CS 4414: Operating Systems

Project 1: Writing a Shell

Michael Eller

14 September 2016

Introduction

In the Unix/Linux operating system, the terminal or shell is a tool for the user to communicate with the kernel. The purpose of this project is to write a shell in C/C++ with some of the features of the actual Unix/Linux shell. Using system calls, the user can input a line of commands and the shell program will parse the text and execute the commands accordingly. This shell supports running programs and supports commands for file redirection and pipes.

The input to the shell is a sequence of lines up to 100 characters. If the input contains more than 100 characters the shell outputs an error message and continues on the next line. Each line consists of tokens, which are separated by spaces. The two types of tokens are operators and words. The only operators are file redirectors and pipes ('<', '>', '|'). Words are allowed to contain the characters A-Z, a-z, 0-9, '-', '.', '/', and '_'. If a word contains characters other than those listed above, the shell will print an error message and then continue parsing with the next line. The only other legal input to the shell is 'exit' which closes the program.

Implementation

This shell implementation consists of three main parts: parsing, file redirection and execution, and piping. The shell itself is basically a state machine wrapped within a while(1) loop. A char* buffer uses fgets() to read the input from STDIN. Once the buffer is filled and checked for errors, the line is split into a vector by spaces. That vector is then split into multiple vectors based on pipes and added to a single parent vector. The vectors within the parent vector represent token groups and are executed individually with their inputs/outputs piped accordingly. Those token groups can contain any command and its arguments along with I/O file redirection.

As previously mentioned, the parsing was implemented as a state machine. The states are an enum called ParseState. The states are as follows: Start, CheckWord, CheckOperator, InputRedirection, OutputRedirection, and Execute Command. Start sets the command argument as the first element in the vector. CheckWord checks the next element and adds it if it is a valid word. If it is not a valid word, the state moves to CheckOperator. That state checks if the token is a file redirector. If the token is not an operator, an invalid input error is thrown. If it is a redirector operator, then the state moves to the appropriate redirection state. The redirection states use the open() system call to open the files after the operator element. Once the state machine reaches the end of the list the state moves to ExecuteCommand. Execute command uses fork() to create a child process and creates a pipe for handling any I/O specification. Using the dup2() system call, the input and output of the forked child process is changed to the appropriate file/ STD stream. The child process then uses the execve() system call to complete the parsing while the parent waits. The exit status code is then printed to stderr.

In order to implement piping. A recursive function was created to handle parsing the various token groups in an ordered fashion. The parent creates a child process for every

token group and strings the outputs together using pipes. The parent process then waits for every child process created.

Problems Encountered

The hardest part of this homework was piping. Once I figured out the pattern for directing the read/write ends of a pipe to the appropriate process, the rest was pretty straightforward. Also, due to the structure of my recursive piping function, printing the exit status codes was especially challenging. In the recursive function, the process that actually executes the command is the child of the child of the parent, so printing to STDOUT of the parent was challenging; however, this problem was eliminated by printing to STDERR instead.

Testing and Analysis

To test the shell, a test input file was created with various test cases. The STDERR messages were directed to a file, and the shell output messages were directed into another file. The scores file is a file used in command 9 of the input file. The 'test_output' file was used in command 7. All test files are shown below.

bash command

```
./msh<test_input_file>outputfile 2>exit_code_file
```

test_input_file

```
ls
gr*p
/bin/ls|/bin/grep a
/bin/ls | /bin/grep a | /bin/grep
/bin/ls
/bin/ls | /bin/grep a | /bin/grep b
/bin/cat scores | /bin/grep uva > test_output
/bin/ls -l | /bin/grep a
/bin/grep a < scores
```

test_output_file

```
// 1
invalid input
// 2
invalid input
// 3
invalid input
// 4
// 5
eit_codes
exit_code_file
exit_codes
foo
makefile
mbe9a.tar
_minted-p1meller
msh
```

```
msh.tar
out
outputfile
p1meller.aux
p1meller.log
p1meller.out
p1meller.pdf
p1meller.synctex.gz
p1meller.tex
scores
shell.cpp
shell.h
shell.o
test_input_file
test_normal
test_output
test_output_file
// 6
mbe9a.tar
// 7
// 8
total 1020
-rwxr--r-- 1 vmuser vmuser    314 Sep 14 15:07 makefile
-rwxr--r-- 1 vmuser vmuser 143360 Sep 14 19:14 mbe9a.tar
-rwxr--r-- 1 vmuser vmuser 143360 Sep  7 00:16 msh.tar
-rwxr--r-- 1 vmuser vmuser   1018 Feb 25  2016 p1meller.aux
-rw-rw---- 1 vmuser vmuser     30 Sep 14 19:18 test_normal
// 9
60-30 uva
55-80 uva
61-33 uva
// 10 (newline character)
invalid input
```

