Communication inter-processus par tubes

On compilera en utilisant la ligne de commande `gcc -std=c99`.

Exercice 3.1 Il s'agit d'écrire une commande `faux-pipe cmd1 cmd2 [liste d'arguments]` qui simule la commande `cmd1 | cmd2 [liste d'arguments]`. On utilisera un fichier intermédiaire temporaire pour stocker la sortie standard de la première commande, ce même fichier servira d'entrée standard de la seconde.

Exercice 3.2 On souhaite écrire un programme `vrai-pipe cmd1 cmd2 [liste d'arguments]` qui effectue l'équivalent de la commande *shell* :

```
cmd1 | cmd2 [liste d'arguments]
```

Contrairement à ce qu'il était demandé à l'exercice précédent, il s'agit ici d'utiliser un *tube* pour relier la sortie de la commande `cmd1` à l'entrée de la commande `cmd2`. Faire en sorte que la commande `vrai-pipe` retourne la valeur de sortie de la première commande lorsque celle-ci n'est pas nulle et celle de la seconde autrement.
Vérifiez que l'un des processus ne se termine pas si vous ne fermez aucune extrémité du tube... Quels sont les appels à close() réellement utiles ?

Exercice 3.3 Réalisez une expérience permettant de mesurer la taille du buffer interne d'un tube.

Exercice 3.4 Que se passe-t-il lorsqu'un processus essaye de lire sur un tube alors que celui-ci est fermé en écriture et donc qu'il n'y a plus d'écriture possible sur ce tube ? Même question lorsqu'un processus essaye d'écrire sur un tube alors que celui-ci est fermé en lecture.

Exercice 3.5 On souhaite disposer d'une commande `log` grâce à laquelle on pourrait exécuter un programme quelconque en récupérant automatiquement une copie de sa sortie standard dans un fichier. En quelque sorte, il s'agit de dédoubler chacun des caractères envoyés sur sa sortie standard : un exemplaire est reproduit sur la sortie standard par défaut alors que l'autre est enregistré dans un fichier.
Le programme `log` prend en premier argument le nom du fichier à utiliser pour la sortie. Les arguments suivants constituent la commande à exécuter :

```
log <file> <command> [ <args> ]
```

Pour fixer les idées, voici un exemple d'utilisation de ce programme :

```
[machine] log fichier ls -la
drwxr-xr-x  2 rnamyst rnamyst    80 2008-10-26 22:29 ./
drwxrwxrwt  7 root    root      180 2008-10-26 22:27 ../
-rw-r--r--  1 rnamyst rnamyst     0 2008-10-26 22:29 fichier
-rw-r--r--  1 rnamyst rnamyst 20041 2008-10-26 22:27 toto.pdf
[machine] cat fichier
drwxr-xr-x  2 rnamyst rnamyst    80 2008-10-26 22:29 ./
drwxrwxrwt  7 root    root      180 2008-10-26 22:27 ../
-rw-r--r--  1 rnamyst rnamyst     0 2008-10-26 22:29 fichier
-rw-r--r--  1 rnamyst rnamyst 20041 2008-10-26 22:27 toto.pdf
[machine]
```

Question 1 Pour réaliser le dédoublement de la sortie standard d'un processus, une solution est de rediriger sa sortie standard vers un *tube* à l'extrémité duquel un second processus pourra alors facilement extraire tous les caractères et les sauver dans un fichier en même temps qu'il les affichera sur sa sortie standard.
Le programme `log` va donc créer un processus fils et un tube partagé par le père et le fils. Le fils, par exemple, pourra exécuter la commande spécifiée en paramètre, tandis que le père se chargera de l'affichage et de l'écriture dans le fichier. Donnez le code complet du programme `log` en utilisant autant que possible des fonctions d'entrée/sortie de haut niveau.

