La programmation client/serveur avec sockets

Exo15 Serveur de calcul et utiliser un serveur concurrent:

Q15.1 Adapter le serveur qu'il puisse exécuter plusieurs opérations simultanément en créant dynamiquement des processus.(père/plusieurs fils)

```
#include <stdio.h>    #include <stdlib.h> //Client
#include <string.h>   #include <unistd.h>
 #include "fonctionsSocket.h"
int main(int argc, char **argv)
{    int sock, err; // desc de la socket locale+ code err
     char boucle = 'o';
     char operateur;       /* Saisie de l'operation */
     int operande1, operande2, resultat;

  if ( argc != 3 ) {    printf("usage : client nom_machine
no_port\n");
   exit(1);  }
  sock = socketClient_EAD( argv[1], atoi( argv[2]) );
 if ( sock < 0 ) {     printf( "client : erreur socketClient\n");
exit(2);  }

  do {  /** Saisie de l'operation */
    printf("\t donner un operateur : ");
   scanf(" %c", &operateur );
   printf("\t donner l'operande 1 : ");
   scanf(" %d", &operande1 );
   printf("\t donner l'operande 2 : ");
   scanf(" %d", &operande2 );
 /** Envoi de l'operation en plusieurs parties  */
err = send( sock, (void*) &operateur, sizeof(operateur), 0);
 if ( err!= sizeof(operateur) ) { perror("err send
d'operateur");
     shutdown(sock, 2);  exit(3);    }
 err = send( sock, (void*) &operande1, sizeof(operande1),
0);
 if ( err!= sizeof(operande1) ) { perror(" errr le send
d'operande1");
     shutdown(sock, 2);  exit(4);  }
 err = send( sock, (void*) &operande2,sizeof(operande2),
0);
   if ( err!= sizeof(operande2) ) { perror( err le send
d'operande2");
     shutdown(sock, 2); exit(5);   }
  /*Reception du resultat */
err = recv ( sock, (void *) &resultat, sizeof( resultat ), 0 );
   if ( err == -1 ) { perror("client : erreur a la reception");
     shutdown(sock, 2); exit(6);  }
   printf("client : resultat recu : %d\n", resultat);
    /* * On continue ?  */
    printf("client : on continue = o : ");
 /* Attention, l'espace avant le %c permet de vider le buffer
du caractere CR (enter) de la saisie precedente*/
    scanf(" %c", &boucle );
    printf("\n");
  } while ( boucle == 'o' );
  shutdown(sock, 2);
 close(sock);
  return 0;
}
```

