TP C avancé n^o 6
G. Subrenat, Y. Bertrand, N. Courilleau, D. Meneveaux

Licence Informatique L3 - 2021-2022
UFR SFA - Université de Poitiers

Tubes, fork, exec

Table des matières

1 Résumé et cadre de travail

Dans ce TP il s'agit de manipuler la communication entre deux processus lourds via des tubes nommés ou anonymes.
Comme effet collatéral, les fonctions *fork* et *exec* sont également étudiées.

Directive importante : pour tout appel système, (ouverture de fichier, lancement d'un thread, ...) vous devez tester la valeur de retour pour détecter toute anomalie ; on ne présuppose jamais qu'un appel système fonctionne. Un *assert* suffira amplement.

Autre directive : nous n'utiliserons que les entrées-sorties de bas niveau (open, write, ...).

2 Principe de fonctionnement des tubes

Point de cours

Deux processus lourds ne partagent pas, par définition, le même espace mémoire (à l'inverse des processus légers nommés également *threads*).

Une solution est d'utiliser un tube. Un tube est unidirectionnel ; généralement un processus le manipule en écriture pour injecter des données à l'intérieur, et un autre le manipule en lecture et récupère les données injectées par le premier processus.

On parle d'écrivain et de lecteur, ou encore de producteur et de consommateur.

Dans le fonctionnement par défaut la lecture est bloquante : un processus effectuant un ordre de lecture sur un tube vide se bloque jusqu'à ce qu'une donnée soit injectée.

L'écriture dans un tube est légèrement différente. En effet un tube a une capacité limitée (64 Ko sur mon système) et un processus écrivain peut se retrouver devant deux situations :

- La capacité du tube n'est pas atteinte et l'écrivain dépose ses données et continue son code, même si aucun lecteur n'est présent (ou plus exactement n'est en train de lire).
- La capacité du tube est atteinte et l'écrivain se bloque jusqu'à ce que suffisamment de place se libère (due à un lecteur qui a consommé).

Ceci dit, il ne faut jamais que le bon fonctionnement d'un programme dépende de la capacité d'un tube. Une autre façon de présenter les choses est de s'assurer qu'un programme fonctionne dans le cas où la capacité est réduite à zéro ; autrement dit de présupposer qu'un processus ne peut écrire que lorsqu'un lecteur fait une demande de lecture.

Les tubes nommés apparaissent dans le système de fichiers et se manipulent, par les processus, comme des fichiers "normaux". Mais ce n'est qu'apparence : si le processus l'ouvre en écriture c'est un producteur, sinon c'est un consommateur. Et il n'est donc pas logique de l'ouvrir à la fois en lecture et en écriture [a].

Les tubes anonymes sont créés par les processus et ont comme durées de vie celles des processus qui les manipulent. Ils n'apparaissent pas dans le système de fichiers et ne peuvent être manipulés que par des processus issus du même *fork* (cf. exercices dédiés).

a. Ceci dit, c'est techniquement possible sur mon système, ce qui n'est pas une raison pour le faire.

3 Bash et tubes nommés

a) La commande

```
$ mkfifo montube
```

crée un tube nommé *montube*. En listant les fichiers vérifiez les droits par défaut et que le type du fichier est particulier.

b) Lancez 3 terminaux positionné dans le répertoire contenant le tube. Deux lisent dans le tube avec la commande

```
$ cat < montube
```

(ils sont donc en concurrence) et un écrit dans le tube avec la commande

```
$ cat > montube
```

Écrivez plusieurs lignes et regardez comment les lectures se répartissent. Quel que soit le résultat, vous ne pouvez faire aucune hypothèse sur cette répartition.

c) Arrêtez l'écrivain (avec *Ctrl-D* ou *Ctrl-C*) et notez que les lecteurs se terminent également. Vérifiez que le tube existe toujours dans le système de fichiers.

d) Dans un terminal, envoyez la chaîne "bonjour" dans le tube avec la commande

```
$ echo bonjour > montube
```

On remarque que la commande est bloquée ce qui semble entrer en contradiction avec le fait qu'un tube a un tampon : nous verrons par la suite (en programmant) que c'est l'ouverture du tube qui bloque et non l'écriture.
Lancez un lecteur avec

```
$ cat < montube
```

pour vérifier que tout se débloque.

d) Effacez le tube.

