Brian Team
dot4qu

CS 4414 Machine Problem 4 – Simple FTP Server

Pledge:

On my honor, I have neither given nor received help on this assignment.

Problem:

The goal of this assignment was to implement a barebones FTP server using Berkeley sockets and the POSIX API. By supporting the basic client actions of listing, storing, and retrieving, a deeper understanding of remote filesystems and remote procedure calls was gained. The server must operate in conjunction with the basic ftp client bundled with Linux software. By using resources such as the official FTP RFC specification and several examples on the internet of commands and their return codes, I was able to successfully complete this problem according to the requirements set forth in the assignment.

Approach:

Before beginning to implement any code, sections 4, 5, and 6 of the FTP RFC were studied to determine how to begin the assignment. These sections listed both the possible commands to send from client to server, and also the return codes for each command. It did not provide a detailed explanation for each return code, so the internet was useful in determining exactly what numbers to send back to the client.

The first actual step in implementing the server was to establish the local socket that clients could be used to connect to. The socket was created with a call to 'socket()', specifying an IPv4 type and a full duplex data stream. The next function call was to 'bind' to bind the newly created socket to a sockaddr_in address struct, populated with information describing the port, IP address, and IP protocol to use. The main function then entered its outer while-loop, which was responsible for constantly waiting for new connections from clients whenever its previous connection disconnected. It used a call to 'listen()' to wait for incoming connections, and then a call to 'accept()' to create a new file descriptor for the connected client. The first response code that needed to be sent was "230", telling the client that it had accepted it as a user and was

waiting for commands. Every code being sent by the server was pushed through the ‘write()’ function.

Once a steady client connection had been established, the main function entered its inner while-loop. This loop was responsible for receiving each message from the client, parsing it, and acting upon it. The incoming command was pulled in through the simple ‘read()’ function. While most FTP commands are 4 characters long, there are some that are 3. To handle this, a simple ‘parse_input’ function was written. It took a char buffer filled with the entire command from the client and returned the given command as a single string.

The last function call of the inner while-loop was to the ‘handle_cmd’ function. This was the brains of the entire program, and it handled all of the logic and dispatching of return codes. Besides variable declarations at the start, the entire function was one large if-else statement checking against the command string passed in.

The first command handled was TYPE. This command is sent when the user inputs either “binary” or “ascii” to change the data transfer mode. The actual functionality of this was handled by their FTP client itself, however there were small changes on the server end that needed to be handled to conform to the assignment specification. If the user attempted to initiate a data transfer while still in ASCII mode, the server was to return a “451” error code. This was handled through a simple global variable that was set on each execution of the TYPE command to whatever mode was specified.

The next command implemented was the PORT command. This is an extremely important one, since it is sent by the client every time they enter a command that involves any sort of data transfer. The client first sends this PORT command in the format of “PORT xxx,xxx,xxx,xxx,xxx,xxx”. The first 4 sets of x’s correspond to an IP address while the last two correspond to a port number. This is the IP and port that the client is telling the server to open up a new socket connection on for the data transfer, since it needs a separate connection from the one that it is sending and received commands and codes on. A helper function called ‘parse_ip_and_port()’ was written to pull apart this full string and return separate IP and port strings in the correct format. It worked through a simple iterating loop, substringing when needed based on the number of commas already seen. Once back in the ‘handle_cmd()’ function, the IP and data objects were translated to network objects through both a call to the ‘inet_addr()’ function and simple shifting and arithmetic respectively. Finally, the ‘open_data_socket()’

function was called with the port and IP as parameters. This function was responsible for creating a new socket through the same 'socket()' call previously described. Before the client actually connected to it however, it sent a "200" command to the client, signifying that the port creation was okay. It also sent a "150", signifying that it was about to open the new socket. The socket_in address struct was built with the port and IP, but this time the system call was to 'connect()' rather than 'bind()', since it was connecting to an external port.

Since PORT was sent for every type of data transfer, there was always a second command that immediately followed it. According to the assignment, the only three scenarios that we needed to handle were LIST, RETR, and STOR, for directory listing, getting a file from the server, and storing one, respectively.

