Write up
Comments
Code

____

Operating Systems Homework #5

Maurice Wong, mzw7af@virginia.edu

The assignment was successfully completed.

1. Problem:

The goal of this assignment was to create a barebones FTP server that can properly respond to an FTP client. The server must be able to implement the minimum required commands as outlined by the FTP specification document, which include basic operations for modifying settings and file transfers in both directions. The required FTP commands that needed to be implemented were:

- *USER* – logs in the specified user
- *QUIT* – logs out the user and closes the control connection
- *PORT* – specifies the address and port that subsequent data transfer operations will connect to
- *TYPE* – modifies the data type setting. Only "Image" is supported for this assignment
- *MODE* – modifies the method of data transfer. Only the default mode, "Stream" is supported for this assignment.
- *STRU* – switches the structure types setting. Only the default mode, "File" is supported for this assignment.
- *RETR* – retrieves the specified file and downloads a local copy of it
- *STOR* – puts the specified local file at the specified remote location on the server.
- *NOOP* – no operation, simply sends back confirmation that the connection is valid
- *LIST* – lists the files at the remote server, running a remote "*ls –l*". Although not a required command in FTP, this was required for the assignment.

2. Approach:

This assignment required use of the Berkeley Socket API, which was instrumental in establishing the socket connections for communications. Additionally, the FTP specification document was extensively referenced to create a proper FTP server. Using this API, 3 primary functions were implemented to complete this assignment:

- *main()*
- *openSocket()*
- *performOps()*

The *main()* function runs endlessly, waiting to open a new control connection with users. This function blocks, waiting for a client to connect with the server at the specified port, which is read in as a command line argument to the server program. Once a connection establishes, the program calls *performOps()* to service the client commands.

The *openSocket()* function is able to establish a socket connection. It initially opens a socket for use in the connection. Depending on the arguments received, it can either wait for the client to connect to the server, or it can actively attempt to connect to a client socket. This function uses the socket API library to create, bind, and listen or connect to the specified sockets.

Finally, the heart of the assignment is contained in *performOps()* which runs a continuous loop that listens for commands from a connected client, returns appropriate responses, and performs the requested operations. This function is implemented as a large switch statement that parses incoming commands and their arguments, matching the appropriate command and performing the corresponding operation.

Commands USER, QUIT, NOOP, TYPE, MODE, STRU required minimal logic, since they primarily just returned confirmation codes that the appropriate actions were performed. For the commands that change settings, no logic was actually required since only one setting is valid for each one in this assignment.

The PORT command required careful parsing of the arguments into the IP address and the port to establish the next data socket connection. These arguments were parsed, properly formatted, and stored in variables for use in future commands.

Commands STOR, RETR, LIST required opening data socket connections to the client and transferring data between the server and client. The address and port received from the previously run PORT command were referenced and used to establish a socket connection using the *openSocket()* function. For the STOR and RETR commands, the data socket file descriptor could be treated as any file descriptor, so it was simple to perform the corresponding read/write operations to it and transfer data through the socket connection. In both commands, data was read from either the socket or a local file, and then written to either a local file or socket. The LIST command was trickier since it required running a local "*ls –l* " command. This was accomplished with *popen()*, which essentially treats the output as a file. This output was then written to the data socket as in RETR.

3. Problems Encountered

The most significant problem encountered for this project was interpreting the FTP specification to understand what the appropriate operations were for each command. The descriptions for the commands were rather vague and didn't always specify the appropriate response codes to return for different scenarios. A lot of cross referencing between the status codes and the command descriptions were necessary to fully understand what operations should be performed.

Another major issue encountered was understanding how to interpret the PORT command and performing the file data transfer over the connection. It was unclear that the PORT command was indicating that the client would be listening on the given address and port, and the server had to perform a *connect()* operation to attach to the data socket. Originally, it was assumed that the given address and port meant that the server should listen for a client connection on that port, which is erroneous.

Once the data socket was established, it was unclear how to properly read/write information to the socket connection. However, research into socket examples showed that the socket file descriptors could be treated like any file descriptor, and it was significantly easier after this revelation to read/write data to the socket connections.

Another difficulty encountered was with the LIST command, which required to transfer the output from a shell command on the server to the data socket. Multiple approaches were attempted, including temporary files and manually forking and exec'ing an *ls* process. It was eventually discovered that the *popen()* function could contain the output of the shell command like a file, which was significantly easier to work with.