Question 2 Une solution plus modulaire serait d'utiliser un programme intermédiaire (que l'on appelera `tee`) dont la tâche serait simplement de lire son entrée standard et de la reproduire à la fois sur la sortie standard et dans un fichier dont le nom serait passé en paramètre. Voici un exemple d'utilisation de `tee` :

```
[machine] echo Salut a tous | tee toto
Salut a tous
[machine] cat toto
Salut a tous
[machine]
```

Donnez le code du programme `tee`.

Question 3 En utilisant `tee`, donnez une version plus simple du programme `log`.

Question 4 En regardant attentivement la sortie donnée en exemple pour la commande `log`, on voit la ligne suivante apparaître :

```
-rw-r--r--  1 rnamyst rnamyst     0 2008-10-26 22:29 fichier
```

Expliquez pourquoi.
Avec votre implémentation de `log`, est-il possible que le fichier n'apparaisse pas toujours ? Est-il possible que la taille observée soit non nulle ? Pourquoi ?

Exercice 3.6

On se propose d'écrire un programme réalisant une série d'opérations sur un ensemble de nombres réels de manière parallèle. Ces nombres sont stokés dans un fichier et l'objectif du programme est de fabriquer un fichier de sortie correspondant au résultat de l'application d'une fonction f (dont le type du résultat est également un nombre réel) à chacun de ces nombres. Le fichier de sortie contiendra donc autant de données que le fichier d'entrée.

Question 0 Écrivez un petit programme permettant de générer un fichier contenant k nombres réels (type C «*double*») stockés au format binaire (k étant donné en paramètre). Écrivez également un programme permettant d'afficher le contenu d'un tel fichier sous forme décimale.

0.1 Exécution en "pipeline"

Question 1 On suppose que le programme (appelons le "`calculer`") sera lancé de cette manière dans le shell :

```
[machine] calculer < fichier_entree > fichier_sortie
```

Écrivez une fonction `int nombre_suivant(double *d)` permettant de lire nombre par nombre l'entrée standard. Cette fonction range nomalement son résultat à l'adresse `d` et renvoie 0, sauf si la fin de fichier est rencontrée, auquel cas elle renvoie -1.

Question 2 La fonction f est en réalité la composée de N fonctions (N étant une constante du programme) g_i. Plus précisémment, $f(x) = g_N(g_{N-1}(...g_1(x)...))$. Dans le programme, on suppose qu'un tableau global `g` contient des pointeurs vers ces fonctions (*i.e.* `g[0]` contient l'adresse de g_1, etc.). L'idée est de faire le traitement demandé à l'aide de N processus organisés en chaîne (et reliés par des tubes), chacun s'occupant d'appliquer une fonction g_i à chacun des nombres qu'il reçoit et de communiquer le résultat au processus suivant.
Écrivez le programme `calculer` réalisant une telle exécution "pipelinée". Attention à ce que le fichier de sortie soit complètement rempli lorsque le programme rend la main au shell...
Utilisez au moins trois fonctions g_i (i.e. $N \geq 3$) pour tester votre programme.

0.2 Exécution par blocs

On considère à présent une fonction f quelconque dont on ne sait pas extraire de parallélisme. Pour accélérer le programme `calculer`, on décide alors de créer P processus (P étant une constante du programme) s'occupant chacun d'une partie du fichier d'entrée (et donc de sortie aussi). L'idée est que chaque processus accède directement à la portion de fichier qui le concerne à la fois en écriture et en lecture.

Question 3 On décide de changer le mode d'utilisation de calculer. On utilise maintenant :

```
[machine] calculer fichier_entree fichier_sortie
```

Expliquer pourquoi ce serait une très mauvaise idée de continuer à utiliser des redirections.

Question 4 En sachant que les différents processus risquent d'accéder simultanément en écriture au fichier de sortie sur des portions distinctes, y-a-t'il des précautions particulières à prendre ? Lesquelles ?

Question 5 Écrivez le code du programme `calculer`.