The LIST command could come either by itself or with a specific directory. A cwd variable was maintained to be used for printing the root directory. The bulk of the work done in the LIST if-else portion of the 'handle_cmd()' function was done by the 'read_dir_files()' function. This accepted the directory string (if there was one), and returned a parsed and formatted string of each file or folder in the directory. The 'read_dir_files()' function used the 'fork()' command to spawn two different processes for both of the commands it needed to execute. A detailed explanation of pipes and forking processes can be found in the writeup for Machine Problem 1, the simple shell. The first command executed was "ls –l" with the given directory appended onto the end. This output was piped into the second command's stdin. The second command executed was "awk NR > 1 {print $9, \"\\t\", $5}". This command takes its input, delimits each line into columns based on spaces, and then outputs a new string according to the format received in between the curly braces. This specific format specifies that it should print the ninth column, a tab, then the fifth column. These two columns within the output of the "ls –l" command hold the filename and file size. This ensured that the output matched the assignment specification of "[filename] [tab] [file size]". The output of the "awk" command was piped back into the parent process' input. After waiting for each command's process to finish executing, the parent function read the output from the "awk" process' command and then returned. Back in 'handle_cmd()', the string was formatted to insert CRLFs instead of LFs and then written to the data socket's file descriptor. After a "226" command was sent to signify the end of data transfer, the data socket was closed and the LIST functionality was complete.

The final two commands, STOR and RETR, were extremely similar in their implementation. After handling the PORT command, STOR or RETR was read, accompanied by a directory on the server to either receive the file into or send from.

For STOR, a local file was created with “creat()’, and data was read into a buffer continuously until ‘read()’ function returned zero, signifying no more bytes. This data was then written to the created file with ‘write()’. Another “226” was sent back to the client, the data socket was closed, and then ‘handle_cmd()’ returned.

For RETR, the server file was opened with ‘open()’ and its file size was saved through the ‘stat()’ function. The file data was read into a buffer with ‘read()’, then written to the data socket through ‘write()’. The server sent a “226”, closed the socket, and then returned.

There were several other basic commands that were implemented since the assignment specified that the minimum functionality specified in the FTP RFC must be adhered to. These commands were STRU, USER, QUIT, and NOOP. STRU switched between the File and Record data structures. The assignment specified that only File needed to be supported, so a “504” response told the user that the command was not implement for the “R” parameter. USER simply accepted any username, prompted for a password with a return code of “331” and then accepted any password by responding with “230” confirming the user login. The NOOP was the most basic of them all, simply sending “200” and doing nothing. QUIT was the same, sending a “221” and then closing the socket connection (but remaining running to listen for new connections).

Results:

Upon completion of the program, it could successfully perform all functions listed as necessary by the assignment specification. The TYPE function could switch between ASCII or binary transfer modes. The QUIT function would close the current socket and resume waiting for a new connection. The STRU function would allow the user to set File structure mode but deny Record structure mode. PORT would allow a new data connection to be opened to a client port for transfer of bytes. RETR would allow the client to download a file from the server onto their local machine, and STOR would allow the client to upload a file a file onto the server from their local machine. All commands were tested for multiple common and edge cases such as a long directory listing or the getting/putting of both large and small files. It was given that no error checking in terms of input was necessary, so this functionality was omitted.

Analysis:

There was not a large amount of runtime analysis necessary for this assignment since the only loops that were executing were the outer and inner while-loops in the main function and several small loops in helper functions to iterate over strings. The outer while-loop would only execute once per socket connection, so that was trivial in terms of iterations. The inner while-loop would execute either once or twice per user-input command, so technically it's O(n) where n is the number of commands the user runs.

In terms of space complexity, there were several buffers allocated for the program. The first was the buffer allocated to hold all incoming commands from the client. This was created with a size of 256 bytes under the assumption that no user command could be nearly long and so there was no worry of truncation. This was only allocated once and reused every command. There was a buffer created to hold the directory that accompanied the LIST, STOR, and RETR commands. This was allocated with a size of MAX_PATH, the macro that holds the maximum path length on the Linux machine. By far the two buffers that had the potential to grow the largest were the two buffers allocated for either sending or receiving a file. When sending a file, the file size could be discovered through the opening of the local file, and the buffer was allocated to this size. Thus, however large the file is on the server is how many bytes were allocated for the buffer. When receiving a file, no way to determine the size of the incoming file could be found. Therefore, assuming testing would be done with files no larger than 8MB, the buffer was allocated to 8,388,608 bytes. All dynamically allocated memory was released before each function's exit.