4 Programmation et tubes nommés

Rappelez-vous qu'il faut tester (avec un *assert* par exemple) systématiquement le retour de chaque appel système.

Dans cet exercice le tube pré-existe dans le système de fichiers, i.e. il est créé en ligne de commande avant le lancement des programmes.

a) Faites un programme qui :

- ouvre en écriture le tube nommé dont le nom est passé en paramètre.
- affiche le message "le tube vient d'être ouvert en écriture".
- en boucle lit un caractère au clavier et l'écrit dans le tube. On pourra stopper la boucle sur un caractère particulier (de votre choix).
- ferme le tube

Lancez le programme et vérifiez bien que l'ouverture est bloquante.
Lancez dans un autre terminal un lecteur en mode bash

```
$ cat < montube
```

Pour tester votre programme.

b) De même faites un programmeur qui lit dans le tube caractère par caractère et les affiche à l'écran ; testez-le avec un écrivain en mode bash

```
$ cat > montube
```

Comment réagit le lecteur si on arrête l'écrivain (autrement comment réagit la fonction "*read*") ? Détectez et gérez ce cas de figure dans votre programme.

c) Testez vos deux programmes ensemble.

d) Lancez vos deux programme et commencez à les utiliser. Puis dans une troisième console, supprimez physiquement le tube. Et continuez à utiliser les deux programmes. Que se passe-t-il ?

5 Programmation et tubes anonymes

Point de cours

Un tube anonyme a le même fonctionnement qu'un tube nommé, mais :
- il n'apparaît pas sur le système de fichiers,
- il a une durée de vie liée à celle des programmes le manipulant.

En règle général, le déroulement est le suivant :
- le programme crée un tube anonyme (fonction système *pipe*)
- il se duplique (fonction système *fork*) et le tube est alors présent dans chaque processus
- un processus ferme l'extrémité du tube en lecture, et l'autre ferme celle en écriture
- alors l'un écrit et l'autre lit
- à la fin, chaque processus ferme l'extrémité qu'il avait conservée.

Dans cet exercice vous pouvez être en mode "recherche autonome" pour construire votre programme :
- rappelez-vous comment marchent les fonctions *fork* et *wait*,
- et étudiez la fonction *pipe*.

Vous pouvez aussi vous référer au point du cours en fin de document.

Écrivez le programme suivant :
- le processus crée un tube anonyme
- il crée deux fils
- le père ferme toutes les extrémités du tube et attend la fin des deux fils
- un fils envoie la chaine "bonjour monde !" (mais le code doit marcher pour n'importe quelle chaîne)
- l'autre fils la récupère et l'affiche. Attention ce processus ne connaît pas a priori la longueur de la chaîne qu'il reçoit.

Au fait comment le récepteur connaît la longueur de la chaîne qu'il reçoit (on ne veut pas lire la chaîne caractère par caractère) ?

6 Programmation et tubes anonymes : bidirectionnel

Si l'on veut avoir une communication bidirectionnelle, il est nécessaire d'avoir deux tubes.

Écrivez un programme qui :
- crée les deux tubes anonymes
- se duplique.
- le père lit deux entier au clavier et les envoie au fils
- le fils récupère les deux entiers et envoie la somme au père qui l'affiche
- le programme doit se terminer proprement, notamment en ce qui concerne la fermeture des descripteurs de fichiers (descripteurs de tubes plutôt).

Note : on pourrait faire la même chose avec deux tubes nommés.

7 Exec

Point de cours

La famille des fonctions *exec* permet de substituer le code du processus courant par le code d'un autre exécutable. L'exécutable est désigné par son nom de fichier.

Il est possible de passer des paramètres au nouveau programme comme si on les avait tapés en

ligne de commande.

Les fonctions de type *exec* sont souvent associées à un *fork*.

Note : nous utiliserons uniquement la fonction *execv* dans la suite du TP.