Crée dynamiquement des processus(père/plusieurs fils) Q15.2 serveur en utilisant un pool de processus.

```c
#include <string.h>  #include <stdio.h> //Serveur dynamic
#include <stdlib.h>  #include <signal.h> #include <unistd.h>
 #include <errno.h>  #include <sys/socket.h>
#include <netinet/in.h> #include <sys/types.h>
#include <sys/wait.h>
#include "fonctionsSocket.h"
/* taille du buffeur de reception */
#define TAIL_BUF 100
// Gestion fin processus fils
void finFils()  //void finFils
{//chaque client est traité par un processus différent!!!
    int status;
  // Wait for the child to be finished
 wait( &status );
}
// Fonction de traitement de la requete utilisee par
// le processus fils
void traitReq( int sockTrans )
{
 char         operateur;     /* buffer de reception */
 int          operande1, operande2;
 int          resultat;    /* ... de l'operation */
 int          encore; // test de boucle pour les envois
 pid_t        myPid;       /* Identif du process */
 int          err;         /* code d'erreur */
  encore = 1;
 myPid = getpid();
  while ( encore == 1 ) {
   /** Reception et affichage de l'operation en provenance
du client  * si ce dernier a coupe la connexion, on sort sans
rien recevoir   */
 err = recv( sockTrans, &operateur, sizeof(operateur), 0);
 if (err < 0) { perror(" err dans la reception  d'operateur");
    shutdown(sockTrans, 2);  exit(4);  }

if ( err == 0 ) { printf("serveur %d : fin de la conn client\n",
myPid);
     encore = 0;
   } else {      // reception des operandes
err = recv( sockTrans, &operande1, sizeof( operande1 ), 0);
 if (err < 0) {  perror(" erreur dans la reception d'operande1");
   shutdown(sockTrans, 2);   exit(5);    }
err = recv( sockTrans, &operande2, sizeof( operande2 ), 0);
if (err < 0) { perror(" err dans la reception d'operande2");
   shutdown(sockTrans, 2);   exit(6);   }
printf("serveur %d : voila l'operation recue : %d %c %d\n",
            myPid, operande1, operateur, operande2 );
 switch ( operateur ) {
    case '+' : resultat = operande1 + operande2; break;
    case '-' :  resultat = operande1 - operande2; break;
    case '*' :  resultat = operande1 * operande2; break;
    case '/' :  resultat = operande1 / operande2; break;
    default :  printf("serveur %d : inconnu\n", myPid); resultat = 0;
   }
err = send( sockTrans,(char*)&resultat,sizeof( resultat ), 0 );
 if ( err != sizeof( resultat ) ) {perror("err envoi du resultat");
   shutdown(sockTrans, 2);   exit(7);   }
   } //else}//while
}
```

```c
#include <string.h>  #include <stdio.h>  //Serveur pool
#include <stdlib.h>  #include <signal.h> #include <unistd.h>
 #include <errno.h>  #include <sys/socket.h>
#include <netinet/in.h> #include <sys/types.h>
#include <sys/wait.h>  #include "fonctionsSocket.h"
/* taille du buffeur de reception */
#define TAIL_BUF 100
/* Nombre de processus dans le pool */
#define NB_PROCS_POOL  0
// Gestion fin processus fils
void finFils()
{  int status;
  // Wait for the child to be finished
 wait( &status );
}
// Fonction de traitement de la requete utilisee par
// le processus fils
void traitReq( int sockTrans )
{
 char         operateur;     /* buffer de reception */
 int          operande1;   /* operandes */
 int          operande2;
 int          resultat;    /* ... de l'operation */
 int          encore; // test de boucle pour les envois
 pid_t        myPid;       /* Identif du process */
 int          err;         /* code d'erreur */
  encore = 1;
 myPid = getpid();
  while ( encore == 1 ) {
   /** Reception et affichage de l'operation en provenance
du client  * si ce dernier a coupe la connexion, on sort sans
rien recevoir   */
 err = recv( sockTrans, &operateur, sizeof(operateur ), 0);
 if (err < 0) { perror(" err dans la reception  d'operateur");
    shutdown(sockTrans, 2);  exit(4);  }

if ( err == 0 ) { printf("serveur %d : fin de la conn client\n",
myPid);
     encore = 0;
   } else {      // reception des operandes
err = recv( sockTrans, &operande1, sizeof( operande1 ), 0);
 if (err < 0) {  perror(" erreur dans la reception d'operande1");
   shutdown(sockTrans, 2);   exit(5);    }
err = recv( sockTrans, &operande2, sizeof( operande2 ), 0);
if (err < 0) { perror(" err dans la reception d'operande2");
   shutdown(sockTrans, 2);   exit(6);   }
printf("serveur %d : voila l'operation recue : %d %c %d\n",
            myPid, operande1, operateur, operande2 );