Conclusion:

The purpose of this assignment was to teach the understanding of remote filesystems and basic Berkeley socket implementation. By requiring only a barebones FTP server an understanding of FTP commands, return codes, and system calls could be gained without dealing with many tedious commands. It was also helpful to learn how to read through an RFC specification. Using the 'fork()' and 'exec()' for the LIST functionality helped refresh the workings of multi-process execution.

my_ftpd.h

```
/*
Written by Brian Team (dot4qu)
Date: 11/29/16
This header file is responsible for listing the macros, structs, and
functions of its source file
*/

#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <algorithm>
#include <unistd.h>
#include <linux/limits.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>

using namespace std;

#ifndef MY_FTPD_H
#define MY_FTPD_H

#define BACKLOG_MAX 50
#define CHECK_ERROR(err) { \
                                            if (err < 0) { \
                                                 printf("Error on line
%d in function %s!", __LINE__, __func__); \
                                                 perror("!: "); \
                                                 exit(-1); \
                                            } \
                                       }



const string command_okay = "200 Command okay.\r\n";
const int command_okay_size = command_okay.length();
const string password_response = "230 User logged in.\r\n";
const int password_response_size = password_response.length();
const string stru_failed_response = "504 Command not implemented for
that parameter\r\n";
const int stru_failed_response_size = stru_failed_response.length();
```

```
const string stor_retr_failed_response = "451 Requested action
aborted: local error in processing\r\n";
const int stor_retr_failed_response_size =
stor_retr_failed_response.length();
const string open_data_response = "150 File status okay; about to open
data connection.\r\n";
const int open_data_response_size = open_data_response.length();
const string already_open_ascii_response = "125 Data Connection
already open; transfer starting.\r\n";
const int already_open_ascii_response_size =
already_open_ascii_response.length();
const string close_ascii_response = "226 Listing complete, closing
connection.\r\n";
const int close_ascii_response_size = close_ascii_response.length();
const string quit_response = "221 Goodbye.\r\n";
const int quit_response_size = quit_response.length();
const string data_finished_response = "226 Transfer complete, closing
data connection.\r\n";
const int data_finished_response_size =
data_finished_response.length();

string read_dir_files(string dir);                          /* takes
in directory string, executes system call for ls on directory and
parses return data to correct fmt */
int open_data_socket(in_addr_t ip, int port);          /* generic
function to open a socket given IP and port */
string parse_input(char* input);                            /* takes
in client command strings and parses, returning just command */
int handle_cmd(string cmd, char* input);               /* takesin cmd
string and remaining input buf to perform necessary actions */
void parse_ip_and_port(string param, string *ip, int *port1, int
*port2);   /* takes in param string and pulls part to save as IP and
port seperately */
#endif
```

