CIS 415 Operating Systems

Assignment 1 Report Collection

Submitted to:

Prof. Allen Malony

Author:

*Noah Tigner*

**Report**

**Introduction**

*For this project we constructed a pseudo-shell. The shell can be run in command mode as a command-line-interface, or in file mode by specifying -f and providing an input file. The pseudo-shell approximates several Unix commands. The commands are: ls (list the files in the current directory), pwd (list the path of the current directory), cd <directory> (change directory), mkdir <directory> (make a directory), rm <filename> (delete a file), mv <source> <destination> (move a file), and cp <source> <destination> (copy a file into a new file, preserving the original).*

**Background**

*The above commands were implemented by using Linux system commands. For example, system commands like getcwd and chdir were used extensively. Once the system calls were executed, an important step is to check the returned status code in order to be sure that no errors occurred. The primary resource for learning about these system calls was the Man pages, documentation for Linux system commands. Labs 2 and 3 also served as preparation for this project.*

**Implementation**

*A header file was provided for the commands, and a corresponding command.c file was written to implement the shell commands. The main loop of execution, input handling, and error checking is done in main.c. I attempted to make it the code as modular as possible. Once memory is allocated, the program begins by looping through the input line by line. Each word of the line is then tokenized. C functions such as strcmp and strncmp were used extensively to match a token as either a valid command, an argument, or an incorrect command. If a token is matched to a command, the corresponding command is called. If the command accepts arguments, the arguments are grabbed as tokens and passed with the command to the corresponding function in command.c. If arguments are missing or invalid, the command is not executed and an error message is written. At the end of input file, or upon the user entering “exit”, memory is freed, and execution stops. A makefile is also included.*

**Performance Results and Discussion**

*My project performs as expected. Commands can be entered line by line, or with multiple commands and arguments on the same line, with commands separated by the “;” control code. Proper syntax is checked, and incorrect syntax results in the appropriate error messages. Incorrect syntax includes things like ending a line on a control code, not separating commands by control codes, and unrecognized commands. Additionally, commands that require arguments must be supplied with the correct number of valid arguments, or the command will not execute, and an error message will be written.*

*Output, as well as error messages are written to standard out, as either the shell or output.txt, if run in file mode. The output of the commands matches the expected output as shown in the project pdf. The output of file mode functions correctly as well, with the caveat that strange carriage return encodings seem to affect the ability of the cd command to execute properly. Initially, I downloaded input.txt onto my local machine and then used git to pull it into my VM. This caused the strange silent characters. This issue was fixed by deleting the file and typing it myself.*

*Surprisingly, the most difficult part of this project was handling the “;” characters, and handling the errors associated with their misuse. I was able to catch all of the possible forms of invalid syntax listed. However, the checks for invalid syntax in my code are not terribly concise, and added a lot of repetition and lines of code to the project.*

**Conclusion**

*Implementing system calls was useful experience. The process of designing and building a pseudo-shell taught me a lot about how system calls are executed, how errors are handled, common mistakes that lead to such errors, etc.*

**Code**

Main.c

# include <stdio.h>

# include <stdlib.h>

# include <string.h>

# include <unistd.h>

# include <dirent.h>

# include <stdbool.h>

# include <sys/types.h>

# include "command.h"

void dealloc(char \*buffer, FILE \*infile, FILE \*outfile ) {

/\* Utility to free memory, close files \*/

if(buffer != NULL) { free(buffer); }

if(infile != NULL) { fclose(infile); }

if(outfile != NULL) { fclose(outfile); }

}

int startsWith(char \*string, char \*prefix) {

/\* Utility to check if a string has a prefix \*/

return strncmp(string, prefix, strlen(prefix));

}

int endsWith(char \*string, char \*suffix) {

/\* Utility to check if a string has a suffix \*/

return (strcmp(string+strlen(string)-strlen(suffix), suffix));

}

void parseLine(char \*buffer, size\_t bufferSize, FILE \*infile, FILE \*outfile) {

/\* Steps through line by line, token by toke, calling the appropriate commands \*/

