TP

Flots d'entrées-sorties et tubes

1 Redirection d'entrées-sorties

1.1

Tester/comprendre les commandes suivantes :

```
touch n.txt
echo 1 > n.txt
echo -1 >> k.text
echo 3 >> n.txt
sort < n.txt
sort < n.txt > outfile 2>&1
ls -l | less
man -s2 stat | tail
cat n.txt | wc -l
echo "Hello" | tr "[:lower:]" "[:upper:]"
```

1.2 Rappels

La primitive `dup2` sert à dupliquer un descripteur de fichier. Elle prend en arguments deux descripteurs de fichiers ouverts (source et destination de la copie). Pour rediriger la sortie standard de l'écran vers un fichier (resp. l'entrée standard du clavier vers un fichier), l'idée est d'appeler `dup2` pour recopier le descripteur du nouveau fichier de sortie (resp. d'entrée) vers celui de la sortie standard (resp. entrée standard). Dans les deux cas, il ne faudra pas oublier de fermer également le descripteur original (celui qu'on vient de dupliquer), puisqu'il ne sert alors plus à rien.

1.3

Ecrire un programme qui exécute `ls -l > toto` (exercice vu en TD).

2 Les tubes

2.1 Rappels de cours

Un tube (en anglais *pipe*) est un mécanisme système particulier qui sert de canal de communication entre deux processus. Un tube possède un flot d'entrée et un flot de sortie, et les données sont lues sur le flot de sortie dans l'ordre selon lequel elles sont écrites dans le flot d'entrée. Un tube peut être créé au niveau du langage de commande par l'opérateur `|`, ou au niveau des appels système par la primitive `pipe()`. Le programme suivant est un exemple d'utilisation des tubes :

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/types.h>
5
6 int main() {
7     pid_t pid;
8     int tube[2];
```

