R3.05 - TP 4 - Tubes

Rappel de BUT1

Nous avons découvert les tubes en 1ère année, comme un système permettant de mettre en relation le **STDOUT** d'une commande avec le **STDIN** d'une autre commande, sous la forme suivante, dans le Shell :

```
commande_A | commande_B
```

Ici le **STDOUT** (les **printf()**) de la **commande_A** alimente le **STDIN** (les **scanf()**) de la **commande_B** produisant, elle aussi, des choses sur son **STDOUT** qui, n'étant pas redirigé, aboutit à des affichages à l'écran.

Ceci est censé être un rappel de BUT1, mais si ça ne vous parle pas trop, demandez l'aide de l'enseignant.e car ce qui va suivre nécessite une parfaite compréhension de ce mécanisme de base du Shell.

ATTENTION : nous venons de dire que **printf()** écrit sur **STDOUT** et que **scanf()** lit sur **STDIN**. C'est tout à fait exact mais ces deux fonctions ne sont pas des fonctions système. En fait, on l'a déjà évoqué, ce sont des fonctions évoluées, fournies par la **libc**. Par exemple, **printf()** permet de formater l'affichage de façon assez souple et élaborée. Le code de ces fonctions fait appel aux fonctions système de base que sont **read()** et **write()**. C'est un enrobage des fonctions système de base.

Les tubes sont un mécanisme système. Pour écrire et lire dans un tube, vous devrez obligatoirement faire appel aux fonctions système de base que sont **read()** et **write()**, et aucunement à des **printf()** ou des **scanf()**, sinon ça ne fonctionnera pas !

Programmation des tubes

Les tubes reposent en fait sur deux mécanismes similaires mais indépendants et dont l'usage dépend du contexte.

Ces deux mécanismes s'appellent :

- Les tubes nommés
- Les tubes anonymes

Les tubes du Shell reposent sur le mécanisme des tubes anonymes.

Dans un premier temps, nous allons travailler avec les tubes nommés qui sont plus simples à mettre en œuvre. Nous verrons les tubes anonymes dans un second temps.

A) Tubes nommés

Pourquoi “nommés” ?

Ces tubes sont intitulés “nommés” car ils s'appuient sur des fichiers spéciaux, ce qui donne naturellement un nom (celui du fichier en question) au tube qui lui est rattaché.

Depuis le BUT1, on connaît les fichiers dits “standards”, qu’on identifie par le **-** (tiret) qui précède les droits affichés avec un **`ls -l`** :

```
-rw-r--r--   1 begood  johnny    210 23 sep 15:00 test.c
```

On vous avait promis de voir de nouveaux types de fichiers, des fichiers dits “spéciaux”, qui possèdent une lettre (autre que le **d** qui identifierait un dossier) en 1ère position, devant les droits sur le fichier. Nous y voici avec les tubes !

Première expérimentation

En ligne de commande dans un Terminal, nous allons créer un tube nommé dans votre répertoire de travail :

```
mkfifo mon_tube
```

Faites maintenant un **`ls -l`** et observez la lettre qui précède les droits.

Un tube s’appelle un **`fifo`** car, comme avec une file, ce qui est lu dans un tube arrive dans le même sens que ce qui y a été écrit (**f**irst **i**n **f**irst **o**ut).

La lettre **p** qui figure devant les droits signifie, vous l’aviez deviné, **`pipe`**.

Vous avez ainsi créé un tube nommé, et son nom est donc **`mon_tube`**.

Comme tout fichier, ce tube nommé existe tant que le fichier support de ce tube existe.

Pour le supprimer, un simple **`rm`** fera son office…

Notez aussi que les droits **`UGO`** sur ce tube nommé fonctionnent comme sur n’importe quel fichier standard. Vous avez le droit d’y écrire si vous avez un droit **`w`** et vous avez le droit d’y lire si vous avez un droit **`r`**. Le droit **`x`** est sans aucun effet.

