TD 4

Processus et sockets

Le but des deux exercices qui suivent est d'étudier la mise en place d'un serveur TCP/IP. Un tel serveur installe une socket d'écoute sur une certaine adresse IP de la machine hôte, et sur un certain port. Ensuite, il se met en attente d'une demande de connexion de la part d'un client sur cette socket d'écoute. Lorsqu'un client demande à se connecter sur le port auquel la socket d'écoute a été associée, le serveur obtient une socket de dialogue pour échanger avec le client.

On ne se préoccupe pas des détails de la phase de dialogue, et on suppose que cela est géré par la fonction suivante :

```
void dial(int sd); // sd est une socket de dialogue
```

On fait l'hypothèse que la fonction `dial()` ne crée pas de nouveau fils, et qu'elle laisse la socket de dialogue ouverte quand elle retourne.

Vous disposez des fonctions suivantes pour faciliter la mise en place des sockets. Une fois une demande de connexion acceptée par `accept_connection()`, l'appel retourne une socket de dialogue, et `dial()` est utilisé pour gérer le dialogue.

```
// Tente de créer une socket côté serveur (localhost), à l'écoute sur toutes les
// interfaces, sur le port passé en paramètre. Affiche des messages sur stderr.
// Retourne la socket d'écoute créée en cas de succès, -1 en cas d'échec.
int create_server_socket(const char* port);

// Côté serveur, se met en attente bloquante d'une connexion sur la socket d'écoute s
// (en utilisant la primitive accept() de l'API POSIX). Affiche des messages sur stderr.
// Retourne la socket de dialogue créée en cas de succès, -1 en cas d'échec.
int accept_connection(int s);
```

Ne vous préoccupez pas de la gestion des cas d'erreurs pour les différentes fonctions, et supposez par conséquent que les cas d'erreur ne se produisent jamais.

EXERCICE 1 ▶ Serveurs « simples », et gestion successive des clients

1) Donnez le code de la fonction `main()` d'un serveur « simple » qui se contente de ne traiter qu'un client puis se terminera. Le port sur lequel le serveur doit se mettre à l'écoute sera passé comme seul argument du programme (ne cherchez pas à rattraper les erreurs concernant le passage des arguments).

2) Modifiez votre fonction `main()` de façon à ce que le serveur ne se termine plus après le traitement d'un seul client, et puisse désormais accepter des demandes de connexion successives de clients.

3) Dans la version actuelle, que va-t-il se passer si un ou plusieurs clients tentent de se connecter au serveur alors qu'un client est déjà en cours de traitement? Vous ne pouvez pas connaître la réponse précise sans étudier précisément le code de `create_server_socket()`, mais il vous est demandé ici d'envisager au moins deux possibilités crédibles.

4) Votre serveur est maintenant capable d'attendre indéfiniment de nouvelles demandes de connexion. Reste le problème de sa terminaison : vous devez mettre en place un gestionnaire pour le signal `SIGTERM`, de façon à ce qu'à la réception de ce signal le serveur finisse le traitement du client courant, puis ferme sa socket d'écoute et se termine. Pour cela, vous utiliserez une variable globale booléenne `quit` initialisée à `false`, mais que votre gestionnaire de signal fera passer à `true` : dès que votre serveur trouvera `quit` à `true`, il devra se terminer. Vous supposerez que, si le signal `SIGTERM` est reçu pendant que le serveur est :

- dans `accept_connection()`, alors l'appel à `accept_connection()` retourne -1 et le gestionnaire est exécuté.
- dans `dial()`, alors l'exécution du gestionnaire est retardée jusqu'à la fin de l'exécution de la fonction.

EXERCICE 2 ▶ Un processus par client

1) Maintenant, votre serveur doit créer un fils à l'arrivée de chaque nouveau client, et chaque fils se chargera du dialogue avec un client. Pour varier les plaisirs, votre serveur va se mettre à l'écoute sur le port fixe 9999. Pour l'instant, ne vous préoccupez pas de la mort des fils, ni de la terminaison du serveur.

2) Que se passe-t-il à la mort des fils? Si votre serveur est très sollicité, que risque-t-il d'arriver?

3) Représentez sur un schéma l'évolution dans le temps des processus en jeu quand le serveur est lancé puis deux clients, `client1` et `client2`, viennent s'y connecter. Le processus principal sera désigné par `main()`, et les deux fils créés par `fils1` (pour s'occuper de `client1`) et `fils2` (pour `client2`). Vous indiquerez sur votre schéma la mise en attente d'une connexion par `sd = accept(s)` (pour alléger un peu), et par `sd = sdk` (avec `k` égal à 0 ou 1) la sortie de l'attente avec une socket de dialogue quand `clientk` vient se connecter.

4) Quelle solution impliquant le signal `SIGCHLD` a-t-on déjà utilisée en TP pour gérer la terminaison des processus, et éviter de conserver des zombies? Si on installe un gestionnaire pour un signal dans un processus, et que ce processus se met en attente dans un appel système bloquant, comme `accept()`, alors à la réception du signal l'appel système est interrompu et retourne une erreur. Dans notre cas, quel problème se pose si l'on tente de mettre en place une solution basée sur la réception de `SIGCHLD` par le père, et comment peut-on envisager de résoudre ce problème?

5) On laisse de côté l'idée d'utiliser `SIGCHLD`, et on suppose que l'on dispose d'une fonction `void sweep(void)`, qui, lorsqu'elle est appelée par un processus père, se charger d'éliminer tous les zombies que ce père a laissés derrière lui : comment utiliser cette fonction pour essayer de limiter le nombre de processus zombies créés par notre serveur? Pouvez-vous garantir que le nombre de zombies ne dépassera jamais une certaine borne?