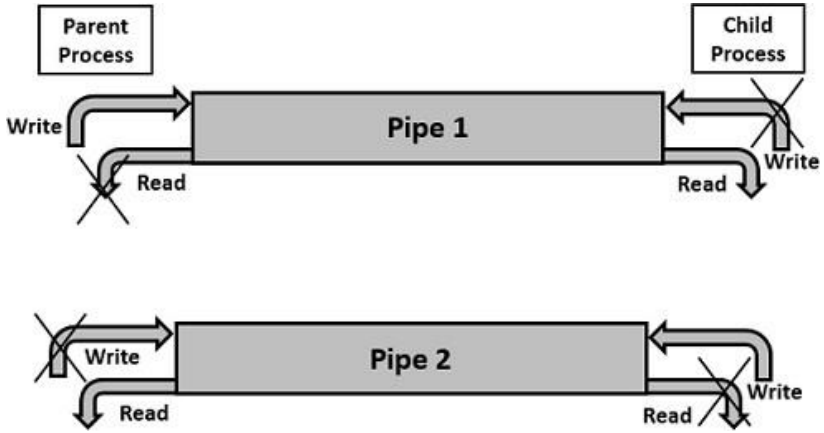
Tube avec deux processus : communication bi-directionnelle

Lecteurs et Écrivains multiples

- Supposons deux processus $P1$ et $P2$ qui utilisent le même tube dans les deux sens.
- Situations problématiques :
  - P1 écrit puis lit, donc P2 est bloqué.
  - P1 écrit, P2 écrit puis lit, donc P2 lit ce qu'il a écrit.

