Operating Systems Homework 4 – Writing Own Shell

Dan Donovan, dpd5nz@virginia.edu

CS4414: Operating Systems
HW4: Writing A Simple Shell
Dan Donovan
dpd5nz

4/22 March 2018

On my honor, I pledge to have neither give nor received unauthorized aid on this assignment.

I was able to complete Machine Problem 4 in accordance with the problem specifications. Running the executable “msh” and either typing in to stdin or reading in from a file are both accepted.

1. Problem:

This machine problem involves writing a simple shell that takes lines of input from the user. These lines are determined to be either invalid or valid lines that adhere to a series of specifications in the problem spec’s shell language guidelines list. When a line is determined to be valid within the shell language, the line is to be broken up into tokens, operators, and words. This MP then asks for the various commands and their associated arguments and filespecs and attempts to execute each command. This MP also asks for the shell to support input/output redirection as well as piping. Beyond that, the shell is to only return the output of commands, an error messages for invalid input and exit status codes.

2. Approach

This assignment differed from homework three as it allowed for significant freedom in algorithm and data structure usage and design. For this assignment I first looked at the general skeleton offered in the shell spec. From there I pinpointed the different types of outcomes for each input line: invalid inputs, executed commands, failure to properly execute commands, and the exit condition.

To help simplify the problem, I decided to implement several helper functions which aimed to parse each line of input and tokenize each line into tokens, operators and words. I then dedicated several chunks of code to implement I/O redirection and piping.

Ultimately I chose an **iterative** approach that solved the problem specifications by first implementing single commands, then integrating piping and finally adding I/O redirection. This approach allowed me to better understand incorrect code behavior as it presented itself.

Helper Functions:

char ** parseLine(char * raw) - This helper function takes in a line of input. In this function, there first is logic to determine if the line has incorrect placement of pipes, or file redirection. This function also performs other sanitizing steps like stripping leading spaces or ending spaces. Any of these input errors return NULL and trigger the program to skip this line of input. For commands that do not have input errors this function returns a 2d char split into *n* number of char * pointers each containing a token that is in that line.

int itemizer(string command, string * argumentP) - this helper function is called on each of the tokens that was previously sanitized by the parseLine helper function. This function takes in a command and makes sure each token does not contain invalid characters like spaces or tabs, and indicates whether the line contains a redirection or not: useful knowledge that will be used in other portions of the program. It then adds the command to a string array that can be easily grabbed and used when the command needs to be executed.

Data Structures:

I used several data structures in each iteration of a while loop that checked for a line of input. The most important and frequently accessed structures were:

ch** tokens - this 2D char array is used in conjunction with the helper functions to contain mostly sanitized inputs for a command that needed to be tokenized.

String * tokenized - this string array contained a list of each of the tokens that are in the string making it easier for pipes redirection and execvp calls.

int pipes[2] - This is the structure I use for the pipe calls that occur in the loop that iterates over each of the commands and associated args. It is the structure that I call dup2 and close on and is used to facilitate piping for multi-command lines.

int pid_status[] - a int array that contains a list of status codes from each exec call that can later be iterated through to print out exit codes.

Algorithms and Step Through

In the main method:

1) While there's more input:
2) Read in a line and check via helper functions whether that line is valid syntactically. If that line is valid, than split it on “|”s to get the number of commands and each one’s associated arguments.
3) Set up pipes and pid_status array.
4) For each of the command (and associated arguments):
    a. Call itemizer and go through one more additional for loop to check for correct and valid characters in the string. In this step if there is redirection, it appends the cwd to relative files paths if needed.
    b. Pid = fork();
    c. If (child process) -- take each command and associated arguments and put it into an array char * argv to be used for executing.

i. If this is a multi-command line: make the corresponding call to dup2 and to close(pipe[certain end]) depending on where in the list of commands to be executed the current command currently is. This links the output of previous commands to the input of the next piped command when the next command is iterated through in step 4.

ii. After piping (if necessary) call execvp for the current command and its associated arguments

d. else(pid > 0 so parent process)

i. close(pipes[1]) and set up the input to the next cycle's dup2 command if necessary

ii. Make a call to waitpid to wait for the child to finish executing

5) Now that all commands in the line have been executed, for each of the exit statues contained in pid_statuses: print out status to standard error (cerr)

6) When there is no more lines of input or if the exit command was made from stdin, then it's time to exit while loop and end the shell.

