Write up
Comments
Code

Operating Systems Homework 5

Gabriel Groover, gtg3vv@virginia.edu

I was able to successfully implement a barebones FTP server in accordance with the specifications.

1. Problem:

The goal for this assignment was to implement a barebones file transfer protocol. Our solution should utilize Berkeley sockets and remote procedural calls to establish a connection and transfer data between the client and server. The server connects with a single client at a time and does not need to support multiple connections. It will support only a few basic server commands to store and retrieve data in the chosen format.

2. Approach

The solution was coded in C++ and made use of two Berkeley sockets, and string manipulation functions designed in the shell homework. In order to establish a connection endpoint, a socket is created and initialized with the correct domain, type, and protocol. The socket will essentially just be a file descriptor that can be read from or written to. The process for creating this socket is taken directly from the Berkeley sockets Wikipedia page shown in class. A struct is used to hold the socket address. The port for this address is taken from the command line argument passed into the main function. The

address for this socket is set to the localhost address, 127.0.0.1, to prevent external connections that could create security vulnerabilities.

Now that a working socket is created, bind is used to associate the socket with the address struct created above. Next, listen is called on the socket to cause it to listen for attempted connections on the specified port. At this point, an infinite loop was used to accept incoming connections. Each iteration of the loop will call accept at the top. This will block until an incoming connection is received from the client, at which point accept will return a new socket to be used for the connection to the client.

A second infinite loop was used within the first to continually read input from the client via recv(). All of the information the client sends over this socket will be commands of the form COMMAND [param 1]…[param n]. Each iteration of this loop checks to see if a new command has been received, and if so, tokenizes that line of input into a vector of strings. At this point the rest of the loop is just a large switch statement based on the commands below. Each part of the switch statement processes the specific command and then sends the correct status code to the client based on the result of that command. The current implementation of the server handles the following commands:

USER [name] – This is used to authenticate users. In this insecure implementation the server returns 230 (logged in successfully) if the correct number of parameters is given.
QUIT – This returns the status code 221 (connection closed) and breaks out of the inner loop so that the server may again start block on accept until a new client attempts to connect.

TYPE [char 1] [char 2] – This command specifies the type of data to be transferred. This implementation only handles the binary image format, and returns an error code for any other image character, or more than one parameter.

STRU/MODE [char] – These specify the structure and transfer mode of the data. This implementation only supports the file structure in stream mode. Other inputs report an error.

PORT [a1,a2,a3,a4,p1,p2] – All data transfers between client and server happen over second socket that is established when the stor/list/retr operation is called. Port initializes the address values for that socket according to the client-specified address and port.

RETR/STOR/LIST [filename] – These three commands require a second data-only socket to be created to send information. Each command first checks that the server is in binary mode before initiating the transfer. Each command executes a system process (cat, ls) and uses dup2 to send the output of that process to the correct location. List/retr sends its output to the file descriptor of the data-only socket and stor sends its output to a new server-side file. The socket is closed when the transfer is complete and all future transfer commands will have to set up another connection using port.

The NOOP command will simply return a successful status code and continue. Any other commands will just incur a 502 (command not implemented) signal. The server will continue to check for input from the client until it receives a quit command. At this point the server will close the connection and wait for a new client.

3. Problems Encountered

Most of the problems encountered in this assignment stemmed from the complexity of the specification. The specification is extremely long and detailed, and it was difficult to extract the pieces of information that were relevant to this barebones implementation. The sections that listed the return codes and state diagrams for each command turned out to be the most useful.

The return codes were one of the biggest problems encountered during development. The built-in ftp client expects very specific return codes in response to each of its commands. The client would hang and it was not clear that the response codes were the reason. This was fixed by implementing all the correct codes from the specification. It also helped to use tcpflow to monitor network traffic and see all the commands the client was sending to the server. The client sometimes abstracts multiple server-side calls into one client-side command. Tcpflow helped to reveal which commands caused these cases so that each command could be implemented properly.

4. Testing

As with previous assignments, this ftp implementation was tested incrementally. First, a basic client connection was established and checked using a simple telnet client. This allowed simple data transfer and string parsing to be tested without requiring correct response codes to be implemented.

Each command was also tested for its most basic functionality before moving on to more unique cases. For example, store and retrieve were tested with simple, absolute

path, small file transfers. Eventually these commands were tested with relative paths, nonexistent files, and very large files, to make sure all conditions were handled correctly. The specifications were particularly useful here in determining the behavior of edge cases such as storing new data in a remote file that already exists.

5. Conclusion

This assignment helped to reinforce an understanding of Berkeley sockets, as well as demonstrating their usefulness with a practical example. As with previous assignments, it was very beneficial to plan development early, and test iteratively. The FTP server was a fitting final assignment as it incorporated many ideas from earlier assignments like file manipulation, piping, string manipulation and forking. This implementation, while functional, leaves a lot of room for future work such as proper user authentication or directory navigation and creation.