my_ftpd.cpp

```
/*
Written by Brian Team (dot4qu)
Date: 11/29/16
This source file is responsible for implementing a simple FTP server
*/

#include "my_ftpd.h"

//GLOBALS
int client_cmd_fd;
int client_data_fd;
char type;
string cwd;
char cwd_buf[PATH_MAX];

string read_dir_files(string dir) {
     pid_t pids[2];                          /* hold pidsof forked
process */
     pid_t current_pid;                      /* holds pid of just forked
process */
     char output[16384];                     /* big ass buffer in case
listing a huge dir */
     int ls_pipe[2], awk_pipe[2];     /* to pipe output of ls back to
awk and awk to parent*/
     int err;                                /* temp var to hold retvals
*/
     string cmd_str;                         /* used to build up full
command w/ ls, dir, and awk */
     char const ls_cmd[] = "/bin/ls\0";
     char** ls_args;
     char const awk_cmd[] = "/usr/bin/awk\0";
     char **awk_args;

     //creating in/out pipes
     err = pipe(ls_pipe); CHECK_ERROR(err);
     err = pipe(awk_pipe); CHECK_ERROR(err);

     //build ls args,  /bin/ls -l dir
     ls_args = (char **) malloc( sizeof(char*) * 4 );
     //malloc and copy /bin/ls as first arg
     ls_args[0] = (char *) malloc( sizeof (ls_cmd) );
     strncpy(ls_args[0], ls_cmd, sizeof(ls_cmd));
     //malloc 3 bytes and copy two chars plus null
     ls_args[1] = (char *) malloc( sizeof("-l") + 1);
     strncpy(ls_args[1], "-l\0", 3);
     //malloc length of dir and copy that many chars plus null
     ls_args[2] = (char *) malloc(dir.length() + 1);
     strncpy(ls_args[2], dir.c_str(), dir.length() + 1);
     ls_args[3] = NULL;

     awk_args = (char**) malloc(sizeof(char*) * 3);
```

```
      awk_args[0] = (char*) malloc(sizeof(awk_cmd));
      strncpy(awk_args[0], awk_cmd, sizeof(awk_cmd) + 1 );
      awk_args[1] = (char*) malloc( 256 );
                  strncpy(awk_args[1], "NR > 1 {print $9, \"\\t\",
$5}\0", 28 );    //hardcoded, sloppy but short on time
                  awk_args[2] = NULL;

      for (int i = 0; i < 2; i++) {
            //forking child process
            current_pid = fork();
            CHECK_ERROR(pids[i]);

            if (current_pid == 0) {
                  //were in the child
                  if (i == 0) {
                        //dont need to change stdin since were not using
it / ls is the first cmd
                        //change ls cmds stdout to middle pipe's write
                        err = dup2(ls_pipe[1], STDOUT_FILENO);
                        CHECK_ERROR(err);
                        err = close(ls_pipe[0]);// CHECK_ERROR(err);
                        //exec ls cmd
                        err = execv(ls_cmd, ls_args);
                  } else {
                        //change awk cmds stdin to middle pipe's read
                        err = dup2(ls_pipe[0], STDIN_FILENO);
                        CHECK_ERROR(err);
                        err = close(ls_pipe[1]);
                        //change awk cmds stdout to end pipe's stdin to
return back to main process
                        err = dup2(awk_pipe[1], STDOUT_FILENO);
                        CHECK_ERROR(err);
                        err = close(awk_pipe[0]); //CHECK_ERROR(err);
                        err = execv(awk_args[0], awk_args);
                        CHECK_ERROR(err);
                  }
            } else {
                  //we're in the parent
                  pids[i] = current_pid;
            }
      } //for int i < 2

      //close all pipes except what we want to read
      err = close(ls_pipe[0]);
      err = close(ls_pipe[1]);
      err = close(awk_pipe[1]);

      waitpid(pids[0], NULL, 0);
      waitpid(pids[1], NULL, 0);

      memset(output, 0, sizeof(output));
      int num_bytes = read(awk_pipe[0], output, sizeof(output));
```

```
	delete[] awk_args;
	delete[] ls_args;

	return string(output);
}

int open_data_socket(in_addr_t ip, int port) {
	int err;				/* temp err value for error
checking */

	//creates socket for IPv4 communication domain with reliable two
way byte stream, and no protocol is necessary to be specified
	int socketfd = socket(AF_INET, SOCK_STREAM, 0);
	CHECK_ERROR(socketfd);

	sockaddr_in addr_to_open;
	//sockaddr_in client_addr;

	//sockaddr_in struct fields and descriptions can be found here
	//http://www.informit.com/articles/article.aspx?p=169505&seqNum=2
	memset(&addr_to_open, 0, sizeof(sockaddr_in));
	addr_to_open.sin_family = AF_INET;			//IPv4
family
	addr_to_open.sin_port = htons(port);
	//translate port from host to network byte order
	addr_to_open.sin_addr.s_addr = ip;			//accept
incoming conns specific IP

	//write out command okay for acknowledgement that we recieved
PORT cmd
	err = write(client_cmd_fd, command_okay.c_str(),
command_okay_size);
	CHECK_ERROR(err);

	//write out a 150 saying we're about to open a data conn
	err = write(client_cmd_fd, open_data_response.c_str(),