3. Problems Encountered

This homework caused a lot of headache and problems for me. The largest roadblock I faced was determining if and how my pipes were functioning. At times they seemed like a black box and it was near impossible to decipher if I was passing to pipes correct input or if that input or output was being passed between commands and through pipes correctly. I overcame this by reading a lot of advice from online forums and by meticulously stepping through my code to determine if there were any areas where I needed another close() command or if there were ever any ends of the pipes I incorrectly closed. I also struggled with making sure my shell conformed to all the lexical specifications for the shell and determining where in a specific command various characters, spaces, pipes and files redirections were allowed. I was able to overcome this by making a comprehensive list of various test cases that tried each of these specifications to understand if my shell was behaving properly.

I learned how important it is to iteratively test my code and to utilized a debugger to understand what portions of a codebase are behaving incorrectly. Additionally, I recognized the importance of thoroughly planning out the logic of my code before actually sitting down to write out the any physical lines of code.

4. Testing

I spoke to this a little already, but my primary methodology of testing was to first determine the specifications for valid input and what to return. After reaching this understanding I thought of a number of commands and inputs lines that tested each component of shell command's validity, and commands that either should or should not execute. I ensured coverage by using test cases that almost acted like unit tests, testing one aspect of my shell and nothing else such that I knew when problems arose.

I relied upon several of the commands and their expected outputs given in the spec, as well as many boundary-like tests cases to test unlikely cases my shell might face. My tests uncovered a number of expected errors. For example my char ** tokens kept having odd characters appended to it in one of the helper methods, a manifestation that occurred only some of the time. Additionally it was this comprehensive testing that allowed me to realize my shell was incorrectly allowing commands that had more than one of the same file redirection to be considered valid lines.

5. Conclusion

This homework showcases the oftentimes hidden and seldom found errors that can reside in improperly set up piping and shell systems. Although, during testing, my shell behaved incorrectly infrequently, it was these unexpected deviations from how my code should have been performing that shows how precise and correct piping and logic has to be with shell setup. I also feel that I now have a strong working knowledge of how pipes and their associated dup2 and close functions work, which is a practical piece of operating systems knowledge that I am glad to have. Ultimately I found this assignment to be a useful exercise in learning how shells function and how processes communicate between children and parents through the usage of pipes and other means.

