T.P. n°4 : API des socket en C : TCP

1. Compilation d'un programme et fonctions

```
#include <stdio.h>
int main(void) {
        printf("Hello World!!!\n");
        return 0;
}
```

Ce programme est à sauvegarder dans le fichier hello.c (utilisez **gedit** sans bloquer le terminal avec &).
Pour créer un fichier exécutable, le plus simple consiste en la commande : **$ gcc hello.c**
Cette commande effectue l'ensemble des phases de la compilation et crée le fichier exécutable **a.out.** Avec l'option **-o** nous pouvons spécifier le nom du fichier exécutable : **$ gcc -o hello hello.c**
Ici le programme exécutable est **hello**. Une autre possibilité, utile lors de compilation séparée, est de passer par l'intermédiaire de fichiers objets (*.o) :

$ gcc -c hello.c
$ gcc -o hello hello.o

L'option **-Wall** permet d'afficher tous les messages de warnings. Il est recommandé de prendre en compte ces messages afin d'avoir les bonnes habitudes de programmation en C. $ gcc -Wall -o hello hello.c

1. Procéder à la compilation et exécution du programme **hello**

Dans une architecture big-endian, les octets sont conventionnellement numérotés de la gauche vers la droite. Dans le mode big endian les octets de poids fort sont placés en tête et occupent donc des emplacements mémoire avec des adresses plus petites. Dans une architecture little-endian, c'est le contraire.
Le protocole IP définit un standard, le network byte order (soit ordre des octets du réseau). Dans ce protocole, les informations binaires sont en général codées en paquets, et envoyées sur le réseau, l'octet de poids le plus fort en premier, c'est-à-dire selon le mode big-endian et cela quel que soit l'endianness naturel du processeur hôte.

2. Ouvrer le fichier big_little_endian.c. et expliquer son code. Compiler et exécuter ce programme.
3. Ecrire un programme qui lit un entier (en 32 bit) et qui affiche sa représentation hexadécimale en big endian et little endian. Pour cela utilisez une des fonctions vu en cours et %x de printf.

2. Socket en TCP

En C, tout est de bas-niveau, c'est à dire que contrairement au java beaucoup de choses sont à faire à la main, et il sera important d'avoir l'API C bien en tête, si nécessaire, une documentation est disponible sur la page du cours. Voici un petit rappel.
Les prototypes des fonctions supplémentaires nécessaires se trouvent dans les fichiers en-têtes suivants.

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <arpa/inet.h>

Les fonctions principales sont :

int socket(int domaine, int type, int protocole);

int bind(int socket, const struct sockaddr *adresse, socklen_t longueur);

int listen(int socket, int attente);

int accept(int socket, struct sockaddr *adresse, socklen_t *longueur);

int connect(int socket, const struct sockaddr *adresse, socklen_t longueur);

int shutdown(int socket, int how);

int close(int socket);

ssize_t send(int socket,const void *tampon, size_t longueur,int options); ou sendto

ssize_t recv(int socket,void *tampon, size_t longueur,int options); ou recvfrom

struct hostent *gethostbyname(const char *name); int h_errno;

int getpeername(int socket, struct sockaddr *name, int *namelen);

1. Un exemple de code d'un client est donné dans le fichier client.c. Editer ce fichier et essayez de comprendre le code. Essayer de corriger ce code. Indication : format des données
2. Compiler et exécuter le client. Commenter ce qui s'est passé.
3. Un exemple de code d'un serveur est donné dans le fichier serveur.c. Editer ce fichier et essayez de comprendre le code. Compiler et exécuter le serveur. Maintenant lancer le client dans un autre terminal
4. Essayer avec un autre binôme de lancer le client sur une machine et le serveur sur une autre machine. Attention les cartes doivent être Bridgées
5. ´Ecrivez un programme C qui se connecte en TCP au service daytime sur le serveur de la salle (demandez au prof d'activer d'abord ce service) en utilisant des sockets.
6. Faites un serveur qui simule le service daytime.

3. Fork() et les processus zombie

Dans cette partie, vous allez apprendre à créer un seveur qui gère plusieurs connexions en parallèle avec l'appel système fork.

1. Pour le serveur, editer le fichier serv_fork.c et après l'avoir modifié, vous vérifierez son fonctionnement en l'interrogeant à l'aide de la commande telnet. Si les bibliothèques réseaux manquent, ajouter les options suivantes : **-lsocket** (pour charger la bibliothèque des sockets) **-lnsl** (pour charger les utilitaires htons...). Si le serveur est arrêté et relancé peu de temps après, il ne peut pas reprendre le même port d'écoute. Pour pouvoir réutiliser immédiatement ce port il faut utiliser la fonction setsockopt (après socket et avant bind) :
   setsockopt(..., SOL_SOCKET, SO_REUSEADDR, ... , ...)
2. Pour le client, editer le fichier cli_fork.c, Vérifier que ce client transmet bien au serveur les informations qu'on lui a données en utilisant le clavier. On peut aussi rediriger stdin (on suppose que le fichier exécutable s'appelle cli_fork) :
   ls -il | cli_fork ... ...
   cat *.c | cli_fork ... ...
   man ls | cli_fork ... ...
3. Faire interroger simultanément le serveur par plusieurs clients situés sur des machines différentes, ceci pour en vérifier le fonctionnement parallèle
4. Dans l'exemple proposé pour le serveur, le signal SIGCHLD n'est pas géré. La commande ps permettra de voir les zombies résultant de la terminaison des processus créés par le serveur.
5. Utiliser la commande netstat ou netstat -P tcp -f inet pour tracer les communications entre clients et serveur. Observer les autres communications entrantes et sortantes.
6. En s'inspirant du fichier serv_sig.c, modifier votre serveur pour qu'il tient en compte la terminaison de ces fils