Voici succinctement (cf. manuel) la description de la fonction *execv* :

- Le code du processus courant est supprimé et remplacé par le code du processus cible (premier argument de la fonction) qui s'exécute à partir de sa première instruction (première instruction de son *main* s'il s'agit d'un programme C [a]).
- protoype : *int execv(const char *pathname, char *const argv[]) ;*
- *pathname* : chemin absolu ou relatif vers l'exécutable à lancer.
- *argv[]* : tableau de chaînes de caractères qui sont, pour le nouveau processus, les arguments qui proviendraient de la ligne de commande s'il avait été lancé dans une console.
- Attention pour *argv* (en supposant qu'on envoie n paramètres) :
  - *argv[0]* contient par convention la même chose que *pathname*
  - *argv[1]* à *argv[n]* sont les n arguments. Ce sont des chaînes de caractères obligatoirement ; donc si on veut faire passer (par exemple) un entier, il faut au préalable le transformer en chaîne (sachant que dans le programme cible il faudra certainement le retransformer en entier).
  - *argv[n+1]* doit contenir *NULL* (pour marquer la fin des arguments)
  - Bref, si on veut passer n arguments, il faut un tableau de $n + 2$ cases.
- Par définition la fonction ne retourne pas sauf s'il y a une erreur (par exemple l'exécutable cible n'existe pas) ; auquel cas la fonction retourne -1.

a. On peut donc tout à fait faire un *exec* sur un script shell, python, ... ou un exécutable issu d'un autre langage.

Écrivez un programme (nommé *exec_ls.c*) qui se substitue à la commande */bin/ls* en lui passant les deux arguments "-l" et "-h".

Écrivez un programme (nommé *liste_arguments.c*) qui affiche la liste de ses arguments.
Puis recopiez le premier programme de l'exercice dans le fichier *exec_liste.c*. Ce dernier, au lieu de se substituer avec la commande */bin/ls* se substitue avec le nouveau programme *liste_arguments*.

Écrivez un programme (nommé *somme.c*) qui prend, en ligne de commande, deux arguments qui sont des entiers et en affiche la somme.
Écrivez un programme (nommé *exec_somme.c*) qui :

- lit deux entiers au clavier (avec *scanf*) ; il s'agit bien de stocker ces deux entiers dans des variables de type *int*.
- se substitue au programme précédent en lui passant les deux entiers en paramètres.

Question philosophique : que se passe-t-il si un programme fait un *exec* sur lui-même ?

8 Fork, exec et tubes anonymes

Point de cours

À la suite d'un *exec*, tous les file descriptors préalablement ouverts (notamment les *pipes*) restent ouverts dans le nouveau code.

Le nouveau code n'en a priori pas conscience : mais le programmeur lui le sait et donc ce n'est pas un problème. Le vrai problème est qu'il ignore leurs valeurs (on rappelle qu'un file descriptor de bas niveau est de type *int*).

Il faut donc, d'une manière ou d'une autre, indiquer au nouveau code les valeurs de ces file descriptors ; par exemple en arguments de la ligne de commandes.

Écrivez un premier programme (*ecrivain.c*) qui :

- prend un paramètre qui est la valeur du file descriptor dans lequel il va écrire
- lit deux entiers en clavier
- les écrit dans le file descriptor en mode binaire
- ferme le file descriptor,
- se termine

Écrivez un deuxième programme (*lecteur.c*) qui :

- prend un paramètre qui est la valeur du file descriptor dans lequel il va lire
- lit deux entiers dans le file descriptor en mode binaire
- ferme le file descriptor,
- affiche la somme des deux entiers
- se termine

Écrivez un troisième programme (*forkexec.c*) qui :

- crée un tube anonyme
- se duplique avec un *fork*
- le fils ferme l'extrémité en écriture et remplace son code par le deuxième programme en lui passant la valeur du file descriptor encore ouvert (rappel : un *exec* ne peut prendre que des chaînes de caractères en paramètres)
- le père ferme l'extrémité en lecture et remplace son code par le premier programme en lui passant la valeur du file descriptor encore ouvert (cf. remarque ci-dessus)

Vérifier que tout se passe bien.

Question : dans le troisième programme, pourquoi le père, après le *execv*, ne peut-il pas faire un *wait* ? D'ailleurs la solution la plus élégante serait que le troisième programme crée deux fils qui communiquent et le père se contente d'attendre leurs fins.