Sources:

```
/*
 * Machine Problem #4 - Writing A Simple Shell
 *
 * CS4414 Operating Systems
 *
 * Dan Donovan
 *
 *
 * up.cpp - c++ file containing functionality for taking input from stdin
and determining if it a valid command / list of command and then executes
those commands. This support file reidirection and piping
 * Dan Donovan Spring 2018
 *
 * The following code display my implementation for parsing commands and
redirection, piping and executing different commands within the input. It
includes my chosen implementation for various helper methods and pipe
usage.
 *
 *
 *     COMPILE: make
 *     OBJECTS: up.o
 *
 *     it creates the executable ./msh excutable can take stdin or read
from a file a list of commands
 *
 *     MODIFICATIONS:
```

```
 *            4/16 - simple uni-command file excution without piping or
redirection
 *            4/17 - more uni-command support and validation for
different inputs and incorrectly    formatted lines of input
 *            4-20 - file redirection functionality
 *            4-23 - pipe functionality and further input validation and
execution
 *            4-24 - final series of input validation and checking
 *
 *
 *
 */




#include <stdint.h>
#include <stdlib.h>
#include <cstddef>
#include <iostream>
#include <sstream>
#include <fstream>
#include <math.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <string>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>



#define null 0;
using namespace std;

/* Global Declarations */
  bool valid; /* boolean valid that holds if a token is validly set up */
  int pipecount; /* number of pipes in the input string */
  int * errorStatus; /* int[] for the various errors and exit codes */
  char ** tokens; /* char 2d array to hold partly sanitized command and
args list from input */
  char operators[3] = {'>','<','|'}; /* type of valid operators */
  int pidstatus; /* status of a progress from execvp */
  char * cwd; /* current working directory */
  char ** args; /* optional 2d array to hold args for execvp command */
  pid_t pid; /* value used to iterate through erorstatus array */
  pid_t pids[50] = {0}; /* array of pids initialized all to 0 */

/* helper function to santize each command and its associated arguments,
returns NULL if any
 invalid commands*/
int itemizer(string command, string* argumentP) {
  int modified = 0; /* if list has file redirection */
   int i = 0;
    int current = 0;
    int start = command.find_first_not_of("\t "); /* first index of
command that isnt a string */
    int end = command.find_last_not_of("\t "); /* last index of command
that isnt a string  */
    i = start;
    if ( start == -1){
      return -1;
    }
    if (start == end && command[start] != ' ' && command[start] != '\t') {
/* if string is only 1 character long but is valid */
       argumentP[0] = command[start];

      return current + 1;
    }
    if (start == end && (command[start] == ' ' || command[start] ==
'\t')){
      return -1;
    }
    if (command[start] == '|' || command[start] == '>' || command[start]
== '<' || command[end] == '|' || command[end] == '>' || command[end] ==
'<'){
      return -1; /* invalid placement of a pipe or redirection operator */
    }

    while (i < end) {

      while ((command[i] == ' ' || command[i] == '\t')) {
       i += 1;
       /* get rid of redundant middle spaces */
       }

      int startOfentry = i;

      while (command[i] != ' ' && command[i] != '\t' && i <= end){
       i += 1;
       }
      /*  token has not been indexed */
      int endOfentry = i;
      string token = command.substr(startOfentry,  (endOfentry -
startOfentry));
      argumentP[current] = token; /* put that token in the current index
of the string array */

      if (token == ">" || token == "<"){
       modified = current; /* the correct amount of operators excluding
the redirection */
      }
      current += 1;
    }
    current -= 1;
  return current; /* return the current index of the last command */
```

```
}

/* helper function that takes the raw input and does various input
validation, and splits the line on "|" and places result, if valid, in a
2D array */
char ** parseLine(char * raw){
  /* various  indexes, pipecount and file descriptor counters */
  valid = true;
  int index = 0;
  int argVal = 0;
  int size = 100;
  int hasOut = 0;
  int hasIn = 0;
  int starting = 0;
  int previousOut = 0;
  pipecount = 0;
  int count = 0;
  int p = 0;
  char ** command = (char **)malloc(strlen(raw) *sizeof(char*));
  if (raw[p] == '|' || raw[p] == '>' || raw[p] == '<'  || raw[strlen(raw)-
1] == '|' || raw[strlen(raw)-1] == '>' || raw[strlen(raw)-1] == '<'){
    /* if the first/last character of a string is a pipe or redirection
return invalid string */
    valid = false;
    return NULL;

  }



    string curr = "";
    /* for each character in the string  */
    for (int i = 0; i < strlen(raw); i++){
      /* if the character is not a pipe  */
      if (raw[i] != '|'){
        if(p == 0 && raw[i] == ' ' ){
          p--;
        }
        else {
          /* add the character to the current string  */
          curr += raw[i];
        }
        p++;

        if(raw[i] == '>'){
          hasOut += 1;
          if (hasOut > 1){
            return NULL;
          }
          previousOut = 1;
        }
        if(raw[i] == '<'){
          hasIn += 1;
          if (hasIn > 1){
            return NULL;
          }
          /* if file redirection is in wrong place */