I pledge that I have not given or received aid on this assignment or report.

```
/*
 * Gabriel  Groover (gtg3vv)
 * HW5 - Simple FTP Server
 * Due: 5/1/2018
 * ftp_server.cpp
 * A simple ftp server that is capable of connecting to a single ftp client at a time.
 * It implements the barebones operations of an ftp server.
 * Code for the server and client connections was taken from the wikipedia page shown in class.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <iostream>
#include <string.h>
#include <vector>
#include <string>
#include <sys/wait.h>
#include <fcntl.h>

using namespace std;

//Split string into vector based on delim
vector<string> tokenize(string s, const char* delim)
{
   vector<string> tokens;
   size_t start = 0;
   size_t loc = s.find(delim);

   //While delim in string
   while (loc != string::npos)
   {
      //Add string from last delim to current to vector
      tokens.push_back(s.substr(start, loc-start));
      start = loc+strlen(delim);

      loc = s.find(delim, loc+1);
   }
   //Push remainder
   tokens.push_back(s.substr(start, s.length()));
   return tokens;
}

//Remove leading/trailing/doubled spaces
string removeSpace(string s)
{
```

```
    string oneSpace = "";
    bool afterSpace = true;
    size_t lastLetter = s.find_last_not_of(" \r\n");

    for (int i = 0; i < s.length(); i++)
    {
        //If repeated space, skip
        if (s[i] == ' ' && afterSpace)
            continue;

        afterSpace = s[i] == ' ';
        if (i <= lastLetter) oneSpace += s[i];
    }
    return oneSpace;
}

//Create and return a socket file descriptor for use in data connections
int setUpSocket(struct sockaddr* ca, int size)
{
    int dataSocketFD;
    dataSocketFD = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (dataSocketFD == -1) {
        perror("cannot create socket");
        exit(EXIT_FAILURE);
    }

    if (connect(dataSocketFD, ca, size) == -1) {
        perror("connect failed");
        close(dataSocketFD);
        exit(EXIT_FAILURE);
    }
    return dataSocketFD;
}

//Main server functionality
int main(int argc, char **argv)
{
    //Socket variables
    struct sockaddr_in sa,ca;
    int ca_size = sizeof(ca);
    int SocketFD = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    char* buffer = (char*) malloc(100);
    bool binary = false;

    //Attempt to create the socket
    if (SocketFD == -1) {
      perror("cannot create socket");
      exit(EXIT_FAILURE);
    }

    //Initialize connection types
```

```
memset(&sa, 0, sizeof(sa));
sa.sin_family = AF_INET;
sa.sin_port = htons(atoi(argv[1]));
sa.sin_addr.s_addr = inet_addr("127.0.0.1");
memset(&ca, 0, sizeof ca);
ca.sin_family = AF_INET;
ca.sin_port = htons(atoi(argv[1]));

//Attempt to bind socket
if (bind(SocketFD,(struct sockaddr *)&sa, sizeof sa) == -1) {
  perror("bind failed");
  close(SocketFD);
  exit(EXIT_FAILURE);
}

//Listen for connections on socket
if (listen(SocketFD, 10) == -1) {
  perror("listen failed");
  close(SocketFD);
  exit(EXIT_FAILURE);
}

//Loop over connections
for (;;) {

    int ConnectFD = accept(SocketFD, (struct sockaddr*)&ca, (socklen_t*)&ca_size);

    //Check if connection successful
    if (0 > ConnectFD) {
        perror("accept failed");
        close(SocketFD);
        exit(EXIT_FAILURE);
    }
    send(ConnectFD, "220\r\n", 5, 0);

    //Loop over input received
    while (1)
    {
        memset(buffer, 0, 100);
        int  numBytes = recv(ConnectFD, buffer, 100, 0);

        //If bytes received
        if (numBytes > 0)
        {
            //Tokenize command
            string command = removeSpace(string(buffer));
            vector<string> tokens = tokenize(command, " ");

            //Handle port command
            if (tokens[0].compare("PORT") == 0)
            {
```

```
    vector<string> byteVals = tokenize(tokens[1], ",");
    string dec_addr = byteVals[0] + "." +
                      byteVals[1] + "." +
                      byteVals[2] + "." +
                      byteVals[3];


    memset(&ca, 0, sizeof ca);
    ca.sin_family = AF_INET;