open_data_response_size);
	CHECK_ERROR(err);

	//connect to new data socket
	err = connect(socketfd, (struct sockaddr *) &addr_to_open,
sizeof(addr_to_open));
	CHECK_ERROR(err);

	return socketfd;
}

string parse_input(char* input_buf) {
	string cmd;				/* holds 3 or 4
letter command, retval */
```

```
      int index = 0;                          /* holds char index to
iterate through input string */
      while (input_buf[index] != ' ' && input_buf[index] != '\r' &&
input_buf[index] != '\n') {
            index++;
      }
      cmd = string(input_buf).substr(0, index);
      return cmd;
}

void parse_ip_and_port(string param, string *ip, int *port1, int
*port2) {
      int commas = 0;                  /* holds number of commas
already iterated over */
      string port;                     /* holds temp port with two
comma delimited numbers before parsing */

      for (int i = 0; i < param.length(); i++) {
            //first increment commas if were on one
            if (param[i] == ',') {
                  commas++;
            }
            if (commas == 4) {
                  //if we've hit the 4th comma, we have complete IP
                  *ip = param.substr(0, i++);
                  port = param.substr(i);
                  //replacing commas with periods
                  replace(ip->begin(), ip->end(), ',', '.');
                  break;
            }
      }
      int j;
      for (int i = 0; i < port.length(); i++) {
            if (port[i] == ',') {
                  *port1 = atoi(port.substr(0, i++).c_str());
                  j = i;
            } else if (port[i] == '\r' || port[i] == '\n') {
                  *port2 = atoi(port.substr(j, i).c_str());
                  break;
            }
      }
}


int handle_cmd(string cmd, char* input_buf) {
      string param;                    /* set to value following
cmd in the input string for manipulation */
      string ip_str;                   /* used to break up param
even mroe into seperate port and IP's for new socket */
      string port_str;                 /* see ip */
      int data_port;                   /* holds actual numberical
value of finalized port */
```

```
     int port1, port2;                      /* holds most significant /
least significant port fields for temporary conversion */
     in_addr_t data_ip;                     /* new ip to open data
socket to */
     string temp_dir_string;                /* holds full output string
but with LFs instead of CRLFs */
     char* dir;                             /* holds full string of
directory entries and their filenames formatted according to spec with
CRLF's added */
     string local_filename;                 /* holds filename of file
to stor or retr on server */
     string remote_filename;                /* holds filename of file
to stor or retr on client */
     int local_fd;                          /* holds fd after 'open'ing
a file for putting or sending */
     struct stat file_stats;                /* holds filesize of local
file to either put or send */
     int err;                               /* tempvalue to check
return codes for errors */

     if (cmd == "TYPE") {
          //format: TYPE [param char]\r\n
          //substr to get single char param
          param = string(input_buf).substr(5, 1);
          //set type variable to given char. This will be checked to
ensure I when stor or retr execd
          type = param[0];

          err = write(client_cmd_fd, command_okay.c_str(),
command_okay_size);
          CHECK_ERROR(err);

     } else if (cmd == "PORT") {
          //format: PORT [xxx,xxx,xxx,xxx,xxx,xxx]\r\n. Each x field
can be 1 to 3 chars, comma seperated
          //need parse IP and port out of remaining string
          param = string(input_buf).substr(5);
          //pass ip and port by ref to set their values
          parse_ip_and_port(param, &ip_str, &port1, &port2);
          //convert host notation to network order
          data_ip = inet_addr(ip_str.c_str());
          CHECK_ERROR(data_ip);
          //piecing together port and converting to network byte
order
          port1 = port1 << 8;
          data_port = port1;
          data_port |= port2;
          //opens new data socket for transfer
          client_data_fd = open_data_socket(data_ip, data_port);
          CHECK_ERROR(client_data_fd);

     } else if (cmd == "USER") {
```

```
            //format: USER [username]. Not sure if this needs to be
implemented but handle it anyway

      } else if (cmd == "QUIT") {
            //format: QUIT. close ftp session
            err = write(client_cmd_fd, quit_response.c_str(),
quit_response_size);
            CHECK_ERROR(err);
            err = close(client_cmd_fd);
            CHECK_ERROR(err);
            //return val forces main loop to exit
            return -1;
      } else if (cmd == "MODE") {
            //format: MODE []