```
9     long entier;
10
11    if (pipe(tube) == -1) {
12        fprintf(stderr, "Erreur de creation du tube\n");
13        exit(1);
14    }
15    pid = fork();
16    switch (pid) {
17      case -1:
18        fprintf(stderr, "Erreur de fork\n");
19        exit(2);
20      case 0:
21        close(tube[1]);
22        read(tube[0], &entier, sizeof(entier));
23        printf("Je suis le fils, j'ai lu l'entier %ld dans le tube\n", entier);
24        close(tube[0]);
25        break;
26      default:
27        close(tube[0]);
28        srandom(pid);
29        entier = random();
30        printf("Je suis le pere, j'envoie l'entier %ld a mon fils\n", entier);
31        write(tube[1], &entier, sizeof(entier));
32        close(tube[1]);
33    }
34    return 0;
35 }
```

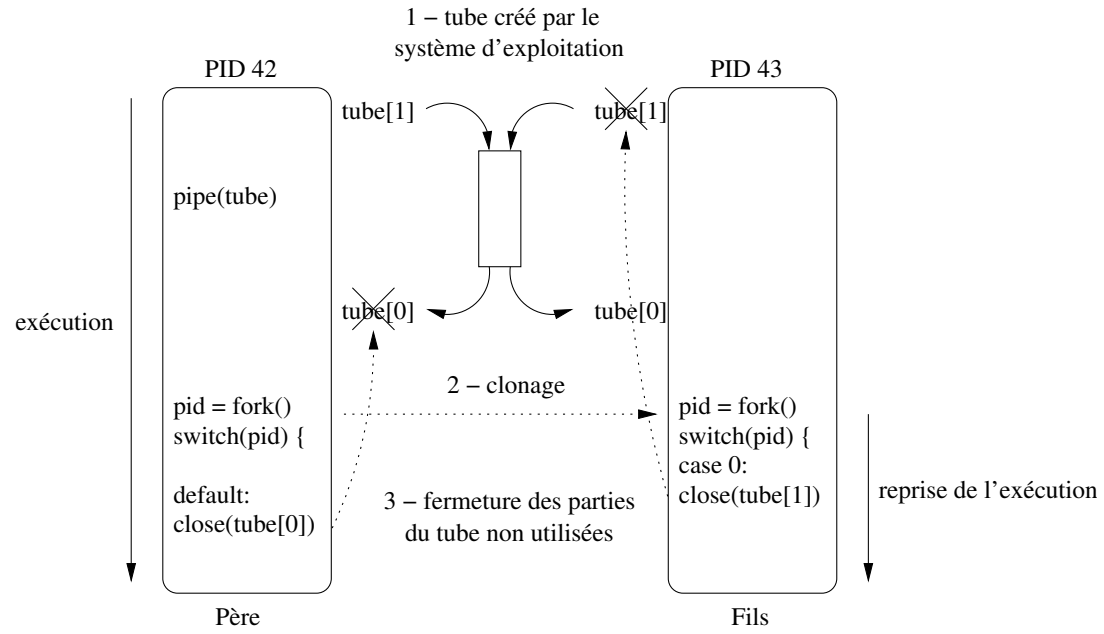

FIGURE 1 – Exemple d'utilisation d'un tube.

Le comportement de ce programme est illustré par la figure 1. Nous pouvons remarquer plusieurs choses :

- le tube est un objet indépendant des processus, créé et géré par le système d'exploitation. Les processus ont connaissance du tube via des points d'accès en lecture et en écriture (appelés descripteurs de fichier), dont la valeur est initialisée par l'appel système `pipe()` lors de la création du tube. Plus précisément, un descripteur de fichier est une structure de donnée maintenue par le

noyau dans son espace mémoire et permettant de conserver des informations sur l'état de l'accès à un objet correspondant à une séquence d'octets (fichier, tube, ...). Du point de vue de l'utilisateur il est identifié par une valeur entière attribuée par le système.
- le tube doit être créé avant le `fork()`, qui copie les descripteurs de fichiers ouverts lors du clonage. Dans le cas contraire, les processus n'ont pas moyen de partager les points d'accès au tube.
- après le `fork()`, les deux processus connaissent les deux points d'accès au tube. Ils peuvent donc tous les deux lire et écrire dans le tube. Néanmoins, le tube est un canal de communication monodirectionnel : si les deux processus écrivent dans le tube, leur données vont se mélanger et être inexploitables.
- chaque processus doit fermer les parties du tube qu'il n'utilise pas. Cela permet de libérer les ressources système associées au point d'accès fermé et cela permet également de propager la fin de fichier dans le tube lorsque le processus qui écrit aura terminé et fermé l'entrée du tube (la fin de fichier est propagée dans un tube lorsque toutes ses entrées sont fermées).
- les accès au tube se font via des primitives de lecture et d'écriture dites de bas niveau : `read()` et `write`, qui permettent respectivement de lire/d'écrire une suite d'octets depuis/vers un point d'accès donné par un descripteur de fichier.

2.2

Ecrire un programme qui crée un tube, puis un fils (rappelons que le fils hérite des descripteurs du père). Le père doit fermer son accès à la sortie du tube, puis effectuer une boucle de lecture sur le clavier. Chaque ligne lue est envoyée en entrée du tube. Cette boucle s'arrête lorsque l'utilisateur presse ctrl-d.

Le fils doit fermer l'entrée du tube, puis effectuer une boucle de lecture sur la sortie du tube. Chaque ligne lue doit être affichée à l'écran (les lignes sont donc affichées progressivement, une par une). Cette boucle s'arrête lorsque l'on détecte que l'entrée du tube a été fermée par le père.

Les fonctions de bas niveau `read()` et `write()` seront utilisées pour les lectures et les écritures.

Question 1 *Comment est ce que le père peut détecter que l'utilisateur veut terminer le programme ?*

Question 2 *Ecrire le programme qui met en oeuvre la spécification.*

Question 3 *Que se passe-t-il si le fils ne ferme pas son accès à l'entrée du tube ? Expliquez ?*

2.3

On reprend le principe de la question précédente en considérant trois processus. Le père crée un tube, puis deux fils. Le fils 1 envoie dans le tube ce qu'il lit au clavier, et le fils 2 affiche à l'écran ce qu'il lit dans le tube.

Le père doit se terminer (et rendre la main au shell) après la fin de ses deux fils.

Question 4 *Quelles sont les conditions d'arrêt pour chacun des fils ?*

Question 5 *Quelle précaution particulière faut-il prendre par rapport à celle du fils 2 ?*

Question 6 *Ecrire le programme qui met en oeuvre la nouvelle spécification.*

3 Un exercice en plus

3.1 Préparation

On souhaite écrire un programme où sont lancés n processus qui vont chacun générer une nombre aléatoire et l'afficher à l'écran. Voici un exemple d'exécution avec 6 processus.

```
$ ring1 6
processus pid 25387 node 2 val = 1430826605
processus pid 25388 node 3 val = 48523501
processus pid 25389 node 4 val = 822619539
processus pid 25390 node 5 val = 1591287596
processus pid 25385 node 0 val = 2047288621
processus pid 25386 node 1 val = 1731323093
$
```

6 processus sont donc lancés. Le nombre de processus est passé en paramètre du programme. Nous vous conseillons de créer les processus en chaine, cad, chaque processus crée un fils, qui crée un fils, etc.

Les processus sont identifiés par leur *pid* (retourné par le `fork`), un numéro de noeud (numéro d'ordre dans la création) et un nombre aléatoire généré par la fonction `random()`. Pour avoir plus d'informations sur cette fonction, vous pouvez les obtenir avec le manuel (`man random`). Prenez garde à ce que les séquences de nombres aléatoires générées par chaque processus soient différentes (voir la fonction `srandom()`).

3.2 Déterminer la plus grande valeur

Nous repartons de l'exercice précédent et allons maintenant faire communiquer les processus, pour déterminer quel est le plus grand nombre qui a été généré aléatoirement.

Pour cela, ajoutez des tubes pour interconnecté les processus : Le premier processus est connecté au deuxième, qui est connecté au troisième, qui est connecté au quatrième, etc.

Le processus initial devra envoyer sa valeur générée aléatoirement à son fil, qui devra la comparer à sa propre valeur et transmettre la plus grand valeur à son fils, et ainsi de suite. Tous les processus devront afficher la valeur qu'ils ont générée et la plus grande valeur observée.