    ca.sin_port = htons(atoi(byteVals[4].c_str())*256 + atoi(byteVals[5].c_str()));
    int res = inet_pton(AF_INET, dec_addr.c_str(), &ca.sin_addr);

    send(ConnectFD, "200\r\n", 5, 0);
}
//Handle user command
else if (tokens[0].compare("USER") == 0)
{
    if (tokens.size() == 2)
        send(ConnectFD, "230\r\n", 5, 0);
    else
        send(ConnectFD, "501\r\n", 5, 0);

}
//Handle Quit
else if (tokens[0].compare("QUIT") == 0)
{
    send(ConnectFD, "221\r\n", 5, 0);
    binary = false;
    break;

}
//Handle type
else if (tokens[0].compare("TYPE") == 0)
{
    if (tokens[1].compare("I") != 0 or tokens.size() > 2)
        send(ConnectFD, "504\r\n", 5, 0);
    else
    {
        binary = true;
        send(ConnectFD, "200\r\n", 5,0);
    }

}
//Handle mode
else if (tokens[0].compare("MODE") == 0)
{
    if (tokens[1].compare("S") != 0 or tokens.size() != 2)
    {
        send(ConnectFD, "504\r\n", 5, 0);
        continue;
    }
```

```
   send(ConnectFD, "200\r\n", 5,0);
}
//Handle STRU
else if (tokens[0].compare("STRU") == 0)
{
   if (tokens[1].compare("F") != 0 or tokens.size() != 2)
   {
      send(ConnectFD, "504\r\n", 5, 0);
      continue;
   }
   send(ConnectFD, "200\r\n", 5,0);

}
//Handle RETR
else if (tokens[0].compare("RETR") == 0)
{
   if (!binary)
   {
      send(ConnectFD, "451\r\n", 5, 0);
      continue;
   }

   if (open(tokens[1].c_str(), O_RDONLY) < 0)
   {
      send(ConnectFD, "550\r\n", 5, 0);
      continue;
   }

   int dataSocketFD = setUpSocket((struct sockaddr *)&ca, sizeof ca);
   send(ConnectFD, "150\r\n", 5, 0);

   int pid = fork();
   if (pid == 0)
   {
      dup2(dataSocketFD, 1);
      if (tokens.size() > 1)
         execl("/bin/cat", "/bin/cat", tokens[1].c_str(), (char*) NULL);
      else
         execl("/bin/cat", "/bin/cat", (char*) NULL);
   } else
   {
      waitpid(pid, NULL, 0);
      close(dataSocketFD);
      send(ConnectFD, "250\r\n", 5, 0);
   }
}
//Handle STOR
else if (tokens[0].compare("STOR") == 0)
{
   if (!binary)
   {
```

```
            send(ConnectFD, "451\r\n", 5, 0);
            continue;
        }
        int dataSocketFD = setUpSocket((struct sockaddr *)&ca, sizeof ca);
        send(ConnectFD, "150\r\n", 5, 0);
        remove(tokens[1].c_str());
        int outfile = open(tokens[1].c_str(), O_RDWR | O_CREAT, 0666);

        int pid = fork();
        if (pid == 0)
        {
            dup2(dataSocketFD, 0);
            dup2(outfile, 1);
            execl("/bin/cat", "/bin/cat", (char*) NULL);
        } else
        {
            waitpid(pid, NULL, 0);
            close(dataSocketFD);
            send(ConnectFD, "250\r\n", 5, 0);
        }
    }
    //Handle no-op
    else if (tokens[0].compare("NOOP") == 0)
    {
        send(ConnectFD, "200\r\n", 5, 0);

    }
    //Handle List
    else if (tokens[0].compare("LIST") == 0)
    {
        if (!binary)
        {
            send(ConnectFD, "451\r\n", 5, 0);
            continue;
        }
        int dataSocketFD = setUpSocket((struct sockaddr *)&ca, sizeof ca);
        send(ConnectFD, "150\r\n", 5, 0);

        int pid = fork();
        if (pid == 0)
        {
            dup2(dataSocketFD, 1);
            if (tokens.size() > 1)
                execl("/bin/ls", "/bin/ls", "-l", tokens[1].c_str(), (char*) NULL);
            else
                execl("/bin/ls", "/bin/ls", "-l", (char*) NULL);

        } else
        {
            int status;
            waitpid(pid, &status, 0);
```

```
                if (status > 0)
                    send(ConnectFD, "450\r\n", 5, 0);
                else
                    send(ConnectFD, "250\r\n", 5, 0);
                close(dataSocketFD);

            }

        }
        //Respond to unknown command
        else {
            send(ConnectFD, "502\r\n", 5, 0);
        }
      }
    }

    //Close connection
    if (shutdown(ConnectFD, SHUT_RDWR) == -1) {
        perror("shutdown failed");
        close(ConnectFD);
        close(SocketFD);
        exit(EXIT_FAILURE);
    }
    close(ConnectFD);
  }

  close(SocketFD);
  return EXIT_SUCCESS;
}
```