Programmation système et réseaux

Corrigé du TP 11 : Initiation aux sockets

Arnaud Giersch Guillaume Latu

Commencez par récupérer, dans le répertoire */export/ens/giersch/PSR/TP11/*, les fichiers *client.c* et *serveur.c*. Lisez les codes *client.c* et *serveur.c*. Les fonctions de lecture et d'écriture sur les sockets sont incomplètes et vous devrez les compléter dans les exercices suivants. Par la suite, vous compilerez ces codes en donnant à `gcc` les options `-lresolv -lsocket -lnsl`. Vous noterez les réponses des questions 5 et 6 dans le fichier *reponses.txt*.

1. Créez un shell-script `gccplus` qui appelle `gcc` en lui transmettant ses paramètres plus les trois options `-l...` données dans l'introduction. Vous pourrez ensuite compiler vos programmes avec `gccplus` au lieu de `gcc`. Exemple :

   ```
   gccplus -o serveur serveur.c
   ```

   Correction :

   ```
   #! /bin/bash
   gcc "$@" -lresolv -lsocket -lnsl
   ```

2. Complétez le serveur *serveur.c* en implémentant la fonction gerer_la_connexion. Cette dernière devra lire caractère après caractère sur le descripteur de fichier de la socket correspondant à la connexion avec un client. Pour chaque caractère lu, on l'écrira sur la sortie standard. Lorsqu'il est lancé, le serveur affiche le numéro port sur lequel il écoute.

   Vous testerez ensuite le serveur en le lançant, puis en exécutant au sein d'un autre shell la commande `telnet -ex edumath <port d'écoute>`. Le résultat attendu étant que le serveur écrive sur la sortie standard ce que vous tapez au sein du client.

   Remarque : pour sortir de `telnet`, il suffit de taper « x » puis, à l'invite « `telnet>` », taper le mot « quit ».

   Correction :

   ```
   void gerer_connexion (int sockfd)
   {
       int c, e;

       while ((e = read (sockfd, &c ,1) == 1)) {
           if (write (1, &c, 1) != 1) {
               perror ("! write ()");
               exit (2);
           }
       }
       if (e == -1) {
           perror ("! read ()");
           exit (2);
       }

       close (sockfd);
   }
   ```

3. Complétez le client *client.c* en implémentant la fonction jouer_avec_la_connexion. De la même manière que précédemment, vous lirez sur l'entrée standard caractère après caractère. Vous recopierez chaque caractère lu sur la socket.

   Correction :

```
void jouer_avec_la_connexion (int sockfd)
{
    int c, e;

    while ((e = read (0, &c ,1) == 1)) {
        if (write (sockfd, &c, 1) != 1) {
            perror ("! write ()");
            exit (2);
        }
    }
    if (e == -1) {
        perror ("! read ()");
        exit (2);
    }

    close (sockfd);
}
```

4. Le serveur actuel n'accepte qu'une seule connexion à la fois. On désire le modifier pour qu'il puisse recevoir des caractères provenant de plusieurs clients en même temps.

   À l'aide de la commande fork(), créez des fils au sein du serveur pour gérer de telles connexions multiples. On rappelle que chaque connexion commence au retour de la fonction accepter_connexion. Les processus chargés de la connexion devront évidemment se terminer à la fin de celle-ci.

   (i) Dans un premier temps, chaque connexion sera gérée par le processus père. Votre nouveau serveur sera sauvé dans le fichier *serveur1.c*.

   Correction : *On ne modifie que la fonction* main *:*

```
int main (void)
{
    pid_t pid;
    int port = 0;
    int socket_de_connexion;
    int socket_pour_ecouter = creer_socket_pour_ecouter (&port);

    fprintf (stderr, "# serveur en écoute sur le port %d\n", port);

    while(1) {
        /* on ne gere qu'une connexion a la fois */

        /* Le serveur attend une connexion sur socket_pour_ecouter.
         * Quand un client se connecte, la connexion est ouverte et le
         * serveur peut échanger avec lui en utilisant la valeur de
         * retour de accepter_connexion : socket_de_connexion.
         */
        socket_de_connexion = accepter_connexion (socket_pour_ecouter);

        /* création d'un fils pour continuer le travail de serveur
         * pendant que le père s'occupe de la connexion */
        if ((pid = fork ()) == -1) {
            perror ("! fork ()");
            exit (1);
        } else if (pid != 0) {
            const char *joli_nom =
                parametres_connexion(socket_de_connexion, DISTANT);
            fprintf (stderr, "# connexion acceptée pour %s\n", joli_nom);

            gerer_connexion (socket_de_connexion);

            fprintf (stderr, "# connexion raccrochée pour %s\n", joli_nom);
```

```
            exit (0);
        }

    }
}
```

(ii) Dans un second temps, chaque connexion sera gérée par le processus fils. Vous stockerez ce serveur dans le fichier *serveur2.c*.