4. Testing

To test this server, an FTP client was installed and used to connect to the created FTP server. Multiple commands were run on the FTP client to invoke the proper FTP server commands and test each one. Additionally, the *quote* command on the FTP client was used to specifically invoke FTP server command that did not map directly to FTP client commands. Every FTP server command was invoked and the results inspected to ensure that the appropriate status codes were being returned. Specifically, files were transferred from both server to client and client to server to ensure that data connection transfers were successful.

5. Conclusion

This assignment was useful in learning about establishing communications through sockets and transferring data. Following the FTP specifications and understanding the Berkeley socket API were crucial in the success of this assignment. Since FTP communication requires proper sending and acknowledging of the correct FTP codes, it was important to thoroughly read the documentation and understand what was required in the implementation. Ultimately, this assignment resulted in a more thorough understanding of network communications and the server-client model.

```c
/**
 * Homework 5: Barebones FTP Server
 *
 * CS4414 Operating Systems
 * Fall 2017
 *
 * Maurice Wong - mzw7af
 *
 * main.c - FTP server program
 *
 * COMPILE:     make
 * OBJECTS:     main.o
 * RUN:         ./my_ftpd
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <ctype.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void performOps(int ConnectFD, int control_port);
void upperCaseString(char* str);
int openSocket(int port, char* address);

int main(int argc, char** argv) {
    int SocketFD = openSocket(atoi(argv[1]), "0");
    if (SocketFD < 0) {
        exit(EXIT_FAILURE);
    }

    for (;;) {
        int ConnectFD = accept(SocketFD, NULL, NULL);

        if (0 > ConnectFD) {
            perror("accept failed");
            close(SocketFD);
            exit(EXIT_FAILURE);
        }
        char* connection_received = "220 Service ready for new user.\r\n";
        send(ConnectFD, connection_received, strlen(connection_received), 0);
        performOps(ConnectFD, atoi(argv[1]));

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

    close(SocketFD);
    return EXIT_SUCCESS;
}


/**
 * Opens a socket to the specified port.
 * If address == "0", then trying to listen. Else connecting to ip address
 * Returns -1 if error creating socket,
 * -2 if error binding socket, -3 if error listening on socket.
 * -4 error connecting to socket
 * Returns the socket FD on success
 */
int openSocket(int port, char* address) {
    struct sockaddr_in sa;
    int SocketFD = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (SocketFD == -1) {
        return -1;
    }
    memset(&sa, 0, sizeof sa);
    sa.sin_family = AF_INET;
    sa.sin_port = htons(port);
    if (strcmp(address, "0") == 0) { // bind and listen
        sa.sin_addr.s_addr = htonl(INADDR_ANY);

        if (bind(SocketFD,(struct sockaddr *)&sa, sizeof sa) == -1) {
            close(SocketFD);
            return -2;
        }

        if (listen(SocketFD, 10) == -1) {
            close(SocketFD);
            return -3;
        }
    } else { // connect
        inet_pton(AF_INET, address, &sa.sin_addr);
        if (connect(SocketFD, (struct sockaddr *)&sa, sizeof sa) == -1) {
          perror("connect failed");
          close(SocketFD);
          return -4;
        }
    }
    return SocketFD;
}

/**
 * Loop to perform control operations in control connection
 */
void performOps(int ConnectFD, int control_port) {
    int full_command_len = 10000;
    char full_command[full_command_len];
    char* command;
    char* dataSocket_addr;
    int dataSocket_port;
    int binaryMode = 0;
    // listen for user commands
    for (;;) {
        memset(full_command, '\0', full_command_len);
        int recv_status = recv(ConnectFD, full_command, full_command_len, 0);
        if (recv_status <= 0) { // error receiving message or shutdown
            break;
        }
        command = strtok(full_command, " \n\r");
        upperCaseString(command);
        // SWITCH BASED ON WHAT COMMAND IS RECEIVED
        if (strcmp(command, "USER") == 0) { // USER
            char* undefined_command = "230 User logged in, proceed.\r\n";
            send(ConnectFD, undefined_command, strlen(undefined_command), 0);
        } else if (strcmp(command, "QUIT") == 0) { // QUIT