exit_code_file

```
512
512
512
0
0
Usage: /bin/grep [OPTION]... PATTERN [FILE]...
Try '/bin/grep --help' for more information.
512
0
0
0
0
0
0
0
0
0
```

test_output

```
60-30 uva
55-80 uva
61-33 uva
```

scores

```
60-30 uva
25-25 not
```

```
55-80 uva
34-46 not
62-10 not
61-33 uva
```

These test cases verify the shell's ability to:

- perform file redirection
- run commands with or without arguments
- check for invalid input
- handle piping

Conclusion

This shell implementation satisfies the specifications described in the introduction and is fully functional. Exit status codes are printed to STDERR. The shell can also run programs and perform file I/O. The implementation of piping was probably more difficult than it had to be; however, the shell is functional and only about 370 lines with comments.

Code

makefile

```
# Michael Eller
# mbe9a
# CS 4414 Fall 2016

OBJS = shell.o
CC = g++
DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)

msh : $(OBJS)
        $(CC) $(LFLAGS) $(OBJS) -o msh

shell.o : shell.h shell.cpp
        $(CC) $(CFLAGS) shell.cpp

clean:
        \rm *.o *~ msh

tar:
        tar cfv mbe9a.tar msh shell.cpp shell.h makefile
```

shell.h

```
#ifndef SHELL_H
#define SHELL_H

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <ctype.h>
#include <sys/wait.h>
#include <string.h>
#include <cstring>
#include <errno.h>
#include <vector>
using namespace std;

enum ParseState{Start, CheckWord, CheckOperator, InputRedirection, OutputRedirection, ExecuteCommand};
struct executable{char * command; vector<char*> args;};
bool word(char* str);
int lt_gt(char* str);
void parse(vector<char*> list, int single);
void pipe_recursion(vector<vector<char*> > vec, int pipes[], int size, int input, int output);
void close_pipes(int pipes[]);

#endif
```