9 Fork, exec et tubes nommés

Le but est de reprendre tout ce qui a été vu jusqu'à présent. On demande à ce que tous les appels systèmes soit testés (un *assert* est pleinement suffisant). C'est un exercice d'entrainement un peu long.

Il y a plusieurs paires worker/master, chaque programme étant indépendant c'est à dire qu'il possède son propre code source et son propre exécutable.

Un master reçoit deux arguments en lignes de commandes :

— le nom du tube lui permettant d'envoyer des données à son worker,

— le nom du tube lui permettant de recevoir le ou les résultats du worker.

Un worker a exactement les mêmes paramètres.

Il y a un protocole de communication qui est propre à chaque paire en fonction des calculs demandés.

Il y a un programme principal (l'orchestre) dont le but est de lancer les workers et les masters.

9.1 Orchestre

Le fonctionnement de l'orchestre est le suivant :

— il demande à l'utilisateur un numéro compris entre 1 et le nombre de paires master/worker,

— crée les deux tubes nommés,

— lance les master et worker désignés qui communiqueront avec les deux tubes

— attend la fin des deux programmes

— efface les tubes

— recommence

Pour simplifier, on suppose que les noms des exécutables des masters et des workers sont codifiés :

— masters : "*master_<n>*" où n varie de 1 au nombre de masters,

— worekrs : "*worker_<n>*" où n varie de 1 au nombre de workers,

Au lancement de l'orchestre, en ligne de commandes, on lui indique le nombre de paires master/worker.

Par convention, le numéro 0 signifie que l'on arrête l'orchestre.

On demande à ce que les tubes aient des noms uniques et clairs. On entend par "unique" que le même nom ne doit pas être utilisé deux fois. Supposons par exemple qu'on ait 3 paires de worker/master ; voici, au cours de la vie de l'orchestre, des exemples de noms de tubes qu'il crée :

— *pipeMasterToWorker_1_4* et *pipeWorkerToMaster_1_4*
c'est la 4me fois que l'orchestre lance une paire worker/master et les tubes relient *worker_1* et *master_1*.

— *pipeMasterToWorker_3_15* et *pipeWorkerToMaster_3_15*
c'est la 15me fois que l'orchestre lance une paire worker/master et les tubes relient *worker_3* et *master_3*.

— *pipeMasterToWorker_1_23* et *pipeWorkerToMaster_1_23*
c'est la 23me fois que l'orchestre lance une paire worker/master et les tubes relient *worker_1* et *master_1*.

— ...

Note : comme deux paires master/worker ne tournent pas en même temps, on aurait pu réutiliser les mêmes noms ; la contrainte est purement pédagogique.

9.2 worker 1/master 1

Le master demande deux entiers à l'utilisateur, les envoie au worker qui retourne la somme.

9.3 worker 2/master 2

Le master demande une chaîne de caractères à l'utilisateur, l'envoie au worker qui retourne la même chaîne privée des voyelles.

Les chaînes transmises doivent être réduites au minimum et on ne fait aucun a priori sur la longueur des-dites chaînes.

9.4 worker 3/master 3

À votre initiative !

10 Tubes anonymes, point de cours

Présentation succincte : un tube anonyme est créé par un processus qui ensuite se duplique (*fork*) ; le tube est ensuite utilisé par le père et tous les fils.

Le code dans la suite de la section est réduit à l'essentiel :

- il n'est pas commenté
- on ne vérifie pas les erreurs (ce qui est indispensable de faire dans un vrai code).

Le code commenté est fourni.

Le code, fil conducteur de la section, illustre le père qui envoie un *float* à son fils via un tube anonyme.

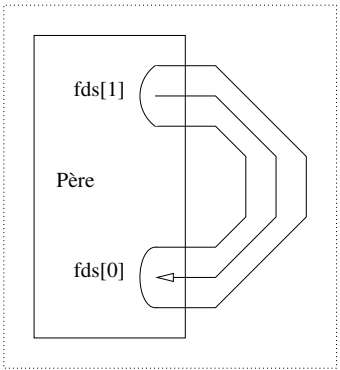
10.1 Création du tube anonyme

C'est la fonction *pipe* qui se charge de cette opération : non seulement le tube est créé, mais il est prêt à l'emploi (sans qu'il soit nécessaire de faire des *open*).

code.c

```
8  int main()
9  {
10     int fds[2];
11
12     pipe(fds);
```

Graphiquement on peut considérer que le tube est extérieur au processus, mais que les extrémités sont internes.