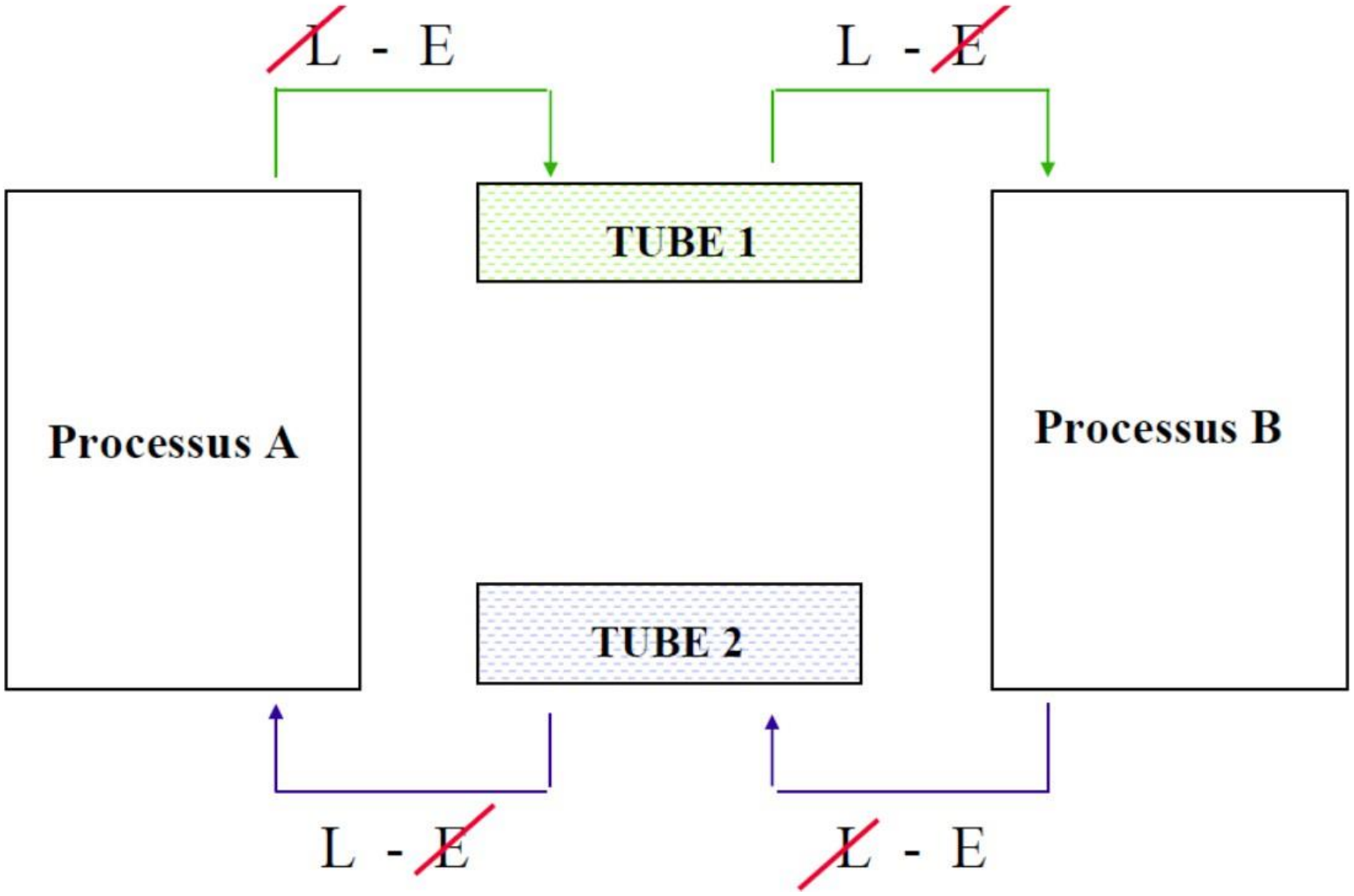
Fonctionnement normal

- L'utilisation unidirectionnelle permet d'être certain du comportement :



Tube avec deux processus : communication unidirectionnelle

- Modèle Question/Réponse : Fermeture des descripteurs inutiles



La table des descripteurs

- On peut lier la sortie d'un tube à **stdin** :
  - Les informations qui sortent du tube arrive comme une entrée standard et on peut utiliser scanf, ...
- ou l'entrée à **stdout** :
  - Les informations qui sortent par stdout sont écrites sur le tube et on peut utiliser printf, ...
- On va utiliser la fonction **dup** ou **dup2** qui permettent de dupliquer des entrées de la table des descripteurs du processus

Tubes

Tubes nommés

Tubes nommés (FIFO)

- Fichier avec un nom (donc accessible par n'importe quel processus connaissant ce nom et disposant des droits d'accès au tube).
- Ils sont affichés lors de l'éxécution d'une commande **ls -l** et sont caractérisés par le type p (fichier physique de type **p** : existence d'un **i-node**).
- Ils permettent de transmettre des données entre des processus qui ne sont pas attachés par des liens de parenté.

Tubes nommés (FIFO)

- La commande shell est **mknod** et, plus généralement, la commande **mkfifo** :

```
> mkfifo nom_fichier
```

- En langage C nous utiliserons l'interface suivante (mknod existe aussi mais n'est pas conseillée pour la portabilité du code) :