 switch ( operateur ) {
    case '+' : resultat = operande1 + operande2; break;
    case '-' :  resultat = operande1 - operande2; break;
    case '*' :  resultat = operande1 * operande2; break;
    case '/' :  resultat = operande1 / operande2; break;
    default :  printf("serveur %d : inconnu\n", myPid); resultat = 0;
   }
err = send( sockTrans,(char*)&resultat,sizeof( resultat ), 0 );
 if ( err != sizeof( resultat ) ) {perror("err envoi du resultat");
   shutdown(sockTrans, 2);   exit(7);   }
   } //else}//while
}
```

```
//**********----------------------------------********
// Fonction principale
int main(int argc, char** argv)
{
   int  sock_cont, sock_trans;  // desc sockets locales
   struct sockaddr_in nom_transmis
    socklen_t      size_addr_trans;
     int pid;            /* PID du processus fils */

   if ( argc != 2 ) {printf ( "usage : serveur no_port\n" ); exit( 1 );  }
/* Gestion fin des processus fils = pour eviter zombies     */
signal( SIGCHLD, finFils); //reveil un P dont un fils vient
demourir
   size_addr_trans = sizeof(struct sockaddr_in);
    /* * Creation de la socket, protocole TCP  */
      sock_cont = socketServeur_EAD( atoi( argv[1] ) );
   if ( sock_cont < 0 ) { printf( " err  socketServeur\n" ); exit( 2 ); }
    /* * Boucle du serveur */
   for (;;) {
     /* * Attente de connexion  cree socket transmission */
     sock_trans = accept(sock_cont,
           (struct sockaddr *)&nom_transmis,
           &size_addr_trans);
    if (sock_trans < 0) { perror("err  accept"); exit(3);   }
     pid = fork();   //crée un fils si retour=0
switch ( pid ) {
    case 0 :  // Processus fils, appel de la fonction
                // de traitement des requetes
            close( sock_cont);
            traitReq( sock_trans );

             /* * arret de la connexion et fermeture */
            shutdown(sock_trans, 2);
             close(sock_trans);
            // fin du fils
             exit(0);
               break;

     case -1 : // erreur
        perror("Creation processus fils");

          // Fermeture des sockets et fin
          close( sock_cont );
           shutdown(sock_trans, 2);
           close(sock_trans);
            exit(3);

     default : // processus pere
   // Ferme la socket connectee et attend la suite
       close(sock_trans);

    }
   }
  return 0;
}
```

```
//********************----------*****
// Fonction principale
int main(int argc, char** argv)
{
   int   sock_cont, sock_trans,  i;  /* i iteration */
    struct sockaddr_in nom_transmis;
     socklen_t      size_addr_trans
     int   pid;            /* PID du processus fils */

   if ( argc != 2 ) {printf ( " serveur no_port\n" ); exit( 1 );   }
    /* * Gestion fin des processus fils = pour eviter zombies
   signal( SIGCHLD, finFils);
    size_addr_trans = sizeof(struct sockaddr_in);
    /* * Creation de la socket, protocole TCP   */
       sock_cont = socketServeur_EAD( atoi( argv[1] ) );
  if ( sock_cont < 0 ) { printf( "err socketServeur\n" ); exit( 2 );    }
  /* * Initialisation du pool  */
  for ( i = 0 ; i < NB_PROCS_POOL ; i++) {
     pid = fork();

 switch ( pid ) {
     case 0 :  // Processus fils, appel de la fonction
                 // de traitement des requetes
            printf("Le processus %d est lance\n", i );
     /** Boucle des requetes*/
      for (;;) {
     /*  * Attente de connexion  cree socket transmission */
     sock_trans = accept(sock_cont,
              (struct sockaddr *)&nom_transmis,
              &size_addr_trans);
 if (sock_trans < 0) { perror("serveur :  erreur sur accept"); exit(3)}

     traitReq( sock_trans );
    /** arret de la connexion et fermeture */
     shutdown(sock_trans, 2);
     close(sock_trans);
   }
    // Jamais atteind
   break;
     case -1 :  perror("Creation processus fils");
        // Fermeture des sockets et fin
       close( sock_cont ); shutdown(sock_trans, 2);
         close(sock_trans);
          exit(4);
    default : // processus pere  break;
    }
   }
 // Le processus pere travaille comme les autres
    // Boucle des requetes
    for (;;) {     /* * Attente de connexion  */
     sock_trans = accept(sock_cont,
           (struct sockaddr *)&nom_transmis,
           &size_addr_trans);
       if (sock_trans < 0) {perror("err sur accept");exit(5);   }

        traitReq( sock_trans );
       shutdown(sock_trans, 2);
       close(sock_trans);
   }
  return 0;
}
```