À ce stade le processus pourrait écrire dans le tube puis lire ce qu'il a écrit [1] ; mais l'intérêt est très limité.

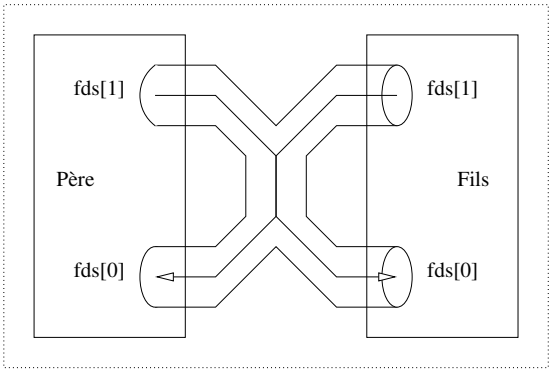
10.2 Duplication du processus

Après un *fork* du processus, les choses se compliquent mais deviennent plus intéressantes.

code.c

```
14     if (fork() == 0)
```

Le tube est toujours unique [2] mais chaque processus a accès aux deux extrémités.



En règle général il n'est pas recommandé que plusieurs processus aient accès simultanément à une même extrémité d'un tube. Dans notre cas ce n'est pas le but recherché :

- le père n'utilise que l'extrémité en écriture (*fds[1]*)
- le fils utilise l'autre (*fds[0]*)

10.3 Fermetures des extrémités inutiles

La première chose que doit faire un processus après un *fork* est de fermer immédiatement l'extrémité dont il ne se sert pas :

1. à condition de ne pas écrire plus que la capacité du tube
2. Ce qui ne serait pas le cas si on l'avait créé après le *fork* car chaque processus aurait eu son propre tube.

- le père doit fermer *fds[0]*
- le fils doit fermer *fds[1]*

Voici le code du fils :

code.c (partie fils)

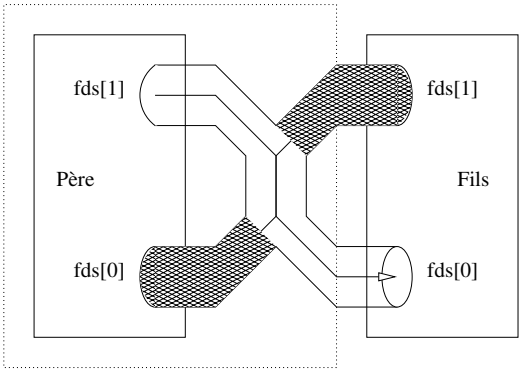
```
15  {
16      // fils : reçoit les données
17
18      close(fds[1]);
```

Et le code symétrique du père :

code.c (partie père)

```
27  {
28      // père : envoie les données
29
30      close(fds[0]);
```

Ce qui graphiquement conduit à :



Et il reste un tube "normal" du père vers le fils.

10.4 Échange d'informations

L'envoi et la réception de l'information se font classiquement avec des *read* et *write*

Pour rappel voici le code du père :

code.c (partie père)

```
34      float f = 3.14159265358979323846264338327950288419 71;
35      printf("[père] j'envoie le float %g\n", f);
36      write(fds[1], &f, sizeof(float));
```

Et celui du fils :

code.c (partie fils)

```
20      float f;
21      read(fds[0], &f, sizeof(float));
22      printf("  [fils] j'ai reçu %g\n", f);
```

10.5 Libération des ressources

En fin de code, le fils doit fermer l'extrémité qu'il utilisait :

code.c (partie fils)

```
24      close(fds[0]);
```

Et le père ferme également l'extrémité restante, mais il doit aussi attendre la fin de son fils :

code.c (partie père)

```
36      close(fds[1]);
37
38      wait(NULL);
```