      } else if (cmd == "STRU") {
            //format STRU [param char]. Pull either F or R, deny if R
            //substr to get single char param
            param = string(input_buf).substr(5, 1);
            if (param == "R") {
                  err = write(client_cmd_fd,
stru_failed_response.c_str(), stru_failed_response_size);
                  CHECK_ERROR(err);
            } else if (param == "F") {
                  //were good, staying with default file structure
                  err = write(client_cmd_fd, command_okay.c_str(),
command_okay_size);
                  CHECK_ERROR(err);
            }

      } else if (cmd == "RETR") {
            //first things first check type var
            if (type == 'A' || type == 'a') {
                  close(client_data_fd);
                  err = write(client_cmd_fd,
stor_retr_failed_response.c_str(), stor_retr_failed_response_size);
                  CHECK_ERROR(err);
                  return 0;
            }
            //type is I, good to  move data
            //need to get filename from remaining buffer
            param = string(input_buf).substr(5);
            //pull crlf off of it
            int index;
            for (index = param.length() - 1; index > 0; index--) {
                  if (param[index] == '\r')
                        break;
            }
            param = param.substr(0, index);

            //open file on server
            local_fd = open(param.c_str(), O_CREAT);
```

```
            CHECK_ERROR(local_fd);
            //get filesize
            err = stat(param.c_str(), &file_stats);
            //temp buf for transfer of file
            char file_buf[file_stats.st_size];
            //read file off server into buf
            err = read(local_fd, file_buf, file_stats.st_size);
            CHECK_ERROR(err);
            //write file from buf to data socket
            err = write(client_data_fd, file_buf, file_stats.st_size);
            CHECK_ERROR(err);
            //226 data finished
            err = write(client_cmd_fd, data_finished_response.c_str(),
data_finished_response_size);
            CHECK_ERROR(err);
            //close data socket
            err = close(client_data_fd);
            CHECK_ERROR(err);

      } else if (cmd == "STOR") {
            //first things first check type var
            if (type == 'A' || type == 'a') {
                  close(client_data_fd);
                  err = write(client_cmd_fd,
stor_retr_failed_response.c_str(), stor_retr_failed_response_size);
                  CHECK_ERROR(err);
                  return 0;
            }
            //type is I, good to open socket and move data
            //need to get filename from remaining buffer
            param = string(input_buf).substr(5);
            //pull crlf off of it
            int index;
            for (index = param.length() - 1; index > 0; index--) {
                  if (param[index] == '\r')
                        break;
            }
            param = param.substr(0, index);

            //open file on server
            local_fd = creat(param.c_str(), 0777);
            CHECK_ERROR(local_fd);
            //8mb temp buf for transfer of file
            char* file_buf = (char *) malloc(sizeof(char) * 8388608);
            int filesize = 0;
            //read file off server into buf byte by byte until errors
            while (err > 0) {
                  err = read(client_data_fd, &file_buf[filesize], 1);
                  filesize++;
            }
            //compensate for last increment when it exited loop
            filesize--;
```

```
            //write file from buf to data socket
            err = write(local_fd, file_buf, filesize);
            CHECK_ERROR(err);
            //226 data finished
            err = write(client_cmd_fd, data_finished_response.c_str(),
data_finished_response_size);
            CHECK_ERROR(err);
            //close data socket
            err = close(client_data_fd);
            CHECK_ERROR(err);

      } else if (cmd == "NOOP") {
            //format: NOOP\r\n. only requires okay back
            err = write(client_cmd_fd, command_okay.c_str(),
command_okay_size);
            CHECK_ERROR(err);

      } else if (cmd == "LIST") {
            //format: LIST [dir]\r\n
            if (input_buf[4] != '\r') {
                  param = string(input_buf).substr(5);
                  int i = 0;
                  for (; i < param.length(); i++) {
                        if (param[i] == '\r' || param[i] == '\n') {
                              break;
                        }
                  }
                  param = param.substr(0, i);
            } else if (input_buf[5] == '.') {
                  if (input_buf[6] != '.') {
                        //parent dir
                  } else {
                        //current dir
                        param = cwd;
                  }
