Prénom, Nom :	*Groupe :*

Test de Réseau
le 19 janvier 2007

Conseils et demandes

N'utilisez pas de rouge. Écrivez lisiblement et assez grand. Prenez votre temps et regardez tout le document. N'oubliez-pas que même en `C++` les `int` et les `char *` ne sont pas de « classes », n'ayant pas de méthode du style `.size()`. Utilisez des wrappers PARTOUT. La syntaxe d'un ***appel correct*** de `Write()` est

```
Write(descripteur, tampon, nombreDOctetsATransfererDepuisleTampon);
```

où `descripteur` est un **entier** et tampon est un ***pointeur***. Pour utiliser un **`string maChaine`** il faut faire

```
Write(descripteur, maChaine . c_str(), maChaine . size());
```

Pour lire, vous allez utiliser une fonction spéciale `LireN()`, avec le même type de paramètres, mais là, pas de `string`s du tout. Ces fonctions seront suffisantes pour rédiger tout ce qui est demandé concernant le transfert de données.

Commencez par les questions qui suivent, et passez juste cinq ou dix minutes dessus.

1 Questions – une seule réponse correcte

1. Une *socket* est
   (a) un processus spécial, nommé aussi *serveur*
   (b) un type particulier de mutex
   (c) un point de communication entre processus
2. Un *protocole de communication* est
   (a) un programme écrit en C++ utilisant des sockets
   (b) un ensemble de règles gérant tous les cas possibles pour le dialogue réseau
   (c) un processus qui écoute pour les connexions entrantes
3. Le *protocole IP* est un protocole
   (a) couche liaison de données
   (b) couche transport
   (c) couche physique
   (d) couche réseau
4. Une *adresse IP* contient
   (a) trois octets
   (b) quatre octets
   (c) cinq octets
5. Un *serveur DNS* sert
   (a) à faire arriver les packets IP à destination – Direct Node Selection
   (b) à distribuer les news – Distributed News Service
   (c) à obtenir la correspondance entre noms de sites et leurs adressses IP – Domain Name System
6. La fonction `htons()`
   (a) convertit un nombre entier long du format hôte vers le format réseau
   (b) convertit un nombre entier court du format réseau vers le format hôte
   (c) convertit un nombre entier court du format hôte vers le format réseau

7. On a besoin des fonctions `htons()`, `htonl()`, `ntohs()` et `ntohl()`
   (a) pour passer d'ASCII à UNICODE
   (b) pour pouvoir transférer les caractères accentués par ftp
   (c) parce que ce qui transite sur le réseau est en big endian, alors que différentes architectures ont leurs ordres particuliers (little endian, etc,), et on dispose ainsi d'une convention uniforme pour transmettre des valeurs numériques

2 À votre service

Complétez le code de la fonction suivante, qui met en place une socket côté serveur et l'enregistre pour l'écoute

```
int ServerInit(::uint16_t & NumPort, int fileAttente = 5) {
    const int sd = ::socket (PF_INET, SOCK_STREAM, 0);
    if(-1 == sd) throw CExcFctSyst ("socket()");
    ::sockaddr_in Addr;
    memset (& Addr, 0, sizeof (Addr));
    Addr.sin_family      = AF_INET;
    Addr.sin_port        = NumPort;
    Addr.sin_addr.s_addr = INADDR_ANY;
    if (::bind (sd, reinterpret_cast < ::sockaddr *> (&Addr),
                sizeof (Addr))) {
        Close (sd);
        throw CExcFctSystFile ("bind()", sd);
    }
    ::socklen_t Lg = sizeof (Addr);
    if (::getsockname (sd, reinterpret_cast < ::sockaddr *> (&Addr), &Lg)) {
        Close (sd);
        throw nsSysteme::CExcFctSystFile ("getsockname()", sd);
    }
    NumPort = Addr.sin_port;
    ::listen (sd, fileAttente);
    return(sd);
}
```