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *nomfichier, mode_t mode);
```

Où :

nomfichier : le chemin d'accès au tube nommé.
mode: les droits d'accès des différents utilisateurs à cet objet (S_IRUSR, S_IWUSR, S_IXUSR).

- La valeur renvoyée par mkfifo est **0** s'il **réussit,** ou **-1** s'il **échoue**, auquel cas errno contient le code d'erreur (plus d'info : man 2 mkfifo).

Comportement par défaut

- **Tube vide :** Lecture bloquante (sauf si ouverture en O_NDELAY)
- **Tube non vide :** Attente d'une quantité suffisante de données à lire.

Utilisation

Séquence classique :

- Un processus en lecture open(O_RDONLY)
- Un processus en écriture open(O_WRONLY)
- L'ouverture d'un tube nommé est bloquante par défaut (Sinon, utilisation de O_NONBLOCK).

Primitives

- L'ouverture d'un tube nommé par un processus s'effectue avec la primitive

  desc=open(nom_du_tube, mode).

- Le processus effectuant l'ouverture doit posséder les droits correspondants sur le tube.

- La primitive renvoie un descripteur **desc** correspond au mode d'ouverture spécifié (lecture seule, écriture seule, lecture/écriture).

- Les modes : O_RDONLY, O_WRONLY.

Primitives

- La lecture s'effectue avec la commande **read(desc, buf, nb)**
- L'écriture s'effectue avec la commande **write(desc, buf, nb)**
- La fermeture s'effectue avec la commande **close(desc)**
- On peut aussi utiliser **unlink(nom_du_tube)** ou **rm nom_du_tube**)

Primitives

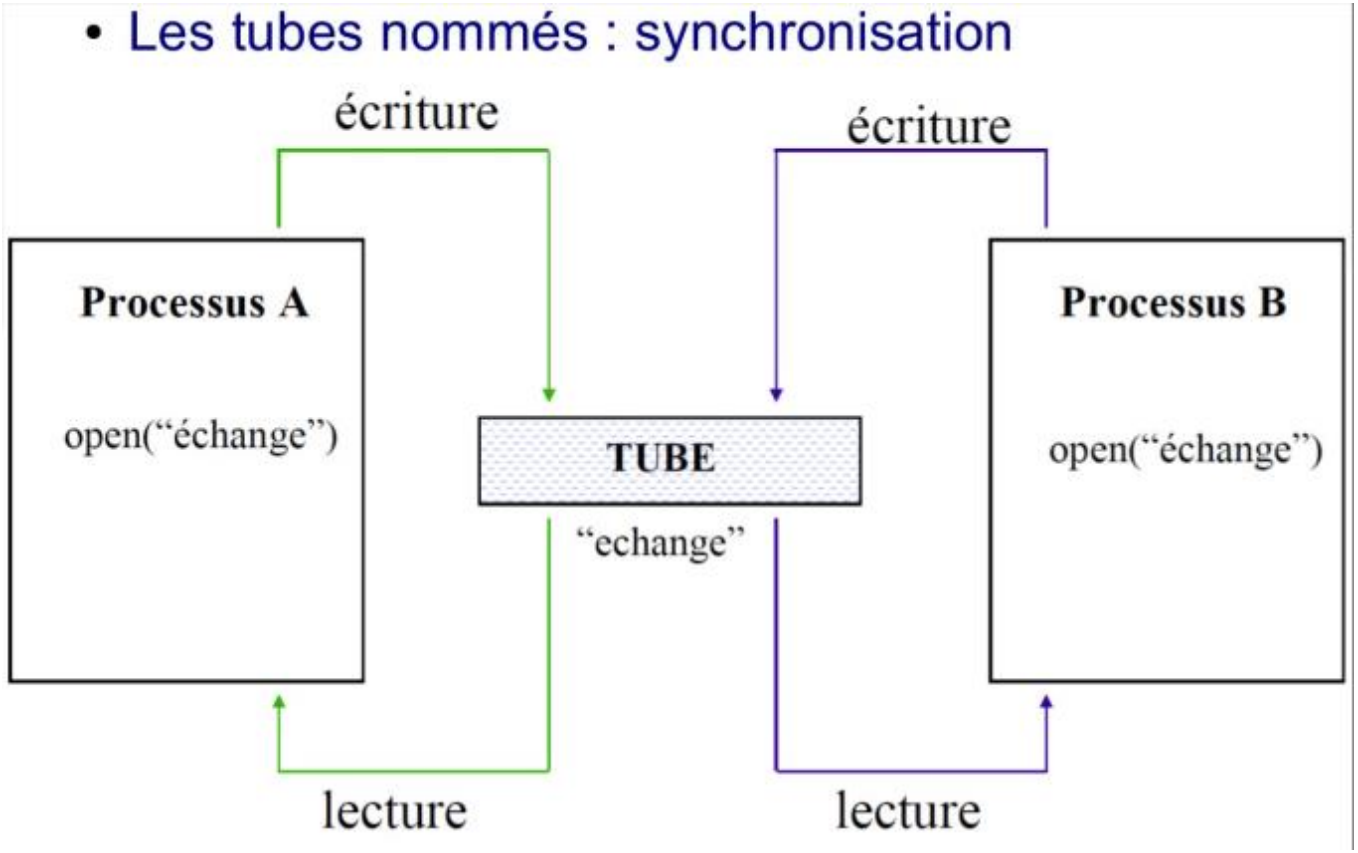
- Par défaut, la primitive **open()** appliquée au tube nommé **est bloquante.** Ainsi, la demande d'ouverture en lecture est bloquante tant qu'il n'existe pas d'écrivain sur le tube.

- D'une 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 **rendez-vous** en un point particulier de leur exécution.

- Il faudra néanmoins, lors de dialogues dans les deux sens entre deux processus, s'assurer que les ordres d'ouverture sont faits dans le bon sens, sinon cela pourra provoquer un **interblocage** !

Synchronisation

Synchronization



Exemple

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main() // programme ecrivain.c
{
        mode_t mode; int tub;
        mode = 0644; // ou S_IWUSR | S_IRUSR | S_IRflRP |
            S_IROTH;
        mkfifo("fictub", mode);
        tub = open("fictub",O_WRONLY);
        write(tub,"0123456789",10);
        close(tub);
        exit(0);
}
```

Exemple

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main() // programme lecteur.c
{
        char buf[11]; int tub;
        tub = open("fictub",O_RDONLY);
        read(tub,buf,10);
        buf[11]='\0';
        printf("J'ai lu %s\n", buf);
        unlink(''fictub'');// ou close(tub);
        exit(0);
}
```

Exemple

Pour exécuter :

Ouvrir un terminal et lancer :
>./ecrivain

Ouvrir un autre terminal et taper :
>ls -l

Ouvrir un autre terminal et lancer :
>./lecteur