Correction :

```
int main (void)
{
    pid_t pid;
    int port = 0;
    int socket_de_connexion;
    int socket_pour_ecouter = creer_socket_pour_ecouter (&port);

    fprintf (stderr, "# serveur en écoute sur le port %d\n", port);

    while(1) {
        /* on ne gere qu'une connexion a la fois */

        /* Le serveur attend une connexion sur socket_pour_ecouter.
         * Quand un client se connecte, la connexion est ouverte et le
         * serveur peut échanger avec lui en utilisant la valeur de
         * retour de accepter_connexion : socket_de_connexion.
         */
        socket_de_connexion = accepter_connexion (socket_pour_ecouter);

        /* création d'un fils pour s'occuper de la connexion */
        if ((pid = fork ()) == -1) {
            perror ("! fork ()");
            exit (1);
        } else if (pid == 0) {
            const char *joli_nom =
                parametres_connexion(socket_de_connexion, DISTANT);
            pid = getpid ();
            fprintf (stderr, "#[%u] connexion acceptée pour %s\n",
                     (unsigned)pid, joli_nom);

            gerer_connexion (socket_de_connexion);

            fprintf (stderr, "#[%u] connexion raccrochée pour %s\n",
                     (unsigned)pid, joli_nom);
            exit (0);
        }
    }
}
```

5. Pour lequel des deux derniers serveurs et dans quelle situation observe-t-on des zombies ? Pour observer des zombies vous regarderez le manuel de `ps` et le deuxième champs du résultat de la commande `ps -l -u $USER`. Pourquoi la présence de ces zombies ?

   Correction : *Les zombies sont observés lorsque les fils sont chargés de gérer la connexion, car le père ne récupère pas l'état de ses fils. Dans l'autre version, lorsque le père chargé de la connexion meurt, son état est lu par le processus init (1).*

6. Comme dans la question 4, écrivez dans le fichier *serveur3.c* un serveur capable de gérer plusieurs connexions simultanées. Dans cette troisième version, c'est à un petit-fils de gérer la connexion. Le père crée un fils, qui crée le petit-fils chargé de la connexion. Le fils se suicide immédiatement après avoir créé le petit-fils.

Correction :

```
int main (void)
{
    pid_t pid;
    int port = 0;
    int socket_de_connexion;
    int socket_pour_ecouter = creer_socket_pour_ecouter (&port);

    fprintf (stderr, "# serveur en écoute sur le port %d\n", port);

    while(1) {
        /* on ne gere qu'une connexion a la fois */

        /* Le serveur attend une connexion sur socket_pour_ecouter.
         * Quand un client se connecte, la connexion est ouverte et le
         * serveur peut échanger avec lui en utilisant la valeur de
         * retour de accepter_connexion : socket_de_connexion.
         */
        socket_de_connexion = accepter_connexion (socket_pour_ecouter);

        /* création d'un petit-fils pour s'occuper de la connexion */
        /* (i) création du fils */
        if ((pid = fork ()) == -1) {
            perror ("! fork ()");
            exit (1);
        } else if (pid == 0) {
            /* (ii) création du petit-fils */
            if ((pid = fork ()) == -1) {
                perror ("! fork ()");
                exit (1);
            } else if (pid == 0) {
                /* (iii) le petit-fils gère la connexion */
                const char *joli_nom =
                    parametres_connexion(socket_de_connexion, DISTANT);
                pid = getpid ();
                fprintf (stderr, "#[%u] connexion acceptée pour %s\n",
                         (unsigned)pid, joli_nom);

                gerer_connexion (socket_de_connexion);

                fprintf (stderr, "#[%u] connexion raccrochée pour %s\n",
                         (unsigned)pid, joli_nom);
                exit (0);
            }
            /* (iv) le fils se suicide */
            exit (0);
        }
        /* (v) le père attend la fin du fils */
        waitpid (pid, NULL, 0);
    }
}
```

Comment et pourquoi cette dernière version permet d'éviter les processus zombies observés à la question 5 ?

Correction : *En prenant soin de faire que le père attende la fin de ses fils (avec la fonction* wait *ou* waitpid*), on n'observe plus de zombies. Lorsque les processus chargés de la connexion (c.-à-d. les petits-fils) meurent, comme leur père n'est plus là (il s'est suicidé immédiatement), leur état est lu par le processus init (1).*

7. (*question subsidiaire*) On désire modifier le client pour qu'il n'utilise plus les appels systèmes read et write, mais les fonctions « bufferisées » de la bibliothèque standard getchar et putc. Consultez pour cela le manuel de fdopen (section 3C du manuel), puis écrivez le nouveau client dans le fichier *client1.c*.

Correction : *Il suffit de réécrire la fonction* jouer_avec_la_connexion.

```
void jouer_avec_la_connexion (int sockfd)
{
    FILE *f;
    int c;

    if ((f = fdopen (sockfd, "w")) == NULL) {
        perror ("! fdopen ()");
        exit (2);
    }
    setbuf (f, NULL);

    while ((c = getchar ()) != EOF)
        putc (c, f);

    fclose (f);
}
```