Programmation Système et Réseau

Communication Inter-Processus (IPC)

3 - Les Tubes (Pipes)

Mohamed Maachaoui

Juan Ángel Lorenzo del Castillo

Seytkamal Medetov

Son Vu

CY Tech

ING2 - GSI

2020-2021

mohamed.maachaoui@cyu.fr

Table of Contents

Table of Contents

Pipe “commande shell”

- La commande “`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.
- 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.
- **Remarque :** le processus qui écrit ne peut pas lire des informations, et inversement. Il faut donc créer deux tubes si on souhaite que les processus établissent réellement un dialogue.

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 processus ayant accès à ce fichier.
- **En script :** l'utilisation de pipe permet d'enchaîner des commandes (donc processus différents) en leur transmettant des informations.

Table of Contents

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 **destructrice** : une information ne peut être lue qu'une seule fois dans un tube.
- Il existe deux sortes de tubes :
  - Anonyme
  - Nommé (FIFO)

Tubes anonymes (non nommés)

- Fichier logique
- Deux descripteurs (lecture/écriture)
- Aucune référence dans le système de fichier (fichier anonyme)
- Norme SVr4 :
  - Unidirectionnel (SUN OS bidirectionnels)
  - Un descripteur pour la lecture, un descripteur pour l'écriture
- Lien de parenté obligatoire (processus créateur et ses descendants créés après le tube)
- `open()` ne peut pas être utilisé
- 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

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) $\rightarrow$ accès impossible

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.

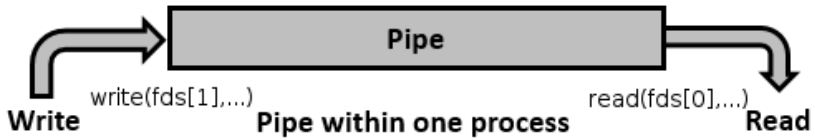
Tubes anonymes : écriture

- 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



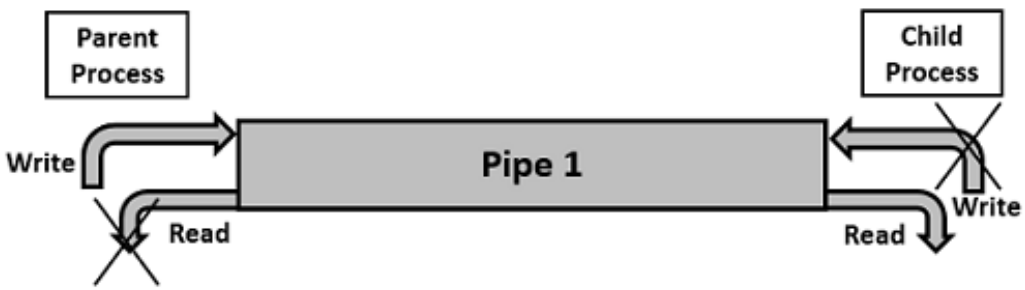

```c
#include<stdio.h>
#include<unistd.h>

int main() {
   int fds[2];
   int returnstatus;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   returnstatus = pipe(fds);

   if (returnstatus == -1) {
      printf("Unable to create pipe\n");
      return 1;
   }

   printf("Writing to pipe - Message 1 is %s\n", writemessages[0]);
   write(fds[1], writemessages[0], sizeof(writemessages[0]));
   read(fds[0], readmessage, sizeof(readmessage));
   printf("Reading from pipe - Message 1 is %s\n", readmessage);
   printf("Writing to pipe - Message 2 is %s\n", writemessages[1]);
   write(fds[1], writemessages[1], sizeof(writemessages[1]));
   read(fds[0], readmessage, sizeof(readmessage));
   printf("Reading from pipe - Message 2 is %s\n", readmessage);
   return 0;
}
```