          if (pipecount != 0) {
            return NULL;
            valid =false;
          }
        }
      }
      /* otherwise the character is a pipe character */
      if(raw[i] == '|'){
          if (raw[i-1] != ' '){
            /* if the previous character is not a string then it is invalid
input */
            valid = false;
            return NULL;
          }
          if (raw[i+1] != ' '){
            valid = false;
            return NULL;
          }
          pipecount += 1;
          /* if there are two many file redirections  */
          if (hasOut > 1 || hasIn > 1 || (hasOut > 0 &&  hasIn > 0)){
            valid = false;
            return NULL;
          }
          /* if file redirection from output is incorrect it is invalid */
          if (previousOut == 1){
            valid = false;
            return NULL;
          }
        /* null terminate the string */

        curr += '\0';
        /* for the current index of the 2d array allocate an accurate
amount of memory and copy from the current string into this index */
        command[count] = (char *) malloc(curr.length() + 1 * sizeof(char));
        for ( int i = 0; i < curr.length();i++){

          command[count][i] = curr[i];
        }

        /* recent most of the flags, the curr string, and add 1 to the
current count */
        p = 0;
        count += 1;
        curr = "";
        valid = true;
        hasOut = 0;
        hasIn = 0;
        starting = 0;
        previousOut = 0;
      }
```

```
    }

      /* if there is no pipes or it is last command */
       if (curr.length() < 1){
       return NULL;
       }

      command[count] = (char *) malloc((curr.length() + 1) *
sizeof(char));

      int m = 0;
      /* malloc space in the 2d char array for this last command */
       for ( m = 0; m < curr.length();m++){
         command[count][m] = curr[m];

       }
       command[count][m] = '\0';
    /* return the 2d char array */
    return command;

}


/* main method -- can take in a input file an argument if one does not
want to use standard in */
int main() {



  /* set up a buffer and get the name of the current working directory */
  char  buff[300];
  if (getcwd(buff,250) == NULL){

  }

  /* start of do while loop  */
  commands:
  do  {
  /* set up the current line and clear it each time the loop is called  */
  char currentLine[110];
  memset(&currentLine[0],0,110);
  string fullLine = "";

         /*read from standard in  */
    getline(cin, fullLine);

    if (cin.eof() == true){
      break;
    }
    /* if the line reads "exit" then escape the shell  */
    if (fullLine == "exit") {
      pipecount = 0;
      break;
      }
```

```
   /* if the length of the line is greater than 100 characters the line is
too long and is considered invalid input  */
  if (fullLine.length() > 100 ){
    cout << "invalid input" << endl;
    goto commands;
  }


  int length = fullLine.length();
  int y = 0;
  for (y = 0; y < fullLine.length(); y++){
    /* get the length of the line and then null terminate it*/
    currentLine[y] = fullLine[y];
  }
  currentLine[y] = '\0';

  /* call and return the parseLine helper function  */
  char ** tokens = parseLine(currentLine);
  if (tokens == NULL){
    cout << "invalid input" << endl;
    goto commands;
  }

  /* set the initial p_id index */
  int p_id = 0;
  int num_token_groups = pipecount + 1; /* number of distinct tokens
groups  */
  int limit = pipecount + 1;
  int pipes[2]; /* set up the pipes array */

   int pid_status[pipecount + 1]; /* set up the pid status array */
   int in = 0;
   /* for each distrinct token group in the command  */
  for (int current = 0; current < limit; current++){

    /* initialized the inputfile and outputfile strings */
    string inputFile = "";
    string outputFile = "" ;
    string tokenized[101];
    /* get and convert the current token group from the 2d array */
    char * command = tokens[current];
    std::string strcommand(command);

    /* call the second helper function to further sanitize and separate
input into token groups */
    int a = itemizer(strcommand, tokenized);
    if(a == -1){
      /* if invalid then go to next input line and output invalid input */
      cout << "invalid input" << endl;
      goto commands;
    }

    int size = a+1;
    int realSize = 0;
    /* pipe the pipes array */
    pipe(pipes);

    /* for each character in the current token group  */
    for (int j = 0; j < size; j++){
         /* check to make sure a token is one of the valid perscribed
tokens allowed */
          if(tokenized[j] != ">" && tokenized[j] != "<"){
          bool contains_not_approved =
(tokenized[j].find_first_not_of("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmno
pqrstuvwxyz01234566789-./_") == string::npos);
          /* if it is not, exit */
          if(contains_not_approved == false){
            cout << "invalid input" << endl;
            goto commands;
          }

          }

          if (j == 0){
            /* get the first character and if it is not "/" that it must
be
             a relative command, and the cwd needs to be prepended to it*/
            string command = tokenized[j];
            if (command[0] != '/'){
              int l = 0;
              string cwd = "";
              while(l < 250 && buff[l] != '\0'){
                  cwd = cwd + buff[l];

                  l = l + 1;
                  }
                  command = cwd + "/" + command;