shell.cpp

```
#include "shell.h"

// macro maximum input length
#define MAX_LENGTH        100

// variable used to determine number of pipes to use
// global for recursive purposes
int length;

// a valid word must only contain the characters in this array
char validchars[66] = {'A','a','B','b','C','c','D','E','e','F','f','G','g','H','h','I','i',
'J','j','K','k','L','l','M','m','N','n','O','o','P','p','Q','q','R','r','S','s','T','t','U',
'u','V','v','W','w','X','x','Y','y','Z','z','0','1','2','3','4','5','6','7','8','9','-','_','.','/'};

// check if char* is only made of validchars
// return true or false
bool word(char* str) {
        bool test = false;
        for (unsigned int i = 0; i < strlen(str); i++) {
                for(int i = 0; i < 66; i++) {
                        test = false;
                        if(*str == validchars[i]) {
                                test = true;
                                break;
                        }
                }
                str++;
        }
        return test;
}

// determine file redirections within token
// outputs an int that means either both are present or only one-- and which one
// for parsing < and > without spaces
int lt_gt(char* str) {
        bool lt = false;
        bool gt = false;
        for(char* iter = str; *iter; ++iter) {
                if (*iter == '>') {
                        gt = true;
                } else if (*iter == '<') {
                        lt = true;
                }
        }
        if(lt && gt) {
                return 3;
        } else if (gt) {
                return 2;
        } else if (lt) {
                return 1;
        } else {
                return 0;
        }
}

// function to print an error message to stdout
void PrintError() {
        printf("invalid input\n");
}

// state machine implementation for parsing token groups and executing them
// piping is not handled in this function (other than files)
void parse(vector<char*> list, int single) {
        vector<char*>::iterator iter = list.begin();
        ParseState state = Start;
        char* command;
```

```
vector<char*> args;
int inFile = 0;
int outFile = 0;
while(!list.empty()) {
        // go to execute state, break while loop
        if(iter == list.end()) {
                        list.clear();
                        state = ExecuteCommand;
        }
        switch(state) {
                // get command
                case Start:
                        if(!word(*iter)) {
                                PrintError();
                                list.clear();
                                break;
                        } else {
                                command = *iter;
                                args.push_back(*iter);
                                iter++;
                                state = CheckWord;
                        }
                        break;
                // if word, add to args
                case CheckWord:
                        if(!word(*iter)) {
                                state = CheckOperator;
                        } else {
                                args.push_back(*iter);
                                iter++;
                        }
                        break;
                // if operator, handle files, otw -> ERROR
                case CheckOperator:
                        if(strcmp(*iter, "<") == 0) {
                                iter++;
                                state = InputRedirection;
                        } else if(strcmp(*iter, ">") == 0) {
                                iter++;
                                state = OutputRedirection;
                        } else {
                                PrintError();
                                list.clear();
                        }
                        break;
                // handle input file pipe
                case InputRedirection:
                        inFile = open(*iter, O_RDONLY);
                        iter++;
                        state = CheckWord;
                        break;
                // handle outout file pipe
                case OutputRedirection:
                        outFile = open(*iter, O_WRONLY | O_TRUNC | O_CREAT,S_IRUSR | S_IRGRP | S_IWGRP | S_IWUSR);
                        iter++;
                        state = CheckWord;
                        break;
                // set up and call execvp by forking a child process
                case ExecuteCommand:
                        int fd[2];
                        pipe(fd);
                        int pid_u = fork();
                        if(pid_u == 0) {
                                // set up input pipe
                                if(inFile != 0) {
                                        dup2(inFile, 0);
                                        close(inFile);
                                } else {
```

```
                                        close(fd[1]);
                                // set up output pipe
                                } if (outFile != 0) {
                                        dup2(outFile, 1);
                                        close(outFile);
                                }
                                // form c_args array
                                char* c_args[args.size() + 1];
                                vector<char*>::iterator iter2;
                                int i = 0;
                                for(iter2 = args.begin(); iter2 < args.end(); iter2++) {
                                        c_args[i] = *iter2;
                                        i++;
                                }
                                // add NULL as the last element
                                c_args[i] = NULL;
                                // use the cwd as the PATH
                                char* cwd = (char*)malloc(100*sizeof(char));
                                cwd = getcwd(cwd, 100);
                                setenv("PATH", cwd, 1);
                                extern char** environ;
                                // execute and check for error
                                int exec;
                                exec = execve(command, c_args, environ);
                                if(exec < 0) {
                                        printf("invalid input\n");
                                        exit(errno);
                                }
                        } else {
                                //close the file pipes and print the status code to stderr
                                close(fd[0]);
                                close(fd[1]);
                                int status = 0;
                                waitpid(pid_u, &status, 0);
                                fprintf(stderr, "%d\n", status);
                        }
                        break;
                }
        }
        return;
}

// simple function to close all pipes passed to it in the form of an int array
void close_pipes(int pipes[]) {
        for (int i = 0; i < length; i++) {
                close(pipes[i]);
        }
}

// piping is performed recursively, every token group results in a forked child calling the 'parse' function on it
// base case (vector size 0)
// special initial case (vector size = initial size i.e. first time the function is called)
// pipes[] is the array of pipes to use, size is the current size of the vector, input is for the read end, output -> write
void pipe_recursion(vector<vector<char*> > vec, int pipes[], int size, int input, int output) {
