TP3 : Multi-utilisateurs

Dans ces exercices, nous allons apprendre à gérer les appels à fonction bloquants.

Dans l'exercice précédant, nous avons vu comment échanger des messages entre deux programmes distants, mais nous avons du ordonner les actions du serveur par exemple, pour attendre les différents évènements dans l'ordre (connexion du premier client, connexion du second, puis message du premier). Cet ordonnancement était nécessaire, car les différents appels (*accept(), send(), recv()*, etc) sont bloquants. Dans ces exercices, nous allons lever ces limitations en permettant aux différents programmes d'attendre sur plusieurs descripteurs simultanément.

La fonction fork()

Cette fonction permet de dupliquer un processus. Une fois cette commande exécutée, deux programmes différents seront exécutés. Les différentes variables sont dupliquées dans chacun de ces programmes (c'est-à-dire que changer la valeur d'une variable dans un des programmes de la changera pas dans l'autre), par contre les descripteurs (de socket par exemple) sont partagés entre les deux programmes. Une fois la fonction appelée, les deux processus exécutent les mêmes instructions se trouvant après l'appel à la fonction fork() ;
Appelons père et fils les deux processus résultants de l'appel à la fonction fork(). Sil'on désire exécuter un code différent après l'appel à cette fonction, on pourra faire une condition sur la valeur de retour de la fonction *fork().* En effet, *fork()* retourne 0 dans le processus fils, et retourne un nombre positif (le PID du fils) dans le processus père.
Voici un exemple vu en cours de l'usage de cette fonction :

```
#!/usr/bin/env python2.7

import sys, os

l = [1,2,3]
f = open('out.txt', 'w',0)

childPid = os.fork()

if childPid == 0:
    print "This is the child"
    f.write('this is the child writing\n')
    l.append(4)
    print "l in the child: ", l
else:
    print "This is the parent, child PID: ", childPid
    f.write('this is the parent writing\n')
    l.append(25)
    print "l in the parent: ", l
```

Exerice 1 : Programme serveur avec deux clients et communication bi-directionnelle

(Socket TCP uniquement)

Dans cet exercice, nous allons programmer un serveur qui acceptera la connexion de deux clients. Nous allons rester sur le même principe que précédemment, à savoir que le serveur devra attendre la connexion des deux clients avant toute autre opération. Une fois que les deux clients seront connectés, le serveur devra attendre un message soit du client 1, soit du client 2. Lorsqu'il reçoit un message, le serveur le diffuse vers l'autre client. Un client se connecte au serveur, et ensuite doit attendre à la fois sur une entrée clavier, ou la réception d'un message. Remarquons que le serveur n'a plus besoin d'envoyer un numéro au client qui indiquerait son rôle.

Afin de mettre en œuvre ce fonctionnement, réaliser les tâches suivantes :

- Reprendre le programme serveur et l'un des clients de l'exercice 2.1.
- Sur le serveur : après avoir accepté les deux connexions des deux clients, le serveur fait appel à la fonction *fork()* : le processus père attend un message d'un des clients, et le processus fil attend un message de l'autre client. Lorsque le serveur reçoit un message, il le retransmet immédiatement à l'autre client.
- Sur le client : connecter le client au serveur, puis appeler la fonction *fork()*. Le processus père attend une entrée au clavier, alors que le processus fils attend un message du serveur. Lorsque l'utilisateur entre une chaîne de caractères, le client l'envoie au serveur. Lorsqu'un message arrive du serveur, le client l'affiche.

La fonction select()

La fonction *select()* est une autre approche qu'on peut suivre pour gérer les appels bloquants. La fonction *select()* permet de scruter plusieurs descripteurs simultanément, et d'observer si une opération d'entrée / sortie, ou une erreur s'est produite. Ainsi on peut définir la liste des descripteurs qu'on désire observer, et on pourra traiter les différents évènements sur ces descripteurs.

La signature de la fonction est la suivante :

```
readable , writable , exceptional = select.select(rlist, wlist, xlist[, timeout])
```

- rlist : Liste des descripteurs en lecture
- wlist : Liste des descripteurs en écriture
- xlist : Attente sur une condition exceptionnelle (erreur)
- timeout : (optionnel) Temps maximum en secondes à attendre pour qu'un descripteur soit prêt
- La fonction retourne un triplet de listes de descripteurs qui sont prêts, soit en lecture, en écriture, ou qui ont eu une erreur.

Dans le cadre de nos travaux, nous n'utiliserons que la *rlist*, et donc la liste *readable*. Il faut alors passer en paramètre de la fonction une liste vide ([]) pour *wlist* et *xlist*).

Exemple en pseudo-code :

```
s = socket.socket(…)
s.bind(…)
s.listen(…)
input = [s]

readable, writable, exceptional = select.select(input, [], [])
```

Ce dernier appel à select est bloquant, et si un des descripteurs dans la liste *input* a un message à lire, alors il sera dans la liste *readable*. Il vous suffit alors de parcourir la liste *readable* pour effectuer les bonnes actions (typiquement un *accept()* ou un *recv()*)

Exercice 2 : programme multi-utilisateurs et la fonction select

La fonction *select* permet d'attendre sur plusieurs descripteurs simultanément. En effet, dans le programme précédant, chaque processus est bloqué sur l'attente d'un descripteur. Le select permet à un processus d'attendre sur plusieurs descripteurs, comme l'attente d'un message de la part de n'importe quel client connecté.

Nous allons reprendre le travail de l'exercice précédant, et conserver le même programme client. Un client se connecte, et attend soit une entrée clavier, soit un message du serveur. Par contre, le serveur pourra accueillir un nombre non déterminé à l'avance de clients (non limités à 2 donc).

Pour réaliser cela, reprendre le programme serveur, et permettre au serveur :

- D'accepter toute nouvelle connexion de client
- D'accepter tout nouveau message d'un client connecté (et dans ce cas de le diffuser à tous les autres)
- De gérer les déconnexions des clients

Ce travail est à réaliser en TCP dans un premier temps, et ensuite à reprendre en UDP.