            } else {
                  //else keep cwd the same as it is since no dir
supplied
                  param = cwd;
            }

            //build string of given directory files and filesizes
            temp_dir_string = read_dir_files(param);

            //get number of newlines so we know how big to malloc dir
buf
            int newlines = count(temp_dir_string.begin(),
temp_dir_string.end(), '\n');
            dir = (char *) malloc(temp_dir_string.length() + 2 *
newlines);
            for (int i = 0, dir_index = 0; i <
temp_dir_string.length(); i++, dir_index++) {
```

```
                if (temp_dir_string[i] == '\n') {
                     dir[dir_index++] = '\r';
                }
                dir[dir_index] = temp_dir_string[i];
           }

           err = write(client_data_fd, dir, temp_dir_string.length() +
newlines);

           err = write(client_cmd_fd, close_ascii_response.c_str(),
close_ascii_response_size);
           CHECK_ERROR(err);

           err = close(client_data_fd);
           CHECK_ERROR(err);

           delete dir;

     } else {
           //unknown/unsupported cmd
     }
     return 0;
}



int main(int argc, char **argv) {
     int port = 0;                                          /* holds
the port read in as cmdline param to open server on */
     int socketfd = 0;                                      /* holds
socket filedescriptor once initialized */
     client_cmd_fd = 0;                                     /* holds
fd for incoming client socket connection */
     sockaddr_in server_addr;                          /* struct to
hold socket information for binding */
     sockaddr_in client_addr;                          /* struct to
hold socket info for connected client */
     char input_buf[256];                              /* buffer to
hold command input from client on control line */
     int recvd_bytes;                                  /* holds number
of bytes recieved when reading from client */
     string cmd;                                                /*
holds command pulled from user input string */
     type = 'a';                                                /*
holds current transfer type. A for ascii, I for image/binary */
     int err;                                               /* used
for holding temp return values and checking for erros */

     //ensuring only one cmdline param and grabbing port no.
     if (argc != 2) {
           printf("Not enough args.\n");
           return -1;
```

```
	}
	port = atoi(argv[1]);

	char* temp = getcwd(cwd_buf, PATH_MAX);
	if (temp != NULL) {
		cwd = string(cwd_buf);
	}

	//creates socket for IPv4 communication domain with reliable two
way byte stream, and no protocol is necessary to be specified
	socketfd = socket(AF_INET, SOCK_STREAM, 0);
	CHECK_ERROR(socketfd);

	//sockaddr_in struct fields and descriptions can be found here
	//http://www.informit.com/articles/article.aspx?p=169505&seqNum=2
	memset(&server_addr, 0, sizeof(sockaddr_in));
	server_addr.sin_family = AF_INET;                //IPv4 family
	server_addr.sin_port = htons(port);              //translate port
from host to network byte order
	server_addr.sin_addr.s_addr = INADDR_ANY;   //accept incoming
conns from all IPs

	//binding new socket to port entered when program initally run
	err = bind(socketfd, (struct sockaddr *) &server_addr,
sizeof(server_addr));
	CHECK_ERROR(err);

	while(1) {
		//listen on port to open for connections
		err = listen(socketfd, BACKLOG_MAX);
		CHECK_ERROR(err);

		socklen_t client_addr_size = sizeof(client_addr);
		//accept any incoming connections and save client info into
client_addr struct
		client_cmd_fd = accept(socketfd, (struct sockaddr *)
&client_addr, &client_addr_size);
		CHECK_ERROR(client_cmd_fd);

		//write success string for recieved password
		err = write(client_cmd_fd, password_response.c_str(),
password_response_size);
		CHECK_ERROR(err);

		while(1) {
			//pull in next input string
			memset(input_buf, 0, 256);
			recvd_bytes = read(client_cmd_fd, input_buf,
sizeof(input_buf));
			CHECK_ERROR(recvd_bytes);

			//pull out cmd from input string
```

```
                cmd = parse_input(input_buf);

                err = handle_cmd(cmd, input_buf);
                //means we recieved a QUIT, socket is closed in
handle_cmd func so we just need to exit process
                if (err < 0) {
                      close(client_cmd_fd);
                      break;
                }
           }
      }
      return 0;
}
```

makefile

```
# Written by Brian Team (dot4qu)
# Date: 11/29/16
# This makefile is responsible for compiling and linking HWr, the
simple FTP server implementation

CC=g++
CFLAGS=-m32
DEPS=my_ftpd.cpp my_ftpd.h
OBJS=my_ftpd.o

%.o: %.cpp $(DEPS)
      $(CC) -c -o $@ $< $(CFLAGS)

my_ftpd: $(OBJS)
      $(CC) -o $@ $^ $(CFLAGS)

clean:
      @rm -f *.o my_ftpd
```