// Variables

char \*token;

char \*next;

char \*arg1;// = (char\*)calloc(64, sizeof(char));

char \*arg2;// = (char\*)calloc(64, sizeof(char));

char whitespace[2] = " ";

size\_t input;

bool command = false;

bool cc = false;

if(infile == NULL) {

infile = stdin;

command = true;

printf(">>> ");

}

while((input = getline(&buffer, &bufferSize, infile)) != -1) {

// Remove newline

char \*newline = strchr(buffer, '\n');

if(newline) {

\*newline = 0;

}

// Tokenize the input string

token = strtok(buffer, whitespace);

while(token != NULL) {

if(strncmp(token, "exit", 4) == 0) {

dealloc(buffer, infile, outfile);

exit(0);

}

else if(strncmp(token, ";", 1) == 0) {

write(1, "Error! Unrecognized command: \n", 31);

token = NULL;

}

else if(strcmp(token, "ls") == 0) {

next = strtok(NULL, whitespace);

if(next != NULL) {

if(strncmp(next, ";", 1) == 0) {

listDir();

next = NULL;

token = strtok(NULL, whitespace);

if(token == NULL) {

write(1, "Error! Unrecognized command: \n", 31);

}

}

else {

write(1, "Error! Incorrect syntax. No control code found.\n", 49);

next = NULL;

token = NULL;

}

}

else {

listDir();

next = NULL;

token = strtok(NULL, whitespace);

}

}

else if(strcmp(token, "pwd") == 0) {

next = strtok(NULL, whitespace);

if(next != NULL) {

if(strncmp(next, ";", 1) == 0) {

showCurrentDir();

next = NULL;

token = strtok(NULL, whitespace);

if(token == NULL) {

write(1, "Error! Unrecognized command: \n", 31);

}

}

else {

write(1, "Error! Incorrect syntax. No control code found.\n", 49);

next = NULL;

token = NULL;

}

}

else {

showCurrentDir();

next = NULL;

token = strtok(NULL, whitespace);

}

}

else if(strcmp(token, "cat") == 0) {

arg1 = strtok(NULL, whitespace);

if(arg1 == NULL) {

write(1, "Error: no file listed\n", 23);

}

else {

next = strtok(NULL, whitespace);

if(next != NULL) {

if(strncmp(next, ";", 1) == 0) {

displayFile(arg1);

next = NULL;

token = strtok(NULL, whitespace);

if(token == NULL) {

write(1, "Error! Unrecognized command: \n", 31);

}

}

else {

write(1, "Error! Incorrect syntax. No control code found.\n", 49);

next = NULL;

token = NULL;

}

}

else {

displayFile(arg1);

next = NULL;

token = strtok(NULL, whitespace);

}

}

arg1 = NULL;

}

else if(strcmp(token, "mv") == 0) {

arg1 = strtok(NULL, whitespace);

arg2 = strtok(NULL, whitespace);

if(arg1 == NULL || arg2 == NULL) {

write(1, "Error: missing arguments\n", 26);

}

else {

char \*temp = calloc(32, sizeof(char));

strcpy(temp, arg2);

if(startsWith(arg1, "../") == 0 && strcmp(arg2, ".") == 0) {

strcpy(temp, arg1+3);

}

next = strtok(NULL, whitespace);

if(next != NULL) {

if(strncmp(next, ";", 1) == 0) {

moveFile(arg1, temp);

next = NULL;

token = strtok(NULL, whitespace);

if(token == NULL) {

write(1, "Error! Unrecognized command: \n", 31);

}

}

else {

write(1, "Error! Incorrect syntax. No control code found.\n", 49);

next = NULL;

token = NULL;

}

}

else {

moveFile(arg1, temp);

next = NULL;

token = strtok(NULL, whitespace);

}

free(temp);

}

arg1 = NULL;

arg2 = NULL;

}

else if(strcmp(token, "cp") == 0) {

arg1 = strtok(NULL, whitespace);

arg2 = strtok(NULL, whitespace);

if(arg1 == NULL || arg2 == NULL) {

write(1, "Error: missing arguments\n", 26);