            char* quit_command = "221 Service closing control connection.\r\n";
            send(ConnectFD, quit_command, strlen(quit_command), 0);
            break;
        } else if (strcmp(command, "PORT") == 0) { // PORT
            int i = 0;
            char address[20];
            int address_len = 0;
            for (i = 0; i < 4; i++) { // ip address
                char* temp = strtok(NULL, ",");
                if (temp == NULL) {
                    char* port_failure = "500 PORT not received.\r\n";
                    send(ConnectFD, port_failure, strlen(port_failure), 0);
                }
                strcpy(address + address_len, temp);
                address_len += strlen(temp) + 1;
                address[address_len-1] = '.';
            }
            address[address_len-1] = '\0';
            int p1 = atoi(strtok(NULL, ","));
            int p2 = atoi(strtok(NULL, ","));
            int portNum = (p1 << 8) | p2; // combine p1 and p2 to form port number
            dataSocket_addr = address;
            dataSocket_port = portNum;
            char* port_success = "200 PORT established.\r\n";
            send(ConnectFD, port_success, strlen(port_success), 0);
        } else if (strcmp(command, "TYPE") == 0) { // TYPE
            char* type = strtok(NULL, " \n\r");
            if (type != NULL && strcmp(type, "I") == 0) {
                binaryMode = 1;
                char* type_image = "200 Switching to Image type.\r\n";
                send(ConnectFD, type_image, strlen(type_image), 0);
            } else {
                char* type_image = "504 Command not implemented for that parameter.
                 \r\n";
                send(ConnectFD, type_image, strlen(type_image), 0);
            }
        } else if (strcmp(command, "MODE") == 0) { // MODE
            char* type = strtok(NULL, " \n\r");
            if (type != NULL && strcmp(type, "S") == 0) {
                char* stream_mode = "200 Switching to Stream mode.\r\n";
                send(ConnectFD, stream_mode, strlen(stream_mode), 0);
            } else {
                char* stream_mode = "504 Command not implemented for that parameter.
                 \r\n";
                send(ConnectFD, stream_mode, strlen(stream_mode), 0);
            }
        } else if (strcmp(command, "STRU") == 0) { // STRU
            char* type = strtok(NULL, " \n\r");
            if (type != NULL && strcmp(type, "F") == 0) {
                char* file_mode = "200 Switching to File mode.\r\n";
                send(ConnectFD, file_mode, strlen(file_mode), 0);
            } else {
                char* file_mode = "504 Command not implemented for that parameter.
                 \r\n";
                send(ConnectFD, file_mode, strlen(file_mode), 0);
            }
        } else if (strcmp(command, "RETR") == 0) { // RETR
            if (!binaryMode) { // CHECK IF IN BINARY MODE
                char* non_binary = "451 Requested action aborted: local error in
                 processing\r\n";
                send(ConnectFD, non_binary, strlen(non_binary), 0);
                continue;
            }
            // OPEN DATA CONNECTION
            int dataSocketFD = openSocket(dataSocket_port, dataSocket_addr);
            if (dataSocketFD >= 0) {
                char* data_connection = "150 Data connection established.\r\n";
                send(ConnectFD, data_connection, strlen(data_connection), 0);
            } else {
                char* data_connection = "425 Can't open data connection.\r\n";
                send(ConnectFD, data_connection, strlen(data_connection), 0);
                shutdown(dataSocketFD, SHUT_RDWR);
                close(dataSocketFD);
                continue;
            }
            // READ FROM FILE AND WRITE TO DATA SOCKET
            char* pathname = strtok(NULL, " \n\r");
            if (pathname == NULL) {
                char* transfer_done = "501 pathname invalid.\r\n";
                send(ConnectFD, transfer_done, strlen(transfer_done), 0);
            }
            FILE *fp;
            fp = fopen(pathname, "r");
            if (fp == NULL) {
                char* transfer_done = "501 pathname invalid.\r\n";
                send(ConnectFD, transfer_done, strlen(transfer_done), 0);
            }
            char content[100000];
            while (fgets(content, 100000, fp) != NULL) {
                write(dataSocketFD, content, strlen(content));
            }
            fclose(fp);