Source : tutorialspoint.com

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

Exemple :

```
 1 #include <stdio.h>
 2 #include<unistd.h>
 3  int main()
 4 {
 5   int p[2];
 6   char buf[20];
 7
 8   pipe(p);
 9   switch (fork())
10   {
11     case −1: exit(1);
12     case 0:
13       close(p[1]);
14       read(p[0], buf, sizeof(buf));
15       printf("%s bien recu \n",buf);
16       break;
17     default:
18       close(p[0]);
19       write(p[1], "Bonjour", sizeof("Bonjour"));
20   }
21 }
```

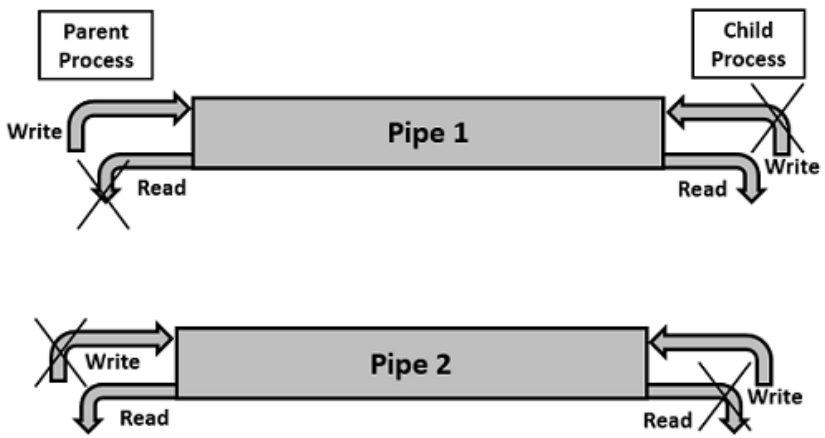
Tube avec deux processus : communication bi-directionnelle

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.

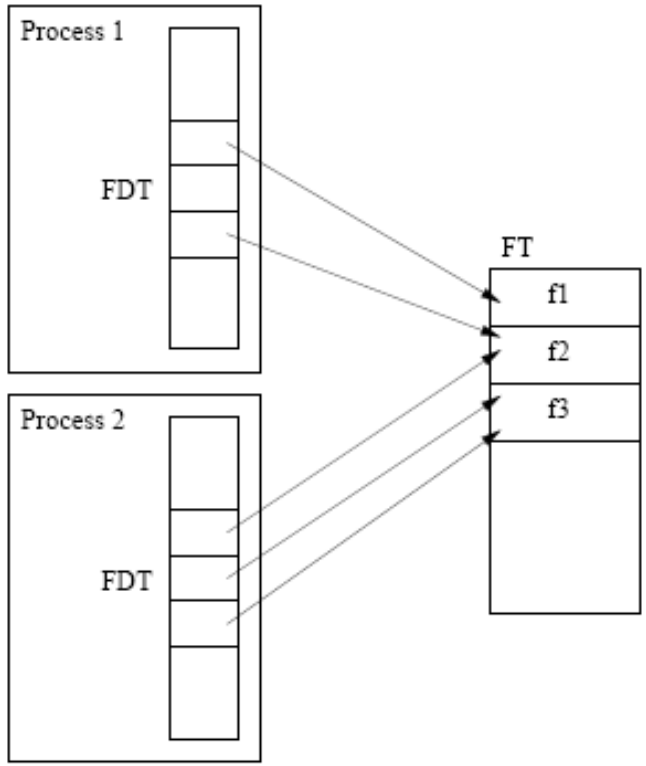
Fonctionnement normal

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



La table des descripteurs

- Deux descripteurs de fichiers de deux processus différents peuvent pointer vers le même fichier (c'est le cas de f2)
- Ceci peut se produire dans le cas de fork. En effet, le fork duplique les processus et donc également la table des descripteurs de fichiers.
- Il se peut également que deux descripteurs de fichiers du même processus pointent vers le même fichier, grâce aux appels système `dup` et `dup2`.