}

else {

char \*temp = calloc(32, sizeof(char));

strcpy(temp, arg2);

if(startsWith(arg1, "../") == 0 && strcmp(arg2, ".") == 0) {

strcpy(temp, arg1+3);

}

next = strtok(NULL, whitespace);

if(next != NULL) {

printf("NEXT: %s\n", next);

if(strncmp(next, ";", 1) == 0) {

copyFile(arg1, temp);

next = NULL;

token = strtok(NULL, whitespace);

printf("%s\n", token);

if(token == NULL) {

write(1, "Error! Unrecognized command: \n", 31);

}

}

else {

write(1, "Error! Incorrect syntax. No control code found.\n", 49);

next = NULL;

token = NULL;

}

}

else {

copyFile(arg1, temp);

next = NULL;

token = strtok(NULL, whitespace);

}

free(temp);

}

arg1 = NULL;

arg2 = NULL;

}

else if(strcmp(token, "mkdir") == 0) {

arg1 = strtok(NULL, whitespace);

if(arg1 == NULL) {

write(1, "Error: missing arguments\n", 26);

}

else {

next = strtok(NULL, whitespace);

if(next != NULL) {

if(strncmp(next, ";", 1) == 0) {

makeDir(arg1);

next = NULL;

token = strtok(NULL, whitespace);

if(token == NULL) {

write(1, "Error! Unrecognized command: \n", 31);

}

}

else {

write(1, "Error! Incorrect syntax. No control code found.\n", 49);

next = NULL;

token = NULL;

}

}

else {

makeDir(arg1);

next = NULL;

token = strtok(NULL, whitespace);

}

}

arg1 = NULL;

}

else if(strcmp(token, "rm") == 0) {

arg1 = strtok(NULL, whitespace);

if(arg1 == NULL) {

write(1, "Error: missing arguments\n", 26);

}

else {

next = strtok(NULL, whitespace);

if(next != NULL) {

if(strncmp(next, ";", 1) == 0) {

deleteFile(arg1);

next = NULL;

token = strtok(NULL, whitespace);

if(token == NULL) {

write(1, "Error! Unrecognized command: \n", 31);

}

}

else {

write(1, "Error! Incorrect syntax. No control code found.\n", 49);

next = NULL;

token = NULL;

}

}

else {

deleteFile(arg1);

next = NULL;

token = strtok(NULL, whitespace);

}

}

arg1 = NULL;

}

else if(strcmp(token, "cd") == 0) {

arg1 = strtok(NULL, whitespace);

if(arg1 == NULL) {

write(1, "Error: missing arguments\n", 26);

}

else {

next = strtok(NULL, whitespace);

if(next != NULL) {

if(strncmp(next, ";", 1) == 0) {

changeDir(arg1);

next = NULL;

token = strtok(NULL, whitespace);

if(token == NULL) {

write(1, "Error! Unrecognized command: \n", 31);

}

}

else {

write(1, "Error! Incorrect syntax. No control code found.\n", 49);

next = NULL;

token = NULL;

}

}

else {

changeDir(arg1);

next = NULL;

token = strtok(NULL, whitespace);

}

}

arg1 = NULL;

}

else if(strcmp(token, "\n") != 0) {

write(1, "Error: Unrecognized command!\n", 30);

token = strtok(NULL, whitespace);

}

}

if(command) {

write(1, ">>> ", 4);

}

}

dealloc(buffer, infile, outfile);

}

int main(int argc, char \*argv[]) {

setbuf(stdout, NULL);

// Variables

FILE \*infile;

FILE \*outfile;

// Allocate memory for the buffer

char \*buffer;

size\_t bufferSize = 32;

size\_t input;

buffer = (char\*)malloc(bufferSize \* sizeof(char));

// File Mode

if(argc == 3 && (strncmp(argv[1], "-f", 2) == 0 || strncmp(argv[1], "-file", 5) == 0)) {

outfile = freopen("output.txt", "w+", stdout);

infile = fopen(argv[2], "r");

if(infile == NULL) {

write(1, "Input file missing\n", 20);

dealloc(buffer, NULL, NULL);

exit(1);

