F2B205B - Programmation Système et Réseaux
Partie Unix

C. Lohr

16 Janvier 2009

Modalités

- Document de cours autorisés (polys, sujets de TP, listings de TP)
- Réponses à rendre sur copie séparée.
- Durée indicative : 40mn

1 Première question : réaliser `envoie_fichier()`

On vous demande de présenter votre implémentation d'une fonction appelée `envoie_fichier()`[1] qui réalise la copie de données entre deux descripteurs de fichiers.

L'annexe 3 documente à la façon du man ce que serait cette fonction.

2 Deuxième question : appels systèmes bloquants

Les appels systèmes de type lecture/écriture sur des descripteurs de fichiers sont dits *bloquants*. Nous avons vu en TP qu'il était possible de les rendre non bloquants[2], mais par défaut ils sont bloquants.

Expliquez en quelques lignes l'intérêt d'avoir des appels systèmes *bloquants*. Que se passe t-il lorsque qu'on les rend non bloquants ?

3 Troisième question : trouvez l'erreur

L'annexe 3 donne le code d'un petit serveur TCP concurrent. Ce code est syntaxiquement correcte et compile sans problème.

- Tout d'abord, expliquez en quelques phrases en quoi ce programme est un *serveur TCP*, et d'autre part en quoi il est *concurrent*. Vous indiquerez notamment grâce à quels appels systèmes cela est possible.

- Malheureusement, ce programme a un comportement inattendu. Lors de son exécution, lorsque le client ferme sa connexion, apparaît alors le message d'erreur suivant :

```
Erreur accept : Bad file descriptor
```

Ce code comporte une erreur, une erreur que les trois quarts des élèves font la première fois qu'ils essaient de programmer un petit serveur concurrent. On vous demande d'une part de trouver et corriger l'erreur, et d'autre part d'expliquer pourquoi cette erreur provoque ce message d'erreur.

[1] Les connaisseurs veront là quelques similitudes avec l'appel système `sendfile()`.

[2] Pour mémoire : on utilise l'appel système `fcntl()` pour positionner le drapeau `O_NONBLOCK`.

Annexes

ENVOIE_FICHIER(3) Manuel du programmeur SLR-TW3S

NOM

`envoie_fichier` - Transfert de données entre descripteurs de fichier

SYNOPSIS

```
ssize_t envoie_fichier( int out_fd, int in_fd, size_t count) ;
```

DESCRIPTION

`envoie_fichier` copie des données entre deux descripteurs de fichier.

`in_fd` doit être un descripteur de fichier ouvert en lecture, et `out_fd` un descripteur ouvert en écriture. Typiquement, `out_fd` correspond à une socket.

L'argument `count` est le nombre d'octets à copier entre les descripteurs de fichiers.

VALEUR RENVOYÉE

Si le transfert a réussi, le nombre d'octets écrits dans `out_fd` est renvoyé. Sinon, **`envoie_fichier`** renvoie -1.

Erreur sur un petit serveur TCP concurrent

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/socket.h>
5  #include <netinet/in.h>
6  #include <netdb.h>
7  #include <stdio.h>
8  #include <signal.h>
9  #include <string.h>
10
11 #define BUFSIZE 512
12
13 void communication(int soc, struct sockaddr_in *from)
14 {
15   struct hostent *hp;
16   char buf[BUFSIZE];
17   int r;
18   char *message = "Envoyez votre message : ";
19
20   /* Reconnaissance de la machine cliente */
21   hp = gethostbyaddr((char *)&(from->sin_addr),
22                      sizeof(struct in_addr), from->sin_family);
23   if (hp == NULL) {
24     fprintf(stderr, "Erreur gethostbyaddr\n");
25     shutdown(soc, 2);
26     return;
27   } else
28     printf("Machine appelante: %s\n", hp->h_name);
29
30   /* Boucle de communication */
31   for (;;) {
32     /* Ecriture socket */
33     r = write(soc, message, strlen(message));
34     /* lecture socket */
35     r = read(soc, buf, BUFSIZE);
36     if (r <= 0) {
37       printf("fin client %s\n", hp->h_name);
38       break;
39     }
40     /* envoi sur le terminal de ce que le serveur a recu */
41     buf[r] = '\0';
42     printf("%s : %s\n", hp->h_name, buf);
43   }
44 }
45
46 int main(int argc, char **argv)
47 {
48   int s, ns, r, pid, port;
49   socklen_t fromlen;
50   struct sockaddr_in sin, from;
51
52   if (argc != 2) {
53     printf("Usage : %s port_serveur\n", argv[0]);
54     exit(1);
55   }
56
57   port = atoi(argv[1]);
58   if (port < 5000) {
```

```
59      printf("donnez un numero de port superieur a 5000\n");
60      exit(1);
61    }
62
63    /* Construction de l'adresse locale (pour bind) */
64    memset((char *)&sin, (unsigned char)0, sizeof(sin));
65    sin.sin_family = AF_INET;
66    sin.sin_addr.s_addr = INADDR_ANY;
67    sin.sin_port = htons(port);
68    /* Creation de la socket */
69
70    s = socket(PF_INET, SOCK_STREAM, 0);
71    if (s == -1) {
72      perror("socket");
73      exit(2);
74    }
75
76    /* Association d'un port a la socket */
77    r = bind(s, (struct sockaddr *)&sin, sizeof(sin));
78    if (r == -1) {
79      perror("bind");
80      exit(3);
81    }
82    /* Positionnement de la socket en mode listen */
83    /* la machine d'etats TCP est positionnee dans l'etat listen */
84    listen(s, 5);
85
86    /* Pour eviter les zombies */
87    signal(SIGCHLD, SIG_IGN);
88    fromlen = sizeof(from);
89    /* Boucle Serveur */
90    for (;;) {
91      ns = accept(s, (struct sockaddr *)&from, &fromlen);
92      if (ns < 0) {
93        perror("Erreur accept");
94        exit(1);
95      }
96      pid = fork();
97      switch (pid) {
98        case -1:
99          perror("fork impossible");
100         continue;
101         break;
102       case 0:                    /* fils */
103         close(s);
104         communication(ns, &from);
105
106         break;
107       default:                   /* pere */
108         close(ns);
109         break;
110     }
111   }
112 }
```