            shutdown(dataSocketFD, SHUT_RDWR);
            close(dataSocketFD);
            char* transfer_done = "226 Transfer Complete.\r\n";
            send(ConnectFD, transfer_done, strlen(transfer_done), 0);
        } else if (strcmp(command, "STOR") == 0) { // STOR
            if (!binaryMode) { // CHECK FOR BINARY MODE
                char* non_binary = "451 Requested action aborted: local error in
                 processing\r\n";
                send(ConnectFD, non_binary, strlen(non_binary), 0);
                continue;
            }
            // OPEN DATA SOCKET CONNECTION
            int dataSocketFD = openSocket(dataSocket_port, dataSocket_addr);
            if (dataSocketFD >= 0) {
                char* data_connection = "150 Data connection established.\r\n";
                send(ConnectFD, data_connection, strlen(data_connection), 0);
            } else {
                char* data_connection = "425 Can't open data connection.\r\n";
                send(ConnectFD, data_connection, strlen(data_connection), 0);
                shutdown(dataSocketFD, SHUT_RDWR);
                close(dataSocketFD);
                continue;
            }
            char* pathname = strtok(NULL, " \n\r");
            if (pathname == NULL) {
                char* transfer_done = "501 pathname invalid.\r\n";
                send(ConnectFD, transfer_done, strlen(transfer_done), 0);
            }
            FILE *fp;
            fp = fopen(pathname, "w");
            if (fp == NULL) {
                char* transfer_done = "501 pathname invalid.\r\n";
                send(ConnectFD, transfer_done, strlen(transfer_done), 0);
            }
            // READ DATA FROM SOCKET AND WRITE TO LOCAL FILE
            char content[100000];
            int bytes_read;
            do {
                bytes_read = read(dataSocketFD, content, 10000);
                fwrite(content, 1, bytes_read, fp);
            } while(bytes_read != 0);
            fclose(fp);
            shutdown(dataSocketFD, SHUT_RDWR);
            close(dataSocketFD);
            char* transfer_done = "226 Transfer Complete.\r\n";
            send(ConnectFD, transfer_done, strlen(transfer_done), 0);
        } else if (strcmp(command, "NOOP") == 0) { // NOOP
            char* noop_command = "200 Command okay.\r\n";
            send(ConnectFD, noop_command, strlen(noop_command), 0);
        } else if (strcmp(command, "LIST") == 0) { // LIST
            char ls_command[1000];
            strcpy(ls_command, "/bin/ls -l ");
```

```
            char* pathname = strtok(NULL, " \n\r");
            if (pathname != NULL) {
                strcat(ls_command, pathname);
            }
            // OPEN DATA SOCKET CONNECTION
            int dataSocketFD = openSocket(dataSocket_port, dataSocket_addr);
            if (dataSocketFD >= 0) {
                char* data_connection = "150 Data connection established.\r\n";
                send(ConnectFD, data_connection, strlen(data_connection), 0);
            } else {
                char* data_connection = "425 Can't open data connection.\r\n";
                send(ConnectFD, data_connection, strlen(data_connection), 0);
                shutdown(dataSocketFD, SHUT_RDWR);
                close(dataSocketFD);
                continue;
            }
            FILE *fp;
            // RUN LS -L COMMAND AND WRITE TO DATA SOCKET
            fp = popen(ls_command, "r");
            char ls_content[10000];
            while (fgets(ls_content, 10000, fp) != NULL) {
                write(dataSocketFD, ls_content, strlen(ls_content));
            }
            int exit_status = WEXITSTATUS(pclose(fp));
            shutdown(dataSocketFD, SHUT_RDWR);
            close(dataSocketFD);
            if (exit_status == 0) {
                char* transfer_done = "226 Transfer Complete.\r\n";
                send(ConnectFD, transfer_done, strlen(transfer_done), 0);
            } else {
                char* transfer_done = "501 ls pathname invalid.\r\n";
                send(ConnectFD, transfer_done, strlen(transfer_done), 0);
            }

        } else { // unsupported commands
            char* undefined_command = "500 Syntax error, command unrecognized.\r\n";
            send(ConnectFD, undefined_command, strlen(undefined_command), 0);
        }

    }
}

/**
 * Turn a string into uppercase (used for interpreting comands)
 */
void upperCaseString(char* str) {
    while(*str) {
      *str = (toupper(*str));
      str++;
    }
}
```