Duplication de descripteurs

- L'opération de duplication d'un descripteur de permet de créer une copie du descripteur qui pointera vers le même fichier ouvert.

```
int dup(int desc);
```

- La fonction `dup()` prend comme argument un descripteur valide. Elle cherche le plus petit descripteur libre e (non utilisé) dans la table de descripteur de fichiers et copie desc dans e. La fonction retourne le nouveau descripteur e ainsi créé.

```
int dup2(int olddesc,int newdesc);
```

- La fonction `dup2()` ferme le descripteur `newdesc` s'il était ouvert et fait une copie de `olddesc` dans `newdesc`.
- `dup` et `dup2` renvoient le nouveau descripteur, ou -1 s'ils échouent, auquel cas `errno` contient le code d'erreur :
  - **EBADF** : `olddesc` n'est pas un descripteur valide, ou `newdesc` n'est pas dans les valeurs autorisées pour un descripteur.
  - **EMFILE** : Le processus dispose déjà du nombre maximum de descripteurs de fichiers autorisés simultanément, et tente d'en ouvrir un nouveau.

Duplication de descripteurs

Redirection de la sortie d'un processus vers un fichier

- La commande `ps -a > nom_fichier` redirige la sortie standard de la commande `ps` vers le fichier de nom `nom_fichier`
- Si un processus veut rediriger sa sortie standard (descripteur "1") vers un fichier dont il a accès par descripteur, il suffit que :
  - il ouvre le fichier qui doit servir de nouvelle sortie, par exemple avec `open` (cela lui donne un nouveau descripteur de fichier. En pratique, il vaut 3)
  - il ferme ensuite le descripteur 1 (la sortie),
  - il duplique avec `dup` le descripteur obtenu par `open` et le système lui attribue le plus petit numéro disponible (ici "1")
  - il ferme ensuite le descripteur obtenu avec `open`
- Le processus a donc de nouveau trois descripteurs de fichiers 0, 1 et 2, mais le descripteur 1 désigne maintenant le fichier ouvert avec open
- Enfin les écritures standard (`fwrite`, `printf`, ...) iront écrire directement dans le fichier.

Table of Contents

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.
- 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).

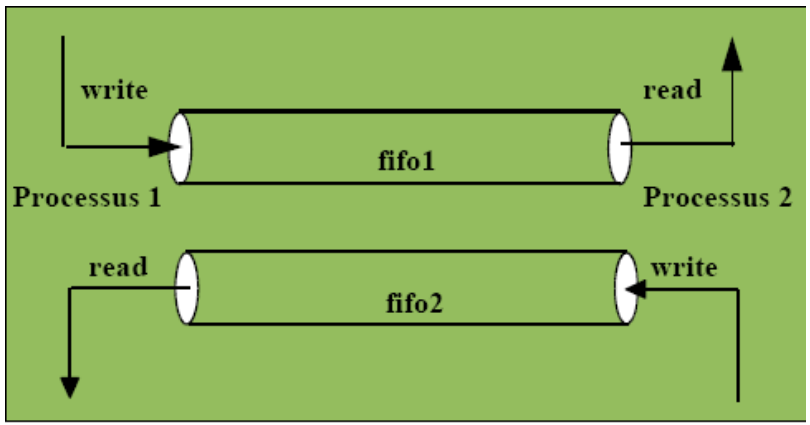
Primitives

- L'ouverture d'un tube nommé par un processus s'effectue avec la primitive `open()`.
- Le processus effectuant l'ouverture doit posséder les droits correspondants sur le tube. La primitive renvoie un descripteur correspond au mode d'ouverture spécifié (lecture seule, écriture seule, lecture/écriture).
- 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** !

Exemple

Séquence d'ouverture correcte

```
/* pour le processus 1 */
int d_lecture, d_ecriture;
...
d_ecriture=open("fifo1", O_WRONLY);
d_lecture=open("fifo2", O_RDONLY);

/* pour le processus 2 */
int d_lecture, d_ecriture;
...
d_ecriture=open("fifo2", O_WRONLY);
d_lecture=open("fifo1", O_RDONLY);
```



Fermeture et destruction d'un tube nommé

- La lecture et l'écriture sur le tube nommé s'éffectuent en utilisant les primitives **read()** et **write()**.
- La fermeture d'un tube nommé s'effectue en utilisant la primitive **close()**.
- La destruction d'un tube nommé s'effectue en utilisant la primitive **unlink()**.