Q15.3 Avantages des différentes solutions ?

La seconde solution évite d'avoir à créer un processus à chaque connexion du client ce qui est assez lourd. Mais lorsque tous les processus du pool sont occupés avec un client, il n'y a plus personne pour répondre aux nouvelles demandes de connexion. (demande reste en attente)

Si le temps moyen de traitement d'une connexion n'est pas très long, la seconde solution aussi est satisfaisante plus rapide et moins consommatrice de processus. En revanche si les temps de connexion sont longs il vaut mieux utiliser la première solution : en plus le temps de création du processus sera amorti sur ce temps mais il y a un risque de dépasser le nombre maximum de processus autorisé par le système.

Exo16 Serveur multi-Services

Le processus xinetd permet de regrouper en un seul processus l'accès à différents services (daemon) pour éviter de lancer autant de processus que de service.
Lorsque le processus reçoit une demande de connexion, il lance l'exécution du programme correspondant comme par exemple: /usr/sbin/in.telnetd pour telnet ou /usr/sbin/rlogind pour rlogin. Chacun des services est accédé à travers son propre numéro de port comme ftp/21 telnet/23 time/37
Q16.1 Proposer un squelette d'implantation d'un processus xinetd (la config du serveur xinetd /etc/xinetd.d)

Pour réaliser ce serveur multi-service, il est nécessaire de créer autant de socket que de services à rendre.
À chaque socket, un numéro de port est affecté à l'aide de la fonction bind.
Le serveur possède ainsi plusieurs socketServeur avec chacune son propre numéro de port. Par exemple, il créera une socket sur le port 21 pour offrir des services ftp.

Pour rendre l'ensemble des services à la fois, le serveur se met en attente sur l'ensemble de ces sockets avec un select. De cette manière, il n'est réveillé que lorsque l'un de ses services est invoqué. Lors de son réveil, après la fonction select, il consulte l'ensemble des descripteurs prêts pour identifier la socket sur laquelle il a reçut la requête. Une fois la socket identifiée, il peut savoir quelle est l'application qui lui est demandée puisque chaque socket a un numéro de port différent. Par exemple, si la socket prête en réception est celle à laquelle il a donné le port 23, il sait que le client lui demande un telnet. Il sait alors quel service il doit lancer.

Pour pouvoir continuer à rendre le service xinetd, le processus serveur se duplique, à l'aide de la fonction fork. C'est le processus fils qui prend en charge le traitement de la requête, donc l'exécution de l'application. L'exécution d'une application par un processus se fait par la fonction exec qui a pour effet de recouvrir le processus existant avec le code de la nouvelle application. Lorsque l'application se termine, le processus fils meurt et le père reçoit un signal SIG CHILD qui lui signale la fin de ce processus fils. Du point de vue des sockets, le processus père doit systématiquement fermer les sockets de communication qui sont crées par la fonction accept sans attendre la fin du processus fils. Attention, il ne doit pas fermer la connexion car cela affecterait aussi la connexion du fils.
La gestion de la connexion étant indépendante de la gestion des sockets. Il faut noter que le processus fils, qui exécute l'application, profite toujours de la table des descripteurs de fichiers telle qu'elle est définies dans le processus père. En effet, cette table se transmet par le fork et elle est conservée par l'exec. Certaines applications redirigent ainsi leurs entrées-sorties à travers la socket connectée. Il est alors possible d'exécuter des applications interactives (ex : telnet, ftp ou ssh) à distance à condition que le processus client transmette ce qu'il saisit sur la socket.