3 Exercice – client et serveur FTP simpliste

3.1 Travail à faire

3.1.1 Fonction auxiliaire `LireN()`

```
int LireN(int socket, char *buff, int count = 0,
          bool qSetNull = true) {
    if(count > 0) {
        int nbTotLus(0), patience(PATIENCE);
        for(bool qNotEOL (true); patience && nbTotLus < count && qNotEOL;) {
            int nbLus (0);
            try {
                nbLus = Read(socket, buff + nbTotLus, 1);
                nbTotLus += nbLus;
                qNotEOL = (buff[nbTotLus - 1] != '\n');
            }
            catch (...) {
                Write(socket, FatalTxt . c_str(), FatalTxt . size());
                exit(1);
```

```
            }
            if(!nbLus) {
                ::sleep(1);
                patience--;
            }
        }
        if(qSetNull) {
            buff[nbTotLus-1] = '\0';
        }
        return nbTotLus;
    }
    return 0;
}
```

Cete fonction lit au plus `count` *octets, les déposant dans* `buff`*, s'arrêtant au premier "\n" et rendant le nombre d'octets lus (**attention**, le "\n" **n'est pas** à mettre dans* `buff`*) ; si* `qSetNull` *est vrai, alors avant de finir, cette fonction rajoute un "\0" dans* `buf`*, à la fin de ce qu'elle vient y déposer.*

3.1.2 Authentification serveur

```
void ServFTP(::uint16_t NumPort) {
    const int socketEcoute (ServerInit(NumPort));
    cout << "Le numero de port principal d'ecoute est "
         << ::ntohs (NumPort) << endl;
    Signal(SIGCHLD,TraiterZombies);Signal(SIGTERM,Quitter);
    while(1) {
        int socketCommande;
//_// ECOUTE avec Accept() pour obtenir socketCommande _____________________
        for(bool qRetryListening (true); qRetryListening;) {
            try {
                socketCommande = Accept (socketEcoute);
            }
            catch (CException & e) {
                if (e . GetCodErr() != EINTR) throw;
                continue;
            }
            qRetryListening = false;
        }
////////////////////////////////////////////////////////////////
// (juste l'appel socketCommande = Accept (socketEcoute); compte aussi
////////////////////////////////////////////////////////////////
        if(Fork()) {
//_// FERMETURE socket de commandes ___________________________________
            Close(socketCommande);
////////////////////////////////////////////////////////////////
            continue;
        }
//_// FERMETURE socket Ecoute (donc on est dans le fils)__________
        Close(socketEcoute);
////////////////////////////////////////////////////////////////
        for(bool qValidUser = false; !qValidUser;) {
            const string loginTxt("login: \n");
            const string passwdTxt("Password: \n");
            char userName[USERNAMELENGTH+1], passwd[PASSWDLENGTH+1];
//_// DIALOGUE D'AUTHENTIFICATION SUR socketCommande (PAS DE stdin/stdout!!!)
```

```
//_// POUR RECUPERER userName ET passwd ____________________________________
            Write(socketCommande, loginTxt . c_str(), loginTxt . size());
            LireN(socketCommande, userName,  USERNAMELENGTH);
            Write(socketCommande, passwdTxt . c_str(), passwdTxt . size());
            LireN(socketCommande, passwd,    PASSWDLENGTH);
////////////////////////////////////////////////////////////////
            qValidUser = validerUser(userName,passwd);
            if(!qValidUser) {
//_// ECRITURE du message de invldUsrTxt sur socketCommand _____________
                Write(socketCommande, invldUsrTxt . c_str(), invldUsrTxt . size());
////////////////////////////////////////////////////////////////
            }
        }
//_// ECRITURE du greetingTxt sur socketCommande ____________________
        Write(socketCommande, (greetingTxt + "\n") . c_str(), greetingTxt . size() + 1);
////////////////////////////////////////////////////////////////
        uint16_t dataPort(0);
//_// OBTENTION socketDataEcoute AVEC ServerInit()_____________________________
//_// TOUT EN REMPLISSANT dataPort AUSSI_____________________________________
        const int socketDataEcoute(ServerInit(dataPort));
////////////////////////////////////////////////////////////////
        ostringstream buff;
//_// LA BONNE VALEUR DE dataPort ____________________________________________
        buff << ntohs(dataPort);
////////////////////////////////////////////////////////////////
        const string socketDataEcouteMsgTxt(dataPortTxt + buff . str() + "\n");
//_// ECRITURE DE socketDataEcouteMsgTxt SUR socketCommande _____________________
        Write(socketCommande, socketDataEcouteMsgTxt . c_str(),
                socketDataEcouteMsgTxt . size());
////////////////////////////////////////////////////////////////
        // fin de l'initialisation de la connexion
```