Par contre, il n’est pas possible de se déplacer dans le tube avec un **`lseek`** par exemple (**`man 2 lseek`**) qui est réservé à des fichiers standards uniquement. Ceci s’explique aisément par le fait que les données ne sont présentes dans un tube que jusqu’à leur lecture et qu’une lecture lit dans le même sens que les écritures ont eu lieu.

Essayez d'écrire quelque chose dans ce tube. Attention, n'utilisez pas un éditeur de texte mais utilisez un **`echo quelque_chose > mon_tube`** ou un **`cat un_fichier > mon_tube`** par exemple.

Que se passe-t-il ?

Explication : par défaut, les tubes sont bloquants. Cela signifie qu'un écrivain (le **`echo`** ou le **`cat`** dans l'essai précédent) reste bloqué en écriture tant qu'aucun lecteur n'est "en bout de tube".

Interrompez votre commande avec **`CTRL+C`**. A ce stade, rien n'a été écrit dans le tube par manque de lecteur, comme expliqué précédemment.

Maintenant, tentez de lire le contenu du tube par : **`cat mon_tube`** ou **`cat < mon_tube`**

Que se passe-t-il cette fois-ci ? Si vous avez un doute, faites-vous confirmer votre raisonnement avant de poursuivre.

Comment sortir de cette impasse ? Proposez une solution simple.

Lecteur en C

Vous allez maintenant coder un programme de lecture dans un tube nommé. On conserve le même fichier **`pipe`** que précédemment, puisqu'il existe déjà.

Créez un programme C, nommé **`upper`**, qui va :

- Ouvrir, en lecture seule, le tube nommé. Attention, on rappelle que les tubes sont des mécanismes système. Il faut donc les manipuler (ouverture, lecture, écriture) avec des fonctions système adaptées.
- Lire tout ce qui vient du tube[1], convertir et afficher chaque lettre minuscule en MAJUSCULE. Voici un bout de code utile pour transformer un caractère de min en MAJ :

```
if ((c >= 'a') && (c <= 'z')) {
     c = c - 'a' + 'A';
}
```

Note : si lire dans un tube est bloquant tant il n'y a pas d'écrivain ou tant qu'il n'y a rien de nouveau provenant de l'écrivain, la lecture se débloquera automatiquement quand l'écrivain fermera son accès au tube, indiquant par cette action qu'il n'y a plus rien qui proviendra de lui. Ca s'apparente alors à une fin de fichier si on avait affaire à un fichier standard. Ce qui démontre, s'il fallait encore le faire, l'intérêt de toujours refermer ce qui a été ouvert, dès lors qu'on en a plus besoin !

[1] Conseil : lisez plutôt caractère par caractère.

Maintenant que vous avez compris, tenez-en compte dans votre boucle de lecture du tube.

Pour tester votre programme, lancez-le en 1er et ensuite, dans un second Terminal, lancez ceci :

```
echo "Ceci est un TEST" > mon_tube
```

Observez ce qu'affiche votre programme **upper**. Est-ce que ça fonctionne ? Ça devrait afficher la même chaîne en majuscules ! Sinon, faites appel à l'enseignant.e pour vous mettre sur la voie. Votre programme doit (en principe) s'arrêter seul à la fin de la lecture du tube.

Ecrivain en C

Créez maintenant un programme en C, nommé **ecrivain**, qui envoie, sur le tube nommé précédent, chaque argument passé en ligne de commande à **ecrivain**.

N'oubliez pas de lancer **upper** dans un Terminal, avant de tester **ecrivain** dans un autre. Observez ce qu'affiche **upper**. Obtenez-vous le résultat attendu ?

Tentez maintenant de lancer **ecrivain** en 1er. Est-il bloqué ? Se débloque-t-il quand le lecteur (**upper**) est lancé dans un second Terminal ?

Note : le blocage est effectif en lecture comme en écriture.

A partir du moment où un lecteur ouvre le tube, ça débloque l'écrivain.