        // get the last item and then remove it from the vector
        vector<char*> current = vec.back();
        vec.pop_back();
        // base case, only pipe input
        if (vec.size() == 0) {
                int pid_r = fork();
                if(pid_r == 0) {
                        // parse the token group
                        dup2(pipes[length - 2], 0);
                        close_pipes(pipes);
                        parse(current, 1);
                        exit(0);
                } else {
                        // done
```

```
                        return;
                }
        // initial case, only pipe output
        } else if (vec.size() == unsigned(size)) {
                int pid_r = fork();
                if(pid_r == 0) {
                        dup2(pipes[1], 1);
                        close_pipes(pipes);
                        parse(current, 0);
                        exit(0);
                }else {
                        input += 2;
                        output += 2;
                        pipe_recursion(vec, pipes, size, input, output);
                }
        } else {
                // else pipe based on position in groups
                int pid_r = fork();
                if(pid_r == 0) {
                        dup2(pipes[input], 0);
                        dup2(pipes[output], 1);
                        close_pipes(pipes);
                        parse(current, 0);
                        exit(0);
                } else {
                        input += 2;
                        output += 2;
                        pipe_recursion(vec, pipes, size, input, output);
                }
        }
}

// while(1) loop, gather input, format vectors
int main(int argc, char** argv) {
        char *buf = (char*)malloc(sizeof(char)*1024);
        char *token;
        vector<char*> tokens;
        vector< vector<char*> > groups;
        while(1) {
                // fill char buffer with console input
                fgets(buf, 1024, stdin);
                if(feof(stdin)) {
                        exit(0);
                }
                // remove trailing newline or carriage return
                for (int x = 0; (unsigned)x < strlen(buf); x++) {
                        if ( buf[x] == '\n' || buf[x] == '\r' ) {
                                buf[x] = '\0';
                        }
                }
                // throw error if over 100 chars
                if (strlen(buf) > MAX_LENGTH) {
                        printf("invalid input");
                }
                // exit program if input is 'exit'
                else if (strcmp(buf, "exit") == 0) {
                        break;
                }
                else {
                        // split buffer based on spaces and add to a vector
                        token = strtok(buf," ");
                        // if it's just spaces, start over
                        if(token == NULL) {
                            PrintError();
                            continue;
                        }
                        while (token != NULL) {
                                tokens.push_back(token);
```

```
        token = strtok(NULL," ");
}
vector<char*>::iterator it;
vector<char*> temp;
// split each token based on pipes and put into nested list
for (it = tokens.begin(); it < tokens.end(); it++) {
        if (strcmp(*it, "|") == 0) {
                groups.push_back(temp);
                temp.clear();
                continue;
        }
        else if (it == tokens.end() - 1) {
                temp.push_back(*it);
                groups.push_back(temp);
                temp.clear();
                continue;
        }
        temp.push_back(*it);
}
tokens.clear();
vector<vector<char*> >::iterator its;
vector<vector<char*> > groups_real;
for(its = groups.begin(); its < groups.end(); its++) {
        // this for loop was added to handle file redirectors
        // not separated by spaces
        vector<char*> print = *its;
        for (it = print.begin(); it < print.end(); it++) {
                if(!word(*it)) {
                        int check = lt_gt(*it);
                        char * dummy = *it;
                        if(strcmp(*it, "<") == 0) {
                                tokens.push_back((char*)"<");
                        } else if (strcmp(*it, ">") == 0) {
                                tokens.push_back((char*)">");
                        // both
                        } else if (check == 3) {
                                char * lt = strtok(dummy, "<");
                                tokens.push_back(lt);
                                tokens.push_back((char*)"<");
                                lt = strtok(NULL, "<");
                                tokens.push_back(lt);
                                char * gt = strtok(lt, ">");
                                tokens.push_back((char*)">");
                                gt = strtok(NULL, ">");
                                tokens.push_back(gt);
                        // greater than
                        } else if (check == 2) {
                                char * gt = strtok(dummy, ">");
                                tokens.push_back(gt);
                                tokens.push_back((char*)">");
                                gt = strtok(NULL, ">");
                                tokens.push_back(gt);
                        // less than
                        } else if (check == 1) {
                                char * lt = strtok(dummy, "<");
                                tokens.push_back(lt);
                                tokens.push_back((char*)"<");
                                lt = strtok(NULL, "<");
                                tokens.push_back(lt);
                        // if it's another configuration, it's invalid
                        } else {
                                PrintError();
                        }
                } else {
                        tokens.push_back(*it);
                }
        }
        groups_real.push_back(tokens);
```

```
                        tokens.clear();
                }
                vector<vector<char*> > reverse;
                vector<vector<char*> >::iterator r_it;
                // reverse the groups for recursive piping
                for(r_it = groups_real.end() - 1; r_it >= groups_real.begin(); r_it--) {
                        reverse.push_back(*r_it);
                }
                groups_real.swap(reverse);
                int size = groups_real.size();
                if(groups_real.size() > 1){
                        // if there are pipes, use recursive piping function
                        length = (groups_real.size() - 1) * 2;
                        int pipes[length];
                        for (int i = 0; i < length / 2; i++) {
                                pipe(pipes + 2*i);
                        }
                        pipe_recursion(groups_real, pipes, groups_real.size(), -2, 1);
                        close_pipes(pipes);
                        for(int i = 0; i < size; i++) {
                                wait(NULL);
                        }
                // else just parse the one group
                } else {
                        parse(groups_real.front(), 1);
                }
                // clear all the lists
                temp.clear();
                tokens.clear();
                groups.clear();
                groups_real.clear();
                // continue
        }
    }
    return 0;
}
```