3.1.3 Traitement commandes serveur

```
        for(bool qSession (true); qSession;) {
            // boucle traitant chaque commande, selon ces etapes
            //   1. lecture commande
            //   2. analyse commande
            //   3. (dedoublement et) traitement
            // Etape 1: Lecture commande
            char receivedCmdArg[MAXMSGLENGTH+1];
            LireN(socketCommande, receivedCmdArg, MAXMSGLENGTH);
            // Etape 2: Analyse commande
            string receivedArgTxt;
            Command commande(analyserCmd(receivedCmdArg,&receivedArgTxt));
            // Etape 3a: Traitement cas "simples"
            if(commande == CMD_UNKNOWN) {
                Write(socketCommande, cmdUnknTxt . c_str(),
                      cmdUnknTxt . size());
                Write(socketCommande, receivedCmdArg, CMDLENGTH);
                Write(socketCommande, "\n", 1);
                continue;
            }
```

```
if(commande == CMD_BYE) {
    Write(socketCommande, (byebyeTxt + "\n") . c_str(),
          byebyeTxt . size() + 1);
    return;
}
Write(socketCommande, (processingTxt + "\n") . c_str(),
      processingTxt . size() + 1);
// Etape 3b: Traitement cas "complexes"
if(commande != CMD_CD) {
    if(Fork()) {
        Wait();
        continue;
    }
}
int sdt (-1);
for(bool qReTryAccept (true);qReTryAccept; ) {
    try {
        sdt = Accept (socketDataEcoute);
        qReTryAccept = false;
    }
    catch (CException & e) {
        if (e . GetCodErr() != EINTR) throw;
    }
}
const int socketData(sdt);
try {
    switch(commande) {
        case CMD_CD:
            Dup2(socketData, STDOUT_FILENO);
            try {
                ChDir(receivedArgTxt . c_str());
                cout << "Changed to " << receivedArgTxt << "\n";
            }
            catch(CException &exc) {
                cout << "ERROR Could not cd to "
                     << receivedArgTxt << " "
                     << strerror(exc . GetCodErr())
                     << "\n";
            }
            catch(...) {
                cout << "FATAL Could not cd to "
                     << receivedArgTxt << " "
                     << strerror(errno)
                     << "\n";
            }
            Close(socketData);
            if(socketData != STDOUT_FILENO) Close(STDOUT_FILENO);
            break;
        case CMD_LS:
            Dup2(socketData, STDOUT_FILENO);
            execl("/bin/ls","ls","-l",0);
            throw CExcFctSyst ("execl()");
            break;
        case CMD_GET:
```

```
                    Dup2(socketData, STDOUT_FILENO);
                    execl("/bin/cat","cat",receivedArgTxt . c_str(),0);
                    throw CExcFctSyst ("execl()");
                    break;
                case CMD_PUT: {
                    const int outFile (Open(receivedArgTxt . c_str(),
                                            O_RDWR | O_CREAT, 0600));
                    Dup2(socketData, STDIN_FILENO);
                    Dup2(outFile,    STDOUT_FILENO);
                    execl("/bin/cat","cat",0);
                    throw CExcFctSyst ("execl()");
                    break;
                }
                default:
                    cerr << "FATAL ERROR: Should never get here.\n";
                    exit(1);
            } // switch
        } // try
        catch(CException & exc) {
            expliquerErreur(exc, socketData);
        }
        catch(...) {
            const string cmdExcErrTxt("Unknown error. ");
            Write(socketData, cmdExcErrTxt . c_str(),
                  cmdExcErrTxt . size());
        }
        if(commande != CMD_CD) return; // fin du fils traitant la demande
    } // fin de la boucle de traitement de chaque commande
} // fin de la boucle infinie du p&egrave;re
}
```