Le lecteur fainéant et l'écrivain abusé

Contrairement à ce qui se passe avec le lecteur qui obtient une fin de fichier quand l'écrivain ferme le tube, quand c'est le lecteur qui ferme le tube en premier, l'écrivain reçoit deux informations qui permettent de détecter la fermeture de l'autre côté du tube :

- Un code d'erreur en retour de la fonction **write()** (devinez lequel) ainsi qu'une valeur **errno = EPIPE**.
- Un signal **SIGPIPE**, qui est déclenché à chaque tentative d'écriture dans le tube, à partir du moment où il a été refermé par le lecteur. Il faut évidemment qu'il ait été préalablement ouvert par le lecteur, puisque l'écriture est bloquante tant qu'aucun lecteur n'est au bout du tube !

Vous avez donc suffisamment d'alertes pour repérer un tel problème.

Notez au passage que que lire dans un tube fermé par l'écrivain est une situation normale, alors que l'inverse est une situation d'erreur, ce qui est logique puisque c'est

généralement l'écrivain qui est le mieux placé pour savoir quand l'histoire peut se terminer !

Sur la base de votre code **`upper.c`**, créez un **`upper2.c`** qui ouvre et referme immédiatement le tube, ce qui permettra à l'écrivain d'être débloqué, mais ne lui laissera pas suffisamment de temps pour écrire quelque chose dans le tube avant de rencontrer une erreur.

Sur la base de votre code **`ecrivain.c`**, créez un **`ecrivain2.c`** qui :

- Affiche le code de retour de ses **`write()`** et la valeur de **`errno`**,
- Appelle **`perror()`** après chaque **`write()`**,
- Met en place un détournement du signal **`SIGPIPE`** pour afficher un message.

Astuce : par sécurité, mettez des **`sleep(1)`** dans chaque passage de boucle, avant de faire le **`write()`** pour être sûr que le lecteur a eu le temps de fermer le tube car, même si c'est peu probable, il est quand même possible que l'écrivain ait le temps suffisant pour écrire tout ce qu'il doit entre l'ouverture et la fermeture du tube par le lecteur (**`upper2`**).

Testez.

Avant de passer aux tubes anonymes, il convient de préciser une chose importante et en principe évidente si vous avez bien suivi.

Même si, techniquement, on pourrait avoir plusieurs écrivains et plusieurs lecteurs sur un même tube, il est fortement déconseillé de tenter de le faire. Le résultat serait très imprévisible.

Un tube = **un et un seul lecteur** ainsi qu'**un et un seul écrivain**

B) Tubes anonymes

Les principes des tubes anonymes sont similaires à ceux des tubes nommés, sur les points suivants :

- Un lecteur est bloqué dans l'attente de données à lire. C'est le comportement par défaut, mais il peut être modifié pour ne plus être bloquant, si besoin.
- Un seul lecteur et un seul écrivain sont mis en relation par le tube.
- Une lecture dans un tube fermé par l'écrivain est une situation normale qui correspond à une fin de fichier.
- Une écriture dans un tube fermé par le lecteur est une situation d'erreur et doit être traitée en conséquence.

Les tubes nommés permettent de mettre en relation des processus qui n'entretiennent aucune relation particulière entre eux. Chacun ouvre le tube, l'un en lecture et l'autre en écriture. D'ailleurs, compte tenu du caractère bloquant des lectures et des écritures, un même processus ne peut donc pas avoir les deux rôles (écrivain et lecteur).

Et c'est sur ce point que les tubes anonymes se distinguent car il mettent en relation des processus particuliers, qu'on a vus dans un TP précédent :

> Un tube anonyme = un processus **père** et un processus **fils**.

Les tubes anonymes servent à un père et son fils pour échanger des données de façon obligatoirement unidirectionnelle, pour les mêmes raisons de blocage évoquées précédemment.

Ainsi, le père peut être, au choix, l'écrivain ou le lecteur, son fils prenant alors l'autre rôle.

Son intitulé de "tube anonyme" vient du fait que ce n'est plus un fichier qui sert de support au tube mais un appel à une fonction système, comme on va le voir tout de suite.

La fonction pipe()

Avec la fonction système **`fork()`**, que vous connaissez déjà, la fonction système **`pipe()`** est au cœur de la mise en œuvre d'un tube anonyme.