                  tokenized[j] = command;
                 }
          }
          /* if redirection to std in, make sure it is valid and file name
is correct */
          if (tokenized[j] == "<") {
             if (current != 0) {
              cout << "invalid input" << endl;
              goto commands;
              }

             if (j + 1 < size) {
              if (realSize == 0) {
              realSize = j;
              }
              inputFile = tokenized[j + 1];
                if (inputFile[0] != '/'){
                int l = 0;
                string cwd = "";
                while(l < 250 && buff[l] != '\0'){
```

```
                cwd = cwd + buff[l];

                l = l + 1;
                }
                inputFile = cwd + "/" + inputFile;
               }
                bool inputfile_contains_not_approved =
(inputFile.find_first_not_of("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqr
stuvwxyz01234566789-./_") == string::npos);

          if(inputfile_contains_not_approved == false){
            cout << "invalid input" << endl;
            goto commands;
          }
               j++;

             }

          }
          /* if it is output redirection, make sure it is valid and file
name is correct and valid as well  */
        else if (tokenized[j] == ">"){
          if (current !=  limit - 1) {
             cout << "invalid input" << endl;
             goto commands;
             }

          if (j + 1 < size) {
            if(realSize == 0){
            realSize = j;
            }
             outputFile = tokenized[j+1];
             if (outputFile[0] != '/'){
             int l = 0;
             string cwd = "";
             while(l < 250 && buff[l] != '\0'){
              cwd = cwd + buff[l];

               l = l + 1;
             }
             bool outputfile_contains_not_approved =
(outputFile.find_first_not_of("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopq
rstuvwxyz01234566789-./_") == string::npos);

          if(outputfile_contains_not_approved == false){
            cout << "invalid input" << endl;
            goto commands;
          }
             outputFile = cwd + "/" + outputFile;

             }

             j++;
             }
```

```
}
       else {}
         }

         /* if there was redirection then I have to modify the number of
tokens in this group */
      if (realSize != 0){
       size = realSize;
      }

    /* fork */
    pid = fork();
      /* if child process */
      if (pid == 0) {

        /* set up the args array for the execvp command for later */
       char ** args = (char**)malloc(sizeof(char*) *50);
       for (int f = 0; f < size + 1; f++){
         args[f] = (char *) malloc(100 * sizeof(char));
       }

       char * argv[size + 1];
       int element = 0;
       char * ready = "";

       for( element = 0; element < size; element++){

           string ex = tokenized[element];
           if (ex == ">" || ex == "<" ){

             break;
           }
           int g = 0;
           for (g = 0; g < ex.length(); g++){
             args[element][g] = ex[g];
           }

           args[element][g] = '\0';
           argv[element] = args[element];
       }
       argv[element] =  '\0';

       /* if it is the first token in the group */
       if (current == 0){
         /* if there is redirection */
        if (inputFile.length() > 0) {

              int iF = open(inputFile.c_str(), O_RDONLY);
              if (iF == -1) {
               // cerr << strerror(errno) << endl;
                }
                /* make sure to take input from this file  */
              dup2(iF, STDIN_FILENO);
        }

        }
        /* if there are pipes we need to handling piping */
        if (pipecount > 0){

        /* if not the first command in the series  */
        if (current > 0  ){
          /* get input from the previous pipe */
          dup2(in, STDIN_FILENO);

          close(pipes[0]);
        }
        /* if not the last command in the series  */
        if ( current < num_token_groups - 1){
          /* write the output the write end of the pipe */
          dup2(pipes[1], STDOUT_FILENO);
          close(pipes[1]);

        }

        }
        /* if it is the last input in the series */
        if (current == num_token_groups - 1){
          /* if there is an output file */
          if (outputFile.length() > 0){
            int we_oF = open(outputFile.c_str(), O_WRONLY|O_CREAT);
            if (we_oF == -1) {
               //  cerr << strerror(errno) << endl;
                 }
                 /* output into that file  */
            dup2(we_oF,STDOUT_FILENO);
          }



        }

       int exVal = -1;

       int h = 0;
       /* exec the command with the given arguments that were formatted
above */
       h = execvp(argv[0], argv);
       /* exit if there is an error */
       exit(h);
       }
     else{
       /* parent process */

       /* close the write end of pipe and get the read end of pipe to
connect to next iteration through loop if needed  */
       close(pipes[1]);
       in = pipes[0];
```

```
      /* call wait id to wait for the child process to finish */
      waitpid(pid, &pid_status[p_id],0);
      p_id++;
    }
  }
     /* print out each of the valid commands exit codes */
    for(int d = 0; d <= pipecount; d++){
      cerr << pid_status[d] << endl;
    }

    free(tokens); /* free up the 2d char array */
  } while(1);
  /* end of method main */
  return 0;
}
```