}

parseLine(buffer, bufferSize, infile, outfile);

return 0;

}

// Command Mode

else {

parseLine(buffer, bufferSize, NULL, NULL);

return 0;

}

}

Command.c

*# include <stdio.h>*

*# include <string.h>*

*# include <stdlib.h>*

*# include <stdbool.h>*

*# include "command.h"*

*# include <fcntl.h>*

*# include <unistd.h>*

*# include <dirent.h>*

*void listDir() { // ls*

*char ls[256];*

*struct dirent \*drnt;*

*DIR \*dir;*

*char \*filename;*

*if(getcwd(ls, sizeof(ls)) == NULL) {*

*write(1, "Error: unable to read from current directory\n", 46);*

*}*

*else {*

*if((dir = opendir(ls)) == NULL) {*

*write(1, "Error: unable to open current directory\n", 41);*

*}*

*else {*

*while((drnt = readdir(dir)) != NULL) {*

*filename = drnt->d\_name;*

*if(!startsWith(filename, ".") == 0) {*

*strcat(filename, " ");*

*write(1, filename, strlen(filename));*

*}*

*}*

*write(1, "\n", 2);*

*}*

*}*

*closedir(dir);*

*}*

*void showCurrentDir() { // pwd*

*char ls[256];*

*if(getcwd(ls, sizeof(ls)) == NULL) {*

*write(1, "Error: unable to read from current directory\n", 46);*

*}*

*else {*

*strcat(ls, "\n");*

*write(1, ls, strlen(ls));*

*}*

*}*

*void makeDir(char \*dirName) { // mkdir*

*int status;*

*char cd[64];*

*char path[256];*

*// path/dirName*

*getcwd(cd, sizeof(cd));*

*strcpy(path, "");*

*strcat(path, cd);*

*strcat(path, "/");*

*strcat(path, dirName);*

*int success = mkdir(path, 0777);*

*if(success != 0) {*

*write(1, "Error: couldn't create directory\n", 34);*

*}*

*}*

*void changeDir(char \*dirName) { // cd*

*int success = chdir(dirName);*

*if(success != 0) {*

*write(1, "Error: coudn't change directory\n", 33);*

*}*

*}*

*void copyFile(char \*sourcePath, char \*destinationPath) { // cp*

*int in\_size, in\_descriptor, out\_size, out\_descriptor;*

*char \*in\_buffer = (char\*)calloc(256, sizeof(char));*

*in\_descriptor = open(sourcePath, O\_RDONLY);*

*if(in\_descriptor < 0) {*

*write(1, "Error: can't open file\n", 24);*

*}*

*else {*

*in\_size = read(in\_descriptor, in\_buffer, 256);*

*out\_descriptor = open(destinationPath, O\_WRONLY | O\_CREAT, 0777);*

*if(out\_descriptor < 0) {*

*write(1, "Error: can't open file\n", 24);*

*close(in\_descriptor);*

*}*

*else {*

*out\_size = write(out\_descriptor, in\_buffer, strlen(in\_buffer));*

*close(in\_descriptor);*

*close(out\_descriptor);*

*}*

*}*

*free(in\_buffer);*

*}*

*void moveFile(char \*sourcePath, char \*destinationPath) { // mv*

*int success = rename(sourcePath, destinationPath);*

*if(success != 0) {*

*write(1, "Error: can't move file\n", 24);*

*}*

*}*

*void deleteFile(char \*filename) { // rm*

*int status = unlink(filename);*

*if(status != 0) {*

*write(1, "Error: can't delete file\n", 26);*

*}*

*}*

*void displayFile(char \*filename) { // cat*

*int size, descriptor;*

*char \*buffer = (char\*)calloc(1024, sizeof(char));*

*strcpy(buffer, "");*

*descriptor = open(filename, O\_RDONLY);*

*if(descriptor < 0) {*

*write(1, "Error: can't open file\n", 24);*

*}*

*else {*

*size = read(descriptor, buffer, 1024);*

*write(1, buffer, strlen(buffer));*

*close(descriptor);*

*}*

*free(buffer);*

*}*