Le prototype de la fonction **`pipe()`** est :

```
int pipe(int fd[2]);
```

et nécessite quelques explications :

- **`pipe()`** crée le tube anonyme qui n'a d'existence que dans la mémoire vive de l'ordinateur. Alors qu'un tube nommé est un fichier spécial accessible par le FS sur disque, le tube anonyme est une sorte de fichier spécial résidant en mémoire.

  Une fois créé, **`pipe()`** ouvre (et garde ouvert) ce tube 2 fois : une fois en écriture et une seconde fois en lecture.

- **`fd[2]`** est un tableau de 2 entiers (si, si !) qui revient de l'appel à **`pipe()`** rempli par les 2 descripteurs résultant de l'ouverture du tube en lecture et en écriture. **`fd[0]`** est le descripteur correspondant à l'ouverture du tube en lecture et **`fd[1]`** est le descripteur correspondant à l'ouverture du tube en écriture.

  Ainsi, au retour de l'appel à **`pipe()`**, le processus peut écrire dans le tube en utilisant **`fd[1]`** et y lire en utilisant **`fd[0]`**. Évidemment, comme on l'a déjà évoqué, un seul et unique processus ne peut pas lire et écrire, mais quel intérêt aurait-on à utiliser un tube avec un seul processus ?

En introduction de ce chapitre, nous avons vu que **`pipe()`** est un morceau d'un mécanisme permettant à deux processus d'échanger des données, et que ces deux processus ont une relation particulière : ils sont **père** et **fils**.

On se rappelle aussi qu'une particularité importante d'un processus **`fork()`**é est qu'il hérite tout de son père[2], c'est-à-dire aussi les fichiers ouverts avant le **`fork()`** !

Vous l'aurez compris, si le père crée un tube avec **`pipe()`** avant de faire[3] un **`fork()`**, le fils hérite donc de ce tube qui, rappelons-le, est ouvert deux fois, l'une en lecture et l'autre en écriture. Ainsi, père et fils peuvent désormais, l'un lire dans le tube en utilisant le **`fd[0]`** et l'autre écrire dans le tube en utilisant le **`fd[1]`**.

On vient ainsi de mettre en place un mécanisme d'échange, par un tube anonyme, entre père et fils.

Ce qui est expliqué dans ce paragraphe sur **`pipe()`** est essentiel. Si vous ne comprenez pas, demandez à votre enseignant.e de vous l'expliquer.

Pas à pas

Pour mettre en place le mécanisme de tube anonyme, il faut suivre ces étapes :

- Le père crée d'abord un tube anonyme à l'aide de la fonction système **`pipe(int fd[2])`**. Attention, vous devez traiter les erreurs éventuelles.
- Le père crée ensuite un fils à l'aide de la fonction système **`fork()`**
- Côté père, on referme le descripteur de fichier (issu du **`pipe`**) qui ne lui servira pas (**`fd[0]`** sert à lire, **`fd[1]`** sert à écrire dans le tube)
- Côté fils, on referme aussi le descripteur inutile (c'est donc l'autre)
- Père et fils lisent ou écrivent dans le tube (en fonction du sens choisi). Regardez dans le manuel ce que les fonctions **`read()`** et **`write()`** acceptent comme paramètres et aussi ce qu'elles retournent. C'est essentiel pour le bon fonctionnement de votre programme.
- NB : L'écrivain sait ce qu'il va envoyer, le lecteur ne sait pas à l'avance ce qu'il va recevoir. Le lecteur doit donc certainement boucler "tant qu'il y a des données", contrairement à l'écrivain.
- Quand l'écrivain ferme le tube. De cette manière, il indique qu'il n'a plus rien à envoyer, ce qui met aussi fin à la lecture du côté lecteur.

Mettez en œuvre ce mécanisme dans un programme **`tube_anon.c`** et testez-le en envoyant, par l'écrivain, un petit message et en affichant ce qui est reçu par le lecteur.

Encore une fois, sollicitez l'enseignant.e si vous êtes bloqué ou si vous avez besoin de confirmer votre compréhension du problème ou de votre solution.

[2] A l'exception de son **`PID`**.
[3] Le mot essentiel ici est : **AVANT**